修士学位論文

題目

プログラム動作理解支援を目的とした
オブジェクトの振舞いの同値分割手法

指導教員
井上 克郎 教授

報告者
宗像 聡

平成 22 年 2 月 8 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 21 年度 修士学位論文

プログラム動作理解支援を目的とした
オブジェクトの振舞いの同値分割手法

宗像 聡

内容梗概

クラスは，オブジェクト指向プログラムにおける基本的なソフトウェア部品である．オブジェクト指向プログラムは役割の異なる複数のクラスから構成されており，適切にメンテナンスをするには，クラスを理解する，つまりクラスの正しい使い方やそのオブジェクトの振舞いについて，十分に理解することが必要である．クラスを理解するには，そのクラスのオブジェクトの実際の振舞いを実行履歴から抽出し，可視化する手法が有効である．しかし，理解したいクラスのオブジェクトが多数生成される場合，オブジェクトそれぞれの振舞いを逐一確認することは，労力が大きく現実的ではない．

そこで本研究では，注目するクラスのオブジェクト群を，その振舞いの同値性に基づいて，同値分割する手法を提案する．ユーザが注目するクラスについて，そのすべてのオブジェクトを振舞いの同値性に基づいて同値分割すると，互いに振舞いが同値であるオブジェクトからなる同値類が複数得られる．各同値類から 1 つずつオブジェクトを選び，その振舞いを図として可視化することで，そのクラスの特徴的な振舞いのみをユーザに提示する．提案手法を GUI ベースのツールとして実装し．幾つかのオープンソースソフトウェアに対して適用することで，クラス理解に対する有効性を検証した．

主な用語

プログラム理解 (Program Comprehension)
オブジェクト指向プログラミング (Object-Oriented Programming)
動的解析 (Dynamic Analysis)
可視化 (Visualization)
部分理解 (Partial Comprehension)

目 次

1 はじめに

クラスは，オブジェクト指向プログラムにおける基本的なソフトウェア部品である．オブジェクト指向プログラムは役割の異なる複数のクラスから構成されており，適切にメンテナンスするには，クラスを理解する，つまりクラスの正しい使い方やそのオブジェクトの振舞いについて，十分に理解することが必要である [38]．理解することがプログラム全体の理解に不可欠であるようなクラス (キークラス) として，ユーザインターフェースクラス，プログラム中の主要なデータを表すデータクラス，分散システムで相互運用を補助する技術 (CORBA, RMI, DCOM など) で使用されるミドルウェアクラスなどが挙げられている [28, 42].

クラスの理解は，ソースコードやドキュメントの読解から始まることが多い．しかし，それらの文書に，クラスの正しい使い方や．そのオブジェクトの振舞いについての仕様が逐一記述されているとは限らない [1, 24]．また，記述されていたとしても，更新を怠りソースコードの内容と一致していないことがある [18]．そこで，対象のプログラムを解析することで，クラスの設計情報の復元が試みられる．特に，プログラムを実行することができる場合には，実行時に収集した動作情報を解析，理解したいクラスの特定オブジェクトの振舞いを図として可視化することで，クラスの実際の使われ方や振舞いを直観的に理解することができると言われている．これまでにも，動作情報から単一オブジェクトの振舞いを可視化する手法は多数提案されていて [9, 13, 27, 31, 41]，実際に被験者を用いて対照実験を実施し，クラス理解に有効であることを確認した手法もある [26].

しかし，理解したいクラスのオブジェクトがプログラム実行時に多数生成される場合には，これらの手法は適用することが困難である．なぜなら，同一クラスのオブジェクトであっても互いの振舞いには違いがある場合があり [33, 34]，振舞いを可視化するオブジェクトの選び方によって，獲得できる知識が異なると考えられるためである．すべてのオブジェクトの振舞いを確認すれば，実行履歴に含まれている振舞いについては網羅できるが，オブジェクトが多数ある場合には労力が大きく現実的ではない．キークラスの 1 つにあげられているデータクラスでは，実行時にそのオブジェクトが多数生成されることが予想でき，特に適用が困難であると考えられる．

そこで，本研究では，単一オブジェクトの振舞いを可視化する手法を，プログラム実行時にオブジェクトが多数存在するクラスについても効果的に適用できるように，注目クラスのオブジェクト群を，それらの振舞いの同値性に基づいて，同値分割する手法を提案する．ユーザが注目するクラスについて，そのすべてのオブジェクトを同値分割すると，互いに振舞いが同値であるオブジェクトの同値類が複数得られる．各同値類から 1 つずつオブジェクトを選び，振舞いを図として可視化，最終的に，そのクラスの特徴的な振舞いのみがユーザ

に提示される．ユーザは各図を比較することで，クラスに共通の動作と，少数のオブジェクトに特徴的な動作とを区別して調査することができる．また，本研究では，4 つの観点が異なる同値関係を定義しており，(1) 各同値類の振舞いを比較し，(2) 興味ある同値類についてさらに分割するという手順を再帰的に繰り返すことで，ユーザは興味ある振舞いのオブジェクトを効率良く選ぶことができる．提案手法を GUI ベースのツールとして実装し，複数のオープンソースソフトウェアに対して適用した．適用結果から，提案手法がクラス理解に有効であることを確認した．

本論文の構成は次の通りである．まず，2 章の前半で背景として，メンテナンス作業におけるプログラム理解とクラス理解の重要性について述べ，2 章の後半で問題提起として，オブジェクトが多数生成されるクラスではその動作を確認することが困難であることを論じる．次に，3 章で提案手法の詳細を説明し，4 章で提案手法を実装したツールについて説明，5 章で有効性評価として，既存のソフトウェアシステムに提案手法を適用した結果を述べる．6 章で関連研究を述べ，最後に，7 章でまとめと今後の課題をあとがきとして記す．

2 背景

ソフトウェアシステム開発では，オブジェクト指向プログラミングがますます普及しつつある．特に，C++や Java など，クラスの概念を持つオブジェクト指向プログラミング言語では，継承・カプセル化・多態性を効果的に活用することでコーディング，テスト，コード再利用の効率を向上させることができるため，大規模ソフトウェアシステムやフレームワークの開発でよく使用されている [25].

オブジェクト指向プログラムでは，実行時に動的に生成されるオブジェクトが相互にメッセージを交換しあうことで，全体が動作し機能を実現している．そのため，プログラムを理解しようとするとき，ソースコードのような静的な記述から，実行時に生成されるオブジェクト群の振舞いをイメージしていかなければならない．しかし，静的な記述から，動的束縛などを伴う複雑なオブジェクト群の動作や関連性を理解することは，非常に困難である [19, 40]．そこで，オブジェクト指向プログラムの理解には，生成されるオブジェクト群の動的な振舞いを図として可視化することが有効であると主張されており [15, 17, 22]，これまでにも多くの可視化手法が提案されてきた [23, 29, 33, 35].

2.1 プログラム理解

プログラム理解 (Program Comprehension) の厳密な定義は無いが，Cornelissen らは論文 [8] で Biggerstaff らの主張 [5] を基にした，次の定義を用いている．

定義．プログラム理解 (Program Comprehension)

> A person understands a program when he or she is able to explain the program, its structure, its behavior, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program.

この定義では，例えば，”int z = x + y;”という 1 命令文については，”int z = x + y;”であると説明したときではなく，”2 つの数値の和を求める”と説明できたときに，よりこれを理解していると判断される．

本研究でも，上記の定義をもって，”プログラムを理解する”と表現する．

2.2 動的解析によるプログラム理解

プログラムの解析は，静的解析と動的解析に分類される．静的解析は，主にソースコードやドキュメントなどの静的な情報を対象とする解析を指し，動的解析は，主にプログラム実行時に収集した動作情報を対象とする解析を指す [3].

現実的なプログラムでは，静的解析だけではプログラム理解が困難な場面が多く存在し，適宜動的解析を用いることで，より効果的にプログラム理解を行うことができる．例えば，1 つの手続き内の処理が，多数の制御構文を用いた複雑な制御構造により実現されている場合，ソースコード読解によるプログラム理解が困難になる．このとき，ある実行シナリオで実際に使用されたステートメントのみを動的解析により抽出し，それらにまず注目することは，プログラム理解を行う際の有効な足掛かりとなる．

また，複雑な呼び出し関係の解析にも，動的解析は有効である．手続き間の呼び出し関係の解析は，プログラム理解で主要な活動の 1 つであり [21, 36]，Eclipse[11] などの統合開発環境では，手続き間の関連を把握しながらソースコードの閲覧ができるように，エディタと同期する形で呼び出し関係を解析するツールが提供されている．しかし，ソースコードの静的解析では実行時に起こりうる呼び出し関係すべてを考慮するため，多くの手続きから呼び出されている手続きでは，多数の呼び出し関係が提示されてしまう．この場合に，この手続きの呼び出し関係を調べるには，提示された多数の関連する手続きについてソースコードを横断的に閲覧する必要があり，労力の増大が生じる．動的解析を用いて，ある実行シナリオで実際に実現した呼び出し関係のみを抽出し可視化することは，プログラム理解の有効な足掛かりとなる．

動的解析を静的解析と比べたときの利点をまとめると，次の通りである [8].

- **正確性．** どの手続きがどんな順序で呼び出されたか，どの変数がどこで使用されたかなどの，実際のシステムの振舞いに関する事柄について正確に解析できる．静的解析では，エイリアスの解決など，実行時の動作を正確に把握することは，現在のところ困難である．
- **ゴール指向．**システムの興味ある機能のみを動作させる実行シナリオを用いることで，その機能に関係する動作に注目して解析できる．

また，次の欠点があるとされている．

- **不完全性．** 動的解析では必ず，実際に起こりうる実行系列の内の一部しか解析できない．
- **シナリオ決定の難しさ．** システムの興味ある機能のみを動作させるような実行シナリオを作成することが難しい．
- **スケーラビリティ．** 解析環境やユーザが，処理または認識できないほど大量の動作情報を生成する場合がある．

- **観測問題．** リアルタイムシステムなどでは，動的解析のための観測自体が，システムの動作に影響を与える可能性がある．

本研究は，特にスケーラビリティに注目するものである．多数のオブジェクトが生成されるクラスでは，各オブジェクトの振舞いを逐一確認することは労力が大きく，現実的ではない．それぞれの振舞いの同値性に基づいてオブジェクト群を同値分割することで，振舞いの理解を支援する．

2.3 クラス理解

クラスは，オブジェクト指向プログラムにおける基本的なソフトウェア部品である．オブジェクト指向プログラムは役割の異なる複数のクラスから構成されており，適切にメンテナンスをするには，クラスの正しい使い方やそのオブジェクトの振舞いについて十分に理解することが重要である [38]．理解することがプログラム全体の理解に不可欠であるようなクラス (キークラス) として，ユーザインターフェースクラス，プログラム中の主要なデータを表すデータクラス，分散システムで相互運用を補助する技術 (CORBA, RMI, DCOM など) で使用されるミドルウェアクラスなどが挙げられている [28, 42]．また，Zaidman らは，Web マイニング手法を応用して，理解することがプログラム全体の理解に効果的であるクラスを推測する手法を提案している [42].

多くの場合，クラスの理解にはまず，ソースコードやドキュメントの読解が試みられる．しかし，それらにクラスの正しい使い方や．そのオブジェクトの振舞いの仕様が逐一記述されているとは限らず [1, 24]．また，記述されていたとしても，更新を怠りソースコードの内容と一致していないことがある [18]．そこで，プログラムを解析することで，クラスの設計情報を復元するということが行われる．特に，対象のプログラムを実行することができる場合には，実行履歴を解析，理解したいクラスの特定オブジェクトの振舞いを図として可視化することで，実際の使われ方や振舞いを直観的に理解することができるとされている．これまでにも，実行履歴から単一オブジェクトの振舞いを可視化する手法は複数提案されてきた [9, 13, 31, 41].

2.3.1 オブジェクトトレース

あるオブジェクトについて，それと関係する動作情報のみを実行履歴から抽出したものをオブジェクトトレースと呼ぶ．オブジェクトトレースは，単一オブジェクトの振舞いを可視化する手法でよく用いられる [27]．オブジェクトトレースとして，どのような動作情報を抽出するかは，解析の目的に応じて異なってくる．Gschwind らは，オブジェクトのインターフェースや使い方について解析したい場合は，他のオブジェクトからのメソッド呼び出しを，

オブジェクトの実装について解析したい場合は，他のオブジェクトに対するメソッド呼び出しを，プログラムのどこで使われたのかを解析したい場合は，どのクラスから使用されたのかを，動作情報として抽出すればよいと主張している [13]．Gschwind らは，ケーススタディとして，あるオブジェクトについてそのオブジェクトトレースを UML シーケンス図で可視化し，単一オブジェクトの振舞いの理解に有効であることを示した．

2.3.2 UML シーケンス図

UML シーケンス図とは，UML で定義されている相互作用図の 1 つで，オブジェクト間のメソッド呼び出しやオブジェクトの生成などのメッセージ通信を，時系列に沿って示した図である [20]．UML シーケンス図の例を図 1 に示す．横軸はオブジェクトを表しており，図の上部に 1 つずつオブジェクトが並ぶ．縦軸は時間軸を表しており，下方に行くほど時間が経過している．各オブジェクトからは縦方向に点線が引かれており，これがオブジェクトの生存区間を表している．オブジェクト間のメッセージ通信は，時系列順に，送信元オブジェクトから，送信先オブジェクトに対して矢印を引くことで表現する．メッセージがメソッド呼び出しだった場合は，そのメソッドの実行区間を，送信先 (=呼び出し先) オブジェクトの生存ライン上に，縦長の長方形で表現する．メソッド実行により戻り値の授受があった場合には，送信元 (=呼び出し元) オブジェクトまで矢印を引き表現する．メッセージがオブジェクトの生成だった場合は，生成されるオブジェクトをメッセージと同じ高さに書く．このようにして，オブジェクト間のメッセージ通信を時系列に沿って直観的に表現する．

UML シーケンス図を用いることで，時系列に沿ったオブジェクト群の振舞いを直観的に理解することができる．また，1 枚の UML シーケンス図には実際に実現したメソッド呼び出し系列が記録されており，2.2 節で説明したように，複雑なプログラムを理解する際の足掛かりとしても有効である．

2.3.3 DOPG(Dynamic Object Process Graph)

DOPG(Dynamic Object Process Graph) は，オブジェクトトレースから生成する動的な OPG(Object Process Graph) である [27]．実行履歴から生成するプログラム全体の制御フローグラフから，注目する単一オブジェクトについてスライスした，インタープロシージャルな制御フローグラフであるとも表現できる．DOPG では，ソースコード上のループ構造を保持したまま実行時情報を提示することができる．そのため，ループ中の小さな振舞いの差に影響されることなく，UML シーケンス図などと比べて，より小さな図でオブジェクトの振舞いを表現できる．DOPG の各頂点はソースコード位置と 1 対 1 に対応しており，ソースコードとのトレーサビリティが高い．実際に，DOPG を提案した Quante らは，DOPG の各

