11-2006

Unit Testing Considered Useful

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

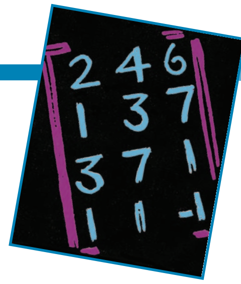
Konstantin Läufer
Loyola University Chicago, klaeufer@gmail.com

Benjamin Gonzalez

Recommended Citation

G. K. Thiruvathukal, K. Laufer and B. Gonzalez, "Unit Testing Considered Useful," in Computing in Science & Engineering, vol. 8, no. 6, pp. 76-87, Nov.-Dec. 2006. doi: 10.1109/MCSE.2006.124

Editors: George K. Thiruvathukal, gkt@cs.luc.edu
Konstantin Läufer, laufer@cs.luc.edu



UNIT TESTING CONSIDERED USEFUL

By George K. Thiruvathukal, Konstantin Läufer, and Benjamín González

TESTING IS AN IMPORTANT PART OF APPLICATION DEVELOPMENT. HARDWARE ENGINEERS, IN PARTICULAR, HAVE A LONG ESTABLISHED HISTORY OF TESTING FOR THE obvious reason that it's awfully hard to rebuild a microprocessor every time a bug pops up in the design stage—not to mention the enormous headaches such bugs generate on the software side. To that end, programmers use hardware design languages such as the VHSIC Hardware Description Language (VHDL) for field-programmable gate arrays (FPGAs) and very large-scale integration (VLSI).

VHDL's key advantage for system design is that it lets developers describe (model) and verify (simulate) system behavior before synthesis tools translate the design into real hardware (gates and wires). If only software worked the same way, the world would be a much better place. Instead, users of virtually any operating system are the direct casualties of the "unrecoverable application error" or the "unhandled exception." In such cases, a hexadecimal memory address offers the only words of comfort, possibly with a textual explanation as well, although the address sometimes makes more sense than the text.

Unfortunately, the scientific and engineering software community is sometimes slow to adopt new development ideas. This was especially true for object-oriented programming (OOP), which is now used in several projects, especially programming libraries. However, many computational efforts are still based on C or Fortran. Without passing a value judgment—both can deliver great performance—some aspects of programming in these languages confound the notion of testing. The lack of a proper exception model, for example, means that error codes (often a large list of them) are the mechanism for dealing with failure; this is okay for straightforward code, but it isn't the stuff of which good software engineering is accomplished.

In this installment of Scientific Programming, we'll discuss the role of testing in the software development process and examine ways to leverage automated unit testing in your projects.

Testing's Role in the Software Development Process

Understanding a bit of history behind the software development process can shed some light on why testing still isn't as prevalent today as it ought to be. In early software engineering approaches, the model placed great emphasis on a more or less sequential set of steps:

1. business requirements,
2. functional requirements,
3. detailed design,
4. coding/implementation,
5. testing, and
6. production (release).

(Incidentally, the IEEE has a recommended practice for software requirements specifications, which appears to have been updated as recently as 1998; http://ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=15571.)

The process of constructing software per these early models was notoriously labor-intensive. In many real software projects, teams got bogged down with the task of trying to construct as complete a set of business requirements as possible. Once they understood the business requirements, they had to map them to functional requirements. The team then prepared a detailed design document, often based on the technical expertise and technology preferences within the company or organization, followed by coding and testing. It might seem hard to believe that anyone would construct a meaningful project this way, but the industry is replete with complex software systems that still follow similar processes—even for software written in modern languages such as C# and Java.

Although we can't blame these models for the lack of adequate testing in most software applications, much of what professional programmers learned from them over the years might have conditioned them to think that testing comes at the end. Even in the most enlightened organizations, test-

 Copublished by the IEEE CS and the AIP 1521-9615/06/$20.00

Läufer's Lounge

myLife Beyond iLife?

In June 2002, when I got my first Apple (an original PowerBook Titanium running OS X), I had primarily been a Windows user at home and work for several years. I managed my digital life with Microsoft Outlook, which integrates personal information manager (PIM) functionality including email, address books, calendars, to-do lists, and notes. I didn't see it as a problem that I was tied to a particular desktop for everything but email because I spent most of my time at my desk at home anyway. Because I had used IMAP for a long time, I could already access my email from anywhere, and I used the unison file synchronizer to mirror my documents bidirectionally with the file server at the office. To switch my digital life to the PowerBook for a minimal transaction cost, all I had to do was migrate all the information from the Windows version to the OS X version of Outlook, and I was all set—well, for a while, at least.

Migration to iLife

Then I noticed how the integration of the OS X address book with the rest of the system, especially iChat and the built-in fax functionality, was really nice. And Apple Mail looked and felt much slicker than the corresponding modules in Outlook. I decided to jump into the iLife style all the way. As I remember, it was easy enough, and Apple provides lots of help.

Moving my IMAP email accounts was trivial, and Outlook exported all my contacts as vCards in a way my new address book could understand without problems. I also managed to migrate my Outlook calendar and to-do list information to iCal. Again, I was all set and things worked very smoothly, especially with the ability to carry the PowerBook around between home, work, and anywhere else. I even had all my photos and music with me all the time, which is great for traveling and visiting relatives.

