情報処理学会論文誌　プログラミング　Vol.8　No.4　1–14 (Dec. 2015)

単体テストを用いたチュートリアルの自動生成手法

三上 裕明[1,a)]　坂本 大介[1]　五十嵐 健夫[1]

受付日 2015年4月3日, 採録日 2015年7月28日

概要：API の理解と使い方の学習は，プログラマにとって重要であるが，時間のかかる作業である．API を理解するための方法の 1 つとして，チュートリアルがある．チュートリアルは，複数のサンプルコードおよびその説明と，それらのリストからなるドキュメントであり，ライブラリの基本的な機能の使い方を提示する．現在，チュートリアルはライブラリの開発者が記述しているが，チュートリアルを書くことは手間がかかる作業である．そのため，チュートリアルが更新されず，最新の API の使い方がチュートリアルで説明されていない場合がある．この問題に対処するため，本論文では，単体テストからチュートリアルを自動的に生成する手法を提案する．本手法では最初に，単体テストから実行可能なサンプルを生成する．次に，サンプルの説明を生成するために，プログラム可視化の技法を用いる．また，サンプルコードのリストを作るために，単体テスト間の依存関係を用いる．具体的には，実行時情報を用いて単体テストの依存関係を抽出し，そしてその依存関係をもとにサンプルコードを順序付けする．本手法の有効性を確認するためにユーザスタディを実施した．この結果，本手法を用いて生成されたチュートリアルを用いることで，自動生成された既存の API ドキュメントの場合よりも API を効率的に理解できるという結果が得られた．

キーワード：開発支援，ドキュメント，チュートリアル，単体テスト

Automatic Generation of Tutorial from Unit Tests

Hiroaki Mikami[1,a)]　Daisuke Sakamoto[1]　Takeo Igarashi[1]

Received: April 3, 2015, Accepted: July 28, 2015

Abstract: Understanding and learning the usage of the APIs are important for programmers, but those are time-consuming tasks. A tutorial is one way to understand the usage of the APIs. It is a document composed of some sample codes, their explanations and a list of sample codes, and describes the basic usage of the library. Tutorials are usually written by library-developers; however, writing a tutorial is a tedious work. Therefore, there are cases that how to use the latest APIs is not described in a tutorial because developers do not update the tutorial. To deal with this issue, we present a method that automatically generates a tutorial from unit tests. This method first generates executable sample codes from unit tests. It then uses program visualization technique to explain the sample codes. Furthermore, it uses the dependencies between tests to make a list of sample codes: it extracts the dependencies between tests by using run-time information, and generates the order of the sample codes by using those dependencies. The results of a user study showed that tutorials generated by this method are more effective in helping programmers learn APIs than the existing auto-generated document.

Keywords: development support, documentation, tutorial, unit-test

1. はじめに

近年のソフトウェア開発では，ソースコードの質の向上や開発コストの低減のために，第三者が提供するライブラリ（オープンソースライブラリなど）が多く用いられている [9], [11]．特にこのようなライブラリなしで，現代的なソフトウェアを開発することはきわめて難しい．しかし，ライブラリの API（Application Programming Interface）は複雑であるため，API の使い方を理解し，実際に使用することは，開発者にとって難しく，時間のかかる作業である [22], [23], [25]．

[1] 東京大学大学院
The University of Tokyo, Bunkyo, Tokyo 111–0033, Japan
[a)] mhiroaki@is.s.u-tokyo.ac.jp

API の理解を助けるために，ほとんどのライブラリはドキュメントを提供している．ドキュメントは API の理解に役立つ一方で，その記述や保守はきわめて難しいことが過去の研究で示されている．そのため，多くのドキュメントは現状に則さない，間違いの多いものとなってしまう [7], [8].

多くの研究で，コードサンプルは API の理解，使用に重要であることが示されており [22], [23], [25]，多くの自動コードサンプル生成手法が提案されている [4], [13], [17], [28], [30]．これらの手法では，生成したサンプルを Javadoc などの API ドキュメントか，コード補間システムの中で用いている．たとえば，*eXoaDocs* [17] は Javadoc にコードサンプルを追加することを目的としている．また，MAPO [28] は，ライブラリを使用しているレポジトリを解析し，コードスニペットを補間するツールである．しかし，API ドキュメントやコード補間システムは次のような問題を抱えている．

- どの要素（関数，メソッド，クラスなど）が重要か判断することが難しい [25],
- 複雑な API の使用方法を理解するには適さない [7].

このような API ドキュメントの問題を補うために，いくつかのライブラリではチュートリアルと呼ばれるドキュメントを提供している．チュートリアルは，サンプルコードを用いたライブラリの説明のリストからなる．リストの順序は，アルファベット順のような機械的順序でなく，順番に読めば開発者が API を理解できるようになっている．チュートリアルを用いることによって，開発者は API の基本的な使い方を理解できる [7].

一方で，チュートリアルはライブラリ開発者が記述しなければならず，チュートリアルの記述は API ドキュメントの記述よりも難しい [7]．そのため，チュートリアルは，API ドキュメントよりも実際のプログラムと乖離しやすい．たとえば，チュートリアルで用いられているコードサンプルが最新の API ではコンパイルできない，といった問題が起きる可能性がある．チュートリアルで用いるコードサンプルの作成を支援する技術 [1], [14] は存在するが，ライブラリ開発者がチュートリアルで用いられるすべてのコードサンプルを書かなければならないことには変わりがない．この問題に対処するため，本論文ではチュートリアルを自動生成する手法を提案する．この手法は単体テスト，プログラム可視化，テスト間の依存関係に関連している．

アジャイル開発の普及とともに，単体テストは近年一般的な手法となっている [3]．テストコードからコードサンプルを生成することにはいくつかの利点がある [13]．まず，テストコードは実行可能であり，単体で完結している．また，ライブラリ開発者によって記述されているため，テストコードは信頼性が高く最新の API が用いられている．プログラム可視化は，プログラマにプログラムの実行時の振舞いを示す技術である．この技術は，ソースコード（プログラムの静的な面）とプログラムの実行過程（動的な面）の関係を理解するのに効果的である [5], [15]．テスト間の依存関係は，テストのデバッグ，保守に関連した概念である．テスト間に依存関係が存在すると，小さなソフトウェアのバグがドミノ倒しのように多くのテストの失敗を引き起こす．そのため，テストコードには依存関係がないことが望ましいが，完全に依存関係をなくすことはできない [10], [18].

本論文で提案する手法は 4 つのモジュールからなる．1）テストコード解析モジュールは，他のモジュールの前処理として，テストコードを解析しテストプログラムを分類する．2）コードサンプル抽出モジュールは単体テストから実行可能なコードサンプルを生成し，それらをクラスタリングする．3）説明生成モジュールはコードサンプルをプログラム可視化技術を用いて説明した HTML ページを生成する．最後に，4）チュートリアル生成モジュールは，テスト間の依存関係を用いてコードサンプルの順序関係を作り，これを用いてチュートリアルを生成する．特に，テスト間の依存関係を用い順序関係を作るアルゴリズムは，本論文の最も大きな提案であり，本研究の主たる貢献である．

提案手法の有用性を評価するためにユーザスタディを行った．ユーザスタディでは，Javadoc を用いるよりも提案手法によるチュートリアルを用いたほうが，開発者はより多くのプログラミングタスクを遂行することが確認された．これらについて報告する．

2. 関連研究

2.1 API 理解の困難性

API を学ぶことの難しさに関する多くの研究 [22], [23], [25] がなされている．Robillard ら [22], [23] はドキュメントを分析して，コードサンプルが API の理解に重要であることを示した．Scaffidi [25] は，ドキュメントの問題点について調査し，チュートリアルと API ドキュメントそれぞれの長所，短所を示した．

