Debugger for Java

Advanced Programming
March, 2015
Group 32

Motivation

- Because some bugs can’t be detected at compile-time, it is needed to detect bugs during the execution of the application.

- However the programming languages are not powerful enough to inspect the state of the program.

Goal

- In order to inspect the program at runtime the team built a debugger to instrument a Java program so that when an exception is thrown, the program is stopped and the programmer is presented with a command-line interface with several debugging mechanisms

Solution

- To reach the goal, the approach that we used consists in intercept all the method calls and replace it's own body in order to call a well known method that invokes the original method using reflection and catch any exception if that exists. When an exception is throw the debugger is executed.

DebuggerCLI Class

```
package ist.meic.pa;

import java.lang.reflect.*;
import javassist.*;

import java.io.*;
import java.lang.reflect.*;
import java.util.Iterator;
import java.util.Scanner;
import java.util.Stack;

/*
 * DebuggerCLI class
 */
public class DebuggerCLI {

    public static void main(String[] args) {
        try{
            Translator translator = new DebuggerTranslator();
            ClassPool pool = ClassPool.getDefault();
            Loader classLoader = new Loader();
            Class<?> History =classLoader.loadClass("ist.meic.pa.History");
            classLoader.addTranslator(pool,translator);
            String[] restArgs= new String[args.length - 1];
            Method m = History.getMethod("pushStack", new Class[]{ Object.class,String.class,Object[].class,String.class});
            System.arraycopy(args,1, restArgs,0,restArgs.length);
            m.invoke(null, new Object[] {null, args[0], restArgs,"main"});
            try {
                classLoader.run(args[0],restArgs);
            } catch (Throwable e) {
                e.printStackTrace();
                }

    }catch (Exception e){
        //TODO
        }
    }
}
```

All classes are loaded via classLoader, and the class given in the arguments is called with the rest of the arguments.

DebuggerTranslator Class

```
package ist.meic.pa;

import javassist.CannotCompileException;

public class DebuggerTranslator implements Translator {

    @Override
    public void start(ClassPool pool) throws NotFoundException,CannotCompileException {
        // TODO Auto-generated method stub
    }

    @Override
    public void onLoad(ClassPool pool, String className) throws NotFoundException, CannotCompileException {
        CtClass ctClass = pool.get(className);
        if(!ctClass.getPackageName().contains("ist.meic.pa") && !ctClass.getPackageName().contains("javassist")){
            makeUndoable(ctClass);
        }
    }



    void makeUndoable(CtClass ctClass) throws NotFoundException,CannotCompileException {

        for (CtMethod ctMethod : ctClass.getDeclaredMethods()) {
            final String template = "{"
                    + "$_ =($r) ist.meic.pa.MethodTranslator.genericMethod($0, $class, \"%s\", $args, $sig); }";

            ctMethod.instrument(new ExprEditor() {
                public void edit(MethodCall m)  throws CannotCompileException {
                    String modif = String.format(template, m.getMethodName());
                    m.replace(modif);
                }

            });

        }
```

The code is injected in order to call the function genericMethod each time a method is called

MethodTranslator Class

```java
package ist.meic.pa;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

/*
 * Class MethodTranslator catches the exception thrown by the invocation of the method and calls
 * the DebuggerShell
 */
public class MethodTranslator {
    public static Object c = null;
    public static Object genericMethod(Object obj , Class<?> objClass, String methodName,Object[] args, Class<?>[] paramTypes) throws Throwable {
        History.calledClasses.push(objClass);
        History.calledObjects.push(new ObjectFieldValue(obj,objClass.getName(),args,methodName));
        History.calledArgs.push(args);
        try{
            Method method = objClass.getMethod(methodName,paramTypes);
            method.setAccessible(true);
            c = method.invoke(obj, args);
        }catch(InvocationTargetException e){
            System.out.println(e.getCause());
            DebuggerShell.main(e);
        }
        History.calledObjects.pop();
        History.calledArgs.pop();
        History.calledClasses.pop();
        return c;
    }

}
```

Pushes all the arguments to the stack and invoke a method, if that method returns an exception the debuggerShell is called.