Multiple Computers

Things changed in late 2004: I got an iMac G5 at work and set up a Gentoo Linux storage/media server with desktop functionality at home. I now needed a way to work productively on any of my three machines. I didn't consider a .Mac account for keeping them in sync because I didn't expect it to work with the Linux box, which had become my main machine. Besides, I was reluctant to pay a recurring fee for something that I

continued on p. 78

ing teams conducted a few black-box tests prior to production, but in the worst case, they left testing to the users, who were more than happy to report "constructive" feedback after spending large sums of money.

In the mid-1990s, the dot-com bubble created pockets of new thinking about the development life cycle. The Internet not only made it possible to exchange information readily and quickly, it also led to the notion of projects operating on Internet time—a concept that's even more prevalent today with many people working on projects 24/7/365 (that is, 24 hours a day, seven days a week, 365 days a year) in multiple time zones. This sea change required a rethinking of development processes in general because following a traditional life cycle could easily make even the most trivial of projects take years to complete.

In the late 1990s, extreme programming (XP) explored the seminal ideas for integrating requirements, coding, and testing. Early proponents understood all too well that testing and coding must go hand in hand to ensure that testing actually happens. The deeper consequence of XP is that integrating the two can lead to a better understanding of requirements and allow for the possibility of refining them on the fly. Clearly, it's a good idea to discover whether a subset of requirements is feasible before writing a complete and complex software system.

The Design Space

Before we dive into the technical aspects of testing, let's take a brief look at what we call the design space (www.faqs.org/faqs/software-eng/testing-faq/).

Granularity defines the part of the system we want to test. Typically, we distinguish among *unit testing*, which tests the smallest compilable component (a unit) in isolation by replacing its dependencies with stubs; *integration testing*, which applies to a complex component that comprises multiple atomic components; and *system testing*, which applies to the entire system. Note that if we view the whole system as a recursively composed unit, we can take advantage of unit-testing techniques at any level of granularity.

Transparency indicates the level of knowledge of a compo-

continued from p. 77
could recreate via a combination of standard services. But before I found a solution, I accepted a position in academic administration and became office-bound for a year and a half.

An Incomplete Migration to Linux

I once tried to migrate all my OS X PIM functionality to a Linux equivalent such as KDE's kontact (www.kontact.org) or Gnome's Evolution (www.gnome.org/projects/evolution), but I ran into a few glitches. The worst is that the OS X address book's export functionality to vCard and other formats is seriously broken (it has problems with foreign characters, annotations, and so on). I could never get the information out of my address book in a way that didn't require a lot of manual postprocessing, so I gave up. Migrating calendar items from iCal was comparatively simple, but korganizer (kontact's calendaring module) kept crashing when it tried to open calendar files from iCal.

Needing the PowerBook after All

Besides being quite incomplete, this state of affairs had other shortcomings. For example, The photos on the Linux system are organized in a way that's incompatible with iPhoto, so mirroring to the PowerBook doesn't work, but simply keeping the photos on the server and accessing them from the PowerBook results in a huge lag when moving from one photo to the next. Plus, what if I wanted to travel and take my photos along? I'd still need them on the PowerBook. I also found the user experience with Linux photo applications to be inferior to iPhoto, so I fell behind in keeping my collection organized.

Then came the day we had a big party in our condo's hospitality room. I hadn't played DJ for a while and wasn't keen on juggling lots of CDs, so I decided to use the PowerBook as my principal music source. Because I organized my music collection on the server, I had to migrate it back to the PowerBook for the party, but things worked out really great.

What I learned from these two situations is that I need the right tool for the job; otherwise, the job might not get done promptly or at all. For the highest convenience and best experience, I prefer to keep my photos and music collections organized on the PowerBook using iPhoto and iTunes and back them up to the server. Furthermore, unlike Microsoft or Linux, Apple has consistently delivered on a combination of ergonomics and aesthetics that leads to a superior overall user experience.

nent's implementation with which we design the test cases for that component. Typically, behavioral (black-box) testing assumes knowledge only of the component's specification or public interface; structural (white-box) testing assumes full knowledge of the component's implementation and might try to achieve test-quality metrics such as code coverage. In practice, test suites often fall between these two extremes, giving rise to the term *gray-box testing*, in which the behavioral test's design is informed by knowledge of some of the component's structural aspects—for example, its internal states.

Automation describes whether we do the testing manually or automatically. A key insight that emerged along with lightweight approaches such as XP is that testing is less likely to occur if it's tedious. Conversely, if testing is automatic and takes only the press of a button, it's much more likely to occur. Nevertheless, real users must conduct certain types of testing, such as user acceptance testing (UAT).

Unit Testing by Example

Because most of the craze for unit testing started with JUnit (www.junit.org), the Java-based unit-testing framework, we'll give examples here in Java. That said, JUnit's ideas are available in just about every other major programming language, including C/C++ (http://check.sourceforge.net), Python (native support as of Python 2.4), Ruby, and C#. Perhaps most important, we can use unit testing without having native support for OOP.

