ソフトウェアエンジニアリングシンポジウム 2015
IPSJ/SIGSE Software Engineering Symposium (SES2015)

オブジェクト図のアニメーション

山﨑翔[†1]　久保田吉彦[†2]　紫合治[†2]

複雑化するプログラムを理解することは，プログラムの保守・変更の際に必須であり時間を要する．プログラマは理解するべきプログラムの箇所を特定し，どのように動作をするのかを想像し，プログラムを変更する．特に，オブジェクト指向プログラムにおいては，インスタンスの生成やメッセージのやり取りをイメージしてプログラムの動作を正しく理解する必要がある．本論文ではオブジェクト指向によるプログラムの動作を拡張オブジェクト図によって表現し，プログラムが動作することによって変更する状態を，図を変更させることでプログラムの理解を補助させる手法を提案する．

Object Diagram Animation

SHO YAMAZAKI[†1]　YOSHIHIKO KUBOTA[†2]
OSAMU SHIGO[†2]

Understanding complicated program is indispensable and takes the time at maintenance and alteration of program. Programmer can change his or her program by determining places of the program to understand and imagining the dynamic behavior of the program. As for Object-Oriented program in particular, programmer needs to understand the behavior of program correctly by imagining instance generation and message exchanging. In this paper, we propose methods and system to express Object-Oriented programs behavior by the movie or animation of extended object diagrams.

1. はじめに

オブジェクト指向言語は広く普及し多数のシステムがオブジェクト指向言語によって構築されている．そのように構築された多数のシステムは，システムの使用者の要請によって機能の拡張や不具合の修正が必要となる．システム変更はシステム構築者が作成したドキュメントやソースコードを読み, システム変更に必要な箇所を特定し理解する．オブジェクト指向言語によって構築されたシステムはオブジェクト間の相互作用によって動作するため, UML やソースコードのような静的な表現には現せない時間と共に変化するシステムの状態が存在する．システム変更者はクラス図などの設計文書やソースコードなど，静的な表現を元にシステムの動作を想像することに開発時間を消費する傾向がある．多数のオブジェクトを生成するような大規模なプログラムや複数のスレッドが同時に動くようなプログラムではその問題がより明確に表れる．

オブジェクト指向プログラムの理解には，プログラムの動作中に現れるオブジェクトの状態とオブジェクト間の相互作用を理解することが重要である．オブジェクトの状態を表現する統一された表現として UML のオブジェクト図を用いることはできるが，システムのある瞬間の状態を記述するオブジェクト図は静的な表現となるため，状態の変化やメッセージ送受信を観察することはできない．

システムのある状態から次の状態への変化を連続して提示することで，システムの動的な側面を表現することができる．我々は，Java 言語で記述されたプログラムを対象として，プログラム実行時のオブジェクトの生成，オブジェクトの状態の変化, 相互作用を表現する手法を提案する．表現には拡張したオブジェクト図を使用し刻々と変化するシステム内部の動作をアニメーションで提示する．

以下に第２節で本研究の関連研究について述べ，第３節では本システムで用いる拡張オブジェクト図とアニメーションについて説明する．第４節では本システムの構成について概要を紹介し，さらに第５節では実際に本システムを用いた評価結果について説明する．最後に第６節でまとめと今後の課題について述べる．

2. 関連研究

オブジェクト図を作成する方法は大きく分けて三つ，手描き，自動ツールによる静止画の生成または動画の生成である．第一の方法としては，紙と鉛筆または astah*UML[1] などのモデリングツールである（図 1）．これらは何もない状態から描ける手軽さ，自分で作図することによる図への理解の深まり，自分の好みでレイアウトやデザインをある程度変更できるという利点がある．しかし，これらの方法では自動生成ではなく自分の手で描かなければならないため一つのプログラムに対して何枚も図を用意して変化の様子を示すことは難しい．

†1 東京電機大学大学院情報環境学研究科
Graduate School of Information Environment, Tokyo Denki University
†2 東京電機大学情報環境学部
School of Information Environment, Tokyo Denki University

