e-mail it!

Incremental development with Ant and JUnit
Using unit test to improve your code in small steps

Malcolm Davis
Consultant
November 2000

A small change in your software development habits can have a huge payoff in the quality of your software. Incorporate unit testing into your development process and see how much time and effort you save in the long run. This article explores the benefits of unit testing, in particular using Ant and JUnit, with code samples.

Contents:

One of the basics of a great development process is testing. Verification of work is an important part of any profession. A doctor might confirm a diagnosis with a blood test. Boeing tested airplane components during the development of the 777. Why should software development be any different?

Previously, tight coupling of the GUI and business logic in an application limited the ability to create an automated test. As we learned to decouple business logic from the interface, through layers of abstraction, automated testing of single modules of code replaced manual testing via the GUI.

Integrated Development Environments (IDEs) now display errors as you type your code, have intel-sense for quick look-up of a method in a class, use syntax coloring, and have many other features. Consequently, before you compile code that you have changed, you have a good idea that the class will build, but will your modifications break some function?

The change bug has bitten every developer. During the modification of code, a bug might be introduced, which is not found until after compilation when the code might be manually tested via a user interface. Then, you spend days tracking down an error introduced by the change. This happened to me recently on a project where the back-end database was changed from Informix to Oracle. For the most part, the changes went smoothly. However, the lack of unit test on the database layer, or systems using the database layer, led to a great deal of time spent trying to resolve change bugs. I spent two days tracking down one database syntax change, in someone else’s code. (Yes, that person is still my friend.)

Despite its benefits, testing does not excite the average programmer, and initially it did not excite me. How many times have you heard "it compiles, therefore it must work"? The principle of "I think, therefore I am" does *not* apply to high quality software. To encourage programmers to test their code, the processes must be simple and painless.

This article begins with a simple class that one writes when learning to program in the Java language. I will then show you how I would write a unit test for this class, after which I will add the unit test to the build process. Finally, we will see what happens when I introduce a bug into the code.

Start with a typical class

The typical first Java program contains a `main()` that prints "Hello World." In Listing 1, I create an instance of the HelloWorld object and call the `sayHello()` method, which prints the customary saying.

Listing 1. My first Java "Hello world" application

```
/*
 *  HelloWorld.java
 *  My first java program
 */

class HelloWorld {
    /**
     * Print "Hello World"
     */
        void sayHello() {
          System.out.println("Hello World");
     }

    /**
     * Test
     */
    public static void main( String[] args ) {
        HelloWorld world = new HelloWorld();
        world.sayHello();
    }
}
```

The `main()` method is my test. Wow! I have code, documentation, testing, and example code all in one module. Amen to Java! As my programs grew larger, however, this method of development quickly started to show drawbacks:

- Clutter
  The larger the class interface, the larger my `main()` grew. The class could become huge just due to proper testing.
- Code bloat
  The product code is larger than necessary because of the test. But I didn't want to deliver the test, just the product.
- Testing is unreliable
  Since `main()` is part of the class, `main()` has access to private member and methods that other developers would not have access to via the class interface. For this reason, this method of testing is error prone.
- Difficult to automate a test
  To automate I would still have to create a second program to pass in arguments to `main()`.

Class development
For me, class development started with coding the `main()` method. I would define the class and the usage of the class as I wrote `main()`, and then implement the interface. This also began to show obvious drawbacks. One drawback was the number of arguments I was passing into `main()` to execute the test. Secondly, `main()` itself became cluttered with calls to submethods, setup code, and so on. There were occasions when `main()` was larger than the rest of the class implementation.

A simpler process
It's easy to see the pitfalls in my original approach. So, let's look at an alternative process that will simplify life. I still design my code via an interface and show example usage, just as I did with my original `main()`. The difference is that I put the code in a separate class. This separate class also just happens to be my "unit test." This technique provides several benefits:

- A mechanism for designing the class
  Because I'm developing via an interface, I am less likely to take advantage of internal class features. But since I am the developer of the target class, I have a window into its internal workings. Therefore the test is not a true black box. This point alone is reason enough to require that the developer responsible for writing the target class be responsible for developing the test, and not someone else.
- An example of class usage
  By separating the example from the implementation, developers can more rapidly get up to speed. No more stumbling over source code. This separation also helps eliminate the temptation of developers to take advantage of internal class

features that may not be there in the future.

