ソフトウェア工学　138－19
（２００２．７．１２）

クラス間関係の簡約によるフレームワークの視覚化

清水 誠介*）丸山勝久**

*立命館大学大学院理工学研究科　　**立命館大学理工学部情報学科

概要

本論文では，フレームワーク理解のために，UML ダイアグラムを用いたフレームワークの視覚化を提案する．プログラムの視覚化が，その挙動や設計者の意図の理解に有効であることは言うまでもない．問題は，フレームワークに内在する膨大な情報をどのように扱うかである．提案手法は，フレームワークとそれを用いたプログラムとの境界に着目して重要部を特定する．利用者に必要な部位のみを表示することで，フレームワークの全体像の把握が容易となる．

Framework Visualization by Using Simplified Diagrams

Seisuke Shimizu* and Katsuhisa Maruyama**

*Graduate School of Science and Engineering, Ritsumeikan University
**Department of Computer Science, Ritsumeikan University

Abstract

The paper proposes a method that visualizes object-oriented frameworks for supporting to understand them. A framework consists of a large number of classes and contains the relationships of them. Therefore simple visualization is inadequate to understand the framework. The proposed method focuses on the boundary of a framework, and then identifies important classes necessary to learn it. Simplified diagrams excluding unnecessary classes make it easy for programmers to use frameworks.

1) はじめに

フレームワークとは再利用を目的としたアプリケーションの骨組みであり，クラス間に協調関係を持ったクラスライブラリである[1]．フレームワークを用いたソフトウェア開発では，プログラムコードの再利用だけでなく，設計，さらには，フレームワークが対象とする問題領域に対する分析や開発者の経験の再利用にもなる．また，フレームワークは統一性(Uniformity)という重要な利点を持つ．同一のフレームワークを用いることで，操作性や規約が統一され，利用者にとって使いやすいソフトウェアが開発される．このようにフレームワーク指向開発は，開発期間・開発コスト・開発人員などの削減，および，より高品質なソフトウェアの開発につながる．

しかしながら，フレームワークは，その特性から以下に示す問題点を持つ．

(1) フレームワークは一般に大規模かつ複雑である．そのため，フレームワークを用いるためには，フレームワークを習得するための長い期間と多くのコストがかかる．また，フレームワークを用いたソフトウェア開発では，その利用者がホットスポット[2]の中からカスタマイズ対象とする部位を適切に選択しなければならない．このため，利用者は，フレームワークについての十分な理解が必要である．十分な理解をせずに利用した場合，フレームワークから利益が受けられないだけでなく，変更や理解が困難なソフトウェアを生み出してしまう恐れがある．

⑵ フレームワークの設計は，繰り返しが必要である．すなわち，フレームワークは再設計の繰り返しによる進化を必要とする．再設計が行われるのであれば，同時に設計書としての UML ダイアグラムを書き直さなければならない．しかしながら，大規模かつ複雑なフレームワークに対して，設計の変更が生じる度に UML ダイアグラムを作成し直すことは非常に面倒であり，開発現場では書き直すことなくコーディング作業を行うことが少なくない．このため，UML ダイアグラムによるソースコード理解は重要であるにもかかわらず，それが完全であるとはいいきれない．

本論文では，フレームワーク利用者が容易にフレームワークを理解するための支援として，デザインリカバリ[3]に基づくフレームワークの視覚化手法を提案する．本提案手法では，フレームワークを用いて構築したアプリケーションのソースコード解析の結果，および，実行結果から，クラス図とシーケンス図を作成する．ここで，フレームワークを構成するすべてのクラスに関する情報を視覚化する手法では，フレームワークの巨大さゆえにその効果が期待できない．このため，本研究では，フレームワークとそれを用いたプログラムとの境界に存在するクラスに着目し，境界とその周辺のクラスに視覚化する範囲を制限する．ここで定義する境界部のクラスは，フレームワークのホットスポットを含むため，その挙動を理解するための重要部とみなせる．

以下に本提案手法の利点を示す．

- クラス図とシーケンス図において，利用者に必要な部位のみを視覚化し，不必要な部位を隠すことができる．その際，フレームワークの利用法に応じて重要な部位を特定する．よって，フレームワークの全体像の把握が容易になる．

