Short Paper

Inter-Component Exception Handling for Robust Android Apps

Kwanghoon Choi
Yonsei University, Wonju, Korea
kwanghoon.choi@yonsei.ac.kr

Byeong-Mo Chang
Sookmyung Wonmen's University, Seoul, Korea
chang@sookmyung.ac.kr

Abstract

This paper proposes an inter-component exception handling mechanism for robust Android apps. The current design of Android platform allows exception handling to be only applied to the inside of each component of the apps, but some exceptions need to be propagated and handled outside of the component. We have developed a domain-specific library that extends existing Android components with inter-component exception handling, for programmers easy to utilize it to make Android programs more robust. In addition, we have performed case studies to demonstrate the usefulness of the proposed mechanism. As far as we know, this is the first proposal on inter-component exception handling for Android apps.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords Android, Java, Exception, Component

1. Introduction

Exception handling in Java is an important feature to improve the robustness of Java programs. For example, Figure 1 shows a simplified Java program that may throw one of two exceptions, *NoSuchOperator* and *ArithmeticException ("divide by zero")*, when users try to do a calculation with an unsupported operator or with zero as a divisor. Once the *calc* method throws such an exception, it is propagated along the call stack to a caller, the *main* method where it is handled by the catch block.

Every Android program is a Java program with APIs in Android platform, presumably using exception handling. Android is Google's open-source platform for mobile devices, and it provides the APIs (Application Programming Interfaces) necessary to develop applications for the platform in Java [1]. An Android program consists of *components* such as activities, services, broadcast receivers and content providers.

An activity can start other activities by sending intents to Android platform, which invokes methods of callee activities. However, uncaught exceptions from callee activities can not be propagated to the caller activity, because all uncaught exceptions from callee are propagated along the method call stack until they arrive at Android platform. When uncaught exceptions arrive at Android platform, the program terminates abnormally.

```
class Calculator {
   void main()
   { int a, b, r;   char op;
     // read a, op, and b      from a user
     try { r = calc(a, op, b);
          // display the result  }
     catch(NoSuchOperator e)
         { // Not support the operator } }
   int calc(int a, char op, int b)
   { if ( op == '+' )      return a + b;
     else if (op == '-'))  return a − b;
     else if (op == '*')   return a ∗ b;
     else if (op == '/')   return a / b;
     else throw new NoSuchOperator();    }
}
```

Figure 1. A Java Program Using Exceptions

Clearly, this design of Android platform gives components no chance to recover exceptions from other component, which makes Android programs less robust by not being able to fully utilize the feature of Java exception propagation. Moreover, many components in Android programs have a relationship on *"who activates whom"*, which is very similar to a caller-callee relationship in method invocation. Particularly, Android platform internally has an activity stack, which is the same as a call stack in method invocation, to maintain the who-activates-whom relationship. Android programs could be more robust if a caller (activating) activity could catch any exception thrown by a callee (activated) activity.

In this paper, we propose a mechanism for inter-component exception handling in Android programs, which can make Android programs more robust. We will show that this mechanism can be implemented by providing new component APIs (e.g., ExceptionActivity class), which extends the existing Android components (e.g., Activity class) with inter-component exception handling. Programmers can utilize inter-component exception handling by writing component programs with the new extended APIs. In addition, we will perform case studies to evaluate its usefulness.

