Journal of Information Processing Vol.27 257–267 (Mar. 2019)

[DOI: 10.2197/ipsjjip.27.257]

Regular Paper

Easy-going Development of Event-Driven Applications by Iterating a Search-Select-Superpose Loop

Masashi Nishimoto[1,a) Keiji Nishiyama[1,b) Hideyuki Kawabata[1,c) Tetsuo Hironaka[1,d)

Received: June 1, 2018, Accepted: December 4, 2018

Abstract: Today's application development process depends heavily on the usage of application programming interfaces (APIs) for many kinds of frameworks. Time spent searching for appropriate API members and understanding their usages tends to occupy much of the time required for the whole development process. This paper proposes a new approach for developing application programs based on APIs in a simple way: through code development by iterating a *Search-Select-Superpose* (SSS) loop. The approach comprises three phases. In the *Search* phase, the user searches for a way to implement a desired *functionality* by combining API calls. The search results are shown to the user as a list of *outlines* (sets of words) attached to code skeletons. A code skeleton, chosen in the *Select* phase, is then merged with the program at hand in the *Superpose* phase. The entire process is implemented through the construction of an indexed dataset composed of code skeletons extracted from open-source repositories, and through the use of a tool to control the SSS loop. We have developed a prototype of the proposed system. In this paper, the design and implementation of the proposed system are described. The effectiveness of the system was confirmed through empirical results from experiments with event-driven Android application development.

Keywords: code search, API usage, application development, event-driven programming, Android

1. Introduction

Today's application development process depends heavily on the use of application programming interfaces (APIs). Acquiring understanding of API usages, however, is not an easy task for an ordinary programmer, because there are too many frameworks (and thus, APIs) to choose from, API documentation might not be up to date, and API specifications can change rapidly [8], [18]. One study found that 34.2% of all queries containing the word "java" and given to existing search engines were questions on API usages [4]. Indeed, the process of searching for appropriate API members, understanding their usages, and building code using them tends to occupy much of the time required for the whole development process.

Is knowledge of the usage of each individual API member enough to build an application? The answer is "no" if mutual relations or restrictions among them are considered. In addition, many code patterns consist of cooperating APIs — many functionalities implemented in application programs, such as file access, user account authentication, network connection, and audio data playback, involve combinations of multiple API calls. To build an application, a user must know a number of idioms based on various APIs. There are many research activities for the purpose of supporting program development based on APIs [2], [3], [5], [10], [17], [19], [23], [24], [29], [30].

1 Graduate School of Information Sciences, Hiroshima City University, Hiroshima 731–3194, Japan
a) nishimoto.masashi@ca.info.hiroshima-cu.ac.jp
b) nishiyama@ca.info.hiroshima-cu.ac.jp
c) kawabata@hiroshima-cu.ac.jp
d) hironaka@hiroshima-cu.ac.jp

It would be ideal for the user to search for an API-based idiom by using a description of the desired functionality, and for the obtained pattern including multiple API calls to be easily patched into the user's code at hand. In this paper, we present an automation of these processes. Specifically, we propose a new approach for developing application programs based on APIs in a simple way: through the development of code by iterating a *Search-Select-Superpose* (SSS) loop. By iterating the SSS loop, the user can introduce one new functionality into the code in each cycle.

Here, we describe briefly the idea of application development by iterating the SSS loop. The loop body consists of three phases. In the first phase, called the *Search* phase, the user searches for a way to implement a desired *functionality* by combining API calls. The user is not required to know the exact names of the API members. The search results are shown to the user as a list of *outlines*. Each outline is associated with a set of API members accompanied by information on their usages as a code skeleton. An appropriate dataset consisting of sets of API members with corresponding outlines should be prepared in advance, possibly by gathering source files from open-source repositories. The API members in each set are mutually related and meant to cooperate to implement a particular functionality. In fact, the outline shown to the user is a set of weighted words arranged to properly describe the functionality implemented by the corresponding API member set.

In the second phase, called the *Select* phase, the user selects one of the outlines listed in the previous phase. The outcome of the *Search* phase could result in a lengthy list of outlines. Because the list is sorted in an order corresponding to what many programmers would expect, however, the selection should not impose a heavy burden on the user.

The purpose of the third phase of the SSS loop body is to incorporate the selected set of API members with some information to build the intended functionality into the code at hand. We call this the *Superpose* phase, because we have developed a tool to merge a selected code skeleton into the user's code, like superposing one layer onto a base. In superposition, our tool accounts for the structure of the code at hand to insert external code. That is, it attempts to reuse existing identifiers consistently while merging each segment of a skeleton, and it creates new identifiers to point to certain objects or name additional methods, if needed.

In general, we cannot expect that any kind of functionality is necessarily describable in a query language. Thus, it would be difficult to obtain exactly the intended API member sets by using a tool in the *Search* phase that is based on repository mining. It is also obviously unattainable to build a fully automatic tool for the *Superpose* phase that always works correctly, in the sense that the behavior of the resultant code is always exactly the same as the user's intention. Some types of applications based on APIs, however, like the event-driven programs typical of mobile applications for Android and iOS, have a great degree of formality. Thus, we assume that a semi-automatic superposition process at an acceptable level for such development can be achieved.

Accordingly, we have developed a prototype system to help the user perform a SSS loop iteration. The system consists of two tools: the *SSS-DSGen* dataset generator, and the *SSS-Editor* tool. The *SSS-DSGen* constructs an indexed dataset of code skeletons that can be used for program development by iterating the SSS loop. The dataset, which is called a *SSS-DS*, contains information on idiomatic usages of API member sets gathered automatically from input source programs. The *SSS-Editor*, on the other hand, works as a user interface for our system and controls the iteration of the SSS loop to enable rapid development of applications combining API member sets.

In this paper, we describe the design and implementation of the prototype system for iterating the SSS loop, and we show the effectiveness of this approach through experimental results with event-driven Android application development.

The rest of this paper is organized as follows. In Section 2, we give an overview of the system to iterate the SSS loop. Section 3 gives the details of how the *SSS-DSGen* constructs an indexed dataset from source programs. In Section 4, the design and implementation of the *SSS-Editor* are described. The prototype implementation of the system for iterating the SSS loop is described in Section 5. We evaluate our approach of iterating the SSS loop for application development through experimental results described in Section 6. Finally, in Section 7, related work is summarized, before we conclude in Section 8 by summarizing key points and mentioning our future work.

2. Iteration of a SSS Loop

2.1 Motivating Example: Application Development with Android Framework

A code fragment shown in **Fig. 1** (a) is a typical program in Java for Android at the beginning of application development. It defines a class `MainActivity` that extends a class of the Android

```
public class MainActivity extends AppCompatActivity {
  private SensorManager sManager;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    suppher.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    sManager =
    (SensorManager)getSystemService(
                        Context.SENSOR_SERVICE);
  }

  @Override
  protected void onResume() {
    super.onResume();
  }
}
```

(a) Before modification

```
public class MainActivity extends AppCompatActivity {
  private SensorManager sManager;
> private Sensor mAccelerometer;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    suppher.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    sManager =
      (SensorManager)getSystemService(
                          Context.SENSOR_SERVICE);
>   mAccelerometer =
>   sManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
  }

  @Override
  protected void onResume() {
    super.onResume();
>   sManager.registerListener(this, mAccelerometer,
>                   SensorManager.SENSOR_DELAY_GAME);
  }

> @Override
> protected void onPause() {
>   sManager.unregisterListener(this);
> }
}
```

(b) After modification (lines preceded by '>' were inserted)

Fig. 1 Adding a functionality to a code fragment in Java for Android.

framework, `AppCompatActivity`, and contains callback methods `onCreate` and `onResume`, which are expected to be overridden by the user's code.