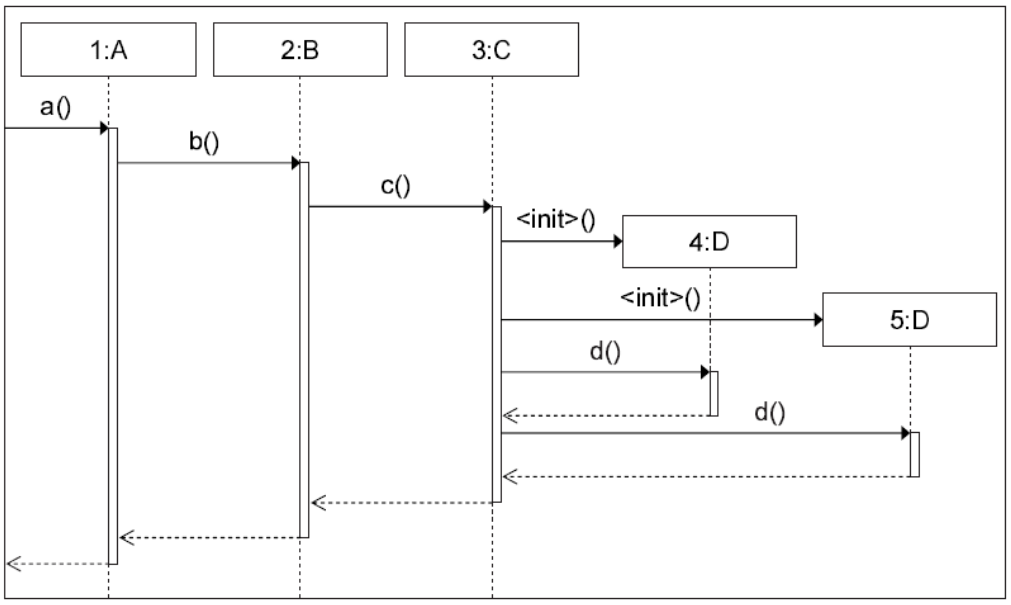


図 1: UML シーケンス図

頂点からソースコードエディターにジャンプする機能を，開発した DOPG 可視化ツールに組み込んでいる．複数のソースコードから構成されている一連の処理を，DOPG では 1 枚の図で表現できるため，ソースコードを横断的に閲覧する際に有効な足掛かりとなる．

ただし，DOPG ではメソッド呼び出しの発生順序や発生回数，呼び出し元オブジェクトのクラスは正確に表現できない．もし，これらの情報が必要なときは，UML シーケンス図などの他の動的表現と併用する必要があると考えられる．

DOPG は，8 種類の頂点と，3 種類の辺から構成される (ただし，参考文献 [28] では，さらにスレッドの開始・終了を表す頂点などが，新たに追加されている).

- **Read 頂点:** 自身のフィールドの値を参照する式を表す頂点
- **Write 頂点:** 自身のフィールドの値を定義する式を表す頂点
- **Call 頂点:** メソッド呼び出しを行う式を表す頂点
- **Entry 頂点:** メソッドの実行開始を表す頂点
- **Return 頂点:** メソッドの実行終了 (Return 文，例外スロー) を表す頂点
- **Start 頂点:** DOPG の始点を表す頂点

- **Final 頂点:** DOPG の終点を表す頂点
- **Condition 頂点:** 条件分岐文を表す頂点
- **Conditional 辺:** 条件分岐により生じた制御フローを表す辺
- **Unconditional 辺:** 条件分岐を挟まない制御フローを表す辺
- **Invocation 辺:** メソッド呼び出しにより生じた制御フローを表す辺

オブジェクトトレースとして，自身へのメソッド呼び出しとフィールドアクセス，及び制御フローに関わる他のメソッド呼び出しと条件分岐を実行履歴から抽出，各イベントを対応する頂点に変換，頂点間を対応する辺で接続することで，DOPG が生成される (図 2).

参考文献 [26] では被験者を用いて対照実験を実施しており，オープンソースソフトウェア GanttProject で使用されているクラス GanttTask については，Eclipse だけを使ってプログラム理解を試みたチームより，Eclipse と DOPG 可視化ツールを併用したチームの方がより効果的にプログラムを理解できていたことが示されている．ただし，オープンソースソフトウェア ArgoUML で使用されているクラス ClassDiagramGraphModel では，両グループ間の理解度に有意な差がなかったことが示されており，これは DOPG が大規模になりすぎると読解が困難になるためだと説明している．

2.3.4 呼び出し関係図

呼び出し関係図とは，オブジェクトとその周辺クラスとの呼び出し関係を，有向グラフとして可視化した図である [17]．この有向グラフは頂点として，オブジェクトとクラスを用いる．ある頂点のオブジェクト，あるいは頂点のクラスに属するオブジェクト群の内 1 つ以上が，ある別の頂点に属するオブジェクトにメソッド呼び出しをしていた場合に，この頂点間に有向辺を作成する．このグラフをオブジェクトの振舞いの比較に用いることで，オブジェクトと相互作用するクラス群の差異を効果的に理解できる (図 3).

2.4 多数のオブジェクトが存在するクラスの理解

実行履歴解析により単一オブジェクトの振舞いを可視化するクラス理解支援手法は，注目クラスのオブジェクトがプログラム実行時に多数存在する場合，単純に適用することはできない．その根拠を，次に示す．

```
00  int main () {
01   int i = 0;
02   Stack *s1 = init ();
03   Stack *s2 = read ();
04   reverse (s2, s1);
05   do
06   { pop(s1);
07    i = i + 1; }
08   while (!empty (s1));
09  }
```

```
10  void reverse
11    (Stack *from, Stack *to)
12  {
13   while (!empty (from))
14    push (to, pop (from));
15  }
```

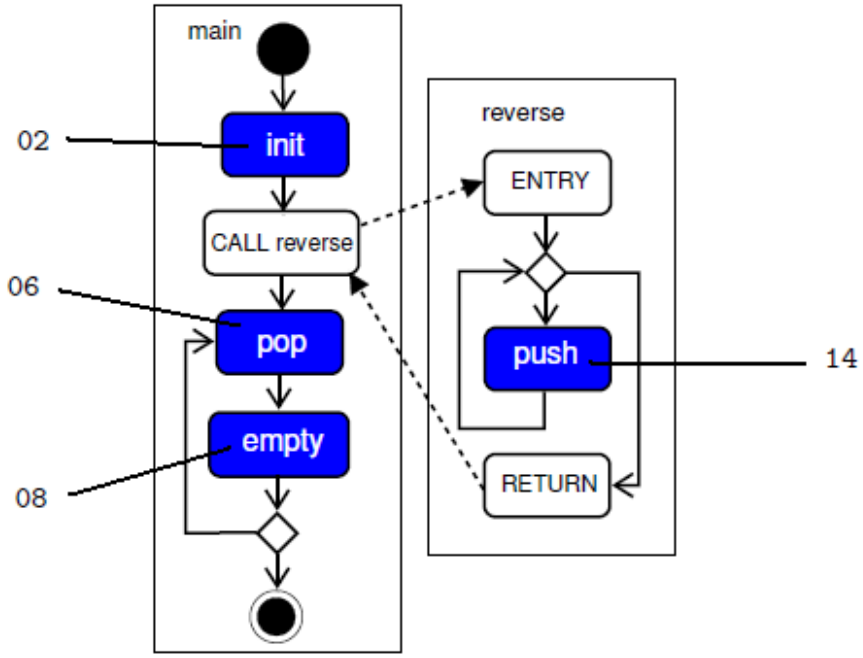


図 2: ソースコード例とスタック $s1$ について抽出した DOPG 図．全体の制御フローグラフから，スタック $s1$ に対する操作に関係する箇所だけを抽出する．

根拠 1．振舞いの確認に要する労力の増大

注目クラスのすべてのオブジェクトについて可視化すれば，その実行履歴に含まれている振舞いについてはすべて網羅できる．しかし，プログラム実行時に注目クラスのオブジェク

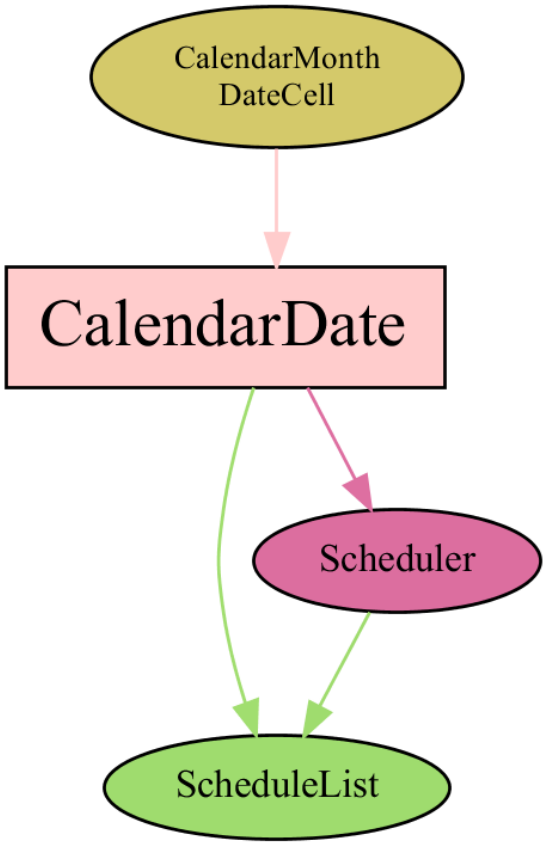


図 3: カレンダー表示プログラム Scheduler で使用されているクラス CalendarDate のあるオブジェクトについての呼び出し関係図．CalendarDate オブジェクトは，DateCell クラスのあるオブジェクトから，メソッド呼び出しされていることなどがわかる．

トが数百個以上生成されることは，キークラスにデータクラスがあげられていることからも十分に考えられ，この場合，すべてのオブジェクトの振舞いを逐一確認することは労力が大きく現実的ではない．同じクラスのオブジェクトが多数存在したとしても，その多くは互いに振舞いが類似していると期待されるため，そのクラスの特徴的な振舞いのみを可視化することができれば，より効果的に理解を行うことができると考えられる．

実際に，Java で実装された予定管理プログラム Scheduler で使用されているクラス CalendarDate について簡単な実行シナリオで動的解析をしたところ，プログラム実行時に生成された 731 個のオブジェクトの振舞いは，呼び出し関係図で表現すると 3 パターン，DOPG 図で表現すると 6 パターンしかなかった．表現が同じ (テキスト，デザインが等しい)2 つの図をそれぞれ参照する必要はなく，このケースにおいては，相互に表現の異なる図のみを参照する方が，より効果的にクラスを理解できると考えられる．

根拠 2．オブジェクトの選択の難しさ

同じクラスのオブジェクトであっても，その使われ方や振舞いはそれぞれに異なるため [34, 33]，オブジェクトが複数ある場合，可視化するオブジェクトの選び方によって獲得できる知識に違いがあると考えられる．例えば，2 つのオブジェクト o_1, o_2 があり，o_1 はメソッド m を他のクラスから呼び出されていて，o_2 は m を呼び出されていない場合，この m の使い方については o_2 の振舞いからは確認できない可能性が高い．そのため，単一オブジェクトの振舞いを可視化するクラス理解支援手法を効果的に適用するには，振舞いの違いを認識してオブジェクトを選ぶ方法を，ユーザに提供する必要があると考える．

実際に，Java で実装されたテトリスゲーム Jetris で使用されているクラス LFigure(オブジェクトは L 字型ブロック 1 つに対応) について簡単な実行シナリオで動的解析をしたところ，プログラム実行時に生成された 8 個のオブジェクトは，(1) テトリスゲームで使用するブロックのサンプルを表示するために生成されたブロック，(2) テトリスゲームプレイ中にユーザ操作により落下と右回転を行ったブロック，(3) 落下中に右回転を試みたが壁に衝突して右回転しなかったブロック，のいずれかに対応することがわかった．それぞれの DOPG は，含んでいる頂点 (=メソッド) がそれぞれに異なっており，このケースでは DOPG ごとに獲得できる知識に違いがあると言える．

そこで，本研究では，単一オブジェクトの振舞いを可視化する手法を，プログラム実行時にオブジェクトが多数存在するクラスについても効果的に適用できるように，注目クラスのオブジェクト群を，それらの振舞いの同値性に基づいて，同値分割する手法を提案する．ユーザが注目するクラスについて，そのすべてのオブジェクトを同値分割すると，互いに振舞いが同値であるオブジェクトの同値類が複数得られる．各同値類から 1 つずつオブジェクトを選び，振舞いを図として可視化，最終的に，そのクラスの特徴的な振舞いのみがユーザに提示される．ユーザは各図を比較することで，クラスに共通の動作と，少数のオブジェクトに特徴的な動作とを区別して調査することができる．また，本研究では，4 つの観点が異なる同値関係を定義しており，(1) 各同値類の振舞いを比較し，(2) 興味ある同値類についてさらに分割するという手順を再帰的に繰り返すことで，ユーザは興味ある振舞いのオブジェクトを効率良く選ぶことができる．

3 提案手法

本研究では，オブジェクトの振舞いに対して同値関係を定義し，オブジェクト群の同値類それぞれから 1 つずつオブジェクトを選び，それらの振舞いを図として可視化する手法を提案する．本研究では観点の異なる 4 つの同値関係を定義しており，利用者は調査したい振舞いの特徴に応じて，同値分割に使用する同値関係を選ぶことができる．

3.1 実行履歴の取得

本手法は，オブジェクト指向プログラムの実行時情報を用いる動的解析手法の 1 つである．利用者は，解析したいプログラムを実行し，動作中に行われた各メソッド呼び出しの情報を，実行履歴として取得する必要がある．各メソッド呼び出しの情報には，次の情報が含まれていなければならない．

- 呼び出し元オブジェクトの識別子，クラス
- 呼び出し先オブジェクトの識別子，クラス
- メソッドシグネチャ(戻り値型，定義クラス，メソッド名，各引数型)

同値分割に使用する情報は上記のみであるが，提案手法と組み合わせて使用する可視化手法のために，次の情報も必要となる．

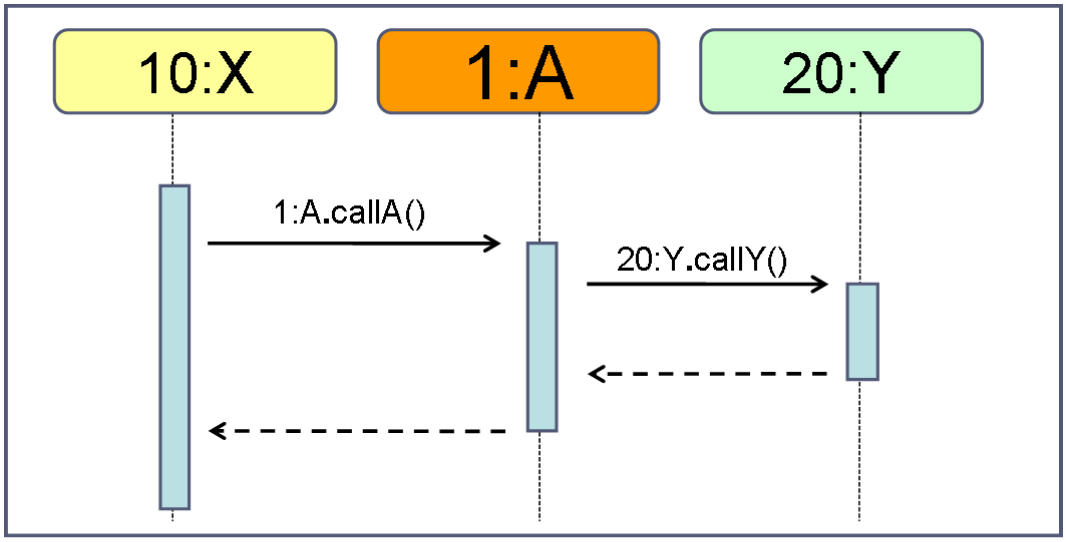
- 実行されたスレッドの識別子
- メソッド呼び出しの実行順序 (タイムスタンプ)

3.2 オブジェクトの振舞いの同値分割

オブジェクトの振舞いの同値関係は，オブジェクトの動作コンテキストに基づいて定義する．どの動作コンテキストを基準にするかにより，観点の異なる同値関係となる．オブジェクトの振舞いの同値関係として，動作コンテキスト集合 $Use(o)$ に基づく $E_{use}(o_s, o_k)$ ，$Used(o)$ に基づく $E_{used}(o_s, o_k)$ ，$Methods(o)$ に基づく $E_{methods}(o_s, o_k)$ ，$Called(o)$ に基づく $E_{called}(o_s, o_k)$ を定義する．

3.2.1 動作コンテキスト集合

あるオブジェクトの動作コンテキスト集合とは，プログラム実行時にそのオブジェクトが相互作用したメソッドやクラスからなる集合である．動作コンテキスト集合として，次の 4



クラスAのオブジェクト1:Aについての各動作コンテキスト集合

(a) Use(1:A) = { X }
(b) Used(1:A) = { Y }
(c) Methods(1:A) = { callA() }
(d) Called(1:A) = { callY() }

図 4: 動作コンテキスト

つを定義する．なお，コンストラクタ呼び出しは，メソッド名が定義クラス名であるメソッド呼び出しとして扱う．

- $Use(o)$: オブジェクト o に対して，メソッド呼び出しを行ったオブジェクトのクラスの集合 (図 4a).
- $Used(o)$: オブジェクト o に，メソッド呼び出しされたオブジェクトのクラスの集合 (図 4b).
- $Methods(o)$: 動作した，オブジェクト o のメソッドの集合 (図 4c).
- $Called(o)$: オブジェクト o が呼び出したメソッドの集合 (図 4d).