Dagenais ら [7], [8] は，オープンソースプロジェクトのドキュメントと，それらの保守方法を分析した．そして，チュートリアルは API ドキュメントよりも保守が難しいこと，チュートリアルは API ドキュメントよりもフレームワークに対するドキュメントとして適していることを示した．

2.2 API 理解を支援する手法

コードサンプルを自動生成する手法はいくつか提案されている．Kim ら [17] は，*eXoaDocs* という，ライブラリを使用しているレポジトリからコードサンプルを抽出するシステムを提案した．Xie ら [28] は，メソッド名，クラス名，パッケージ名を入力として受け取り，メソッドの呼び出し

列をオープンソースレポジトリから抽出するフレームワークを提案した．Nasehi ら [20] は，単体テストからコードサンプルを抽出する利点を示した．また，API の有用性を高めるために，単体テストからコードサンプルを抽出し，それらを API ドキュメントに追加する手法の概略を提案した．Zhu ら [29], [30] は，テストコードからコードサンプルを抽出するアルゴリズムを提案した．このアルゴリズムは，テストシナリオとテストコードのパターンを用いている．テストシナリオとは，ある API の使用方法を示したテストコードの部分プログラムのことである．

コードサンプルを用いたコード補間システムもいくつか提案されている．Mandelin ら [19] は，ある型からある型を得る方法を示したコードサンプルを，ジャンゴロイドと呼び，API の型情報とサンプルコードを用いてジャンゴロイドを生成するアルゴリズムを提案した．Nguen ら [21] が提案した GraPacc は，ユーザが編集中のコードをもとに，適切なコードパターンを発見し，そのパターンをコード補間に用いるツールである．

2.3　ドキュメントの保守手法

ドキュメントの保守に関わる既存研究，既存手法としては，次のようなものが存在する．

Ginosar ら [14] は，リビジョン管理アルゴリズムを用いることで，多段階のコードサンプルの記述を支援するインタフェースを提案した．このインタフェースを用いることで，記述者は，ある段階の変更を他の段階にも反映することができる．Cumiki [1] は，ソースコードに注釈をつけることで，プログラムを理解しやすくする Web サービスである．

CommentWeaver [16] は，開発者が書く必要のあるコメントを減らすための Javadoc の拡張である．Python の doctest モジュール [2] は，docstring（ドキュメントとして扱われるプログラム中の文字列リテラル）から，Python のプログラムとその想定される出力を抽出する．このモジュールは，docstring とプログラムの間に矛盾がないかをチェックするためや，実行可能な例が含まれたドキュメントを書くために用いることができる．

2.4　プログラム可視化技術

デバッグや教育を目的とした，プログラムの動的な振舞いを可視化するツールは複数提案されている．jGRASP [6] は，教育目的に開発された Java の軽量 IDE（Integrated Development Environment; 統合開発環境）であり，データ構造を可視化することができる．Rozenberg ら [24] が提案した Vebugger は，選択されたオブジェクトを HTML と CSS で記述されたテンプレートを用いて可視化することができるデバッガである．Online Python Tutor [15] は，教育を目的とした Web 上で動くプログラム可視化ツールである．ソフトウェアのインストールや設定が必要ないため，初心者には Online Python Tutor は使いやすい．

2.5　テスト間の依存関係

Van ら [26] は，単体テストの可読性と保守性の向上のために，テスト間の依存関係を削減するテストコードのリファクタリング手法を提案した．Gaelli ら [10] は，84%から 96%のテストコードは依存関係を持っていることを 4 つのケーススタディによって示した．

JExample [18] は，依存関係を用いてテストコードのデバッグを容易にする JUnit [*1] の拡張である．プログラマは `@Depends` というアノテーションをつけることで，依存関係を示し，テストケースを再利用することができる．Gaelli ら [12] は，カバレッジ集合（テストケースから呼び出されたメソッドの集合）を用いて依存関係を定義し，XUnit フレームワークを用いて書かれたテストコードから依存関係を抽出するアルゴリズムを提案した．

3.　提案手法

本章では，チュートリアルを自動生成する手法を説明する．**図 1** は，提案手法の概略を示している．最初に，テストコード解析モジュールは，ライブラリのソースコードとテストコードを入力として受け取り，単体テストの実行時情報を用いて，テストプログラムを 3 つのグループに分類する．次に，コードサンプル抽出モジュールは，3 つのグループを入力としてコードサンプルを生成し，それらをクラスタリングする．説明生成モジュールは，抽出されたサンプルを受け取り，コードサンプルとその実行時の振舞いを可視化する HTML ページを生成する．最後に，チュートリアル生成モジュールは，HTML ページとテストコードの 3 つのグループを受け取り，テスト間の依存関係を抽出することでライブラリのチュートリアルを生成する．

本論文において開発されたプロトタイプは，Java で書かれ，JUnit4 を用いてテストされたライブラリを対象としたものである．しかし，本手法そのものは，クラスを用いた一般的なオブジェクト指向言語と，一般的なテスティングフレームワークを用いてテストされたライブラリを対象とする．

3.1　テストコード解析モジュール

このモジュールでは，単体テストの実行時情報を解析することで，テストコードを次のグループに分類する．

- メインメソッドの一部として動作するプログラム，
- メインクラスのメンバとして動作するプログラム，
- スタブやモックオブジェクトのような代用プログラム．

以後，最初のグループをメイン，2 番目のグループをメン

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

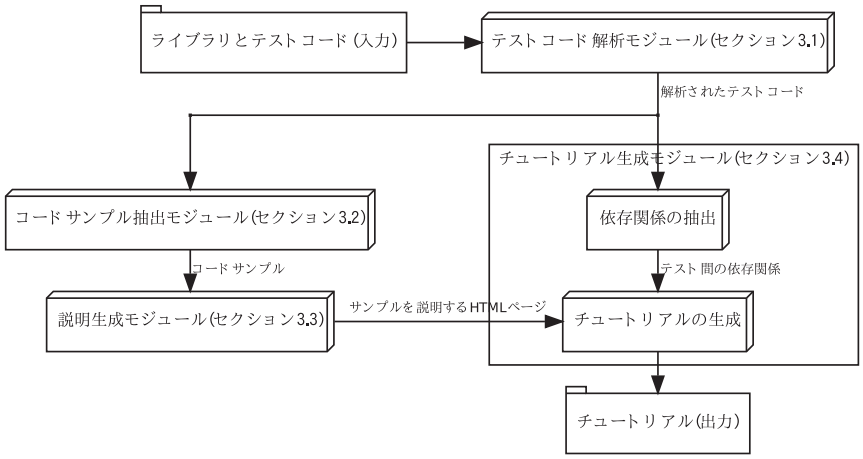

図 1　提案手法の概要
Fig. 1　An overview of the proposed method.

