A Multi-view API Impact Analysis for Open SPL Platform

Eunyoung Kim*, Kangtae Kim**, Hoh Peter In*

* Department of Computer Science and Engineering, Korea University
  Anam-dong Seongbuk-gu, Seoul, 136-701 - Korea
** Samsung Electronics
  Maetan-dong 461 yountong-uu, Suwon - Korea
  marine0704@korea.ac.kr, kangtae.kim@samsung.com, hoh-in@korea.ac.kr

Abstract — Open platform is introduced and become popular in Consumer Electronics domain, which rise API maintenance issues. As a platform evolves, API changes can break existing applications. Some tools and ideas have been proposed to solve the migration of application.

However it is critical to maintain API which changes by requesting new feature.

This paper proposes a multi-view impact analysis model based for platform maintenance. This model trace between APIs and their correspondent work products such as implementations, test case and applications which use APIs

Keywords — API, Impact Analysis, platform evolution, open platform, SPL

I. INTRODUCTION

Open platform has been introduced and become popular in the domain of consumer electronics [10]. One of the keys to success in the user-centric market is to have the ability to deliver, as soon as possible, the features that customers want. Open platform allows for the rapid creation of new service without the need to modify existing platform. Using an open platform a developer could add features or functionality that the platform vendor hadn't completed or hadn't conceived of. An open platform may allow the developer to change existing functionality. In other words, under the open platform system, the control over applications has been handed over to an external developer or 3rd party other than the platform developer. These characteristics render the API maintenance harder on the open platform.

In the studies on the evolution of APIs, several former researches focus on how to update applications against API’s change. Specifically, the studies focus on the traceability of applications that support the API changes to prevent the break of the traditional applications developed upon modification of the framework. When distributing a new API on the framework, comparison is conducted with the existing API to detect the API changes. This detection mechanism, however, becomes effective only after changes are made the framework.

From the viewpoint of platform maintenance, it is important to secure the traceability of applications upon changes to the API. No less important is how to efficiently maintain the API itself upon occurrence of additions and changes arising out of the need for new functions.[6]. It is necessary to analyze, prior to the changes made to the platform, the possible impacts of the conceived changes, and to determine whether or not the changes can bring about the benefits worthy of the efforts. Therefore, a brand new approach to API impact analysis is needed for maintainability of open SPL platforms. And the approaches should take into account the following:

First, traceability should be guaranteed of the applications whose development has used the API of the platform. It should be determined whether the previous application has migrated or not, once a new version of the platform is distributed. Successful tracing of API-based applications leads to successful detection of the areas affected by the changes. The information, further, leads to the determination of whether it is beneficial to modify the existing applications or to develop new ones.

Second, it is also necessary to be able to trace the platform internal work products that are affected by the API changes. The requirement arises out of the need to maintain the platform in such a way as to provide new features required of the platform at the minimum costs. For example, the impact of an API A change is traced to other APIs that depend on API A. Those APIs might be derived from the API A, or might specify parameters using API A as the type identifier. The change is also traced to all the classes that implement API A and its derived APIs. Those classes may require some modifications as well. Modified classes should be subject to test to assure that the changed functions work without any glitches. Based on the analytical results, tradeoffs can be performed between two decisions: whether to create a new API in consideration of the affected area, or whether to modify the relevant outputs, while maintaining the existing API.

Proposed in this study is a method of analyzing multi-view API impacts on open SPL platform in embedded software. It analyzes impacts of API changes with traceability between APIs and their correspondent work products such as API implementations, TestCase and applications. It is applied to...
open platform products tailored for the B2B market in printing domain. This type of platforms provides a full spectrum of customized components supported by Java and other Web technologies. Three releases were applied to the platform. From the source code of each of them, we were able to trace the applications that used the platform, the implementation classes that realized the APIs constituting the internal structure of the platform, and the cases that tested them.

In the remainder of this paper is organized, as follows: Section 2 describe the literature on open platform and change impact analysis; In Section 3 sets forth the overall review of our proposed API analysis and the details about what suitable for open platform of the printing domain, followed by case studies applying our model to three releases are introduced in Section 4; and, Section 5 concludes our study with suggestions for future works.

