Jaunt: A Productive Detour of Automated Testing

Miles Lasater
Yale University
P.O. Box 201284
New Haven, CT 06520

Advisor
Prof. Carsten Schürmann
Yale University

Spring 2001
1 Abstract

Jaunt is a Java-based and Java-targeted automated testing tool that runs tests on classes with randomly generated data. Meant to augment unit testing conducted by developers in methodologies such as eXtreme Programming, or other test-first design methods, Jaunt should be integrated into the design-code-test cycle. Similar approaches with generating random input have produced meaningful results [MFS90, Woo99, FM00, MKL+00], and we are unaware of a Java implementation of this type of testing.

When Jaunt is executed on a class, it reads the methods of that class. Next it analyzes each method for the type of each parameter, generates a random object that matches each type and executes the method. The method is invoked without an object, in the case of static method, or it is invoked within an object of randomly generated state.

The ability to generate random objects of the eight primitive types, String and Object are built into Jaunt. To create random objects of other types, Jaunt randomly selects a constructor, feeds it randomly generated parameters, and then invokes a random number of methods on the object, in an effort to change its state. If unsatisfied with the built-in random generation for a particular object, users are encouraged to create their own. For some types of classes, this may produce the best results.

Making use of introspection and reflection, Jaunt can be used on classes even when the source code is not available. In this way, Jaunt differs from other automated test data generation tools that rely on access to the source to execute properly. A user other than the original developer can use Jaunt to help verify a program’s or class’ robustness.

Although currently Jaunt runs as a stand-alone piece of software, future development will integrate it with JUnit, a popular Java-based unit tester, to allow for greater ease of use. Jaunt used in this capacity will allow for the automatic generation of repeatable tests, which become part of the test suite.
2 Introduction

2.1 Background

Given the high cost of bugs in software, we should be highly motivated to rid any software for which we are responsible of bugs. It is easy to assert this admonishment, but finding and eliminating bugs is difficult. Some argue that the mind that delights in creating cannot properly feel the sadistic thrill of destroying the object of creation. Perhaps familiarity with the purpose of code breeds blindness about its inability to carry out that purpose perfectly. Whether or not one believes these arguments, we know that the general problem of proving a program’s correctness (“bug-freeness”) is non-trivial.

In addition to the difficulty of the locating bugs there is significant inertia against proper testing. The widespread “waterfall” project management paradigm, which calls for analysis, design, implementation and test, in ordered atomic steps, leaves testing until the last. Due to optimism, mismanagement or the difficulty inherent in prediction, interim deadlines are often missed and testing is cut short, in order to hit the final deadline. Particularly in commercial projects the motivation to ship is strong. Lenient software warranties that allow companies to ship products “as-is” provide little motivation to eliminate errors. Often the economic driver is to ship a new version with more functionality, rather than to eliminate bugs in the older version, as long as it the old version works well enough.

Whatever the reason, it is a cliché that software developers hate debugging and despise testing.

Cliché it may be, but is it true? One movement in software development has challenged this assumption and caught the imagination of many, including this author. Witness the virtual explosion of interest in eXtreme Programming (XP), along with one of its central tenants that developers should test often, even as often as every few minutes.[Bec99, BF01, GB, Iva] This frequency is made possible by automated testing tools.

Automated testing tools are nothing new. But pushing testing earlier into the development cycle and making it part of the coding process is a break from traditional approaches. By putting testing back in the hands of developers who use a tight design-code-test cycle, using automated testing tools teams report that productively increases and few bugs make it out of the development codebase. So, a number of developers have embraced this style because they believe that:
• writing tests before implementing helps with improving design and therefore can be accomplished by developers engaged in creation,

• creating tests before coding and having the machine execute the tests reduces the problem of familiarity,

• the tight cycle, different from the waterfall approach, does not leave all testing until the end, so there is not pressure to deeply curtail it, and

• working with code that has fewer bugs improves the productivity of developers so there is an economic benefit from this type of automated unit testing.

Another “bug” of sorts that is not addressed by the popular automated testing tools is the incomplete or inaccurate documentation. It is another cliché that engineers do not enjoy documenting their work. One manager, Keenan of Keenan Systems, encountered this problem so often that we crafted a clever rationale and motivator for developers to share their knowledge. If you are the only one that understands this system, he reasons, you become indispensable to it and therefore cannot be moved to another project. When it breaks you will go wherever this system goes; even if that means you must travel at inconvenient times to inconvenient places. Documentation is freedom, he claims.

