日本ソフトウェア科学会第 28 回大会 (2011 年度) 講演論文集

細かい粒度でコードの再利用を可能とするメソッド内メソッドとその効率の良い実装方法の提案

平松 俊樹　千葉 滋

本研究では，Java 言語へのオーバーライド可能なメソッド内メソッドの導入と，その効率的な実装方法を提案する．プログラミングにおいて，コードの再利用は有益であるが，Java 言語ではメソッドより小さい，ブロック単位でのコードの再利用ができない．本システムでは，外側のローカル変数を参照できるメソッド内メソッドを用いてメソッドを分割でき，またそれをサブクラスでオーバーライドすることで，細かい粒度でのコードの再利用を可能としている．また，コード変換時に可能であればメソッド内メソッドをインライン展開することで，実行時の性能を犠牲にせずにこのシステムを実装する．

1　はじめに

プログラミングにおいて，コードの再利用はとても有益である．コードを再利用することで，生産性の向上，同一コードのコード中における散逸の防止などが期待できる．オブジェクト指向言語においては，クラス間の継承関係を利用してメソッドやフィールドなどを再利用することが可能である．

しかし，Java 言語においてはコードの再利用の最小単位はメソッドであり，それより小さな単位，つまりブロック単位でのコードの再利用ができない．このため，メソッドの一部分を再利用し一部分を変更するといったことを既存の Java 言語において実現することは困難である．

このようなメソッドの一部の変更を行うために，メソッドの一部を別のメソッドとして定義しなおし，それをサブクラスにおいてオーバーライドすることでメソッドの一部を上書きするという方法が考えられる．しかし，あるメソッドが別のメソッド内で宣言されたローカル変数にアクセスすることは不可能であるため，メソッドの一部を別のメソッドとして定義しなおした場合には，必要なローカル変数を引数として定義しなおされたメソッドに渡さなければならない．さらに，定義しなおされたメソッドの内部から，元のメソッドのローカル変数への代入がある場合には，定義しなおされたメソッドの返り値を用いて，元のメソッドにおいてローカル変数への代入を行うなどの作業が必要となる．以上の様な作業が必要となるため，メソッド内の一部を別のメソッドとした場合にはコードが煩雑になる恐れがある．

そこで，本研究では，Java 言語に，オーバーライド可能でメソッド内に定義可能なメソッドを導入するシステムと，その効率的な実装方法を提案する．本システムにおいては，メソッド内に定義されたメソッドは，自身の定義をその内部に含むメソッドのローカル変数にアクセスすることができる．そのため，本システムを用いた場合には，既存の Java 言語においてメソッドの一部を別のメソッドとした場合の様な，引数としてのローカル変数の引き渡し，返り値を元にしたローカル変数への代入といった作業を行う必要がなくなる．また，そのメソッド内メソッドをオーバーライドすることで，より細かい粒度でコードの再利用を行うことができる．

本稿の残りは，次のような構成からなっている．第

A proposal of method-in-methods for fine-grained code reuse and its efficient implementation

Toshiki Hiramatsu, 東京工業大学大学院 数理・計算科学専攻, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology.

```
class Max {
  int calc(int[] a) {
    int sum = 0;
    int max = a[0];
    int ave;
    for (int i = 0; i < a.length; i++) {
      sum += a[i];
      if (a[i] > max)
        max = a[i];
    }
    ave = sum / a.length;
    return max - ave;
  }
}
```

図 1　メソッドへの分割が難しい例

2 章では Java 言語におけるコードの分割と再利用について，第 3 章ではオーバーライド可能なメソッド内メソッドについて述べる．第 4 章では本システムの実装について，第 5 章では性能を調べるための実験について，第 6 章ではまとめと今後の課題について述べる．

2　Java 言語におけるコードの分割と再利用

大きなメソッドを小さなメソッドに分割することには，メソッドが見やすくなる，意味のあるまとまりに分割してメソッド名がつけられる等の利点がある．また，分割したメソッドをオーバーライドすることで，メソッドの一部分を変更することができ，再利用性が高まる．

ところが，ローカル変数への代入を含むようなメソッドの場合には，分割するのが難しいことがある．簡単な例として，図 1 を挙げる．これは，整数型配列の要素の平均値と最大値の差を求めるものである．