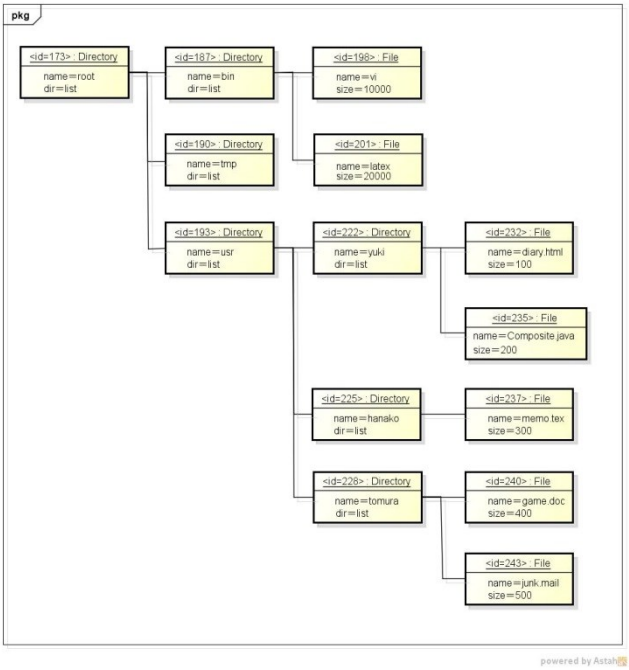

図 1　astah*UML によるオブジェクト図

第二の方法として，JIVE などの UML 自動生成ツール[2][3]を用いるという方法もある（図 2）．こちらの方法の場合，元となるプログラムから多数のオブジェクト図を短時間で生成できるという利点がある．しかし，それらを並べて見比べてみるとオブジェクトの包含関係の変更などにより着目していたオブジェクトの位置が変わって見失う，図と図の間で何が起きてどこに変化が生じたのか一目でわからないなどの問題が生じる．

第三の方法として本研究のようなアニメーションの例としては Jeliot3[4]などの研究が存在する（図 3）．

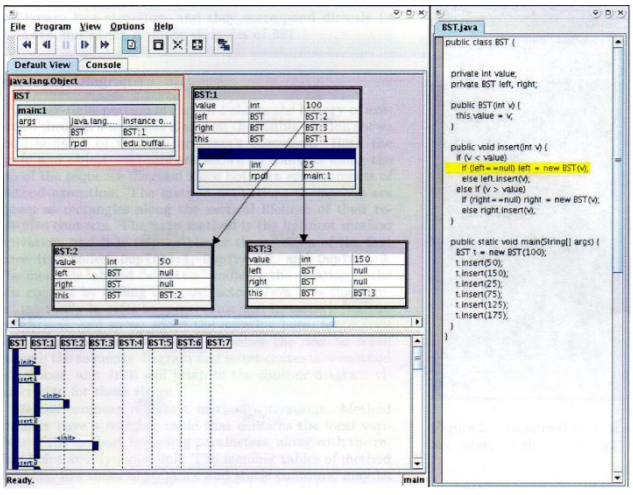

図 2　JIVE によるオブジェクト図

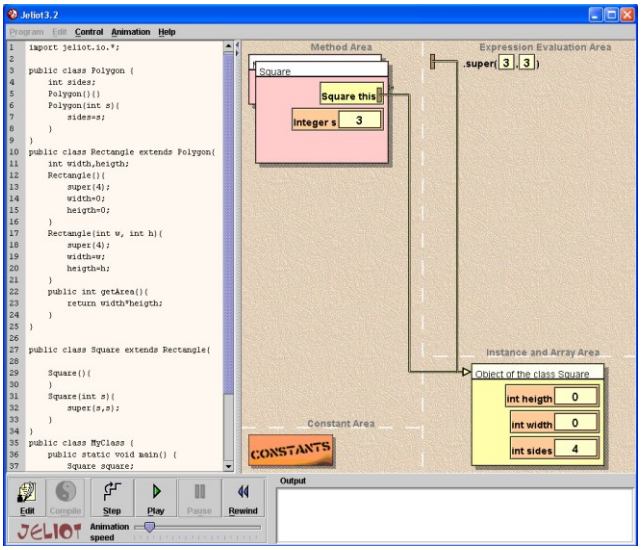

図 3　Jeliot3 による図

Jeliot3 では左上に実行中のメソッド，右上は現在実行中の演算，左下には定数やクラス変数，右下には現在生成されているオブジェクトがそれぞれ示されている．このシステムの欠点としては，メソッドがどのオブジェクトで実行されているのかはメソッド名の部分に記述されているのみであるため，分かりにくい点が挙げられる．また生成されたオブジェクトは横一列に並んでしまい，オブジェクト同士の関係が分かりにくいという欠点もある．我々が取ったのは、このうちの第三の方法である自動生成ツールによる動画の生成である．Jeliot3 の欠点に対してオブジェクトが実行しているメソッドをフィールド変数と共に表示し，オブジェクトの包含関係を矢印と図の自動レイアウトによって対処することとする．

3. オブジェクト図アニメーションの提案

3.1 拡張オブジェクト図

本システムではプログラム実行の流れに合わせてオブジェクト図が変化していく，という目的を実現するために従来のオブジェクト図へいくつかの拡張を加えた拡張オブジェクト図を規定した（図 4）．

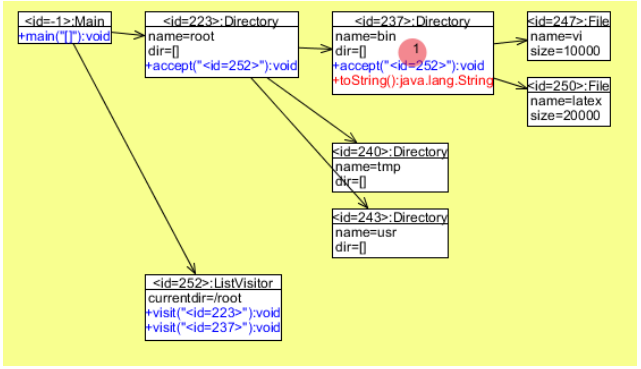

図 4　拡張オブジェクト図

以下にその変更点を示す．

- オブジェクト名の代わりに各オブジェクトが固有で持つオブジェクト ID を表示する．
- 二段目のセクションにそのオブジェクトが持つインスタンス変数だけでなくそのオブジェクトがその時点で実行中のメソッドを表示する．
- 従来のクラス図においてメソッドの引数の部分に記述されていた引数の型・名前の代わりにメソッド呼び出し時に渡された実引数を，基本データ型または文字列型の場合はその値を，それ以外のオブジェクトの場合はそのオブジェクトのオブジェクト ID を表示する．
- オブジェクト同士の関係をリンクではなく，参照側から非参照側への矢印で表現する．
- オブジェクトだけではなくスタティックなメソッドを持つクラスもオブジェクトと同様の形式で表示する．