- ソースコードから自動生成される UML ダイアグラムは，常にフレームワークと正当性を持つ． よって，フレームワークを変更した際に，UML ダイアグラムを変更する必要はない．さらに，すべてのフレームワークに対して，UML ダイアグラムを用いた理解支援が可能となる．

上記の利点を持つ本提案手法をシステム化し，活用することによって，フレームワーク習得のための期間とコストの削減が期待できる．

以降，2 章では，本提案手法である視覚化手法の概要について述べ，3 章でその詳細を説明する．そして，4 章でまとめと今後の課題を述べる．

2) 視覚化手法の概要

本章では，フレームワークの視覚化手法を述べ，提案する視覚化システムの概要を示す．

2.1) フレームワークの視覚化

プログラムの視覚化手法としてはさまざまなものがある．たとえば，プログラム制御やデータの流れ，データやモジュール関係などを視覚化するツールは数多く存在する．特に，文献[4]では，プログラム理解を目的とし，構造化プログラムに内在する依存関係を表示するツールが提案されている．また，文献[5]の手法では，単にプログラムそのものを視覚化するのではなく，プログラムの実行時の情報を視覚化することにより，プログラム理解支援を行う．

本論文と同じように，オブジェクト指向プログラミング言語を対象としたものには文献[6]がある．これは，プログラム内に存在するクラス間の依存度に着目した視覚化を行っている．依存関係が特に強い，まとまったクラス群を島と呼び，視覚化の対象となる要素をクラスから島にすることによって表示項目を抑えている．これは，クラス間の依存関係に基づく点が本手法と異なる．

また，Smalltalk を対象とした視覚化ツールとして ObjectOrrery[7]がある．このシステムは，クラス間の依存関係，メソッドとインスタンス変数間の依存関係，メソッド間の依存関係，実行時のインスタンスの振る舞いに着目し，それらを視覚化する．さらに，本論文と同じようにオブジェクト指向プログラムのソースコードを解析し，UML ダイアグラムを作成 (抽出) するツールとして JBuilder[8]が有名である．このツールでは，ソースコードからクラス図を作成可能であるが，着目する 1 つのクラスに関連のあるクラス群だけが表示される．よって着目したクラス単体での理解に適しているが，プログラムの全体像の把握には向いていない．

本提案手法では，フレームワーク理解において重要であると考えられるフレームワークの境界部に着目する．フレームワークとそれを用いたプログラムとの境界に存在するクラスに基づき，視覚化の対象となるクラスを決定する．さらに，フレームワーク理解においては，単にクラス間の関係だけでなく，メッセージの流れが重要であると考え，制限された