これを分割する場合には，仮に C++言語のような参照渡しが Java にあるとすると，理想的には図 2 のようにしたい．しかし実際には Java には参照渡しが無いため，メソッド calc で宣言されたローカル変数 sum, max, i にメソッド calcSum から代入をすることができず，このような分割は不可能である．また，C++言語の参照渡しは呼ぶ側から値渡しと区別がつかず，副作用の有無が分からないという欠点がある．

仮にこのような分割が可能であったとすると，この

```
class Max {
  int calc(int[] a) {
    int sum = 0;
    int max = a[0];
    int ave;
    for (int i = 0; i < a.length; i++) {
      calcSum(sum, max, i, a);
    }
    ave = sum / a.length;
    return max - ave;
  }
  void calcSum(int& sum,
       int& max,
       int i,
       int[] a) {
    sum += a[i];
    if (a[i] > max)
      max = a[i];
  }
}
```

図 2　理想的な分割

```
class Min extends Max {
  void calc(int[]).calcSum(int& sum,
       int& max,
       int i,
       int[] a) {
    sum += a[i];
    if (a[i] < max)
    max = a[i];
  }
}
```

図 3　図 2 のコードを再利用

実装を再利用して，サブクラスで配列の要素の平均値と最小値の差を求めるメソッドを定義できる．これを図 3 に示す．もちろんこの場合も，Java には参照渡しが無く，メソッド calcSum からローカル変数 sum, max, i に代入ができないため許されない．

3　オーバーライド可能なメソッド内メソッド

本研究では，Java 言語にオーバーライド可能なメソッド内メソッドを導入する．これにより，従来のメソッドを用いてコードを再利用した場合と比べ，引数としてのローカル変数の引き渡し，オブジェクトとし

```
class Max {
  int calc(public int[] a) {
    public int sum = 0;
    public int max = a[0];
    int ave;
    for (public int i = 0; i < a.length; i++){
      void calcSum() {
        sum += a[i];
        if(a[i] > max)
          max = a[i];
      }
      calcSum();
    }
    ave = sum / a.length;
    return max - ave;
  }
}
```

図 4　メソッド内メソッドの定義

ての計算結果の受け渡しと，それをもとにしたローカル変数への代入といった作業なしに，コードの再利用，メソッドの一部の変更が可能になる．

3.1　構文

本システムは，Java 言語を拡張することで実現している．図 4 にメソッド内メソッドの定義の例を示す．これは図 1 のコードをメソッド内メソッドを用いて書き直したものである．本システムでは，この図のメソッド calc 内にある calcSum のようにメソッド内にメソッドを定義することができる．また，引数とローカル変数に public 修飾子を付けて宣言することができる．public のついた変数については，次節で説明する．

また，メソッド内メソッドをオーバーライドするコードの例を図 5 に示す．この図は図 3 をメソッド内メソッドを用いて書き直したものである．この図のように，外側のメソッド名とメソッド内メソッドの名前を".(ドット)"を用いてつなげて宣言することで，メソッド内メソッドのオーバーライドを行うことができる．

3.2　メソッド内メソッドからのローカル変数の参照

```
class Min extends Max {
  void calc(int[]).calcSum() {
    sum += a[i];
    if(a[i] < max)
    max = a[i];
}
```

図 5　メソッド内メソッドのオーバーライド

メソッド内メソッドは，それがオーバーライドされない限り，その外側のメソッドで宣言されたローカル変数を参照することができる．図 4 の例では，メソッド内メソッド calcSum からは，外側のメソッド calc で宣言されたローカル変数 sum, max, i, ave および引数の a が参照可能である．

また，本システムでは，ローカル変数に public 修飾子を付けることが可能である．オーバーライドしたメソッド内メソッドから参照できるのは public 修飾子のついたローカル変数と引数のみである．図 5 の例では，クラス Min のメソッド calcSum から参照できるのは，クラス Max でメソッド calc で public 宣言されたローカル変数 sum, max, i および引数 a のみであり，public 修飾子の付いていないローカル変数 ave は参照することができない．オーバーライドしたメソッド内メソッドから参照可能なローカル変数は public 変数のみであるという制限を設けることで，カプセル化を保つことができる．

3.3　メソッド内メソッドの呼び出し制限

メソッド内メソッドは，以下の二つの形式の呼び出しのみが可能である．

- あるメソッド内メソッドからの，同じメソッド内メソッドの再起呼び出し
- あるメソッドからの，そのメソッドのボディ内に直接定義が書かれたメソッド内メソッドの呼び出し

