Vol.2018-CE-143 No.7
2018/2/17

オブジェクト指向学習支援のためのメンバの可視性の視覚化

恒川 廉　山田 俊行

概要：オブジェクト指向プログラミングの学習において，継承・カプセル化・ポリモーフィズムの概念の理解は重要である．本研究では，プログラムの各ステップごとにオブジェクトを視覚化し，プログラムの動作の理解を助けることで初学者に対するオブジェクト指向プログラミングにおける基本概念の学習を支援する手法を提案する．オブジェクトと参照型変数の参照関係と，クラスの継承関係と，インスタンス変数とメソッドの可視性をステップ毎に視覚化する．

キーワード：オブジェクト指向プログラミング，視覚化，可視性，初学者支援，学習支援

Visualizing Visibility of Members for Learning Object-Orientation

Ren Tsunekawa　Toshiyuki Yamada

Abstract: For learning Object-Oriented programming, understanding inheritance, encapsulation and polymorphism is important. In this study, we propose a method for support for novices to learn Object-Oriented programming by visualizing objects of a program stepwise to help their understanding program behavior. We visualize reference relationships between objects and reference variables, inheritance relationships between classes and visivility of instance variable and method in every step of program.

Keywords: object oriented programming, visualization, visibility, novice support, learning support.

1. 研究背景

オブジェクト指向プログラミングの学習において，継承・カプセル化・ポリモーフィズムなどの概念の理解は重要である．しかし，プログラミング初学者にとって，これらの概念を理解することは難しい．その主な要因として，ソースコードを読むだけではそれらの動的な振る舞いがわからないことが挙げられる．例えば，オブジェクト指向プログラミングではカプセル化を用いて情報隠蔽が行われるが，情報隠蔽を理解するためには，オブジェクトの持つインスタンス変数・メソッドへのアクセスの可否が，着目しているオブジェクトごとに変化する様子を理解する必要がある．ポリモーフィズムの理解には，着目する参照型変数がプログラム実行時にどのクラスのオブジェクトを参照するかを理解する必要がある．これらを理解するには，ソースコードからプログラムの動作を予測しなければならない．しかし，初学者にとって，ソースコードからプログラムの動作を正確に予測することは困難である．

本研究では，プログラム実行時の情報を用いて，参照型変数とオブジェクトの参照関係と，オブジェクトの持つフィールド・メソッドの可視性を，プログラムのステップ毎に視覚化することで，初学者に対するオブジェクト指向プログラミング学習を支援する手法を提案する．

2. 関連研究

Jeliot 3 [1] は Java で書かれたプログラムを 1 ステップずつ実行し，その様子をアニメーションを用いて視覚化するツールである．参照型変数とオブジェクトの参照関係を視覚化し，親クラスのインスタンス変数を子クラスに示すことで継承を視覚化する．これらの視覚化を通じてオブジェクト指向プログラミングの学習を支援している．しかし，Jeliot 3 で表示される図では，どのインスタンス変数が親クラスから継承したのか図からわからない．また，インスタンス変数やメソッドの可視性を表示する機能がない

ことも問題点である．

3D プログラミング環境である Alice[3] の目的は，仮想世界の中で 3D モデルのオブジェクトの追加・削除・移動などの操作をプログラミングすることでオブジェクト指向プログラミングの基礎を学習することである．プログラムを実行することで 3D モデルのアニメーションを動かし，自身の書いたプログラムがどのように動作するか確認できる．オブジェクトの高さや重さなどのパラメータへはメソッドを通じてのみアクセスできるように制限されており，カプセル化や情報隠蔽の概念も学習できる．しかし，ポリモーフィズムへのサポートがないことや，仮想世界でオブジェクトを操作する以外のことができないことが問題点である．

プログラムを図で表す方法として UML[9] がある．クラス図・オブジェクト図・シーケンス図などの複数の図を用いてプログラムを表現する．これらの図を読むことで，ソースコードを読まなくてもプログラムの概要を把握できる．しかし，初学者にとって，UML の図をすべて正確に読み取ることは困難である．また，初学者が UML を習得するには学習に時間がかかるため，初学者がオブジェクト指向プログラミングを学習する際に UML を用いるのは効率的ではない．