- A `main()` without the class clutter
  I'm no longer limited by `main()`. Previously I passed in multiple parameters to `main()` to test different configurations. Now I can create separate test classes, each maintaining individual setup code.

We can go one step further by putting this separate unit test object into the build process. By doing this we can provide a way of automating the validation process.

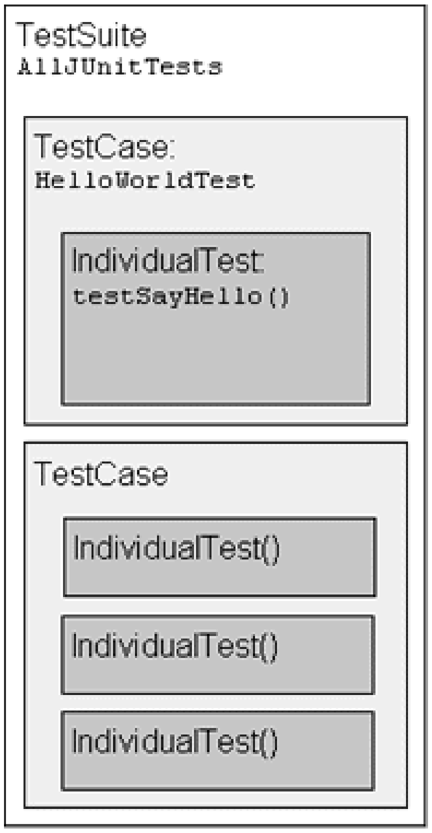
- We verify that any changes we make do not adversely affect someone else.
- Instead of waiting for assembly testing or nightly build test, we can test the code before we check into source control. This helps catch bugs early in the process, thereby lowering the cost of producing quality code.
- By providing an incremental test process, we provide a better implementation process. Just as IDEs help us catch syntax or compile bugs as we type, incremental unit tests help us catch code-change bugs as we build.

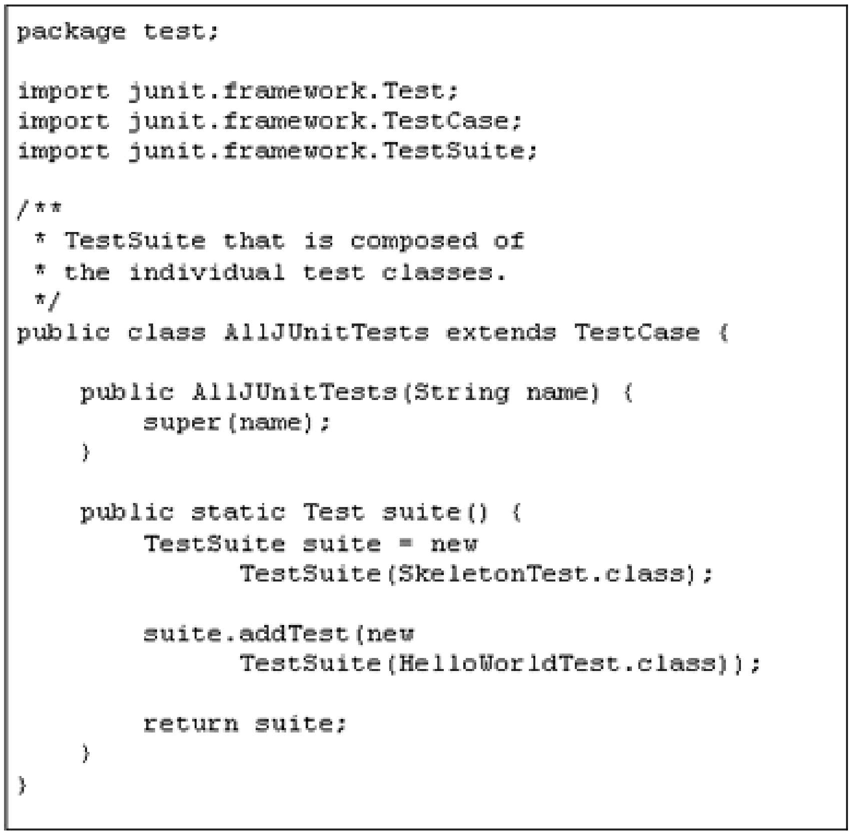
Automate unit testing with JUnit
To automate testing, you need a testing framework. You can develop your own, buy one, or use some open-source tool like JUnit. I'm into JUnit for several reasons:

- I do not have to write my own framework.
- It is open source, so I do not have to buy a framework.
- Other developers in the open-source community use it, so I can find a lot of examples.
- It allows me to separate test code from product code.
- It is easy to integrate into my build process.

Test Layout
Figure 1 displays the JUnit TestSuite layout with a sample TestSuite. Each test is made up of individual test cases. Each test case is an individual class that extends TestClass and contains my test code, that is, the code that used to be in my `main()`. In the example I have added two tests to the TestSuite: a SkeletonTest that I use as a starting point for all new classes and my HelloWorld class.

Figure 1. TestSuite layout





```
package test;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

/**
 * TestSuite that is composed of
 * the individual test classes.
 */
public class AllJUnitTests extends TestCase {

    public AllJUnitTests(String name) {
        super(name);
    }

    public static Test suite() {
        TestSuite suite = new
              TestSuite(SkeletonTest.class);

        suite.addTest(new
              TestSuite(HelloWorldTest.class));

        return suite;
    }
}
```

The test class HelloWorldTest.java

By convention, the test class consists of the same name as the class I am testing, but with *Test* appended to the end. In this case our test class is `HelloWorldTest.java`. I copied the code from the SkeletonTest, and added a `testSayHello()` to test `sayHello()`. Note that HelloWorldTest extends TestCase. The JUnit framework provides `assert` and `assertEquals` methods that we can use for verification. `HelloWorldTest.java` is displayed in Listing 2.

Listing 2. HelloWorldTest.java

```
package test.com.company;

import com.company.HelloWorld;
import junit.framework.TestCase;
import junit.framework.AssertionFailedError;

/**
 * JUnit 3.2 testcases for HelloWorld
 */
public class HelloWorldTest extends TestCase {

    public HelloWorldTest(String name) {
        super(name);
    }

    public static void main(String args[]) {
        junit.textui.TestRunner.run(HelloWorldTest.class);
    }

    public void testSayHello() {
        HelloWorld world = new HelloWorld();
        assert( world!=null );
        assertEquals("Hello World",  world.sayHello() );
    }
}
```

The `testSayHello()` looks similar to my original main method in `HelloWorld.java`, but there's one major difference. Instead of executing a `System.out.println` and doing a visual confirmation of the results, I have added an `assertEquals()` method. If the two values are different, `assertEquals` will print out the value for both inputs. You may have noticed that this will not work! The `sayHello()` method in HelloWorld does not return a string. If I had written the test first, I would have caught this. I have coupled the "Hello World" string to the output stream. So, I rewrite the class HelloWorld as shown in Listing 3, eliminating the `main()`, and changing the return type of `sayHello()`.

Listing 3. Hello world test class.

```
package com.company;

public class HelloWorld {
    public String sayHello() {
        return "Hello World";
    }
}
```

If I had kept the `main()` and fixed the coupling, it might have looked something like the following:

```
 public static void main( String[] args ) {
        HelloWorld world = new HelloWorld();
        System.out.println(world.sayHello());
    }
```

The new `main()` looks remarkably similar to `testSayHello()` in my test program. Yes, this does not look like a real world problem (this is the issue with contrived examples), but it makes a point. Writing your `main()` in a separate application can improve your design and help you design for testing at the same time. Now that we have created a test class, let's integrate it into the build by using Ant.

