メソッド抽出リファクタリングにおける テストケースの自動生成

三宅 皐[1,a)]　紙名 哲生[2,b)]　丸山 勝久[2,c)]

概要：リファクタリングにおいて外部的振る舞いを保存するため，その適用によって影響を受けるソースコードに対して十分なテストを実施することが一般的である．しかしながら，ExtractMethod リファクタリングの適用において新規にメソッドを抽出しても，そのメソッドの外部的振る舞いを直接検査するテストケースは存在しない．このような状況において，このメソッドに対してさらなるリファクタリングを適用する場合，新たにテストケースを用意する必要があるが，その作業は開発者にとって面倒である．そこで，本論文では，ExtractMethod の適用において新規に抽出したメソッドに対して，その実行に関係する入力変数と出力変数の値を収集するログコードを挿入し，もとのメソッドのテストケースを用いて，そのメソッドを実際に実行することで，テストケースを自動生成する手法を提案する．

1. はじめに

近年のソフトウェア開発や保守において，リファクタリングは主要な活動である．リファクタリングとは，外部的振る舞いを変更することなく既存のコードの設計を改善する活動である [1], [2]．たとえば，ExtractMethod（メソッド抽出）リファクタリングは，長いメソッドを分割したり，重複コードを削除したりするために，既存のメソッドから新規にメソッドを抽出するコード変換である [2]．抽出したメソッドの振る舞いを適切に表現した名前を付けることで，ソースコードの理解性や変更容易性が向上する．

リファクタリングの適用において，コード変換前後で外部的振る舞いを保存することは必須の特性である [1], [3]．実際のソフトウェア開発および保守において，この特性を保証するためにはテストが不可欠である [4]．開発者は，リファクタリング対象のソースコードに対して十分なテストケースを用意し，リファクタリング適用前後のテスト結果が同一であるかどうかを検査する．いま，もとのソースコードのメソッド m_s に ExtractMethod を適用することで，新規にメソッド m_n を抽出した場合を考える．ここでは，m_s から m_n を抽出した後に残るコード断片を含み，m_n を呼び出すメソッド m_r とおく（m_s と m_r のシグニチャは同一である）．この場合，m_s の外部的振る舞いが保存されている（m_r と m_s の外部的振る舞いが同一である）ことを検査するために，m_s のテストケースがそのまま利用できる．いいかえると，m_s に対するすべてのテスト結果が，m_n を呼び出す m_r に対するすべてのテスト結果と一致すればよい．

このようなテストにより，m_s に対する ExtractMethod が安全に実施されたことが確認できたとしても，開発者がリファクタリング作業を完了したと考えるのは早計である．なぜなら，この時点で検査されたのはあくまでも m_s の外部的振る舞いであり，m_n の外部的振る舞いを検査するテストケースは存在しないからである．ここで，m_n を呼び出す m_r が今後も一切変更されないのであれば，m_n を直接的に検査するテストケースをわざわざ用意する必要はない．しかしながら，今後の開発や保守において，m_r が削除されたり，m_r や m_n 内部のコードが書き換えられたりする場合がある．このような場合，m_n の外部的振る舞いの変化を検査するためのテストケースが存在しないことが問題となる可能性がある．たとえば，m_n を呼び出す m_r が削除されてしまうと，同時に m_r に対するテストケースも削除される（残しておいてもテストは実行できない）．この場合，m_n の外部的振る舞いを検査するテストケースが失われる．また，m_r の本体が改変されることで，メソッドの呼び出し条件が変わることもめずらしくない．この場合も m_n の外部的振る舞いを検査するテストケースの一部が失われることがある．

このような問題を解決するためには，ExtractMethod の適用と同時に，m_n に対するテストケースを用意してお

[1] 立命館大学 大学院情報理工学研究科
[2] 立命館大学 情報理工学部
[a)] mii@fse.cs.ritsumei.ac.jp
[b)] kamina@acm.org
[c)] maru@cs.ritsumei.ac.jp

