Automating Regression Testing of Java Programs the JSnoopy Way

Theodore S. Norvell
Electrical and Computer Engineering
Memorial University of Newfoundland

theo@engr.mun.ca

Abstract

As software systems evolve, errors sometimes sneak in; software that has been tested on certain inputs may fail to work on those same inputs in the future. Regression testing aims to detect these errors by comparing present behaviour with past behaviour. JSnoopy automates the processes of capturing behaviour across one or more interfaces, of replaying the calls to those interfaces, and of comparing the current behaviour to previously captured behaviour. JSnoopy uses Java's reflection capabilities, so that instrumenting an existing interface requires only replacing a single line of code.

1 Overview

Software development of large systems is sometimes a two steps forward, one step back process. The aim of regression testing [1] is to check that once the system behaves correctly for one class of inputs, it continues to behave correctly on that class of inputs, even as new features are being added and, more generally, as its behaviour on other classes of inputs is deliberately altered. This way any steps back can be quickly identified and dealt with. However, regression testing by manual means is time consuming and unlikely to be done as a routine matter. It is therefore worthwhile to consider automating regression testing to the extent possible.

JSnoopy is a Java library that allows automation of regression testing by recording all events (calls, returns, and throws) across major interfaces in an application. At a later date, the initiating calls can be replayed and the activity recorded again. The two records of activity are then compared and any differences are reported to the software engineer conducting the test.

Consider a Java program written using a model-view architecture; that is, a relatively stateless GUI portion of the program is layered above the business portion of the program which contains most of the state and logic for the application. For simplicity, suppose that, following the "façade" pattern [2], the model part of the program is represented by a single class Model implementing an interface ModelInterface and that, at run-time, this class has a single instance which we will call "model". The model updates the view by calling methods in an interface, ViewInterface, implemented by an object "view" representing the view portion of the system. We wish to do regression testing on the model portion of the application. JSnoopy would be set up to record all activity across the ModelInterface of the model object and across the ViewInterface of the view object. From a regression testing point-of-view, the model object is a black-box; calls to the model go in, and, in response, calls to the view come out. The ModelInterface interface of the model object is called an "instrumented interface" as is the ViewInterface of the view object.

2 Traces

An event is a pair consisting of a string, which identifies an instrumented interface in the system and one of the following

- call methodname($\texttt{arg}_0$, $\texttt{arg}_1$, ..., $\texttt{arg}_{n-1}$)
- return methodname value
- throw methodname value

Where the method names, arguments, and values are all strings.

A "trace" is a sequence of events. In a "proper trace", method calls and returns (whether normal or exceptional) are properly nested. Any call in a proper trace which is preceded by an equal number of calls and returns (whether normal or exceptional) is considered a "primary call".

JSnoopy operates in one of three modes. In normal mode, events are ignored. In recording mode, events are recorded to form a new trace. In playback mode, primary call events from a trace are injected into the system while, at the same time, events are recorded to form a new trace. After leaving playback mode the old and new traces are compared.

A simple GUI allows the user to set the mode of JSnoopy, to save and load traces, and to view pairs of traces with the differences indicated. After the user has been presented with a new trace that differs from an old trace, they may overwrite the old trace file with the new trace. This is appropriate if the system's behaviour has changed, but not in a way that indicates a defect.

3 Instrumenting Interfaces

One of the design goals of JSnoopy was to make its use easy and for its use to have a very low impact on the design and coding of applications. Consider our simple model/view application described above. The code for the main program might be as shown in Figure 1.

```
class Main {
    static public void main( String[] args ) {
        // Create the model and the view
        ModelInterface model = new Model() ;
        ViewInterface view = new View( model ) ;
        ... } }
```

Figure 1: Original main method for an example application.

In order to add JSnoopy to this system, one must ensure that the appropriate interfaces are instrumented. This is done by modifying the code as shown in Figure 2.

The first block of code initializes JSnoopy based on the value of a system property "jsnoopy". Such system properties are typically set via a command line argument to the command used to initialize the Java virtual machine. If the argument to method jsnoopy.JSnoopy.setActive is false, then JSnoopy is initialized to a passive state and has no ability to record or playback traces. If the argument is true, then JSnoopy is initialized as "active" and the JSnoopy GUI will appear to allow the supervising software engineer to use the system.