2. Overview of Android/Java

An Android program is a Java program with APIs in Android platform. Using the APIs, one can build mobile device user interfaces to make a phone call, play a game, and so on. An Android program consists of components whose types are Activity, Service, Broadcast Receiver, or Content Provider. Activity is a foreground process equipped with windows such as buttons and text inputs. Service is responsible for background jobs, and so it has no user interface. Broadcast Receiver reacts to system-wide events such as notifying low power battery or SMS arrival. Content Provider supports various kinds of storage including database management systems.

```
class Main extends Activity {
  void onCreate() {
    addTextInput(1); // for 1st operand
    addTextInput(2); // for operator
    addTextInput(3); // for 2nd operand
    addButton(1); // for (=) button
    /* initialize the main screen */ }
  void onClick(int button) {
    int a, b;    char op;
    // read a,op,and b from the text input windows
    Intent i = new Intent();
    i.setTarget("Calc");
    // put a, op, and b into i
    i.setArg( ... a, op, b ... );
    try { startActivityForResult(i); } // Useless
    catch(NoSuchOperator exn) // Exception Handling
      { /* handle the exception */ } }
  void onActivityResult(int resultCode, Intent i) {
    if (resultCode == RESULT_OK) {
      // read r from i
      int r = ... i.getArg() ... ;
      // display the result
    } }
}
```

```
class Calc extends Activity {
  void onCreate() {
    addButton(1); // for return
    // display the return button
  }
  void onClick(int button) {
    int a, b, r;
    char op;
    Intent i = getIntent();
    // get a, op, b from i
    ... i.getArg() ... ;
    if (op == '+') r = a + b;
    else if (op == '-') r = a - b;
    else if (op == '*') r = a * b;
    else if (op == '/') r = a / b;
    else throw new NoSuchOperator();

    // put r into i
    i.setArg( ... r ... );
    // set i as a return value
    setResult(RESULT_OK, i);
    finish(); // dismiss this activity
  }
}
```

Figure 2. An Android Calculator Program with Useless Exception Handling (in the onClick method of Main)

Components interact with each other by sending events called *Intent* in Android platform to form an application. The intent holds information about a target component to which it will be delivered, and it may hold data together. For example, a user interface screen provided by an activity changes to another by sending an intent to Android platform, which will destroy the UI screen and will launch a new screen displayed by a target activity specified in the intent.

To make Android programs robust, a caller activity should be able to handle exceptions thrown by callee activities. However, the design of Android program does not support such an inter-component exception handling. To illustrate this problem, Figure 2 presents an Android-based calculator program where the caller activity Main cannot catch exceptions from the callee activity Calc.

Activity is a class that represents a screen in Android platform, and Main and Calc extending Activity are also classes representing a screen. Initially, Android platform creates a Main object, it invokes the $onCreate$ method to add three input text windows and one button with integer identifiers as arguments. Now, a user can enter two integers and one operator to press the button, and then the $onClick$ method is invoked to perform some action for the button. Intent is a class that represents an event to launch a new screen. It specifies the name of an activity class that represents the new screen and some data passed to the callee. The $onClick$ method sets "Calc" and values read from the text input windows in the new intent object, and then it requests launching by invoking $startActivityForResult$. Android accepts the request and changes the current UI screen by stacking Calc on Main, calling Calc's $onCreate$ method to setup a button for return of the calculation result back to Main. On pressing the button, Android invokes Calc's $onClick$ method, which gets the operator/operands from the intent (obtained by *getIntent()*), calculates it to set the result to the intent, and dismisses Calc. Then Main appears again, and Android invokes the $onActivityResult$ method to pass RESULT_OK as resultCode and the intent with the result as i, and to display it.

As well as the normal execution flow as explained above, the exceptional execution flows might happen. When a user enters other than the four arithmetic operators, Calc's $onClick$ method will throw an exception, *NoSuchOperator*. When a user enters, say, "1 / 0", the division in the method will throw ArithmeticException (*"divide-by-zero"*). In such exceptional cases, the exceptions thrown by the Calc's *onClick* method will be propagated to Android platform who invoked the method, not to (the *onClick* method of) Main who activated Calc. Android platform then catches the exceptions, but it has nothing to do sensibly but stop the execution abnormally. Hence, the exceptions will never reach the try/catch block surrounding the invocation *startActivityForResult(i)* in the Main's *onClick* method. This explains why the exception handling is useless.

This is a limitation of the design of Android platform solely depending on the Java exception semantics. One might introduce a try/catch block surrounding the if statements in the Calc's *onClick* method to handle both of the exceptions, but could not proceed further to get alternative inputs without going back to Main.