3.2.2 $E_{use}(o_s, o_k)$

2 つのオブジェクト o_s, o_k について，それぞれの動作コンテキスト集合 $Use(o_s), Use(o_k)$ が完全一致するとき，かつそのときに限り，同値関係 $E_{use}(o_s, o_k)$ が成り立つ．

$$E_{use}(o_s, o_k) \leftrightarrow Use(o_s) = Use(o_k)$$

クラスはプログラム中でそれぞれに異なる役割を担っていて，複数の役割の異なるクラスが協調動作することで，システムの機能は実現される [1, 30, 32]．そのため，呼び出し元のクラスの集合が異なる 2 つのオブジェクトでは，それぞれ異なる機能から使われていた可能性があり，同値関係 $E_{use}(o_s, o_k)$ は，このような参加した機能の違いからくる振舞いの差を識別する．

3.2.3 $E_{used}(o_s, o_k)$

2 つのオブジェクト o_s, o_k について，それぞれの動作コンテキスト集合 $Used(o_s), Used(o_k)$ が完全一致するとき，かつそのときに限り，同値関係 $E_{used}(o_s, o_k)$ が成り立つ．

$$E_{used}(o_s, o_k) \leftrightarrow Used(o_s) = Used(o_k)$$

呼び出し先のクラスの集合が異なる 2 つのオブジェクトでは，それぞれ異なる機能を実現した可能性があり，同値関係 $E_{used}(o_s, o_k)$ はこのような実現した機能の違いからくる振舞いの差を識別する．

3.2.4 $E_{methods}(o_s, o_k)$

2つのオブジェクト o_s, o_k について，それぞれの動作コンテキスト集合 $Methods(o_s), Methods(o_k)$ が完全一致するとき，かつそのときに限り，同値関係 $E_{methods}(o_s, o_k)$ が成り立つ．

$$E_{methods}(o_s, o_k) \leftrightarrow Methods(o_s) = Methods(o_k)$$

動作したメソッドの集合が異なる 2 つのオブジェクトは，他のオブジェクトからそれぞれ異なる使い方をされた可能性があり，同値関係 $E_{methods}(o_s, o_k)$ は，このような使われ方の違いからくる振舞いの差を識別する．

3.2.5 $E_{called}(o_s, o_k)$

2つのオブジェクト o_s, o_k について，それぞれの動作コンテキスト集合 $Called(o_s), Called(o_k)$ が完全一致するとき，かつそのときに限り，同値関係 $E_{called}(o_s, o_k)$ が成り立つ．

$$E_{called}(o_s, o_k) \leftrightarrow Called(o_s) = Called(o_k)$$

呼び出したメソッドが異なる 2 つのオブジェクトは，それぞれ異なる実装を処理に利用した可能性があり，同値関係 $E_{called}(o_s, o_k)$ は，このような利用した実装の違いからくる振舞いの差を識別する．

3.2.6 再帰的な同値分割

ある同値分割から得た同値類について，異なる同値分割を再帰的に適用することができる．例えば，同値関係 $E_{use}(o_s, o_k)$ に基づくオブジェクト群 O の同値分割 $G_{use}(O) = \{G_1, \cdots, G_l\}$ から得た同値類 $G_i \in G_{use}(O)$ について，さらに同値関係 $E_{methods}$ により同値分割した場合，同値分割 $G_{use,methods}(G_i) = \{G_{i1}, \cdots, G_{im}\}$ が得られる．この分割で得た同値類は，含まれるすべてのオブジェクトで同値関係 $E_{use}, E_{methods}$ が成り立つ．

3.2.7 時間計算量

同値分割では，ある要素が属する同値類を，各同値類の高々1 つの要素と同値関係を判定することで決定できる．そのため，分割対象のオブジェクト群の要素数を n ，分割後の同値類数を g とすると，分割に要する時間計算量は $O(gn)$ である．オブジェクトが多数存在する場合は $g \ll n$ であることが期待されるため，実質的な時間計算量は $O(n)$ に近づく．

類似度を用いて振舞いのクラスタリングを行う手法では，時間計算量は厳密解法で指数時間，近似解法でも線形時間より大きく [34]，提案手法は時間計算量の観点で有利である．

3.3 振舞いの可視化

オブジェクト群の同値分割により得られる同値類から 1 つずつオブジェクトを選び出し，呼び出し関係図，DOPG 図，あるいは UML シーケンス図として可視化する．表 1 に示すように各可視化手法はそれぞれ観点や抽象度が異なり，識別できる振舞いの差もそれぞれに異なる．そのため，クラス理解の目的に合わせて，可視化手法と適用する同値関係を選ぶことになる．

オブジェクトの振舞いの可視化にあたっては，利用者が選択したオブジェクトについて，あるいは選択した同値類からランダムに 1 つ選んだオブジェクトについて，その振舞いを可視化する．同値類ごとに 1 つずつオブジェクトを選択し，同値類ごとの振舞いの違いを強調して可視化することが可能である．なお，UML シーケンス図として可視化する場合には，選択オブジェクトが呼び出し元であるメソッド呼び出し，呼び出し先であるメソッド呼び出し，及び，それらに推移的に到達可能なメソッド呼び出しからなるオブジェクトトレースを可視化する．

表 1: 可視化手法の比較

手法	Use	$Used$	$Methods$	$Called$	抽象度
呼び出し関係図	○[1]	○	×	×	高
DOPG 図	△	△	○	△	中
UML シーケンス図	○	○	○	○	低

○は違いを完全に識別できることを，△は条件的に識別できることを，×は全く識別できないことを表している．

4 実装

提案手法をツールとして実装した．ツールは 3 つのソフトウェアから構成されていて，Amida-Agent は実行履歴の取得と動的モデル構築，Amida-OGAN は実行履歴の解析と可視化，Amida-Viewer[43] はシーケンス図の可視化を担当している．この内，Amida-Viewer は我々のチームで以前に開発したものであり，今回新たに開発した部分は Amida-Agent と Amida-OGAN である．コーディングにはすべて Java 言語を用いた．ソースコードの総行数はコメントを含めて，Amida-Agent が約 29000 行，Amida-OGAN は約 15000 行であった．

4.1 アーキテクチャ

本ツールは，動的モデル構築部，GUI，分割部，ソースコード表示部，呼び出し関係図生成部，シーケンス図生成部，DOPG 生成部から構成されている．ツールのアーキテクチャを図 5 に示す．

4.2 Amida-Agent

Amida-Agent は Java プログラムの実行を監視し，実行履歴情報を収集するトレーサである．動作情報の収集にはバイトコード変換支援ライブラリ Javassist[44] を用いており，動作情報を出力するコードを解析対象のプログラムに埋め込むことで，プログラム実行時に実行履歴を生成している．バイトコード変換によるトレースは，プログラム自身がロギングを直接行うため，JVMTI[37] などを介するよりも高速に動作する．また，Java1.5 以上であれば基本的にコンパイラ・VM を選ばないといった利点がある．一方，クラスファイルのサイズ上限を超えてコードを埋め込むことはできないため，コード量が多いなどの要因でサイズが大きいクラスに対しては，情報を取得できない場合があるという欠点もある．

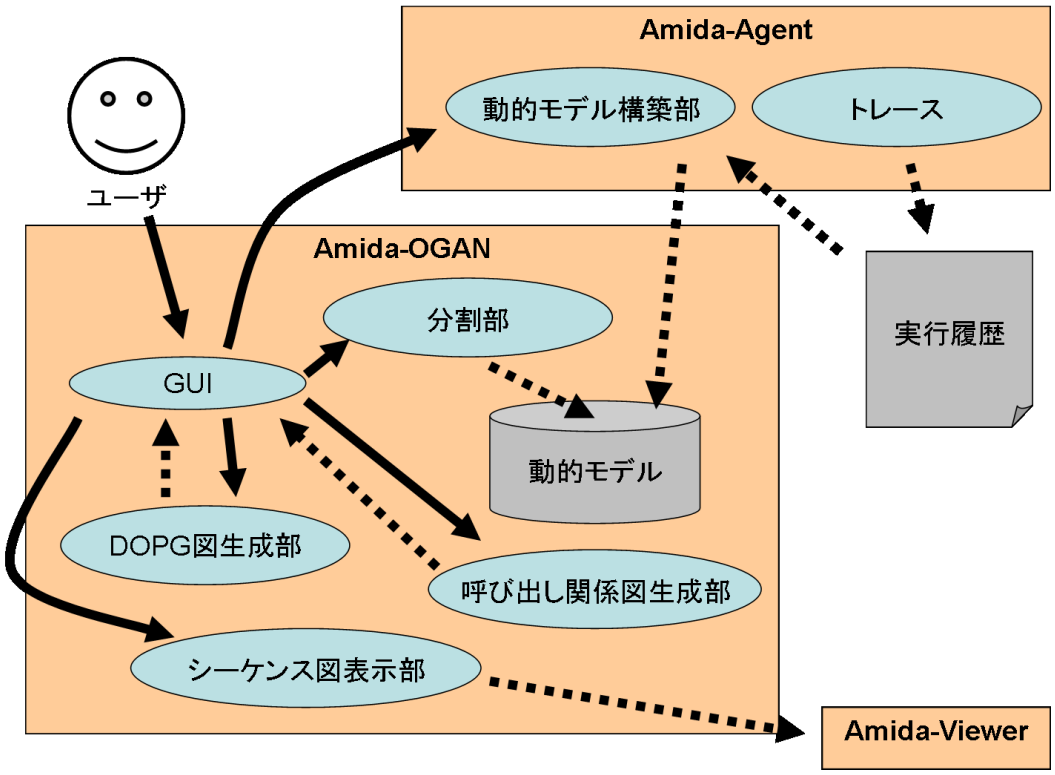


図 5: 本研究で実装したツールのアーキテクチャ．実線は制御の流れを，点線はデータの流れを表す．

収集可能な実行履歴

Amida-Agent では，次のイベントを実行履歴として収集することができる．

- クラスイニシャライザーの実行，復帰
- コンストラクタの呼び出し，復帰
- メソッドの呼び出し，復帰
- フィールドの参照，定義
- New 演算の実行
- ローカル変数の定義
- 配列要素の定義
- カバレッジ (到達行)

また，各イベントから次の詳細情報を収集することができる．

- 実行順序 (タイムスタンプ)
- 実行時刻 (リアルタイム)
- 実行スレッドの ID と名前
- コンストラクタ・メソッドの定義クラス，シグネチャ，修飾子
- コンストラクタ・メソッド呼び出しの各引数の値と型，戻り値と型
- フィールドの定義クラス，シグネチャ，修飾子
- フィールドアクセスの代入値，型
- ローカル変数の定義クラス，定義メソッド，シグネチャ
- ローカル変数アクセスの代入値，型
- 配列要素アクセスのインデックス，代入値，型
- 実行したステートメントの記述ファイル名，行番号
- 出現したオブジェクトの ID，型
- 発生した例外の型，メッセージ
- クラス java.lang.String のオブジェクトが表現する文字列

3.1 節で述べた情報よりも項目数が多いが，これは今後の研究に備えた実装となっているためである．

パッケージ名，クラス名，メソッド名，変数名などの条件に基づいて，収集するイベントを限定する設定を行うことができる．

実行履歴は動作情報のタイプ (イベント情報，型情報，シグネチャ情報など) ごとにフォーマットの異なる複数のファイルとして出力される．動的モデル構築部には，実行履歴に含まれる情報を，イベントオブジェクトの列として取り扱うためのクラス群が用意されている．

4.3 Amida-OGAN

Amida-OGAN は，Amida-Agent で取得した実行履歴を読み込み，対話的に実行履歴を解析することで，プログラム理解を支援するツールである．クラスに注目し，そのクラスのオブジェクトを選択した分割基準で分割，各オブジェクトグループの振舞いを可視化，各振舞いを比較・読解し，クラスについての理解を深める．

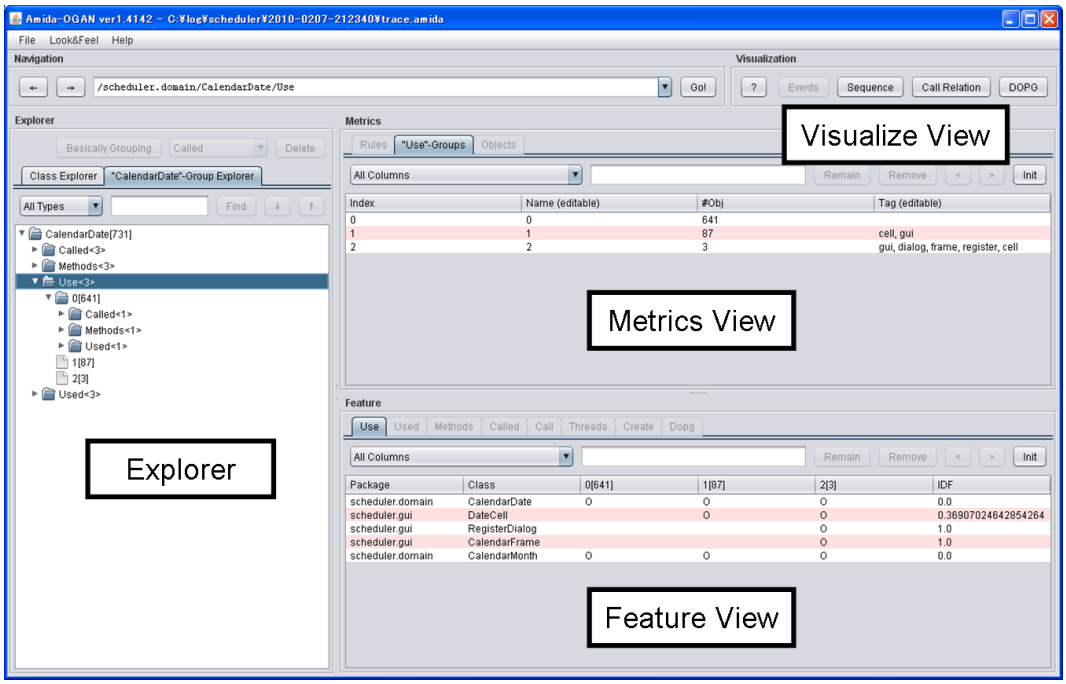


図 6: Amida-OGAN メイン画面

GUI

Amida-OGAN のメイン画面を図 6 に示す．GUI は，オブジェクトグループの分割や選択を行う Explorer，選択したグループの情報を表示する MetricsView，選択したグループの動作コンテキスト集合を表示する FeatureView，選択したグループの振舞いを可視化する VisualizeView，可視化した振舞いを表示する DiagramDialog の 5 つに分けられる．

Explorer(図 7)

Explorer では，クラスの選択 (図 7a)，オブジェクトグループの選択と分割 (図 7b) ができる．ツリービューによりグループ階層が表示され，分析したいグループを選択，上部のコンボボックスから分割基準を選択することで，選択グループが分割される．

Metrics View(図 8)

Metrics View では，Explorer でグループが選択されている場合は，そのグループに含まれている各オブジェクトの識別子，最初に出現した時のタイムスタンプ，最後に出現した時のタイムスタンプ，メソッドを呼び出された回数，メソッドを呼び出した回数を表示する (図 8a)．Explorer で分割基準が選択されている場合は，その分割で得た各グループの連番 (インデックス)，名前，オブジェクト数，タグを表示する (図 8b)．

Feature View(図 9)

Feature View では，Explorer でグループが選択されている場合は，そのグループ内のすべてのオブジェクトで共通している動作コンテキスト集合 (=このグループを得た分割基準) を表示する (図 9a)．Explorer で分割基準が選択されている場合は，その分割で得た各グループを列，動作コンテキスト集合の各要素を行とする直行表で，対応関係を表示する (図 9b).