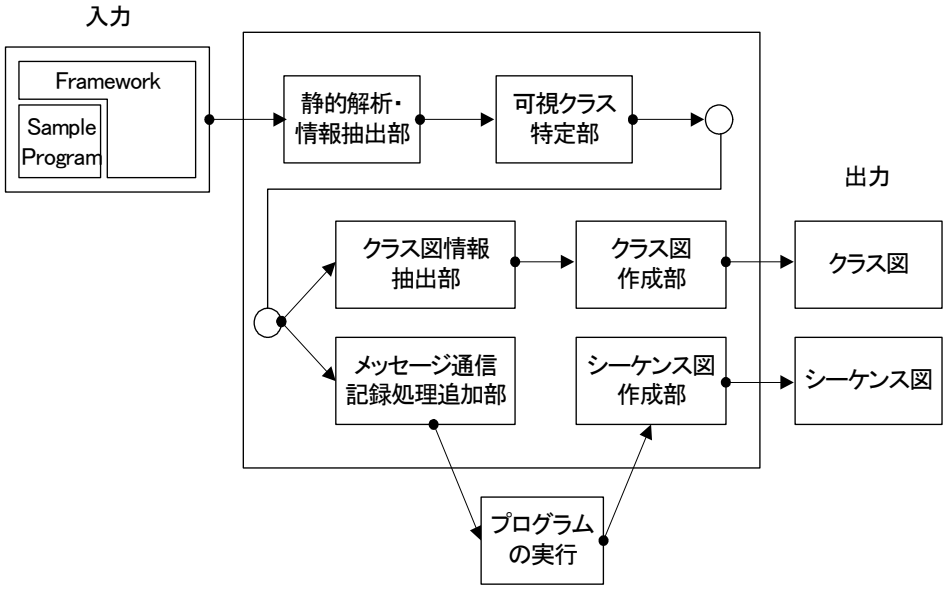

図 1) システムの概要

クラスに対するシーケンス図も作成する．このように本手法は，視覚化の際に制限されたクラス図とシーケンス図を作成することが特徴であり，その制限法が従来の手法と大きく異なる．

2.2) 視覚化システム

本論文で提案するシステムの概要を図 1 に示す．以下に，各部の説明を述べる．)

a) 入力

フレームワーク自身とそれを用いたサンプルプログラムのソースコードを入力とする．サンプルプログラムとは，利用者がフレームワークを用いて構築中のプログラム，あるいは，フレームワーク学習のために用意された例題プログラムを指す．ただし，これにはフレームワークを含まない．

一般的に，フレームワーク利用者がアプリケーションを構築する際，フレームワークの内部すべてを理解する必要はない．本手法では，サンプルプログラムを入力として用いることで，利用者にとって現在関心のある部分，必要となる部分だけを重要部として特定し，理解支援を行うことができる．

また，対象とするオブジェクト指向プログラミング言語として Java[9]を扱う．以降，視覚化の手順やオブジェクト指向技術における決まり事は，全て Java の規則に従うものとする．

b) 出力

本視覚化システムは，UML[10]におけるクラス図とシーケンス図を作成する．クラス図は，フレームワークの静的な構造を把握するのに有効である．シーケンス図は，1 つのユースケースにおけるオブジェクト群の振る舞いを，時間軸を基準として把握するのに役立つ．提案手法における図は，一般的な UML ダイアグラムを一部拡張したものであり，詳細は 3.2 節と 3.3 節において述べる．

c) 静的解析・情報抽出部

入力として与えられたフレームワークとサンプルプログラムから構築された，ひとつの完成されたアプリケーションのソースコードを静的に解析する．全てのクラスに対して静的解析を行うことにより，視覚化に必要な情報を抽出する．解析器の実装には，JavaCC (Java Compiler Compiler) [11]を用いる．

d) 可視クラス特定部

静的解析・情報抽出部の結果として得られた情報から可視クラスを特定する．可視クラスとは，フレームワーク理解の際に重要なクラスのことであり，3.2 節で説明する．本視覚化システムでは，この可視クラスに関する情報のみを表示する．

e) クラス図情報抽出部

特定された可視クラスを用いることにより，クラス図に必要となる情報を選定する．具体的には，可視クラスの内部情報(クラス名，属性，メソッド)，可視クラス間の関係（継承，関連）を抽出する．

f)　クラス図作成部

クラス図情報抽出部において抽出した情報に基づき，クラス図を描画する．

)

g)　メッセージ通信記録処理追加部

本手法では，シーケンス図を作成するために，ソースコードの静的解析と実行時のメッセージ記録を組み合わせる方法を採用する．メッセージ通信記録処理追加部では，フレームワークおよびサンプルプログラムにおいて生成されるオブジェクト間のメッセージ通信（メソッド呼び出し）を記録するための処理を入力ソースコード中に埋め込む．その際，可視クラスとそれ以外のクラスとを区別する．

h)　シーケンス図作成部

記録コードが埋め込まれたプログラムを実行することにより，メッセージ通信を記録する．記録されたメッセージに基づき，可視クラスに関係するオブジェクト群に対するシーケンス図を描画する．

3)　視覚化手法

本章では，フレームワーク理解において重要部だけを特定するための新しい概念として，境界クラスおよび可視クラスを提案する．さらに，これらのクラスに基づき作成されるクラス図とシーケンス図を示す．