To make more robust Android programs, we believe that Android platform should support an enhanced mechanism, which we call *inter-component exception handling*, to give caller activities more chance for handling exceptions from callee activities.

3. Inter-Component Exception Handling

Our goal is to redesign Activity APIs by incorporating our idea of inter-component exception handling so as to develop more robust Android programs. We design a domain-specific library for inter-component exception handling, as in Figure 3. Basically, the library introduces new component classes (e.g., ExceptionActivity) inherited from the existing ones (e.g., Activity) to override the interface methods (e.g., onCreate) that Android platform invokes, by adding the inter-component exception handling capability. To use this library, Android programs must have the counterparts methods (e.g., OnCreate) rather than the original Android interface methods.

• Overriding Android Interface Methods: The inter-component exception handling (ICEH) library puts a fence at each border between an Android program and the platform to catch all exceptions from the program and to propagate them following the component activation stack. For example, the $onCreate$ method of ExceptionActivity in Figure 3 invokes its counterpart method $OnCreate$, which is surrounded by a try/catch block to catch all uncaught exceptions from the method and to propagate them to some caller

```
class ExceptionActivity extends Activity {
  void onCreate() {
    theIntent = getIntent();
    try { OnCreate(); }
    catch(Throwable exn) { Throw(exn); }
  }
  void OnCreate() { }
  void onActivityResult(int resultCode,Intent i){
    if (resultCode == RESULT_OK) {
      try { OnActivityResult(resultCode, i); }
      catch(Throwable exn) { Throw (exn); }
    }
    else { // resultCode == RESULT_EXN
      Throwable exn = i.exn;
      try { if (catcher == null
                 || catcher.handle(exn) == false)
              throw exn; }
      catch(Throwable unhandled_exn) {
        if (theIntent.calledByExnActivity != null)
           Throw(unhandled_exn); //return to the caller
        else // no more caller exists.
           // exit with the unhandled exception
      }
    }
  }
  void OnActivityResult(int resultCode,Intent i){ }
```

```
  void onClick(View v) {
    try { OnClick(v); }
    catch(Throwable exn) { Throw(exn); }
  }
  void OnClick(View v) { }

  void TryActivityForResult(Intent i, Catch c) {
    i.calledByExnActivity = TRUE; // not null
    catcher     = c;
    super.startActivityForResult(i);
  }

  void Throw(Throwable exn) {
    Intent i = new Intent();
    i.exn = exn; // put exn into the intent i
    setResult(RESULT_EXN,i); //set a return value
    finish(); // dismiss this activity
  }

  Intent theIntent;
  Catcher catcher;
}

interface Catch {
  boolean handle(Throwable exn);
}
```

Figure 3. The Inter-Component Exception Handling Library

activity. The other interface methods to Android platform (e.g., *onClick* and *onActivityResult*) have the similar fence structure as this.

Figure 4 shows an Android calculator program using ExceptionActivity. For example, in the Main class, the original interface method *onCreate* is replaced with its counterpart method *OnCreate* to make use of ICEH library.

- Inter-component *Throw* Construct: To propagate an exception across components, the ICEH library offers "*Throw(exn)*", which packages the exception into an intent to set as a return value by invoking *setResult* with RESULT_EXN. The *onActivityResult* method of the caller will receive the intent together with the exceptional result code. Note that the standard Java construct "throw exn" can also be used to throw and propagate an exception within a component.
- Inter-component *Try/Catch* Construct: The ICEH library offers a caller component chances to handle exceptions thrown by callee activities as follows.

```
TryActivityForResult(i, new Catch() {
  boolean handle(Throwable exn) {
    if (exn instanceof E)
         { // handles the exception
           return true;  }
    else return false;
  }
});
```

where Catch is a Java interface for building a *catcher* object for inter-component exception handling. The *handle* method is designed to return true if the exception is processed. Otherwise it returns false and the exception will be re-thrown.