Visualize View(図 10)

Visualize View には，呼び出し関係図の生成を指示するボタン，UML シーケンス図の生成を指示するボタン，DOPG 図の生成を指示するボタンがある．あるボタンがクリックされたとき，Explorer でグループが選択されている場合は，そのグループの振舞いをボタンに対応する図で可視化する．Explorer で分割基準が選択されている場合は，その分割で生成された各グループの振舞いを可視化する．

Diagram Dialog(図 14)

Diagram Dialog は，呼び出し関係図や DOPG 図を表示する，メイン画面とは独立したダイアログである．複数の図を並べて配置することができる．具体的には，呼び出し関係図生成部や DOPG 図生成部から渡された SVG ファイルを，オープンソースの SVG 描画ライブラリ Batik[2] を利用して可視化している．図はマウスで操作でき，左ドラッグで移動，ホイールスクロールで拡大縮小する．また，図中の各要素にマウスカーソルをのせると，ポップアップで詳細情報が表示される．DOPG 図ではポップアップ (図 11) 内のボタンをクリックすることで，その要素 (メソッド呼び出し) を中心とするシーケンス図やソースコード (図 13) を表示できる．

分割部

Explorer で選択されたグループを，指定された分割基準で分割する．

呼び出し関係図生成部

Explorer で指定されたグループ，またはグループ群の振舞いに対応する呼び出し関係図を 3.3 節で説明したように生成し，Diagram Dialog に渡す．具体的には，呼び出し関係図を表現する SVG ファイル [39] をグラフ可視化ツール GraphViz[12] で生成している．

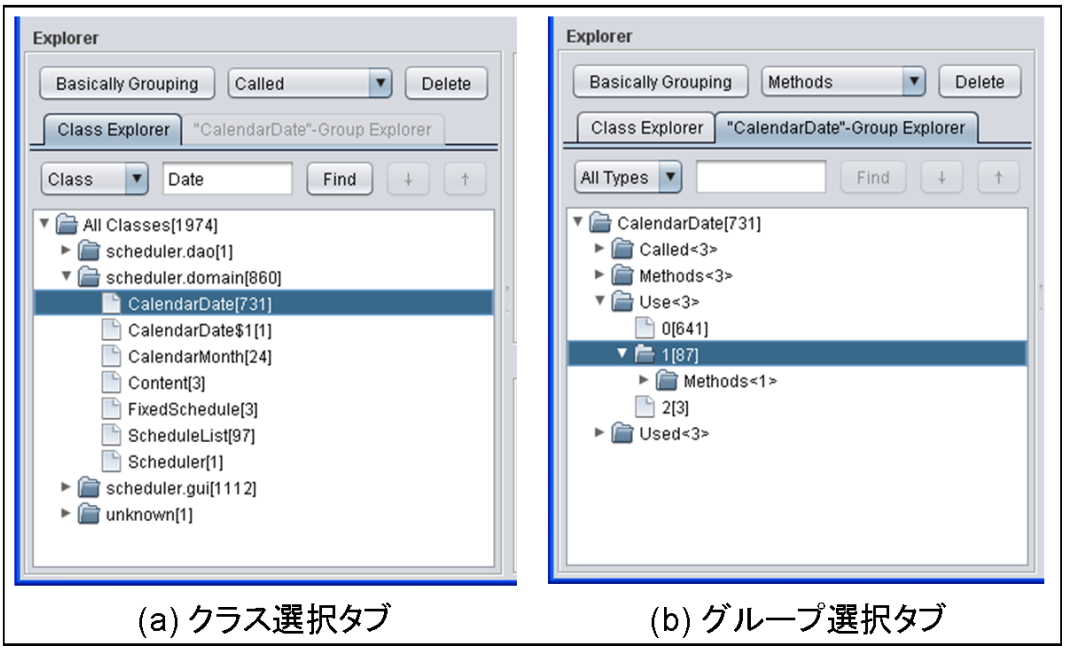


図 7: Explorer

図 8: Metrics View

シーケンス図生成部

Explorer で指定されたグループ，またはグループ群の振舞いに対応するオブジェクトトレースを 3.3 節で説明したように生成し，Amida-Viewer に渡す.

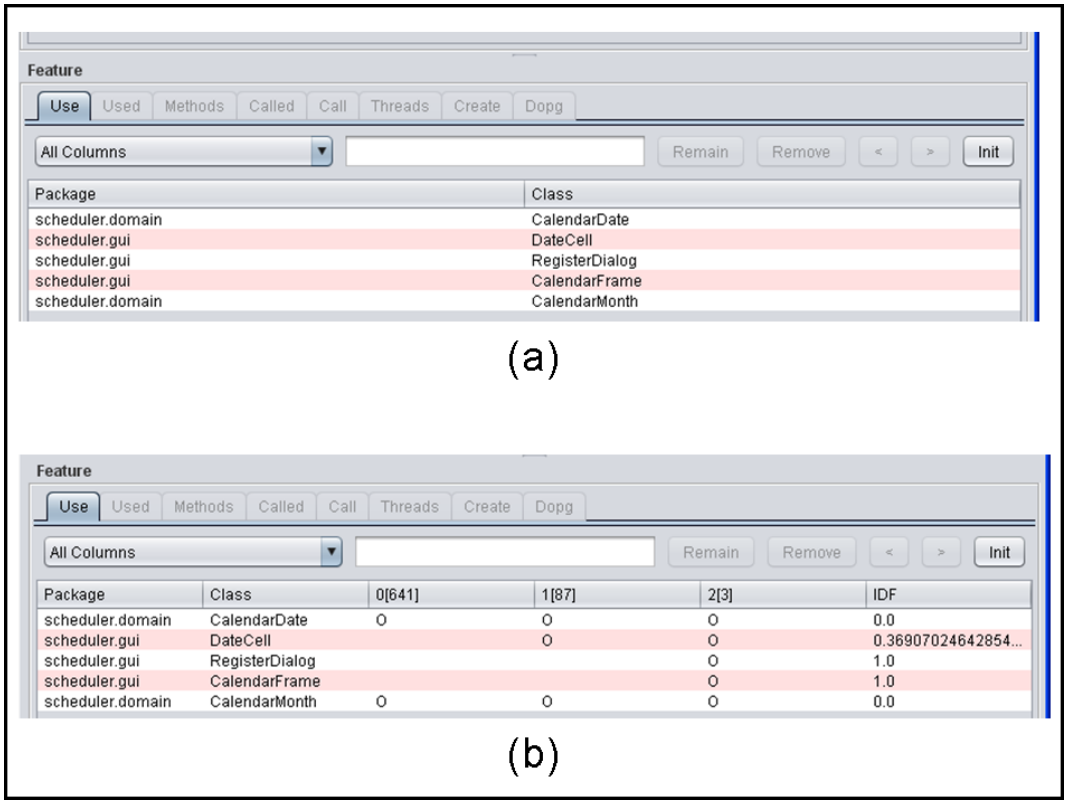


図 9: Feature View

図 10: Visualize View

Amida-Viewer

オブジェクトトレースから UML シーケンス図を生成し可視化する (図 12).

DOPG 図生成部

Explorer で指定されたグループ，またはグループ群の振舞いに対応する DOPG 図を 3.3 節で説明したように生成し，Diagram Dialog に渡す．具体的には，DOPG 図を表現する SVG

