Simplifying Contract Testing

Article *in* Doctor Dobbs Journal · April 2014

CITATIONS
0

READS
94

1 author:

Claude N Warren, Jr.
Aiven Oy
13 PUBLICATIONS **121** CITATIONS

SEE PROFILE

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.

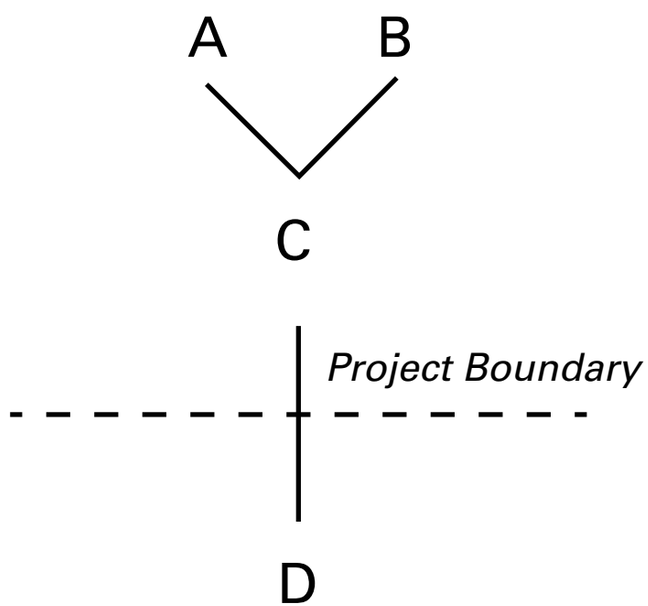

Figure 1. The relationship between multiple interfaces to a project class.

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
```

```
@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