Let's start by looking at the built-in Java library support for the notion of an array-based list (known as `ArrayList`). An array list is more or less what you'd think it is—an expandable list structure that has an underlying array representation. The Java software development kit's (SDK's) documentation provides a paragraph that explains it concisely:

> Each `ArrayList` instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an `ArrayList`, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

So what would be involved in testing this class? The class itself provides several methods (do a Web search on "ArrayList" to see a sample), but we'll focus here on just a few of them for the purpose of writing unit tests:

- `boolean add(Object o)` appends the specified element to the end of this list.
- `void clear()` removes all the elements from this list.
- `Object get(int index)` returns the element at the specified position in this list.
- `Object remove(int index)` removes the element at the specified position in this list.
- `int size()` returns the number of elements in this list.

The Rest of the Migration

The remaining challenge was to restore the best possible PIM functionality without getting tied to a particular computer or needing a .Mac account.

The main showstopper continued to be the address book export. I tried different settings and various tools purported to help with this task, but nothing worked. Then I remembered Plaxo (www.plaxo.com), an online address book that updates itself when your contacts on the system change their information. The basic Plaxo service is free; it has toolbars for syncing with various PIMs, so I gave it a try—and it worked wonders! Now I have my hundreds of contacts on the Web with the option to sync with desktop clients. Upon recommendation, I switched to Gmail in June 2006, and I'm glad I did because of Gmail's superior organization, search, and spam control. However, until it offers a proper import feature, I'm simply keeping my old mail on our IMAP server at work.

George Thiruvathukal suggested that I try Google Calendar, and I was happy to find that it imported all my iCal files. I'm not currently using any centralized mechanism like the one in Outlook for taking brief notes. Instead, I keep notes on my Web site organized by content area and make them public in certain cases. The only thing left to tackle was a proper to-do list; I'm now using Toodledo (www.toodledo.com) because it comes closest to my needs. I like to categorize, prioritize, and date my to-do items, so I hope Google soon adds an adequate service to its portfolio—preferably with iCal import.

For writing longer documents, I was never much of a Microsoft Word fan because it forces you to work too much at the visual level. For high-quality research papers, the gold standard is still LaTeX, and for more than three years I've used LyX (www.lyx.org), a graphical LaTeX front end that follows the WYSIWYM paradigm (what you see is what you mean). For most other writing, especially of the collaborative type, I just started using Writely (www.writely.com), which works extremely well. In fact, I used it to write this sidebar. Writely is a prominent example of a new breed of slick Web 2.0 applications that make you feel like you're using a local desktop application, but it supports collaboration and keeps your documents on their servers. A recent *Red Herring* article (www.redherring.com/Article.aspx?a=18053) mentions 17 of these "MS Office killers," including complete office suites as well as single-purpose applications for presentations, spreadsheets, and so on. I plan to evaluate some of them very soon.

- `Object[] toArray()` returns an array containing all the elements in this list in the correct order.

Many more methods exist, but this selection captures the essence. As you'd expect, we can add or remove items from the list; we can even clear out the entire list by using a special (convenience) method. We can also examine list state via the `get(int index)`, `toArray()`, or `size()` methods.

The `ArrayList` class is a great example for unit testing for two reasons:

- It's easy to understand (virtually every programmer knows it).
- Despite its simplicity, some interactions can easily unveil bugs. A common mistake is when one method says something different from another—for example, if `toArray()` returns a different number of elements than `size()`.

So how do we test the `ArrayList` class? Our goal is to have a sufficient number of test cases with the hope that we test all the methods in one way or another.

```
import junit.framework.TestCase;

public class ArrayListTests extends TestCase {

  public void testAdd() {
    ArrayList<String> aList = new ArrayList<String>();
    boolean result = aList.add("A String");
    assertTrue(result);
    Object[] arrayItems = aList.toArray();
    assertEquals(1, arrayItems.length);
    assertEquals("A String", arrayItems[0]);
  }

  public void testClear() {
    // ...
  }

  /* and more */
}
```

Figure 1. Boilerplate of a JUnit test case. Our goal is to test all the methods in one way or another, so the first step is to create a subclass of `TestCase`.

The first step to writing JUnit test cases is to create a subclass of `TestCase`. Figure 1 shows a boilerplate. The figure also shows the test case's basic anatomy:

- Each test case is a method that begins with the prefix "test" and returns void (nothing). Newer versions of JUnit don't require this naming convention but do require some knowledge of metadata. We'll stick with the slightly older syntax here (it's still fully supported and works across different languages).
- Test-case creation requires us to derive the test class from the `TestCase` base class. The JUnit framework will examine only classes that extend `TestCase` for test methods.
- Test cases aren't guaranteed to be called in any particular order, so we assume that each test is completely independent.
- The test cases can call any code in the language as long as the appropriate library code has been imported (or `#included` for our C++ readers).