ObjectFieldValue Class

```java
package ist.meic.pa;

/*
 * Class ObjectFieldValue
 */
public class ObjectFieldValue {
    Object object;
    String className;
    Object[] args;
    String method;

    /*
     * Constructor
     */
    public ObjectFieldValue(Object object2, String className2, Object[] args2, String method2){
        this.object=object2;
        this.className=className2;
        this.args=args;
        this.method=method2;
    }

    public Object getObject(){
        return this.object;
    }

    public String getClassName(){
        return this.className;
    }

    public Object[] getArgs(){
        return this.args;
    }
    public String getMethod(){
        return this.method;
    }
}
```

History Class

```java
package ist.meic.pa;

import java.util.Iterator;
/*
 * Class History holds the stack with information about Objects, methods called, arguments called
 * and classes called
 */
public class History {
    static Stack<ObjectFieldValue> calledObjects = new Stack<ObjectFieldValue>();
    static Stack<String> calledMethods =new Stack<String>();
    static Stack<Object[]> calledArgs = new Stack<Object[]>();
    static Stack<Class<?>> calledClasses = new Stack<Class<?>>();

    public static void pushCalledObjectsStack (ObjectFieldValue object){
        calledObjects.push(object);
    }
    public static void pushStack(Object object2, String className2, Object[] args2, String method2){
        ObjectFieldValue obj = new ObjectFieldValue (object2,className2,args2,method2);
        pushCalledObjectsStack(obj);
        calledArgs.push(args2);
    }

    public static Stack<ObjectFieldValue> returnPreviousCalledObject(Stack<ObjectFieldValue> stack){

        Stack<ObjectFieldValue> stackWithoutLastElement = (Stack<ObjectFieldValue>) calledObjects.clone();
        stackWithoutLastElement.pop();
        return stackWithoutLastElement;
    }
}
```

In calledObjects Stack the objects are saved the objects, in calledMethods the method names are saved, in calledArgs the used Args are saved and in calledClasses the Classes called are saved

returnPreviousCalledObject returns the stack without the first element

DebuggerShell Class

```
public class DebuggerShell {
        public static boolean canContinue = true;
        public static void main(Exception e) throws Throwable{
                while(canContinue){

                        String[] input;
                        Scanner in = new Scanner(System.in);
                        System.out.print("DebuggerCLI:> ");
                        input = in.nextLine().split(" ");

                        switch (input[0]){

                        case "Info":
                                CommandInfo.main();
                                break;

                        case "Get":
                                CommandGet.main(input);
                                break;

                        case "Abort":
                                CommandAbort.main();
                                break;

                        case "Set":
                                CommandSet.main(input);
                                break;

                        case "Retry":
                                CommandRetry.main();
                                break;

                        case "Return":
                                CommandReturn.main(input);
                                break;

                        case "Throw":
                                History.calledObjects.pop(
                                History.calledArgs.pop();
                                throw e.getCause();
                        }
                }
                canContinue=true;
}
```

According to the input it calls the right method, if the flag canContinue is false it exits the shell

CommandInfo Class

```
public class CommandInfo {
	public static void main(){

		Object lastObject = History.calledObjects.peek().getObject();
		Stack<ObjectFieldValue> copyOfCalledObjects = History.calledObjects;

		System.out.print("Called Object:	");
		System.out.println(lastObject);

		Field[] fields = History.calledClasses.peek().getDeclaredFields();
		String output = "";

		// Append the name of the fields to the string output
		for (int i=0; i< fields.length;i++){
			if(i==(fields.length -1)){
				output=output.concat(fields[i].getName());
			}else{
				output=output.concat(fields[i].getName() + " ");
			}
		}

		System.out.print("	Fields:	");
		System.out.println(output);

		System.out.println("Call stack:");

		//	Clone the stacks in order to have different stacks
		Stack<ObjectFieldValue> stack = (Stack<ObjectFieldValue>) copyOfCalledObjects.clone();
		Stack<Object[]> stackArgs = (Stack<Object[]>) History.calledArgs.clone();

		//Prints all the methods and the arguments that the method has been called
		for(int i=0;i< stack.capacity(); i++){
			ObjectFieldValue currentObject = stack.pop();
			System.out.print(currentObject.className + ".");
			System.out.print(currentObject.getMethod());
			printArgs(stackArgs.pop());
		}
	}


	/*
	 * Prints the arguments
	 */
	public static void printArgs(Object[] calledArgs){
		String output="";
		System.out.print("(");

		// Append the name of the arguments to the string output
		for (int i=0; i< calledArgs.length;i++){
			if(i==(calledArgs.length -1)){
				output=output.concat(calledArgs[i].toString());
			}else{
				output=output.concat(calledArgs[i].toString() + ",");
			}
		}
		System.out.println(output + ")");
	}
}
```

CommandSet Class

```
public class CommandSet {

    public static void main(String[] input) throws InvocationTargetException{

        Field[] allFields = History.calledObjects.peek().getObject().getClass().getDeclaredFields();


        //Searches for the field given in input
        for(Field field : allFields){
            if(field.getName().equals(input[1])){
                try {
                    Object obj= null;
                    field.setAccessible(true);

                    //Gets the type of the field
                    Class type = field.get(History.calledObjects.peek().getObject()).getClass();
                    Object fieldValue =field.get(History.calledObjects.peek().getObject());

                    //invoke valueOf in the given type with input
                    for(Method m : type.getDeclaredMethods() ){
                        if(m.getName().equals("valueOf") && m.getGenericParameterTypes()[0].equals(input[2].getClass()) && (m.getGenericParameterTypes().length==1)
                            obj =m.invoke(type, input[2]);
                        }
                    }
                    //Sets the field of the object
                    field.set(History.calledObjects.peek().getObject(), obj);


                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

It searches for the method **ValueOf** with the argument of the type of the given field and invokes the mehod **ValueOf,** returning that way the desired object.
Then it sets the field

CommandGet Class

