Testability, Test Automation and Test Driven Development for the Trick Simulation Toolkit

John M. Penn[1]
L-3 Communications, Houston, Texas, 77058, USA

This paper describes the adoption of a Test Driven Development approach and a Continuous Integration System in the development of the Trick Simulation Toolkit, a generic simulation development environment for creating high fidelity training and engineering simulations at the NASA Johnson Space Center and many other NASA facilities. It describes the approach, and the significant benefits seen, such as fast, thorough and clear test feedback every time code is checked into the code repository. It also describes an approach that encourages development of code that is testable and adaptable.

I. Introduction

The Trick Simulation Toolkit is a simulation development environment for creating high fidelity training and engineering simulations at the NASA Johnson Space Center and many other NASA facilities. The Trick environment provides a common architecture for user defined simulations. Trick builds executable simulations using user supplied simulation definition files and user supplied "model code". For each Trick based simulation, Trick automatically provides job scheduling, checkpoint/restore, data-recording, interactive variable manipulation (variable server) and an input processor. Also included are tools for plotting recorded data and various other supporting utilities and libraries.

Trick is written in C/C++ and Java and supports both Linux and MacOSX computer operating systems.

II. Trick Had a Testing Problem

Prior to adopting a new development and testing approach in 2010, Trick testing consisted primarily of running a few large simulations with the hope that their scale and complexity would exercise most of Trick's code and expose any recently introduced bugs. This proved inadequate in that it was allowing many bugs to be introduced and remain undetected until after being released to the Trick user community.

Whereas there did exist unit tests for some of Trick's code, most had none. What existed were ad hoc test applications that were frequently out of date. Because they were not consistently being run, the tests themselves weren't being tested. Also contributing to the problem of insufficient unit testing was that functions and classes were often difficult to isolate into independently testable units because they contained unnecessary dependencies on other parts of the Trick framework.

Class and function interfaces often were complex[6] and unclear. Usage of global variables created obscure interfaces. Within composite (i.e., class and struct) function parameters, it wasn't always clear which members were necessary for the function and which weren't.

Class and function responsibilities too were often complex[6], unrelated and unclear. Sometimes, the expected behavior in response to possible combinations of inputs was simply undefined.

III. Understanding Why The Software Was Difficult to Test

By tracing the sources of dependencies in functions and classes, and by using the concepts of *coupling* and *cohesion*, one can see that the difficulties of isolating functions and classes for testing are caused by inadequate *separation of concerns* and the use of global variables.

[1] Senior Engineer, 1002 Gemini St, Suite 200

Coupling is the degree to which the components of a system are dependent upon one another,[2] regardless of whether they should be.

Cohesion is the degree to which the concerns of a component are about the same idea.[2] If each concern in a set is an aspect of the same idea, then the cohesion of the set is said to be *high*. Concerns that are not related are said to have *low* cohesion.

Separation of Concerns is the software design principle of organizing code by dividing it into separate parts that address separate concerns.[1] It is strongly related to the concepts of coupling and cohesion. Systems whose components are well organized into separate concerns typically exhibit low coupling and high cohesion. Those that are poorly organized and don't separate concerns typically have high coupling and low cohesion.

A. Dependencies in Functions

Suppose we model a function as a dependency tree as shown in Fig.1 and described below:

1) Every useful function is associated with one or more responsibilities.
2) Responsibilities are fulfilled using interface variables and/or function-calls.
3) Global, external interface variables can create file dependencies because they can be defined elsewhere in separate compilation units.
4) Every interface variable has a data type.
5) Function calls can create file dependencies because they may require function implementations that are defined elsewhere in separate compilation units.
6) Data type usage can create file dependencies if they are defined elsewhere in a different file.

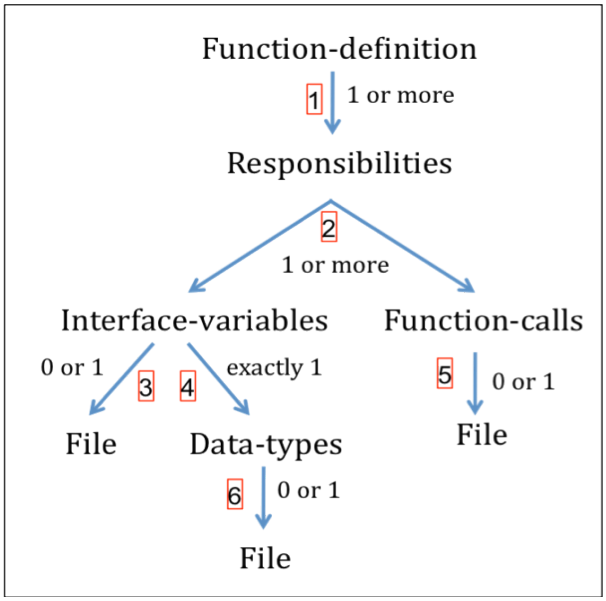

Figure 1 - Generalized Function Dependencies

From this model, we can see that functions can create file dependencies by:

1) Adding interface variables that are either global or that introduce new data types.
2) Adding calls to new (subordinate) functions.

Because interface variables can result in file dependencies, unnecessary interface variables can result in unnecessary file dependencies, thereby making a function less isolatable for testing.

Also notice that a function with multiple responsibilities binds the dependencies of those responsibilities together. For example, suppose a function contains two responsibilities: R1 and R2. If R1 results in two file dependencies and R2 results in three file dependencies, then the function will have between three and five file dependencies. It will have three dependencies if R1 and R2 have two dependencies in common and five if R1 and R2 have no common dependencies.

Cohesive responsibilities are more likely to have common dependencies than non-cohesive ones. That is, related things are more likely to have something in common than unrelated things. Therefore, a combination of cohesive responsibilities tends to create fewer dependencies than a combination of non-cohesive ones. If the responsibilities were separated into two functions, then both would have fewer dependencies than the original, and each would be less complex[6] and more easily isolated for testing. By separating responsibilities into one per function, dependencies can be minimized.

B. Dependencies in Composite Data types

Suppose we model a composite data type as shown in Fig. 2 and described below:

1) Every composite data type is associated with one or more concerns or *topics*.
2) Topics are represented by data members and/or member function interfaces.
3) Every data member has a data type.
4) Member function interfaces can create file dependencies.
5) Data types can create file dependencies.

From this, we can see that composite data types can create file dependencies by:

1) Adding member variables that introduce new data types.
2) Adding member functions.

Like functions, composite data types can have similar problems:

1) Unnecessary member variables can create unnecessary data type dependencies, which can in turn create unnecessary file dependencies.
2) Combining non-cohesive topics within a composite data type has the effect of binding the unrelated file dependencies of its constituents. This too can make compilation units unnecessarily dependent upon one another. As with functions, this can make classes more difficult to test.

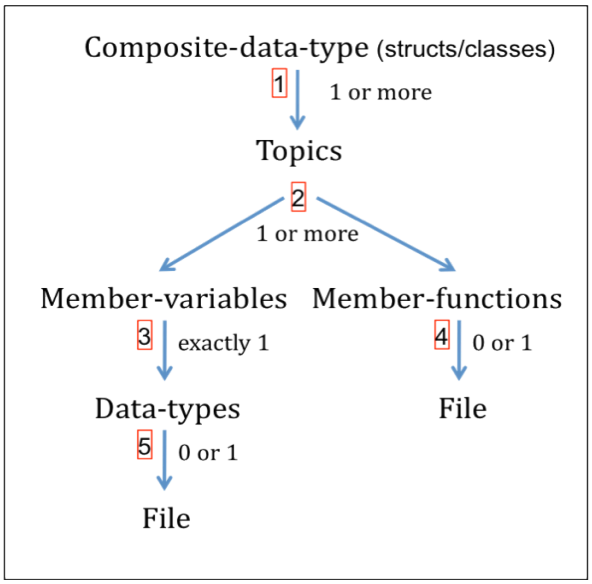

Figure 2 - Generalized Composite Data type Dependencies

C. Global Variables and Testability

Aside from creating ambiguous interfaces, global variables create another testability problem. Because the values within global variables are preserved between function calls, multiple test cases within a test application can affect each other's behavior. Note that composite class types with static members can potentially suffer from the same problem.

D. Making the Software Testable

Summarizing the above, one can make functions and classes isolatable by doing the following:

1) Avoid the use of global variables, as they represent an obscured part of a function's actual interface. They can cause different invocations of the same function to affect one another.
2) Separate the concerns/topics of composite data types into multiple, cohesive data types
   i) to avoid obscuring interfaces with unrelated parameter data and
   ii) to avoid binding unrelated topics and their dependencies, which make compilation units unnecessarily dependent on one another.
3) Separate function responsibilities, especially those that are unrelated, into individual functions to minimize dependencies, thereby making the functions isolatable.