Let's take a close look at the `testAdd()` method. We'll focus just on its body:

```
public void testAdd() {
  ArrayList<String> aList =
    new ArrayList<String>();
  boolean result = aList.add("A String");
  assertTrue(result);
  Object[] arrayItems = aList.toArray();
  assertEquals(1, arrayItems.length);
  assertEquals("A String", arrayItems[0]);
}
```

We can construct a test case simply by thinking about what ought to happen. As we'll see in later examples, though, what ought to happen doesn't always imply success. For now, let's assume that we're talking about what it means to `add()` an element to an array list under normal circumstances:

- The simplest case is to add an item to an empty list.
- Per the Java documentation, when an item is added using `add()`, the result should also be true (meaning that the `ArrayList` collection increased by one—that is, it went from 0 to 1).
- One way of testing whether it was successful is to look at the actual array of objects, which should have a length of 1; the only item in the array (at index 0) should be the item we added.

Although this test case is simple, it shows what actually goes into it.

The key to writing effective test cases is to understand what should happen and then to make *assertions* along the way. An assertion (a construct that comes from formal logic systems) is a statement that must be true or else the entire method (list of statements) is false. In practical terms, however, a false assertion really means that something we expected didn't occur. The flaw could be in the test case, the code under test, or both (which is very rare).

The `testAdd()` case is just one way of testing the `add()` method. We can also test the `add()` method by using `size()` and `get(int index)`, which let us examine the number of items in the list and an item at a particular position, respectively:

```
public void testAddUsingSizeAndGet() {
  ArrayList<String> aList =
    new ArrayList<String>();
  int aListSize = aList.size();
  boolean result = aList.add("A String");
  assertTrue(result);
  assertEquals(aListSize + 1, aList.size());
  assertEquals("A String", aList.get(0));
}
```

Now we see the art involved in testing. Here, we're testing the `size()` method results by looking at the `size()` before and after an item is added to the list. You might be tempted to check that the `size() == 1` instead of `size() == "the old size" + 1`. Although both are fine to a certain extent, the example shown here doesn't fully depend on the `size()` method's correctness. All we truly know about `add()` is that the list size should increase by one from its old size. In addition, testing that the `size()` result increases by the number of items added also allows us the possibility of creating a stress test in which we add a huge number of items to the list. Then we can just check every so often to see that the list has the correct size.

So far, our basic examples have focused on testing the expected, but equally important is the need to test the unexpected. Let's consider the following test to `get()` an item from an empty list:

```
public void testGetFromEmptyList() {
  ArrayList<String> aList =
    new ArrayList<String>();
  try {
    aList.get(0);
    fail("an expected exception did not" +
      " occur");
```

Can I Do this Stuff in C/C++?

As we mention in the main text, unit testing is available for virtually any language, including C/C++, which has multiple frameworks that can fit almost every need it has (see www.opensourcetesting.org/unit_c.php). In our specific case, we use CppUnit (http://cppunit.sourceforge.net/), an open-source framework designed for unit testing in C++. We can construct a brief example using the STL vector class, which is the closest we can get to a Java `ArrayList` without doing our own implementation.

The program's structure is very similar to the JUnit equivalent, as we can see here:

```
class StringVectorTest
: public CPPUNIT_NS::TestFixture {

private:
  vector<string> *aList;

public:
  void setUp()
  {
    aList = new vector<string>;
  }
  void tearDown()
  {
    aList->clear();
    delete aList;
  }
  void testAdd()
  {
    string testString = "A String";
    aList->push_back(testString);
    CPPUNIT_ASSERT(aList->size() == 1);
    CPPUNIT_ASSERT((*aList)[0] == "A String");
  }
}
```

However, because C++ doesn't have reflection (or introspection) capabilities like Java, setting up the test suite takes a little extra work. Specifically, we need to use CppUnit's `TestSuite` class to keep track of which methods to call

continued on p. 82

We've constructed an empty list here: when the test method attempts to access an item from the list, it can't possibly succeed, thus proving the power of unit testing and working with a language that has true exception handling. If the exception isn't generated, therefore leaving the catch block unexecuted, the code will continue through to the `fail()`, which is another type of assertion method that guarantees *false*. (In case you're curious, no `success()` method exists because success would have the effect of an NOP [no operation] when it comes to testing; it would be an assertion that's always *true*, meaning that test-case processing would continue.)

It might not be obvious yet, but writing good test cases requires imagination. As you start writing test cases, you start learning more about how the class might be used because you use it yourself. We believe testing isn't just a part of understanding requirements and creating reliable software—it's also a key ingredient in creating usable software.

How to Run Test Cases

The JUnit framework provides a `TestSuite` class for running multiple test cases together. For each `TestCase`, you can even choose which test methods should be included in the suite. The framework also provides `TestRunners` for textual and graphical user interfaces (see the "Can I Do this Stuff in C/C++?" sidebar).

If you use Eclipse, you won't have to worry about `TestSuites` or `TestRunners` because this functionality is built into its JUnit support. To run all JUnit `TestCases` within a package or file, simply right-click on the package or file in the Package Explorer view and choose Run As > JUnit Test. You can run all test cases in the project this way.

Eclipse shows test results in the JUnit view as a tree you can drill down into. For each test method, you see three possible outcomes: success (indicated by a green checkmark), failure (indicated by a black x), or error (indicated by a red x). The difference between the two is that failure indicates a failed JUnit assertion: the test ran but didn't pass, whereas error indicates that some other exception occurred to preclude the test from running properly. Figure 2, for example, shows a `NullPointerException` in the `testMultiplyBy` method.