Suppose that we want to add a functionality to obtain values from an accelerometer and use them for some computation. Note that we cannot accomplish this task by adding just one method from an API — we need to add a set of methods. In addition, those methods must often be inserted at separate positions in the source. For example, in the accelerometer case, as shown in Fig. 1 (b), we might have to scatter additional lines throughout the file.

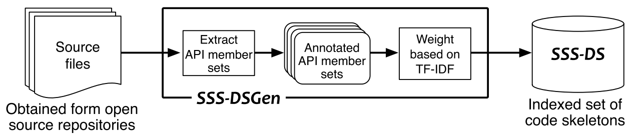
Our motivation for developing the SSS loop iteration system is to make it possible to manage such kinds of modifications semi-automatically. With the proposed system, the user can obtain a set of API methods, e.g.,

{ `getSystemService`, `getDefaultSensor`,
`registerListener`, `unregisterListener` },

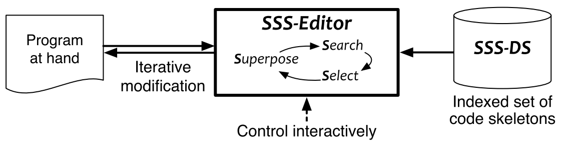
by inputting the keywords "get, sensor, accelerometer." After selecting the set from among various candidates, the user can then *superpose* it into the program at hand — each API call is inserted into the body of a suitable method in the user class. The system automatically inserts overriding methods if an appropriate one does not exist in the user's program.

2.2 System Structure

In this paper, we propose an approach to application devel-


Source files
Obtained form open source repositories
Extract API member sets
Annotated API member sets
Weight based on TF-IDF
SSS-DSGen
SSS-DS
Indexed set of code skeletons

(a) Construction of an indexed dataset



(b) Application development by iterating the SSS loop

Fig. 2 Structure of the system for iterating the SSS loop.

opment based on iterating the SSS loop. The entire process is implemented with two components: the *SSS-DSGen* dataset generator and the *SSS-Editor* tool. **Figure 2** shows the structure of our prototype. The *SSS-DSGen* constructs an indexed set of code skeletons, called a *SSS-DS*, which the *SSS-Editor* then uses. Section 3 describes the design of the *SSS-DSGen* and each of its components, shown in Fig. 2 (a). Section 4 then explains the details of the *SSS-Editor*, shown in Fig. 2 (b).

3. *SSS-DSGen*: Construction of *SSS-DS*

The *SSS-DSGen* is a tool for constructing an indexed dataset from a set of source files, as shown in Fig. 2 (a). The indexed dataset, called a *SSS-DS*, is a set of code skeletons. Each code skeleton implements a particular functionality by combining API members,[*1] accompanied by a set of words as an annotation (or outline). The annotation for a code skeleton is made of words extracted from the names of API members used in the skeleton. To develop applications through interactive sessions with the *SSS-Editor*, the *SSS-DS* is used while iterating the SSS loop, as shown in Fig. 2 (b).

3.1 Extracting Code Skeletons Through Data Dependence

To extract a code skeleton from source files obtained from, e.g., open-source repositories, the *SSS-DSGen* analyzes each source file to find out how API methods cooperate.

We consider how the source file shown in **Fig. 3**, taken from Google Samples[*2], is processed by the *SSS-DSGen*. First, several graphs are constructed by analyzing data dependencies between API members. In the graph example shown in **Fig. 4**, nodes with underlined labels are API member names, while elliptical nodes are variables. The labels in boxes indicate the method names from which those API calls were obtained. In the figure, identifiers are connected by arrows labeled as **arg**, **gen**, or **points to**, meaning that the value is passed to the method as an argument, the method (or a class constructor) generates the value or the identifier pointing to the object has the method, respectively. We extract one code skeleton (a set of syntax trees and additional information) from each graph.

Unlike other tools that extract sets of API calls, the extraction of sets of API members by the *SSS-DSGen* is unaffected by the

```
public class DeviceScanActivity extends ListActivity {
    private LeDeviceListAdapter mLeDeviceListAdapter;
    private BluetoothAdapter mBluetoothAdapter;

    @Override
    protected void onCreate(Bundle ...) {
        ...
        if (!getPackageManager().hasSystemFeature(
             PackageManager.FEATURE_BLUETOOTH_LE)) {
            Toast.makeText(this, ...).show();
            finish();
        }

        final BluetoothManager bluetoothManager =
          (BluetoothManager)getSystemService(
          Context.BLUETOOTH_SERVICE);
        mBluetoothAdapter = bluetoothManager.getAdapter();
        ...
    }
    ...
    @Override
    protected void onPause() {
        super.onPause();
        scanLeDevice(false);
        ...
    }
    @Override
    protected void onListItemClick(ListView ...) {
        ...
        if(mScaning){
           mBluetoothAdapter.stopLeScan(mLeScanCallback);
        ...
    }
    private void scanLeDevice(final ...) {
        if(enable) {
            ...
            mBluetoothAdapter.startLeScan(mLeScanCallback);
        } else {
            mBluetoothAdapter.stopLeScan(mLeScanCallback);
            ...
        }
    }
    ...
    private BluetoothAdapter.LeScanCallback mLeScanCallback =
        new BluetoothAdapter.LeScanCallback() {
        @Override
        public void onLeScan(final BluetoothDevice ...) {
            ...
        }
    };
    ...
}
```

Fig. 3 A code fragment in Java for Android, taken from `DeviceScanActivity.java` in Google Samples.

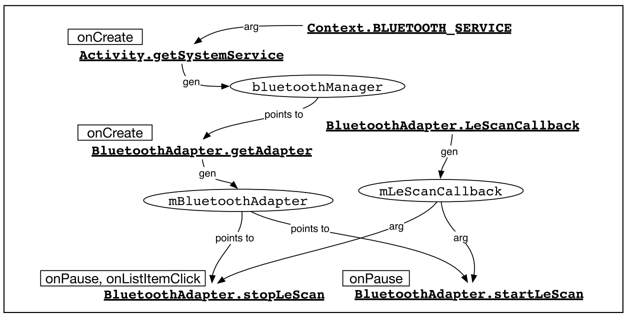

onCreate
Activity.getSystemService
arg
Context.BLUETOOTH_SERVICE
gen
bluetoothManager
points to
onCreate
BluetoothAdapter.getAdapter
BluetoothAdapter.LeScanCallback
gen
gen
mBluetoothAdapter
mLeScanCallback
points to
points to
arg
arg
onPause, onListItemClick
BluetoothAdapter.stopLeScan
onPause
BluetoothAdapter.startLeScan

Fig. 4 A graph illustrating closely related API members obtained from the code fragment in Fig. 3.

borders of methods, because the analysis is based on data dependence. In addition, to obtain globally reusable information, we use inline expansion to account for locally defined private methods.

A graph constructed according to data dependencies could be quite large, and an entire source file could be represented by just one graph. For event-driven programs interacting with the outside environment, however, which are typical for Android and iOS applications, the obtained graphs would be small, because such programs usually handle many separate events that are mutually independent.

3.2 Annotating and Weighting Each API Member Set

In general, multiple graphs are obtained from a source file. We

*1 In this paper, we use the word "API member" not only for methods of libraries or frameworks but also named constants defined in them.
*2 https://github.com/googlesamples

consider each graph such as that shown in Fig. 4, to represent a concrete example of the implementation of a particular functionality that can be treated as an idiom. Looking at the names of API members appearing in Fig. 4, we can guess that the functionality implemented by this set of API members is to "manage a Bluetooth adapter to start and stop scanning." From this intuition, we generate a set of words from the names of the API members in each set to make an annotation for the set.

After splitting API member names into smaller words by parsing common conventions such as camel case and snake case, we normalize them by using morphological analysis to remove variations in word forms. Then, to index the names for searching, we use the term frequency–inverse document frequency (TF-IDF) [12] to weight each word in each annotation. The *SSS-Editor* then uses the cosine distance measure in a vector space model to rank and reorder the results of an API member set search.