Items 2) and 3) are both reasons to follow Robert C. Martin's *Single Responsibility Principle*. This principle of object-oriented development states that, "A class should have only one reason to change."[3] Martin explains :

> If a class has more than one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class' ability to meet the others. This kind of coupling leads to fragile designs that break in unexpected ways when changed.[3]

In addition to making software designs fragile, it is apparent that coupled responsibilities also make them difficult to test.

These rules are now fundamental to the approach of improving Trick's software and making it more testable.

IV. Adding Automation and Consistency

A test framework can bring consistency to how tests are created and executed, and how results are reported. When framework-based test applications integrate well with a continuous integration system, testing can be performed automatically and regularly.

A. Unit Test Framework

The Trick Project chose Googletest as a C++ unit test framework. Googletest is based on the xUnit architecture, like the JUnit test framework, which was already being used for Trick's Java software unit testing. In addition to other operating systems, Googletest supports both Linux and Mac OS X, the operating systems supported by Trick. Googletest is licensed Open Source (BSD).

A Googletest test application consists of

1) tests
2) test cases
3) test fixtures
4) assertions.

A test application may contain multiple test cases. Test cases are named collections of tests. Trick unit tests consist of setup and calls to the code being tested, and one or more assertions. An assertion call evaluates test values and conditions to determine whether the test passes or fails. Googletest provides many different kinds of assertion macros. A test case may also have an associated test fixture, a support class (named for the associated test case) that can provide a common environment for that test case.

Figure 3 shows an example Googletest based Trick unit test application. This example shows two tests, called *getDataType_1* and *getDataType_2*. The test case is named *TypeDictionaryTest*. As always the test case shares the name of the text fixture, if one exists. Assertions used in this test application are EXPECT_EQ and EXPECT_NE.

TypeDictionaryTest.cpp (~/trick_de...im_services/DataTypes/testing) - VIM

```
#include <gtest/gtest.h>
#include <stddef.h>
#include "TypeDictionary.hh"
#include "CompositeDataType.hh"
#include "CompositeValue.hh"
#include "PrimitiveDataType.hh"
#include "DataTypeTestSupport.hh"

// Test Fixture
class TypeDictionaryTest : public ::testing::Test {
    protected:
    TypeDictionary *typeDictionary;
    TypeDictionaryTest() { typeDictionary = new TypeDictionary; }
    ~TypeDictionaryTest() { delete typeDictionary; }
    void SetUp() { /* Done right before test is run. */ }
    void TearDown() { /* Done right after test is run. */}
};


TEST_F(TypeDictionaryTest, getDataType_1) {

    /* TypeDictionary::getDataType should return NULL if the specified
       typeName is not defined in the TypeDictionary.*/

    const DataType* dataType = typeDictionary->getDataType("non-existent-type");

    EXPECT_EQ( NULL, dataType);
}

TEST_F(TypeDictionaryTest, getDataType_2) {

    /* TypeDictionary should be pre-loaded with all of the builtin types */

    const DataType* dataType;

    dataType = typeDictionary->getDataType("void");
    EXPECT_NE( (void*)NULL, dataType);

    dataType = typeDictionary->getDataType("char");
    EXPECT_NE( (void*)NULL, dataType);

    dataType = typeDictionary->getDataType("short");
    EXPECT_NE( (void*)NULL, dataType);

    dataType = typeDictionary->getDataType("int");
    EXPECT_NE( (void*)NULL, dataType);

    dataType = typeDictionary->getDataType("long");
TypeDictionaryTest.cpp                                    1,1           Top
```