An Extended Example: Dimensional Analysis

Array lists are interesting, but they aren't exactly a real and meaningful application to *CiSE* readers. Recently, we've been thinking about modern programming language design, which has made significant progress in the past few decades by introducing higher levels of abstraction for concepts (such as OOP) and new refinements (such as aspect-oriented programming [AOP]). However, something's eerily unsettling about the way they do calculations—particularly how they handle scalar data. Within recent memory, scientific "computing" was a victim of miscommunication, as described in this Wikipedia entry for "Exploration of Mars" (http://en.wikipedia.org/wiki/Exploration_of_Mars):

> Following the success of Global Surveyor and Pathfinder, another spate of failures occurred in 1998 and 1999, with the

continued from p. 81

when we run that suite. Each test also requires a test name string along with a function pointer to the actual method. Finally, if we want to run our tests from the command line, we must add the suite to a `TestRunner` object and invoke its run method:

```
class StringVectorTest
: public CPPUNIT_NS::TestFixture {
public:
  static CppUnit::Test *suite()
  {
    CppUnit::TestSuite *suite =
      new CppUnit::TestSuite(
        "StringVectorTest" );
    suite->addTest(
      new CppUnit::TestCaller
        <StringVectorTest>( "testAdd",
          &StringVectorTest::testAdd ) );
    suite->addTest(
      new CppUnit::TestCaller
        <StringVectorTest>( "testEmptyList",
          &StringVectorTest::testEmptyList ) );
    suite->addTest(
      new CppUnit::TestCaller
        <StringVectorTest>( "testErase",
          &StringVectorTest::testErase ) );
    suite->addTest(
      new CppUnit::TestCaller
        <StringVectorTest>( "testClear",
          &StringVectorTest::testClear ) );
    return suite;
  }
}

int main( int argc, char **argv)
{
  CppUnit::TextUi::TestRunner runner;
  runner.addTest( StringVectorTest::suite() );
  runner.run();
  return 0;
}
```

As you can see, although it takes some extra work, you can set up unit tests for your C++ application in a very straightforward manner.

Japanese Nozomi orbiter and NASA's Mars Climate Orbiter, Mars Polar Lander, and Deep Space 2 penetrators all suffering various fatal errors. Mars Climate Orbiter is infamous for Lockheed Martin engineers' mixing up the usage of imperial units with metric units, causing the orbiter to burn up while entering Mars' atmosphere.

It's a bit strange that this could ever happen in the sciences, but this isn't an isolated incident (think, too, of how often it goes unreported). The odd part is that scientists pioneered units of measurement and dimensional analysis as a technique. Why doesn't code involving scalar and array mathematics universally carry units of measurement and allow for automatic dimensional analysis?

We decided to tackle this question by showing that the idea (in its essential form) is achievable. Although we wanted to teach testing ideas with something meaningful from *CiSE*, we ended up with the beginnings of a dimension-aware calculator, which in turn led us to an even better example of how testing applies at multiple levels. The core idea of dimensional analysis is that we maintain an expression in reduced product form at all times. For example, we can rewrite meters, meters/second, and kg meters/second in product form as `meters` (already in the right form), `meters * seconds^-1`, and `kg * meters * seconds^-1`.

We start our example code with the `Dimension` interface for building unit expressions:

```
public interface Dimension {
  void multiply(String unit, int power);
  void multiply(String unit);
  void divide(String unit, int power);
  void divide(String unit);
  void multiply(Dimension another);
  void divide(Dimension another);
  int getLength();
  boolean hasUnit(String unit);
  int getPower(String unit);
  boolean hasUnits();
  Collection<UnitPower> getUnits();
}
```

The heart of manipulating unit expressions via dimensional analysis is symbolic manipulation. The nice thing about interfaces (which are intentionally free of implementation details) is that they can capture the essence of how something might be used (in this case, a given dimension); Figure 3 shows the code for the class `OrderedDimension`, which implements the interface. At this point, you might want to download the code from our Web site (http://snapshots.cs.luc.edu/etl/). We can now build up an expression for kg meters per second as follows:

```
Dimension d=new OrderedDimension();
d.multiply("kg");
d.multiply("meters");
d.divide("seconds");
```

Now we have a symbolic expression `kg meters/seconds`.

We call it an "ordered" dimension because the goal is to keep the units in the order in which `multiply()` and `divide()` calls are made, subject to term cancellations (which happen automatically, on the fly). For example, if an expression is built for `kg meters/second`, we really don't want it rewritten as `seconds^-1 kg meters`. Although it's still correct, we erred on the side of keeping the terms in the user-specified order, which we believe is more of a usability issue than a design issue.