II. RELATED WORKS

A. Open platform trend

The consumer electronics industry is shifting its attention to a development system based on open platform. Let’s take the mobile sector for example. The competition in the mobile platform market is getting fiercer, as the spectrum of the market participants gets wider and wider. For instance, tapping into the market are traditional OS vendors such as Microsoft and Symbian, search engines such as Google and Yahoo!, and even manufacturers like Nokia and Samsung.

RTOS-based feature phones are losing grounds in the market, while open platform-based smart phones are gaining a speedy momentum. As a result, the mobile industry is faced with a higher demand for more diverse services and functions. To catch up with the demand, the API is evolving. Previously, it was sufficient for an API to be able to provide limited functions tailored for local resources. Now, however, it is being transformed to help create a variety of services through disclosure of the API to the public (i.e., through open platform).

An open platform should enable a third-party developer to implement functions currently not available on it, and reimplement or replace the existing functions. [7] Securing the openness necessitates constructing a reconfigurable architecture that allows interchangeability among components, and defining an API to better use it. To define the API for that purpose, it is, in turn, necessary to part away from the traditional designing.

In other words, focus is not on functions of a single application; rather, it is on how to design the architecture to enable prevention of duplication in consideration of common functions among multiple applications, and accommodation of the functions that might be necessary in the future. [8]

III. MULTI-VIEW IMPACT ANALYSIS MODEL

This section describes the views on our proposed API impact analysis. API. There are two major views as to the analysis of API impacts. One is about analysis of applications using the API of a platform, which support application migration. The other is about tracing platform internal work products, which support evolution of the platform.

![Figure 1. Multi-view API Impact Analysis Model](image1)

Figure 1 illustrates the relations of work products to the API on a platform. The line (1) indicates the relation of an application to the platform. The application uses the API provided by the platform. (2) to (4) depict the internal structure concerned with provision of platform functions to the API. Specifically, the line (2) analyzes the inheritanceship of APIs, while the line (3) analyzes classes that maintain implementing relations with the API. The line (4) traces the TestCase that is to verify whether or not the API function has been implemented as intended.

In the next sub-section, the impact analysis views are to be explained in detail, which are shown in Figure 1.

A. Application View

![Figure 2. API-based Application](image2)

Figure 2. API-based Application

Analysis is conducted of the API-based applications. An application is rendered impossible to implemented, once its method is removed or its parameter is changed. As shown in Figure 2, for example, if a change occurs to a parameter, the application that it for use should be changed. More specifically, if the draw (e.g., int p1, int p2, int p3, and int p4), which serves as the method of the rect API, is changed to another draw (e.g., int start, int width, and int height), the method of the rect object shall be altered.
An application can use the rect API in either of the two ways; namely, an instance of the rectImpl1 that is an API-implementing class is generated for reference, or the rectImpl1 is used either as a rect API or as a shape-type reference variable.

The following sums up how an application uses an API:

### Table 1. API Usage Types for Application

<table>
<thead>
<tr>
<th>Usage Type</th>
<th>Sample Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Variable</td>
<td>rect r = new rectImpl1();</td>
</tr>
<tr>
<td>Parent API Variable</td>
<td>r = new rectImpl();</td>
</tr>
<tr>
<td>Parameter Typed</td>
<td>void DoubleDraw(shape s)</td>
</tr>
</tbody>
</table>

- Affected application search algorithm

```
Input: one of APIs A

apps = getApplicationUsage( A );

while ( a ≠ Ø ) do
    a = getParent( A );
    apps = apps U getApplicationUsage(a);
    replace A with a
end
```

The algorithm uses the following functions: (i) The getApplicationUsage(A) returns a list of classes that generate instances of A type, or contain codes used as a parameter of the method; or (ii) the getParent(A) returns a parent API of A.