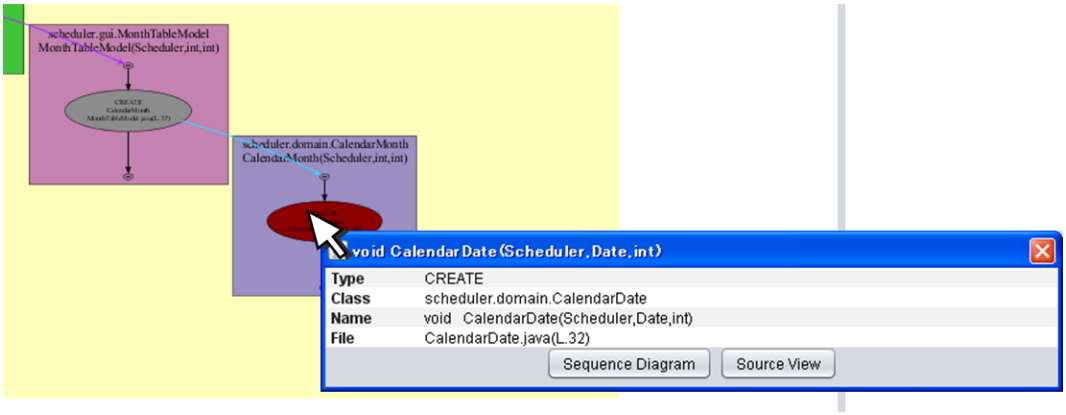


図 11: DOPG 図中の頂点にマウスを重ねると表示されるポップアップ

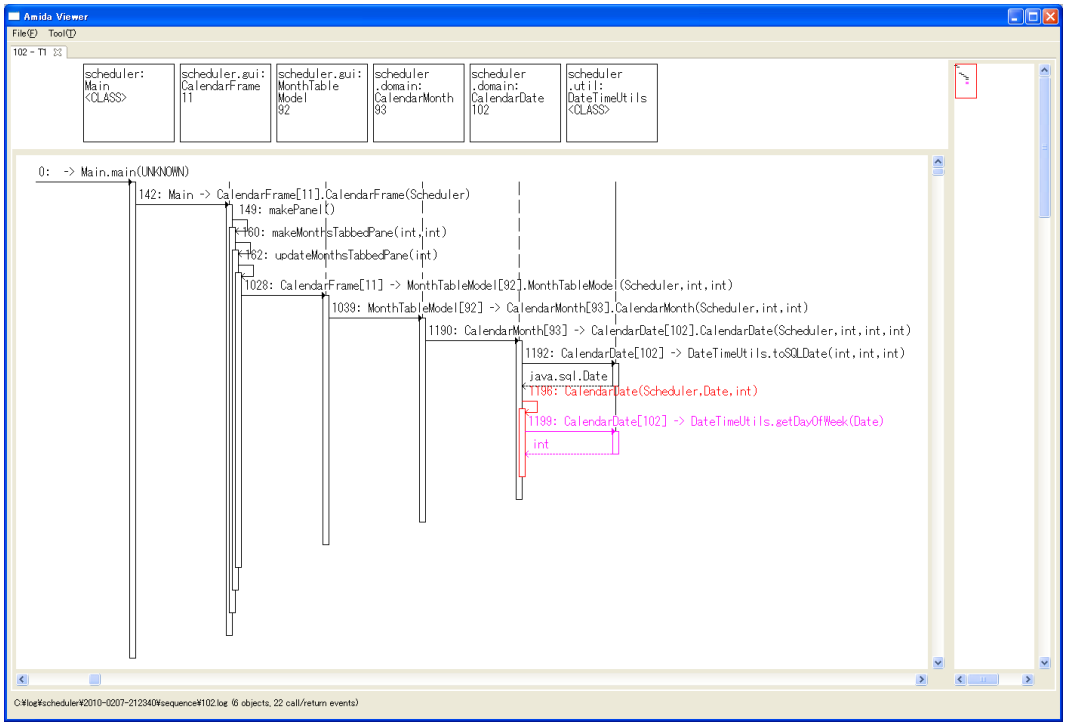


図 12: Amida-Viewer．利用者が選択したイベントに対応するイベントを強調して表示する．

ファイルをグラフ可視化ツール GraphViz で生成している．

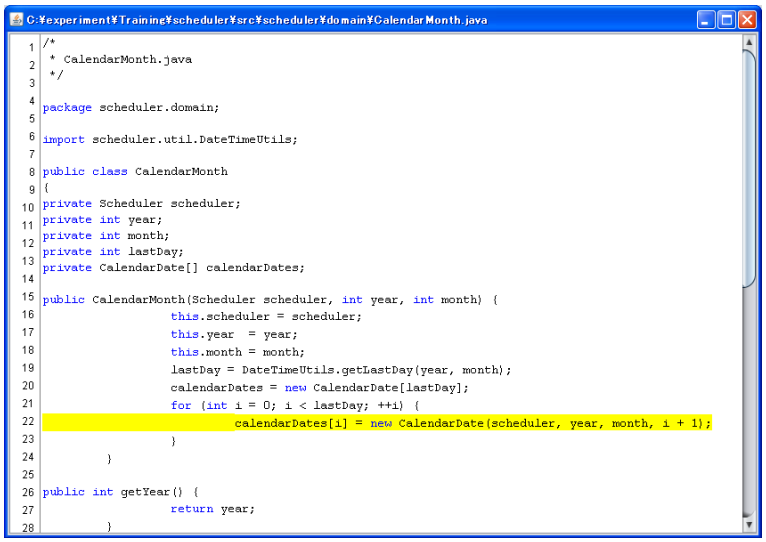


図 13: SourceDialog．利用者が選択したイベントに対応するコードを強調して表示する．

ソースコード取得部

Diagram Dialog で指定されたソースコード位置に対応するソースファイルを読み込み，別ダイアログで表示する．

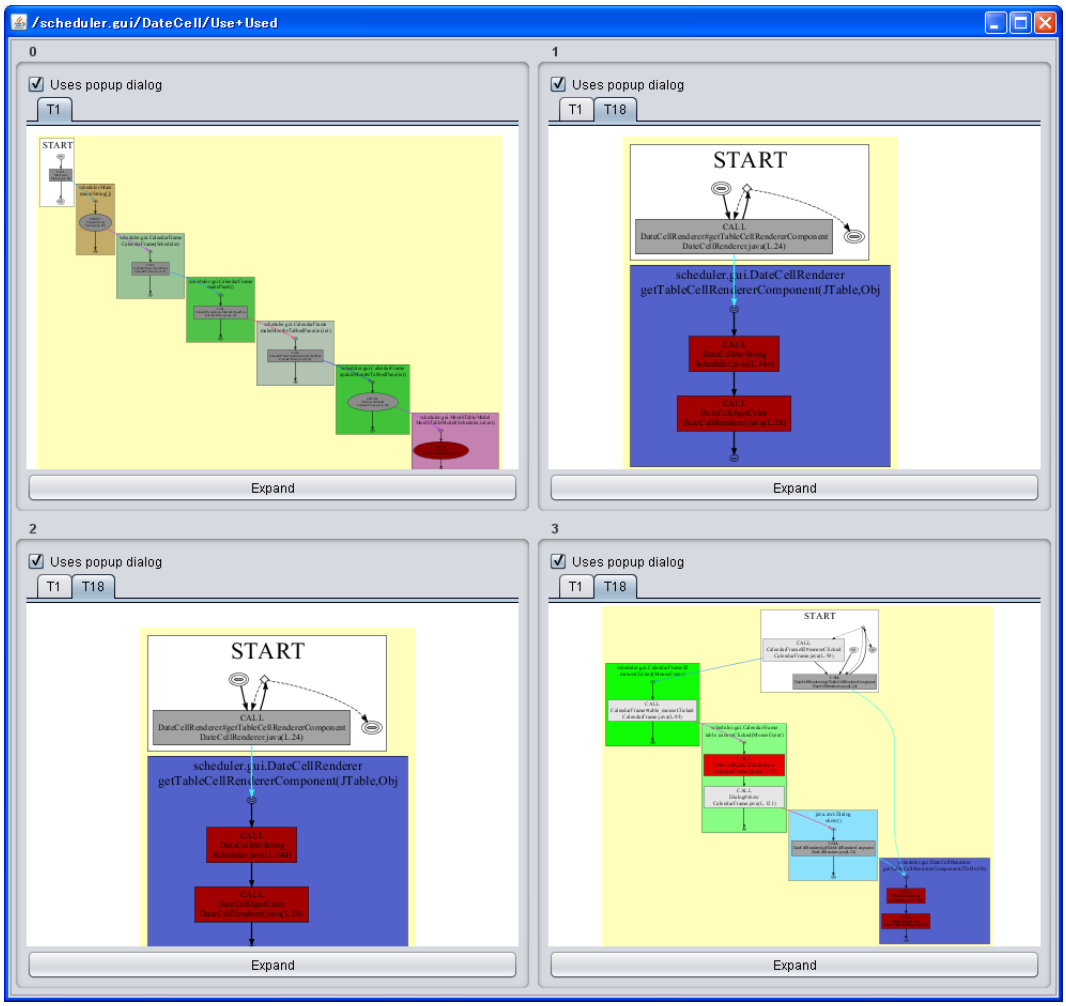


図 14: DiagramDialog．利用者が選択した複数のオブジェクトの振舞いを並べて表示する．

5　評価実験

提案手法のクラス理解に対する有効性を検証するために，実装した Amida-OGAN を，3 つのオープンソースソフトウェアに適用した．Scheduler に対しては，その動作の詳細を分析するケーススタディを実施した．また，MASU と JHotDraw に対して，各クラスを同値分割した結果を調査した．

5.1　ケーススタディ：Scheduler の分析

Scheduler は小規模の予定管理用カレンダープログラムである．図 15(a) に起動直後のウィンドウを，図 15(b) に予定の登録に用いるダイアログのスクリーンショットをそれぞれ示す．

(a) メイン画面　　(b) 登録ダイアログ

図 15: Scheduler のスクリーンショット

ユーザが予定を記入したい日付に対応するセル (日付セル) をクリックすると，新たに登録ダイアログが開き，予定を編集できる

本ケーススタディでは，CalendarDate クラスについて理解を試みる．分析のために，Scheduler を起動し，それぞれ異なる日付に計 3 つの予定を追加する実行シナリオで，実行履歴を取得した．この実行シナリオで得た実行履歴には 7523 個のメソッド呼び出しイベントと，1974 個のオブジェクトが含まれていた．ただし，java,javax,sun,com.sun パッケージに含まれるクラスのイベントは除外している．

取得した実行履歴には，CalendarDate クラスのオブジェクトが 731 個出現していた．これらの振舞いをすべて確認することは，労力が大きく現実的ではない．そこで，CalendarDate クラスの 731 個のオブジェクトに対して，それぞれが参加した機能の違いからくる振舞いの差を識別するために，同値関係 E_{use} に基づいて同値分割を行った．同値分割の結果，3 つの同値類 $G_{use}(CalendarDate) = \{s_1, s_2, s_3\}$ に分割された．各同値類のオブジェクト数と動作コンテキスト集合 Use を表 2 に示す．これらの同値類では，同じ同値類に属するオブジェクトは，呼び出し元のクラスの集合が完全一致する．そこで，同値類ごとの振舞いの差を確認するため，各同値類の振舞いを呼び出し関係図として可視化した．結果を図 16 に示す．図 16 より，大多数のオブジェクトを含む s_1 では CalendarMonth オブジェクトから呼び出されているだけであることなど，同値類ごとに振舞いが大きく異なることがわかる．詳細にみると，同値類 s_2 ，s_3 では，図 16 の破線で示した箇所に，共通する呼び出し関係が確認できる．この共通する呼び出し関係には，日付セルの役割を持つ DateCell オブジェクトからの呼び出しが含まれていることから，同値類 s_2 ，s_3 はメイン画面から使用されていると予測できる．同値類 s_3 では，図 17 より，予定データ登録時に用いたダイアログの役割を持つ RegisterDialog オブジェクトから呼び出されていることが確認でき，予定データの追加機能に参加したと予測できる．

表 2: E_{use} に基づく同値分割 $G_{use}(CalendarDate) = \{s_1, s_2, s_3\}$

Group	#Instances	$Use(o)(o \in s_k)$
s_1	641	CalendarDate,CalendarMonth
s_2	87	CalendarDate,CalendarMonth, DateCell
s_3	3	CalendarDate,CalendarMonth, DateCell,CalendarFrame,RegisterDialog

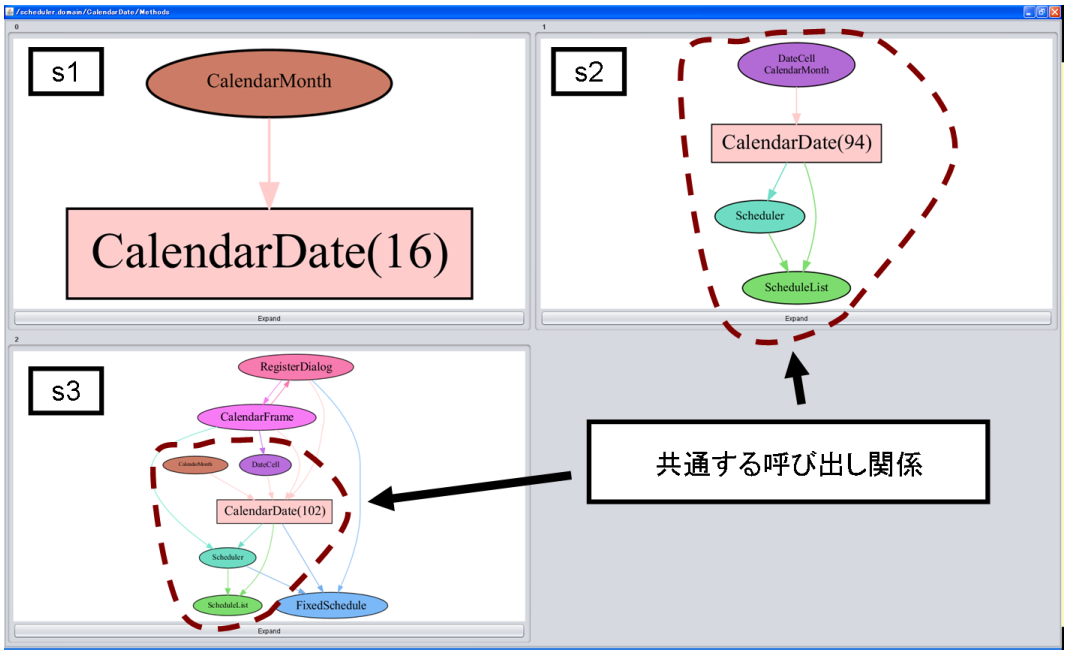


図 16: CalendarDate の各同値類 $G_{use}(CalendarDate) = \{s_1, s_2, s_3\}$ についての呼び出し関係図．同値類 s_2 ，s_3 では，破線で示した箇所に，共通する呼び出し関係が確認できる．

より詳細に振舞いを確認するために，CalendarDate クラスの各同値類 s_1, s_2, s_3 の振舞いを，それぞれ DOPG 図で可視化した．結果を，s_1 については図 18 に，s_2 については図 19 に，s_3 については図 20 にそれぞれ示す．なお，同値類 s_1 から選ばれたオブジェクトはスレッド 18 で動作していなかったため，スレッド 18 についての DOPG 図は存在しない．また，スレッド 1 の DOPG 図は，各同値類から選ばれたオブジェクトすべてで同型だったため，s_2, s_3 の DOPG 図は省略している．

スレッド 1 の振舞いは，DOPG 図で確認する限り完全に同じ呼び出し系列・同じソースコードから実現されており，CalendarMonth オブジェクトのコンストラクタ内の処理で，各

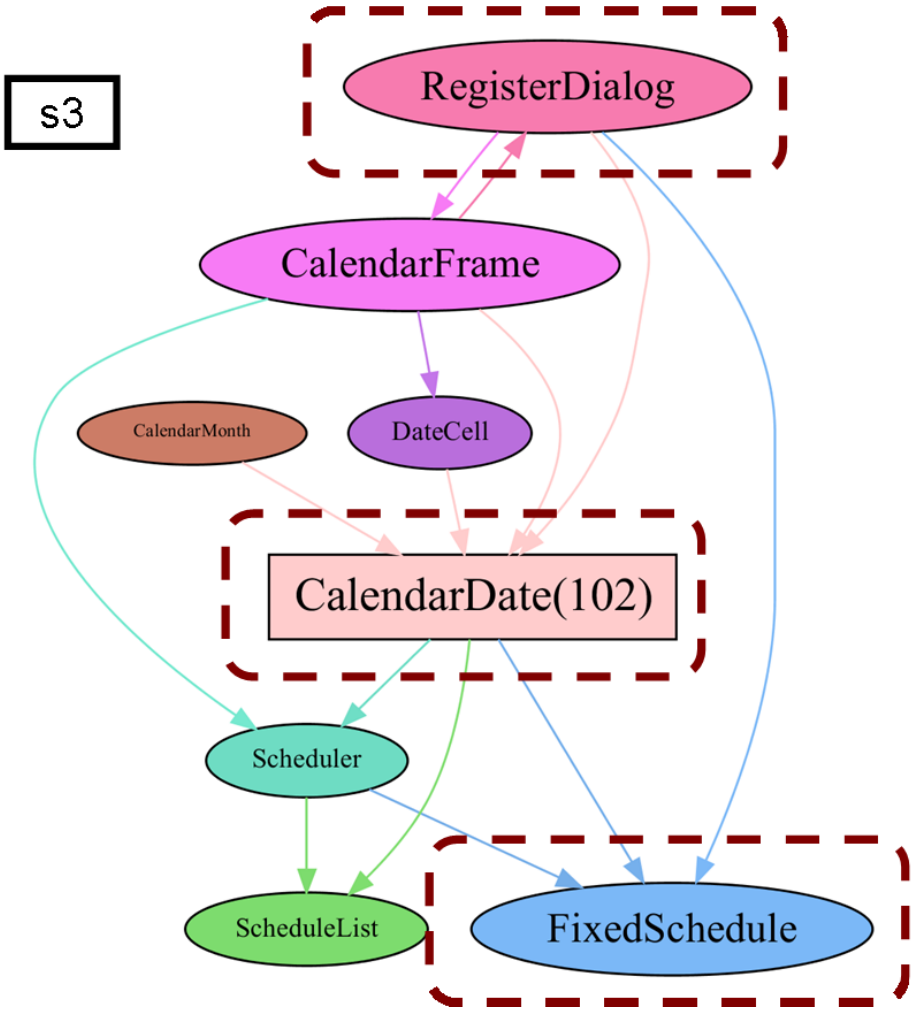


図 17: 同値類 s_3 についての呼び出し関係図．登録ダイアグラムと予定データを CalendarDate オブジェクトが仲介している様子が確認でき，予定追加機能に参加したと予測できる．

CalendarDate オブジェクトは生成されていることがわかる．

スレッド 18 の振舞いは，同値類 s_2 と s_3 で異なる．

同値類 s_2 について，図 19 からは，DateCellRenderer オブジェクトからの描画要求に応えるために，DateCell オブジェクトが CalendarDate オブジェクトに予定データの状態を問い合わせていることがわかる．この振舞いをより詳細に確認するために，同値類 s_2 の振舞いを UML シーケンス図として可視化した．結果を図 21 に示す．図 21 からは，DateCell オブジェクトから予定の状態を問い合わせられた後，CalendarDate オブジェクトは，2 つの役割の異なる予定データ (ScheduleList オブジェクト) の状態を確認していることがわかる．この 2 つの予定データはメソッド名から，日付に対応する予定データと，曜日ごとの予定

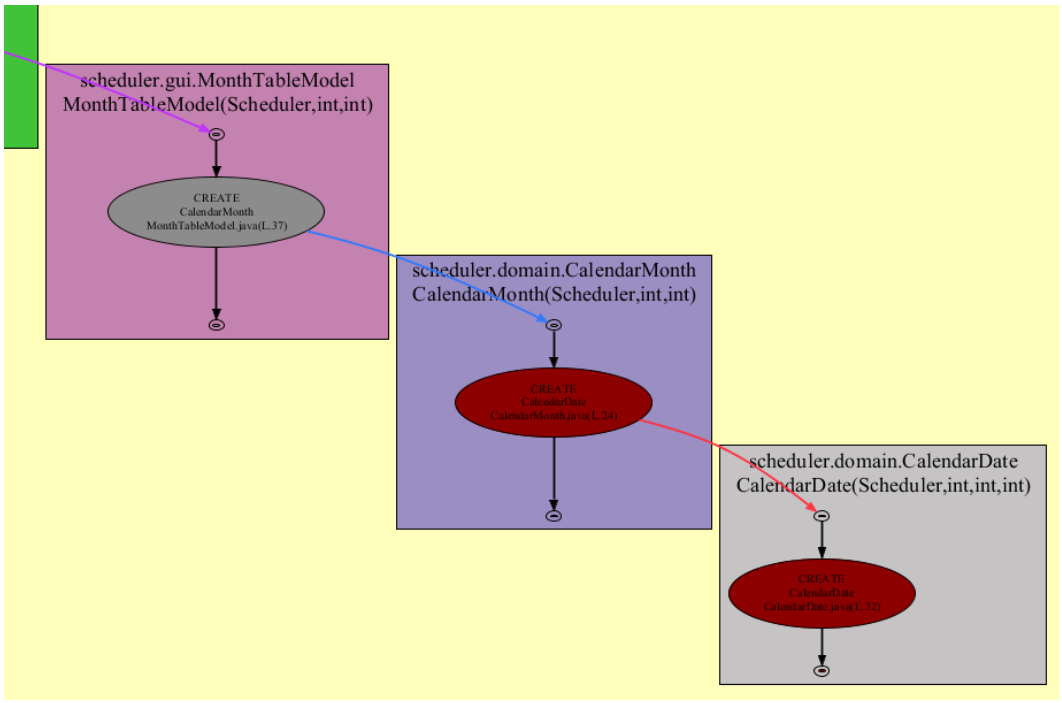


図 18: 同値類 s_1 のスレッド 1 についての DOPG 図.