4. *SSS-Editor*: Iterating a *Search-Select-Superpose* Loop

Here, we describe the behavior of the prototype *SSS-Editor* shown in Fig. 2 (b), which controls the iteration of the SSS loop. The *SSS-Editor* uses the indexed dataset *SSS-DS* described in Section 3.

4.1 *Search* Phase

The user inputs a set of words that might represent a functionality to implement. As described in the previous section, the search results are ranked in order of similarity with respect to the query by using the cosine distance in a vector space model based on TF-IDF values. Currently, the query language is simply a sequence of words. Each word is morphologically normalized before evaluating cosine distances.

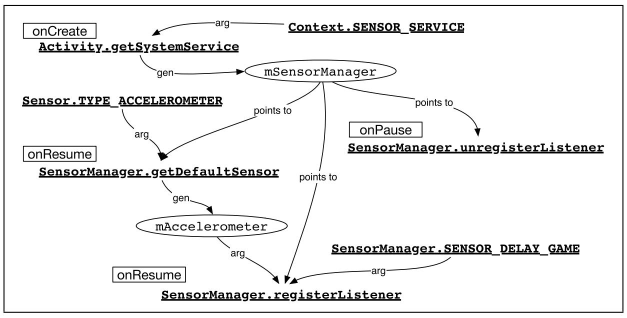
4.2 *Select* Phase

The result of a search is a list of sets of words. Each set of words, or outline, is associated with a set of API members with some information on their usages via a code skeleton. To simplify the selection process, we display the results as a list view of tag clouds. In each tag cloud, words that have large TF-IDF values are printed in large characters. Additional information such as the fully qualified names of API members is also supplied to the user, in case it is difficult to distinguish sets by only looking at the tag-cloud representations.

4.3 *Superpose* Phase

The *SSS-Editor* first inspects the user's code and gathers information such as definitions of variables and methods, and existing API calls. Then, it attempts to merge the selected code skeleton with the user's code. Roughly, the *SSS-Editor* attempts to

- insert each method call in the skeleton into the body of the same method from which the method call was obtained;
- use existing API members and variables as much as possible to embed the structure of the data dependence graphs into the code; and
- create variables to maintain intermediate objects and define methods to insert API calls, if needed.



onCreate
Activity.getSystemService
arg
Context.SENSOR_SERVICE
gen
mSensorManager
Sensor.TYPE_ACCELEROMETER
points to
points to
onPause
SensorManager.unregisterListener
arg
onResume
SensorManager.getDefaultSensor
points to
gen
mAccelerometer
arg
SensorManager.SENSOR_DELAY_GAME
arg
onResume
SensorManager.registerListener

Fig. 5 A graph obtained from a Java for Android file, `AccelerometerPlayActivity.java` in Google Samples.

The modification of the code is confirmed by the user at each step.

Suppose we merge the code skeleton shown in **Fig. 5**. The *SSS-Editor* detects constraints such as calls to the methods `registerListener` and `getDefaultSensor` having to be inserted in the method `onResume`, a call to `unregisterListener` in `onPause`, and so on. If some methods in which API calls are going to be inserted are not defined in the code at hand, the *SSS-Editor* introduces new definitions of them (possibly with an `@Override` annotation). In some applications, overridden methods such as `onCreate` might not be the same as the ones in a code skeleton. For example, some applications might be constructed by extending `AppCompatActivity` or `ListActivity` instead of `Activity`. The *SSS-Editor* thus checks for type compatibilities and notifies the user if manual modification of the source is required.

Arguments and return values for inserted calls are considered so as to maintain the relations represented in the graph. For example, in the case of the graph shown in Fig. 5, the *SSS-Editor* searches for an identifier of type `Sensor` in the code at hand to arrange the call to `registerListener`. If it is found in `onResume`, then the *SSS-Editor* tries to use it. If it is not, or if the user does not let the *SSS-Editor* use it, the tool defines a fresh identifier in the appropriate scope and uses it.

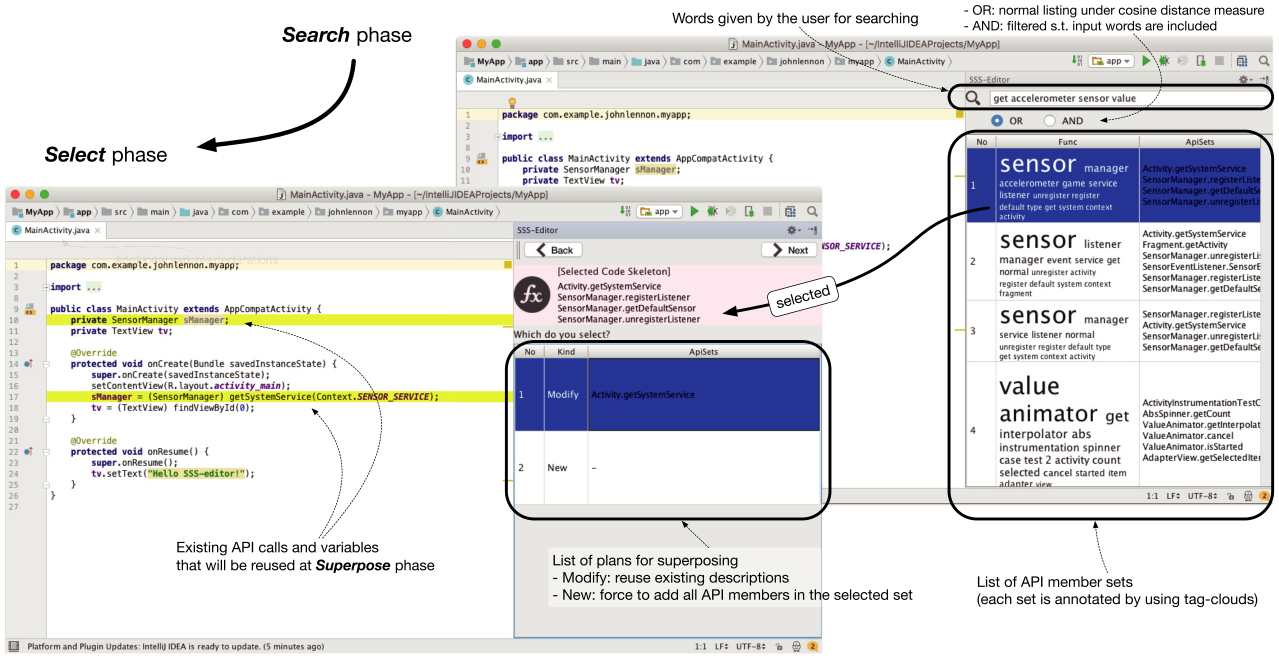
In general, it is impossible to build a fully automatic tool for the *Superpose* phase that will always work correctly, at least in the sense that the behavior of the resultant code will be exactly the same as the user intends. Thus, we offer two strategies for superposition: one is to reuse API calls in the code at hand as much as possible, and the other is to insert all API calls in the selected code skeleton.

5. Implementation

We have developed a prototype of the system shown in Fig. 2. The Java parser in the Eclipse Java development tools (JDT) and Apache Lucene were used to construct the *SSS-DSGen* and *SSS-Editor*. The *SSS-Editor* was then implemented as a plugin for Android Studio.

Figure 6 shows the appearance of the *SSS-Editor*. The user can edit the source code in the left pane. The right pane is used to control searching, selecting, and superposing.