### B. API View

- Impacted implementation search algorithm

```
Input: one of APIs A

apps = getApplicationUsage( A );

while ( a ≠ Ø ) do
    a = nextAPI( A );
    childs = children( a );
    for each c` ∈ childs do
        if ( hasChildren( c` ) )
            childs= childs U children( c` );
    end
end
```

The algorithm uses the following functions: 1) nextAPI(A) returns the first item in the chin A, 2) children(a) returns a list of classes that inherit from a, and 3) hasChildren(c`) return true if number of children(NOC) of c` > 0, false

### C. Implementation View

Analysis is conducted of the impact of an API change on other APIs. Extensibility is secured by defining an API as an inheritance tree. Once defined, the parent class and the child class form mutual dependency, and end up affecting each other. This view traces all APIs belonging to the inheritance tree of the defined API. Therefore, it becomes possible to discern all the APIs subject to indirect impact of the API to be changed. If n2 is changed in Figure 3, the change affects the implementation of the n2, and the applications using it. Furthermore, ripple effects reach the implementation of the APIs inheriting n2 (i.e. n5, n6, n7 and n8) and the applications using them, which affecting only n6 and implementations and applications directly related to it.

- Impacted API search algorithm

```
Input: A set of APIs A

while( A ≠ Ø ) do
    a = nextAPI( A );
    childs = children( a );
    for each c` ∈ childs do
        if ( hasChildren( c` ) )
            childs= childs U children( c` );
    end
    Remove a from A
end
```

The algorithm use the following functions: 1) nextAPI(A) returns the first item in the chin A, 2) children(a) returns a list of classes that inheri from a, and 3) hasChildren(c`) return true if number of children(NOC) of c` > 0, false

### Figure 3. API Inheritance Tree

Analysis is carried out of the implementations that realize APIs. This view is defined, based on Java, since Java allows an interface or a class to be used as an API. Therefore, the relationship subject to analysis may vary, depending on the API type of the affected implementation. Figure 4 shows two ways of implementing an interface. One is generate a class that implements the body of an interface’s abstract method. Herein, the keyword is “implements.” The other method is to inherit and extend the previously implemented class. In the latter case, “extends” is the keyword to use. Just like a general class, a class defined through an API is inheritable and implementable. Analysis of API implementations requires that of the following three cases:

### Table 2. Types of API Implementation

<table>
<thead>
<tr>
<th>API Type</th>
<th>Sample Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>class rectImpl implements rect</td>
</tr>
<tr>
<td>Interface</td>
<td>class roundRectImpl extends rectImpl</td>
</tr>
<tr>
<td>Class</td>
<td>Class</td>
</tr>
</tbody>
</table>
• Affected implementation search algorithm

Input: A set of APIs A

while( A ≠ ∅ ) do
  ( a ) = nextAPI(A);
  if ( a.type = “Interface” )
    impls = getImplements( a );
    for each i′ ∈ impls do
      impls = impls U getExtends( i′ );
    end
  end
  else if ( a.type = “class” )
    impls = getExtends( a )
  end
  Remove a from A
end

D. TestCase View

Figure 5. Relationship between API and TestCase

This view traces the TestCases that are mapped to APIs. Analysis of affected TestCases begins by identifying the implementation class of the API to which the change is made. Subject next to tracing are the TestCases that test implementation classes. Figure 5 shows the relationship between APIs and TestCases. One API can have several implementation classes that are tested by several TestCases. For example, the API B has impleB and impleB’. The impleB has multiple TestCases, since a TestCase is created as a unit of an API method. Prior to the TestCase view, the analysed implementation view and the TestCase data should be inputted. Then, the implementation class that implements the corresponding API should be retrieved from the implementation view, based on the name of the API. Then, the names of API and methods are mapped on the TestCase data. In the process, IDs are obtained of the relevant TestCases. A tester prepares the TestCase data, which consist of test scenario, test procedure, input, and output.

• Impacted TestCase search algorithm

Input: A set of APIs A

while( A ≠ ∅ ) do
  ( a ) = nextAPI(A);
  impls = mappingImplement( a );
  for each i′ ∈ impleA do
    TCs = mappingTestCase( a );
    Remove a from A
end

The algorithm use the following functions: 1) mappingImplement(a) returns a list of implementation classes that are matched with a name with API name in implementation view. 2) mappingTestCase(a) returns a list of TestCases that are matched with a name with API name in TestCase data.