Our basic implementation strategy is to keep an `ArrayList` of `UnitPower` objects, which is a generic collection. (As of Java 1.5, we can write all Java collections as collections of some type, which means we don't need to work with the `Object` class unless we really want to.) The `UnitPower` class is simply a wrapper for keeping a unit and its exponent together. It has several `set*()` and `get*()` methods that we can use to set or get the unit of measurement and its exponent. Let's look at the `multiply()` method, which has most of this class's guts:

```
public void multiply(String unit,
    int power) {
  int unitPos = findUnit(unit);
  if (unitPos < 0) {
    if (power != 0)
      unitExpr.add(
        new UnitPower(unit, power));
  } else {
    UnitPower unitFound =
      unitExpr.get(unitPos);
    unitFound.setPower(
      unitFound.getPower() + power);
    if (unitFound.getPower() == 0)
      unitExpr.remove(unitPos);
  }
}
```

Quite a bit of work is involved in multiplying a new term into the unit expression:

- We must see whether we can actually find the unit of measurement elsewhere in the expression.
- If we can't find the unit, we add the `(unit, power)` to the current expression, if `power != 0`. In dimensional analysis, terms with a 0 power vanish right away.
- If we find the unit, we add the powers of the old and new to form a new `(unit, power)` pair. As in the previous bulleted item, if the sum of the powers adds to 0, the unit vanishes. In this case, we must `remove()` the unit.

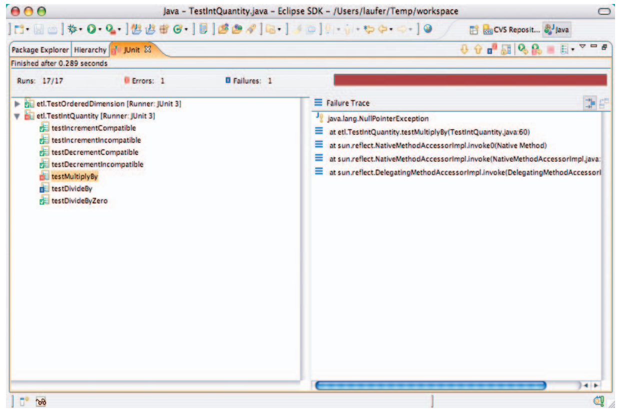


Figure 2. Test results in Eclipse JUnit view. Three possible outcomes exist for each test method: success, failure, or error.

Clearly, this short piece of code demonstrates testing's potential. We can see, for example, that several things can go wrong:

- When multiplying by a term already in the expression, we should combine the new term with an existing one and adjust its power.
- When a term vanishes, we should reduce the expression's length (by 1).
- Adding a term that has a power of 0 shouldn't affect the unit expression.

Other `multiply()` and `divide()` methods exist, but they're special cases of this particular `multiply()` method. In all cases, the work of actually doing the `multiply()` or `divide()` is delegated—for example, `multiply(String unit)` calls `multiply(unit, 1)` to do its work; `divide(String unit, int power)` calls `multiply(unit, -power)` to do its work, and so on. For completeness, we can also `multiply()` or `divide()` by another `Dimension` instance, but these, too, are delegated to the `multiply(String unit, int power)` method.

Figure 4 shows a few test cases. Again, for brevity's sake, we consider only a few here and leave the rest to the full version of these test cases for self-study.

These test cases are all independent of each other but increase in complexity from the top of the figure down. Table 1 summarizes by listing the tests (without the "test" prefix in the name) along with an explanation of the general strategy and expectations for each. Many tests exist besides those described in Table 1—in fact, in terms of code size, much more testing code exists than actual implementation code.

```java
public class OrderedDimension implements
   Dimension {

  private ArrayList<UnitPower> unitExpr;

  public OrderedDimension() {
    unitExpr = new ArrayList<UnitPower>();
  }

  private int findUnit(String name) {
    int pos = 0;
    for (UnitPower item : unitExpr) {
      if (item.getName() == name)
        return pos;
      pos++;
    }
    return -1;
  }

  public void multiply(String unit,
     int power) {
    int unitPos = findUnit(unit);
    if (unitPos < 0) {
      if (power != 0)
        unitExpr.add(
          new UnitPower(unit, power));
    } else {
      UnitPower unitFound =
        unitExpr.get(unitPos);
      unitFound.setPower(
        unitFound.getPower() + power);
      if (unitFound.getPower() == 0)
        unitExpr.remove(unitPos);
    }
  }

  public void multiply(String unit) {
    multiply(unit, 1);
  }

  public void divide(String unit, int power) {
    multiply(unit, -power);
  }

  public void divide(String unit) {
    multiply(unit, -1);
  }
  public void multiply(Dimension another) {
    for (UnitPower term : another.getUnits()) {
      multiply(term.getName(), term.getPower());
    }
  }

  public void divide(Dimension another) {
    for (UnitPower term : another.getUnits()) {
      divide(term.getName(), term.getPower());
    }
  }

  public int getLength() {
    return unitExpr.size();
  }

  public boolean hasUnit(String unit) {
    return findUnit(unit) >= 0;
  }

  public int getPower(String unit) {
    int unitPos = findUnit(unit);
    if (unitPos < 0)
      return 0;
    UnitPower unitPower = unitExpr.get(unitPos);
    return unitPower.getPower();
  }

  public boolean hasUnits() {
    return unitExpr.size() > 0;
  }

  public Collection<UnitPower> getUnits() {
    return Collections.
      unmodifiableCollection(unitExpr);
  }

  /* Some details omitted for conciseness, such as
  the toString() method to dump the internal
  representation */
}
```

Figure 3. The `OrderedDimension` class. It implements the `Dimension` interface.

