Next

Dr.Dobb's

May 2013

Testing: How to test to maximize quality

Best practices, new approaches, tools, and techniques. >>

ALSO INSIDE

Simplifying Contract Testing >>

The Relationship Between Testability and Good Design >>

Table of contents >>

UBM Tech

www.drdobbs.com

Previous Next

Dr.Dobb's

CONTENTS

May 2014

FEATURES

4 Simplifying Contract Testing
by Claude Warren

Contract tests help to prove code correctness. If every object interface is defined as an interface, and every interface has a contract test that covers all methods and their expected operation, and all objects have tests that mock the objects they call as per the interface definition, then running the entire suite of tests demonstrates that the interconnection between each object works and is correct.

12 The Relationship Between Testability and Good Design
by Michael Feathers

There seems to be an eerily consistent connection between testability at the unit level and good design. Almost uniformly, code that is hard to test has design problems. When you fix the design problems, it becomes easy to test.

15 From the Vault: Unit Testing in C: Tools and Conventions
By James W. Grenning

A unit test harness is a software package that allows a programmer to express how production code should behave. A unit test harness's job is to provide a common language

More on DrDobbs.com

The Corruption of Agile

What was intended as a set of personal practices has become a doctrine. And despite the mainstream adoption of Agile, the loss of its original intent has undermined its effectiveness.

http://www.drdobbs.com//240166698

Lambda Expressions in Java 8

The single most important change in Java 8 enables faster, clearer coding and opens the door to functional programming. Here's how it works.

http://www.drdobbs.com/240166764

LINQ-like List Manipulation in C++

Using the open-source Cpplinq project to get Language-Integrated Query capabilities in C++11

http://www.drdobbs.com/240166882

Use Frequent Branches to Simplify Code Reviews

If you end up working this way, you'll be using your version control not only to track what has been done but also how it was done.

http://www.drdobbs.com/240166895

Java SE 8 Beyond Lambdas: The Big Picture

In addition to Lambdas, Java 8 features changes to the way annotations are handled, array processing, VM size, garbage collection, and so much more.

http://www.drdobbs.com/240166881

Fitting static code analysis into your testing toolkit isn't as hard as you thought

Effective code analysis doesn't mean absolute paralysis

More and more development teams are standardizing on static code analysis tools to improve testing efficiency and build better code. Are they more trouble than they're worth? Here's the real story.

Q What do static code analysis tools do?

They find bugs so you don't have to. It's as simple as that. SCA boils down to three things: better code at check-in, less costly development cycles and shorter time-to-market. Why? Because you don't have to worry about finding these bugs.

Q How efficient are they?

There are two ways to think about this. First, it's easy to save time on finding the most common defects (null pointer use, buffer overflow, unreachable code, etc.) through automation. Second, finding the most complex problems is best solved by proven algorithms that understand inter-procedural dependencies and cover every execution path better than any person can.

Q What's a good example of SCA?

Consider a function that dereferences a pointer set by another function. Manual unit testing of either function in isolation may not reveal that the pointer being dereferenced could be NULL. Static code analysis, on the other hand, would find the problem. Going further, consider the same situation but having the two functions developed by two different teams. The chances of the NULL pointer dereference reaching the customer becomes higher if the test coverage isn't there. Again, SCA covers everything.

Capers Jones of Namcook Analytics found that, without tools and processes like static code analysis, developers are less than 50 percent efficient at finding bugs in their own software.

Q What about false positives?

Static analysis tools emphasize reducing false negatives rather than false positives, so that you get the most rigorous coverage possible. It's critical then, to allow easy tuning of the analysis to account for the unique rules of your project and include or exclude code (such as macros or conditionally compiled code) appropriately.

Q We're great programmers already, why waste time with SCA?

SCA follows a strict set of rules and never gets tired, allowing developers to focus on the right things. A busy developer may miss an issue or think that it'll never happen. How many times have you miscounted loop iterations or got lost when tracing conditional logic? Because static code analysis understands the entire state space of your software, it never gets lost or assumes a problem is too insignificant.

Q Development timelines are short, how do I fit SCA in?

The best SCA tools aren't standalone products, rather ones that allow you to work within your existing environments (such as your command line or IDE). Even better, they give you feedback at the earliest possible point: as you're typing code. This way, the tool fits you.

Q Overall, how do I get the best measure of security and reliability in my code?

Modern SCA pushes code quality onto the desktop, as code is written, well before it's checked in. But it's just one piece of the puzzle. Combine it with deep scanning tools for open source software and visual debugging for multi-CPU apps and you'll have a bulletproof analysis strategy for efficient and reliable testing.

Keep Connected

Challenge more misconceptions about static code analysis:

Read the Myths About Static Code Analysis white paper

www.klocwork.com

Simplifying Contract Testing

Contract testing — the testing of specified interfaces and actions promised in documentation — is crucial to program validation, but difficult to do on Java classes that extend multiple interfaces.

By Claude Warren

The Interface Segregation Principle (ISP), which is the "I" in the list of SOLID coding principles, states that a client should not rely on interfaces it does not use (http://is.gd/et4PS6). That is, large interfaces should be made small enough that clients actually will use them. ISP therefore implies that an application should have many small interfaces that are combined to create more-complex objects. Contract testing holds that the correctness of an application or component can be assured by producing unit tests, also called isolated object tests, that validate interface implementations to ensure that they do not violate the interface contract. There have been several articles discussing the advantage of this type of testing (see http://is.gd/sUF2cj and http://is.gd/4DQ3Pk); however, I can find none that addresses how to test the implementation of the contract when several interfaces are combined in a single object.

Contract tests check the contract that is defined by a Java interface and associated documentation. The interface defines the method signatures, while the documentation often expands upon that definition to specify the behavior the interface is expected to perform. For example, the Map interface defines `put()`, `get()`, and `remove()`. But the human-readable documentation tells the developer that if you `put()` an object, you must be able to `get()` it unless the `remove()` method has been called. That last statement defines two tests in the contract.

Another example can be found in the Apache Jena Graph interface (http://is.gd/KF6Lbq), which has an `add(Triple)` method. From the interface, it is obvious that this method adds a triple to the graph. What is not clear is that when a triple is added, the graph must report that addition to all the registered listeners. The Graph contract test validates that this action occurs.

A more extreme case is `java.io.Serializable`, where there are no methods to test but the documentation (http://is.gd/dvYM7O) tells us that all serializable objects must contain only serializable objects or implement three private methods with very specific signatures (only two methods prior to Java 1.4). In addition, all classes derived from `Serializable` classes are themselves serializable. See the `Serializable` javadoc for details. (An example contract test for the `Serializable` interface is provided in the examples for junit-contracts; see http://is.gd/XK8H2W.) The basic argument for the use of contract tests is that they can help prove code correctness. That is, if every object interface is defined as an interface, and every interface has a contract test that covers all methods and their expected operation, and all objects have tests that mock the objects they call as per the interface definition, then running the entire suite of tests demonstrates that the interconnection between each object works and is correct.