従って，図 4 のメソッド calc から呼び出せるメソッド内メソッドは，calc 内にその定義が書かれたメソッド内メソッドである calcSum のみである．

4 ソースコード変換による実装

本システムでは，ソースコードの変換を行うことによって，メソッド内メソッドを実現する．以下に，スーパークラスにおけるメソッド内メソッドの定義および，サブクラスにおけるメソッド内メソッドのオーバーライドの際のソースコード変換の方法について述べる．

4.1 スーパークラスにおけるメソッド内メソッド

図 4 のコードを例に，変換について説明する．クラス Max では，メソッド calc の内部にメソッド内メソッド calcSum が定義されている．このとき，本システムでは，三つのメソッド calc, $calc, calcSum およびクラス PublicVar$calc を作成し，クラス Max のメンバとする．

メソッド calc は，元のメソッド calc のボディ内の全ての calcSum の呼び出しを calcSum のボディに置き換えたものである．この変換を行うことによって，メソッド呼び出しや，メソッドの返り値を元にしたローカル変数への代入などの作業がなくなり，メソッドの実行速度の向上が望める．

メソッド$calc は，元のメソッド calc のボディの先頭に，calc における public 変数を集めたクラスである PublicVar$calc クラスのインスタンスを作成する文を追加し，public 宣言されたローカル変数 sum, max 等へのアクセスを `pVar$calc.sum`，`pVar$calc.max` 等へと変更し，メソッド calcSum を呼び出す際の引数に pVar$calc を追加したものである．

メソッド calcSum は，メソッド内メソッド calcSum を通常のメソッドとして変換したものであり，引数として `PublicVar$calc pVar$calc` が追加される．また，メソッド calcSum のボディ内の，メソッド calc において public 宣言されたローカル変数 sum 等へのアクセスは，メソッド$calc のボディにおける場合と同じく，`pVar$calc.sum` 等へと変更する．

クラス PublicVar$calc は，メソッド calc のボディにおいて public 宣言されたローカル変数に相当するフィールドを持つクラスである．この例の場合では，クラス PublicVar$calc はフィールドとして int 型の sum, max, i および int 型の配列 a を持つ．

```
class Max {
  int calc(int[] a) {
    int sum = 0;
    int max = a[0];
    int ave;
    for (int i = 0; i < a.length; i++) {
      sum += a[i];
      if (a[i] > max)
        max = a[i];
    }
    ave = sum / a.length;
    return max - ave;
  }
  int $calc(int[] a) {
    PublicVar$calc pVar$calc =
      new PublicVar$calc();
    pVar$calc.sum = 0;
    pVar$calc.max = a[0];
    int ave;
    for (pVar$calc.i = 0;
      pVar$calc.i < pVar$calc.a.length;
      pVar$calc.i++) {
      calcSum(pVar$calc);
    }
    ave = pVar$calc.sum /
      pVar$calc.a.length;
    return pVar$calc.max - ave;
  }
  void calcSum(PublicVar$calc pVar$calc) {
    pVar$calc.sum += pVar$calc.a[i];
    if (pVar$calc.a[pVar$calc.i] >
      pVar$calc.max)
      pVar$calc.max = a[pVar$calc.i];
  }
  class PublicVar$calc {
    int sum;
    int max;
    int i;
    int[] a;
  }
}
```

図 6 図 4 を変換したもの

メソッド内メソッド calcSum が再起呼び出しされる場合には，メソッド calc 作成時に calcSum() をそのボディと置き換えない．この場合は，メソッド calc において宣言された全てのローカル変数に相当するフィールドを持つクラスのオブジェクトを，メソッド calcSum 呼び出し時の引数に追加する．以上の変換を行うと，図 4 のコードは図 6 のようになる．

```
class Min extends Max {
  void calcSum(PublicVar$calc pVar$calc) {
    pVar$calc.sum += pVar$calc.a[i];
    if(pVar$calc.a[pVar$calc.i] <
      pVar$calc.max)
      pVar$calc.max =
        pVar$calc.a[pVar$calc.i];
  }
  int calc(int[] a) {
    return super.$calc(a);
  }
}
```

図 7　図 5 を変換したもの

4.2　メソッド内メソッドのオーバーライド

図 5 のコードを例として，メソッド内メソッドをオーバーライドするクラスの変換について説明する．

このオーバーライドは，スーパークラス Max の変換の結果作成されるメソッド calcSum をオーバーライドすることによって実現する．そのために，スーパークラスにおけるメソッド calcSum の場合と同様の変換を行い，メソッド内メソッド calcSum を通常のメソッドにする．