データであると予測できる．ここまでの分析から，同値類 s_2 は，各日付セルの描画処理に関わったオブジェクト群だとわかった．

同値類 s_3 についての DOPG 図には，同値類 s_2 の振舞いに存在した描画処理に関係する振舞いが含まれていた．これは，図 16 で確認したように，同値類 s_2, s_3 の呼び出し関係図に共通する箇所があることに対応する．また，図 20 に示すように，同値類 s_3 についての DOPG 図の別の部分からは，ユーザのマウス操作を受けて，RegisterDialog オブジェクトから予定データの追加を要求されていることがわかる．この振舞いをより詳細に確認するために，UML シーケンス図として可視化した．結果を図 22 に示す．図 22 からは，RegisterDialog オブジェクトからの予定データ追加要求を受けて，CalendarDate オブジェクトが予定データを予定データベース (Schedule オブジェクト) に追加していることが読み取れる．ここまでの分析から，同値類 s_3 は，予定データ追加処理に関わったオブジェクト群だとわかった．

ここまでの分析で，CalendarDate クラスの役割として，次のことがわかった．

- 各日付セルは描画に用いる色情報を生成するために，CalendarDate を仲介して，予定データの状態を取得している．このとき，CalendarDate は予定データの状態として，当日の予定データと曜日ごとの予定データの状態を，それぞれ調べている．
- 登録ダイアログは追加ボタンが押されたあと，CalendarDate を仲介して，予定デー

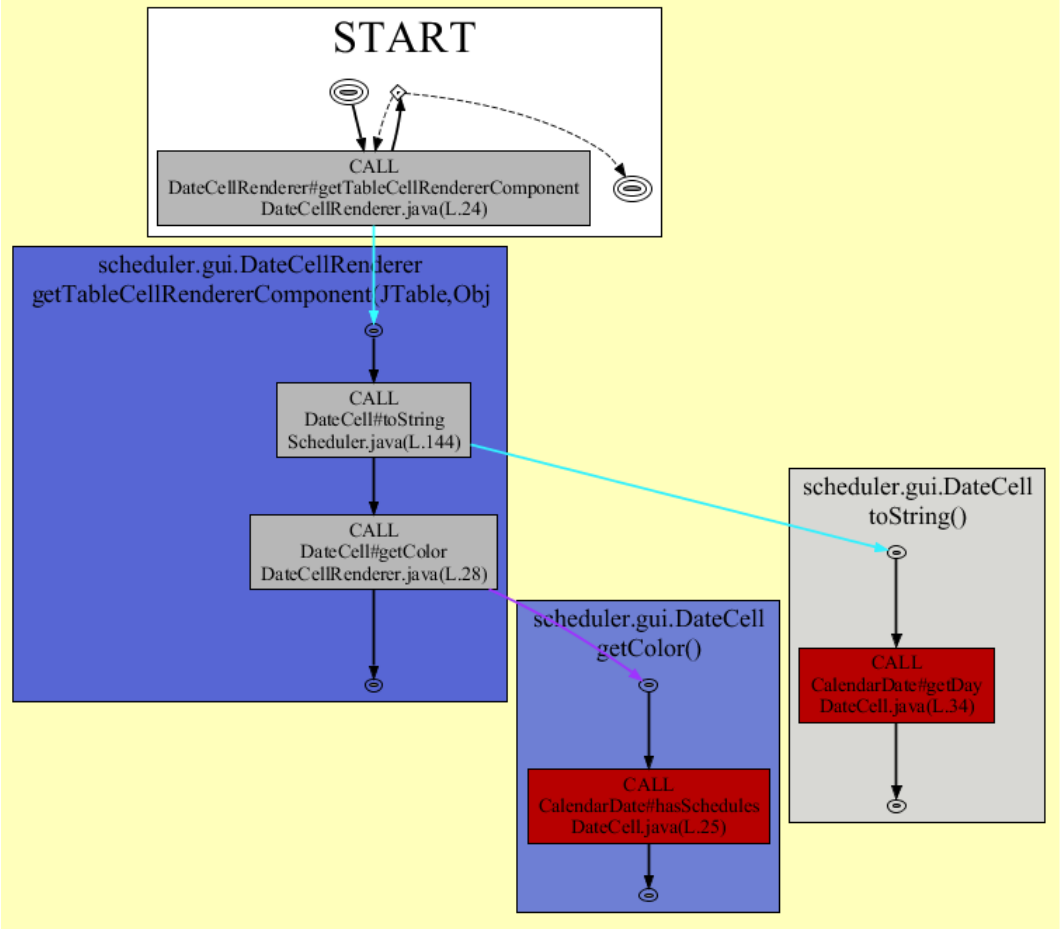


図 19: 同値類 s_2 のスレッド 18 についての DOPG 図．DateCellRenderer からの描画要求に応えるために，DateCell が色情報とテキスト情報を CalendarDate をからの応答に基づいて生成している．

タベースに予定を追加している．

後により詳細にプログラム理解を行ったところ，同値類 s_1 はプログラム実行中に表示されなかった日付のデータ，s_2 は表示された日付のデータで描画機能に参加したもの，s_3 は表示された後に予定が追加されたデータで描画機能と予定追加機能に参加したものにそれぞれ対応していることがわかった．同値関係 E_{use} に基づく同値分割の目的は，参加した機能の違いからくる振舞いの差を識別することであるため，このケーススタディでは，目的を達成できていると考えられる．

本ケーススタディで注目した CalendarDate クラスでは，大多数のオブジェクトは GUI と予定データに関わっていなかった．そのため，すべてのオブジェクトからランダムにオブ

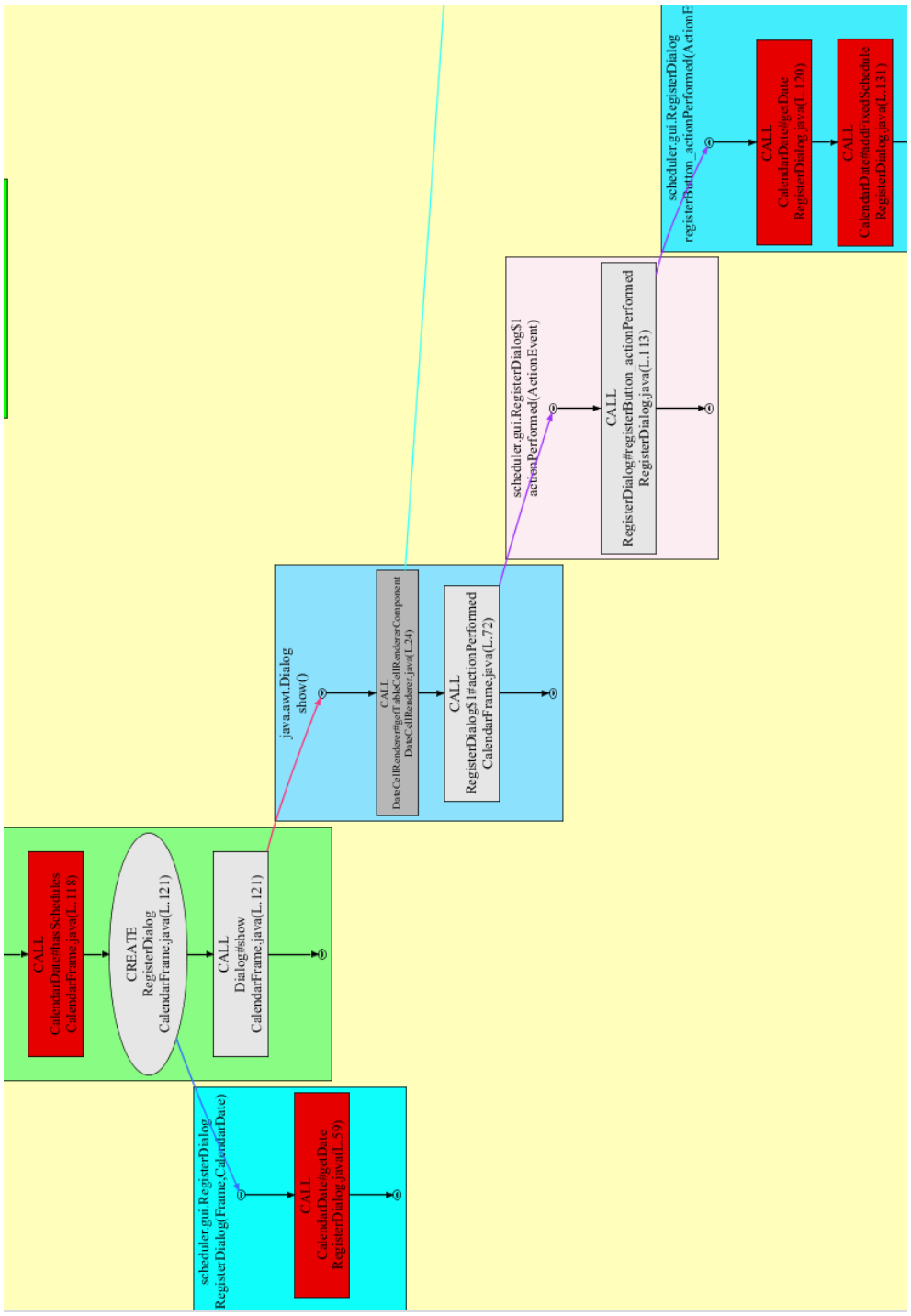


図 20: 同値類 s_3 のスレッド 18 についての DOPG 図．ユーザ操作を受けて，RegisterDialog が CalendarDate に予定データの追加を要求している．

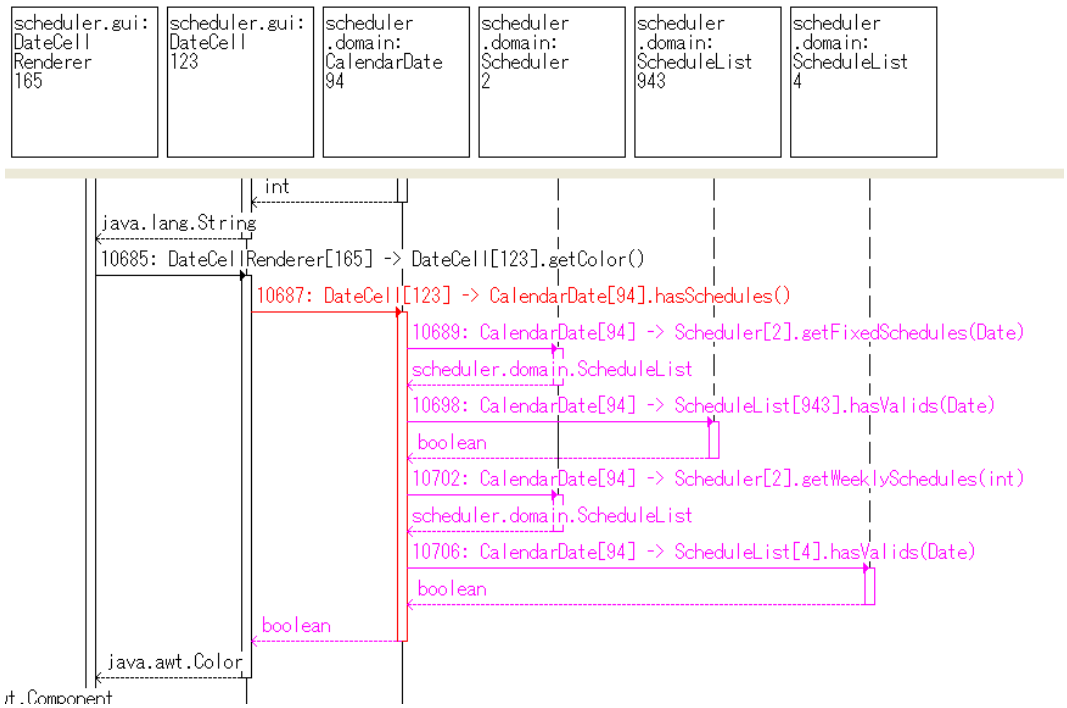


図 21: 同値類 s_2 のスレッド 18 についてのシーケンス図．DateCell からの問い合わせに応えるために，2 つの役割の異なる ScheduleList の状態を確認している．

ジェクトを 1 つ選択し可視化しただけでは，描画処理と予定追加処理についての知識は獲得できない可能性が高い．また，すべての振舞いを確認することは，労力が大きく困難である．本ケーススタディでは，CalendarDate クラスの 731 個のオブジェクトを，その振舞いの同値性に基づいて同値分割することで，初期化，描画処理，予定追加処理が含まれる振舞いのみをそれぞれ提示することができた．そのため，728 個のオブジェクトの振舞いについては，詳細な分析を省略することで，分析時間を短縮することができた．

CalendarDate クラスのすべてのオブジェクトについて呼び出し関係図を生成し，互いに同型判定をすることで，全部で 3 パターンあることがわかった．同型 (デザイン，テキストが同一) な呼び出し関係図を比較・参照することで，新たに得られる知識は少ないと考えられる．本ケーススタディでは，各同値類 s_1, s_2, s_3 から可視化した 3 つの呼び出し関係図は，この 3 パターンを網羅していた．そのため，呼び出し関係図を可視化手法に用いた場合には，CalendarDate クラスの理解に効果的な振舞いのみを可視化できていると言える．

しかし，CalendarDate クラスのすべてのオブジェクトの DOPG を解析すると，6 パターンあることがわかった．これは (1) 実行スレッドの違い，(2)2 階層以上の呼び出し階層の違いからくるものだった．この DOPG 図の違いを，図 23,24 に示す．実行スレッドや推移的な