The other change to the system involves the creation of the model and view objects. As soon as these objects are created they are passed to JSnoopy via method instrument to be instrumented. The instrument method returns an object implementing the appropriate interface, and which behaves, in most respects, identically to the original model or view object.

The instrument method has three parameters

```
class Main {
    static public void main( String[] args ) {
        // Initialize jsnoopy.
        boolean activateJSnoopy ;
        try { activateJSnoopy = System.getProperty( "jsnoopy" ) != null ; }
        catch( SecurityException e ) {activateJSnoopy = false ; }
        jsnoopy.JSnoopy.setActive( activateJSnoopy ) ;

        // Create the model and the view
        ModelInterface model = (ModelInterface)
                jsnoopy.JSnoopy.getInstrumentor().instrument(
                "model",
                new Model(),
                ModelInterface.class ) ;
        ViewInterface view =(ViewInterface)
                jsnoopy.JSnoopy.getInstrumentor().instrument(
                "view",
                new View( model ),
                ViewInterface.class ) ;
        ... } }
```

Figure 2: Main function modified for JSnoopy

- name: a string used to identify the instrumented interface in traces
- baseObject: a base object to be instrumented
- interfaceToInstrument: a java.lang.Class object which is the run-time representation of an interface.

It is required that baseObject implement the interface represented by interfaceToInstrument. If JSnoopy is in a passive state, the instrument method simply returns baseObject, thus there is no effect on the system. If, however, JSnoopy is active, the instrumentor object creates a proxy object which implements the interface represented by interfaceToInstrument. The proxy object is returned from the instrument method. The proxy objects maintain a link to the base object. When a method of the proxy object is invoked, the information about the call may be recorded by JSnoopy, and then the base object is invoked with the same method and the same arguments. On return from the invocation of the method on the base object, the proxy object may record a return event, and then the invocation of the method on the proxy object returns with the same result. A similar mechanism allows recording of exceptions.

The key point here is that the creation of the proxy objects is fully automated by JSnoopy and requires no modification of the underlying classes. The only code that needs to be added to the system is at the point of the creation of the instrumented interfaces.

4 Usage Considerations

Although there is very little visible impact on the system this is not to say that JSnoopy can be trivially added to any existing system.

First, the instrumented interfaces must literally be interfaces as opposed to classes. That is, the programmer must compose a Java interface to represent any instrumented interface. In many cases, this interface would be written anyway in the interests of good software design.

Second, the values (arguments, return values, exceptions) that cross any instrumented interface must be representable as strings. This is because JSnoopy records these values in the trace as Unicode strings. The reason for this is to make the traces easily readable; this is particularly important when a change in behaviour is detected and the differences between two traces must be presented. There is no problem for values of primitive type (for example boolean and int) or arrays. For objects that are not arrays, the toString method is used. It would be possible to serialize objects that appear in traces, but the resulting loss in readability of traces would be too much of a problem.

Furthermore, arguments to primary calls must only have arguments that are primitive, strings, or arrays of primitive or string values. This is because JSnoopy must be able to recreate the arguments of the primary call exactly, in order to inject the primary call during replay. In future versions, serialization may allow a wider variety of parameters for primary calls.

Third, JSnoopy does not work in the presence of concurrency. All calls that cross instrumented interfaces are expected to come from the same thread. Without this requirement it would be difficult to determine which calls are truly primary and which arise in the system as a reaction to primary calls. Without being able to identify primary calls, replay is not possible.

Fourth, JSnoopy requires determinism. The main source of nondeterminism in Java is concurrency, which has already been ruled out on other grounds. Other sources of nondeterminism include use of random numbers and dependence on information that is not constant, such as the current time, or the state of the file system. The point of JSnoopy is not to detect deviation of a system from acceptable behaviour, but rather deviation from past behaviour which has been judged acceptable. Thus new but different acceptable behaviour results in false negative reports, which must be dealt with.

5 Implementation

5.1 Proxies

JSnoopy's creation of proxies is greatly aided by the use of Java's reflection capabilities in the java.lang and java.lang.reflect packages of the standard Java library.