One well-known tool that significantly aids developers in creating quality documentation, especially for libraries, is Javadoc. By inserting specially formatted comments into the code itself, the developer creates signposts for Javadoc to use the source code itself to create a series of documentation web pages.

In this paper, we will provide further motivation for Jaunt (§2.2), explain our approach and Jaunt’s execution (§3), its implementation (§4), and conclude with a discussion of related work(§5.1), and outline of future work (§5.2).

2.2 Motivation for Jaunt

The popular Java unit testing tool JUnit requires coders to write their own tests. While this has advantages (mentioned earlier), the downside is that if they do not foresee certain types of parameter inputs, they might fail to
test for them. If well trained and conscientious, the developer will test for boundary conditions. However, this might not always be the case.

Jaunt, developed to augment the arsenal of an XP-coder or unit-tester, does not replace JUnit. In fact, it was originally intended to integrate with JUnit. Jaunt does not rely on the user to write tests, but rather dynamically, pseudo-randomly, creates input for the object under test. Creating random input, in some cases, almost garbage input, manipulates the class in unexpected ways, shaking free, by force, unforeseen exceptions. Jaunt, therefore, is useful for exposing certain types of bugs.

Another purpose of Jaunt is to find incomplete documentation. By forcing the class under test to raise different types of exceptions, it allows the tester to see a broad range of exceptions. These can be cross-checked against the documentation to see if the exception should be added. Eventually, this cross-checking will be automated, at least in part, in conjunction with Javadoc.

Both when testing code against documentation and when testing a program for correctness, it is important to recognize the difference between two classes of exceptions. One is true exceptions, or error conditions, that mean that something has gone wrong with execution that was recognized by the runtime engine or the program itself. These true errors, or “bad” exceptions, probably cannot be recovered from and the conditions that give rise to them should be eliminated from the program. In other words, they are real bugs.

The second type are “good” exceptions or those that carry semantic meaning. Often exceptions that are declared, they serve the purpose of passing messages back from a method that had some difficulty. For example, a method responsible for parsing a String into an Integer, may find that String is not formatted correctly for parsing. An exception thrown at this point, is a message to the calling method that the input was malformed. In cases like these, it is useful to stick to this messaging structure rather than expecting the calling method to check for the proper formatting of the String. If it could recognize the proper format, it would be a long way towards parsing it.

With that background in mind, let us turn our attention the Jaunt approach to automated testing.
3 Approach

Jaunt runs on any Java .class file to exercise each of its methods on random input in an attempt cause the class to raise an unexpected, unwanted, mishandled or undocumented exception. Input passed as parameters into the methods of the class being tested will be of the proper type, but may not have the state explicitly or implicitly expected by the method. Passing parameters that are valid in syntactic and type, but perhaps not in state, is part of Jaunt’s strategy to unearth bugs. For example, a method called with the signature of “void openFile(String filename)” may choke on Strings that are not valid filenames. Jaunt may aid a developer in discovering this and the problem may be corrected with better exception handling, or a more specific type, like java.io.File, might be used instead.

Jaunt randomly generates test data and then executes the class on that data using Java’s introspection/reflection capabilities. When Jaunt is executed on a class, it reads the methods of that class. Next it analyzes each method for the type of each parameter, generates a random object that matches each type and executes the method. The method is invoked without an object, in the case of static method, or it is invoked within an object of randomly generated state.

When the method under test throws an exception, the exception is reported to the user along with the input that caused the exception. At this point, the user can specify if this particular exception should be ignored when raised by this method, or if further cases should be displayed. Users may want to ignore exceptions considered good ones. A command line switch also allows users to ignore exceptions that are declared by the method. By using this option, users are acting as if all declared exceptions are good ones.

In order to understand Jaunt in more detail, we will examine the process used by Jaunt when run on the included class ToBeTested. On start-up Jaunt parses the command line input and then prompts the user for the names of the classes the user wishes to test. After each name is entered Jaunt attempts to load the class and notifies the user if there is some error. Usually, the error is because of a misspelling or an improperly set CLASSPATH. In this case, let us assume the user entered “ToBeTested”.