3. 提案手法

本研究では以下の３つの視覚化によりでオブジェクト指向プログラミングの学習を支援する手法を提案する．

- プログラムの各ステップごとにオブジェクトを視覚化
- オブジェクトの持つインスタンス変数とメソッドを図示
- メンバの可視性の変化を視覚化

プログラムをステップ実行し，生成されたオブジェクトとそのメンバの可視性を１ステップごとに図示することで，メンバの可視性の変化を視覚化する．これらの視覚化により，オブジェクト指向における継承・カプセル化・ポリモーフィズムの概念の学習を支援する．

3.1 オブジェクトと参照型変数の視覚化

本研究では，プログラム実行中に生成されるオブジェクトとそれを参照する参照型変数を視覚化するために，以下の情報を用いる．

- オブジェクトを参照する参照型変数の型と識別子
- オブジェクトを生成するクラスのクラス名とメンバ
- オブジェクトの持つインスタンス変数・メソッドの可視性
- 継承しているクラスとそのメンバ

プログラム実行中に生成されたオブジェクトと，そのオブジェクトを参照している参照型変数を図示する方法を説明する．

図 1 参照型変数の図示
Fig. 1 Displaying Deference Variable

図 2 オブジェクトの図示
Fig. 2 Displaying Object

3.1.1 参照型変数の図示

オブジェクトを参照する参照型変数は，左右２つに区切られた四角で表現する．左部にその変数の型であるクラス名を，右部に変数の識別子を書く．参照型変数の表し方を図１に示す．参照型変数のみ左部の四角の背景色を変えることで，参照型変数であることを強調する．

この図示により，オブジェクトを参照している変数の型がソースコードを見なくても図からわかる．

3.1.2 オブジェクトの図示

オブジェクトは上部にラベルの付いた四角で表現する．ラベルにはそのオブジェクトのクラス名を示す．オブジェクトを表現する四角の内部にそのオブジェクトが持つインスタンス変数と，メソッドを示す．インスタンス変数は左右２つに区切られた四角で表現し，左部に型を，右部に変数名を書いて表現する．メソッドも同様に左右２つに区切られた四角で表現し，左部にメソッドの返り値の型を，右部にメソッド名を書いて表現する．オブジェクトの表し方を図２に示す．

このようにオブジェクトを視覚化すれば，生成されたオブジェクトがどんなインスタンス変数とメソッドを持っているか図からわかる．

3.1.3 参照関係の図示

参照型変数とオブジェクトの参照関係を矢印を用いて図示する．参照型変数から，その変数が参照しているオブジェクトに向けて矢印を書く．参照関係の表し方を図３に示す．

このように参照関係を図示すれば，各参照型変数がどのクラスのオブジェクトを参照しているかが図からわかる．

3.2 継承の図示

オブジェクト指向の特徴である継承を図示する．3.1.2 節で述べたのオブジェクトの図示に加え，親クラスのオブジェクトの図を子クラスのオブジェクト図に入れ子にして

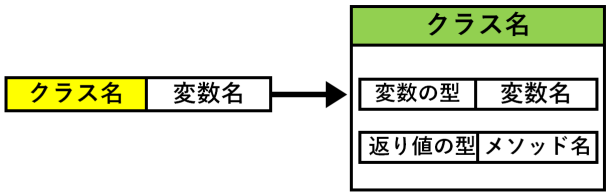

図 3 参照関係の図示
Fig. 3 Displaying Reference

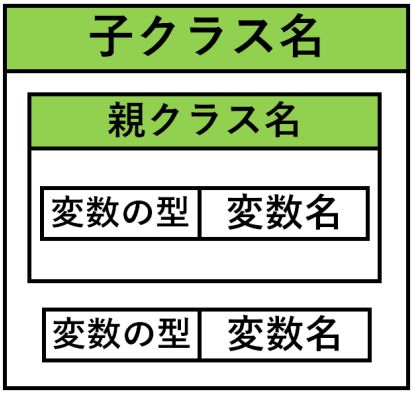

図 4 継承関係の図示
Fig. 4 Displaying Inheritance

示すことでクラス間の継承関係を表す．図 4 に継承関係の図示の方法を示す．

親クラスの持つメンバは親クラスの図中に示し，子クラスの持つメンバは子クラスの図中に示す．そうすることで，どのメンバを親クラスから継承しているかが図からわかる．

また，サブクラスがスーパークラスのメソッドをオーバーライドしている場合は，スーパークラスのメソッドを表している図の背景色を暗くする．例として，Human クラスを継承した Student クラスを考える．このとき，Human クラスの返り値を持たず，引数を取らない Hello というメソッドを Student クラスがオーバーライドしている場合の図を図 5 に示す．