Instead of the code above, one could imagine the useless exception handling code as in Figure 2. Because the *startActivityForResult* method behaves asynchronously, it immediately returns after requesting Android platform to launch an activity specified in the intent, and so no exception from the activity can be caught here.

Let us now consider inter-component exception propagation in detail. Suppose the *OnClick* method of Calc (extending ExceptionActivity) in Figure 4 throws a *NoSuchOperator* exception. The corresponding interface method *onClick* of Calc nets the exception by its try/catch block. Using "*Throw(exn)*", the exception is considered as a special activity result, and an intent with the exception is dispatched to the *onActivityResult* method of the caller activity (Main). To distinguish exceptional activity results from normal ones, the inter-component throw construct tags the results with a result code, RESULT_EXN. Normal results will be tagged with RESULT_OK. Note that Intent class is assumed to have a field *exn*, which holds an exception.

```
class Main extends ExceptionActivity {
   void OnCreate() {/* the same code as in Fig.2 */}
   void OnClick(int button) {
      // the same code before the try/catch block
      TryActivityForResult(i, new Catch() {
         boolean handle(Throwable exn) {
            if (exn instanceof NoSuchOperator)
               { // handle the unsupported operator
                 return true; }
            return false;
         } });
   }
   void OnActivityResult(int resultCode,Intent i)
    {  // the same code     }
}
class Calc extends ExceptionActivity {
   void OnCreate() { /* the same code */ }
   void OnClick(int button) {/* the same code */}
}
```

Figure 4. An Android Program using Inter-Component Exception Handling

On a caller's receiving a result, the caller examines the result code in the *onActivityResult* method to decide what to do next. If it is a normal result, the method calls a (probably, user-defined) *OnActivityResult* method to return it. If it is an exception, we invoke the exception handling method *handle* of a *catcher*, which is once saved by *TryActivityForResult*. When the catcher succeeds in the exception handling, the current activity gets back to the normal state. Otherwise, we re-throw the exception either by propagating it to the next caller activity or by terminating the whole program abnormally when there is no more (ICEH-aware) caller. Note that Intent class is also assumed to have a field *calledByExnActivity* to indicate if there is more (ExceptionActivity extended) caller. The field is set by *TryActivityForResult*, and normal Android activities (extending Activity) leave it as NULL.

Therefore, in Figure 4, the ICEH library propagates each *NoSuchOperator* exception thrown by the *OnClick* method in Calc to the *handle* method of a catcher created at the *OnClick* method in Main. After the exception is processed, the program gets back to a normal state. This shows the robustness of the program in Figure 4.

This approach has a few advantages besides providing the robustness. It demands no change on Android platform, and so all commercial Android-based devices can get benefit from it immediately. It does not increase much the memory footprint of Android programs since they can share the same library. Developing new Android programs with the proposed library demands no more efforts than that without it. A disadvantage is that existing Android programs must be rewritten to use the library. However, the rewriting to adopt the library is straightforward and involves marginal modifications of the programs, as it is shown by the difference between the two Android programs with and without the library in Figure 4 and Figure 2, respectively.

4. Case Study

In a few case studies, we have found our inter-component exception handling useful in Android programs:

- Restructuring two existing real Android programs, BluetoothChat and NotePad, to find that the proposed library can help to handle some exceptions to prevent them from abnormal termination.
- Developing Android programs with the proposed library from the beginning to help programmers to diagnose the reasons of exceptions better by providing component-level traces.

First, we have restructured BluetoothChat and NotePad, which have been developed by Google as Android example programs. BluetoothChat is a text-based communication program using bluetooth. It consists of two activities and one broadcast receiver where the main activity launches the other discovery activity to find out any near bluetooth equipped devices to chat with. Once such a device is found, the broadcast receiver will receive an intent holding information about the device. NotePad is a memo program to create, to edit, and to delete memos in mobile database. It consists of three activities, one for listing memos, another for editing a memo, and third for editing a memo title.