バ，3 番目のグループをスタブと呼ぶ．

メイン，メンバ，スタブを，**図 2** のプログラムを例として説明する．このプログラムは `java.io.PriorityQueue` クラスの簡単なテストであり，JUnit4 の典型的な使用例でもある．`testAdd` テストケースを実行するためには，`setup`，`testAdd` メソッドを順々に実行する必要がある．この 2 つのメソッドは，テストケースにおいてメインメソッドのような働きをしている．したがって，これらはメインに属する．`checkQueue` メソッドはキューに関する一般的なアサーションメソッドであり，メインメソッドの一部とは考えにくい．したがって，このメソッドはメインではなくメンバに属する．最後に，`MockComparator.java` で宣言されている `MockComparator` クラスは，モッククラスであるので，スタブに属する．

テストコードを 3 つのリストに分類するために，本論文では，Java や C++では `this` と呼ばれる，現在実行中のオブジェクトに着目した．そして，メイン，メンバ，スタブを次のように定義した．

メインは，次の 2 つの条件のうち，少なくとも 1 つを満たす要素（メソッド，フィールド，コンストラクタなど）のリストである．

- テストケースで最初に呼び出されたメソッドである．
- 自身の `this` と，最初のメソッド呼び出しの `this` が等しく，呼び出し元のメソッドが存在しない．

メンバは，次の 2 つの条件のうち，少なくとも 1 つを満たす要素のリストである．

- 自身の `this` とメインに属する要素の `this` が等しく，呼び出し元のメソッドが存在する．
- static な要素であり，メインのいずれかが宣言されたクラスで宣言されている．

最後に，スタブは，次の条件をすべて満たす要素のリストである．

- テストコードで宣言されている．
- テストケースで一度は呼び出された．
- メインにもメンバにも属さない．

提案手法では，テストケースの実行中におけるメソッド呼び出しやフィールドアクセスの情報を取得し，各テストケースをこの定義に従って分析する．

図 2 の `testAdd` テストケースを，このアルゴリズムの入力例として考える．この例では，提案手法は，テストケースを実行しながら次のように解析する．

1　まず，`setup` が呼び出される．メインの定義から，これをメインに追加する．
2　`MockComparator` のコンストラクタが呼び出される．コンストラクタの `this` と，1 の `this` が異なり，テストコードでコンストラクタが宣言されているため，`MockComparator` のコンストラクタをスタブに追加する．
3　`PriorityQueueTest<String>` のコンストラクタが呼び出される．これはテストコードで宣言されていないので無視する．
4　`queue` フィールドへのアクセスが発生する．フィールドの `this` と 1 の `this` が等しく，このフィールドアクセスは `setup` メソッドという呼び出し元を持つため，`queue` フィールドをメンバに追加する．
5　同様に実行が終了するまでテストケースを分析する．

最終的なアルゴリズムの出力は，以下のとおりとなる．

メインの要素　`setup`，`testPush`
メンバの要素　`queue`，`checkStack`
スタブの要素　`MockComparator` のコンストラクタ

3.2　コードサンプル抽出モジュール

このモジュールでは，まず 3.1 節で説明した 3 つのグループを用いて実行可能なコードサンプルを生成する．次に，コードサンプルをクラスタリングし，それぞれのクラスタに対し代表的なコードサンプルを選択する．

MockComparator.java

```
package test;
import java.util.Comparator;

public class MockComparator extends Comparator<
    String> {
    public int compare(String e1, String e2) {
        ...
    }
}
```

PriorityQueueTest.java

```
package test;
import static org.junit.Assert.*;
import org.junit.*;
import java.util.PriorityQueue;

public class PriorityQueueTest {
    protected PriorityQueue<String> queue;
    public static void checkQueue(PriorityQueue<String>
        queue, int size, String front) {
        assertEquals(size, queue.size());
        assertEquals(front, queue.element());
    }

    @Before public void setup() {
        queue = new PriorityQueue<String>(11, new
            MockComparator());
    }

    @Test public void testAdd() {
        this.queue.add("Foo");
        this.queue.add("Bar");
        checkQueue(queue, 2, "Bar");
    }

    @Test public void testRemove() {
        ...
    }
}
```

図 2　典型的な JUnit4 を利用したテストケース
Fig. 2　A typical example of test case using JUnit4.

MockComparator.java

```
package test;
import java.util.Comparator;

public class MockComparator extends Comparator<
    String> {
    public int compare(String e1, String e2) {
        ...
    }
}
```

GeneratedSample.java

```
package test;
import static org.junit.Assert.*;
import org.junit.*;
import java.util.PriorityQueue;

public class PriorityQueueTest {
    protected PriorityQueue<String> queue;
    public static void checkQueue(PriorityQueue<String>
        queue, int size, String front) {
        assertEquals(size, queue.size());
        assertEquals(front, queue.element());
    }

    public void main() {
        {
            queue =
                new PriorityQueue<String>(11, new
                    MockComparator());
        }
        {
            this.queue.add("Foo");
            this.queue.add("Bar");
            checkQueue(queue, 2, "Bar");
        }
    }
}
```

図 3　図 2 中の `testAdd` テストケースから抽出した実行可能なコードサンプル
Fig. 3　The executable code example that is extracted from `testAdd` test case in Fig. 2.

3.2.1　コードサンプルの生成

実行可能なサンプルを生成するために，メインの要素を実行順に並べてメインメソッドを作る．そして，それとメンバの要素を含むクラスをメインクラスとして生成する．このとき，テストケース本体が宣言されているクラスの名前をメインクラスの名前とする．最後に，スタブが宣言されているファイル全体をサンプルに追加する．

再び，図 2 中の `testAdd` テストケースを入力例として考える．`org.junit.Test` アノテーションがついており，メインに属することから，テストケースの本体は，`PriorityQueueTest.testAdd` メソッドである．したがって，メインクラスの名前は，`PriorityQueueTest` となる．`queue` フィールドと，`checkStack` メソッドはメンバに属しているので，これらは `PriorityQueueTest` のメンバとして宣言される．次に，`setup` と `testAdd` メソッドを並べて，メインメソッドを生成する．メインとメンバの要素は `PriorityQueueTest` で宣言されているので，パッケージ宣言と import 文を `PriorityQueueTest.java` から取得し，これをサンプルに追加する．最後に，スタブの要素は，`MockComparator.java` で宣言されているので，`MockComparator.java` をサンプルに追加する．

最終的に，提案アルゴリズムは，図 2 中の `testAdd` テストケースから，**図 3** のようなサンプルを生成する．

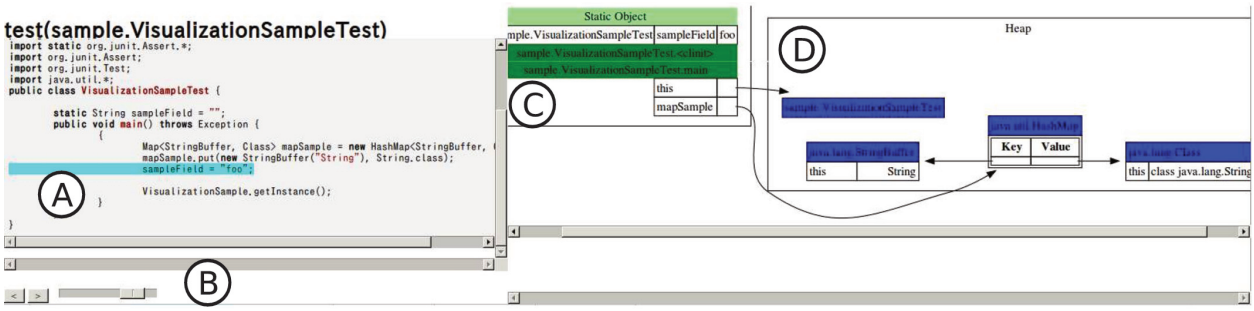

図 5　本手法で生成される HTML ページ．A) サンプルコード，B) ステップ実行するためのスライダとボタン，C) スタックを可視化する領域，D) ヒープを可視化する領域

Fig. 5　A html page generated by this method. A) the sample code, B) the buttons and the slider that step forward and backward, C) the area visualizing stack and D) the area visualizing heap.

```
public void testCase1() {
    Stack<int> stack = new Stack<int>();
    for (int i = i; i < 10; i++) {
        stack.push(i);
    }
    assertEquals(9, stack.lastElement());
}
public void testCase2() {
    Stack<int> stack = new Stack<int>();
    for (int i = i; i < 6; i++) {
        stack.push(i);
    }
    assertEquals(5, stack.lastElement());
}
```

図 4　同じ API 使用例を示しているが，類似度が 1 ではない例

Fig. 4　An example that the similarity is not 1 but the programs explain completely same API usage.