このように図示すれば，どのメソッドがオーバーライドされているか図からわかる．

3.3 可視性の視覚化

オブジェクト指向プログラミングでは，インスタンス変数やメソッドにアクセス制限を設けることで情報隠蔽を行う場合がある．その場合，現在着目するオブジェクトからアクセスできないメンバは図示しないようにすることで可視性を視覚化する．

private なインスタンス変数 name と，public なメソッド hello を持つ Human クラスを考える．

```
1:  class Human {
2:    private String name;
```

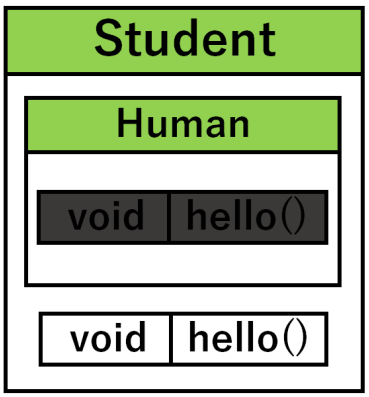

図 5 オーバーライドの図示
Fig. 5 Displaying Overwride

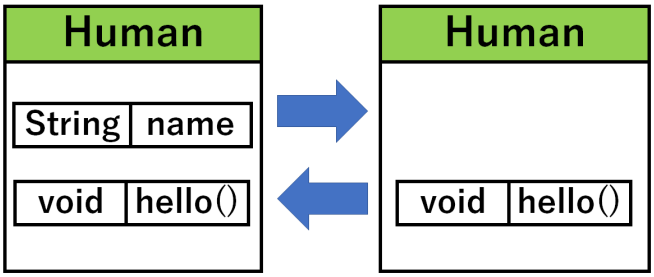

図 6 可視性の視覚化
Fig. 6 Displaying Visibility

```
3:    public void hello() {
4:      System.out.println("I'm a Human.");
5:    }
6:  }
```

この場合，name は private なので他クラスからはこの name にアクセスできない．この Human クラスのオブジェクトを視覚化した図を図 6 に示す．

着目しているオブジェクトからアクセスできる場合は図 6 の左側のように図示し，アクセスできない場合は右側のように図示する．右側のように name を図示しないことで，着目しているオブジェクトから name にアクセスできないことを視覚的に示す．

このように図示すれば，着目しているオブジェクトから，他のオブジェクトの持つメンバへのアクセスの可否が図からわかる．

3.4 ステップ毎の視覚化

プログラムをステップ実行しながら，第 3.1 節から第 3.3 節までに示した視覚化を行う．左側にソースコードを表示し，右側に本手法で視覚化したオブジェクトの図を表示する．プログラム実行の，ステップ毎に右側の図を更新する．図 7 にステップ毎に本手法で視覚化した例を示す．