3.1)　ソースコードの静的解析

)本手法において，クラス図を作成するためには，ソースコードからクラスおよびクラス間関係を抽出する必要がある．抽出する情報を次に列挙する．

a)　クラス（クラス名）
b)　クラス内のメソッド
c)　クラス内のフィールド
d)　継承（extends，implements）
e)　メソッド呼び出し

本論文では，クラスＡがクラスＢを継承している場合，$A \xrightarrow{I} B$ と表現する．また，クラスＡ内のコードがクラスＢ内のメソッドを呼び出している場合，ＡがＢを参照していると呼び，$A \xrightarrow{M} B$ と表現する．これらは，アクセス範囲（修飾子）を考慮して，継承先や参照先クラスを決定する．

いま，図２のクラス図において，Client クラス（インスタンス）が Windows 型で宣言された多相変数

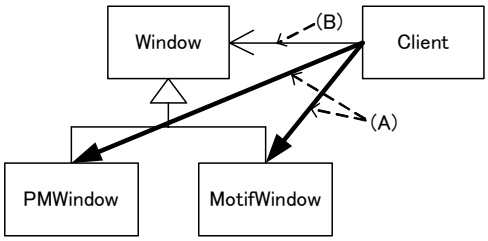

図２)　多相性を含む参照

（多態変数）を介して PMWindow，MotifWindow クラス（インスタンス）を参照する可能性があるとする（図２の(A)）．このように解析対象のソースコードが多相性を含む場合，静的解析では実際にインスタンス化されているクラスを厳密に特定することが困難である．そこで，本手法では，インスタンスの動的な型を無視し，宣言された静的な型に基づき参照先インスタンスのクラスを決定する．たとえば，図２において，Client の呼び出し先インスタンスが PMWindow や MotifWindow であっても，そのインスタンスの型が Window として宣言されている場合は，Client の参照先を Window とみなす(図２の(B))．

3.2)　クラス図の構築

フレームワーク全体を単純に視覚化するのでは，作成された図が複雑になりすぎ，フレームワーク理解が困難である．そこで，本論文では，フレームワーク内に存在するすべてのクラスを表示するのではなく，利用者の理解に有用なクラスだけを表示するために，境界クラスを定義する．さらに，境界クラスとの関係から，視覚化の際に表示対象となる可視クラスと，表示対象としない不可視クラスを定義する．可視クラスと可視クラス間の関係を含むクラス図が本提案システムの出力となる．

3.2.1)　境界クラス

境界クラスとは，対象フレームワーク内において，サンプルプログラムのクラスと直接関係を持つクラスのことである．いいかえると，フレームワーク内にあり，フレームワークの外との境界にあるクラスのことである．境界クラスの定義を以下に示す．

境界クラス*(BC: Border Class)*

$$SP \xrightarrow{I,M} BC$$

*SP(Sample Program)*は，入力となるサンプルプログラムに含まれるクラスを指す．$\xrightarrow{I,M}$ は，継承(I)およびメソッド呼び出し(M)を表す．

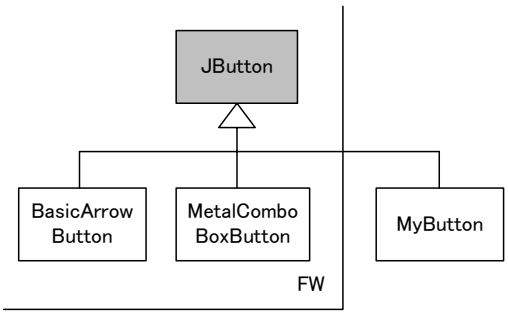

図3) 境界クラス

図3に境界クラスの例を示す．FWの文字がある側がフレームワーク内部，MyButtonがサンプルプログラムのクラスである．MyButtonと直接継承関係を持つJButtonクラス（灰色に塗られたクラス）が境界クラスである．図3では継承の例を示したが，メソッド呼び出しでもよい．

フレームワーク内で，フレームワーク外のクラスと直に関係を持つクラスは境界クラスのみである．よって，境界クラスはフレームワーク内のクラスの中で最も外側にあるクラスといえる．また，境界クラスは，境界に存在するという性質上，フレームワークの可変部分（ホットスポット）とみなせることが多い．そのため，境界クラスの構造や挙動を理解することは，フレームワーク理解に大きく貢献する．