3.2.2　似通ったコードサンプルのクラスタリング

テストコードには，しばしば似通ったテストケースが存在するので，そのようなサンプルをまとめる必要がある．本手法では，そのためのアルゴリズムは，文献 [30] で提案されたアルゴリズムをもととした．このアルゴリズムでは，サンプル間の類似度を用いてこれをクラスタリングし，それぞれのクラスタに対して代表的なサンプルを 1 つ選択する．

類似度は，サンプルのメソッドの呼び出し列を用いて定義される．本論文では，メソッド呼び出し列は，メインまたはメンバから呼び出されており，テストコードで宣言されていないメソッドの列とした．しかし，この定義には問題がある．たとえば，定義によれば，図 4 の 2 テストケースの間の類似度は，0.75 となる．しかし，これらのテストケースが示す API の使用例はまったく同じである．これを解決するために，メソッド呼び出し列から式 (1) で定義された $s[i,j]$ をすべて削除することで，連続した重複メソッド呼び出し列を削除する．

$$s[i,j] = s[j+1, 2j-i+1] \tag{1}$$

ここで，$s[i,j]$ は，i 番目から j 番目までのメソッド呼び出し列の部分列を表す．

3.3　説明生成モジュール

このモジュールでは，プログラム可視化技術を使い，コードサンプルの説明を生成する．プログラム可視化は，Online Python Tutor [15] で用いられた技術を基礎とした．元手法よりも扱うサンプルが複雑になるので，これに対処するため，オブジェクトのエンコード方法を 2 つの場合において変更した．

まず，もしインスタンスが説明対象のライブラリで宣言されていないのであれば，`toString` メソッドを用いてエンコードする．これは，API を理解するのに重要でない情報を小さくするためである．

次に，もしインスタンスが `java.util` パッケージの `Map`，`List`，`Set`，`Array` のいずれかに属する場合，専用のテンプレートを用いてエンコードする．これは，マップ，リスト，集合のような Java のデータ構造は，実際には複雑なクラスであり単純にエンコードすると見づらくなるためである．

図 5 に，本手法で生成される HTML ページの例を示す．

3.4　チュートリアル生成モジュール

このモジュールでは，テスト間の依存関係を抽出し，これを用いてサンプルコードのリストを作る．そして，特定の API の使用方法を説明するために，それぞれの API に対して同様にリストを作る．

3.4.1　テスト対象メソッドの識別

依存関係の抽出において用いるために，テスト対象のメソッドを識別する．本論文で用いるアルゴリズムは，文献 [13] で提案されたスライシングを用いるアルゴリズムと，文献 [30] で提案されたテストケース名を用いるアルゴリズムの 2 つを基礎としている．

本アルゴリズムでは，まず，特定のテストケースのメインまたはメンバから呼び出されており，テストコードで宣言されていないメソッドを探し，それらをテスト対象メソッドの候補とする．次に，スライシングによるアルゴリズムの適用を試みる．このアルゴリズムはテストケースにアサーションがあることを仮定しているが，テストケースの中にはアサーションがないものも存在する．

もしテストケースにアサーションがない場合，テストケース名を用いたアルゴリズムを適用する．しかし，テストケース名にテスト対象メソッドの情報がない場合もある．このような場合の類似度を下げるために，テストケース名と候補メソッドの名前の類似度を式 (2) のように変更した．そして，すべての候補の類似度が *threshold* と名づけたパラメータよりも低ければ，テストケース名を用いたアルゴリズムの結果を棄却する．本論文では，*threshold* は 0 とした．

$$Similarity = \frac{2 \times |C1 \cap C2|}{|C1| + |C2| - 2 \times |Common|} \qquad (2)$$

ここで，$C1$ はキャメルケースに従ってメソッド名を分割した単語集合を表し，$C2$ はテストケース名を分割した単語集合を表す．$Common$ はテストケース名とすべての候補メソッド名の単語集号の共通部分を表す．

2 つのアルゴリズムのどちらでもテスト対象を識別できなかった場合，最後に呼び出されたメソッドをテスト対象のメソッドとする．これは，アサーションのないテストケースは，プログラムが最後まで実行できるかどうかをテストすることが多いため，最後に呼び出されたメソッドはテストケースで最も重要であると考えられるからである．

3.4.2　テスト間の依存関係の抽出

本項では，テスト間依存関係の定義を説明する．このアルゴリズムは，文献 [12] で提案された依存関係を基礎としている．

元論文における依存関係は，単体テストのデバッグを効率的に行うためのものであるが，提案する依存関係は，チュートリアルに使用するテストケースを決定し，チュートリアルを作るための順序関係を生成するためのものである．そして，これらの目的を達成するために，直接依存と間接依存の 2 種類の依存関係を定義した．

直接依存において，テストケース X がテストケース Y に依存することは，X のテスト対象メソッドすべてが Y のメインまたはメンバから呼び出されていることと同値である．また，間接依存において，テストケース X がテストケース Y に依存することは，X のテスト対象メソッドすべてが Y の実行中に呼び出され，かつ X が直接依存の定義で Y に依存しないことと同値である．直接依存の定義で X が Y に依存するとは，Y が示す API 使用方法が，X の API 使用方法の理解の手助けになるということを意味する．そして，間接依存の定義で X が Y に依存するとは，

Compiler.java

```
public class Compiler {
    ...
    public Compiler() { }
    public Compiler(Settings settings) { ... }
    public String convertTabToSpace(String str) {
        return str.replace("\t", "␣␣␣␣");
    }
    public AST parse(String text) {
        String program = convertTabToSpace(text);
        return ... ;
    }
    public byte[] compile(AST ast) {
        return ... ;
    }
}
```

TestCase.java

```
@Test public void testConvertTabToSpace() {
    assertEquals("␣␣␣␣", new Compiler().
        convertTabToSpace("\t"));
}
@Test public void testParse() {
    Compiler lang = new Compiler();
    AST ast = lang.parse(...);
    assertFalse(ast != null);
    ...
}
@Test public void testCompile() {
    Compiler lang = new Compiler(new Settings());
    AST ast = lang.parse(...);
    byte[] resutls = lang.compile(ast);
    ...
}
```

図 6　テスト間依存関係の例
Fig. 6　An example of the dependencies between tests.

Y の API 使用方法が X の内部処理を理解するのに必要であることを意味する．

2 つの依存関係の例として，**図 6** のプログラムを考える．まず，3.4.1 項のアルゴリズムから，`testConvertTabToSpace` のテスト対象は，`Compiler.convertTabToSpace` であり，`testParse` のテスト対象は `Compiler.parse` であり，`testCompile` のテスト対象は，`Compiler.compile` であることが分かる．次に，`testCompile` は，`Compiler.parse` をメインの要素（`testCompile`）から呼び出しているので，直接依存の定義で，`testCompile` は `testParse` に依存する．最後に，`testParse` は，`Compiler.convertTabToSpace` を `Compiler.parse`（メインの要素でもメンバの要素でもないメソッド）から呼び出しているので，間接依存の定義で，`testParse` は `testCovertTabToSpace` に依存する．

以上から，図 6 のプログラムにおける依存関係は，**図 7** のようになる．点線は間接依存を表し，実線は直接依存を表している．

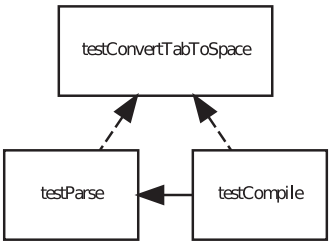

図 7　図 6 における依存関係．点線は間接依存を，実線は直接依存を表す

Fig. 7　The dependencies between tests in Fig. 6. The dotted line represents *indirect dependencies* and the normal lines represent *direct dependencies*.