図 7 は Student クラスのコンストラクタを呼び出した時

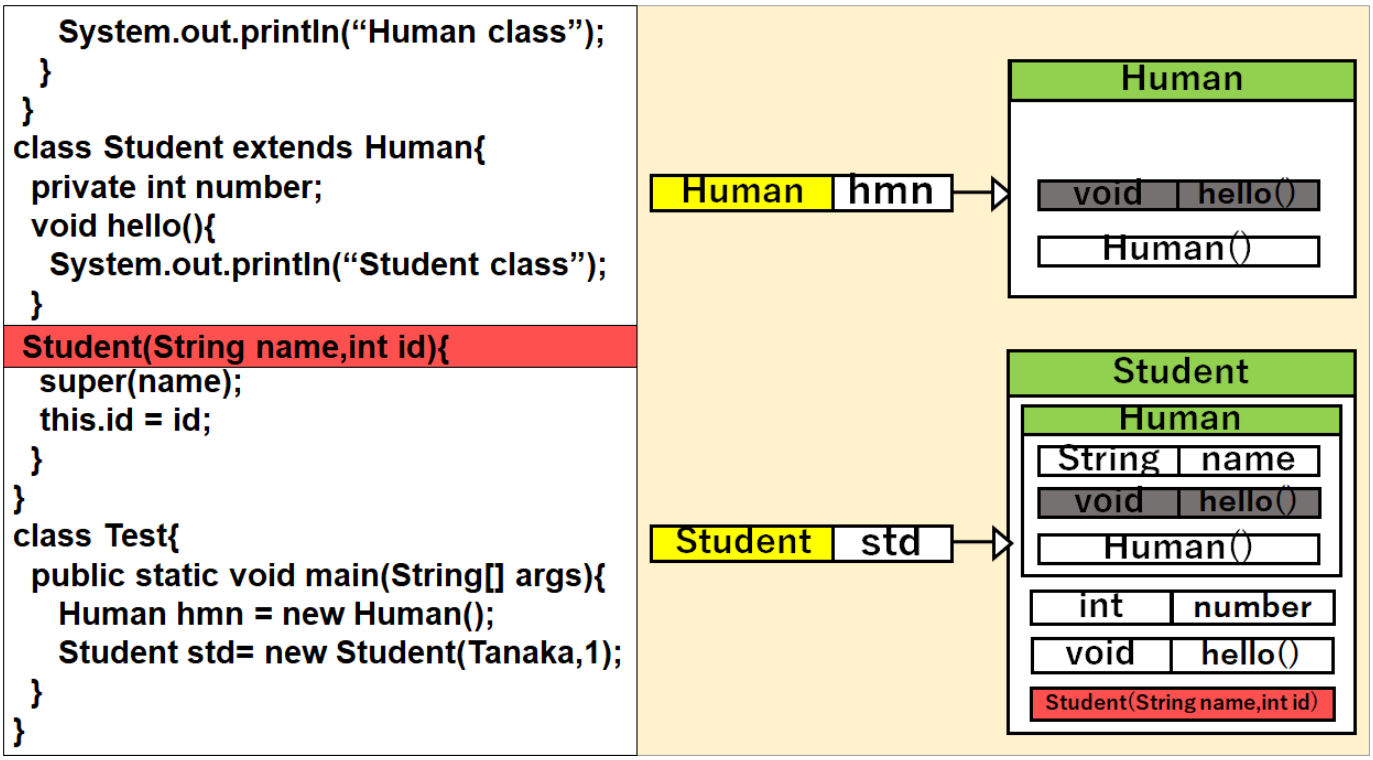

図 7 ステップ毎の視覚化
Fig. 7 Displaying Stepwise visualizaion

点の実行状態をで提案手法で視覚化した図である．プログラム中で，現在実行しているステップと，実行中のメソッドと，アクセスされているインスタンス変数は背景を強調色にしてそれらを強調する．

プログラムのステップによって，着目するオブジェクトが変化する．また，着目するオブジェクトによって，生成された各オブジェクトの持つインスタンス変数・メソッドへのアクセスの可否が変化する．その変化に伴って右側に表示されている図が変化する．初学者は，プログラムをステップ実行し，右側の図の変化を見ることで，ソースコードを追わなくてもメンバの可視性の変化が視覚的にわかる．

3.5 カプセル化への応用

第 3 節で提案した視覚化手法を，カプセル化されていないプログラムとカプセル化されたプログラムに適用する．

例として，次の Java プログラムを考える．

```
1:  class Human {
2:    public String name;
3:  }
```

この Human クラスはカプセル化されておらず，クラスのメンバにインスタンス変数 name しか定義されていない．そのため，name を操作するには直接 name にアクセスしなければならない．このクラスのオブジェクトを視覚化した図を図 8 に示す．

図 8 を見ると，生成された Human クラスのオブジェクトが name 以外のインスタンス変数や，メソッドを持っておらず，カプセル化されていないことが一目でわかる．

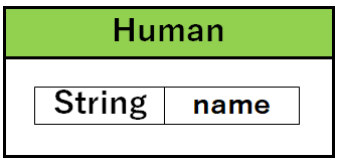

図 8 カプセル化されていない場合
Fig. 8 Displaying Not Encupsulated Object

このプログラムに対して，カプセル化した次のプログラムを考える．

```
1:  class Human {
2:    private String name;
3:    public String getName() {
4:      return name;
5:    }
6:    public void setName(String name) {
7:      this.name = name;
8:    }
9:  }
```

この Human クラスは，name というインスタンス変数と，name を操作するためのメソッド getName と setName が定義されている．インスタンス変数とそれを操作するメソッドが定義された，カプセル化されたクラスである．name にはアクセス修飾子 private がついており，他のオブジェクトからアクセスできない．name へアクセスする場合は