Testing at the GUI Level

The JUnit approach handles successive levels of integration testing as long as the test methods interact with the components under test only through method invocation. This is no longer the case for certain situations, such as system testing a GUI or Web-based application. (Luckily, several JUnit extensions can handle these cases; in a future issue, we'll explore how to test Web applications at various architectural levels.)

Let's get back to the GUI level by using an extension of JUnit called jfcUnit (http://jfcunit.sourceforge.net). Our example is a dimensional calculator based on the dimensional

```java
public class TestOrderedDimension extends
   TestCase {

  public void testNoUnits() {
    Dimension u = new OrderedDimension();
    assertFalse(u.hasUnits());
  }

  public void testMisingUnit() {
    Dimension u = new OrderedDimension();
    assertFalse(u.hasUnit("meters"));
    assertEquals(0, u.getPower("meters"));
  }

  public void testBasicMultiply() {
    Dimension u = new OrderedDimension();
    u.multiply("meters");
    assertEquals(1, u.getLength());
    assertTrue(u.hasUnit("meters"));
    assertEquals(1, u.getPower("meters"));
  }

  // ...

  public void testMultiplyDivide() {
    Dimension u = new OrderedDimension();
    u.multiply("meters");
    u.divide("seconds");
    assertEquals(2, u.getLength());
    assertTrue(u.hasUnit("seconds"));
    assertTrue(u.hasUnit("meters"));
    assertEquals(-1, u.getPower("seconds"));
    assertEquals(1, u.getPower("meters"));
  }

  // ...

  public void testMultiplyUnitsTerm
     Cancellation() {
    Dimension u = new OrderedDimension();
    Dimension v = new OrderedDimension();
    // create meters/seconds
    u.multiply("meters");
    u.divide("seconds");
    // create seconds
    v.multiply("seconds");
    u.multiply(v);
    // expect meters only in the result
    assertEquals(1, u.getLength());
    assertTrue(u.hasUnit("meters"));
    assertEquals(1, u.getPower("meters"));
  }
```

Figure 4. Test cases. These test cases independently test the correctness of different capabilities in the `OrderedDimension` class.

Table 1. Summary of test cases shown in Figure 4.

Test case	Description	Testing strategy
`NoUnits`	Has no units whatsoever	No terms are added via `multiply()` or other calls. We check whether the expression has any terms by calling `hasUnits()`, which should be false. JUnit provides `assertFalse()` so we needn't negate what we're trying to test.
`MissingUnit`	References a unit not in the unit expression	We start with much the same code as `NoUnits` but check for a term that we know isn't part of the expression. Although the expression doesn't have the unit "meters," we should still ask for unit power because any unit taken to the 0th power is 1.
`BasicMultiply`	Multiplies a single unit into an existing unit expression	This test ensures that we can add at least one term to the expression. Here, we're simply forming an expression for "meters" whose length should be 1; the expression should have the unit "meters" in it (regardless of power); and the power itself should be 1.
`MultiplyDivide`	Mixes `multiply()` and `divide()` calls	We carefully crafted this test to ensure that the `divide()` call results in a unit with a negative exponent `(meters / seconds == meters^1 * seconds^-1)`.
`MultiplyDivide UnitsTerm Cancellation`	Involves `Dimension` instances, in which each contains its own unit expression	Testing builds confidence; recognizing that the core machinery is working correctly, we thus exercise the core `multiply()` and `divide()` methods and have two `Dimension` instances interact via a `multiply()`, resulting in some unit terms dropping out of the result.

```
public class IntQuantity {

  private int value;
  private Dimension units;

  // constructors and accessors omitted for brevity

  public void increment(IntQuantity another) throws DimensionAnalysisException {
    if (units.equals(another.units))
      value += another.value;
    else
      throw new DimensionAnalysisException(units, another.units);
  }

  public void decrement(IntQuantity another) throws DimensionAnalysisException { /* ... */ }

  public void multiplyBy(IntQuantity another) {
    units.multiply(another.units);
    value *= another.value;
  }

  public void divideBy(IntQuantity another) { /* ... */ }
}
```

Figure 5. `IntQuantity` **class. This class ties a value and a unit of measurement together to a dimensional quantity.**

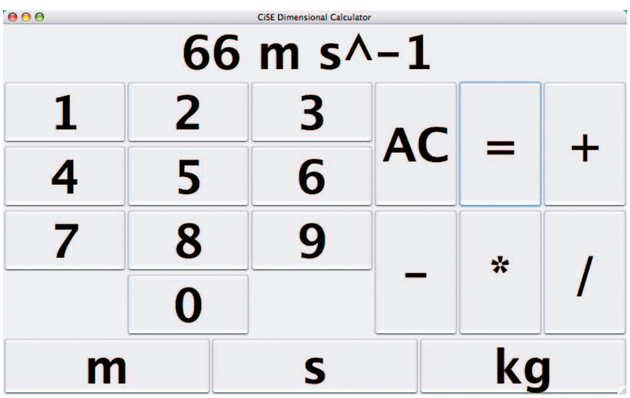


Figure 6. Simple *CiSE* calculator. This screenshot shows the dimensional quantity of 66 m/sec.

analysis classes we previously discussed. First, we need a simple class, `IntQuantity`, to tie a value and a unit of measurement together to a dimensional quantity. To keep things simple, we'll limit the discussion to integer values (see Figure 5).