In particular the Java library class java.lang.reflect.Proxy has a method

```
public static Object newProxyInstance( ClassLoader loader, Class[ ] interfaces, InvocationHandler h)
```

which returns an object that implements the given interfaces. This object routes all invocations to the given InvocationHandler. JSnoopy uses a custom implementation of the InvocationHandler interface to record events and to forward invocations to the base object.

An example of the typical sequence of calls, while recording, is shown in Figure 3. The view object invokes method go of the ModelInterface, the InvocationHandler for the proxy object records the call and the return in the trace. In between, the InvocationHandler calls the actual model object. This call is effected using the invoke method of the Method class, with a reference to the model object passed in as an argument. Both the Method object representing the go method and a representation of any arguments are parameters of the invoke method of the InvocationHandler interface.

JSnoopy keeps WeakReferences to the proxy objects so that the use of JSnoopy will not interfere with the garbage collection of the base objects.

5.2 JUnit Integration

JUnit [3] has become popular among Java programmers for unit testing. Unit testing is complementary to regression testing in that tests are usually on small components rather than the broad system. In JUnit, tests are represented by classes implementing the Test interface. JSnoopy provides an abstract class ReplayTestSuite which implements the Test interface. The user can then easily

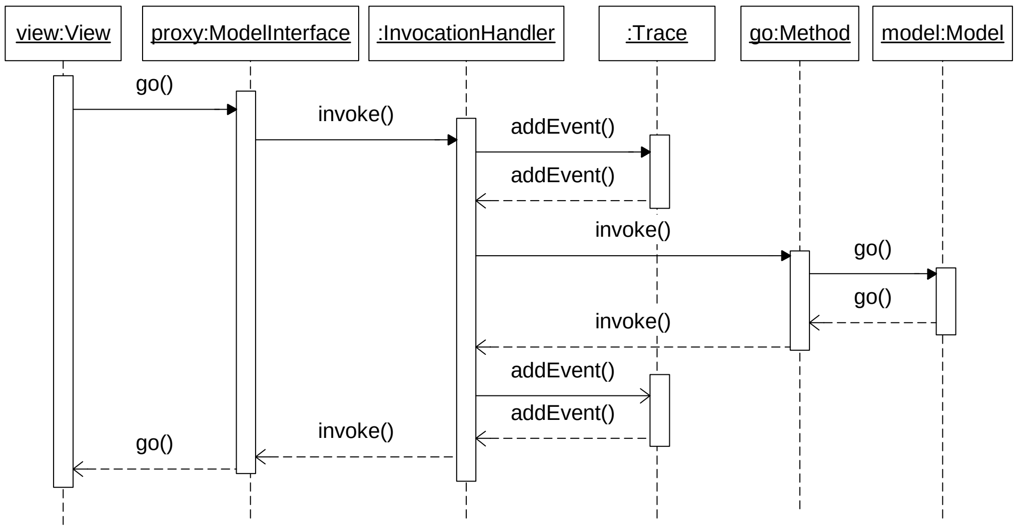

Figure 3:

extend ReplayTestSuite to obtain a class representing a suite of tests, each test running a different trace file. This allows any of JUnit's test runners to be used to run the JSnoopy tests.

5.3 Distribution

JSnoopy is distributed in a choice of two JAR files. One JAR file contains the full JSnoopy system and is intended for use during testing. The other consist of only a few classes; in this configuration JSnoopy always remains passive. The second JAR file is intended for shipping with the final product. Thus the few lines of code added to the application to support JSnoopy need not be removed from the final product, and the use of JSnoopy need only add a few (currently four) kilobytes to the final shipped version of a product.

6 Conclusion

JSnoopy automates the processes of recording, replaying, and comparing traces in Java applications. There is potentially a large benefit for software developers to improve the quality of their products, to be assured of that quality, and to decrease the effort to achieve quality.

References

[1] Robert V. Binder, Ed., Testing Object-Oriented Systems, Addison-Wesley, 2000.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns, Addison Wesley, 1995.

[3] Erich Gamma and Kent Beck, "JUnit: Testing resources for extreme programming," 2002, URL:www.junit.org.