IV. Case Study

This section introduces a case study wherein our proposed multi-view API analyser has been applied to P platform in the printing domain. P platform is an open architecture platform for B2B clients. It supports customization and extensibility of various functions of the next generation multi-function printers. Any developer and independent software vendor can develop her own application using P-platform APIs. Two technologies support the solution applications: Java technology and Web technology.

To verify the validity of our proposed API analysis model, analytical tools have been developed, based on the model. The tools extract the necessary information from each view by means of static analysis of source codes. Application of the tools to three versions of P platform yielded the results as shown in Table 1. The numbers shown in the table are counted at the baseline of the released version. The term “build LOC” refers to the line of the source codes that were actually incorporated during the construction. The category “class number” indicates the number of the classes defined for implementation of APIs, while the category “API” refers to the number of the interfaces and classes that were released to the public.

APIs were classified into the following three classes: interface, abstract class and class. Especially, items such as data type, enumerations and exception handler are classified under “class,” since they do not need any type of implementation.

<table>
<thead>
<tr>
<th>Table 3. Sizes of Released Codes for P Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release</td>
</tr>
<tr>
<td>Build LOC</td>
</tr>
<tr>
<td>Class #</td>
</tr>
<tr>
<td>API #</td>
</tr>
<tr>
<td>Interface</td>
</tr>
<tr>
<td>Abstract Class</td>
</tr>
<tr>
<td>Class</td>
</tr>
</tbody>
</table>

In this section, the analytical results of each view are explained from either of the following two viewpoints: (Due to limitations and security issues, the results concerning C releases are be explained hereunder)
- Application Analysis
  Table 4 lists some of the applications using Queue APIs. It shows that the method of passing Queue API-type parameters is located in the BoxContainer class of the Box plug-in.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Application</th>
<th>Class</th>
<th>Usage Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plug-in</td>
<td>Box</td>
<td>BoxContainer</td>
<td>Parameter</td>
</tr>
<tr>
<td>Plug-in</td>
<td>Scan</td>
<td>JobHandler</td>
<td>Variable</td>
</tr>
<tr>
<td>Plug-in</td>
<td>Scan</td>
<td>ScanManager</td>
<td>Parameter</td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As explained in the preceding section, applications are used by generating classes not only as relevant APIs, but also as parent API. Therefore, applications affected by the Queue API include both the applications using the Queue and those using the parent API. Table 5 only shows how the Queue and parent APIs are used. Parent 1 is used as variable three times, while Parent 2 is used as parameter four times. Thus, the number of applications affected by changes to the Queue API represents the sum of the number of the applications using the Queue and that of the applications using parent APIs.

<table>
<thead>
<tr>
<th>Parameter #</th>
<th>Variable #</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Parent 1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Parent 2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Parent 3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Platform Analysis
  Two cases were cited to explain the impacts of the API changes on the platform. One is the API at the lowest leaf node of the API structure, and the other is the API that has children APIs. Let’s take Figure 2 for example. The rect circle and the concrete API represent the former, while the abstract API, or the shape, represents the latter. The existing platform is free from any impact, even upon addition of concrete APIs like triangle. Thus, to use applications in need, one has to simply add triangle API inheriting shape APIs, and implement them. When a change is made to the rectangle API, modification should be made to the class that implements it, followed by a test to verify its operability.

In addition, when a change is made to the shape API, the area subject to impact gets enlarged, because it is necessary to implement newly generated or modified methods on the shapes of the circles and the rectangles inheriting the shape. Table 6 charts out the analytical results of two APIs on either end of inheritance. The concrete API has a single implementation class, and 29 TestCases. On the other hand, in the case of the abstract API, it is necessary to trace and analyze not only the affected APIs, but the children APIs.