Figure 3 - Example Googletest-Based Trick Unit Test Case

B. Unit-test Results

A Googletest test application, in addition to printing test results to a terminal window, will, by specifying the `--gtest_output` flag, generate an JUnit XML test report. An example of which is shown in Fig. 4.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites tests="4" failures="0" disabled="0" errors="0" time="0" name="AllTests">
  <testsuite name="TypeDictionaryTest" tests="4" failures="0" disabled="0" errors="0" time="0">
    <testcase name="getDataType_1" status="run" time="0" classname="TypeDictionaryTest" />
    <testcase name="getDataType_2" status="run" time="0" classname="TypeDictionaryTest" />
    <testcase name="addTypeDefinition_1" status="run" time="0" classname="TypeDictionaryTest" />
    <testcase name="addTypeDefinition_2" status="run" time="0" classname="TypeDictionaryTest" />
  </testsuite>
</testsuites>
```

Figure 4 - Example XML Test Report

JUnit XML test reports provide a means of communicating test results to other automation tools, such as an integration server.

C. Continuous Integration Server

The Trick Project chose Jenkins as a continuous integration (CI) server. Jenkins is a Java application that monitors executions of one or more *jobs*, such as building and testing software projects, as defined by the user.

Job execution can be configured to be periodic or (more typically) to be triggered by some event, such as committing source code to the repository. Results are served as HTML pages. The Jenkins "dashboard" page displays whether a job execution has passed or failed. Jenkins' source code management integration, its test framework integration and its detailed build histories provide useful information for failure diagnosis. Jenkins is licensed Open Source (MIT).

Trick's Jenkins configuration defines multiple jobs, each associated with a different combination of

1) Trick development branch (Trick 10 or 13)
2) Machine data path size (32/64 bit)
3) Operating System distribution (Scientific Linux, Centos, Ubuntu and MacOSX).

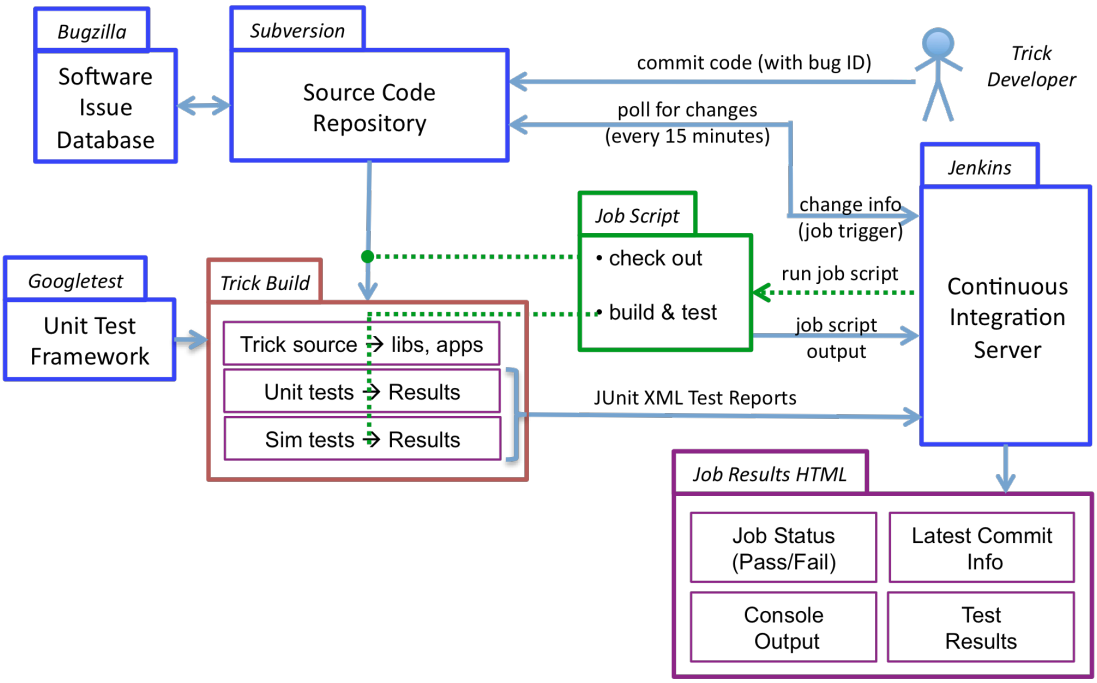

Figure 5 - Trick Continuous Integration System

As represented in Fig. 5, each of these jobs is configured to poll the source code repository every 15 minutes for changes. A code commit (check-in) to the repository triggers execution of each of the job scripts.

Each job script then:

- Checks out the latest version of Trick source code, on its respective target machine.
- Builds Trick libraries, user applications, unit test applications and simulation test applications.
- Executes the suite of unit test applications.
- Executes the suite of test simulations.

Each of the test applications generates reports in the Junit XML test report format and places them in a common directory.

Within 25 minutes of a source code commit, Jenkins produces build reports for each job. Build reports include:

- Whether the job succeeded or failed.
- An identification of the latest check-in by bug ID and description.
- All console output generated by the job build script.
- A detailed report of all tests.

V. Adopting Test Driven Development

In refactoring[4] Trick's code to make it more testable, it quickly became apparent that, aside from verification, writing unit tests while developing or refactoring the software to be tested was yielding valuable design insight.

Writing tests while refactoring the code served to repeatedly test the testability of that software. Because testability requires a separation of concerns, developing unit tests while refactoring the software was helping to expose flaws in the organization of the software as it was being developed. It became apparent that this, in essence, was why Test Driven Development (TDD)[5] works.

TDD is a technique "rediscovered" by Kent Beck that advocates developing code in small parts by repeating the following steps[5]:

1. Write a new test that verifies a new desired behavior.
2. Run all of the tests to see if the new one fails.
3. Write code that causes the new test to pass.
4. Refactor the code to clean it up and remove duplication.

Changing Trick's *existing* code to be testable tended to iterate between the first and fourth step above until a clear concise test could be developed after which all of the tests were run. The ability to write a clear, concise test was itself a test of the software.