3.2.2) 可視クラス

境界クラスは，フレームワーク理解にとって非常に重要なクラスである．しかしながら，境界クラス以外にも，フレームワークを理解する上で必要となるクラスが存在する．本手法では，利用者に対して表示するクラスを可視クラスと呼び，可視クラスを次のように定義する．

可視クラス*(VC: Visible Class)*とは，次の(a)～(b)のいずれかのクラスである．

a) サンプルプログラムのクラス*(SP)*
b) 境界クラス*(BC)*
c) *BC*の祖先クラス*(AC: Ancestor Class)*

$$BC \xrightarrow{I}{}^{*} AC$$

d) *BC*の子孫クラス*(DC: Descendant Class)*

$$DC \xrightarrow{I}{}^{*} BC$$ （インタフェースは除く）

ただし，$\xrightarrow{I}{}^{*}$は，継承関係の反射的推移包含を表す．また，フレームワークに含まれる*VC*以外のクラスを不可視クラス*(Invisible Class)*と呼ぶ．

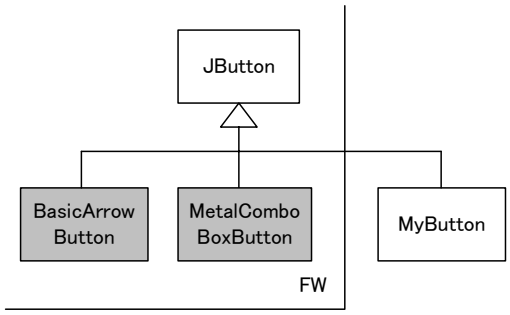

図4) 子孫クラス

a),b)に関しては，フレームワークとサンプルプログラムとの協調関係を見るために必須である．c)の*AC*は，境界クラスのソースコードには直接現れない継承された項目（メソッド，フィールド）との関係を把握するために必要となる．d) の*DC*は，2つの理由から必要である．1つは，*DC*が*BC*を利用する際のサンプルになる可能性をもつためである．これは，サンプルプログラムと同様にフレームワークを利用する際の参考になる．もう1つは，多相性が実現されているとき，本手法では参照先が型宣言での静的な型になるためである．この場合，*DC*が動的な参照先となり，*DC*がフレームワーク理解の参考になる．

図4に*DC*の例を示す．灰色に塗られたクラスが*DC*である．また，図4では，すべてのクラスが可視クラスとなる．

3.2.3) 出力となるクラス図

静的解析・情報抽出部での解析結果，可視クラス特定部での可視クラスを用いて，クラス図を作成する．本手法で表示対象となるクラスは可視クラスのみである．可視クラス間に継承と参照の関係が同時に存在する場合，継承関係のみを表示する．また，本手法では，クラス間関係の視覚化を目的とするため，クラス内部でのメソッド呼び出しは表示しない．

ここで，可視クラスだけを表示し，可視クラス間に含まれる不可視クラスをすべて表示対象外にした場合，可視クラス間の間接的な関係が分からなくなる． たとえば，図5のような場合を考える．図5において灰色のクラスが可視クラスであり，Panelクラスは不可視クラスである．可視クラスのみが表示対象であるため，視覚化の際には，Panelクラスを図から削除しなければならない．しかしながら，単に削除しただけでは，Buttonクラスと Scrollbar クラス間における間接的なつながりが分からなくなる．そこで，本手法では，図6のように，可視クラス Button と Scrollbar を点線（以降，間接線と呼ぶ）で結ぶことによっ