くことが望ましい．特に，m_n に対してさらにリファクタリングを適用する際には，m_n の外部的振る舞いが保存されていることを検査するテストケースの存在は必須である．残念ながら，このようなテストケースを作成するためには m_n の振る舞いを十分に理解する必要があり，これは開発者にとって面倒な作業である．このような問題は文献 [5] でも指摘されている．

そこで，本論文では，Java で記述されたソースコードに対する ExtractMethod の適用において，リファクタリング対象のもとのメソッドのテストケースから，新規に抽出するメソッドに対するテストケースを自動生成する手法を提案する．この手法では，もとのメソッドに対して，JUnit[*1]で記述された十分なテストケースが存在することを前提とする．まず，ExtractMethod の適用後のソースコードを静的に解析することで，その適用により新規に抽出するメソッドの入力変数と出力変数を特定する．入力変数はメソッドの実行に必要な情報（基本型の値やユーザ定義型のインスタンス）を保持し，出力変数はメソッド実行後の結果を保持する．次に，入力変数の値を記録するコード（ログコードと呼ぶ）をメソッドが呼ばれる直後に挿入する．さらに，出力変数の値を記録するログコードをメソッドから戻る直前に挿入する．このようにしてログコードを挿入した状態で，ExtractMethod の適用におけるもとのメソッドのテストケースを利用して，ログコードが挿入されたメソッドを実行し，入力変数の値と出力変数の値を収集する．最終的に，入力変数の値と出力変数の値に基づき，ExtractMethod において新規に抽出するメソッドのテストケースを生成する．

本論文の構成は以下のとおりである．2 章では，ExtractMethod の適用により新規に抽出するメソッドに対するテストケースが不在となることの問題について，例題を用いて説明する．3 章では，Java ソースコードの静的解析と Java プログラムの実行結果を組み合わせることで，新規に抽出するメソッドに対するテストケースを自動生成する手法を提案し，この手法に基づき生成したテストケースを示す．最後に，4 章で，本手法の限界と今後の課題を示す．

2. ExtractMethod の適用における問題

本章では，既存のメソッドに対して ExtractMethod を適用した際，新たに抽出されたメソッドに対するテストケースが存在しないことで問題が発生する具体例を示す．まず，リファクタリング対象のソースコードとその外部的振る舞いを検査するテストケースを示す．次に，このソースコードに対して ExtractMethod を適用して新規にメソッドを抽出した結果を示す．最後に，新規メソッドに対するテストケース不在による問題を説明する．

[*1] http://junit.org/junit4/

```
public class Customer {
  private String name = "";
  public int discount = 0;
  ...

  public Customer(String name) {
    this.name = name;
  }

  public String statement(OrderRecord record) {
    if (record == null || record.getSize() == 0) {
      return "No order";
    }

    /* From here: ExtractMethod */
    int outstanding = 0;
    for (Order order : record.orders) {
      outstanding += order.getAmount(discount);
    }
    /* To here: ExtractMethod */

    return name + "'s amount: " +
      String.valueOf(outstanding);
  }
}
```

```
public class OrderRecord {
  List<Order> orders = new ArrayList<Order>();
  ...

  public void addOrder(Order order) {
    orders.add(order);
  }

  public int getSize() {
     return orders.size();
  }
}
```

```
public class Order {
  ...

  public Order(String title, int price) { ...
  }

  public int getAmount(int discount) { ...
  }
}
```

図 1　リファクタリング対象のソースコード

(1) リファクタリング対象のソースコード

図 1 に ExtractMethod の対象となるソースコードを示す．クラス Customer には，メソッド statement()，フィールド変数 name と discount が定義されている．statement() では，レンタルショップシステムにおいて各顧客の注文情報を保持する OrderRecord と割引率を保持する discount から支払金額を計算し，明細書に記載する料金を含むメッセージを作成している．ここでは，クラス OrderRecord と Order の詳細は省略する．

```
public class CustomerTest {

  @Test
  public void testStatement1() {
    Customer customer = new Customer("C1");
    customer.discount = 10;

    OrderRecord record = new OrderRecord();
    Order order = new Order("Movie", 200);
    record.addOrder(order);

    assertEquals(customer.statement(record),
      "C1's amount: 180");
  }

  @Test
  public void testStatement2() {
    Customer customer = new Customer("C2");
    customer.discount = 20;

    OrderRecord record = new OrderRecord();
    Order movie = new Order("Movie", 200);
    Order music = new Order("Music", 100);
    record.addOrder(movie);
    record.addOrder(music);

    assertEquals(customer.statement(record),
      "C2's amount: 240");
  }

  @Test
  public void testStatement3() {
    Customer customer = new Customer("C3");
    customer.discount = 30;

    OrderRecord record = new OrderRecord();

    assertEquals(customer.statement(record),
      "No order");
  }
}
```