クラス Max のメソッド calc は calcSum がすでにインライン展開されたものであるため，クラス Min におけるメソッド calcSum の上書きを反映できない．したがって，クラス Min のオブジェクトに対してメソッド calc を呼んだ際には，calcSum をインライン展開していない，メソッド$calc が呼ばれるようにしなければならない．そのために，`super.$calc();` を行うメソッド calc を作成し，クラス Min のメンバに登録する．以上の変換を行うと，図 5 のコードは図 7 のようになる．

5　実験: マイクロベンチマーク

本システムの性能を確かめるために，図 6 の Max および図 7 の Min と，図 8 のコードの Max, Min の各クラスの calc メソッドに要素数 5 の配列を引数に渡して 1000000 回ずつ実行し，かかった時間を計測した．図 8 のコードは，図 6 および図 7 のコードを，本システムを用いずに素朴に記述した場合を想定したものである．このコードでは，calc メソッドで宣言

```
public class Max {
  int calc(int[] a) {
    int sum = 0;
    int max = a[0];
    int ave;
    for (int i = 0; i < a.length; i++) {
      Values values =
                calcSum(sum, max, i, a);
      sum = values.sum;
      max = values.max;
    }
    ave = sum / a.length;
    return max - ave;
  }
  Values
    calcSum(int s, int m, int i, int[] a) {
    s += a[i];
    if (a[i] > m)
      m = a[i];
    Values v = new Values();
    v.sum = s;
    v.max = m;
    return v;
  }
  class Values {
    int sum;
    int max;
  }
}
class Min extends Max {
  Values
    calcSum(int s, int m, int i, int[]a) {
    s += a[i];
    if (a[i] < m)
      m = a[i];
    Values v = new Values();
    v.sum = s;
    v.max = m;
    return v;
  }
}
```

図 8　本システムを用いずに記述した例

されたローカル変数に代入するための値を，calcSum メソッド内で Values クラスのオブジェクトを作成し，そのフィールドに代入することで calc メソッドに渡している．calc メソッドでは，返されたオブジェクトをもとに，ローカル変数への代入を行っている．

計測結果は表 1 のようになった．クラス Max の場合はメソッド内メソッドを用いたほうが，用いない場合に比べて 0.17 倍の時間で実行が終了し，クラス

表 1 実行時間（ミリ秒）

クラス	時間
本システムを用いた Max	20
本システムを用いた Min	70
通常の Java で書いた Max	119
通常の Java で書いた Min	108

Min の場合はメソッド内メソッドを用いたほうが，用いない場合に比べて 0.65 倍の時間で実行が終了した．

実行時間の差は，主にオブジェクト作成の回数によるものと思われる．図 6 の Max クラスの場合は，calc メソッド呼び出しに際してオブジェクトの作成を一度も行わないが，図 8 の Max クラスの場合は，calc メソッド内の calcSum メソッドの呼び出し毎にオブジェクトの作成が行われる．また，図 7 の Min クラスの場合は，calc メソッドの実行毎にオブジェクトの作成が行われるが，図 8 の Min クラスの場合は，Max クラスのときと同様に，calc メソッド内の calcSum メソッドの呼び出し毎にオブジェクトの作成が行われるため，図 7 の場合と比べて，calc の引数である配列の要素数倍の回数のオブジェクトの作成が行われることになる．

6 関連研究

6.1 Beta

オブジェクト指向言語 Beta [3] では，メソッドのネスト化が可能であり，ネスト化されたメソッドをオーバーライドすることができる．以下に Beta で記述されたネスト化されたメソッドを示す．

```
V:< (# x: @Integer;
       VV:< (# do l1; inner; l2 #);
    #)
```

この例においては，メソッド V の内部にメソッド VV が定義されている．メソッド VV からは V に定義された Integer 型の変数 x が参照可能である．また，VV 内の `do l1; inner; l2` は，l1 と l2 を実行することを表しており，inner はメソッドがオーバーライドされる際の各文の実行順序を定めるためのものである．

このように，Beta ではメソッド内にメソッドを定義することは可能であるが，メソッドをオーバーライドする際に，スーパークラスにおける振る舞いが取り除かれない．以下にこの例を示す．先の例のメソッド VV をサブクラスにおいて，以下のように記述してオーバーライドする場合を考える．

```
VV::< (# do l3; inner; l4 #)
```