3.2 アニメーション

本システムではオブジェクト図の変化をアニメーションとして見せるための機能が用意されている．

図 4 の<id=237>オブジェクトの中央に描かれている赤い円がある．これはあるスレッドがその時点で処理を行っているオブジェクトの位置を示すものでトークンと呼んでいる．トークンの中央に書かれている 1 という数字がスレッドを識別するスレッド ID を表している．トークンはその時点で起動しているスレッドの数だけ表示されており，最大 9 色で塗り分けている．マルチスレッドプログラムでは一つのオブジェクトのメソッドを複数のスレッドが同時に実行することがある．この時，本システムでは同じオブジェクトの上に少しずつトークンの位置をずらし，重ならないように描画している（図 5）．

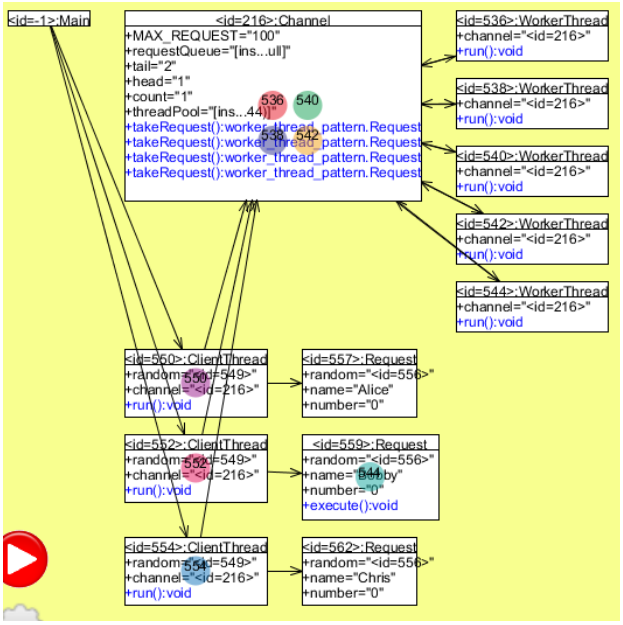

図 5　マルチスレッドにおけるトークン

また，このようなマルチスレッドプログラムが対象の際，オブジェクト図に書かれるメソッドの文字色が赤くなるのはスレッドごとに最後に実行されたメソッド，つまりすべてのトークンの位置ではなく，その中でも最後に実行されたメソッド，つまり最後に動いたトークンに対応するメソッドである．現在はこのようになっているが，今後スレッドごとに赤文字の最新メソッドが表示されるよう変更する予定である．

本システムにおいてオブジェクト図の位置関係などのレイアウトはすべて自動で行われる．他のオブジェクトを参照している側を親，参照されている側を子と呼び，オブジェクトのレイアウトは基本的に親を左，子を右とし，同じ親の子同士は横方向の位置を揃えたうえで縦に並べるツリー構造で表現されている．しかし，オブジェクト同士の関係には相互参照などツリー構造では複雑になり表現が難しい場合も考えられる．その場合には自動でもう一つのレイアウトモードであるメッシュ構造へと切り替わるようになっている．この時，相互参照は双方向の矢印で表されている（図 6）．プログラムの規模によってはオブジェクト図が画面に収まらない大きさになる可能性がある．その問題に対処するために画面内でマウスドラッグをすることで図の移動や図の横に用意してある拡大・縮小ボタンを押すことで図そのものの大きさを任意に調節できるようになっている．

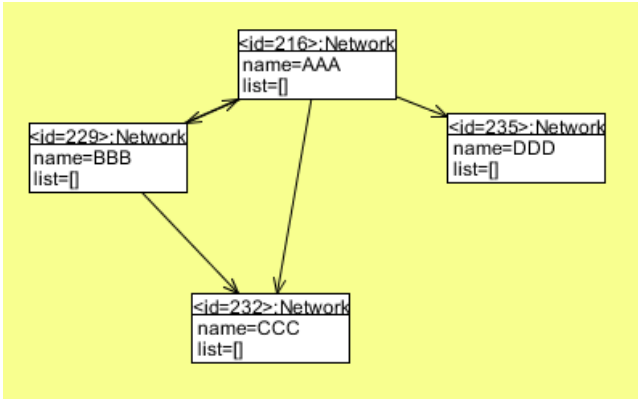

図 6　メッシュ構造の例

本システムの画面下部には再生/停止ボタンとスライダーを用意している（図 7）．ボタンはそれを押すたびにアニメーションの再生を一時停止，または再開させることができる．これによりある場面で図をじっくりと観察したいといった場面にも対応できる．もちろん一時停止中にも図の移動や拡大・縮小は可能である．スライダーはアニメーション全体の中で現在どの位置を再生しているかを表示するものである．それと同時に，これを現在位置から右へ動かすことで早送りを，左へ動かすことで巻き戻しをそれぞれ行うことができる．この機能はアニメーションの途中でオブジェクトの値がおかしいことに気付いたとき一度巻き戻して原因を探る，などの使い方を想定して用意したもので