"The basic argument for the use of contract tests is that they can help prove code correctness."

If we know that `A` calls `B` properly, and `B` calls `C` properly, then we can infer by transitivity that `A` calls `C` properly. Thus we can, with some work, prove that the code is correct.

Contract tests will not discover misconfiguration. For example, if class `A` uses a map and expects a map that can accept `null` keys, but the configuration specifies a map implementation that does not accept `null` keys, the contract test will not detect the error. But then, this error is a configuration error caused by a missing requirement for the configuration. Contract tests will also not uncover misuse of methods or classes. In the simple case, if `A` calls a power function on `B` instead of an intended multiplication function, the contract test will not discover it. However, the other side of the testing equation — collaboration testing — should catch it.

The Problem

At its most basic level, contract testing says that if you have an interface `A` there should be a test `AT` that tests the contracts that the interface prescribes. For example, in the Apache Jena project, there is an interface Model (http://is.gd/csmsn6)that has a method `createResource()`. In addition to returning that resource, the implementation must assure that if the `getModel()` is called on the returned resource, the model that created the resource is returned. The `Model` contract test would perform that test.

Because `A` is an interface, `AT` must be able to test any instance of `A`; thus, it must be either be a class with an abstract method that gets the implementation of `A` under test or a concrete class with a setter that sets the implementation of `A` under test.

For the simple solution, assume `AImpl` is the concrete implementation of `A` under test, and `ATImpl` is the concrete implementation of

AT. ATImpl is a fairly simple class that extends AT and implements the abstract getter or setter. So far, things are clear. However, unlike classes, multiple interfaces may be implemented by a class or extended by another interface. This leads us to the case where multiple abstract tests must be combined to create a complete contract test.

Assume that interfaces A and B are defined and that interface C extends them. Each has abstract tests: AT, BT, and CT, respectively. As we have seen, AT and BT are simple and fairly straightforward to write. The problem is with CT. Because CT is an abstract class, it can only derive from one base class, not both AT and BT as is required for complete testing. In addition, in keeping with DRY principles (http://is.gd/Om7K7H), we don't want to reimplement AT or BT, particularly since a change in A or B would not necessarily be picked up by the embedded implementation of AT or BT. The problem becomes more complex when we consider that C may be an interface published as part of a library or utility project where integrators are expected to implement or extend that interface as represented by D in Figure 1. In this instance, DT should not have to be recoded when A or B or C change. The impact of changes to the interfaces higher up the tree should be limited to only the associated test class.

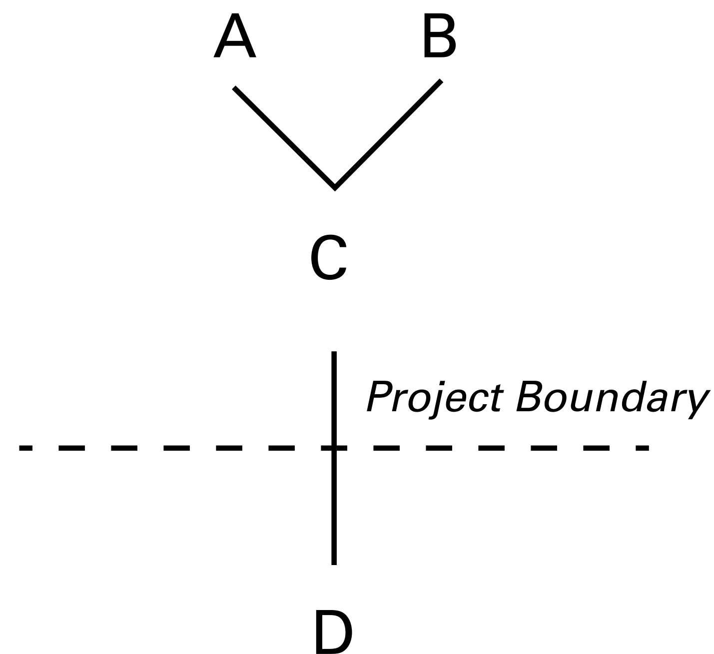


Figure 1. The relationship between multiple interfaces to a project class.

The problem then becomes how to implement a test suite where, given a starting class, multiple abstract tests can be discovered and included.

The Solution

The solution I propose is to create a framework that allows the standard JUnit test engine to discover and aggregate multiple test implementations into a single test suite. This framework introduces three new annotations and a class: @Contract, @Contract.Inject, @ContractImpl, and the class ContractSuite:

- @Contract is applied to a test class and specifies that the class is a contract test for another class. So, in our example, the class AT would have the annotation @Contract(A.class).
- The @Contract.Inject annotation denotes a getter and setter methods for the class under test.
- The @ContractImpl annotation is applied to a concrete test class implementation. This identifies the class under test. In our example, CTImpl would have the annotation @ContractImpl(CImpl.class).
- The @ContractTest annotation is replaces the standard JUnit @Test annotation to keep JUnit from running contract tests without proper configuration.

- The `ContractSuite` class is used with the standard JUnit `@RunWith` annotation to indicate that the class is defining a suite of contract tests to run. In our example, `CT–Impl` would have the annotation `@RunWith(ContractSuite.class)`.

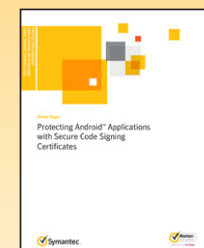
Examples for @Contract and ContractSuite

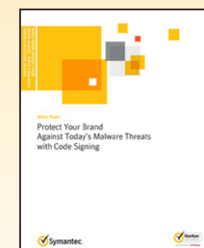
The example code presented here is refactored slightly to achieve a couple of goals. I want to define an abstract test `(CT)` that is the contract test for the interface `C`. I want a concrete implementation of that class `(CImplTest)` that executes just the tests defined in `CT`, and a contract test `CImpleContractTest` that executes the `CImplTest` tests as well as the `AT` and `BT` tests.

```
C, extending A and B
public interface C extends A, B {}

// a single abstract CT test -- tests just the C interface (not A or B)
@Contract(C.class)
public abstract class CT<T extends C> {
...
}

// implements the single CT test –
// Tests just the implementation of interface C
public class CImplTest extends CT<CImpl> {
...
}

// implements the suite of CT tests
// Tests interface C
// as well as the A and B interfaces
```

Symantec Resource Center

Protect Your Applications—and Reputation—with Symantec EV Code Signing
DOWNLOAD NOW


Protecting Android™ Applications with Secure Code Signing Certificates
DOWNLOAD NOW


Protect Your Brand Against Today's Malware Threats with Code Signing
DOWNLOAD NOW

Securing Your Software for the Mobile Application Market
DOWNLOAD NOW

Securing the Mobile App Market
DOWNLOAD NOW

Symantec.

```
@RunWith(ContractSuite.class)
@ContractImpl(CImpl.class)
public class CImpleContractTest{}
```

Assume that `A` and `B` have similar `AT` and `BT` classes associated with them via the `@Contract` annotation.