図 2　statement() に対するテストコード

statement() には，その振る舞いが正しいことを確認するために，あらかじめ図 2 に示す 3 つのテストケース（テストメソッド）が用意されている（実際には，3 つのテストケースだけでは不十分である）．それぞれのテストケースの内容を次に示す．

T1　顧客が 1 つの注文を行った場合の料金メッセージを確認する（testStatement1()）

T2　顧客が 2 つの注文を行った場合の料金メッセージを確認する（testStatement2()）

T3　顧客が何も注文しなかった場合の料金メッセージを確認する（testStatement3()）

(2) ExtractMethod の適用

このような状況において，開発者が statement() に ExtractMethod を適用することで，支払金額を計算するコード断片をメソッド getOutstanding() として抽出した場合を考える．抽出対象のコード断片は，図 1 に示すコメント “From here” から “To here” にはさまれる部分である．ExtractMethod 適用後のソースコードを図 3 に示す．getOutstanding() が statement() から呼び出されていることが分かる．

```
public class Customer {
  ...

  public String statement(OrderRecord record) {
    if (record == null || record.getSize() == 0) {
      return "No order";
    }

    int outstanding = getOutstanding(record);

    return name + "'s amount: " +
      String.valueOf(outstanding);
  }

  private int getOutstanding(OrderRecord record) {
    int outstanding = 0;
    for (Order order : record.orders) {
      outstanding += order.getAmount(discount);
    }
    return outstanding;
  }
}
```

図 3　ExtractMethod 適用後のソースコード

ここで，リファクタリングは外部的振る舞いの保存を保証するコード変換である．そこで，statement() に対する 3 つのテストケースを用いて，このメソッドの振る舞いを検査する．この例では，3 つのテストケースにおける実行結果が ExtractMethod の適用前後で同一となるので，外部的振る舞いが保存されていることが確認できる（実際には，他のメソッドに対するテストも実施される）．

(3) 新規メソッドに対するテストケース不在による問題

図 3 に示すソースコードを利用してレンタルショップシステムが運用されていた後，明細書の形式に変更要求が発生したとする．そこで，今後も明細書の形式に対する変更要求が予想されるため，Customer の statement() では明細書の料金メッセージを作成することをやめることにし，statement() を Customer から削除した．これにより，同時に statement() に対するテストケースも削除された．さらに，料金の計算方法は変わらないので getOutstanding() はそのまま Customer に残し，そのアクセス修飾子を private から public に書き換えた．これにより，getOutstanding() は Customer の外部からも呼び出せるようになる．このようなアクセス修飾子の書き換えでは，getOutstanding() の外部的振る舞いは変わらない．

このような状況において，開発者が getOutstanding() に対してリファクタリングを適用する場面を考える．たとえ

```
public class Customer {
  ...

  public int getOutstanding(OrderRecord record) {
    return record.getOutstanding(discount);
  }
}
```

```
public class OrderRecord {
  ...

  public int getOutstanding(int discount) {
    int outstanding = 0;
    for (Order order : orders) {
      outstanding += order.getAmount(discount);
    }
    return outstanding;
  }
}
```

図 4　MoveMethod 適用後のソースコード

ば，MoveMethod を適用して，図 4 のソースコードを作成したとする．この場合，Customer の getOutstanding() の外部的振る舞いが変わっていないことを確認するために，そのメソッドに対するテストケースが必要である．しかしながら，以前に適用した ExtractMethod において，Customer の getOutstanding() に対するテストケースは特に作成されていない．さらに，Customer から statement() が削除されたことにより，getOutstanding() を実行するテストコードはもはや存在しない．そこで，MoveMethod における外部的振る舞いの保存を保証するために，開発者は新たに Customer の getOutstanding() に対するテストケースを用意しなければならない．このようなテストケースを作成するためには，テスト対象のメソッドを理解する必要があり，これは開発者にとって面倒な作業である．