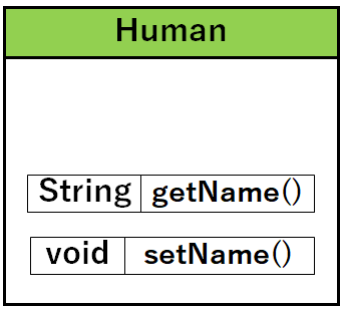

図 9 カプセル化されている場合
Fig. 9 Displaying Encapsulated Object

メソッドを通じてのみアクセスされるようにカプセル化を用いて情報隠蔽をしている．このクラスのオブジェクトを視覚化した図を図 9 に示す．

他のオブジェクトから name へアクセスできないため，他のオブジェクトに着目している場合は name は図示されず，図 9 のように getName メソッドと setName メソッドのみが図示される．どちらかのメソッドが呼び出されているときは name が図示される．メソッドを通じてインスタンス変数を操作する様子が図の変化から読み取れる．カプセル化や情報隠蔽がどのようなものなのか図からわかる．

初学者は図 8 と図 9 を比較することで，カプセル化されたクラスと，カプセル化されていないクラスのオブジェクトの違いを視覚的に学べる．また，カプセル化を用いて情報隠蔽を行っているクラスが，他クラスからどのように見えるのかが図からわかる．

3.6 ポリモーフィズムへの応用

第 3.5 節と同様に，本手法による視覚化をポリモーフィズムを利用しているプログラムに適用する．ポリモーフィズムとは，メソッド呼び出しに対する動作が参照しているオブジェクトによって決定されることである．ポリモーフィズムを利用した Java プログラムの例を示す．

```
1:  class Shape {
2:    public int x;
3:    public int y;
4:    public int height;
5:    public int width;
6:    abstract public void draw();
7:  }
8:  class Square extends Shape {
9:   public void draw(){
10:     // 四角形を描画するメソッド
11:   }
12: }
13: class Triangle extends Shape {
14:   public void draw(){
15:     // 三角形を描画するメソッド
16:   }
17: }
```

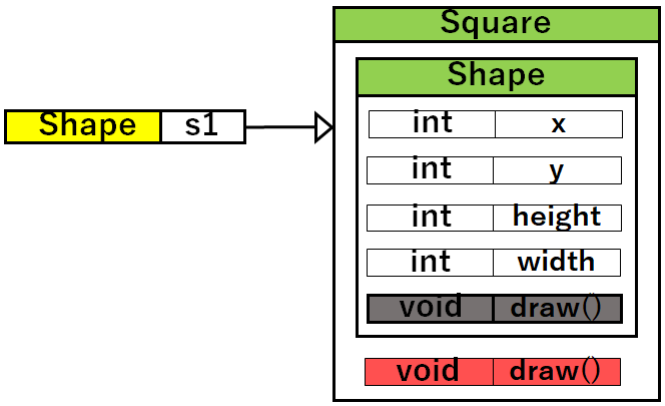

図 10 30 行目での視覚化の図
Fig. 10 Visualization After Execution of ine 30

```
18: class Test {
19:   public static
20:     void main(String[] args){
21:       Shape s1 = new Square();
22:       Shape s2 = new Triangle();
23:       s1.draw();
24:       s2.draw();
25:   }
26: }
```

Shape クラスは図形を扱うためのクラスで，座標を保持するインスタンス変数 x，y と図形の大きさを保持するインスタンス変数 height と width を持つ．そして，図形を描画するためのメソッドとして，抽象メソッド draw を定義している．この Shape クラスを Square クラスと Triangle クラスが継承し，抽象メソッドである draw を実装している．

ポリモーフィズムが利用されているのが 30 行目と 31 行目である．30，31 行目では Shape クラスの参照型変数の s1 と s2 で draw メソッドを呼び出している．s1 では Square クラスの draw メソッドが呼び出され，s2 では Triangle クラスの draw メソッドが呼び出される．同じメソッド呼び出しでも，参照しているオブジェクトによって，Square クラスと Triangle クラスのどちらの draw メソッドが呼び出されるのかが決まる．

同じメソッド呼び出しにもかかわらず動作が異なる場合があるため，初学者はソースコードだけを見てもプログラムの動作が理解しづらい．そこで，本研究の提案手法を適用して学習支援を行う．30 行目と 31 行目のステップを実行している時の図をそれぞれ図 10 と図 11 に示す．