During this process, Jaunt is loading the matching java.lang.Class for the named class using the static method Class.forName(). The Class for ToBeTested is then stored in the Codebase HashMap so that Jaunt remembers the classes it is currently testing.
The user is prompted for another class name, but in our example, we only want to run tests on ToBeTested, so assume the user enters a blank line to terminate the list.

Next, Jaunt prepares the objects responsible for returning random objects of the correct type. These ClassTest’s are stored in a ClassTestMap, which links a given Class to the proper ClassTest. The ability to generate random objects of the eight primitive types, String and Object are built into Jaunt, so each has its own ClassTest. For example, IntInputTest, which extends ClassTest, returns a random java.lang.Integer, the wrapper object for the primitive int. Each primitive is treated in this way, so if one wants the underlying primitive, one must unwrap it using getValue().

Having completed the set-up, Jaunt now iterates over each Class in the Codebase. In this case, it selects ToBeTested and grabs its methods. Reading the methods of the class is achieved by use of Java’s Class.getMethods() call. Information about the method is returned in a java.lang.reflect.Method. Next, each method is tested using Jaunt’s invokeMethod().

If the method is non-static, in order to invoke it, we must have an object of the correct type to invoke it with. To this end, InvokeMethod() creates a random object of the type that declared the Method under test. This is accomplished by a look-up in the ClassTestMap and a call to the appropriate random() method. In the case of ToBeTested.noNegative(), no object is created at this point. In the case of ToBeTested.allPrim(), one is created.

If the class currently being tested is not one for which a ClassTest already exists in the ClassTestMap, then the ClassTestMap is responsible for creating, storing and then returning a GenericClassTest that will serve the purpose. This is the facility that allows Jaunt to create random objects of types it has not encountered before. To create random objects of these types, the GenericInputTest randomly selects a constructor, feeds it randomly generated parameters (again with look-ups in ClassTestMap and calls to the appropriate random()), and then invokes a random number of methods on the object, in an effort to change its state.

If unsatisfied with the results of the GenericInputTest or the other built-in ClassTest’s, users are encouraged to create their own. For some types of classes, this may produce the best results. In order to create one’s own ClassTest, extend ClassTest, implement random() and either add one’s new test to the constructor of ClassTestMap to ensure it is automatically loaded or putClass() it into the map before test execution.

To continue, invokeMethod() then finds the types of the parameters of the
method using Method.getParameterTypes(), creates random objects for each of the parameters and runs the method on those parameters. Method.invoke() is used to run the method on given underlying object and parameter list. In our example, noNegative() is fed an int which has been unwrapped by invoke() from an Integer. About half the time, the random number will be negative and therefore noNegative() will throw the exception OutOfBounds.

When an exception is thrown, like OutOfBounds, the user is notified of the method, the exception type and the input which caused the exception. At this point the user can choose to see more exceptions like this or ignore further ones.

invokeMethod() runs the method many more times on random input. The default number of times is 100, but it can be changed via a command line switch. Once a particular method is tested, Jaunt tests the next one, and so on until all the methods of that class are tested. If there are no more classes to test, as in our example, Jaunt exits.

As mentioned before and should be apparent in this example, Jaunt relies heavily on Java’s introspection capabilities. Using Java’s built-in introspection provides the benefits of not having to parse source code. If Jaunt had its own built-in parsing it might become out-of-step with the Java specification. By leveraging the javac parser, Jaunt ensures that the Java version being employed by the user is the one it uses. Another effect of using introspection is that Jaunt can be used to test compiled code.

4 Implementation

Implemented and tested using Java 2 SDK, Standard Edition (Version 1.3.0 Beta Refresh for Linux) using the bundled JVM: Java 2 Runtime Environment, Standard Edition (build 1.3.0beta_refresh-b09) and Java HotSpot Client VM (build 1.3.0beta-b07, mixed mode). Jaunt, to date, has only been run on Java 2 code and may not work properly with Java 1.1 or earlier.

4.1 Command Line Input

There a number of command-line switches that change the behavior of Jaunt when it is run. They include:

- -help or -h, which print the usage statement and then quit
• -i, which causes Jaunt to ignore exceptions that are declared by the currently tested method. For example, since java.lang.Integer(String) declares that it throws NumberFormatException, when testing the class, we might not want to be notified of semantic exceptions like this.

• -n int, sets the maximum number of classes can be tested in one Jaunt run. For example, if input is coming from a file, one might want to only test the first 10 classes. Default is 256.