When JUnit runs the `CImplContractTest` test, it uses the `ContractSuite` class. That class discovers all the interfaces implemented by `CImpl` and then locates their associated tests. It uses the implementation of interface `C` specified by the `ContractImpl` annotation.

Introduction of the Producer

While the preceding discussion talks about injecting the class under test, the contract testing system actually uses an `IProducer` interface to define an object that can create new instances of the class under test and provides a mechanism to clean up after the test runs. The `@Contract.Inject` annotation is used on a method that returns the `IProducer` for the class under test. The Interface is defined as:

```
public interface IProducer<T> {
    public T newInstance();
    public void cleanUp();
 }
```

Expanding the Example

The `@ContractTest` annotation was added because several test runners attempt to instantiate and execute generic test classes, while others do not. (It appears that the JUnit classes could use some contract tests of their own.) Use of this annotation prevents these test runners from reporting false failures.

```
// define the contract test
@Contract(C.class)
public abstract class CT<T extends C> {

    private IProducer<T> producer;

    @Contract.Inject
    protected abstract IProducer<T>
           setProducer(IProducer<T> producer) {
       this.producer = producer;
    }

    protected final IProducer<T> getProducer() {
          return producer;
    }

    @After
    public final void cleanupCT()
       {
      producer.cleanUp();
    }

    @Test
    public void testCMethod() {}
}
```

```
// a test that only runs the contract test
// This is useful for debugging the contract test.
public class CImplTest extends CT<CImpl> {

    public CImplTest() {
        setProducer( new IProducer<CImpl>() {
            @Override
            public CImpl newInstance() {
                return new CImpl();
            }
            @Override
            public void cleanUp() {
             // does nothing
            }
        });
      }
}

// A contract suite.
// run as a contract suite
@RunWith(ContractSuite.class)
// is the suite for CImpl
@ContractImpl(CImpl.class)
public class CImplContractTest {
    // the producer for the CImpl
    private IProducer<CImpl> producer =
         IProducer<CImpl>() {
             @Override
             public CImpl newInstance() {
                 return new CImpl();
             }
             @Override
             public void cleanUp() {
                 // does nothing
             }
         };

    // method to inject our test instance into test classes
      @Contract.Inject
    public IProducer<CImpl> getProducer() {
        return producer;
    }
}
```

When JUnit runs the `CImplTest` test, only `testCMethod()` will run. However, when the `CImplContractTest` class is run, all of the `AT`, `BT` and `CT` tests will be run using the producer from `CTImplContract-Test` to create an instance of `CImpl`, which is the instance of `C`, `B`, or `A` depending on the test.

Real-Life Usage

As noted previously, Apache Jena has an interface called Graph. At this time, Apache Jena experimental `new` tests (http://is.gd/v7ecTw) are written using the junit-contracts package to test its framework and to provide packaged tests for integrators developing new instances of the Graph interface.

Graph is an extension point in the Jena project that enables integrators to quickly add additional storage types and strategies. It is expected

that integrators will create implementations of this class that function deep inside the Jena application architecture. It is imperative then that the implementors correctly implement the contract of the interface. There are several tests currently provided by the Apache Jena team that the implementer can run. However, there is not one clear test that tests the complete Graph contract. If the experimental tests are accepted by the Jena project, then they will provide a `GraphContract` test and an abstract `GraphProducer` that implements `IProducer<Graph>`. The reason for the abstract `GraphProducer` is that some graphs must be closed to properly shut them down, but the test interface may create multiple graphs. The `AbstractGraphProducer` tracks the graphs that are created and properly closes them during cleanup. It also has extension points to handle additional operations after the graph is closed. The `AbstracGraphProducer` code looks something like this:

```
public abstract class AbstractGraphProducer<T extends Graph>
implements
        IProducer<T> {

    /**
     * List of graphs opened in this test.
     */
    protected List<Graph> graphList =
         new ArrayList<Graph>();

    /**
     * The method to create a new graph.
     *
     * @return a newly constructed graph of
     * type under test.
     */
    abstract protected T createNewGraph();

    @Override
    final public T newInstance() {
        T retval = createNewGraph();
        graphList.add(retval);
        return retval;
    }

   /**
    * Method called after the graph is closed.
    * This allows the implementer to perform
    * extra cleanup activities, such as deleting
    * the file associated with a file-based graph.
    *
    * By default this does nothing.
    *
    *
    * @param g The graph that is closed
    */
    protected void afterClose(Graph g) {
    };
    @Override
    final public void cleanUp() {
        for (Graph g : graphList) {
            if (!g.isClosed()) {
                g.close();
```

```
            }
            afterClose(g);
        }
        graphList.clear();
    }
}
```

Implementers of the Graph interface only need to include the Jena test classes in their project test and code a simple contract running test class.

```
// A contract suite.

// run as a contract suite
@RunWith(ContractSuite.class)

// is the suite for MyGraphImpl
@ContractImpl(MyGraph.class)
public class MyGraphContractTest {
    // the producer for the MyGraph
       private IProducer<MyGraph> producer =
        new AbstractGraphProducer<MyGraph>() {
        @Override
        public MyGraph createNewGraph() {
            return new MyGraphImpl();
        }
    };

// method to inject our test instance into test classes
    @Contract.Inject
    public IProducer<MyGraph> getProducer() {
        return producer;
    }
}
```

Running this test will execute all of the contract tests that are applicable to the `MyGraph` class. If Apache Jena updates their tests, they will be executed. The `MyGraph` developer does not need to know that the `Graph` interface extends `GraphAdd`, nor is the developer concerned with the organization of the interfaces that `Graph` implements. The contract with the developer is that Apache Jena will provide contract tests that will test the Graph interface hierarchy. The advantage for the developer is that he/she does not have to respond to refactoring of the interfaces except where the signature of `Graph` methods change. Furthermore, since the contract tests are from the team that developed the interface, the developer can be fairly certain the tests are correct and that if the code passes them, it will function correctly in the environment.

Claude Warren is an IT Architect and Java Developer with more than 20 years of development experience. He is also a committer on the Apache Jena project. Code discussed in this article is available under the Apache 2.0 license at https://github.com/Claudenw/junit-contracts.