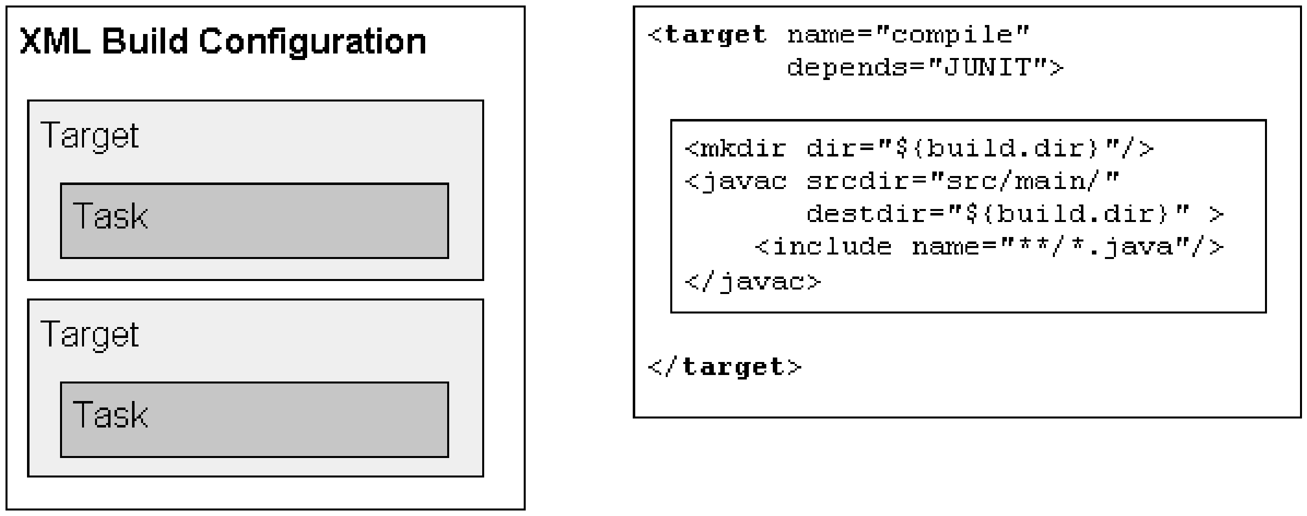
Integrate testing into the build with Ant
The Jakarta Project refers to the Ant tool as the "make without make's wrinkles." Ant is becoming the de facto standard in the open-source world. The reason is simple: Ant is written in the Java language, which allows the build process to work on multiple platforms. This feature simplifies collaboration between programmers on separate OS platforms, a requirement for the open-source community. Develop *and* build on your platform of choice. Ant features include:

- Class extensibility
  Java classes are used to extend build features instead of using shell-based commands.
- Open source
  Because Ant is open source, class extension examples are plentiful. I find learning by example to be great.
- XML configurable
  Ant goes beyond just being Java based. Ant uses an XML file for configuration of the build process. Given that builds are hierarchical in nature, using XML to describe the make process is logical. Also, if you know XML, learning how to configure a build is easier.

Figure 2 outlines a configuration file. The configuration file is made up of a target tree. Each target contains tasks that get executed. A task is code that can be executed. In the example, *mkdir* is a task of target *compile*. *mkdir* is a task built into Ant that creates a directory. Ant comes with a healthy list of built-in tasks. You can also add your own functionality by extending the Ant task class.

Each target has a unique name and optional dependencies. Target dependencies are required to execute prior to the execution of the target's own task list. In Figure 2, the JUNIT target is required to run prior to executing the task in compile target. This type of configuration allows you to have multiple trees in a configuration.

Figure 2. Ant XML build diagram



```
<target name="compile"
        depends="JUNIT">

  <mkdir dir="${build.dir}"/>
  <javac srcdir="src/main/"
         destdir="${build.dir}" >
     <include name="**/*.java"/>
  </javac>

</target>
```

The similarities to the classical make utilities are striking. That is to be expected since a make is a make. But keep in mind the differences: cross platform, extensible via Java, configurable via XML, and open source.

Download and install Ant
Start by downloading Ant (see Resources). Decompress Ant to your tools directory. Add the Ant `bin` directory to your path. (On my machine that was `e:\tools\ant\bin`.) Set the ANT_HOME environment variable. In NT that means going to your system properties and adding ANT_HOME as a variable with a value. ANT_HOME should be set to the Ant root directory, the directory that contains the `bin` and `lib` directories. (For me that was `e:\tools\ant`.) Make sure you have a JAVA_HOME environment variable that is set to the directory where the JDK is installed. The Ant documentation has more information on installation.