ある．画面右にあるのはアニメーションの設定を行うためのボタンである．クラス名表示はオブジェクト図のクラス名の部分をクラス名のみ・パッケージ名.クラス名の２パターンから選べる．トークン表示は押すたびにトークンの表示非表示が切り替る．拡大・縮小ボタンで図の拡大率を，再生速度のスライダーでアニメーションの再生速度をそれぞれ調整できる．

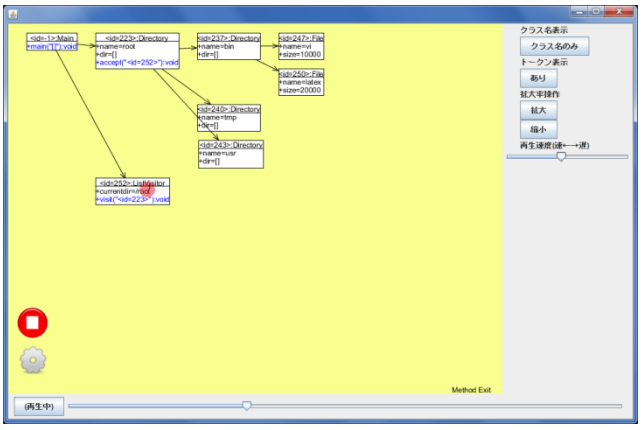

図 7　本システムの画面例

3.3　二つの方式

本システムではバッチ式とリアルタイム式という二つの起動方法を用意している．

バッチ式ではデータ抽出部で取り出したイベント情報をまとめて一度ダンプファイルへと出力する．アニメータ部ではこのファイルからイベント情報を読み出して処理を行う．この方式ではアニメーションの再生にダンプファイルを用いているのでファイルを他人に渡して，受け取った相手がそれを使ってアニメーションを見ることができる．また少しずつプログラムを書き換えて複数のダンプファイルを用意し，プログラムの変更によるオブジェクト図の変化を観察するなどの使い方もできる．

リアルタイム式では取り出したイベント情報をそのままアニメータ部へ送り，アニメータ部が処理をしていくことでプログラムの実行と同時にオブジェクト図が変化していく．よってプログラムの出力とオブジェクト図を比較して，プログラムの動きがおかしくなったときにどのプロセスでどのオブジェクトが何をしていたのかがはっきりとわかる．

4.　システムの実現

4.1　全体構成

本システムは図 8 に示すようにデータ抽出部とアニメータ部の二つに大きく分けることができる．

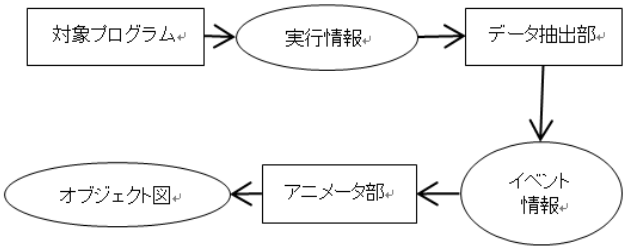

図 8　全体の構成

データ抽出部では Java 言語で記述された監視対象のプログラムから実行情報の抽出を行い，抽出された情報からさらにアニメーションに必要な情報のみを抽出・整形してイベント情報としてアニメータ部へと送る．

アニメータ部ではデータ抽出部から送られてくるイベント情報をもとに処理を行い, オブジェクト図の描画を行う．出力の際には JFrame の中に操作用のボタンやスライダーとともに Processing[5]コンポーネントを埋め込み，Processing コンポーネントの中でアニメーションの描画を行っている．

本システムではバッチ式・リアルタイム式という二種類の起動モードを用意している．バッチ式はデータ抽出部で一度イベント情報をすべてダンプファイルへと出力し，アニメータ部でそのファイルを読み込んでいく．リアルタイム式では監視対象のプログラムの実行によって発生したイベント情報をすぐにストリームでアニメータ部へと送ることで監視対象のプログラムの実行と同時にアニメーションを進めていくものである．

4.2　プログラム実行情報の抽出

Java 言語で記述されたプログラムを実行情報の監視対象とし，Java Platform Debugger Architecture（JPDA）を使用して実行情報を抽出する．監視対象のプログラムを実行している Java Virtual Machine（JVM）の Java Virtual Machine Tool Interface（JVM TI）へ Java Debugging Wire Protocol（JDWP）を通して接続し，Java Debugging Interface（JDI）から渡されてくるイベントオブジェクトからアニメーションに必要な情報のみを抽出する（図 9）．