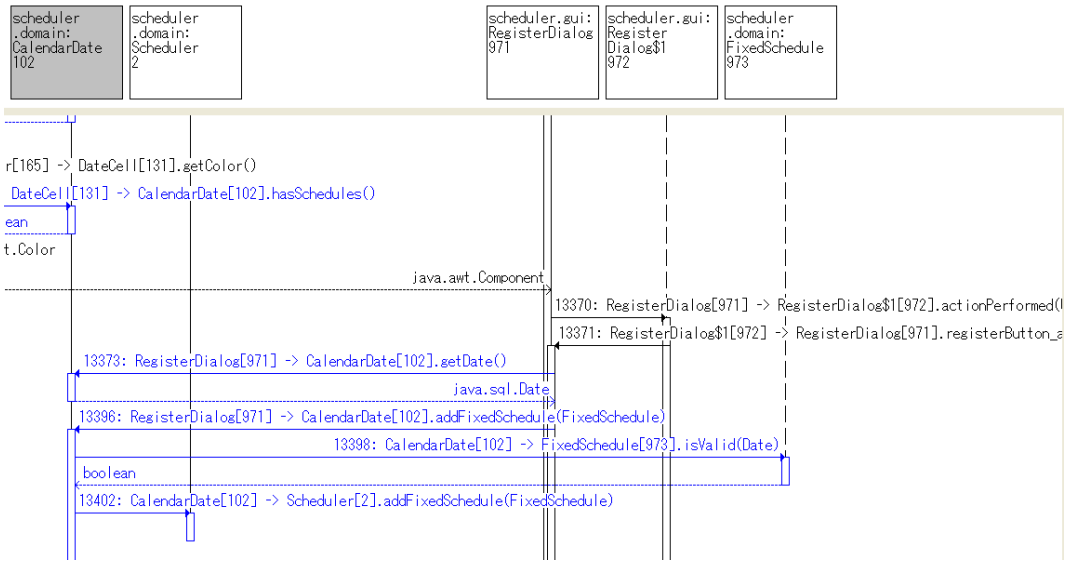


図 22: 同値類 s_3 のスレッド 18 についてのシーケンス図．RegisterDialog からの予定データ追加要求を受けて，実際に予定データを登録している

呼び出しの違いからくる振舞いの差は，本手法ではオブジェクトの振舞いの同値性の判断に用いておらず，識別できない．クラス理解において，オブジェクトの振舞いとして，このような振舞いの差を識別すべきかどうかについては，今後議論を深める必要がある．

本ケーススタディで分析した一連の処理について，各メソッド呼び出しに対応するソースコードをそれぞれ確認することで，CalendarDate クラスの動作，特にこの場合は，Scheduler の GUI と予定データとの相互作用に注目してソースコード読解を進められる．これらの処理は，複数のソースファイルに分散して記述されているため，メソッド間の関連を解析しながら横断的に読解を進める必要がある．オブジェクトの振舞いを分析すると，注目する動作に関連するソースコードの位置が把握できるため，ソースコード読解の際の有効な足掛かりになる．これは第 2.2 節で説明した，動的解析によるプログラム理解のゴール指向という利点からくるものである．

5.2 分割結果の調査

オープンソースソフトウェアである MASU と JHotDraw から取得した実行履歴に提案手法を適用し，各クラスを同値分割した結果を調査した．本調査では，表 3 に示す実行履歴を分析した．

表 4 は，実行時に生成されたオブジェクト数でクラスをカテゴリ分けし，そのカテゴリに属するクラスの個数と，各分割基準ごとの同値類数の最小値/平均値/最大値を示している．

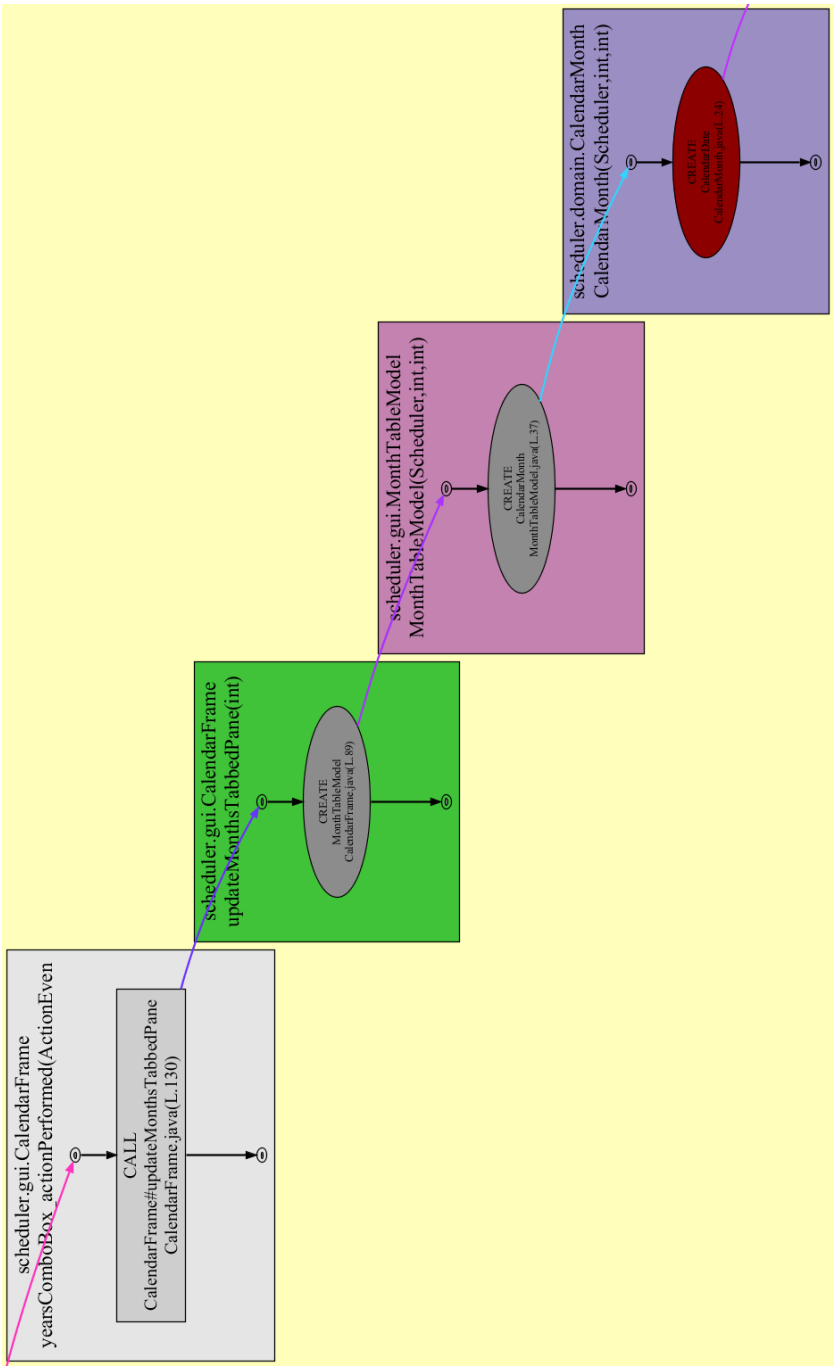


図 23: 同値類 $s1$ に含まれるオブジェクトの内，スレッド 18 で動作したものの DOPG 図. CalendarFrame の yearsComboBox_actionPerformed メソッドからの推移的な呼び出しでコンストラクタを呼び出されている.

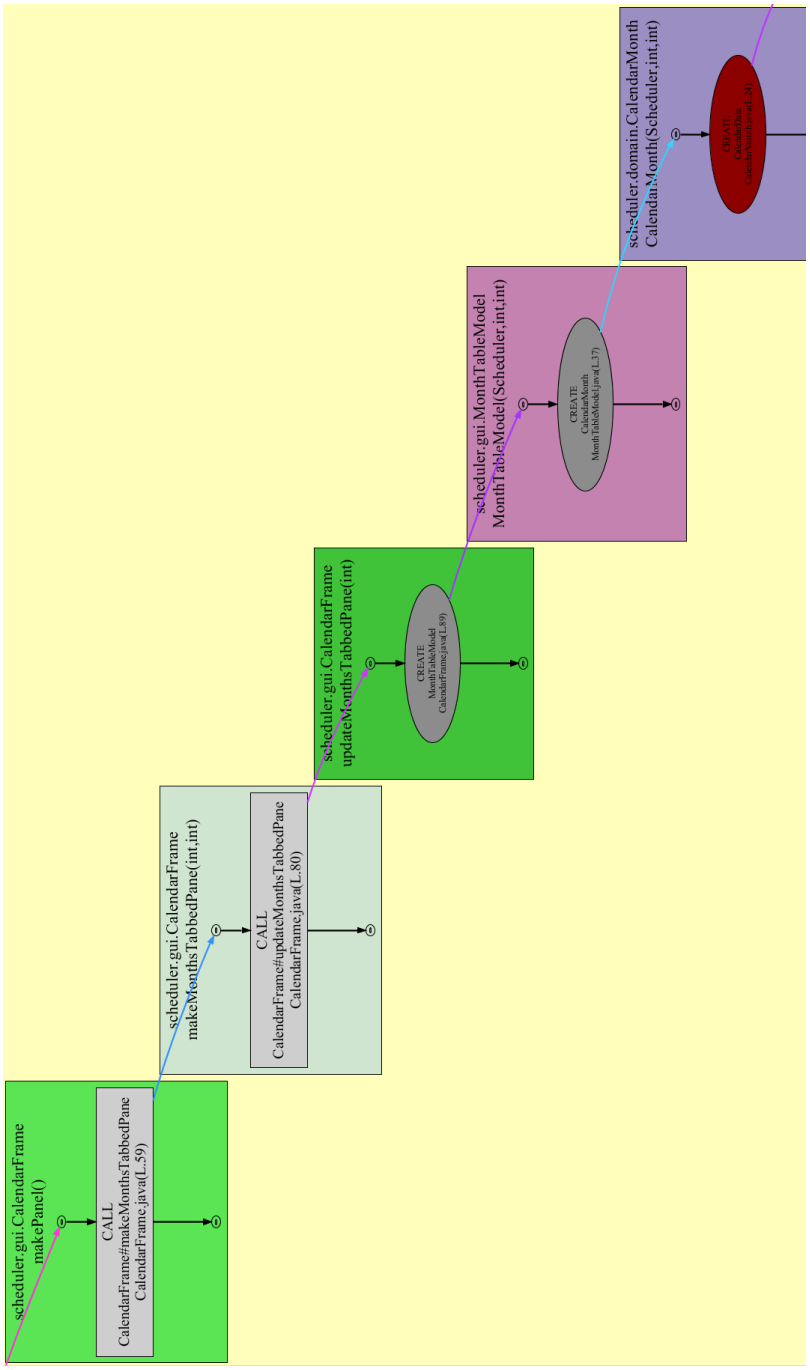


図 24: 同値類 $s1$ に含まれるオブジェクトの内，スレッド 1 で動作したものの DOPG 図．CalendarFrame の makeMonthsTabbedPane メソッドからの推移的な呼び出しでコンストラクタを呼び出されている．

表 3: 分析対象

ソフトウェア	実行シナリオ	出現オブジェクト数
MASU[45]	1 クラスの RFC メトリクスを測定	12414
JHotDraw[16]	矩形を 1 つ作成した後，それをコピー＆ペースト	6286

表 4 から，分割基準 $E_{methods}, E_{called}$ では，どのカテゴリにおいても同値類数の平均値は 6 未満，最大値は 12 未満になっていることがわかる．特に，オブジェクト数が 100 以上のクラスであっても同値類数が少数であることは，同値類ごとに振舞いを可視化し比較する際に有利であり，本研究の目的に合致する．生成されるオブジェクト数によらず，同値類数が大体一定になるのは，各クラスが実際に行う処理のバリエーションが限られていることを示唆している．つまり，ソースコード上では，実現可能性のあるオブジェクトの振舞いのバリエーションは多数あるが，実際のプログラムの実行では，振舞いは少数のパターンに従うのではないかと推測される．

本調査では，オブジェクトが特に多数生成されたクラスは，MASU では StateChangeEvent や NamespaceInfo など，JHotDraw では ResourceBundleUtil や AttributeAction など，(1) データ，(2) イベントオブジェクト・イベントハンドラ，(3) ユーティリティの役割を持つクラスである傾向が高かった．これらのクラスを調査すると，(1) アクセッサ以外のメソッドが少ない，(2) ライフタイムが短い，(3) コールバック以外で多態性を活用する処理が少ない，といった傾向があったため，使われ方を識別する $E_{methods}$ や，実装の違いを識別する E_{called} で，同値類数が特に低かったと推測される．これらのクラスのオブジェクトは，多くの他のクラスから横断的に使用される傾向があるため，そのオブジェクトの振舞いを確認することで，プログラムの全体的な動作に関する理解が得られる場合がある．これは 5.1 節のケーススタディで解析した，CalendarDate クラスについてもあてはまる．本手法は，このようなクラスの動作理解に効果的であると考えられる．

分割基準 E_{use} では，MASU のオブジェクト数 1000〜9999 のカテゴリで同値類数が平均 21.33 個，JHotDraw のオブジェクト数 100〜999 のカテゴリで同値類数が平均 19.14 個と，他の分割に比べて多くなっている．これは，MASU では StateChangeEvent クラスがオブジェクト数 8344 で同値類数 45，JHotDraw では ResourceBundleUtil がオブジェクト数 959 で同値類数 58 個と，多数の同値類に分割されているためであるが，前述のように，このようなイベントハンドラやユーティリティでは，多くのクラスから横断的に使用される傾向があるために，呼び出し元のクラスの違いを識別する Use では，多数の同値類に分割されていると考えられる．

JHotDraw のオブジェクト数 100〜999 のカテゴリで，分割基準 E_{used} での同値類数が最大

表 4: 分割基準ごとの同値類数の最小値，平均値，最大値

ソフトウェア	オブジェクト数	クラス数	E_{use}	E_{used}	$E_{methods}$	E_{called}
MASU	$2 \leq N$	79	1/ 4.42/45	1/2.63/5	1/2.40/10	1/2.37/5
	$1000 \leq N \leq 9999$	3	5/21.33/45	1/1.33/1	3/5.66/3	1/2.00/1
	$100 \leq N \leq 999$	2	5/ 5.50/6	1/1.50/5	1/5.50/10	1/2.00/5
	$10 \leq N \leq 99$	25	1/ 6.24/12	5/4.24/5	1/2.76/1	5/3.36/5
	$2 \leq N \leq 9$	49	1/ 2.36/6	1/1.93/6	1/1.83/3	1/1.87/6
JHotDraw	$2 \leq N$	97	1/ 4.47/58	1/1.89/25	1/1.82/11	1/1.78/10
	$1000 \leq N \leq 9999$	1	2/ 2.00/ 2	1/1.00/ 1	2/2.00/ 2	1/1.00/ 1
	$100 \leq N \leq 999$	7	1/19.14/58	1/4.85/25	1/3.14/11	1/2.57/10
	$10 \leq N \leq 99$	30	1/ 6.76/17	2/2.10/ 2	2/2.00/ 2	2/2.16/ 2
	$2 \leq N \leq 9$	59	1/ 1.61/ 8	1/1.45/ 8	1/1.57/ 6	1/1.50/ 6

25 個と多くなっている．これは AbstractSelectedAction クラスの内部クラス EventHandler の分割結果に対応している．EventHandler クラスは，通知されたイベントオブジェクトから取得した様々なアクションオブジェクトに対してコールバックするが，このコールバック対象のアクションオブジェクトのクラスの違いを分割基準 E_{used} は識別するために，多数の同値類に分割されている．EventHandler クラスの E_{called} に基づく分割では同値類数が 5 個と少数であることから，アクションオブジェクトの使い方には差が少ないと考えられる．このように，多態性を活用した処理では，使用するメソッドに差が少ない場合でも，E_{used} に基づくと多数の同値類に分割されてしまう傾向がある．