Download and Install JUnit
Download JUnit 3.2 (see Resources). Unzip the `junit.zip` and add `junit.jar` to the CLASSPATH. If you unzipped

`junit.zip` into your classpath path, you can test your installation by running the following command:
`java junit.textui.TestRunner junit.samples.AllTests`

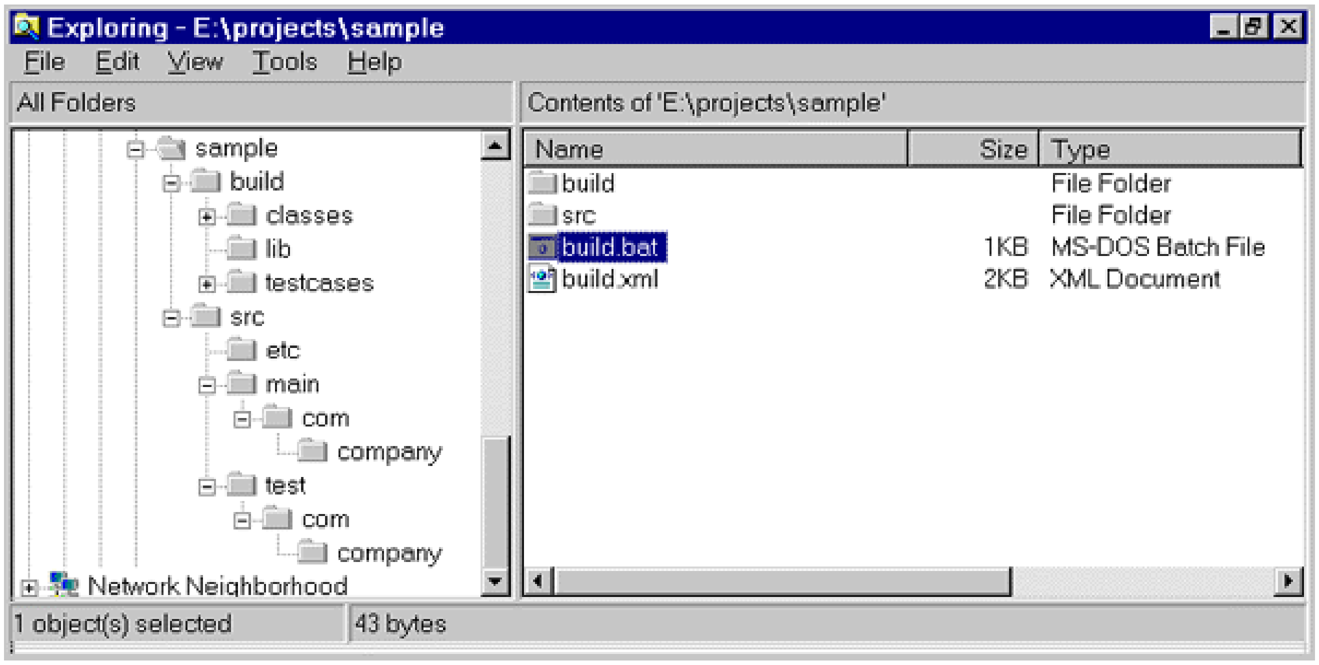
Define the directory structure
Before starting our build and test process, we need a project layout. Figure 3 displays my sample project layout. The following describes the directory structure of the layout:

- `build` -- Temporary build location for class files. The build will create this directory.
- `src` -- Location of the source code. `Src` is subdivided into a `test` folder for all the test code and a `main` folder containing the deliverable code. Separating the test code from the main code provides several features. First, it keeps clutter down in the main code. Second, it allows Package alignment. I am a big fan of aligning classes with the package they are associated with. Test should be with test. This also helps the distribution process because it is highly unlikely that you want to distribute your unit tests to your customers.