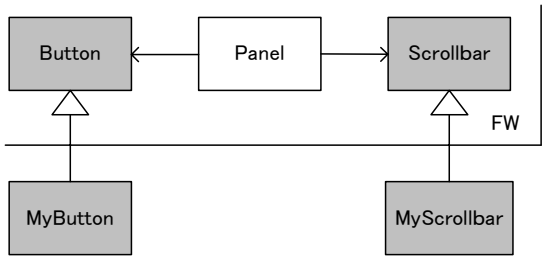

図5) 可視クラス

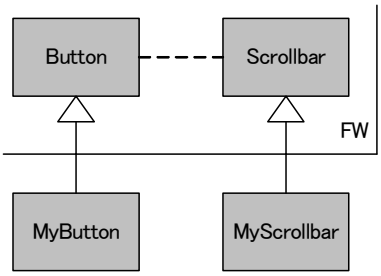

図6) 可視クラスと間接線

て間接的なつながりを示すようにする．ただし，間接線については，継承と参照を区別しない．

3.3) シーケンス図の構築

3.2 節で定義した可視クラスを用いることでシーケンス図を簡約する．本提案手法において出力となるシーケンス図は，特定のユースケースにおいてオブジェクト（インスタンス）間の協調を把握するために有効であり，フレームワーク理解支援に役立つ．

本手法では，シーケンス図作成のために，静的なソースコード解析と動的な（実行時の）メッセージ記録を組み合わせる方法を採用する．一般的に，単純な静的手法のみでは，特定のユースケースに関連するインスタンスを特定することが難しい．反対に，単純な動的手法のみでは，インスタンス間のメッセージ送信をすべて記録しておかなければならず，実行時のオーバーヘッドが大きくなる．

本手法では，あらかじめソースコードを解析しておき，フレームワーク支援に重要であると考えられるクラスのメッセージ通信のみに記録対象を絞り込む．メッセージ通信の記録は，実際にプログラムを実行することによって行う．これらにより，記録するメッセージ通信の数やオーバーヘッドを減少させ，かつ，メッセージ通信に関わるインスタンスの特定を行うことが可能となる．

3.3.1) メッセージ通信記録コードの追加

本手法において，シーケンス図を作成するために抽出する情報を次に列挙する．

a) メッセージ送信元インスタンス名
b) メッセージ送信先インスタンス名
c) メッセージ名

シーケンス図作成のために必要な情報として，上記(a) (b) (c)以外にも，インスタンスの活性区間，メッセージ送信の条件や反復，メッセージの種類などが存在するが，本手法は完全なシーケンス図を構築することが目的ではないため，これらを抽出しない．

本手法では，インスタンス間のメッセージ通信をそのインスタンスに対するメソッド呼び出しで捉え，(a) (b) (c)の情報を抽出するために，実行時のメッセージ呼び出しを記録するコードをフレームワークとサンプルプログラムのソースコードに埋め込む．

可視クラスに関連するメッセージ通信のみを記録するコードを埋め込む手順を以下に示す．

(1) 可視クラス集合を VC とおく．フレームワークあるいはサンプルプログラム内のクラス VC1 が VC に含まれる(VC1 ∈ VC)とき，VC1 内の各メソッド m に対して，(2) ～ (7) の操作を繰り返す．図 7 に記録コードを追加する様子を示す．
(2) m 内において，メソッド呼び出し先インスタンスの型をクラス C とする．C が可視クラスの場合(C ∈ VC)，メソッド呼び出しの引数に this を追加し，呼び出し元のインスタンスを送るようにする（図 7 の(A)）．
(3) C が不可視クラスの場合(C ∉ VC)は，メッセージを実際に記録する Record.recordMessage メソッドを呼び出すコードを追加する（図 7 の(B)）．引数 this はメッセージ送信元インスタンス，文字列 ”C2@invisible” はメッセージ送信先クラス名（invisible は不可視クラスを表す），”m2” はメソッド名（すなわち，メッセージ名）をそれぞれ表している．
(4) m とシグニチャが同じで，中身が空のメソッド m’を作成する（コード中のメソッド名は m）．
(5) m’のシグニチャに Object 型の引数を追加する（図 7 の(C)）．
(6) Record.recordMessage メソッドを m’に追加する（図 7 の(D)）．引数 src はメッセージ送信元インスタンス（メソッド呼び出し元での this に相当），this はメッセージ送信先インスタンス，”m” はメソッド名をそれぞれ表している．

```
public class VC1 {
  private C1 o1;   // Visible Class
  private C2 o2;   // Invisible Class
  public void m(int num, String str) {
    ....
    o1.m1(str);
    o2.m2(str);
    ....
  }
  ....
}
```

```
public class VC1 {
  private C1 o1;   // Visible Class
  private C2 o2;   // Invisible Class
  public void m(int num, String str) {
    ....
    o1.m1(str, this);<---- (A)
    o2.m2(str);
    Record.recordMessage(this, "C2@invisible", "m2");   <---- (B)
    ....
  }
  public void m(int num, String str, Object src) {   <---- (C)
    Record.recordMessage(src, this, "m");
    m(num, str); <---- (E)
  }
  ....
}                                                   <---- (D)
```

図 7 記録コードの追加

(7) 元の処理を呼び出すコードを m'に追加する（図 7 の(E)）.