Comment

The Relationship Between Testability and Good Design

Problems writing good unit tests reveal shortcomings in design and coding.

By Michael Feathers @mfeathers

Recently, the profile of unit testing has risen dramatically. In the 1990s and early 2000s, it was a practice that seemed to languish. As is often the case when people adopt a new technology — back then, organizations were moving from structured design toward object-oriented design — they place all their focus on getting the essentials right and discard practices that don't seem to fit neatly into the new picture. And so they neglected unit testing.

But if we've learned anything over the past 10 years, it is that unit testing is an essential discipline. Tests help us better reason about our code, and they form a regression bedrock that makes refactoring and feature addition much easier. There is, however, a very subtle effect of unit testing that few people discuss. There seems to be an eerily consistent connection between testability at the unit level and good design. Almost uniformly, code that is hard to test has design problems. When you fix the design problems, it becomes easy to test.

Let's take a look at an example.

In Figure 1, we have a class that seems to have decent structure. There is one public method, `evaluate()`, which is the user's point of contact with the class. It delegates its work to a series of private methods. Overtly, there is nothing wrong with this — class `RuleEvaluator` does the job we've intended it to. Now, how should we write unit tests for this class? We should be able to instantiate it, call `evaluate()` with various arguments, and check the results. But suppose we wanted to test `getNextToken()`?

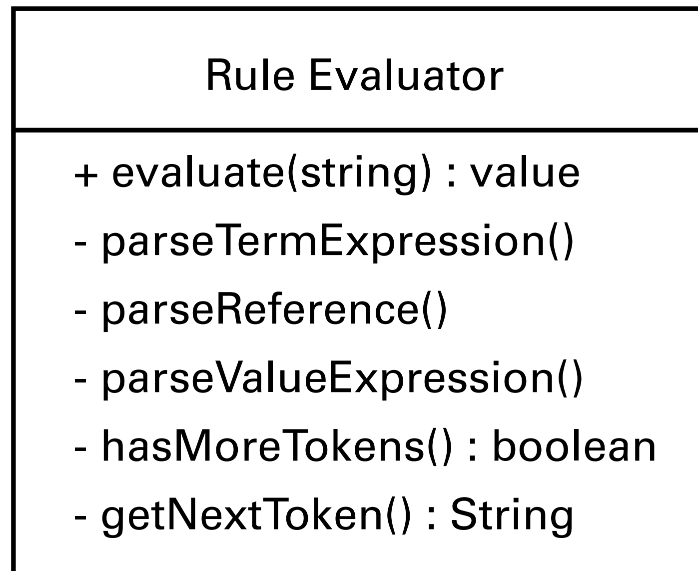


Figure 1: A typical class.

The `getNextToken` method is private. Theoretically, we should be able write tests for `evaluate` that exercise `getNextToken()`, but sometimes it is difficult to engineer the inputs we'd want to test a deeply called method specifically. And, if we go down this route, the tests we write might be indirect and not very self-explanatory.

People often run into this problem and think that it is just a case of testing being difficult work. But let's look again.

Is there a problem with `RuleEvaluator`? It turns out that there is. `RuleEvaluator` has three separate responsibilities: It evaluates, parses, and tokenizes. While code can often exist well for a while in that state, invariably, the separate responsibilities become tangled and the class becomes difficult to maintain. Let's see what happens when we start to fix the design problems.

Figure 2 shows the system after we've extracted a class for tokenization. Our testing situation is much better now. The `getNextToken` method is public on a new class and we can test it easily. Improving the design also fixed the testing problem.

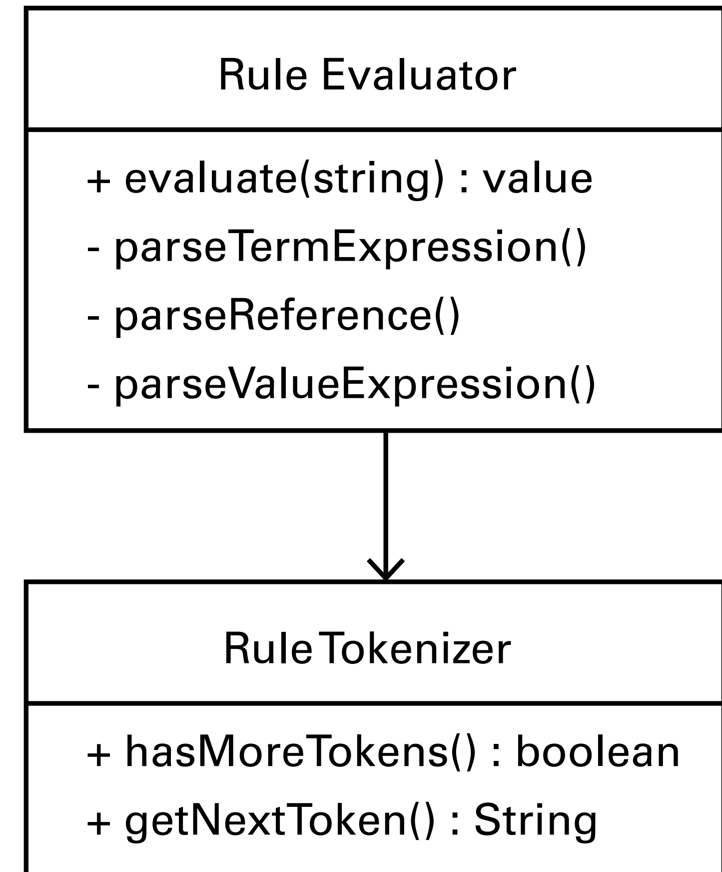


Figure 2: An improved version.

This example might annoy you. You may think that without any need to reuse the code in `getNextToken()` or `hasMoreTokens()`, pulling them out into their own class is pointless — it increases the number of classes needlessly. But making this move does make the code more modular. We could reuse `RuleTokenizer` at some point in the future. At the very least, it is now very clear where we have to go

in the code to make changes in tokenization functionality. Our code is more orthogonal and it is easier to change one thing without impacting another.

If this were the only example of lack of testability being an indicator of poor design, it could just be considered a fluke, but it isn't. In general, pain in unit testing is an indication that there is something wrong. Sometimes this pain is very obvious. For example, if you have to create too many objects to get the one you want to test, it is often an indication of excessive coupling. Other cases are subtler. Let's examine a few.

Other Indicative Smells

It's generally accepted that unit tests should run in isolation. We should be able to execute each of them without affecting the state of any other test. When we don't work this way, we end up with unexplained failures that are hard to reproduce. It often takes a considerable amount of debugging to find the source of these problems. The design problem that this indicates is global mutable state. We've known for decades that global variables can be problematic. Unit testing makes the pain evident and, once again, if we work on the design problem, unit testing becomes easier.