5.3 考察

ケーススタディにより，本手法で，プログラム実行時に多数のオブジェクトが生成されるクラスについて，そのクラスの特徴的な振舞いのみを図として提示できることを示した．提示された振舞いは，そのクラスのオブジェクト数に比べて十分に少数であり，利用者が現実的な時間で確認できるものだった．なお，著者らは，効果的に比較・確認できる振舞いの数は 9 以下だと，ケーススタディやツールの作成にあたり複数人の被験者に使用してもらった時の知見から推測している．9 つより多くの振舞いを図として可視化した場合には，各図を 1 画面に並べると図のサイズが小さくなり，相互に比較することが難しくなるためである．各図を相互に比較することで，各同値類に共通な動作と，少数の同値類に限定的な動作とを

区別して確認することができ，より深くクラスを理解できるほか，興味ある機能に関係する振舞いを効率良く見つけることができると考えている．

ケーススタディで提示された振舞いはそれぞれに実現している処理が異なっており，1 つのオブジェクトの振舞いを可視化するだけでは，確認することのできない処理があることを実際に示した．このような場合に，単一オブジェクトの振舞いを可視化する手法を効果的に適用するには，本手法のように，複数のオブジェクトの振舞いの差を識別して提示する手法が必要だと主張する．

しかし，本手法では次の違いからくる振舞いの差を識別できない．

- オブジェクトが動作した実行スレッドの違い
- 2 階層以上のメソッド呼び出し階層の違い
- メソッド呼び出し順序の違い
- メソッド呼び出しを引き起こしたソースコード位置の違い

これらの違いからくる振舞いの差を識別してオブジェクト群を分割した場合，より多数の同値類に分割されると考えられる．本手法の目的は，多数のオブジェクトの振舞いを確認する際の労力を減らすことだが，多数の同値類に分割される場合には，その効果は低い．クラス理解において，オブジェクトの振舞いとして，どの程度の振舞いの差を識別すべきかについては，今後議論を深める必要がある．

また，ケーススタディを通して，本手法では，利用者は分析対象クラスと分析目的に適した分割基準を選択することで，分析対象プログラムに関する前提知識無しに，分析対象クラスの特徴的な振舞いのみを提示できることを示した．提示された少数の振舞いを相互に比較・確認することで，クラスのオブジェクトに共通する振舞いと限定的な振舞いとを区別して確認することができる．本手法は，このように，分析対象プログラムに関する知識が少ない利用者が，段階的にクラスについての知識を効果的に獲得できることを目的としている．

分割基準と同値類数の調査からは，プログラム中で使用されているクラスの多くで，同値分割で得られる同値類の個数は 10 前後に収まることがわかった．オブジェクト数が多いクラスほど，分割で得られる同値類数は多くなる傾向があったが，増加傾向は緩やかであった．同値類数が少数である場合，同値類ごとに振舞いを可視化し比較・確認することが現実的な時間でできるため，提案手法を効果的に適用できるクラスは多く存在すると言える．

しかし，次のケースでは，同値分割により多数の同値類に分割された．

- **多くの異なるクラスから横断的に使用されるクラス：** 分割基準 E_{use} で多数の同値類に分割される．データクラス，イベントオブジェクトクラス，ユーティリティクラスなど．

- **多態性を活用したメソッド呼び出しを行うクラス：** 分割基準 E_{used} で多数の同値類に分割される．イベントハンドラクラスなど

前述のように，多数の同値類に分割される場合，各振舞いを確認する労力を減らす効果は低い．上記のケースでは，振舞いの些細な差をある程度許容するような分割基準を定義することで，同値類数を減らす必要があると考えている．

一方，これらのクラスでも，$E_{methods}, E_{called}$ に基づく分類では，同値類数が 10 前後に収まっていた．このような場合，まず，$E_{methods}, E_{called}$ で分割した後に，詳細に確認したい振舞いの同値類についてのみ再帰的に E_{use}, E_{used} で分割することで，効率良く振舞いの確認を行うことができると考えている．実際に，このような効果的な分割手順が存在するか，今後より多くのケースで調査していきたい．

本手法の別用途として，2 つのクラス間の特徴的な相互作用の抽出に応用することが考えられる．注目する 2 つのクラスについて，それらのすべてのオブジェクトをそれぞれ同値分割し，2 つのクラスの各同値類間で実際に呼び出し関係のあるオブジェクト組を抽出し，その相互作用を図として可視化することで，2 つのクラスの特徴的な相互作用のみを提示できると考えている．例えば，データクラスとデータ管理クラスの間の特徴的な相互作用を確認すると，典型的なデータ管理手順が理解できると考えられ，プログラムの部分理解 (Partial Comprehension) に有効であると予測している．

6 関連研究

実行履歴を可視化することで，オブジェクト指向プログラムの理解支援を目指す同様の手法は数多く提案されている．実行履歴の可視化にシーケンス図を用いる手法 [6, 10, 15]，クラス間の呼び出し関係図を用いる手法 [17, 35]，その両方を併用する手法 [7]，コールツリーと種々のチャートを併用する手法 [23] などがあり，このような実行履歴可視化ツールの調査もされている [4, 14].

本研究は，特定クラスの効果的な理解を目的に，そのクラスのオブジェクトの振舞いを可視化するものである．同じように，オブジェクトの振舞いに注目した研究は多く行われている．

Reiss らは，あるクラスの使い方を知りたい時，そのクラスのオブジェクトに対するメソッド呼び出しの系列だけを実行履歴から抽出し (=クラス使用例)，それぞれを文字列表現やオートマトンに変換して確認することが有効だと主張した [31]．Salah らは，文字列表現に変換したクラス使用例を編集距離に基づいてクラスタリングし，クラスタごとに含まれるすべての文字列表現を簡潔に表現できる正規表現を求めることで，クラスのより典型的な使用例を抽出する手法を提案した [34]．クラスの使用例は，そのクラスを再利用する時や，使い方の検証をする時に有効だが，本研究のように，プログラム理解を目的としてクラス内部の動作・実装を知りたい時には，情報が不足している．

Systa は，クラスの理解はメンテナンス作業に重要であること，クラスの理解にはそのオブジェクトの振舞いの可視化が有効であること，理解することがプログラム全体の理解に不可欠であるようなキークラスが存在することを主張した [38]．そして，オブジェクトの振舞いを可視化するために，複数の部分シーケンス図からオブジェクトについてのステートチャートダイアグラムを生成する手法を提案した．この手法のケーススタディにおいて，17 個のオブジェクトから 1 つを選択し可視化しているが，選択した理由や基準や妥当性は議論されていない．Dallmeier らは，オブジェクトの内部状態の変化に注目し，Getter メソッドに代表される状態監視用 (inspector) メソッドの戻り値で内部状態を，Setter メソッドに代表される状態変更用 (mutator) メソッドの呼び出しで状態遷移を表現する，オブジェクトの振舞いモデルを提案した [9]．Xie らは，オブジェクトの内部状態をフィールドの値，状態遷移をオブジェクトに対するメソッド呼び出しで表現する，オブジェクト状態機械を提案した．オブジェクトの内部状態は，オブジェクトごとに異なると予想されるが，これらの研究でも，オブジェクトの選択が理解に与える影響については議論されていない．本研究では，オブジェクトが多数生成されるクラスでは，オブジェクトの選択がクラス理解に与える影響は無視できないと考えている．

Richner らは，GUI ベースのツールを用いて，利用者が呼び出し元と呼び出し先のクラス・

オブジェクト，呼び出したメソッドを順次対話的に選択することで，その条件を満たすメソッド呼び出しを起点とする呼び出し系列を，シーケンス図として可視化する手法を提案した [33]．呼び出し元のクラスなど，表示されている要素を選択するたびに，呼び出し先のクラスなど，次の選択候補が自動的に表示される．Richner らは，プログラムに関する知識の少ないユーザが実行履歴を分析するには，分析作業を対話的に支援する必要があると主張しており，本研究でもこの主張を踏襲している．

7　おわりに

プログラム実行時にオブジェクトが多数生成されるクラスでは，各オブジェクトの振舞いを確認する労力が増大し，クラスの理解が困難になる．そこで，オブジェクト群を振舞いの同値性に基づいて同値分割する手法を提案した．分割によりオブジェクト群に特徴的な振舞いのみを図として可視化することができる．また，利用者は各図をそれぞれに比較することで，より効果的にクラスを理解できる．提案手法をツールとして実装し，3 つのオープンソースソフトウェアに対して適用実験を行い，クラスの理解に有効であることを確認した．

今後の課題を，次にあげる．

- **多数の同値類への対策：** オブジェクト群を分割する目的は，オブジェクトの振舞いの確認に要する労力を減らすことであり，同値類数が多い場合にはその効果は低い．些細な振舞いの違いをある程度許容して分割することで，分割数を減らそうと考えている．
- **メンテナンス作業での有効性評価：** 小規模なケーススタディだけでなく，実際に行われているメンテナンス作業で提案手法が有効に作用するのかを，被験者を用いて評価する必要がある．

謝辞

本研究の全過程を通して，常に適切な御指導，御助言を賜りました　大阪大学大学院情報科学研究科 コンピュータサイエンス専攻　井上 克郎 教授に心より深く感謝いたします.

本研究の全過程を通して，常に適切な御指導，御助言を賜りました　大阪大学大学院情報科学研究科 コンピュータサイエンス専攻　松下 誠 准教授に心より深く感謝いたします.

本論文を作成するにあたり，常に適切な御指導，御助言を賜りました　大阪大学大学院情報科学研究科 コンピュータサイエンス専攻　石尾 隆 助教に心より深く感謝いたします.

評価実験の実施にあたり，メトリクス計測フレームワーク MASU の開発者として，作業内容を監修して頂きました　大阪大学大学院情報科学研究科 コンピュータサイエンス専攻　肥後 芳樹 助教，山田 吾郎 氏，斎藤 晃 氏に深く感謝いたします

ツールの作成にあたり，有益な御助言を多数頂きました　大阪大学大学院情報科学研究科 コンピュータサイエンス専攻　楠本研究室の学生の皆様に深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂いた　大阪大学大学院情報科学研究科 コンピュータサイエンス専攻　井上研究室の皆様に深く感謝いたします.

参考文献

[1] G. Arevalo, F. Buchli, and O. Nierstrasz. Detecting implicit collaboration patterns. In *the 11th Working Conference on Reverse Engineering*, pp. 122–131, 2004.

[2] Batik. Batik - java svg toolkit(`http://xmlgraphics.apache.org/batik/`).

[3] T. Bell. The concept of dynamic analysis. In *the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 216–234, 1999.

[4] C. Bennett, D. Myers, M.-A. Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland. A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. In *Special Issue on Program Comprehension through Dynamic Analysis*, Vol. 20, pp. 291–315, 2008.

[5] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *the 15th International Conference on Software Engineering*, pp. 482–498, 1993.

[6] T. A. Corbi. Program understanding: Challenge for the 1990s. In *IBM Systems Journal*, Vol. 28, pp. 294–306, 1989.

[7] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *the 15th International Conference on Program Comprehension*, pp. 49–58, 2007.

[8] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. In *IEEE Transactions on Software Engineering*, p. RapidPost, 2010.

[9] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with adabu. In *the 4th International Workshop on Dynamic Analysis*, pp. 17–24, 2006.

[10] Eclipse. Eclipse test and performance tools platform project(`http://www.eclipse.org/tptp/`).

[11] Eclipse. Eclipse(`http://www.eclipse.org/`).

[12] GraphViz. Graphviz(`http://www.graphviz.org/`).

[13] T. Gschwind and J. Oberleitner. Improving dynamic data analysis with aspect-oriented programming. In *the 7th European Conference on Software Maintenance and Reengineering*, pp. 259–268, 2003.

[14] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pp. 42–55, 2004.

[15] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *the 19th International Conference on Software Engineering*, pp. 360–370, 1997.

[16] JHotDraw. Jhotdraw as open-source project(`http://www.jhotdraw.org/`).

[17] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *ACM SIGPLAN Notices*, Vol. 30, pp. 342–357, 1995.

[18] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *the 28th International Conference on Software Engineering*, pp. 492–501, 2006.

[19] M. Lejter, S. Meyers, and S. P. Reiss. Support for maintaining object-oriented programs. In *IEEE Transactions on Software Engineering*, Vol. 18, pp. 1045–1052, 1992.

[20] OMG. Unified modeling language (uml) 1.5 specification, 2003.

[21] M. J. Pacione, M. Roper, and M. Wood. A novel software visualization model support software comprehension. In *the 11th Working Conference on Reverse Engineering*, pp. 70–79, 2004.

[22] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented system. In *the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 326–337, 1993.

[23] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems*, Vol. 4, pp. 1–16, 1998.

[24] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *the 24th International Conference on Automated Software Engineering*, 2009.

[25] W. Pree, 佐藤啓太 (訳), 金沢典子 (訳). デザインパターンプログラミング (原題:Design Patterns for Object-Oriented Software Development). 凸版印刷, 1996.

[26] J. Quante. Do dynamic object process graphs support program understanding? -a controlled experiment. In *the 16th International Conference on Program Comprehension*, pp. 73–82, 2008.

[27] J. Quante and R. Koschke. Dynamic object process graphs. In *the 10th European Conference on Software Maintenance and Reengineering*, pp. 81–90, 2006.

[28] J. Quante and R. Koschke. Dynamic object process graphs. In *Journal of Systems and Software*, Vol. 81, pp. 481–501, 2008.

[29] D. Rayside, L. Mendel, and D. Jackson. A dynamic analysis for revealing object ownership and sharing. In *the 4th International Workshop on Dynamic Analysis*, pp. 57–64, 2006.

[30] S. P. Reiss. Dynamic detection and visualization of software phases. In *the 3rd International Workshop on Dynamic Analysis*, pp. 1–6, 2005.

[31] S. P. Reiss and M. Renieris. Encoding program executions. In *the 23rd International Conference on Software Engineering*, pp. 221–230, 2001.

[32] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *the 15th International Conference on Software Maintenance*, pp. 13–22, 1999.

[33] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *the 18th International Conference on Software Maintenance*, pp. 34–43, 2002.

[34] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. I. Vokolos. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *the 21st International Conference on Software Maintenance*, pp. 155–164, 2005.

[35] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interactions. In *the 20th International Conference on Software Maintenance*, pp. 72–81, 2004.

[36] M.-A. Storey. Theories, methods and tool in program comprehension: Past, present and future. In *the 13th International Workshop on Program Comprehension*, pp. 181–191, 2005.

[37] Sun Microsystems. Java virtual machine tool interface(http://java.sun.com/javase/6/docs/technotes/guides/jvmti/).

[38] T. Systa. Understanding the behavior of java programs. In *the 7th Working Conference on Reverse Engineering*, pp. 214–233, 2000.

[39] W3C. Scalable vector graphics(http://www.w3.org/Graphics/SVG/).

[40] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. In *IEEE Transactions on Software Engineering*, Vol. 18, pp. 1038–1044, 1992.

[41] T. Xie and D. Notkin. Automatic extraction of sliced object state machines for component interfaces. In *the 3rd Workshop on Specification and Verification of Component-Based Systems*, pp. 39–46, 2004.

[42] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *9th European Conference on Software Maintenance and Reengineering*, pp. 134–142, 2005.

[43] 谷口, 石尾, 神谷, 楠本, 井上. プログラム実行履歴からの簡潔なシーケンス図の生成手法. コンピュータソフトウェア, 第 24 巻, pp. 153–169, 2007.

[44] 千葉. Javassist home page(http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html).

[45] 三宅, 肥後, 井上. メトリクス計測プラグインプラットフォーム masu の開発. ソフトウェアエンジニアリング最前線 2008, pp. 63–70, 2008.