Figure 6 shows a situation in which the user searched for a functionality by inputting the words "get accelerometer sensor value" with a *SSS-DS* constructed by analyz-


Search phase
Select phase
Words given by the user for searching
- OR: normal listing under cosine distance measure
- AND: filtered s.t. input words are included
MainActivity.java - MyApp - [~/IntelliJIDEAProjects/MyApp]
MyApp 〉 app 〉 src 〉 main 〉 java 〉 com 〉 example 〉 johnlennon 〉 myapp 〉 MainActivity 〉
MainActivity.java
app
SSS-Editor
OR
AND
No
Func
ApiSets
Back
Next
selected
Which do you select?
No
Kind
ApiSets
1:1 LF UTF-8
Platform and Plugin Updates: IntelliJ IDEA is ready to update. (5 minutes ago)
Existing API calls and variables
that will be reused at Superpose phase
List of plans for superposing
- Modify: reuse existing descriptions
- New: force to add all API members in the selected set
List of API member sets
(each set is annotated by using tag-clouds)

```
package com.example.johnlennon.myapp;

import ...

public class MainActivity extends AppCompatActivity {
    private SensorManager sManager;
    private TextView tv;
```

get accelerator sensor value

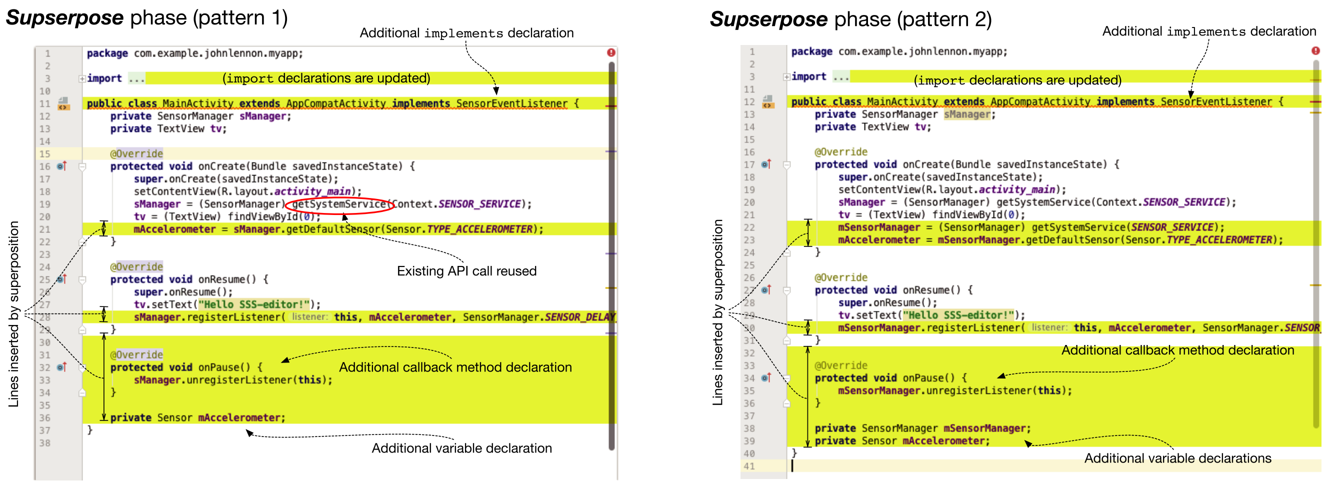
No	Func	ApiSets
1	sensor manager accelerometer game service listener unregister register default type get system context activity	Activity.getSystemService SensorManager.registerListe SensorManager.getDefaultSe SensorManager.unregisterLi
2	sensor listener manager event service get normal unregister activity register default system context fragment	Activity.getSystemService Fragment.getActivity SensorManager.unregisterLi SensorEventListener.SensorE SensorManager.registerListe SensorManager.getDefaultSe
3	sensor manager service listener normal unregister register default type get system context activity	SensorManager.registerListe Activity.getSystemService SensorManager.unregisterLi SensorManager.getDefaultSe
4	value animator get interpolator abs instrumentation spinner case test 2 activity count selected cancel started item adapter view	ActivityInstrumentationTestC AbsSpinner.getCount ValueAnimator.getInterpolat ValueAnimator.cancel ValueAnimator.isStarted AdapterView.getSelectedIter

```
package com.example.johnlennon.myapp;

import ...

public class MainActivity extends AppCompatActivity {
    private SensorManager sManager;
    private TextView tv;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        sManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        tv = (TextView) findViewById(0);
    }

    @Override
    protected void onResume() {
        super.onResume();
        tv.setText("Hello SSS-editor!");
    }
}
```

[Selected Code Skeleton]
Activity.getSystemService
SensorManager.registerListener
SensorManager.getDefaultSensor
SensorManager.unregisterListener

No	Kind	ApiSets
1	Modify	Activity.getSystemService
2	New	–

Fig. 6 Iteration of the SSS loop by using the prototype *SSS-Editor*.


Supserpose phase (pattern 1)
Additional implements declaration
(import declarations are updated)
Existing API call reused
Additional callback method declaration
Additional variable declaration
Lines inserted by superposition
Supserpose phase (pattern 2)
Additional implements declaration
(import declarations are updated)
Additional callback method declaration
Additional variable declarations
Lines inserted by superposition

```
package com.example.johnlennon.myapp;

import ...

public class MainActivity extends AppCompatActivity implements SensorEventListener {
    private SensorManager sManager;
    private TextView tv;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        sManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        tv = (TextView) findViewById(0);
        mAccelerometer = sManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    }

    @Override
    protected void onResume() {
        super.onResume();
        tv.setText("Hello SSS-editor!");
        sManager.registerListener( listener: this, mAccelerometer, SensorManager.SENSOR_DELAY
    }

    @Override
    protected void onPause() {
        sManager.unregisterListener(this);
    }

    private Sensor mAccelerometer;
}
```

(a) Reusing existing descriptions

```
package com.example.johnlennon.myapp;

import ...

public class MainActivity extends AppCompatActivity implements SensorEventListener {
    private SensorManager sManager;
    private TextView tv;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        sManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        tv = (TextView) findViewById(0);
        mSensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
        mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    }

    @Override
    protected void onResume() {
        super.onResume();
        tv.setText("Hello SSS-editor!");
        mSensorManager.registerListener( listener: this, mAccelerometer, SensorManager.SENSOR
    }

    @Override
    protected void onPause() {
        mSensorManager.unregisterListener(this);
    }

    private SensorManager mSensorManager;
    private Sensor mAccelerometer;
}
```

(b) Inserting all API members in the selected set

Fig. 7 Superposition: two kinds of strategies.

ing files from Google Samples (upper right). The user then selected the first candidate, which has four methods (lower left): `getSystemService`, `getDefaultSensor`, `registerListener`, and `unregisterListener`. Once an API member set is selected, the *SSS-Editor* offers two kinds of strategies to modify the user's code in the *Superpose* phase. In each case, the user can see beforehand how the code would be modified, as shown in **Fig. 7**.

6. Evaluation

In this section, we evaluate the effectiveness of the *SSS loop approach* for developing Android application programs. The approach was implemented with a system consisting of two components, i.e., the *SSS-DSGen* and *SSS-Editor*, as shown in Fig. 2. To evaluate the effectiveness of the whole system, we conducted the following two experiments:

(1) an experiment on the automatic reproduction of API usage patterns in existing applications, and
(2) a user study involving Java programmers.

The purpose of the first experiment was to evaluate the quality of search results obtained by the proposed system. In the experiment, we attempted to reproduce API usage patterns in existing applications using the *SSS-Editor* in an automated manner. We evaluated the quality of search results by two specially defined index: recall and precision on API usage patterns.

The second experiment was intended to evaluate the overall performance of our system. Each participant in the user study was given the task of building four tiny Android programs with or without the *SSS-Editor*. We compared the results obtained from both environments and evaluated the usability of the proposed system.

Both experiments were based on *SSS-DSs* constructed from Java source files in Google Samples. The experimental environment is detailed in Section 6.1. The results of the experiments are described in Sections 6.2 and 6.3.

6.1 Environment

For the experiments described in the following sections, we used Java files obtained from Google Samples in January 2017. We collected 158 projects containing 1,797 Java files. For the experiments, we selected all files that define a class extending the Android framework and use at least one Android API call, for a total of 853 files. Here, we refer to the set of selected files as *Samples*.