Development of the Trick Simulation Toolkit began in 1990. It was not developed using the concepts of TDD, which was introduced in the early 2000s. However, it has become apparent that the concepts of TDD are not just beneficial to new software development, but provide the same benefits in the maintenance and refactoring of existing "legacy" code.

VI. Results

Trick currently has more than 1400 unit tests and six Trick based test simulations that are run regularly and provide broad test coverage of the Trick Simulation Toolkit. This is an increase from around 200 ad hoc tests that were run infrequently.

Within 25 minutes of source code being committed to the repository, Trick is checked out, built and tested with the entire suite of unit tests and simulation tests. If a code commit "breaks the build" or fails any test on any of the supported architectures, the development team knows with 25 minutes. In the event that a build fails, Trick's continuous integration system provides necessary information that allows the cause to be quickly identified.

In addition to meeting the original goals of testability and automation, several additional benefits have been observed.

Trick code is becoming more modular and adaptable. Because function and class responsibilities are less restricted by being bound to other responsibilities, and because they have low coupling and high cohesion, their interfaces can often be made more generic. In the redesign of Trick software, for example, this adaptability has led directly to the development of new replaceable components such as:

- *InputProcessor* class - an abstract class that provides a scripting capability to Trick based simulations. The default input processor is a Python interpreter.
- *CheckPointAgent* class - an abstract class used to "dump" and "restore" a simulation's state. Trick currently has two selectable components derived from the *CheckPointAgent* class, one to dump a checkpoint in a "classic" (the old) format and one in the form of Python assignment statements. New *CheckPointAgent* derivatives could just as easily create an XML-based, or other representation, without affecting other Trick software.
- *DataProductView* class - an abstract class used to create an "external" representation of a data plot or table. Two classes derived from DataProductView create GnuPlot and XWindows representations of data plots without duplication and without affecting other Trick software.
- *RealtimeClock* - an abstract clock.
- *Scheduler* - an abstract job scheduler.

Another benefit of unit tests of functions and classes that have clear, concise responsibilities is that they serve as working examples of how to use those functions and classes. Thus, they not only serve as test, but they can also serve as a kind of documentation.

VII. Conclusions

Understanding of the requirements of testable software, test automation tools, and adoption of the Test Driven Development process have dramatically improved testing of the Trick Simulation Toolkit. The Separation of Concerns design principle helps Trick developers to design code that is well organized and testable. Test Driven Development encourages separation of concerns. Trick's test automation system now ensures that unit tests are run regularly and provide fast, consistent build and test feedback.While unit testing is not the only type of testing required to ensure reliable software, Trick's developers and many of Trick's users say that they have more confidence in Trick now than in previous versions.

References

[1]Dijkstra, Edsger W (1982). "On the role of scientific thought". Selected writings on Computing: A Personal Perspective. New York, NY, USA: Springer-Verlag. pp. 60-66. ISBN 0-387-90652-5.

[2]W Stevens, GJ. Myers, L. Constantine, "Structured Design", IBM Systems Journal, 13 (2), 115-139, 1974.

[3]Martin, R. C., and Martin, M., "Agile Principles, Patterns, and Practices in C#", Prentice Hall, Upper Saddle River, 2006, Chapter 8.

[4]Fowler, Martin (1999). Refactoring. Improving the Design of Existing Code. Addison-Wesley. ISBN 0-201-48567-2.

[5]Beck, K. Test-Driven Development by Example, Addison Wesley - Vaseem, 2003

[6]Binstock, A., "Measuring Complexity Correctly", Dr. Dobbs, The World of Software Development, URL: http://www.drdobbs.com/architecture-and-design/measuring-complexity-correctly/240007928 [cited 7 May 2013]