図 10 を見ると，Square クラスのオブジェクトの draw メソッドを表している四角の背景色が変わって強調されている．30 行目の s1.draw() では Square クラスの draw メソッドが呼び出されたことが，図の強調表示からわかる．同様に，図 11 から，31 行目のステップでは Triangle クラスの draw メソッドが呼び出されたことがわかる．

このように，ポリモーフィズムを使うプログラムに提案

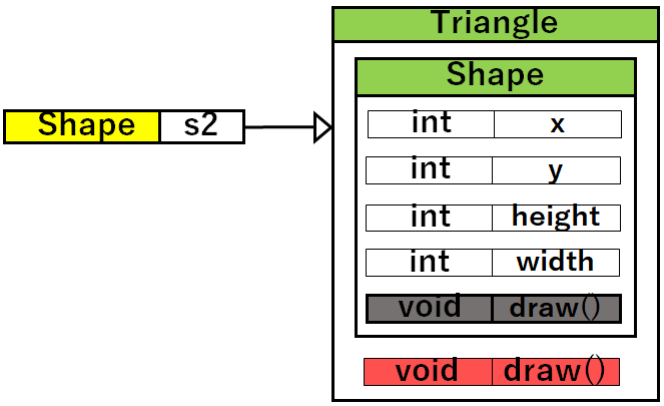

図 11 31 行目での視覚化の図
Fig. 11 Visualization After Execution of line 31

手法を適用すると，メソッド呼び出し時にどのクラスのメソッドが呼び出されているかが図からわかる．このため，初学者は図からポリモーフィズムの働きを理解できる．

4. 関連研究との比較

既存の視覚化手法と，本研究で提案した視覚化手法を比較する．比較対象は，第 2 節で紹介した Jeliot 3 と UML である．

4.1 継承の視覚化の比較

継承の視覚化について，本手法と UML のクラス図による視覚化，Jeliot 3 による視覚化を比較する．次に示すプログラムを視覚化する場合を考える．

```
1:  class Human {
2:    public String name;
3:    public void hello(){
4:      System.out.println("I'm a Human.");
5:    }
6:  }
7:  class Student extends Human {
8:    private int id;
9:    public void hello(){
10:     System.out.println("I'm a Student.");
11:   }
12: }
```

このプログラムを UML のクラス図を用いて表した図を図 12 に示す．

図 12 のクラス図を見ると，どのクラスを継承しているかは図からすぐわかるが，オブジェクトを生成した際にそのオブジェクトがどのメンバを持っているかや，メソッドがオーバーライドされているかどうかは図を読まないとわからない．

次に，Jeliot 3 での視覚化との比較を行う．Jeliot 3 で生成された図を図 13 に示す．

図 13 を見ると，親クラスのメンバと自身のクラスのインスタンス変数が図示されているため，生成されたオブジェクトがどんなインスタンス変数を持つかが図からわかる．しかし，どのインスタンス変数が親クラスのインスタンス変数かがわからない．メソッドに関しては，メソッド呼び出し時の動きがアニメーションで表現されるので，どのようにメソッドが実行されるかはわかるが，オブジェクトが持つメソッドが図示されないため，生成されたオブジェクトがどんなメソッドを持つかがわからない．また，オーバーライドの視覚化がされないので，メソッドがオーバーライドされているかどうかはソースコードを見て確認しないとわからない．

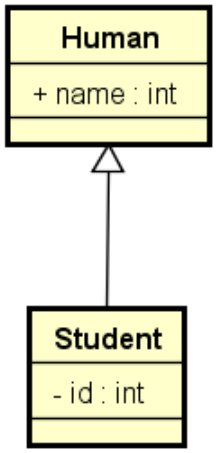

図 12 クラス図による継承の視覚化
Fig. 12 Visualization of Inheritance by Class Diagram

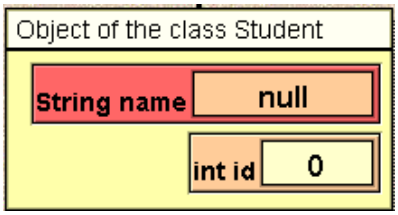

図 13 Jeliot 3 での継承の視覚化
Fig. 13 Visualization of Inheritance by Jeliot 3

これらに対し，本研究の提案手法で視覚化した図を図 14 に示す．

図 14 では，図 12 に比べて，生成されたオブジェクトがどんなメンバを持つかがすぐわかるようになっている．UML や Jeliot 3 では図だけではメソッドのオーバーライドがわかりにくかったが，提案手法では背景色が暗くなっているメソッドがオーバーライドされているということが図からすぐわかる．また，図 13 に比べて，どのメンバがスーパークラスのものなのかがすぐわかる．