6.2 Experiment 1: Reproduction of API Usage Patterns

In Experiment 1, we evaluated the quality of search results obtained by the proposed system. In this experiment, we attempted to reproduce each file (called a *target*) in the *Samples* by iterating the SSS loop with a *SSS-DS* constructed from the files in the *Samples* excluding the target file. To measure the quality of the search results, we compared the differences between the reproduced and the original (target) file in terms of what API members are used in what methods, which we call *API usage patterns (API-UPs)*.

In the rest of this section, we use the phrase "reproduced API-UPs" to refer to the API-UPs extracted from the reproduced file.

6.2.1 Description of Experiment

Experiment 1 was conducted as follows. First, we selected a file from the *Samples* as a target file. Second, we extracted a set of API-UPs (called *target API-UPs*) from the target file, and a set of words (called *keywords*) from the API member names in each target API-UP. Then, after constructing a *SSS-DS* from the *Samples* excluding the target file, we attempted to reproduce the API-UPs in the target by inputting the corresponding sets of keywords repeatedly to the *SSS-Editor*. For each target file, we compared the reproduced API-UPs with the target API-UPs. In this experiment, these steps were repeated for all files in the *Samples*.

Note that one search result consists of a candidate list of outlined API member sets with code skeletons. In ordinary use of the proposed system, the user is responsible for choosing an API member set from the list of candidates. In Experiment 1, however, *we chose the set at the top of the candidate list*, so that iterated experiments for all files in the *Samples* could be carried out mechanically without user intervention. Thus, if many of the target API-UPs were reproduced in Experiment 1, we could conclude that the quality of search results obtained by the proposed system was high.

The following is a more detailed description of the iteration for each target file in the *Samples*:

(1) Denote the target file as f. Construct a *SSS-DS* from the remaining 852 files in the *Samples*. If f uses an Android API member that does not appear in the other 852 files, we skip f, because some API-UPs obtained from it could never be reproduced.

(2) Extract sets of keywords that describe functionalities implemented in f. This is done by first extracting API-UPs and then extracting sets of keywords from them. Each keyword is normalized through a morphological analysis. Each keyword list is sorted in order of frequency. We denote the number of functionalities in f and the list of keywords corresponding to the i-th functionality as $N^{(f)}$ and $K_i^{(f)}$, respectively. In this step, we obtain a list $L^{(f)} = [K_1^{(f)}, \ldots, K_{N^{(f)}}^{(f)}]$, where $L^{(f)}$ is sorted such that $i \le j$ implies $|K_i^{(f)}| \ge |K_j^{(f)}|$.

(3) Iterate the SSS loop mechanically with the *SSS-Editor*. From each element $K_i^{(f)}$ in $L^{(f)}$, a fixed number of words (e.g., 3, 4, or all $|K_i^{(f)}|$ words) is used to invoke the *SSS-Editor*. In the *Select* phase, the code skeleton at the top of the search results is always chosen, and API members in that skeleton are inserted in the *Superpose* phase. Iteration of the SSS loop continues until all elements in $L^{(f)}$ have been processed.

Let $\tilde{f}$ denote the file reproduced from the sets of keywords obtained from f at each iteration of the above steps. At each iteration, we record the following two values, the *recall*(f) and *precision*(f):

$$recall(f) = \frac{|\ api\_pos\_pairs(\tilde{f}) \cap api\_pos\_pairs(f)\ |}{|\ api\_pos\_pairs(f)\ |},$$

$$precision(f) = \frac{|\ api\_pos\_pairs(\tilde{f}) \cap api\_pos\_pairs(f)\ |}{|\ api\_pos\_pairs(\tilde{f})\ |},$$

where $api\_pos\_pairs(f)$ denotes a set of pairs in which each pair (a, m) indicates that an API member a is used in a method m defined in f. The value of $recall(f)$ is thus the ratio of the number of pairs in the reproduced set of API-UPs to the number of pairs in API-UPs obtained from the target file f. On the other hand, the value of $precision(f)$ is the ratio of the number of pairs in the reproduced set of API-UPs to the number of pairs in API-UPs obtained from the reproduced file $\tilde{f}$. In the ideal case, $recall(f)$ and $precision(f)$ are both 1, where $api\_pos\_pairs(f)$ is equal to $api\_pos\_pairs(\tilde{f})$.

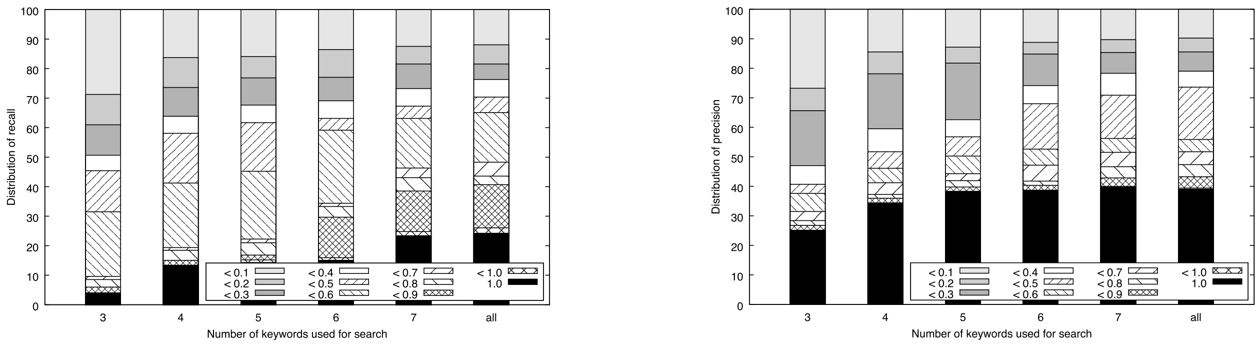
6.2.2 Results of Experiment 1

Figure 8 summarizes the experimental results. In total, reproduction was tried on 553 files. Each stacked bar in Fig. 8 (a) shows the distribution of the values of *recall* for a given number of keywords used in step (3). In the figure, the black part forming the lowest layer in each stacked bar indicates the proportion of target files whose *recall* (or *precision*) was 1 when the given number of keywords was used. The other layers indicate the proportion of target files whose values were in the ranges of $[0.9, 1.0)$, $[0.8, 0.9)$, ..., $[0, 0.1)$, from the bottom to the top. For example, Fig. 8 (a) shows that the proportion of target files whose *recall* values were in the range of $[0.5, 1]$ was about 31% (174 files out of 553) when three keywords were used in the *Search* phase. In other words, more than half the API members in 32% of the files were correctly (in terms of API-method pairs) inserted by the *SSS-Editor*. When a larger number of keywords were used, the *recall* became significantly greater. If we could input all words related to API members in the *Search* phase, more than half of them in 65% of the files (360 out of 553) would be correctly inserted automatically.

Figure 8 (b) similarly shows the distribution of *precision* values. The graph indicates that if we could input enough keywords in the *Search* phase, the results from the *SSS-Editor* would be expected to contain only a small number of irrelevant API members or incorrect insertions.