3.4.3　チュートリアルの生成

このセクションでは，これまでの結果からチュートリアルを生成する手法を説明する．まず，本手法はどのテストケースからも依存されていないテストケースを得，それをライブラリの実際的な使い方を示すサンプルと考える．次に，API の概略を示すために，これらのテストケース内の API 使用例を説明するチュートリアルを生成する．最後に，テストされているすべてのメソッドに対し，それぞれを説明するチュートリアルを生成する．そのために，テスト間の依存関係を用いて，コードサンプルのリストを生成する次のアルゴリズムを利用する．

1　クラス X がクラス Y なしではコンパイルできないとき，X が Y に依存すると定義し，その依存関係を解析する．
2　直接依存，間接依存のどちらにおいても，どのテストケースからも依存されていないテストケースの集合を得，そしてこれを種（seeds）と呼ぶ．
3　直接依存において，種が依存しているテストケースを探し，それを種に追加する．この操作を種が変化しなくなるまで繰り返す．
4　テスト対象のクラス（テスト対象メソッドを宣言したクラス）を利用して，種を分類し，クラスタを得る．
5　クラスタそれぞれに対し，クラスタによって誘導される直接依存の部分グラフを得る．このグラフに対して，メイン，メンバから実行されたメソッドの数を用いてトポロジカルソート [27] を行い，テストケースのリスト（全順序）を生成する．
6　クラスタ間の順序関係を，1 で得た依存関係とテスト間の依存関係を用いて生成する．この順序関係において，X は Y 以上であるとは，クラスタが次の 2 条件のいずれか一方を満たすことと同値である．
   a　テスト間の依存関係に Y の要素から X の要素へのパスが存在する．
   b　1 で得た依存関係に Y の要素のテスト対象クラスから，X の要素のテスト対象クラスへのパスが存在する．

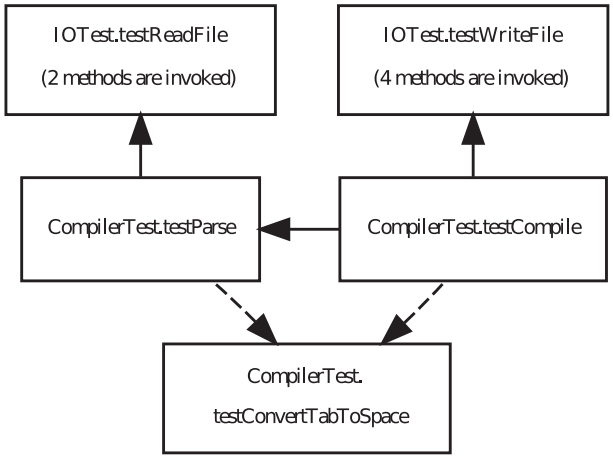

図 8　テスト間の依存関係の入力

Fig. 8　An example of the dependencies between tests.

7　テストケースで使用されているクラスの数を用いてトポロジカルソートを行い，クラスタのリスト（全順序）を生成する．
8　ここまでで得たリストにおいて，テストケースをそれを説明する HTML ページへ変換し，API 概要を示すチュートリアルを得る．それぞれの HTML ページのタイトルは，サンプルの元となったテストケースの名前を用いる．
9　3 から 8 をテスト対象となっているそれぞれのメソッドに対して行う．このとき，種の初期値は，対象のメソッドをテスト対象とするテストケースの集合とする．

図 8 の依存関係をこの手法の入力例として考える．図 8 は，テスト間の依存関係を表しており，点線は間接依存を，実線は直接依存を意味する．また，1 で得られる依存関係はいっさい存在しないと仮定する．

まず，`testCompile` はどのテストケースからも依存されていないので，種の初期値は `testCompile` である．次に，直接依存関係をたどって種を更新し，`testCompile`，`testWriteFile`，`testParse`，`testReadFile` という結果を得る．

種をクラスタリングし，`testReadFile`，`testWriteFile` というクラスタ（クラスタ 1）と，`testParse`，`testCompile` というクラスタ（クラスタ 2）を得る．これは，クラスタ 1 の要素のテスト対象クラスは `IO` であり，クラスタ 2 は `Compiler` だからである．次に，それぞれのクラスタにトポロジカルソートを行い，`testReadFile`，`testWriteFile` というリストと，`testParse`，`testCompile` という 2 つのリストを得る．テスト間依存関係から，`IO` が `Compiler` 以上である，という順序関係を得る．最後に，API 概要をしめすチュートリアルのリストとして，

1　`IOTest.testReadFile`
2　`IOTest.testWriteFile`
3　`CompilerTest.testParse`

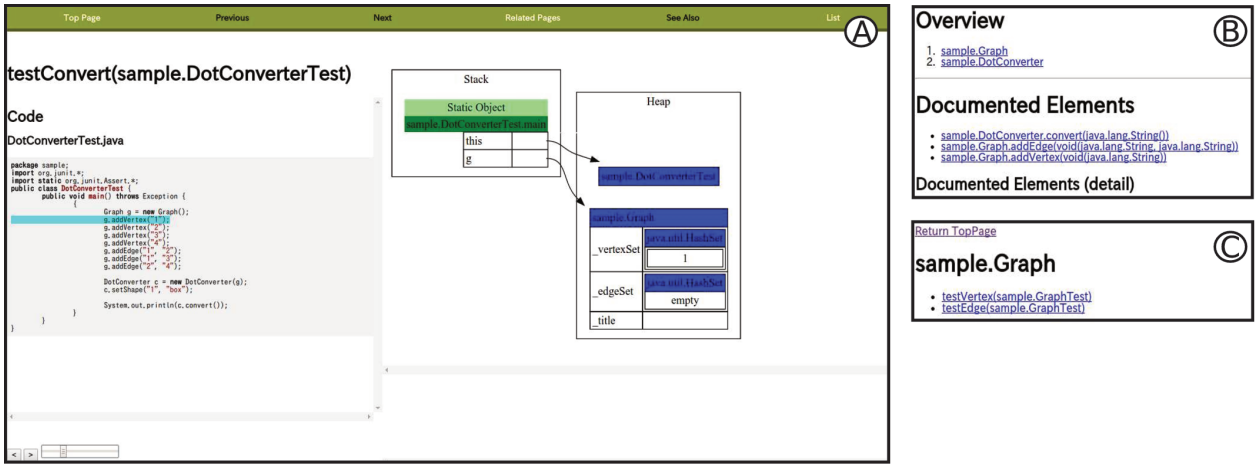

図 9　付録 A.1 から生成されるチュートリアル A) サンプルを表示した HTML ページ，B) チュートリアルのトップページ，C) `Graph` クラスのチュートリアルのリスト

Fig. 9　The tutorial generated from Appendix A.1. A) The html page showing example, B) The top page of the generated tutorial and C) The list of `Graph` tutorial.