監視対象の JVM はシステムのクラスやオブジェクトに関係するイベントも通知する為，あらかじめ監視対象の JVM にフィルタを設定し，監視対象のプログラムで定義されたクラスとそのクラスのオブジェクトに限定してイベントを通知させる．

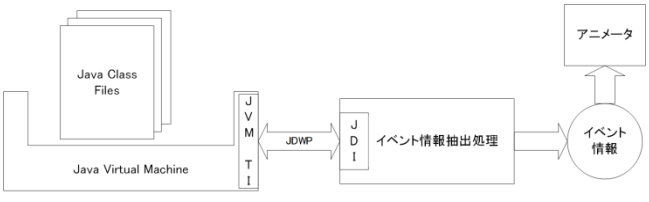

図 9　データ抽出部のシステム構成図

イベント情報抽出処理ではアニメーションに必要なメソッド呼び出し時・メソッド終了時・フィールドの値変更時の 3 つの情報をそれぞれ JDI のイベント（MethodEntryEvent，MethodExitEvent，ModificationWatchpointEvent）から文字列として抽出する．

まず各イベントからは共通して下記の情報を抽出する．

- クラス名
- メソッド名
- オブジェクト ID
- スレッド名
- スレッド ID

MethodEntryEvent，MethodExitEvent からは以下の情報を追加抽出する．

- メソッドの引数の型・名称・値
- オブジェクトのフィールド変数の型・名称・値
- メソッドの返却値の型

ModificationWatchpointEvent からは以下の情報を追加抽出する．

- 変更前/変更後のフィールド変数の型・名称・値

メソッドの引数とフィールド変数については値がプリミティブ型と String 型については型・変数名・値を取得し，監視対象のプログラムの中で定義されたクラスのオブジェクトについてはオブジェクト ID のみを取得する．変数が ArrayList または HashMap の場合には，要素を抽出し同様の処理を行う．

バッチ式でツールを実行する場合には以下で示す例のようなダンプファイルへとこれらの情報を成型し，アニメータ部へと渡す．各行の先頭にある二文字のアルファベットはイベントの識別子であり，ME は MethodEntryEvent，MX は MethodExitEvent，MW は ModificationWatchpointEvent へとそれぞれ対応している．また各情報は”#”を区切り文字として並べられている．

ダンプファイルの形式と実際の例は以下のようになっている．

MethodEntryEvent

形式

ME#パッケージ名.クラス名#オブジェクト ID#モディファイアの値#返却値の型#メソッド名#FIELD#フィールドの個数#モディファイアの値:フィールドの型:フィールド名:フィールドの値#ARGS#引数の個数#引数の型:引数名:引数の値#スレッド名#スレッド ID

例

ME#Visitor.File#286#1#void#<init>#FIELD#2#2:java.lang.String:name:null#2:int:size:"0"#ARGS#2#java.lang.String:name:"memo.tex"#int:size:"300"#main#1

MethodExitEvent

形式

MX#パッケージ名.クラス名#オブジェクト ID#モディファイアの値#返却値の型#メソッド名#FIELD#フィールドの個数#モディファイアの値:フィールドの型:フィールド名:フィールドの値#ARGS#引数の個数#引数の型:引数名:引数の値#スレッド名#スレッド ID

例

MX#Visitor.File#286#1#void#<init>#FIELD#2#2:java.lang.String:name:"memo.tex"#2:int:size:"300"#ARGS#2#java.lang.String:name:"memo.tex"#int:size:"300"#main#1

ModificationWatchpointEvent

形式

MW#オブジェクト ID#モディファイアの値#変数の型#変数名#CURRENT#現在の変数の値#NEXT#変更後の変数の値#スレッド名#スレッド ID

例：

MW#286#2#java.lang.String#name#CURRENT#null#NEXT#"memo.tex"#main#1

途中に含まれている FIELD，ARGS，CURRENT，NEXT は実際のデータではなくデータの区切りを示すサインである．モディファイアの値は public や static などの修飾子の判定に用いる

4.3 アニメーション描画

アニメータ部では大きく分けて三種類，イベントインスタンスリストの生成・イベントインスタンスの処理・レイアウト(描画)の更新に全体の処理を分けることができる．

まず，データ抽出部からイベント情報が送られてきた際にイベントインスタンスリストの生成･追加の処理を行う．その後イベントインスタンスの処理とレイアウトの更新処理を交互に行うことでアニメーションを動かしていく．これらの処理はすべて JFrame の中に埋め込んだ Processing コンポーネントの中で行われており，Swing はボタンやスライダーの操作や画面遷移にのみ用いられている．

データ抽出部から送られてきたイベント情報はまず一つ一つイベントごとに区切られ String 型の配列に格納される．そして各文字列の先頭の識別子によって MEE, MXE, MWE の三種類のイベントインスタンスを生成し，配列に格納されていた順番そのままで一つのイベントインスタンスリストへと格納される．こうすることでイベントインスタンスリストを先頭から取り出していくことでイベントの発生した順番通りに情報を追いかけていくことができる．また，MEE，MXE，MWE はそれぞれ JDI のイベント MethodEntryEvent，MethodExitEvent，ModificationWatchpointEvent に対応している（図 10）．