テストケースが不在であることに起因する問題は，ExtractMethod 適用後に getOutstanding() を呼び出す statement() を削除する場合だけで発生するわけではない．いま，図 3 に示すソースコードにおいて，statement() に対するテストケース **T3** は getOutstanding() を呼び出していない．このような状態において，getOutstanding() を図 5 のように改変した（下線部分）としても，statement() のテスト結果は getOutstanding() の改変前後で同一である．一方，getOutstanding() については，引数である record の大きさが 0 の場合に明かに戻り値の値が改変前後で異なる．よって，この改変は getOutstanding() にとって，リファクタリングではない．しかしながら，statement() のために用意したテストケースの実行結果は，getOutstanding() の外部的振る舞いが変化したことを明示していない．このため，開発者によっては，statement() に対するテストケースの実行結果から，getOutstanding() に対する改変をリファクタリングと勘違いする可能性がある．

```
public class Customer {
  ...

  public String statement(OrderRecord record) {
    if (record == null || record.getSize() == 0) {
      return "No order";
    }

    outstanding = getOutstanding(record);
    return name + "'s amount: " +
      String.valueOf(outstanding);
  }

  private int getOutstanding(OrderRecord record) {
    if (record.getSize() == 0) {
      return -1;
    }

    int outstanding = 0;
    for (Order order : record.orders) {
      outstanding += order.getAmount(discount);
    }
    return outstanding;
  }
}
```

図 5　getOutstanding() 改変後のソースコード

以上をまとめると，ExtractMethod の適用により既存のメソッドから新規にメソッドを抽出した場合でも，抽出したメソッドに対するテストケースが用意されないことが問題である．このような状況において，新規に抽出したメソッドやそれを呼び出すメソッドを改変すると，その改変がリファクタリングであることを，既存のテストケースだけを利用して検査することは困難である．このような問題を解決するためには，ExtractMethod の適用後において，新規に抽出したメソッドに対する十分なテストケースを確実に用意しておく必要がある．

3. テストケースの自動生成手法

本章では，ExtractMethod の適用時に，新規に抽出されたメソッドに対するテストケース（JUnit のテストコード）を自動生成する手法を提案する．テストケースの生成においては，ExtractMethod が外部的振る舞いを保存して適用されたことを前提とする．手法は，大きく以下の 3 つのプロセスで構成されている．

(1) 入力変数と出力変数の決定
(2) 入力変数の値と出力変数の値の収集
(3) テストメソッドの作成

以下，それぞれのプロセスについて，2 章の例を用いて詳細を説明する．

3.1 入力変数と出力変数の決定

Java ソースコード（プログラム）のメソッドに対する単体テストでは，テスト対象のメソッドを実行するために

必要な入力とメソッドを実行した後に得られる期待値を決定する必要がある．JUnit では，入力値を用いてテスト対象のメソッドを呼び出すコードと，実際にメソッドを呼び出した後の出力値と期待値を比較するコードを，`@Test` アノテーションが割り当てられたテストメソッドに記述する（図 2 を参照）．

本手法では，ExtractMethod の適用により新規に抽出したメソッド m_n のソースコードを静的に解析することで，m_n に対する入力変数と出力変数を決定する．その際，Java における変数の型には，基本型とユーザ定義型があるため，それらの違いを考慮する必要がある．

入力変数とは，m_n の本体内部で値が使用される可能性のある以下の変数を指す．

- m_n の仮引数（ローカル変数）
- m_n の本体内部に記述されている
  - 代入文の右辺や式（代入式の右辺やそれ以外の式）に現れるフィールド変数
  - メソッド呼び出しの実引数に現れるフィールド変数