Next we'll build a simple dimensional calculator that adds the ability to attach units of measurement to quantities and consider them in its calculations. Figure 6 shows the dimensional quantity of 66 m/sec entered by pressing the following keys: 6, 6, m, /, s, and =. As with other infix calculators, ours can keep track of a left operand, an operator, and a right operand, and then compute the result when we press another operator or the equal sign. Both operands and the result are instances of `IntQuantity`.

Using jfcUnit, we can write JUnit-style test cases that interact with the calculator at the GUI level just the way a human user would—by pressing buttons and reading the display. Figure 7 shows a couple of auxiliary methods on top of jfcUnit. Below these methods, you can see actual test cases for a dimensionless addition, an addition of square meters, an incompatible addition (failure is expected—the calculator shows Dim Err), and a division.

If you were in doubt, we hope this article has convinced you of the virtues of testing. Even if you don't plan to do more testing on your own code, consider asking someone to write test codes for you. Writing test cases is a great way to learn how code works and whether it does what it's supposed to do. Be careful, though—testing can be addictive. You might find yourself wanting to write up test cases before actually doing the implementation. You might also find that you write much more test code than library or application code. Perhaps that's why JUnit's authors speak of programmers as being "test infected"! We believe testing can make the world a much better place—especially the

```java
/** Reads the text displayed by the calculator. */
protected String getDisplayText() {
  JLabel display = (JLabel)
    new ComponentFinder(JLabel.class).find();
  return display.getText().trim();
}

/** Presses the button with the given label. */
protected void clickButton(String label) {
  JButton b = (JButton)
    new AbstractButtonFinder(label).find();
  getHelper().enterClickAndLeave(
    new MouseEventData(this, b));
}

/** Presses all buttons with the given labels. */
protected void clickButtons(String... labels) {
  for (String label : labels) {
    clickButton(label);
  }
}

public void testButtonPlus() {
  assertEquals("0", getDisplayText());
  clickButtons("3", "3", "\\+");
  clickButtons("6", "6", "=");
  assertEquals("99", getDisplayText());
}

public void testDimensionAdd() {
  assertEquals("0", getDisplayText());
  clickButtons("3", "3", "m", "m");
  assertEquals("33 m^2", getDisplayText());
  clickButton("\\+");
  clickButtons("6", "m", "6", "m");
  assertEquals("66 m^2", getDisplayText());
  clickButton("=");
  assertEquals("99 m^2", getDisplayText());
}

public void testDimensionAddFailed() {
  assertEquals("0", getDisplayText());
  clickButtons("3", "3", "m", "m");
  assertEquals("33 m^2", getDisplayText());
  clickButton("\\+");
  clickButtons("6", "m", "6", "m", "m");
  assertEquals("66 m^3", getDisplayText());
  clickButton("=");
  assertEquals("Dim Err", getDisplayText());
}

public void testDimensionDiv() {
  assertEquals("0", getDisplayText());
  clickButtons("1", "3", "2", "m", "m", "s");
  assertEquals("132 m^2 s", getDisplayText());
  clickButton("/");
  clickButtons("1", "m", "1", "s", "kg", "kg");
  assertEquals("11 m s kg^2", getDisplayText());
  clickButton("=");
  assertEquals("12 m kg^-2", getDisplayText());
}
```

Figure 7. Testing a dimensional calculator with jfcUnit. The three convenience methods at the top are for reading the calculator's display and pressing one or more buttons. The remaining test methods use convenience methods to test dimensionless addition, an addition of square meters, an incompatible addition (failure is expected—the calculator shows Dim Err), and a division. The test methods use assertions to make sure the calculator produces the expected results.

world of scientific and engineering computing, where we go out of our way to be precise but still make mistakes just like everyone else.

George K. Thiruvathukal is an associate professor of computer science at Loyola University Chicago. His research interests include programming languages, operating systems, distributed systems, architecture and design, computing history, and enhancing science and computing education with emerging technologies. Thiruvathukal has a PhD from the Illinois Institute of Technology. He is a member of the ACM and the IEEE Computer Society. Contact him at gkt@cs.luc.edu or http://people.cs.luc.edu/gkt.

Konstantin Läufer is a professor of computer science at Loyola University Chicago. His research interests include programming languages, software architecture and frameworks, concurrent and distributed systems, mobile and embedded computing, human–computer interaction, and educational technology. Läufer has a PhD in computer science from the Courant Institute at New York University. He is a member of the ACM. Contact him via http://people.cs.luc.edu/laufer.

Benjamín González is a software developer intern at Hostway Corporation. He also works as a research assistant at Loyola University Chicago. His research interests include distributed systems, artificial intelligence, software architecture, operating systems, and Web development. González has an MS in computer science from Loyola University Chicago. Contact him at bgonzalez@cs.luc.edu