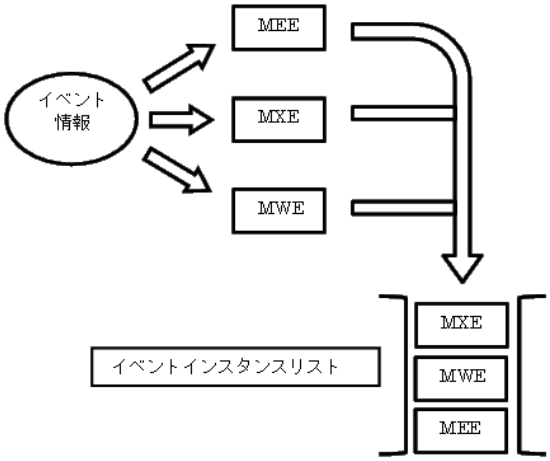

図 10　イベントインスタンスの生成

三つのクラスは共通して各情報を格納するインスタンス変数とそれらの情報を得るメソッド（例えばオブジェクト ID の情報を得るメソッドは「getId()」）と後述のイベントインスタンスの処理で用いる drawEvent()メソッドを持つ．また MEE，MXE クラスはそれらに加えてトークンの情報を更新する setToken()メソッドを持つ．

イベントインスタンス処理では図 11 のように処理が分岐する．

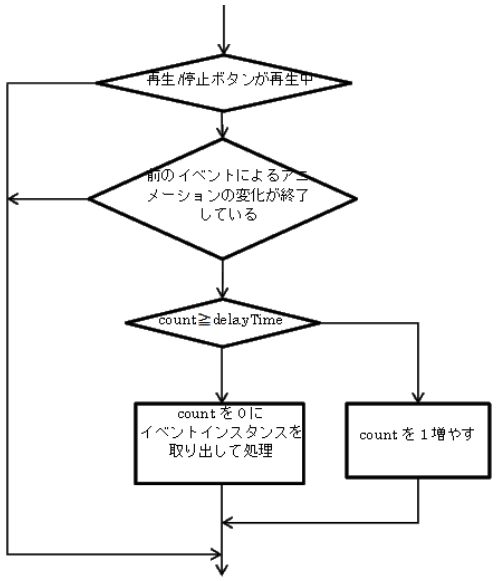

図 11　イベントインスタンス処理のフロー

MEE クラスと MXE クラスでは共通して最初にそれがコンストラクタであるかの判定を行う．コンストラクタであった場合，MEE クラスでは実行中のメソッドを格納するリストに自分を追加した後一つのオブジェクトの情報を保持する Instance オブジェクトを一つ生成し，その Instance オブジェクトリストに格納する．そして，このメソッドの前に同じスレッドで実行していたメソッドを持つ Instance オブジェクトの子に新たに生成した Instance オブジェクトを追加する．MXE クラスでは実行中メソッドのリストから自分と対になる MEE クラスのオブジェクトを探しだしてリストから削除する（図 12）．

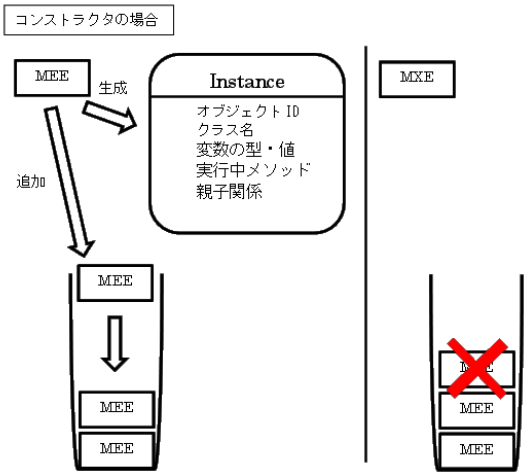

図 12　コンストラクタの場合

コンストラクタでなかった場合，MEE クラスではまず実行中メソッドのリストに自分を追加する．そして，自分が持つオブジェクト ID の値と一致するオブジェクト ID を持つ Instance オブジェクトのフィールドリストに追加する．最後にトークンの位置の更新を行う．トークンのリストからスレッド ID が一致するトークンを探し，見つかればそのトークンを，見つからなければ新しいトークンを一つ生成してフィールドリストの更新を行った Instance オブジェクトの表示位置へ移動させる（図 13）．

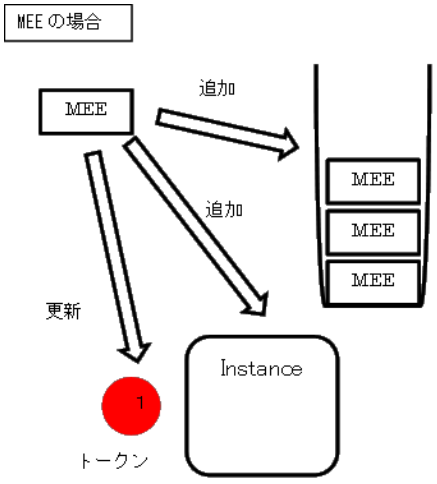

図 13　MEE の場合

MXE クラスではまず，コンストラクタであった時と同様に実行中のメソッドリストから自分の対になる MEE オブジェクトを削除し，自分が持つオブジェクトのフィールド変数の値の情報からオブジェクト ID の参照を持っているものを抜き出して，対応する Instance オブジェクトの子リ