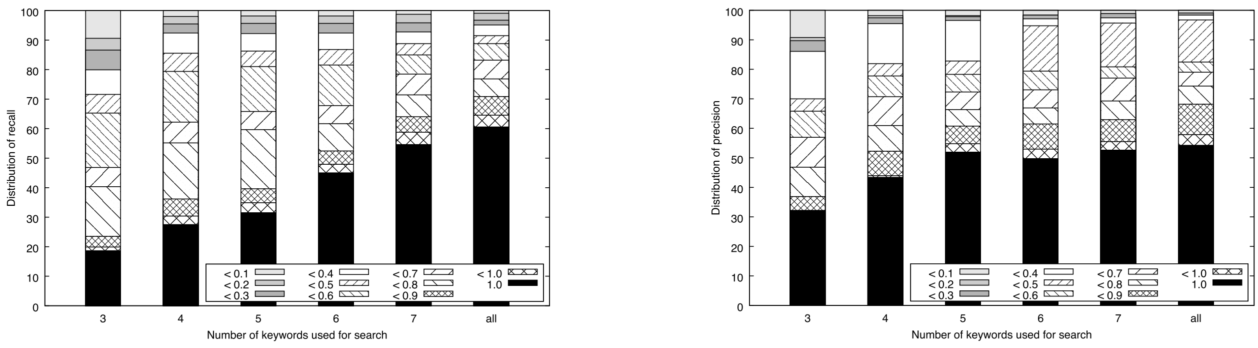
6.2.3 Properties of API-UPs as Plain API Member Sets

Each API-UP obtained in Experiment 1 contained valuable in-



(a) Distribution of the values of *recall* (b) Distribution of the values of *precision*

Fig. 8 Distribution of the values of *recall* and *precision* for files in Google Samples.



(a) Distribution of the values of *recall′* (b) Distribution of the values of *precision′*

Fig. 9 Distribution of the values of *recall′* and *precision′* for files in Google Samples.

formation on what API members were used in what methods for implementing a functionality. However, even limited information on those API members as plain API member sets could be still considered useful [21]. In this section, we discuss the properties of API-UPs as plain API member sets from the results obtained by iterating the process described in Section 6.2.1.

Figure 9 shows the experimental results with the following slightly modified versions of *precision* and *recall*:

$$recall'(f) = \frac{|\, apis(\tilde{f}) \cap apis(f) \,|}{|\, apis(f) \,|},$$
$$precision'(f) = \frac{|\, apis(\tilde{f}) \cap apis(f) \,|}{|\, apis(\tilde{f}) \,|},$$

where $apis(f)$ denotes just the set of Android API members used in f, without information on their inserted positions. Obviously, values of *precision′* and *recall′* will be greater than *precision* and *recall*, respectively. Figure 9 (a) shows the distribution of the values of *recall'*. It indicates, for example, that when three words were used in the *Search* phase, more than half the API members in 65% of the files (361 out of 553) were gathered by the *SSS-Editor*. If we could input all words related to API members in the *Search* phase, more than half of them in 89% of the files (491 out of 553) would be automatically obtained. Likewise, Fig. 9 (b) shows the distribution of the values of *precision′*. The graph indicates that if we could input enough keywords in the *Search* phase, the results from the *SSS-Editor* would not contain many irrelevant API members.

6.2.4 Comparison with API Member Sets Obtained by Another Method

An API member set search tool, which we refer to as CAPIS in this paper, was studied in a previous work [21]. The CAPIS tool enables the user to search for API member sets that are expected to be necessary to implement a desired functionality. The sets obtained by CAPIS are supposed to be full of information that is not easy to gather by just reading through framework documentation or searching open-source repositories [21]. In this section, we compare the properties of API member sets obtained using the CAPIS tool [21] and those of API-UPs obtained using the proposed method.

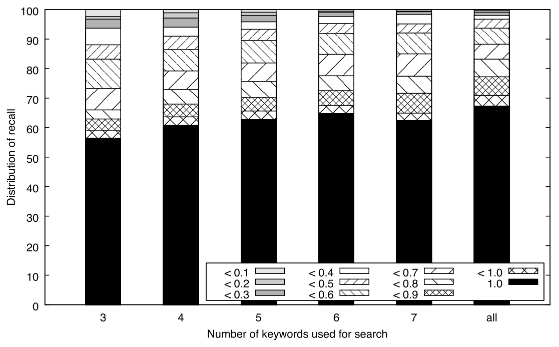
The method in Ref. [21] for extracting API member sets does not use data dependence. Rather, API members are clustered depending on whether they are members of similarly named classes. Thus, the *Superpose* phase using the *SSS-Editor* is not applicable to API member sets obtained by CAPIS. We thus partially applied the three-step procedure described in Section 6.2.1 to the API member sets, i.e., we retrieved API member sets without reproducing each target file in the *Superpose* phase.

The CAPIS results for *recall′* and *precision′* are shown in **Fig. 10** (a) and Fig. 10 (b), respectively. Compared to the results shown in Fig. 9, this figure shows peculiar properties for the dataset obtained by the method of Ref. [21]. For example, *recall′* was high and *precision′* was low, (almost) independently of the number of words used for the search. This implies that API member set information obtained from data dependence is preferable for implementing functionalities to that obtained from similarities of class names.

6.2.5 Summary of Experiment 1 Results

The results of Experiment 1 can be summarized as follows.

- By using the proposed system, information could be effectively obtained on necessary API members and usage pat-


(a) Distribution of the values of *recall′*

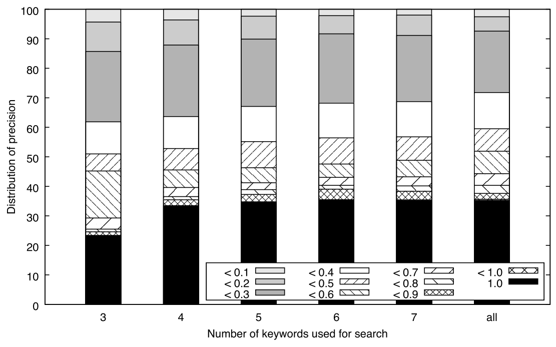

(b) Distribution of the values of *precision′*

Fig. 10 Distribution of the values of *recall′* and *precision′* for API member sets generated by the method of Ref. [21].

terns for constructing typical Android application programs using files such as those in Google Samples.

- Compared to a method for obtaining sets of API members from the class names to which each member belongs [21], the proposed method for extracting API usage patterns from data dependence gave API member set search results that were more precise in terms of *precision′* and *recall′*.

Note that the results shown in Fig. 8 and Fig. 9 were obtained with fully automatic iteration of the SSS loop, which is not the intended usage of the *SSS-Editor*. In fact, automatic superposition often causes it to generate semantically incorrect code. The results suggest, however, that the *SSS-Editor* has the potential to help the user construct outlines of application programs that are often full of API calls.

6.3 Experiment 2: User Study

6.3.1 Description of Experiment

Next, we conducted a user study involving Java programmers. All participants were university students [*3] who had experience in writing Java programs but were beginners in Android programming.

The participants were given tasks of developing tiny Android applications in the following two environments:

E1 The participants could use the Android API reference, Android programs in Google Samples with a simple search facility, and Android Studio.

E2 In addition to the content of **E1**, the participants could use the *SSS-Editor*.

By comparing the scores obtained in environments **E1** and **E2**, we examined the effectiveness of the *SSS-Editor*.

Each task given to the participants consisted of implementing particular functionalities in an empty Android project. Layout designs of appropriate widgets on the screen, such as Button and EditText widgets, were given in advance via XML files. Required permissions for connecting to the Internet and Wi-Fi were given via `AndroidManifest.xml`. Detailed descriptions of the tasks are as follows.

P1 Make the app open Google's homepage (https://www.google.com/) when a button on the screen is pushed.

P2 Assign two buttons separate functionalities: one is used for connecting to Wi-Fi, and the other, for disconnecting.

P3 Make the app pop up a text string in an EditText widget by using a Toast widget.

P4 Make the app rotate a picture (shown in advance) by 360 degrees via rotating animation when a button on the screen is pushed.

[*3] The participants were seven graduates and one undergraduate at Hiroshima City University.

Table 1 Task and environment assignments to the participants.

Participants	1st solve	2nd solve	3rd solve	4th solve
A and B	P1	P2	P3	P4
	E2	E2	E1	E1
C and D	P3	P4	P1	P2
	E1	E1	E2	E2
E and F	P1	P2	P3	P4
	E1	E1	E2	E2
G and H	P3	P4	P1	P2
	E2	E2	E1	E1

Each task had to be finished within a 20-minute period. All necessary API members to solve the problems were included in the *SSS-DS* given for **E2**. The participants' operations and behaviors were recorded.