4　`CompilerTest.Compile`

を得る．

図 9 は提案手法全体の出力例である．これは，付録 A.1 から提案手法で生成されるチュートリアルを示している．

4. ユーザスタディ

提案手法を評価するためにユーザスタディを行った．このユーザスタディの目的は次のとおりである．

1　提案手法によるチュートリアルが，API の理解にどれだけ役立つのかを調査する．

2　提案手法の改善のためのフィードバックを得る．

ユーザスタディのために，Commons CLI[*2]のドキュメントを提案手法によって生成した．Commons CLI は，コマンドライン引数を扱うライブラリである．Commons CLI をユーザスタディで用いた理由は次の 2 つである．まず，このライブラリでは API を適当な方法で使用する方法があるので，チュートリアルが重要であることがあげられる．2 番目に，このライブラリの API はあまり複雑ではないため，短い時間でユーザスタディを行うことができることがあげられる．

比較のため，*UsETeX* [30] によって抽出されたコードサンプルを追加した Javadoc を用いた．*UsETeX* を選択した理由は，*UsETeX* はテストコードからサンプルを抽出するツールであるので，提案手法と同じ情報を用いてドキュメントを生成していることである．

4.1 方法

表 1 のような 3 人の参加者に対し，それぞれ 60 分のユーザスタディを実施した．参加者は，年，性別，プログラミング経験についての事前アンケートに答えた．事前アンケートの結果を表 1 に示す．また，この 3 人の参加者の中に以前 Commons CLI を利用したことのある人はいなかった．

[*2] http://commons.apache.org/proper/commons-cli/

表 1　ユーザスタディの参加者

Table 1　Participants of the user study.

参加者	年齢	性別	プログラミング経験	Java 経験
P1	22	男	7 年	2 年
P2	22	男	3 年	1 年
P3	23	男	12 年	1 年

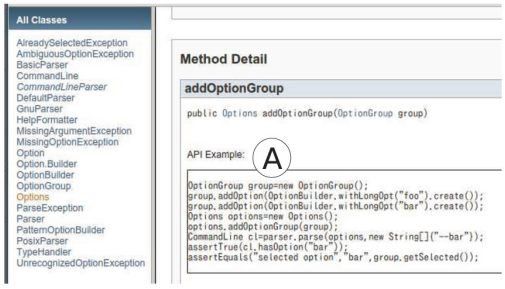

図 10　ユーザスタディで利用した Javadoc A) *UsETeX* により抽出されたコードサンプル

Fig. 10　The Javadoc document used in the user study. A) the code example extracted by *UsETeX*.

事前アンケートのあと，参加者は 2 つのプログラミングタスク（echo コマンドの実装と cat コマンドの実装）を行った．タスク遂行中，参加者は，Javadoc（**図 10**）か，提案手法によるチュートリアル（**図 11**）のどちらかをドキュメントとして利用した．

そして，タスクを行う順番や，使用するドキュメントとタスクの組合せは参加者ごとに変更した．具体的には，P1 は最初に echo コマンドを提案手法によるチュートリアルを用いて実装し，次に cat コマンドを Javadoc を用いて実装

図 11　ユーザスタディで用いたチュートリアル A) チュートリアルのサンプルページ，B) チュートリアルのトップページ

Fig. 11　The tutorial used in the user study. A) the example page of the tutorial and B) the top page of the tutorial.

した．P2 は最初に cat コマンドを Javadoc を用いて，次に echo コマンドを提案手法を用いて実装した．そして，P3 は最初に echo コマンドを Javadoc を用いて，次に cat コマンドを提案手法を用いて実装した．

次に，参加者が行った 2 つの実装タスクについて詳述する．これらのタスクはどちらも，次の 5 つのサブタスクに分割できる．

A. `Options` インスタンスの設定　適切にコマンドライン引数を扱う前準備として，参加者は `Options` のインスタンスを作り，それぞれのオプションをインスタンスに設定する必要がある．

B. コマンドライン引数のパース　コマンドライン引数を実際にパースするために，参加者は，`DefaultParser` のインスタンスを作り，`CommandLineParser.parse` メソッドを用いてコマンドライン引数をパースする必要がある．

C. 特定のオプションの確認　あるオプションが設定されているかどうかを確認するために，参加者は，`CommandLineParser.parse` メソッド呼び出しの後で `hasOption` メソッドを適切に用いる必要がある．

D. オプションに属さない引数の取得　オプションに属さない引数を取得するために，参加者は，`getArgList` メソッドを `CommandLineParser.parse` メソッド呼び出しの後で呼び出す必要がある．

E. ヘルプメッセージの表示　ヘルプメッセージの表示のために，参加者は，`HelpFormatter` のインスタンスを生成し，`printHelp` メソッドと `Options` インスタンスを用いてヘルプメッセージを出力する必要がある．

参加者が Commons CLI に関連するプログラミングタスクにできるだけ長く取り組むように，タスクに関係ないプログラムは事前に用意した．たとえば，echo コマンドの実装で必須だが，Commons CLI には関係ない，文字列のリストを表示するプログラムは事前に用意した．

各タスクの終了後，遂行したサブタスクの数を数えた．タスクの遂行は，上述した API を適切に利用したかどうかで判断する．**図 12** をタスクの遂行の例としてあげる．このプログラムを書いた参加者は，h オプションが設定されているか

```
...
CommandLine cl;
if (cl.hasOption("h")) {
    ...
}
...
```

図 12　参加者によって書かれたプログラムの例．タスク C は遂行できたがタスク B はできていない

Fig. 12　An example of the program written by a participant. He completed Task C but did not complete Task B.

表 2　ユーザスタディの結果．タスクを遂行できたドキュメントがセルに記述される

Table 2　Summary of the user study results. The cells contain the documents that the participant completed a task by using these.

参加者	A	B	C	D	E
P1	tutorial				tutorial
P2	tutorial		both		
P3	tutorial	both	Javadoc	tutorial	tutorial

表 3　事後アンケートの結果．1 はまったく同意できないこと，7 は強く同意できることを表す

Table 3　The results of the post questionnaire. Rating of 1 represents strong disagreement and Rating of 7 represents strong agreement.

参加者	Q1 使いやすさ	Q2 可視化	Q3 サンプルの順序
P1	3	1	7
P2	6	3	4
P3	5	3	5

確認するために，`CommandLine.hasOption` メソッドを利用しているが，`CommandLine` インスタンスを取得する方法は分かっていない．したがって，`CommandLineParser.parse` メソッドを利用できていないので，タスク B は遂行できなかったと見なすが，`CommandLine.hasOption` は適切に使用しているため，タスク C は遂行できたと見なした．

2 つのタスクと遂行したサブタスク数計測の終了後，参加者は次のような事後アンケートに答え，チュートリアルの改善点に関するコメントを記述した．

Q1　Javadoc よりもチュートリアルのほうが使いやすいと感じたか？

Q2　プログラム可視化は API の理解に役に立ったか？

Q3　サンプルの順序は自然だと感じたか？

4.2　結果

ユーザスタディの結果を**表 2** に示す．また，事後アンケートの結果を**表 3** に示す．

4.2.1　事後アンケートの結果について

ユーザスタディでは，条件の違いにかかわらず，Javadoc

よりもチュートリアルを利用したほうが多くのプログラミングタスクを遂行することができた．P2 と P3 は，Javadoc よりもチュートリアルのほうが使いやすいと感じたと事後アンケート（表 3）で答えた．

P1 は，Javadoc にあるコードサンプルのほうが短く，コードを書きやすかったため，Javadoc のほうがチュートリアルのほうが使いやすいと感じたと答えた．P1 は Javadoc を用いた実験では 4 つのサブタスクに関連するコードを書いたが，どれも遂行することはできなかった．一方で，チュートリアルを用いた実験では，3 つのサブタスクに関連するコードを書き，うち 2 つを遂行した．