<table>
<thead>
<tr>
<th>Children API #</th>
<th>Impl. Class #</th>
<th>TestCase #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concrete API</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Abstract API</td>
<td>41</td>
<td>28</td>
</tr>
<tr>
<td>Child 1</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>Child 2</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Child 3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Comparison of Analytical Results of Two Versions

Figure 6 illustrates the percentage share of the implementation classes of the C release that has been affected by the API changes, juxtaposed with the composition of the B release. When compared, 74 APIs were changed, along with 161 classes that implemented them. In detail, the left pie graph shows the share of the implemented APIs out of the total 571 APIs that were defined by the interfaces and the abstract classes requiring implementation in the C release. Of the 571, 446 APIs, or 78% of them, had implementation classes. Specifically, only 74 of the 446 APIs had changed methods, while the 372 APIs of them remained unchanged.

A single API has multiple vehicles of implementation. Therefore, the aggregate number of APIs is outnumbered by that of their classes. In the meanwhile, the right pie represents the implementation classes of the C release. To implement 446 APIs, 3,729 classes were mobilized. 4% of the total 3,729 classes were impacted. 24% of the total 3,729 classes were impacted.

Figure 7. Impacted TestCase by comparing with B and C release

Table 6. Usage of Sample APIs on Platform

<table>
<thead>
<tr>
<th>TestCase of C Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tested</td>
</tr>
<tr>
<td>Not Tested</td>
</tr>
<tr>
<td>Tested (3%)</td>
</tr>
<tr>
<td>Impacted</td>
</tr>
<tr>
<td>195 (24%)</td>
</tr>
<tr>
<td>Not Impacted</td>
</tr>
<tr>
<td>614 (76%)</td>
</tr>
</tbody>
</table>

Figure 7. Impacted TestCase by comparing with B and C release
Figure 7 shows the portion of the TestCases of the C release affected by the API changes through comparison of that of the B release. A total of 809 TestCases were carried out on the 42 APIs of the total, and 195 cases were affected. Based on the information shown through the graphs, only 3% of the cases (i.e. 42 cases) seem to have been subject to test, because the information excluded the data on the previously conducted TestCases. The mapping information of the APIs and their TestCases was carried out, by using combinations of the APIs’ and methods’ names as keys. Modification is ongoing, due to the difference between the names of the existing TestCases and the newly combined names.

V. CONCLUSIONS

In this study, a model for analysis of multi-view API impacts has been proposed, which is designed to better maintain APIs of the B2B open platform used in the printing domain. Our proposed model analyzes the areas affected in the course of implementing and using the APIs of the open platform. The model has been applied to the 3rd release of an open platform under development. As a result, it was possible to trace classes and applications affected by the API changes.

Prior to the present study, developers did not know exactly how much APIs are affected, and how much implementation classes of a platform should be modified to cope with the impact. Upon receipt of new API specifications, the developers determined on their own wether or not any changes had been made to the existing APIs and, conducted modification if a need arouse. In short, developers repeated a boring cycle of modifying and building, until no error was found. In addition, testers, who lacked the background information on the platforms, found it extremely difficult to detect API changes, based on specifications, and, consequently, lost a considerable portion of the inter-impacts among APIs.

Our model is believed to contribute to resolving the inconveniences by providing better information to the stakeholders concerning development of a platform.

First, objective data were provided to stakeholders by analyzing all the work products related to APIs.

Second, since the API impact analysis is based on source codes, it is possible to carry out the updated analysis at any time one wishes. Finally, analytical tools were developed as well to dramatically reduce the time necessary for analysis.

In the past, it took about a week for a tester to determine, based on the recorded specifications, whether or not changes had been made to APIs, and to trace relevant TestCases. Development of the tools led to production of improved quality just for a one-day period.

More efforts will ensue in the future to improve the proposed model and, thereby, to expand the scope of the impact analysis to the level of method or data and to improve the analytical accuracy. The proposed model assumes classes, which constitute APIs, as unit. Thus, it might reveal inconsistency concerning the areas that do not affect actual changes. When analyzing an application, no method is affected if the changed methods of an API are not used.

But the proposed model fails to detect the difference. In this context, it is necessary to narrow down the analytical unit form the current API class to the API method.

We also plan to develop tools that support independent developers who develop applications as part of the SDK of a platform. Application developers should clearly note the platform version through which relevant applications can be executed. Therefore, it is necessary to develop a tool that checks the platform version compatible with an application, and, furthermore, to trace changed APIs and modify them.

REFERENCES