Table 1 lists the task and environment assignments for the participants. The experiments were conducted with the procedure below.

(1) We gave the participants a simple lecture on Android application development.
(2) We gave a simple lecture on the usage of the *SSS-Editor*.
(3) The participants were given some Android programming practice.
(4) The participants performed tasks according to the assignments listed in Table 1.

6.3.2 Results of Experiment 2

Table 2 lists the results of Experiment 2. Each line in Table 2 corresponds to a record for one participant. Boxes denote that the trial was done with the *SSS-Editor*. The participants rated the usefulness of the system on a scale from 1 to 5. From the results listed in Table 2, we calculated the ratio of correct answers with the *SSS-Editor* as 14 : 16 (87.5%). On the other hand, the ratio of correct answers without the *SSS-Editor* was 7 : 16 (43.8%).

There were a few cases in which participants could not build correct programs because of a lack of experience with Java or Android programming (†1).

In environment **E1** (without the *SSS-Editor*), some participants could not build correct programs because of failures in the search

Table 2 User study results, indicating the time in minutes required to solve each problem.

Participants	P1	P2	P3	P4	PCA	Rating (1-5)
A	$\boxed{16}$	$\boxed{8}$	—†3	—†2	50.0%	4
B	$\boxed{10}$	$\boxed{7}$	—†3	—†2	50.0%	4
C	$\boxed{13}$ †6	$\boxed{8}$	—†1	17	75.0%	4
D	$\boxed{\text{—}}$ †5	$\boxed{6}$	—†3	—†2	25.0%	4
E	8	18	$\boxed{9}$	$\boxed{9}$	100.0%	3
F	9	—†4	$\boxed{6}$	$\boxed{14}$ †6	75.0%	4
G	11	6	$\boxed{8}$	$\boxed{13}$ †6	100.0%	3
H	10	—†4	$\boxed{\text{—}}$ †1	$\boxed{4}$	50.0%	4
PCA	87.5%	75.0%	37.5%	62.5%	65.6%	

PCA: Percentage of Correct Answers

for suitable API members (†2, †3, †4). Some had trouble searching APIs because the available search facilities in **E1** were too sensitive to parts of speech, e.g., they should have used "animate" instead of "anime" or "animation" (†2). Some failed to judge suitable API members as correct even though they noticed them while searching (†3). Some could not compose a set of API members by gathering pieces of information obtained from search results to implement a functionality in the code (†4).

On the other hand, in **E2** (with the *SSS-Editor*), participants did not make the kinds of mistakes described above, i.e., important API calls were easily obtained using the *SSS-Editor*. Because an automatically inserted set of sentences sometimes contained needless API calls, however, the participants had to pay attention to remove unnecessary ones. In one case the removal was not performed correctly (†5).

Also, in some cases in **E2**, participants initially misjudged correct API members but changed the decision while taking a second look at them (†6). In **E1**, however, no one paid attention to API members that had been misjudged.

6.3.3 Summary of Experiment 2 Results

From the results listed in Table 2, we can conclude that the proposed system for developing applications was effective. The results are summarized as follows.

- The use of morphological analysis in constructing a *SSS-DS* and searching was effective (to avoid cases like †2).
- The facility to present sets of mutually related API members was useful (to avoid cases like †4).
- The facility to automatically insert API member sets into the code at hand could appropriately help the user develop applications rapidly (to avoid cases like †3 and †4).
- The technique of listing search results in the *Search* phase, i.e., the organized style of showing information to the user, might also be effective for helping the user build applications smoothly (to make cases like †6 possible).

In addition, note the following:

- the participants rated the tool fairly well, with six of eight rating it at 4 (in the range from 1 to 5), but
- two participants who got perfect scores rated the tool at 3.

This result implies that the current prototype of the system is especially useful for beginners in Android programming. In fact, the problems in Experiment 2 were small, and many of them could be easily solved by experienced programmers. A detailed analysis based on experiments using more difficult or large-scale programming projects will be a future work.

7. Related Work

Code search techniques for retrieving information from open-source repositories to support application development have been studied for more than a decade. To the best of our knowledge, however, there have been virtually no systems with functionalities similar to ours, i.e., systems that can search for API usage patterns involving multiple API calls in separate methods and also merge the set of API members into the user's code. In this section, we summarize related studies on information retrieval for API usage.

There are many kinds of approaches to define queries for retrieving API-related information. SNIFF [1] accepts words and searches for code segments related to the input words. For searching, it uses API documentation in addition to comments in source files. Portfolio [15] uses natural language processing (NLP) to retrieve information on chains of API calls. Raychev et al. used statistical language models to accomplish code completion [24]. CodeBroker [30] uses the source code at hand to give contextual information to retrieve code segments, as does Strathcona [5]. Prompter [22] also uses the user's source code at hand and automatically retrieves relevant discussions with code segments from Stack Overflow, a Q&A website. CodeHint [2] even uses dynamic context to offer appropriate code. Lemos et al. proposed an approach of test-driven code search [9], [10], which accepts descriptions for unit testing and obtains suitable method definitions. The system presented in this paper currently just uses morphological analysis for words. Sophisticated query processing would also be effective for our system.

In addition to searching for information related to API usage, some tools synthesize appropriate code segments. Some use simple approaches: Prospector [11] composes unary functions to obtain a "jungloid" by accepting various types of input and output, while XSnippet [26] limits the type of generated code segments to those performing object instantiation tasks. PARSEWeb [28] also accepts a signature for searching and is claimed to outperform Prospector and XSnippet. S^6 [25] accepts additional information beyond signatures for searching API usages. Recent approaches are more advanced: Hunter [29] generates wrappers when it cannot find API members that exactly match the given signatures. AnyCode [3] accepts a description in natural language and synthesizes code that can be inserted at a designated point in the user's code. DroidAssist [17] uses a hidden Markov model (HMM) to recommend API usages for Android applications. SWIM [23] also uses NLP with log data obtained from Bing, a general-purpose search engine. T2API [19] uses a statistical machine translation technique to accept descriptions in English and synthesize code.

Many of these studies utilize data and control dependence analysis for retrieving information on API usage patterns. For example, the above-mentioned DroidAssist [17] uses a graph-based representation of patterns, which is obtained based on data and control dependence analysis described in Ref. [20], in order to extract a statistical model of API usages. LibSync [16], which is a tool for supporting API usage adaptation, is also based on the approach of Ref. [20] with extensions to deal with subtyping relations between objects. Prospector [11] performs data and control dependence analysis similar to that of Ref. [16]. Different from these tools, our method performs data dependence analysis that is more global in the sense that API members invoked in user-defined methods are related based on data dependence analysis beyond the borders of method definitions.

Keivanloo et al. [7] offered a way to obtain descriptions that might be small code segments without API calls. Gilligan [6] is useful in integrating complicated descriptions of a functionality in an application into the code at hand. An approach called multistaging to understand (MTU) [27] is helpful for understanding an existing source code to obtain information for reusing part of it. Our future work will include incorporating these kinds of advanced functionalities into our system.

In addition to the above works, there was one approach [13] that appeared quite similar to ours, but their project went in a different direction [14].

8. Conclusion

We have proposed a simple way to develop applications by iterating a three-step procedure called the *Search-Select-Superpose* (SSS) loop. A prototype tool to support the development process was designed and implemented. Experimental results showed that the proposed approach is effective, partly because of the simple nature of event-driven applications. Our future work will include a detailed evaluation of the approach and a tool for developing large-scale applications. We plan to test the approach with other frameworks based on event-driven programming, such as iOS and GUI applications.

References

[1] Chatterjee, S., Juvekar, S. and Sen, K.: SNIFF: A Search Engine for Java Using Free-Form Queries, *Proc. International Conference on Funcamental Approaches to Software Engineering* (*FASE 2009*), pp.385–400 (2009).