ここで，(2) (3) (6)において，クラスメソッドでは this を使用できない．この場合，this の代わりに文字列 "*ClassName*@static" を用いる．*ClassName* は，呼び出し元メソッドが属するクラス名を指す.

3.3.2) メッセージ通信の記録

3.3.1 で記録コードを埋め込んだプログラムを実行することにより，メッセージ通信の記録を行う.

```
User@1f2a0b → Calculator@13fac (execute)
User@1f2a0b → Calculator@13fac (print)
Calculator@13fac → PrintSpooler@static (print)
PrintSpooler → Printer@invisible (start)
```

図 8 記録結果

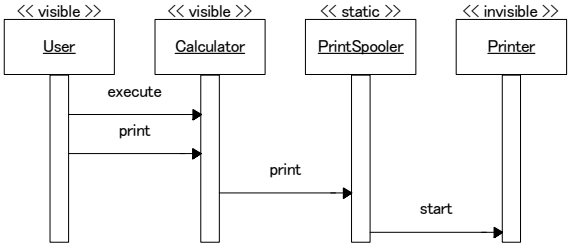

図 9 シーケンス図の例

このとき，記録されるメッセージは，プログラムの実行結果に依存する．記録されるメッセージの例を図 8 に示す.

3.3.3) 出力となるシーケンス図

記録されたメッセージに基づき，シーケンス図を作成する．図 8 から作成したシーケンス図を図 9 に示す.

3.4) 適用例

フレームワークとして, JDK の awt パッケージ等を用い，サンプルプログラムとして，MyApplet クラスと MyListener クラスを用意した．プログラムは，アプレット上でマウスを動かすと，イベントを発生させるような簡単なものである.