入力変数については，基本型とユーザ定義型の区別は不要である．

出力変数とは，m_n の本体内部で値が変更される可能性のある以下の変数を指す．

- m_n の本体内部に記述されている
  - `return` 文に現れるローカル変数やフィールド変数
  - 代入文（式）の左辺に現れるフィールド変数
  - メソッド呼び出しの実引数に現れるフィールド変数（ユーザ定義型のみ）

ここで，メソッド呼び出しの実引数に現れるフィールド変数に対して，ユーザ定義型のみ出力変数に含まれる理由を述べる．いま，メソッド呼び出しの実引数が基本型であれば，引数の値はコピーされて渡されるため，呼び出し先で引数の値が書き換えられることはない．一方，実引数がユーザ定義型の場合は，引数の値としてインスタンスの参照が渡されることになるため，呼び出し先でインスタンスの内容が書き換えられる可能性がある．

入力変数と出力変数は，m_n に対して構文解析を適用するだけで決定可能である．例として，ExtractMethod の適用により新規に抽出されたメソッド図 3 の `getOutstanding()` を取り上げ，その入力変数と出力変数を示す．このメソッドの仮引数は `OrderRcord` 型（ユーザ定義型）の変数 `record` である．また，このメソッドの本体内部で値が使用されているフィールド変数は `int` 型（基本型）の `discount` のみである．また，`getOutstanding()` の戻り値を返す `return` 文に現れるローカル変数は `outstanding` である．代入文の左辺に現れるフィールド変数およびメソッド呼び出しの実引数として現れるユーザ定義型のフィールド変数は存在しない（`diccount` がメソッド呼び出しの実引数に現れるが，それは基本型である）．以上より，`getOutstanding()` に対する入力変数の集合 V_{in} と出力変数の集合 V_{out} は次のようになる．

$V_{in} = \{$ `record`, `discount` $\}$

$V_{out} = \{$ `outstanding` $\}$

3.2 入力変数の値と出力変数の値の収集

ExtractMethod の適用により新規に抽出されたメソッド m_n のテストケースを生成するためには，入力変数の値と出力変数の値を用意すればよい．本手法では，ExtractMethod の適用対象であるもとのメソッド m_s にテストケースがあらかじめ用意されていることを利用して，m_n のテストケースを生成する際に必要な入力変数の値と出力変数の値を収集する．そのために，変数の値を記録するログコードを m_n の本体内部に挿入する．入力変数の値を記録するコードは，m_n が呼ばれる直後（m_n の先頭）に挿入する．また，出力変数の値を記録するログコードは，m_s の本体内部に記述されている `return` 文の実行の直前と m_n の最後に挿入する．

ここで，`return` 文における戻り値は式により表現されているため，“`return x;`” のように常に単一の変数の値が返るわけではない．たとえば，“`return x + 1;`” のように戻り値が計算式で表現されている場合も考えられる．そこで，本手法では，`return` 文における計算式の値を一時変数に代入するコードを挿入し，もとの `return` 文を一時変数の値を返す `return` 文に置き換える．たとえば，以下のような置き換えが実施される．

```
return x + 1; →
_rval1 = x + 1; return _rval1;
```

その上で，`return` 文の実行の直前に一時変数の値を記録するログコードを挿入する．

次に，入力変数の値と出力変数の値を具体的に取得する方法を述べる．Java において，基本型の変数の値はプログラムを実行することで容易に取得することができる．ここで，クラス `java.lang.String` や `java.lang.Integer` はユーザ定義型であるが，それらは不変である（それらのインスタンスの状態を変更することはできない）．さらに，それらに格納される値は文字列リテラルや数値リテラルと直接比較できる．このため，本手法ではこのようなクラスを基本型として扱う．

ユーザ定義型の変数の値はインスタンスへの参照であるため，その値を直接（あるいは直列化して）取得しても，それをテストケースにそのまま記述できるわけではない．そこで，ユーザ定義型の変数を扱う際には，Java のリフレクション API を利用する．具体的には，ユーザ定義型の変数に格納されているインスタンス（のクラス）で定義されているフィールド変数に対応する（クラス `java.lang.reflect.Field` の）インスタンスの一覧を取得し，それぞれのフィールドの値を収集する．もし取得し