• -obj, instructs Jaunt not to test methods that are inherited from java.lang.Object. This useful, because they can often raise intrusive, but often unimportant exceptions when the input is not what they expect.

• -t int, specifies the number of times each method is tested. Default is currently set at 100. Note: if the range of acceptable input is relatively small, the method may never be tested on “normal” input. For example, a randomly generate int across the full range of int may never hit on the 1 to 10 expected and/or meaningful to the method.

• -version or -ver, prints the version information and quits

• -verbose or -v turns on verbose mode which provides the user more feedback on what test are happening at which time.

4.2 Other User Input

When Jaunt runs it prompts the user for the names of class files that should be tested. If the classes cannot be found or loaded Jaunt notifies the user. Using the fully qualified name may help if there is difficulty locating a class. Each class name should be separated by a line break.

Once tests have begun, user interaction is limited to reviewing exceptions raised and choosing if it is a meaningful exception worthy of user notification. When run with no command line switches on java.lang.Integer, the NumberFormatException may be raised on method “void openFile(String filename)”. Jaunt will prompt the user if this is considered an acceptable exception or if it is not and the user would like to see the details on similar exceptions. If the exception is considered acceptable, the user will not be notified if the same exception is raised by that method again.
4.3 Internal Representation

The major classes that do the work in Jaunt are:

- jaunt: Jaunt is the class in which execution takes place. It contains the main() method, parses user input, loads the class files to be tested into the Codebase,

- Codebase: An extension of HashSet to store the Class’es of the classes being tested by Jaunt in its current run.

- ClassTest: An abstract class of test that returns a valid random object of the appropriate class. Implemented children include IntInputTest.

- GenericClassTest: An extension of ClassTest, which is able to return a “random” object of whichever class it is set-up to imitate. It creates the random object by choosing a random constructor, feeding it random inputs until a constructor succeeds. Next, it runs a random number of methods with random inputs to change the state of the object.

- ClassTestMap: An extension of HashMap to store the mapping from the a Class object to the appropriate ClassTest for that class

ClassTests have been implemented for the eight Java primitives (for example, IntInputTest), Object and String. User of Jaunt are encouraged to extend ClassTest to create specific and well-tuned tests for their own classes. GenericClassTest will ensure that there is a test for any class, but a tailor-made one will function better.

5 Discussion

5.1 Related Work

Automated generation of test data is not a new field. Many have made a number of important contributions in this area. There are three major categories of these approaches: [Kor96]

- random test data generators,

- path-oriented test data generators,
• and dynamic test data generators.

One well-known example of random test generation was Fuzz, in which the authors fed large amounts of random characters into the stdin of popular Unix utilities in order to find and identify the cause of program crashes. [MFS90] Others have continued the same tradition of work. [Woo99]

A path-oriented approach that was fairly successful was implemented using genetic algorithms. [MMSW97] What interested the authors was ensuring that the tests generated achieved code coverage, or more precisely condition-decision coverage. One of the drawbacks of Jaunt, and random approaches, in general, is that the tests generated may not cover all the cases.

Some tools, do not generate tests themselves, but rather analyze existing tests to determine if the tests fully exercise the code. One freely available code coverage tool, JesTer, integrates with JUnit. [Iva]

We are not aware of Java implementation of random input to each object method. There well may be one, but we have not found it.

5.2 Future Features

Although a solid base for Jaunt has been established, compiles and runs without apparent error, many future features are planned for implementation. Before Jaunt becomes widely adopted these feature and perhaps others may have to implemented

Enhancements include:

• A more verbose level that shows the parameter inputs in each method called in addition to exceptions thrown would be helpful for users. A stack trace of the thrown exception would also be useful.

• A Jaunt updateable database or configuration file with a information on previous Jaunt runs and user choices about acceptable exceptions would also be beneficial.

• Development of an abstract interface specification language for use with Jaunt is a powerful next step. If one could specify not only that a method should return an integer, but also that the method should return an integer in the range zero to ten, automated testing tools would become more useful. The same can be done with inputs. Although a method may explicitly only require a String, it may actually only be semantically meaningful and useful if the String is the name of a color.
• In an effort to improve documentation, the source underlying the class file could be checked against Javadoc generated documentation or comments to see if thrown but undeclared exception should be included in the documentation.

• To fully integrate into the development cycle of a XP-coder, the automatic creation of JUnit tests would be desirable.

References