The fact that each of our tests should execute as if they were in a hermetically sealed container often helps us see other design issues. Often, I notice that when teams try to make key classes of their system independently testable, they discover that various objects don't release resources when they are destroyed. In the application, this sort of thing may never be noticed. When you are running thousands of unit tests, however, resource leaks become evident very quickly.

One of the most pervasive testing challenges points toward design issues as well. Teams often notice that unit testing in the presence of a third-party API is difficult. A classic example is UI-intensive code. Developers often place computational code in event handlers. It would be nice to be able to test those computational pieces independently, but it is nearly impossible because you can only exercise the code by triggering events from the UI. The design problem is lack of separation of concerns. Again, once the computation concern is separated from the UI concern, the computation is easily tested.

The big question at the end of all of this is, "Why?" Why do testing problems indicate design problems? A few years ago, I read an interesting paper by Thomas Mullen (http://is.gd/ExNVcV). In it, he hypothesized that many of the generally accepted principles of good design are "good" because they mirror our cognitive processes. It is easier for us to understand things in small chunks, and it helps when those chunks are independent. Perhaps the reason why unit testing serves as a good probe of design is because it is, essentially, a cognitive process. When we write tests, we are going through a reasoning process, and we are making our reasoning explicit in test code. Pieces of a program that are hard to understand with tests are likely to be hard to understand without them.

It's an odd situation to be in, isn't it? Pain isn't fun. We can create tooling that eliminates much of the pain of unit testing, but then what would we have? We might just end up with less of the feedback that could trigger us to discover better design.

Michael Feathers is the founder and Director of R7K Research & Conveyance, a company specializing in software and organization design. He is also the author of **Working Effectively with Legacy Code** *(http://is.gd/K5FXn2).*

Comment

From the Vault

Unit Testing in C: Tools and Conventions

Two lightweight testing frameworks make it easy to unit test C code.

By James W. Grenning @jwgrenning

In this article, I look at unit tests using two unit test harnesses that work in C. Along the way, I will also discuss some of the common terminology of automated unit testing.

Let me start by discussing the fundamental tool: the test harness.

What Is A Unit Test Harness?

A unit test harness is a software package that allows a programmer to express how production code should behave. A unit test harness's job is to provide these capabilities:

- A common language to express test cases
- A common language to express expected results
- Access to the features of the production code programming language
- A place to collect all the unit test cases for the project, system, or subsystem
- A mechanism to run the test cases, either in full or in partial batches
- A concise report of the test suite success or failure
- A detailed report of any test failures

I'll shortly look at two popular harnesses for testing embedded C.They are both easy to use and are descendants of the xUnit family of unit test harnesses.

First, I'll employ Unity, a C-only test harness (http://is.gd/tJc88X). Later, I will use CppUTest, a unit test harness written in C++, but not requiring C++ knowledge to use. You'll find that the bulk of the material in this article can be applied using any test harness.

Here are a few terms that will come in handy while reading this explanation:

- Code under test is just like it sounds; it is the code being tested.
- Production code is code that is (or will be) part of the released product.
- Test code is code that is used for testing the production code and is not part of the released product.
- A test case is test code that describes the behavior of code under test. It establishes the preconditions and checks that significant post conditions are met.
- A test fixture is code that provides the proper environment for a series of test cases that exercise the code under test. A test fixture will assist in establishing a common setup and environment for exercising the production code.

To take the mystery out of these terms, let's look at a few tests for something we've all used: `sprintf`. For this first example, `sprintf` is the code under test; it is production code.

`sprintf` is good for a first example because it is a standalone function, which is the most straightforward kind of function to test. The output of a standalone function is fully determined by the parameters passed immediately to the function. There are no visible external interactions and no stored state to get in the way. Each call to the function is independent of all previous calls.

Unity: A C-Only Test Harness

Unity is a straightforward, small unit test harness. It comprises just a few files. Let's get familiar with Unity and unit tests by looking at a couple example unit test cases. If you are a long-time Unity user, you'll notice some additional macros that are helpful when you are not using Unity's scripts to generate a test runner.

A test should be short and focused. Think of it as an experiment that silently does its work when it passes, but makes some noise when it fails. This test checks that `sprintf` handles a format spec with no format operations.

```
TEST(sprintf, NoFormatOperations)
{
    char output[5];
    TEST_ASSERT_EQUAL(3, sprintf(output, “hey”));
    TEST_ASSERT_EQUAL_STRING(“hey”, output);
}
```

The `TEST` macro defines a function that is called when all tests are run. The first parameter is the name of a group of tests. The second parameter is the name of the test. We'll look at `TEST` in more detail later.

The `TEST_ASSERT_EQUAL` macro compares two integers. `sprintf` should report that it formatted a string of length three, and if it does, the `TEST_ASSERT_EQUAL` check succeeds. As is the case with most unit test harnesses, the first parameter is the expected value.

`TEST_ASSERT_EQUAL_STRING` compares two null-terminated strings. This statement declares that output should contain the string `“hey”`: Following convention, the first parameter is the expected value.

If either of the checked conditions is not met, the test will fail. The checks are performed in order, and the `TEST` will terminate on the first failure.

Notice that `TEST_ASSERT_EQUAL_STRING` could pass by accident; if the output just happened to hold the `“hey”` string, the test

would pass without `sprintf` doing a thing. Yes, this is unlikely, but we better improve the test and initialize the output to the empty string.

```
TEST(sprintf, NoFormatOperations)
{
    char output[5] = "";
    TEST_ASSERT_EQUAL(3, sprintf(output, "hey"));
    TEST_ASSERT_EQUAL_STRING("hey", output);
}
```

The next `TEST` challenges `sprintf` to format a string with `%s`.

```
TEST(sprintf, InsertString)
{
    char output[20] = "";
    TEST_ASSERT_EQUAL(12, sprintf(output,
        "Hello %s\n", "World"));
    TEST_ASSERT_EQUAL_STRING("Hello World\n", output);
}
```

A weakness in both the preceding tests is that they do not guard against `sprintf` writing past the string terminator. The following tests watch for output buffer overruns by filling the output with a known value and checking that the character after the terminating null is not changed.

```
TEST(sprintf, NoFormatOperations)
{
    char output[5];
    memset(output, 0xaa, sizeof output);
    TEST_ASSERT_EQUAL(3, sprintf(output, "hey"));
    TEST_ASSERT_EQUAL_STRING("hey", output);
    TEST_ASSERT_BYTES_EQUAL(0xaa, output[4]);
}
TEST(sprintf, InsertString)
{
    char output[20];
    memset(output, 0xaa, sizeof output);
    TEST_ASSERT_EQUAL(12, sprintf(output,
        "Hello %s\n", "World"));
    TEST_ASSERT_EQUAL_STRING("Hello World\n", output);
    TEST_ASSERT_BYTES_EQUAL(0xaa, output[13]);
}
```

If you're worried about `sprintf` corrupting memory in front of `output`, we could always make `output` a character bigger and pass `&output[1]` to `sprintf`. Checking that `output[0]` is still `0xaa` would be a good sign that `sprintf` is behaving itself.