たフィールド変数のインスタンスが再びユーザ定義型であった場合，リフレクションの適用とフィールドの値の取得を繰り返し試みる．理想的には，インスタンス変数の参照関係に循環がなければ，インスタンス変数の参照はすべて基本型のインスタンス変数に（インライン）展開可能であり，基本型の変数の値として取得できるはずである．しかしながら，インスタンスの参照関係をより深い階層までたどったとしても，その値をテストの実行に利用するとは考えにくい．

そこで，本手法では，入力変数と出力変数がユーザ定義型である場合，次のように対応する．

入力変数： テスト対象のメソッドを実行するためには，すべての入力変数の値が適切に設定されている必要がある．しかしながら，すべてのユーザ定義型の変数を基本型の変数に展開することは現実的でない．そのため，ユーザ定義型の入力変数については，その値を収集対象とはしない．その代わりに，ExtractMethod の適用前のメソッドのコード断片をテストケースに埋め込むことで，入力変数の値を実行時に設定する．具体的な方法については，3.3 節で述べる．

出力変数： 取得するインスタンスの参照関係の深さに上限値を設定する．上限値を越えるインスタンスの参照は，取得する変数として扱わない．たとえば．上限値を 0 とすると，ユーザ定義型の出力変数は収集対象から排除する．上限値が 1 の場合は，ユーザ定義型の変数に格納されたインスタンスへの参照を一つたどり，そのインスタンス（のクラス）で定義されている基本型の変数の値のみを収集対象とする．

3.1 節で述べたように，図 3 の `getOutstanding()` に対する入力変数の集合 V_{in} と出力変数の集合 V_{out} は次のようになっている．

$V_{in} =$ { `record`, `discount` }

$V_{out} =$ { `outstanding` }

ここで，`record` はユーザ定義型の変数であるため，値の収集対象から除外する．一方，`discount` と `outstanding` は基本型の変数である．以上より，図 3 の `getOutstanding()` に対して，値を収集する入力変数の集合 W_{in} と出力変数の集合 W_{out} は次のようになる．

$W_{in} =$ { `discount` }

$W_{out} =$ { `outstanding` }

この例では，`outstainding` が基本型であるが，もし V_{out} にユーザ定義型 C の変数 c が含まれていたとすると，C で定義されているフィールド変数の値も取得対象となる．たとえば，クラス `C` に 2 つのフィールド変数（`java.lang.String` 型の `s` と `int` 型の `i`）が含まれていたとすると，`f.s` と `f.i` が値の取得対象となる．

```
public class Customer {
  Logger logger =
    Logger.getInstance(Customer.class.getName());
  ...

  private int getOutstanding(OrderRecord record) {
    logger.log(Level.INFO, discount);
    int outstanding = 0;
    for (Order order : record.orders) {
      outstanding += order.getAmount(discount);
    }
    int _rval1 = outstanding;
    logger.log(Level.INFO, _rval1);
    return _rval1;
  }
}
```

図 6 入力変数の値と出力変数の値を収集するソースコード

実際に変数の値を収集する際には，log4j[*2] のクラス `org.apache.log4j.Logger` を利用する．ログコードを埋め込んだソースコードを**図 6** に示す．このソースコードを用いて，図 2 に示す 3 つのテストケースを実行した際に収集された値は次のようになる．

T1 `discount`:10, `outstanding`:180

T2 `discount`:20, `outstanding`:240

T3 `discount`:—, `outstanding`:—

T3 のテスト実行では，`getOutStanding()` が呼ばれないため，変数の値は収集されない．よって，**T1** と **T2** において収集した値が，本手法により自動生成されるテストコードに埋め込まれる．

3.3 テストメソッドの作成

ExtractMethod の適用により新規に抽出されたメソッド m_n を呼び出すメソッド m_r に対するテスト実行において，収集した入力変数の値と出力変数の値を用いて，m_r に対するテストメソッド t_r から，m_n に対するテストメソッド t_n を作成する．その際，基本型の入力変数については，t_n の本体内部において m_n を呼び出す直前にその値を設定する．