このとき，最終的なメソッド VV の実装は以下のようになる．

```
VV: (# do l1; l3; inner; l4; l2 #)
```

このように，一度スーパークラスのメソッド内に現れた振る舞い（この例では l1, l2）が，サブクラスにおいてオーバーライドされた後にも必ず現れるため，オーバーライドによって完全にメソッドを置き換えることが不可能である．

6.2 Closure Joinpoints

Closure Joinpoints [2] はアスペクト指向プログラミングにおけるジョインポイントを指定するための技術である．この技術を用いることでコードのある領域を明示的に指定し，その領域をメソッドとして切り分けることなくジョインポイントとすることができる．Closure Joinpoints で指定されたジョインポイントに対してアドバイスを織り込むことで，ブロック単位のコードの変更を実現することができる．図 9 に，Closure Joinpoints を用いたコードを示す．

図 9 の例では，9 行目の exhibit から 12 行目にかけてのコードが，明示的に指定されたジョインポイントであり，この部分に対してアドバイスを織り込むことで，変更を加えることができる．20, 21 行目がジョインポイントに対するアドバイスである．

Closure Joinpoints では，アドバイスのコードから元のメソッドのローカル変数へ代入を行うことができない．また，これは領域を指定するための機構であるため，指定した領域をメソッドのように複数回，別の場所で呼び出すなどのことはできない．

6.3 Regioncut

regioncut [1] は，Closure Joinpoints と同じくアスペクト指向プログラミングにおけるジョインポイント

```
//base code
class ShoppingSession {
    int totalAmount = 0;
    ShoppingCart sc = new ShoppingCart();

    void buy(final Item item, int amount) {
        Category category =
        Database.categoryOf(item);
        totalAmount =
        exhibit Buying(int amount, Category c) {
            sc.add(item, amount);
            return totalAmount + amount;
        }(amount, category);
    }
}

//advice
aspect BonusProgram {
    joinpoint int
    Buying(int amount, Category cat);
    int around Buying(int amt, Category cat) {
        if (cat == Item.BOOK) amt += amt / 2;
        return proceed(amt, cat);
    }
}
```

図 9 Closure Joinpoints を用いたコード

を指定するための技術である．regioncut は，コード領域を二つのポイントカット指定子の間の領域として指定することで，コード領域を選択可能にしている．例えば，

```
region[call(* *.a()), call(* *.b())]
```

のようにすることで，この regioncut は

```
a();
int x = 42;
y();
b();
```

のような，メソッド呼び出し a() で始まり，b() で終わるコード領域を指定する．

regioncut では，指定されたコード領域内の変数の値を取得しアドバイスへの引数として渡すことで，アドバイス内でその値を使用できる．しかし，元のメソッドのローカル変数への代入をアドバイス内から行うことは不可能である．

6.4 クロージャ

クロージャを用いると，メソッドの一部をブロック単位で分けることが可能であり，クロージャからはその外側のローカル変数を参照できる．しかし，クロージャはサブクラスで上書きするというような使い方をすることができない．

7 まとめ

細かい粒度でコードの再利用を可能とするメソッド内メソッドとその効率の良い実装方法の提案を行った．本システムにおけるメソッド内メソッドは，単にメソッド内に別のメソッドを定義できるだけでなく，メソッド内メソッドから，その外側のメソッドで宣言されたローカル変数を参照することができる．また，メソッド内メソッドをサブクラスでオーバーライドすることができ，細かい粒度でのコードの再利用が可能となる．さらに，コード変換時にメソッド内メソッドをインライン展開することで，実行速度の向上が期待でき，実験を行ってそれを確かめた．

実装が完成していないため，今後の課題はそれを完成させることである．実装を完成させた後に，実験を行い性能の向上を確かめることも今後の課題である．また，メソッド内メソッドのボディ内に return があった場合に，現在はメソッド内メソッドからのリターンであるが，外側のメソッド全体からリターンする設計もあり得るので，今後考えなければならない．

参 考 文 献

[1] Akai, S. and Chiba, S. : A Designator for Selecting a Code Region in Aspect-oriented Programming Language. Information Processing Society of Japan, 2011.

[2] Bodden, E. : Closure joinpoints: Block Joinpoints without Surprises. AOSD, 2011.

[3] Knudsen, J. L. and Lofgren, M, Lehrmann-Madsen, O. and Magnusson, B.: Object-Oriented Environments - The Mjolner Approach, Pretice Hall, 1994