In C, it is hard to make tests totally fool-proof. Errant or malicious code can go way beyond the end or way in front of the beginning of `output`. It's a judgment call on how far to take the tests. You will see when we get into TDD how to decide which tests to write.

With those tests, you can see some subtle duplication creeping into the tests. There are duplicate `output` declarations, duplicate initializations, and duplicate overrun checks. With just two tests, this is no big deal, but if you happen to be `sprintf's` maintainer, there will be many more tests. With every test added, the duplication will crowd out and obscure the code that is essential to understand the test case. Let's see how a test fixture can help `TEST` cases.

Test Fixtures in Unity

Duplication reduction is the motivation for a test fixture. A test fixture helps organize the common facilities needed by all the tests in one place. Notice how `TEST_SETUP and TEST_TEAR_DOWN` keep duplication out of the `sprintf` tests.

```
TEST_GROUP(sprintf);

static char output[100];
static const char * expected;
TEST_SETUP(sprintf)
{
    memset(output, 0xaa, sizeof output);
    expected = "";
}

TEST_TEAR_DOWN(sprintf)
{
}
static void expect(const char * s)
{
    expected = s;
}
static void given(int charsWritten)
{
    TEST_ASSERT_EQUAL(strlen(expected), charsWritten);
    TEST_ASSERT_EQUAL_STRING(expected, output);
    TEST_ASSERT_BYTES_EQUAL(0xaa, output[strlen(expected) + 1]);
}
```

The shared data items defined after the `TEST_GROUP` are initialized by `TEST_SETUP` before the opening curly brace of each `TEST`. The data items comprise file scope, accessible by each `TEST` and all the helper functions. For this `TEST_GROUP`, there is no cleanup work for `TEST_TEAR_DOWN`. The file scope helper functions, `expect` and `given`, help keep the `sprintf` tests clean and low on duplication.

In the end, it's just plain C, so you can do what you want as far as shared data and helper functions. I'm showing the typical way to structure a group of tests with common data and condition checks.

Now these tests are focused, lean, mean, and to the point.

```
TEST(sprintf, NoFormatOperations)
{
    expect("hey");
    given(sprintf(output, "hey"));
}

TEST(sprintf, InsertString)
{
    expect("Hello World\n");
    given(sprintf(output, "Hello %s\n", "World"));
}
```

Notice that once you understand a specific `TEST_GROUP` and have seen a couple examples, writing the next test case is much less work. When there is a common pattern within a `TEST_GROUP`, each test case is easier to read, understand, and evolve, as change becomes necessary.

Installing Unity Tests

It is not evident from the example how the test cases get run with the necessary pre- and post-processing. It's done with another macro: the TEST_GROUP_RUNNER. The TEST_GROUP_RUNNER can go in the file with the tests or a separate file.

To avoid scrolling through the file, I use a separate file. For the two sprintf tests written, the TEST_GROUP_RUNNER looks like this:

```
#include "unity_fixture.h"
TEST_GROUP_RUNNER(sprintf)
{
    RUN_TEST_CASE(sprintf, NoFormatOperations);
    RUN_TEST_CASE(sprintf, InsertString);
}
```

Each test case is called through the RUN_TEST_CASE macro. Essentially, this RUN_GROUP_RUNNER calls the function bodies associated with each of these macros:

```
TEST_SETUP(sprintf);
TEST(sprintf, NoFormatOperations);
TEST_TEAR_DOWN(sprintf);

TEST_SETUP(sprintf);
TEST(sprintf, InsertString);
TEST_TEAR_DOWN(sprintf);
```

Invoking TEST_SETUP before each TEST means that each test starts out fresh, with no accumulated state. TEST_TEAR_DOWN is called to clean up after each test.

Now that the tests are wired into a TEST_GROUP_RUNNER, let's see how the TEST_GROUP_RUNNERs are called. For this last step, we have to look at main. You will have a main for your production code and one, or more, for your test code. The Unity test main looks like this:

```
#include "unity_fixture.h"

static void RunAllTests(void)
{
    RUN_TEST_GROUP(sprintf);
}

int main(int argc, char * argv[])
{
    return UnityMain(argc, argv, RunAllTests);
}
```

RUN_TEST_GROUP(GroupName) calls the function defined by TEST_GROUP_RUNNER. Each TEST_GROUP_RUNNER you want to run as part of your test main has to be mentioned in a RUN_TEST_GROUP. Notice that RunAllTests is passed to UnityMain.

One unfortunate side effect of using a C-only test harness is that you have to remember to install each TEST into a TEST_GROUP_RUNNER, and the runner is invoked by calling UnityMain. If you forget, tests will compile, but not run — potentially giving a false positive.

Because of this opportunity for error, the designers of Unity created a system of code generators that read your test files and produce the

needed test runner code. To keep the dependencies low for getting started with Unity, I've opted to not use the code-generating scripts and manually wire all the test code. When I discuss CppUTest later in this article, you will see another solution to that problem. But before doing that, let's look at Unity's output.

Unity Output

The tests should be run as part the automated test build. A single command builds and runs your test executable. I prefer to build often, with each small change. This is TDD. I set up my development environment to automatically make all whenever a file is saved. Test output looks like this:

```
make
compiling SprintfTest.c
Linking BookCode_Unity_tests
Running BookCode_Unity_tests
..
-----------------------
2 Tests 0 Failures 0 Ignored
OK
```

Notice that when all tests are passing, the output is minimal. At quick glance, a single line of text says "OK," which means, "All tests passing." In the UNIX style, the test harness follows the "no news is good news" principle. (When a test case fails, as you will see shortly, it reports a specific error message.) It's pretty self-explanatory, but let's decipher the test output and summary line. Notice also that a dot (.) is printed before each test case runs. For a long test run, this lets you know something is happening. The line of hyphens (- - -) is just a separator line for the test summary.

- Tests — the total number of `TEST` cases.
- Failures — the total number of `TEST` cases that failed.
- Ignored — a count of the number of tests in ignore state. Ignored tests are compiled but are not run.