一方，ユーザ定義型の入力変数については，t_r と m_r に含まれるコード断片を再利用して，その値を設定する．そのために，ExtractMethod 適用後のソースコードに対してプログラム依存グラフ [6], [7]（クラス依存グラフ [8]）を構築し，ユーザ定義型の入力変数に対するプログラムスライス [9] を抽出する．ユーザ定義型の入力変数が複数存在する場合は，スライスを結合する．

図 3 に示すソースコードにおける `statement()` および `getOutstanding()`，図 2 に示すソースコードにおける `testStatement1()` から構築したプログラム依存グラフを**図 7** に示す．実線はデータ依存関係，破線は制御依存関係を表す．いま，テスト対象のメソッド `getOutstanding()`

[*2] https://logging.apache.org/log4j/2.x/

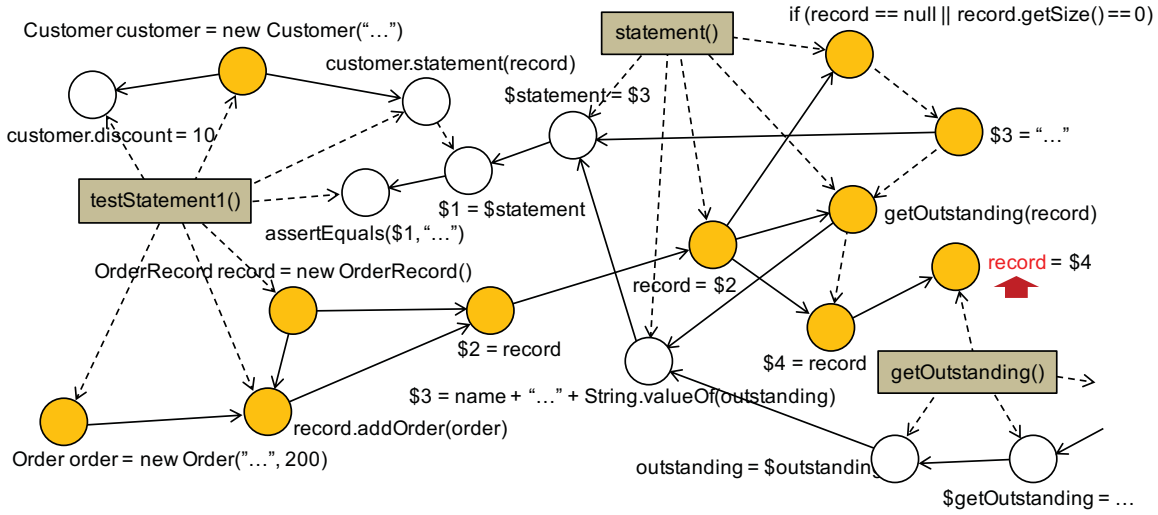

図 7　ExtractMethod 適用後のソースコードに対するプログラム依存グラフ

の仮引数 record（図 7 の矢印）を基準をしてプログラムスライスを抽出すると，黄色の節点に対応する文や式が残る．

最終的に，スライスとして抽出したコード断片において，メソッド呼び出しをインライン展開する．また，コード断片に return 文が現れるときは，その戻り値の型を void に変換する．これで，入力変数に関するテストコードの生成が完了する．このように生成したコードを m_n に対するテストメソッドの最初（入口の直後）に挿入すればよい．

次に，出力変数に関するテストコードの生成について述べる．出力変数はすべて基本型であるため，m_n の呼び出しに対する戻り値と単純に比較すればよい．そのため，m_n の戻り値を格納する一時変数を用意し，その変数の値と出力変数の値を比較するコード断片を挿入する．ここで，図 3 に示すソースコードにおいて，**T1** を用いたテスト実行の際の getOutstanding() の戻り値 outstanding は 180 である．よって，getOutstanding() の戻り値の値を格納する一時変数を o とすると，出力変数に関するテストコードは以下のようになる．

```
int o = customer.getOutstanding(record);
assertEquals(o, 180);
```

このコードをテストメソッドの最後（出口の直前）に挿入すればよい．

このようにして作成したテストコードを図 **8** に示す．statement() の一部がインライン展開されていることが分かる．今後，getOutStainding() に対してリファクタリングを適用する際には，このテストコードが利用できる．