```java
package ist.meic.pa;

import java.lang.reflect.Field;
/*
 * Command Get
 * It receives a variable name and returns the value of the variable on the current moment
 *
 */
public class CommandGet {
    public static void main(String[] input){
        Field[] allFields = History.calledObjects.peek().getObject().getClass().getDeclaredFields();

        /*
         * Searches for the field that was given at the input and prints its value
         * */
        for(Field field : allFields){
            if(field.getName().equals(input[1])){
                try {
                    field.setAccessible(true);
                    System.out.println(field.get(History.calledObjects.peek().getObject()));
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Command Retry

```
public class CommandRetry {
    public static void main() throws Throwable{

        // Clone the stacks in order to have different stacks
        Stack<ObjectFieldValue> stack = (Stack<ObjectFieldValue>) History.calledObjects.clone();
        Stack<Object[]> stackArgs = (Stack<Object[]>) History.calledArgs.clone();

        //Get the last Object called and its info
        Object lastObject = stack.peek().getObject();
        Class lastObjectClass = stack.peek().getObject().getClass();
        String lastMethodName = stack.peek().getMethod();
        Class<?>[] lastParamTypes=(Class<?>[]) stack.peek().getArgs();
        Object[] lastArgs = stackArgs.peek();

        /*
         * Searches for the method that was interrupted and invoke it in the current object
         */
        for(Method m : lastObjectClass.getDeclaredMethods()){
            if (m.getName().equals(lastMethodName)){
                m.setAccessible(true);
                try {
                    m.invoke(lastObject, lastArgs);
                } catch (IllegalAccessException e) {
                    System.out.println(e.getCause());
                } catch (IllegalArgumentException e) {
                    System.out.println(e.getCause());
                } catch (InvocationTargetException e) {
                    System.out.println(e.getCause());
                }
            }
        }

    }
}
```

Re-invoke the method with the last arguments

CommandAbort Class

```
public class CommandAbort {
    public static void main(){
        System.exit(0);
        }
}
```

CommandReturn Class

```java
package ist.meic.pa;

import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class CommandReturn {

    public static void main(String[] args) throws InstantiationException, IllegalAccessException, ClassNotFoundException, IllegalArgumentException, InvocationTargetException{

        //initialize the Map in order to have an instance of the required type
        TypeMap.initializeMap();

        Class lastObjectClass = History.calledObjects.peek().getObject().getClass();
        String lastMethodCalled = History.calledObjects.peek().getMethod();

        //Gets the Method Class for the last called method
        for(Method m: lastObjectClass.getDeclaredMethods()){
            if(m.getName().equals(lastMethodCalled)){

                //Only do for non-void methods
                if(!m.getReturnType().toString().equals("void")){


                    Object obj = TypeMap.typeMap.get(m.getReturnType().toString());
                    Class<?> classType = obj.getClass();

                    //Gets valueOf with the parameter of the returned type
                    for(Method m1 : classType.getDeclaredMethods()){
                        if(m1.getName().equals("valueOf") && m1.getParameterTypes()[0].equals(args[1].getClass())){
                            Object o = m1.invoke(classType, args[1]);
                            MethodTranslator.c=o;

                            // Make the shell stop
                            DebuggerShell.canContinue=false;
                        }
                    }

                }else{
                    // Make the shell stop
                    DebuggerShell.canContinue=false;
                }

            }
        }
    }
}
```

Returns an object from typeMap which represents the return type of the method and invokes the method **valueOf** to return the object with the wanted type.
Then it puts the flag **canContinue** to false in order to stop the debugger.

TypeMap Class

```
public class TypeMap {
     static TreeMap typeMap = new TreeMap();


     public static void initializeMap(){
          typeMap.put("String", new String());
          typeMap.put("double",new Double(3));
          typeMap.put("int",new Integer(5));
          typeMap.put("byte", new Byte((byte)0xe0));
          typeMap.put("short", new Short((short) 6));
          typeMap.put("long", new Long(6));
          typeMap.put("float", new Float(5.8));
          typeMap.put("char", new Character('d'));
          typeMap.put("boolean", new Boolean(true));
     }
}
```

Given the type of return of the method, it returns an object representing that type