P1 と P3 はコードサンプルの順序は自然だと感じたと答えた（表 3）．P1 は，順序には問題ないけれども，コードサンプルに多くの情報が含まれすぎていて，簡単にサンプルを理解することができなかったと答えた．

すべての参加者は，プログラム可視化は API 理解に役に立たなかったと答えた．P1 は，可視化されたプログラムの状態を見るのに時間がかかりすぎると答えた．また，P2 は，プログラム可視化により，コードサンプルの領域が狭められているのが問題だと答えた．

4.2.2　手法の改善点に関するフィードバック

P1 と P2 は，提案手法によるコードサンプルが長く，複雑すぎると答えた．

P2 はプログラム可視化について 2 つの改善方法を提案した．1 つ目は，前回のステップと現在のステップの間の状態の差異を可視化する情報に加えることであり，2 つ目は，コードサンプルと可視化された状態の同期をとることである．たとえば，ユーザがサンプルの 2 行目をクリックしたら，2 行目に対応する状態が可視化されるようになる．

すべての参加者が，自然言語によるコードサンプルの説明の欠如は問題であると指摘した．P1 はコードサンプルの理解のために，より多くのコメントが必要だと答えた．また，P2 は，タイトルからコードサンプルの内容を簡単に推測できることが望ましいと答えた．P3 は，サンプルの入出力が表示されると，コードサンプルの理解が簡単になるのではないかと答えた．P2 と P3 は，メソッドの型をコードサンプルから推測するのは大変なので，チュートリアルにメソッドの型の情報が組み込まれるべきだと話した．

5.　議論と今後の課題

5.1　議論

ユーザスタディにおいて，Javadoc を使うよりも提案手法によるチュートリアルを使用したときのほうが，すべての参加者が多くのタスクを遂行することができた．そして，2/3 の参加者が提案手法によるチュートリアルのほうが Javadoc より使いやすいと感じた．一方で，すべての参加者がチュートリアルのプログラム可視化が API の理解には役に立たないと感じ，チュートリアルの改善点を指摘した．

ユーザスタディの結果は，提案手法によるチュートリアルが，複雑な API の使用方法を理解するうえで API ドキュメントよりも効果的であったことを示唆している．チュートリアルの記述は困難で時間のかかる作業であるため，本手法はチュートリアルの保守に有用だと考えられる．

しかし，本アルゴリズムには制限と問題が存在する．まず，本手法はプログラム可視化技術を使用するために実行可能なコードサンプルを生成する．しかし，実行可能にするために，コードサンプルは複雑なものとなっており，可読性が低下している．さらに，入力によっては提案手法は実行不可能なサンプルを生成してしまう．また，本手法では，コードサンプルの説明にプログラム可視化技術を用いているが，ユーザスタディでこれはチュートリアルに不適切であることが分かった．

5.2　今後の課題

生成されたチュートリアルの有用性の向上に関して，以下の 3 つの課題がある．

1 つ目は，抽出するコードサンプルの質の向上である．コードサンプルの可読性は重要であり，可読性の向上は生成されたチュートリアルの改善につながる．サンプルを単純なサンプルに分割することは，可読性を向上させる可能性がある．

2 つ目は自然言語を用いたコード説明の生成である．ユーザスタディの結果は，自然言語による説明は，プログラム可視化よりもチュートリアルに適していることを示唆しているが，本手法は自然言語による適切な説明を生成できない．コードに対するパターンマッチ技術を用いると，自然言語による説明を生成することができるかもしれない．

最後に，半自動的チュートリアル生成手法への拡張である．Javadoc のように開発者がアノテーションを使えるようにすることで，提案手法はより実用的な手法になりうる可能性がある．使用するアノテーションは，チュートリアルの生成だけでなく，テストケースの理解に役立つものである必要がある．

6.　結論

本論文では，単体テストからチュートリアルを自動生成する手法，特にテスト間の依存関係を利用して，チュートリアルを生成するアルゴリズムを提案した．ユーザスタディによって，本手法がテストケースの自然な順序を生成できたこと，生成されたチュートリアルは，よく知らない API を理解し，それを用いてプログラミングする際に有用であることを示した．しかし，ユーザスタディでは，より効果的なチュートリアルを生成するためのいくつかの改善点も発見された．

謝辞　加藤淳氏と増原英彦氏に，本手法についてさまざまなご教示をいただいたことを深謝する．

参考文献

[1] Cumiki, available from ⟨http://cumiki.com/⟩ (accessed 2015-01-18).
[2] doctest, available from ⟨https://docs.python.org/2.7/library/doctest.html⟩ (accessed 2015-01-18).
[3] Beck, K.: *Test-driven development: By example*, Addison-Wesley Professional (2003).
[4] Buse, R.P. and Weimer, W.: Synthesizing API usage examples, *2012 34th International Conference on Software Engineering* (*ICSE*), pp.782–792, IEEE (2012).
[5] Cross, J., Hendrix, T.D., Barowski, L.A., et al.: Combining Dynamic Program Viewing and Testing in Early Computing Courses, *Computer Software and Applications Conference* (*COMPSAC*), *2011 IEEE 35th Annual*, pp.184–192, IEEE (2011).
[6] Cross II, J.H., Hendrix, T.D., Umphress, D.A., Barowski, L.A., Jain, J. and Montgomery, L.N.: Robust generation of dynamic data structure visualizations with multiple interaction approaches, *ACM Trans. Computing Education* (*TOCE*), Vol.9, No.2, p.13 (2009).
[7] Dagenais, B. and Robillard, M.P.: Creating and evolving developer documentation: Understanding the decisions of open source contributors, *Proc. 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp.127–136, ACM (2010).
[8] Dagenais, B. and Robillard, M.P.: Recovering traceability links between an API and its learning resources, *2012 34th International Conference on Software Engineering* (*ICSE*), pp.47–57, IEEE (2012).
[9] Devanbu, P., Karstu, S., Melo, W. and Thomas, W.: Analytical and empirical evaluation of software reuse metrics, *Proc. 18th International Conference on Software Engineering*, pp.189–199, IEEE Computer Society (1996).
[10] Gaelli, M., Nierstrasz, O. and Ducasse, S.: One-method commands: Linking methods and their tests, *OOPSLA Workshop on Revival of Dynamic Languages* (2004).
[11] Gaffney Jr., J.E. and Durek, T.A.: Software reuse—Key to enhanced productivity: Some quantitative models, *Information and Software Technology*, Vol.31, No.5, pp.258–267 (1989).
[12] Gaelli, M., Lanza, M., Nierstrasz, O. and Wuyts, R.: Ordering broken unit tests for focused debugging, *Proc. 20th IEEE International Conference on Software Maintenance, 2004*, pp.114–123, IEEE (2004).
[13] Ghafari, M., Ghezzi, C., Mocci, A. and Tamburrelli, G.: Mining unit tests for code recommendation, *ICPC*, pp.142–145 (2014).
[14] Ginosar, S., Pombo, D., Fernando, L., Agrawala, M. and Hartmann, B.: Authoring multi-stage code examples with editable code histories, *Proc. 26th Annual ACM Symposium on User Interface Software and Technology*, pp.485–494, ACM (2013).
[15] Guo, P.J.: Online Python Tutor: Embeddable web-based program visualization for CS education, *Proc. 44th ACM Technical Symposium on Computer Science Education*, pp.579–584, ACM (2013).
[16] Horie, M. and Chiba, S.: Tool support for crosscutting concerns of API documentation, *Proc. 9th International Conference on Aspect-Oriented Software Development*, pp.97–108, ACM (2010).
[17] Kim, J., Lee, S., Hwang, S.-W. and Kim, S.: Adding examples into Java documents, *24th IEEE/ACM International Conference on Automated Software Engineering, 2009, ASE '09*. pp.540–544, IEEE (2009).
[18] Kuhn, A., Van Rompaey, B., Haensenberger, L., Nierstrasz, O., Demeyer, S., Gaelli, M. and Van Leemput, K.: JExample: Exploiting dependencies between tests to improve defect localization, *Agile Processes in Software Engineering and Extreme Programming*, pp.73–82, Springer (2008).
[19] Mandelin, D., Xu, L., Bodík, R. and Kimelman, D.: Jungloid mining: Helping to navigate the API jungle, *ACM SIGPLAN Notices*, Vol.40, No.6, pp.48–61 (2005).
[20] Nasehi, S.M. and Maurer, F.: Unit tests as API usage examples, *2010 IEEE International Conference on Software Maintenance* (*ICSM*), pp.1–10, IEEE (2010).
[21] Nguyen, A.T., Nguyen, T.T., Nguyen, H.A., Tamrawi, A., Nguyen, H.V., Al-Kofahi, J. and Nguyen, T.N.: Graph-based pattern-oriented, context-sensitive source code completion, *Proc. 2012 International Conference on Software Engineering*, pp.69–79, IEEE Press (2012).
[22] Robillard, M.P.: What makes APIs hard to learn? Answers from developers, *Software*, Vol.26, No.6, pp.27–34, IEEE (2009).
[23] Robillard, M.P. and Deline, R.: A field study of API learning obstacles, *Empirical Software Engineering*, Vol.16, No.6, pp.703–732 (2011).
[24] Rozenberg, D. and Beschastnikh, I.: Templated Visualization of Object State with Vebugger, *2014 2nd IEEE Working Conference on Software Visualization* (*VISSOFT*), pp.107–111, IEEE (2014).
[25] Scaffidi, C.: Why are APIs difficult to learn and use?, *Crossroads*, Vol.12, No.4, p.4 (2006).
[26] van Deursen, A., Moonen, L., van den Bergh, A. and Kok, G.: *Refactoring test code*, CWI (2001).
[27] Weisstein, E.W.: Topological sort, *From Mathworld–A Wolfram web resource* (2000).
[28] Xie, T. and Pei, J.: MAPO: Mining API usages from open source repositories, *Proc. 2006 International Workshop on Mining Software Repositories*, pp.54–57, ACM (2006).
[29] Zhu, Z., Zou, Y., Jin, Y. and Xie, B.: Generating API-usage example for project developers, *Proc. 5th Asia-Pacific Symposium on Internetware*, p.34, ACM (2013).
[30] Zhu, Z., Zou, Y., Xie, B., Jin, Y., Lin, Z. and Zhang, L.: Mining API usage examples from test code, *2014 IEEE International Conference on Software Maintenance and Evolution* (*ICSME*), pp.301–310, IEEE (2014).