[2] Galenson, J., Reames, P., Bodik, R., Hartmann, B. and Sen, K.: CodeHint: Dynamic and Interactive Synthesis of Code Snippets, *Proc. 36th International Conference on Software Engineering*, *ICSE 2014*, pp.653–663, ACM (online), DOI: 10.1145/2568225.2568250 (2014).

[3] Gvero, T. and Kuncak, V.: Synthesizing Java Expressions from Free-Form Queries, *Proc. 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.416–432 (2015).

[4] Hoffmann, R., Fogarty, J. and Weld, D.S.: Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers, *UIST'07*, pp.13–22 (2007).

[5] Holmes, R. and Murphy, G.C.: Using Structural Context to Recommend Source Code Examples, *Proc. 27th International Conference on Software Engineering*, *ICSE '05*, pp.117–125, ACM (online), DOI: 10.1145/1062455.1062491 (2005).

[6] Holmes, R. and Walker, R.J.: Systematizing pragmatic software reuse, *ACM Trans. Software Engineering and Methodology* (*TOSEM*), Vol.21, No.4 (2012).

[7] Keivanloo, I., Rilling, J. and Zou, Y.: Spotting Working Code Examples, *Proc. 36th International Conference on Software Engineering*, *ICSE 2014*, pp.664–675, ACM (online), DOI: 10.1145/2568225.2568292 (2014).

[8] Lamba, Y., Khattar, M. and Sureka, A.: Pravaaha: Mining Android Applications for Discovering API Call Usage Patterns and Trends, *Proc. 8th India Software Engineering Conference* (*ISEC'15*), pp.10–19 (2015).

[9] Lazzarini Lemos, O.A., Bajracharya, S., Ossher, J., Masiero, P.C. and Lopes, C.: A Test-driven Approach to Code Search and Its Application to the Reuse of Auxiliary Functionality, *Inf. Softw. Technol.*, Vol.53, No.4, pp.294–306 (online), DOI: 10.1016/j.infsof.2010.11.009 (2011).

[10] Lemos, O.A.L., Bajracharya, S., Ossher, J., Masiero, P.C. and Lopes, C.: Applying Test-Driven Code Search to the Reuse of Auxiliary Functionality, *Proc. 2009 ACM Symposium on Applied Computing* (*SAC'09*), pp.476–482 (2009).

[11] Mandelin, D., Xu, L., Bodík, R. and Kimelman, D.: Jungloid Mining: Helping to Navigate the API Jungle, *Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, *PLDI '05*, pp.48–61, ACM (online), DOI: 10.1145/1065010.1065018 (2005).

[12] Manning, C.D., Raghavan, P. and Schutze, H.: *Introduction to Information Retrieval*, Cambridge University Press (2008).

[13] McMillan, C.: Searching, Selecting, and Synthesizing Source Code, *Proc. 33rd International Conference on Software Engineering*, *ICSE '11*, pp.1124–1125, ACM (online), DOI: 10.1145/1985793.1986013 (2011).

[14] McMillan, C.: Searching, Selecting, and Synthesizing Source Code Components, PhD Thesis, The College of William and Mary (2012).

[15] McMillan, C., Poshyvanyk, D., Grechanik, M., Xie, Q. and Fu, C.: Portfolio: Searching for Relevant Functions and Their Usages in Millions of Lines of Code, *ACM Trans. Software Engineering and Methodology*, Vol.22, No.4, Article 37 (2013).

[16] Nguyen, H.A., Nguyen, T.T., Wilson, Jr., G., Nguyen, A.T., Kim, M. and Nguyen, T.N.: A Graph-based Approach to API Usage Adaptation, *Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications*, *OOPSLA '10*, pp.302–321, ACM (online), DOI: 10.1145/1869459.1869486 (2010).

[17] Nguyen, T.T., Pham, H.V., Vu, P.M. and Nguyen, T.T.: Recommending API Usages for Mobile Apps with Hidden Markov Model, *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2015*), pp.795–800 (2015).

[18] Nguyen, T.T., Pham, H.V., Vu, P.M. and Nguyen, T.T.: Learning API usages from bytecode: A statistical approach, *Proc. 38th International Conference on Software Engineering* (*ICSE '16*), pp.416–427 (2016).

[19] Nguyen, T., Rigby, P.C., Nguyen, A.T., Karanfil, M. and Nguyen, T.N.: T2API: Synthesizing API Code Usage Templates from English Texts with Statistical Translation, *Proc. 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, *FSE 2016*, pp.1013–1017, ACM (online), DOI: 10.1145/2950290.2983931 (2016).

[20] Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M. and Nguyen, T.N.: Graph-based Mining of Multiple Object Usage Patterns, *ESEC-FSE'09* (2009).

[21] Nishimoto, M., Kawabata, H. and Hironaka, T.: A System for API Set Search for Supporting Application Program Development (in Japanese), *IEICE Trans. Inf. Syst.*, Vol.J101-D, No.8, pp.1176–1189 (2018).

[22] Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R. and Lanza, M.: Prompter, *Empirical Softw. Engg.*, Vol.21, No.5, pp.2190–2231 (online), DOI: 10.1007/s10664-015-9397-1 (2016).

[23] Raghothaman, M., Wei, Y. and Hamadi, Y.: SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis, *Proc. 38th International Conference on Software Engineering*, *ICSE '16*, pp.357–367, ACM (online), DOI: 10.1145/2884781.2884808 (2016).

[24] Raychev, V., Vechev, M. and Yahav, E.: Code Completion with Statistical Language Models, *Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'14*), pp.419–428 (2014).

[25] Reiss, S.P.: Semantics-Based Code Search, *Proc. 31st International Conference on Software Engineering* (*ICSE'09*), pp.243–253 (2009).

[26] Sahavechaphan, N. and Claypool, K.: XSnippet: Mining For Sample Code, *Proc. 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, *OOPSLA '06*, pp.413–430, ACM (online), DOI: 10.1145/1167473.1167508 (2006).

[27] Sanchez, H., Whitehead, J. and Schäf, M.: Multistaging to Understand: Distilling the Essence of Java Code Examples, *Proc. IEEE 24th International Conference on Program Comprehension* (2016).

[28] Thummalapenta, S. and Xie, T.: Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web, *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering*, *ASE '07*, pp.204–213, ACM (online), DOI: 10.1145/1321631.1321663 (2007).

[29] Wang, Y., Feng, Y., Martins, R., Kaushik, A., Dillig, I. and Reiss, S.P.: Hunter: Next-Generation Code Reuse for Java, *Proc. 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*FSE 2016*), pp.1028–1032 (2016).

[30] Ye, Y. and Fisher, G.: Supporting Reuse by Delivering Task-Relevant and Personalized Information, *Proc. 24th International Conference on Software Engineering* (*ICSE'02*), pp.513–523 (2002).

Masashi Nishimoto received B.E. degree from Hiroshima City University in 2009. He is currently a doctor course student at Graduate School of Information Sciences in Hiroshima City University since 2016. His research interests include software engineering. He is a member of IPSJ, and JSSST.

Keiji Nishiyama received B.E. degree from Hiroshima City University in 2018. He is currently a master course student at Graduate School of Information Sciences in Hiroshima City University since 2018. His research interests include software engineering.

Hideyuki Kawabata received B.E. and Ph.D. degrees from Kyoto University in 1992 and 2004, respectively. Since 2007, he has been a lecturer at Hiroshima City University. His research interests include numerical programming and programming languages. He is a member of ACM, IEEE Computer Society, IPSJ, IEICE, JSIAM, and JSSST.

Tetsuo Hironaka received a Ph.D. degree from Kyushu University in 1993. From 1993 to 1994, he served as a research associate at Kyushu University. From 1994 to 2006, he was an associate professor at Hiroshima City University. Since 2006, he has been a professor at Hiroshima City University. His research interests include computer architectures, reconfigurable architectures, and software engineering. He is a member of IPSJ, IEICE, IEEE, and ACM.