Let's add a failing test to see what happens. Look at the test output, and the intentional error in this test case will be evident:

```
TEST(sprintf, NoFormatOperations)
{
    char output[5];
    TEST_ASSERT_EQUAL(4, sprintf(output, "hey"));
    TEST_ASSERT_EQUAL_STRING("hey", output);
}
```

The failure looks like this:

```
make
compiling SprintfTest.c
Linking BookCode_Unity_tests
Running BookCode_Unity_tests
..
TEST(sprintf, NoFormatOperations)
    stdio/SprintfTest.c:75: FAIL
    Expected 4 Was 3
-----------------------
```

```
2 Tests 1 Failures 0 Ignored
FAIL
```

The failure reports the filename and line of the failing test case, the name of the test case, and the reason for failure. Also notice the summary line now shows one test failure.

Now, let's look at CppUTest.

CppUTest: A C++ Unit Test Harness

Now that we've seen Unity, I will now move to describing CppUTest (http://is.gd/nDGxX4), my preferred unit test harness for C and C++. In full disclosure, I am partial to CppUTest, not only because it is a capable test harness but also because I am one of its authors. The first examples in this article use Unity. The later examples, use CppUTest.

CppUTest was developed to support multiple OS platforms with a specific goal of being usable for embedded development. The CppUTest macros make it so that test cases can be written without knowledge of C++. This makes it easy for C programmers to use the test harness.

CppUTest uses a primitive subset of C++; it's a good choice for embedded development where not all compilers support the full C++ language. You will see that the test cases are nearly identical between Unity and CppUtest. You, of course, can use whichever test harness you prefer for your product development.

This CppUTest test case is equivalent to the second Unity test case found in the section on `sprintf`.

```
TEST(sprintf, NoFormatOperations)
{
    char output[5] = "";
    LONGS_EQUAL(3, sprintf(output, "hey"));
    STRCMP_EQUAL("hey", output);
}
```

Besides the macro names, the test cases are the same.

Let's look at a CppUTest test fixture that is equivalent to the earlier example Unity test fixture (discussed in the "Test Fixtures in Unity" section).

```
TEST_GROUP(sprintf)
{
    char output[100];
    const char * expected;

    void setup()
    {
        memset(output, 0xaa, sizeof output);
        expected = "";
    }
    void teardown()
    {
    }
    void expect(const char * s)
    {
        expected = s;
    }
    void given(int charsWritten)
    {
        LONGS_EQUAL(strlen(expected), charsWritten);
```

```
        STRCMP_EQUAL(expected, output);
        BYTES_EQUAL(0xaa, output[strlen(expected) + 1]);
    }
};
```

Again, it is very similar to the previous example, with all the same concepts represented. One formatting difference is that the CppUTest `TEST_GROUP` is followed by a set of curly braces enclosing shared data declarations and functions. Everything between the curly braces

"One advantage to CppUTest is that tests self-install. There is no need for an external script."

is part of the `TEST_GROUP` and is accessible to each `TEST` in the group. The shared data items (`output`, `expected`, and `length`) are initialized by a special helper function called `setup`. As you might guess, `setup` is called before each `TEST`. Another special function, `teardown`, is called after each `TEST`. (In this example, it is not used.) `expect` and `given` are free-form helper functions that are accessible to all `TEST` cases in the `TEST_GROUP`.

These refactored test cases are identical to the Unity test cases:

```
TEST(sprintf, NoFormatOperations)
{
    expect("hey");
    given(sprintf(output, "hey"));
}

TEST(sprintf, InsertString)
{
    expect("Hello World\n");
    given(sprintf(output, "%s\n", "Hello World"));
}
```

One advantage to CppUTest is that tests self-install. There is no need for an external script to generate a test runner or to manually write and maintain test-wiring code like `RUN_TEST_CASE`, `TEST_GROUP_RUNNER`, and `RUN_TEST_GROUP`. On the minor difference list are the assertion macros; each test harness supports different macros, though there is functional overlap.

You may notice that Unity and CppUTest are suspiciously close in their macros and test structure. Well, there is no real mystery there; they do follow a well-established pattern that I first saw with JUnit, a Java test framework. The more specific similarities are because I contributed the test fixture-related macros to the Unity project.

CppUTest Output

As already explained for Unity, tests run as part of an automated build using make. Test output looks like this:

```
make all
compiling SprintfTest.cpp
Linking BookCode_tests
Running BookCode_tests
..
OK (2 tests, 2 ran, 0 checks, 0 ignored, 0 filtered out)
```

Just like with Unity, when all tests are passing, the output is minimal. Here is how to interpret the summary line of the test run:

- tests — the total number of `TEST` cases.
- ran — the total number of `TEST` cases that ran (in this case, they passed, too).
- checks — a count of the number of condition checks made. (Condition checks are calls such as `LONGS_EQUAL`.)
- ignores — a count of the number of tests in ignore state. Ignored tests are compiled but are not run.
- filtered out — a count of the number of tests that were filtered out of this test run. Command-line options select specific tests to run.

Let's insert an error into the test to see what the output looks like:

```
TEST(sprintf, NoFormatOperations)
{
    char output[5];

    LONGS_EQUAL(4, sprintf(output, "hey"));
    STRCMP_EQUAL("hey", output);
}
```

The failure looks like this:

```
make
compiling SprintfTest.cpp
Linking BookCode_Unity_tests
Running BookCode_Unity_tests
...
stdio/SprintfTest.cpp:75: TEST(sprintf, NoFormatOperations)
expected <4 0x2>
but was <3 0x1>
Errors (1 failures, 2 tests, 2 ran, 1 checks, 0 ignored, 0 filtered out, 0 ms)
```

"If you ever insert an error on purpose into a test case, make sure you remove it, or you risk baking a bug into your code"

The failure reports the line of the failing condition check, the name of the test case, and the reason for failure. Also notice the summary line includes a count of test failures.

If you ever insert an error on purpose into a test case, make sure you remove it, or you risk baking a bug into your code.

Unit Tests Can Crash

One other possible outcome during a test run is a crash. Generally speaking, C is not a safe language. The code can go off into the weeds, never to return. `sprintf` is a dangerous function. If you pass it an

output buffer that is too small, it will corrupt memory. This error might crash the system immediately, but it might cause a crash later. The behavior is undefined. Consequently, a test run may silently exit with an `OK`, silently exit early showing no errors, or crash with a bang.

When you have a silent failing or crashing test, let the test harness help you confirm what is wrong. Sometimes a production code change will cause a previously passing test to fail, or even crash. So, before chasing the crash, make sure you know which test is failing.

Because the test harness is normally quiet except for test failures, when a test crashes, you probably won't get any useful output. Both Unity and CppUTest have a command-line option for running the test in verbose mode (`-v`). With `-v`, each `TEST` announces itself before running. Conveniently, the last `TEST` mentioned is the one that crashed.