付　　録

A.1 アルゴリズムの入力例

A.1.1 ライブラリのソースコード

Graph.java

```
package sample;
public class Graph {
    public class Pair {
        public Pair(String v1, String v2) {
            this.v1 = v1;
            this.v2 = v2;
        }
        public String v1;
        public String v2;
    }
    private java.util.Set<String> _vertexSet;
```

```
    private java.util.Set<Pair> _edgeSet;
    private String _title;
    public Graph(String title) {
        _title = title;
        _vertexSet = new java.util.HashSet<String>();
        _edgeSet = new java.util.HashSet<Pair>();
    }
    public Graph() {
        _title = "";
        _vertexSet = new java.util.HashSet<String>();
        _edgeSet = new java.util.HashSet<Pair>();
    }

    public String getTitle() {
        return _title;
    }
    public void addVertex(String vertex) {
        _vertexSet.add(vertex);
    }
    public java.util.Set<String> getVertexSet() {
        return _vertexSet;
    }
    public void addEdge(String v1, String v2) {
        _edgeSet.add(new Pair(v1, v2));
    }
    public java.util.Set<Pair> getEdgeSet() {
        return _edgeSet;
    }
    public java.util.Set<String> neighbors(String vertex) {
        java.util.Set<String> neighbors = new java.util.
            HashSet<String>();
        for (Pair p: _edgeSet) {
            if (p.v1.equals(vertex)) {
                neighbors.add(p.v2);
            } else if (p.v2.equals(vertex)) {
                neighbors.add(p.v1);
            }
        }
        return neighbors;
    }
}
```

DotConverter.java

```
package sample;

public class DotConverter {
    private Graph g;
    private java.util.Map<String, String> shapes;
    public DotConverter(Graph g) {
        this.g = g;
        shapes = new java.util.HashMap<String, String>();
    }
    public void setShape(String v, String shape) {
        shapes.put(v, shape);
    }
    public String convert() {
        String r = "graph␣" + g.getTitle() + "␣{\n";
        for(String v: g.getVertexSet()) {
            if (shapes.containsKey(v)) {
                r += (v + "[shape=\"" + shapes.get(v) +
                    "\"]\n");
            } else {
                r += (v + "\n");
            }
        }
        for (Graph.Pair p: g.getEdgeSet()) {
            r += p.v1 + "␣--␣" + p.v2 + "\n";
        }
        r += "}";
        return r;
    }
}
```

A.1.2　テストコード

GraphTest.java

```
package sample;

import org.junit.*;
import static org.junit.Assert.*;

public class GraphTest {
    @Test
    public void testVertex() {
        Graph g = new Graph();
        g.addVertex("foo");
    }
    @Test
    public void testEdge() {
        Graph g = new Graph("title");
        g.addVertex("foo");
        g.addVertex("bar");
        g.addEdge("foo", "bar");
    }
}
```

DotConverterTest.java

```
package sample;

import org.junit.*;
import static org.junit.Assert.*;

public class DotConverterTest {
    @Test
    public void testConvert() {
        Graph g = new Graph();
        g.addVertex("1");
        g.addVertex("2");
        g.addVertex("3");
        g.addVertex("4");
        g.addEdge("1", "2");
        g.addEdge("1", "3");
        g.addEdge("2", "4");
```

```
        DotConverter c = new DotConverter(g);
        c.setShape("1", "box");

        System.out.println(c.convert());
    }
}
```

三上 裕明

1992 年生．2015 年東京大学理学部情報科学科卒業．2015 年同大学大学院修士課程．開発環境に関する研究に取り組んでいる．

坂本 大介

2008 年公立はこだて未来大学大学院システム情報科学研究科博士（後期）課程修了．博士（システム情報科学）．国際電気通信基礎技術研究所（ATR）にてインターン，東京大学にて日本学術振興会特別研究員 PD，JST ERATO 五十嵐デザインインタフェースプロジェクト研究員，東京大学大学院情報理工学系研究科コンピュータ科学専攻助教を経て，現在，同大学同研究科特任講師．人とロボットを含む情報環境とのインタラクション設計に関する研究に従事．

五十嵐 健夫

2000 年東京大学大学院においてユーザインタフェースに関する研究により博士号（工学）取得．2002 年 3 月に東京大学大学院情報理工学系研究科講師就任，2005 年 8 月より同助教授，2011 年 5 月より教授．IBM 科学賞，学術振興会賞，ACM SIGGRAPH Significant New Researcher Award，Katayanagi Prize in Computer Science 等受賞．ユーザインタフェース，特に，インタラクティブコンピュータグラフィクスに関する研究に取り組んでいる．