In reality we would have more directories such as `distribution` and `documentation`. We would also have more directories under `main` for packages such as `com.company.util`.

Since directory structures frequently change, it is important to have global string constants in our `build.xml` for these changes.

Figure 3. Project layout diagram



Sample Ant build configuration file
Next, we create a configuration file. Listing 4 displays a sample Ant build file. The key point in the build file is the target named runtests. This target forks and runs an external program. The external program is the `junit.textui.TestRunner` that was installed earlier. We specify what test suite to run with the statement `test.com.company.AllJUnitTests`.

Listing 4. Sample build file

```
    <property name="app.name"   value="sample" />
    <property name="build.dir"  value="build/classes" />

    <target name="JUNIT">
        <available property="junit.present" classname="junit.framework.TestCase" />
    </target>

    <target name="compile" depends="JUNIT">
        <mkdir dir="${build.dir}"/>
        <javac srcdir="src/main/" destdir="${build.dir}" >
            <include name="**/*.java"/>
        </javac>
    </target>

    <target name="jar" depends="compile">
        <mkdir dir="build/lib"/>
        <jar jarfile="build/lib/${app.name}.jar"
             basedir="${build.dir}" includes="com/**"/>
    </target>

    <target name="compiletests" depends="jar">
        <mkdir dir="build/testcases"/>
        <javac srcdir="src/test" destdir="build/testcases">
            <classpath>
                <pathelement location="build/lib/${app.name}.jar" />
                <pathelement path="" />
            </classpath>
            <include name="**/*.java"/>
        </javac>
    </target>

    <target name="runtests" depends="compiletests" if="junit.present">
        <java fork="yes" classname="junit.textui.TestRunner"
            taskname="junit" failonerror="true">
            <arg value="test.com.company.AllJUnitTests"/>
            <classpath>
                <pathelement location="build/lib/${app.name}.jar" />
                <pathelement location="build/testcases" />
                <pathelement path="" />
                <pathelement path="${java.class.path}" />
            </classpath>
        </java>
    </target>
</project>
```

Run the sample Ant build

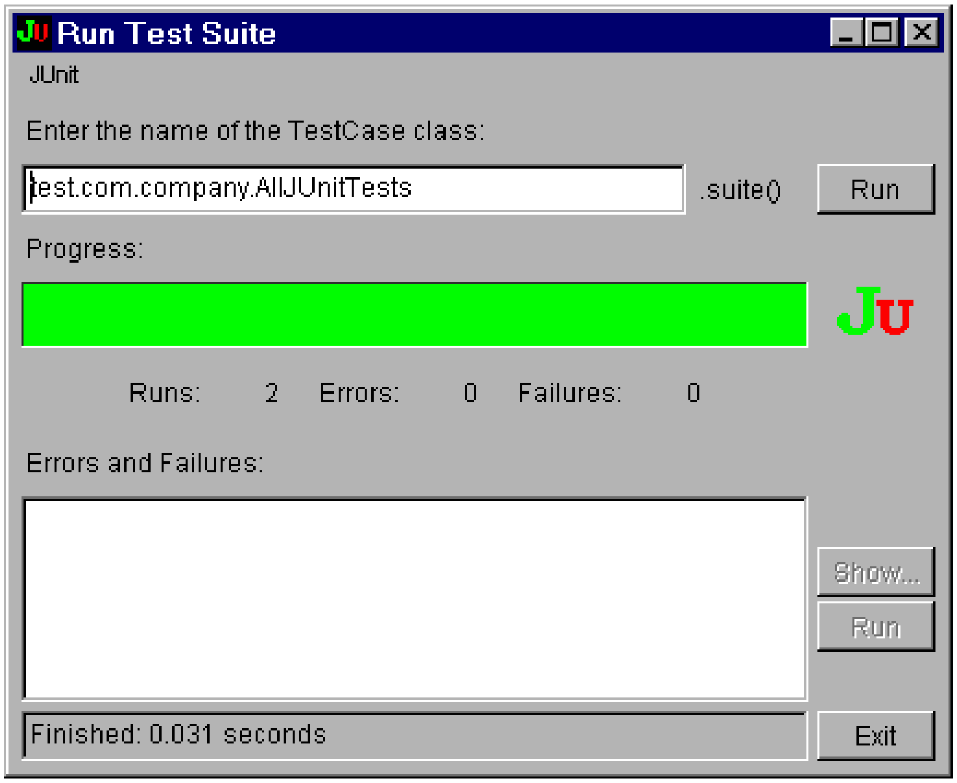
The next step in the development process is to run the build that will create and test the HelloWorld class. Listing 5 displays the result of the build. Each target section is displayed. The cool part is the runtests output statement; it tells us that our entire test suite ran successfully.