ストの更新を行う．例としてフィールド変数の値に ArrayList 型で”[<id=180>,<id=187>]”とあった場合，一度子リストを空にしたのち新たにオブジェクト ID が１８０と１８７の Instance オブジェクトを追加する．そしてこちらでも最後にトークンの位置の更新を行う．もしそのスレッドで実行しているメソッドがもう存在しない場合はトークンのリストからトークンを削除する（図 14）．

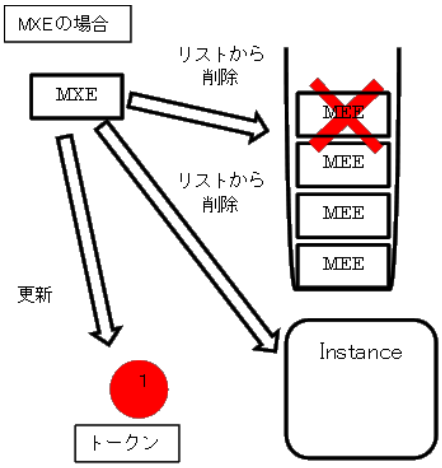

図 14　MXE の場合

MWE クラスの場合は MEE クラスや MXE クラスと同様にオブジェクト ID からフィールド変数の値の更新が行われる Instance オブジェクトを探しだし，値を新しいものへと書き換える．この値がオブジェクト ID の参照であった場合は新しいオブジェクトへ Instance オブジェクトの子リストを更新する（図 15）．

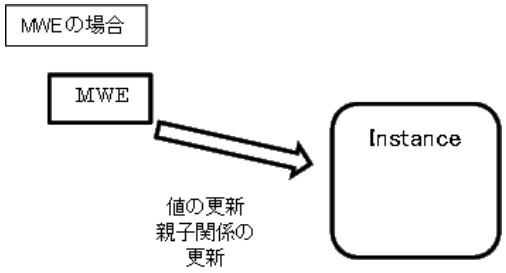

図 15　MWE の場合

レイアウト更新の処理では，各オブジェクト図やトークンの表示位置，サイズなどの更新を行ってから画面に新しい図を表示するという処理を行っている．この時，オブジェクト図やトークンの位置を少しずつ変更していき，連続的な図の動きによってアニメーションの表現を行っている（図 16）．

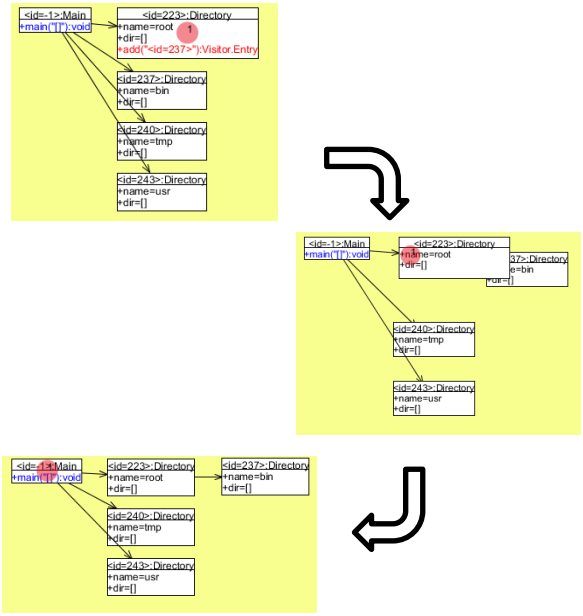

図 16　連続的な動き

上の例では<id=223>の add メソッドの実行によって Main の子であった<id=237>が<id=223>の子として<id=223>の右へ連続的に移動していく様子が描かれている．図の中間の状態では<id=237>の空間を埋めるために<id=240>，<id=243>が上へと移動している．

4.4　抽出部とアニメータ部の接続

データ抽出部とアニメータ部の接続方法は二つの起動方式でそれぞれ異なる．バッチ式では一度ダンプファイルへと情報を落としてからアニメータ部で読み込むというシステム上二つの部分は完全に独立している．一方リアルタイム式では直接情報を送る必要があるため，二つの部分を接続しなければならない．本システムにおいてはソケットを用いたプロセス間通信によって接続を行っている．データ抽出部から情報をアニメータ部側へ送ると同時に監視対象のプログラムの処理を一時停止し，アニメーションの描画が終わったら監視対象のプログラムの次の処理へ進めるよう要求する（図 17）．