Both of Android programs have been developed very robust by Google, but it was not difficult to find out some vulnerability to exploit to cause some exceptions. For example, we can construct an ill-formed intent for discovery of near-by bluetooth devices to send to the broadcast receiver of BluetoothChat to raise the null reference exception. Also, we can make NotePad to raise the out-of-memory exception by copying a memo into itself repeatedly to explode the memo editor activity. In both of the exceptional situations, the original Android programs terminate abnormally, but the restructured ones catch them and they are able to return to the normal program state. This confirms our claim that the proposed inter-component handling makes Android programs more robust.

Next, we have examined Cafe, being developed by a student as a term project in the mobile programming course, to see other benefits from developments with the proposed library. Cafe is an Android program that helps to find cafes. A main activity in Cafe demands users to choose to launch one of a map activity (showing near-by cafes in the map) and a cafe list activity (providing a list of registered names). Once users select a cafe from one of the two activities, a cafe (information) activity will display people's evaluations, and users can write something via an opinion activity.

Novice Android programmers can understand the behavior of activity stack better by the use of the proposed library. In the Java semantics, every exception carries information on the call stack when the exception is thrown. The proposed library augments the information with a trace of activities from the main activity, which narrows the list of causes of the exceptional situation down with a component-level contextual information.

Another finding is the use of inter-component exception handling as a programming idiom of *non-local jumps* between activities (such as *longjmp* in C). For example, there is an activity transition scenario in Cafe from the cafe activity ($cafe{\cdot}map{\cdot}main$) back to the main activity ($main$) crossing the map activity. The cafe activity has only to throw a user-defined exception that the main activity catches. This idiom is applicable to many activity transition scenarios that do not follow the stack discipline exactly.

The source code of our proposed library and case studies is available in http://github.com/kwanghoon/exceptionRefactoring.

5. Related Work and Conclusion

This paper has proposed a domain-specific Android library for inter-component exception handling mechanism for robust Android programs. We have performed a few case studies to assess its usefulness. The proposed mechanism can offer a hint to enhance the limitation of Android platform only resorting to the Java exception handling.

There have been several research works [2–5] including one by Huang and Wu [2], which recognizes the same problem that the traditional code-level exception handling is usually applied to the inside of a component, while some exceptions can only or properly be handled outside of the component. They designed a middleware approach atop EJB and/or CORBA container for (implementation language-neutral) exception handling at architecture level. Contrary to this, our inter-component exception handling is for the mobile platform, Android, in the form of an easy-to-use and user-extensible Java library using class inheritance to intercept exceptions systematically.

In future, we will study a formal aspect of the proposed Android exceptions by extending an Android semantics [6]. Also, it will be interesting to incorporate the proposed library into the Eclipse refactoring tool for Android developers easy to use the library.

References

[1] http://developers.android.com

[2] G. Huang and Y. Wu, Towards Architecture-Level Middleware-Enabled Exception Handling of Component-based Systems, the 14th Int'l ACM SIGSOFT Symp. on Component-Based Software Engineering, Colorado, USA, June 21-23, 2011.

[3] A. Romanovsky, Exception Handling in Component-based System Development, 25th Annual Intl' Computer Software and Applications Conference, pp.580-586, 2001.

[4] F. C. Filho, P. A. C. Guerra, V. A. Pagano and C. M. F. Rubira, "A Systematic Approach for Structuring Exception Handling in Robust Component-based Software", J. Braz. Comp. Soc., Vol.10, No.3, pp.5-19, 2005.

[5] C. Dellarocas, Toward Exception Handling Infrastructures in Component-based Software, International Workshop on Component-based Software Engineering, Vol.31, 1998.

[6] Kwanghoon Choi and Byeong-Mo Chang, "A Type and Effect System for Activation Flow of Components in Android Programs", Information Processing Letters, 2014 (Accepted for Publication).