I could have displayed the JUnit GUI as shown in Figures 4 and 5. All I would have to do is modify the runtest target from `junit.textui.TestRunner` to `junit.ui.TestRunner`. When you use the GUI portion of JUnit, you have to select the exit button to continue with the build process. Using the JUnit GUI makes the package build more difficult to integrate with a larger build process. Also, the text output is more consistent with the build process and can be piped to a text file for a master build record. This is nice for those nightly builds.

Listing 5. Sample build output

```
E:\projects\sample>ant runtests
Searching for build.xml ...
Buildfile: E:\projects\sample\build.xml

JUNIT:

compile:
    [mkdir] Created dir: E:\projects\sample\build\classes
    [javac] Compiling 1 source file to E:\projects\sample\build\classes

jar:
    [mkdir] Created dir: E:\projects\sample\build\lib
      [jar] Building jar: E:\projects\sample\build\lib\sample.jar

compiletests:
    [mkdir] Created dir: E:\projects\sample\build\testcases
    [javac] Compiling 3 source files to E:\projects\sample\build\testcases

runtests:
    [junit] ..
    [junit] Time: 0.031
    [junit]
    [junit] OK (2 tests)
    [junit]

BUILD SUCCESSFUL

Total time: 1 second
```

Figure 4. JUnit GUI test success



Enter the name of the TestCase class: test.com.company.AllJUnitTests .suite()

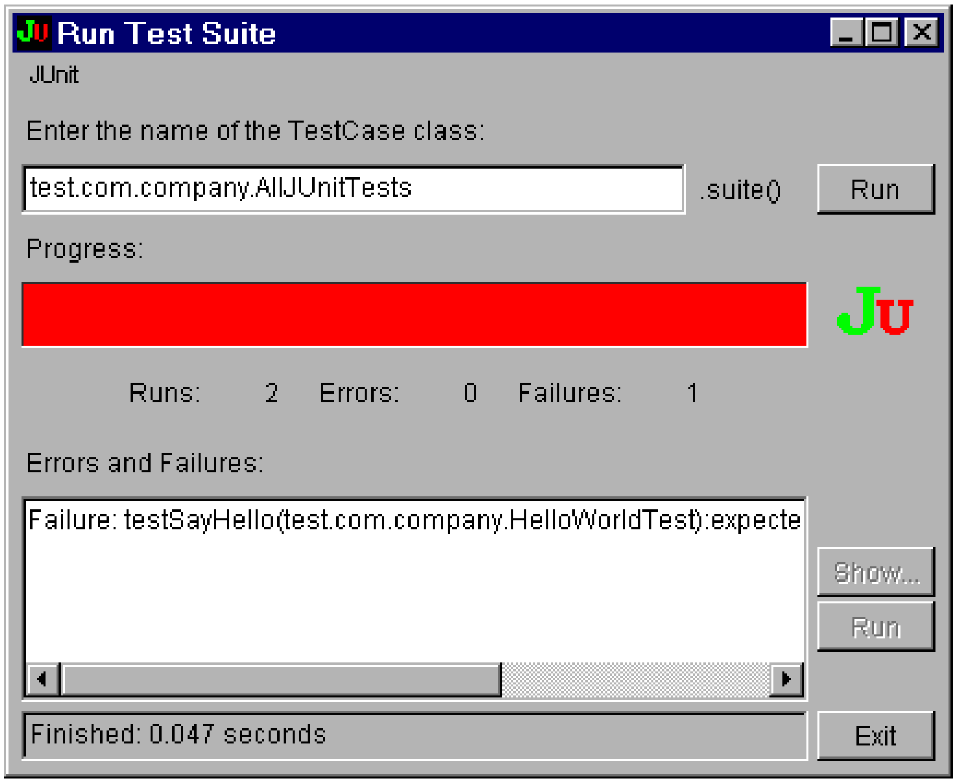
Progress:

Runs: 2 Errors: 0 Failures: 0

Errors and Failures:

Finished: 0.031 seconds

Figure 5. JUnit GUI test failure



test.com.company.AllJUnitTests

Runs: 2 Errors: 0 Failures: 1

Errors and Failures:

Failure: testSayHello(test.com.company.HelloWorldTest):expecte

Finished: 0.047 seconds

See how the test works
Let's break something and watch what happens. It’s late at night and we decide to make our "Hello World" a static string. During the change we *fat finger* the change and make the "o" a "0" as shown in Listing 6.

Listing 6. Hello world class change

```
package com.company;

public class HelloWorld {
    private final static String HELLO_WORLD = "Hell0 World";

    public String sayHello() {
        return HELLO_WORLD;
    }
}
```

As we build our package, we see the error of our ways. Listing 7 shows an error in the runtest. It displays the test class and test method that failed and why it failed. We go back to our code, fix the bug, and away we go.

Listing 7. Sample build error

```
E:\projects\sample>ant runtests
Searching for build.xml ...
Buildfile: E:\projects\sample\build.xml

JUNIT:

compile:

jar:

compiletests:

runtests:
    [junit] ..F
    [junit] Time: 0
    [junit]
    [junit] FAILURES!!!
    [junit] Test Results:
    [junit] Run: 2 Failures: 1 Errors: 0
    [junit] There was 1 failure:
    [junit] 1) testSayHello(test.com.company.HelloWorldTest) "expected:<Hello
               World> but was:<Hell0 World>"
    [junit]

BUILD FAILED

E:\projects\sample\build.xml:35: Java returned: -1

Total time: 0 seconds
```

Not totally painless
The new process is not totally painless. The steps you have to take to make unit testing part of development are:

1. Download and install JUnit.
2. Download and install Ant.
3. Create a separate structure for your build.
4. Implement a test class that is separate from the main class.
5. Learn the Ant build process.

But the benefits outweigh the pain. By making unit testing part of your development process, you can:

- Automate validation to catch change bugs
- Design your classes from an interface perspective
- Provide clear examples
- Avoid code clutter and class bloat in the release package

Achieve 24x7
Insuring quality in a product costs money, but the lack of quality costs more. How can you get the biggest bang for the buck, to insure product quality?

- Review your designs and code. Reviews cost about half of what testing alone can accomplish.
- Confirm module work by unit testing.
  Although testing has always been there, as development practices have evolved, unit testing has become part of the everyday development process.

In my 10 years of development, working for emageon.com is one of the highlights. At emageon.com, design review, code review, and unit testing are everyday events. The daily development habits produce a top-notch quality product. The software had zero downtime in its first year at the customer site, a true 24x7 product. Unit testing is like brushing your teeth: you don’t have to do it, but the quality of life is much better if you do.

Resources

- Download the sample code referenced in this article.
- Download Ant from the Apache Web site. For Ant documentation, FAQs, and downloads, see the Jakarta Project Ant home page.
- The JUnit home page provides additional testing examples, documentation, articles, and FAQs. You can download JUnit 3.2 from www.xprogramming.com.
- "Simple Smalltalk Testing" by Kent Beck discusses a simple testing strategy and a framework to support it.
- See comments on unit testing from some extreme developers.

- To learn about other useful development habits, go to the Extreme Programming Home page.

About the author
Malcolm G. Davis is president of his own consulting company in Birmingham, Alabama. He considers himself a Java Evangelist. When he is not preaching the virtues of Java, he spends time running and playing with his kids. You can reach Malcolm at malcolm@nuearth.com.

What do you think of this article?

○ Killer! (5) ○ Good stuff (4) ○ So-so; not bad (3) ○ Needs work (2) ○ Lame! (1)

Comments?

Submit feedback