You can also filter tests by test group (`-g testgroup`) and test case (`-n testname`). This lets you get very precise about which test cases are running. These are very helpful for chasing down crashes.

The Four-Phase Test Pattern

In Gerard Meszaros' book, *xUnit Testing Patterns* (http://is.gd/8owKf0), he describes the Four-Phase Test, which is what I use, too. The goal of the pattern is to create concise, readable, and well-structured tests. If you follow this pattern, the test reader can quickly determine what is being tested. Paraphrasing Gerard, here are the four phases:

- Setup: Establish the preconditions to the test.
- Exercise: Do something to the system.
- Verify: Check the expected outcome.
- Cleanup: Return the system under test to its initial state after the test.

To keep your tests clear, make the pattern visible in your tests. When this pattern is broken, the documentation value of the test is diminished; the reader has to work harder to understand the requirements expressed by the test.

Conclusion

At this point, you should have a good overview of Unity and CppUTest, and understand how test fixtures and test cases allow a set of tests to be defined. Whether you use them to practice TDD on your C projects or just to ensure higher code quality is entirely up to you.

[If you choose to go the TDD route, though, you might find the remainder of the material in the book from which this article is excerpted to be useful in your work. See below. —Ed.]

This article is excerpted and adapted from* Test-Driven Development for Embedded C *(http://is.gd/I9lqCV), published by Pragmatic Programmers (http://www.pragprog.com/). You can find the complete code by visiting the book's home page (http://is.gd/g7UtEX). James W. Grenning invented Planning Poker, an Agile estimation technique, and is one of the original authors of the Manifesto for Agile Software Development.

Comment

Previous

Table of Contents

Dr. Dobb's

Andrew Binstock Editor in Chief, Dr. Dobb's
andrew.binstock@ubm.com

Deirdre Blake Managing Editor, Dr. Dobb's
deirdre.blake@ubm.com

Amy Stephens Copyeditor, Dr. Dobb's
amy.stephens@ubm.com

Jon Erickson Editor in Chief Emeritus, Dr. Dobb's

CONTRIBUTING EDITORS
Scott Ambler
Mike Riley
Herb Sutter

DR. DOBB'S EDITORIAL
751 Laurel Street #614
San Carlos, CA
94070
USA

UBM TECH
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

INFORMATIONWEEK
Rob Preston VP and Editor In Chief, InformationWeek
rob.preston@ubm.com 516-562-5692

Chris Murphy Editor, InformationWeek
chris.murphy@ubm.com 414-906-5331

Lorna Garey Content Director, Reports, InformationWeek
lorna.garey@ubm.com 978-694-1681

Brian Gillooly, VP and Editor In Chief, Events
brian.gillooly@ubm.com

INFORMATIONWEEK.COM
Laurianne McLaughlin Editor
laurianne.mclaughlin@ubm.com 516-562-5336

Roma Nowak Senior Director,
Online Operations and Production
roma.nowak@ubm.com 516-562-5274

Joy Culbertson Web Producer
joy.culbertson@ubm.com

Atif Malik Director,
Web Development
atif.malik@ubm.com

MEDIA KITS
http://createmarketingservices.com/

UBM TECH
AUDIENCE DEVELOPMENT
Director, Karen McAleer
(516) 562-7833, karen.mcaleer@ubm.com

SALES CONTACTS—WEST
Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

Sales Director, Michele Hurabiell
(415) 378-3540, michele.hurabiell@ubm.com

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, sandra.kupiec@ubm.com

Account Manager, Vesna Beso
(415) 947-6104, vesna.beso@ubm.com

Account Executive, Matthew Cohen-Meyer
(415) 947-6214, matthew.meyer@ubm.com

MARKETING
VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, winnie.ng@ubm.com

Marketing Director, Angela Lee-Moll
(516) 562-5803, angele.leemoll@ubm.com

Marketing Manager, Monique Luttrell
(949) 223-3609, monique.luttrell@ubm.com

Program Manager, Nicole Schwartz
516-562-7684, nicole.schwartz@ubm.com

SALES CONTACTS—EAST
Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Steven Sorhaindo
(212) 600-3092, steven.sorhaindo@ubm.com

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mary.hyland@ubm.com

Account Manager, Tara Bradeen
(212) 600-3387, tara.bradeen@ubm.com

Account Manager, Jennifer Gambino
(516) 562-5651, jennifer.gambino@ubm.com

Account Manager, Elyse Cowen
(212) 600-3051, elyse.cowen@ubm.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kathleen.jurina@ubm.com

BUSINESS OFFICE
General Manager,
Marian Dujmovits
United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030
(516) 562-5000

Copyright 2014.
All rights reserved.

UBM TECH
Paul Miller, CEO
Robert Faletra, CEO, Channel
Kelley Damore, Chief Community Officer
Marco Pardi, President, Business Technology Events
Adrian Barrick, Chief Content Officer
David Michael, Chief Information Officer
Sandra Wallach CFO
Simon Carless, EVP, Game & App Development and Black Hat
Lenny Heymann, EVP, New Markets
Angela Scalpello, SVP, People & Culture
Andy Crow, Interim Chief of Staff

UNITED BUSINESS MEDIA LLC
Pat Nohilly Sr. VP, Strategic Development and Business Administration
Marie Myers Sr. VP, Manufacturing

UBM TECH ONLINE COMMUNITIES
Bank Systems & Tech
Dark Reading
DataSheets.com
Designlines
Dr. Dobb's
EBN
EDN
EE Times
EE Times University
Embedded
Gamasutra
GAO
Heavy Reading
InformationWeek
IW Education
IW Government
IW Healthcare
Insurance & Technology
Light Reading
Network Computing
Planet Analog
Pyramid Research
TechOnline
Wall Street & Tech

UBM TECH EVENT COMMUNITIES
4G World
App Developers Conference
ARM TechCon
Big Data Conference
Black Hat
Cloud Connect
DESIGN
DesignCon
E2
Enterprise Connect
ESC
Ethernet Expo
GDC
GDC China
GDC Europe
GDC Next
GTEC
HDI Conference
Independent Games Festival
Interop
Mobile Commerce World
Online Marketing Summit
Telco Vision
Tower & Cell Summit

http://createmarketingservices.com/

Entire contents Copyright © 2014, UBM Tech/United Business Media LLC, except where otherwise noted. No portion of this publication may be reproduced, stored, transmitted in any form, including computer retrieval, without written permission from the publisher. All Rights Reserved. Articles express the opinion of the author and are not necessarily the opinion of the publisher. Published by UBM Tech/United Business Media, 303 Second Street, Suite 900 South Tower, San Francisco, CA 94107 USA 415-947-6000.