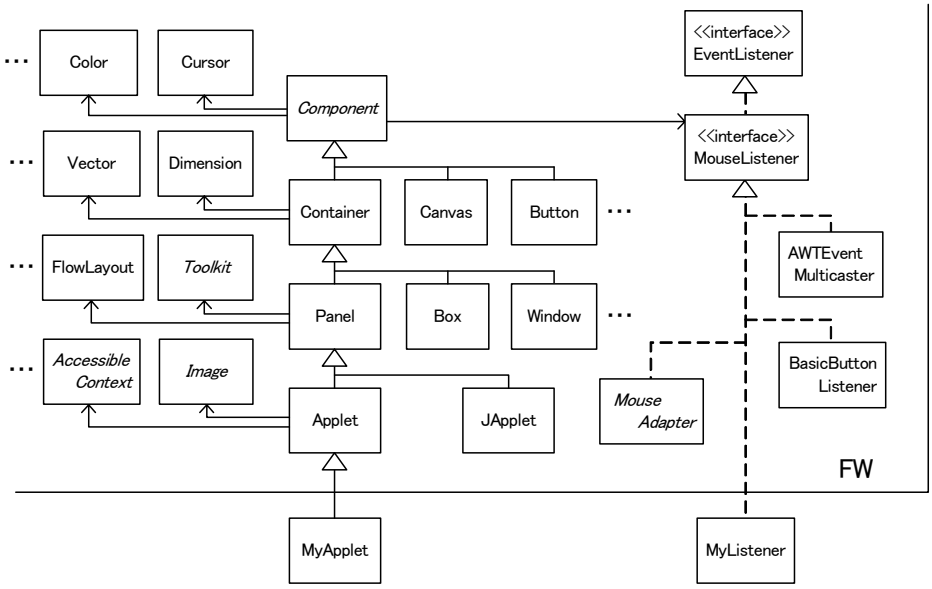

図 10 フレームワークとサンプルプログラム（一部省略）

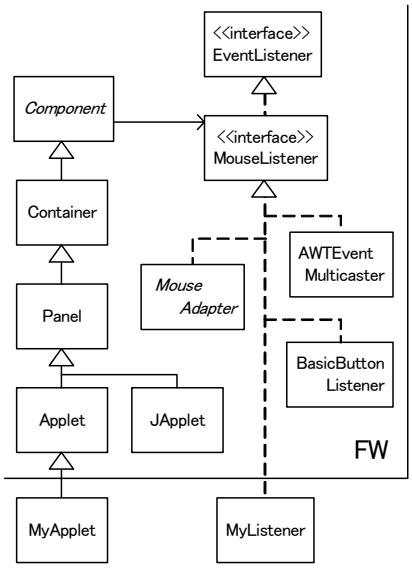

図 11　可視クラスで構成されたクラス図

サンプルプログラムに関係を持つクラス群を図 10 に示す．本提案手法を適用した結果を図 11 に示す．ただし，これらの図一部を省略している．

図 11 では，不可視クラスが削除され，可視クラスのみが表示されている．このため，見やすさが向上しているといえる．シーケンス図に関しても同様に，可視クラスに関連するメッセージのみが表示される．

4) おわりに

本論文では，フレームワークを理解するために重要なクラスとして，境界に着目し，可視クラスを定義した．さらに，クラス図とシーケンス図において，可視クラスに関わる部分のみを視覚化する手法を述べた．表示対象を適切に制限することで，フレームワーク全体像の把握が容易になる．

本手法では，入力にサンプルプログラムが必須である．このため，本研究では，利用者が開発中のアプリケーションをサンプルプログラムとみなすこととした．これによって，例題プログラムが用意されていないフレームワークについても，その利用法に応じた理解支援ができる．しかしながら，本手法により表示される可視クラスはサンプルプログラムに大きく依存する．よって，フレームワーク理解を効率的に行うには，入力となるサンプルプログラムを適切に選択する必要がある．選択方法に関しては，現在検討中である．

また，本手法では，境界クラスを最重要クラスと考え，境界クラスとの関係により可視クラスを特定した．提案した可視クラスを用いたクラス図およびシーケンス図が，フレームワーク理解において，真に有効であるのかどうかを評価する必要がある．このためには，数多くの適用実験が必須である．現在，評価実験のために，視覚化システムの実装を進めている．

参考文献

[1] Mohamed E.Fayad，Douglas C.Schmidt，”Object-Oriented Application Frameworks”，Vol.40，No.10，Communications of the ACM，pp.32-38，1997

[2] Wolfgang Pree，”デザインパターンプログラミング”，トッパン，佐藤啓太，金澤典子 訳，1996

[3] Ted J. Biggerstaff，“Design Recovery for Maintenance and Reuse”，IEEE Computer，pp.36-49，July 1989

[4] 大崎博基，山本晋一郎，阿草清滋，“プログラム理解のための依存関係表示ツール”，日本ソフトウェア科学会 FOSE’96，pp.34-41，1996

[5] 鈴木宏紀，山本晋一郎，阿草清滋，“プログラム実行情報の視覚化による理解支援ツール”，情報処理学会研究報告，vol.98，No.64，pp.77-84，1998

[6] 三ツ井欽一，中村宏明，“オブジェクト指向プログラムの理解のための視覚化技法”，情報処理学会研究報告，vol.94，No.55，pp129-136，1994

[7] Software Research Associates，”ObjectOrrery”，http://osb.sra.co.jp/Smalltalk/ObjectOrrery/

[8] Borland Software Corporation，”JBuilder”，http://www.borland.co.jp/jbuilder/

[9] James Gosling，Bill Joy，Guy Steele，Gilad Bracha，“Java 言語仕様第 2 版”，Pearson Education，村上雅章訳，2000

[10] Grady Booch，James Rumbaugh，Ivar Jacobson，”UML リファレンスマニュアル”，Pearson Education，石塚圭樹 監訳，2002

[11] Metamata and SunMicroSystems，JavaCC，http://www.webgain.com/products/java_cc/