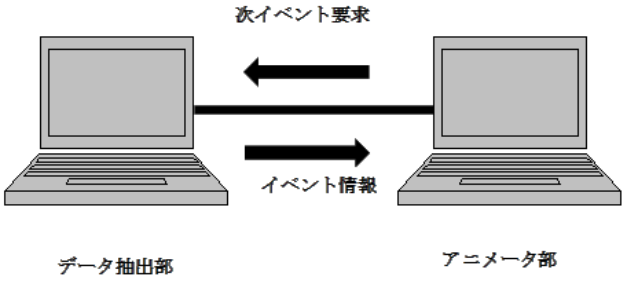

図 17　リアルタイム接続

5.　評価

本システムの評価としていくつかのプログラムを対象に本システムを使用し，その結果をまとめた（表 1）．対象の

プログラムとしてデザインパターンからシングルスレッドとマルチスレッドのものを一つずつ，Visitor パターン[6]と WorkerThread パターン[7]のサンプルプログラムを採用した．ただし WorkerThread パターンの方は記載されているプログラムではリクエストを無限に生成し続けているが，今回のテストの都合上３回生成した時点で終了するように改変してある．

表 1　結果のまとめ

	Visitor	WorkerThread
クラス数	7	5
メソッド数	19	13
イベント数	458	201
コンストラクタ	16	18
メソッド呼出	187	53
メソッド終了	187	43
フィールド値変更	52	69
オブジェクト数	17	19
最大スレッド数	1	8

コンストラクタの項目は呼び出し時の数だけ記載している．実際にはこれと同数の終了イベントが発生している．コンストラクタの数とオブジェクト数が合っていないのはどちらのサンプルプログラムでもスタティックなメソッドである main メソッドが呼び出されたことで Main クラスのオブジェクト図が生成されているためである．最大スレッド数はスレッドが同時に動いた最大の数であり，発生したすべてのスレッドの数ではない．

アニメーションを見てプログラムの動作を観察することで，ソースコードを読み込む場合と比べて，Visitor パターンの方ではトークンが階層構造になっている Directory オブジェクトや File オブジェクトと ListVisitor オブジェクトの間を往復しながらすべてのオブジェクトを回っていく（図 4）．この動きから Visitor パターンの ListVisitor オブジェクトがデータ構造を順々に回りながら処理を行っていくという流れが理解できた．

WorkerThread パターンの方では５つの WorkerThread オブジェクトそれぞれから出てきたトークンが Channel オブジェクトの上に集まって待機し，その後 ClientThread オブジェクトの子として現れた Request オブジェクトに１つトークンが移動してメソッドを実行した後また Channel オブジェクトの上に戻るという動きをした（図 5）．この動きから WorkerThread が先に起動して Channel で待機し，その後に ClientThread がリクエストを出し，それを WorkerThread が処理していくという流れがより明確に理解できた．

しかし，生成されたオブジェクトが増えていくにつれ，図全体のサイズが大きくなり最終的に画面に収まりきらなくなってしまった．現在のシステムでも図の大きさや位置を調節はできるものの，図の表示方法について改善方法を検討していく必要がある．

6. おわりに

ここでは，Java 言語で記述されたプログラムを対象としてプログラム実行時に発生するイベントからデータを抽出し，それをもとにプログラムの実行に伴うオブジェクト図の変化の流れをアニメーション的に表現するシステムについて述べた．これによってオブジェクト指向のプログラムの動作をより深く理解できた．

アニメーションの表示に関して基本的な表示と簡単な操作は行えるようになったが，デザインや操作方法の面でまだまだ改善の余地がある．また，オブジェクト図のレイアウトやスレッド・トークンの表現方法などより理解しやすい表現方法についても検討していく必要がある．

今後の課題としては

- オブジェクト図を操作することで，フィールド変数へブレークポイントを付ける機能[8]の実装．
- ソースコードの表示やスタックの状態を表示などデバッガとして本システムを使用するための機能の実装
- 本システムで表示されるアニメーションを巻き戻すことで監視対象のプログラムの動作を巻き戻す機能の実装
- プログラムの規模に応じた自動拡大率調整と，イベントの発生していないなど影響の少ない範囲での図の縮小・簡略化機能の実装

等が挙げられる．

参考文献

1) astah*UML
http://astah.change-vision.com/ja/
2) GESTWICKI, Paul; JAYARAMAN, Bharat. Methodology and architecture of JIVE. In: *Proceedings of the 2005 ACM symposium on Software visualization*. ACM, 2005. p. 95-104.
3) 中原進；紫合治．オブジェクト指向プログラムの動作の可視化．*研究報告ソフトウェア工学（SE）*，2010，2010. 9：1-8.
4) MORENO, Andrés, et al. Visualizing programs with Jeliot 3. In: *Proceedings of the working conference on Advanced visual interfaces*. ACM, 2004. p. 373-376.
5) Processing
https://processing.org/
6) 結城浩: “増補改訂版 Java 言語で学ぶデザインパターン入門,” ソフトバンククリエイティブ, 2004
7) 結城浩: “増補改訂版 Java 言語で学ぶデザインパターン入門 マルチスレッド編,” ソフトバンククリエイティブ, 2006
8) RESSIA, Jorge; BERGEL, Alexandre; NIERSTRASZ, Oscar. Object-centric debugging. In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012. p. 485-495.