4. おわりに

本論文では，ExtractMethod の適用において，新規に抽出したメソッドに対するテストケースが不在であることに起因する問題を示した．さらに，この問題を解決するために，ExtractMethod の適用対象のメソッドに対す

```
public class CustomerTest2 {

  @Test
  public void testStatement1() {
    Customer customer = new Customer("C1");

    OrderRecord record = new OrderRecord();
    Order order = new Order("Movie", 200);
    record.addOrder(order);

    if (record == null || record.getSize() == 0) {
      return;
    }

    customer.discount = 10;

    int o = customer.getOutstanding(record);
    assertEquals(o, 180);
  }

  @Test
  public void testStatement2() {
    Customer customer = new Customer("C2");

    OrderRecord record = new OrderRecord();
    Order movie = new Order("Movie", 200);
    Order music = new Order("Music", 100);
    record.addOrder(movie);
    record.addOrder(music);

    if (record == null || record.getSize() == 0) {
      return;
    }

    customer.discount = 20;

    int o = customer.getOutstanding(record);
    assertEquals(o, 240);
  }
}
```

図 8　getOutstanding() に対するテストコード

るテストケースを利用することで，新規に抽出したメソッドに対するテストケースを自動生成する手法を提案した．本手法を ExtractMethod の適用に導入することで，開発者の負担軽減が期待できる．

本手法にはさらなる改善の余地がある．3 章に示したように，本手法では実際にテストを実行し，その実行結果に基づき，テスト対象のメソッドの入力と出力を決定している．この方法では，テスト対象メソッドが実行されない場合には，テストケースは生成できない．つまり，本手法では，2 章で述べた問題の一部しか解決していない．具体的には，`statement()` のテストケース **T3** に該当するテストケースが生成されないため，図 5 に示す改変に対して，`getOutstanding()` の外部的振る舞いが保存されているのかどうかを検査することはできない．これを解決するため，`statement()` の内部コードに対して制御フローを解析し，`getOutstanding()` の呼び出しを強制するようなコードを作成することが考えられる．

また，従来のテストケース生成手法と組み合わせることで，テストケースの不足を補うことも考えられる．さらに，本手法において，テストコードの作成プロセスについても十分に洗練されているとはいえない．たとえば，プログラムスライスに含まれるコード断片に `this` が現れる際には適切なインスタンスに変換する必要があるが，現時点は未対応である．このように，テストコードの作成アルゴリズムは場当たり的であり，テストコードが必ずしも正しいことが保証できていない．本手法の実装に取りかかるのと同時に，手法の正しさに関する検証も必須である．また，本論文で示したような例題ではなく，実際のテストケースを用いた実験も今後の課題である．

参考文献

[1] Opdyke, W. F.: Refactoring Object-Oriented Frameworks, Technical report, Ph.D. thesis, University of Illinois, Urbana-Champaign (1992).
[2] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999).
[3] Hafiz, M. and Overbey, J.: Refactoring Myths, *IEEE Software*, Vol. 32, No. 6, pp. 39–43 (2015).
[4] Mens, T. and Tourwé, T.: A Survey of Software Refactoring, *IEEE TSE*, Vol. 30, No. 2, pp. 126–139 (2004).
[5] Passier, H., , Bijlsma, L. and Bockisch, C.: Maintaining Unit Tests During Refactoring, *Proc. PPPJ '16*, pp. 18:1–18:6 (2016).
[6] Ottenstein, K. J. and Ottenstein, L. M.: The Program Dependence Graph in a Software Development Environment, *ACM SIGPLAN Notices*, Vol. 19, No. 5, pp. 177–184 (1984).
[7] Horwitz, S., Ball, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs, *ACM TOPLAS*, Vol. 12, No. 1, pp. 26–60 (1990).
[8] Larsen, L. and Harrold, M. J.: Slicing Object-Oriented Software, *Proc. ICSE '96*, pp. 495–505 (1996).
[9] Weiser, M.: Program Slicing, *IEEE ACM TSE*, Vol. 10, No. 4, pp. 352–357 (1984).