4.2 可視性の視覚化の比較

UML のクラス図では，図 12 にあるようにメンバが public

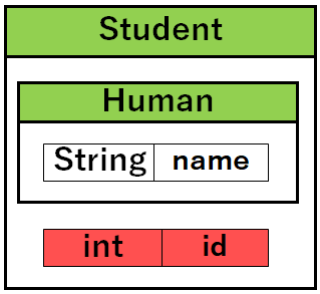

図 14 提案手法での継承の視覚化
Fig. 14 visualization of inheritance by propose method

な場合は+，private な場合はを用いて可視性を表す．また，protected な場合は#，同パッケージ内にあるクラスに対してのみアクセスできる場合は~を用いる．可視性は視覚化できているが，メンバの数が多くなると初学者にとって図を読むことが困難になる．また，ステップによるオブジェクトの持つインスタンス変数やメソッドの可視性の変化はわからない．

また，Jeliot 3 は可視性を視覚化する機能がない．そのため，図からオブジェクトの持つインスタンス変数やメソッドの可視性についてはわからない．可視性についてはユーザーがソースコードを読んで理解する必要がある．

これらに対して，今回提案した視覚化手法では第 3.3 節で示したように，着目するステップでアクセスできるメンバが一目でわかる．また，第 3.4 節で示したように，ステップ毎に視覚化しているため，可視性の変化が図からわかる．UML，jeliot 3 に対して，提案手法の方がメンバの可視性の視覚化がより行えている．

5. 結論と課題

プログラムの各ステップごとにオブジェクトを視覚化することでメンバの可視性の変化を視覚化する手法を提案した．オブジェクトを参照する参照変数と，オブジェクトとの参照関係，オブジェクトの持つメンバ，メンバの可視性を図示することで，ソースコードを追わなくてもオブジェクト指向プログラミングにおける継承・カプセル化・ポリモーフィズムがプログラムのステップ毎に図が変化する様子からわかる．

今後の課題として，今回の提案手法で視覚化を行う多言語に対応したツールの開発が考えられる．この論文では例として Java のプログラムを用いていたが，Java 以外のオブジェクト指向プログラミングができる複数の言語に対して今回の提案手法を適用できるツールの開発ができれば，より多くの初学者に対して学習支援ができる．

また，今回の提案手法に対しての評価の実施も今後の課題である．この手法が実際に初学者のオブジェクト指向プログラミングの学習に対して有効であるか，初学者を対象とした評価実験を実施することを考えている．

参考文献

[1] A. Moreno, N. Myller, and E. Sutinen, “Visualizing programs with Jeliot 3”, Proceedings of the Working Conference on Advanced Visual Interfaces, pp. 373–376, 2004.
[2] U. Sharma, “New Ways of Learning OO Concepts through Visualization & Animation Techniques for Novices.”, International Journal of Advanced Research in Computer Science, 2015.
[3] S. Cooper, W. Dann, Pausch, “Alice: A 3-D tool for introductory programming concepts.”, Proceedings of the 5th Annual CCSC Northeastern Conference 2000, pp. 28-29, April 2000.
[4] M. Klling, “The greenfoot programming environment.”, Proceedings of ACM Transactions on Computing Education, 2010.
[5] M. Klling, B. Quig, A. Patterson, R. John, “The BlueJ system and its pedagogy.”, Proceeding of Computer Science Education, pp. 249-268, 2003.
[6] Guo, Philip J, “Online python tutor: embeddable web-based program visualization for cs education.”, Proceeding of the 44th ACM technical symposium on Computer science education, 2013.
[7] Homer, Michael, N. James, “Combining tiled and textual views of code.” Proceedings of 2nd IEEE Working Conference on Software Visualization, 2014.
[8] Kang, Hyeonsu, Philip J, Guo, “Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations.” Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology, 2017.
[9] Sinan Si Alhir 著, 原隆文訳, 「入門 UML」, オーム社, 2003.