WPI-CS-TR-2009-02 February 2009

Unit Testing of Software Components with Inter-component Dependencies

by

George T. Heineman
heineman@cs.wpi.edu

Unit testing of Software Components with inter-component dependencies

George Heineman
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA 01609
heineman@cs.wpi.edu

Abstract

Test Driven Development (TDD) is a process for software engineering that advocates constructing test cases before writing actual code; indeed, coding is treated as an exercise in validating the test cases. While such an approach appeals to many software developers, one cannot simply apply TDD to component-based software engineering (CBSE). The primary obstacle is the more complex life cycle for software components that must be packaged, deployed and executed within software containers or deployment environments. In this paper we describe two case studies that show different ways by which TDD can be applied to CBSE. Our focus remains on the dependencies that exist between components and how to manage these dependencies during testing to still enable successful unit testing.

1. Introduction

Test Driven Development (TDD) is a software development technique that has gained popularity as of late because of the direct benefit of amortizing the testing effort throughout the entire development cycle (Beck, 2002). The primary contribution of this approach is to require automated tests to be written before any code is designed or added to an existing, working system. Using rapid, brief iterations, developers are able to make immediate progress on satisfying specific test cases designed to test external behavior. Then through repeated refactoring effort, the code structure can be improved, and can always be validated against the existing test cases.

The tight iterative development loop consists of several steps:

1. Add a new test case
2. Run all existing tests and validate that the new test fails
3. Write code to ultimately ensure that the test will succeed
4. Run all existing tests and validate that all succeed
5. Refactor code as necessary, and continue with step 1.

The process as described is agnostic with regard to component technology, except for the presumed ability to run a set of tests. One might adopt the strategy that all test cases are carried out natively on the code (i.e., as Java classes or C code). However, this point of view will not be satisfactory because the component code is expected to execute as demanded by the underlying component model. In fact, you must test the code in a testing environment that most closely matches the execution environment in which the component is to execute.

The problem identified by this paper is that components invariably have dependencies upon other components. While the ideal case is that each component is wholly independent, it is not always practical or possible. The trouble with software components is that the focus is primarily on the ways in which the components are deployed and composed, rather than on the (often mundane) ways by which the component could be tested. We'll use the following definition in this paper.

> A **Software Component** is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard (Heineman & Council, 2001).

Many of the dependencies that a software component has may never be explicitly declared and may only be discovered at assembly time, or sometimes (even worse) at run-time. The challenge for component testers is to be able to properly assemble the run-time structures necessary for the unit testing required. For this paper, we avoid discussing platform dependencies that a component may have (i.e., it may properly execute using JDK 1.6 but not JDK 1.5) and focus solely on inter-component dependencies.

There are two possible flavors of inter-component dependencies: **concrete** dependencies on other components and **abstract** dependencies on an interface provided by another component. In this paper we present case studies to explore the challenges faced by unit testers having to deal with both of these flavors. A concrete dependency exists when a component makes direct reference to functionality provided by another component outside of any interface construct; we simulate this issue using the C-based product line case study described in section 2. When an abstract dependency exists, the tester must somehow be able to provide some component that provides the desired interface; we simulate this issue using the CompUnit-based case study described in section 3. Even though component developers strive to minimize these dependencies, it may not be possible to eliminate them together, which leads to problems during testing.

1.1 Mock objects

One of the most common approaches to unit testing with dependencies is to introduce mock objects (Fowler, 2007) that have clear expectations of the calls they are to receive. One of the more popular frameworks to support Mock objects is JMock (http://www.jmock.org) The obvious extension is to introduce mock components, yet these components must then also be packaged, deployed, installed and assembled into test applications. Since components must execute within an assembly, you need to prepare a full run-time infrastructure to execute the components. Additionally, whereas it is possible to simply construct mock objects, using standard class constructors, mock components require a larger amount of scaffolding to complete.

1.2 Software Component Life Cycle

Kung-Kiu Lau (2007) has described an ideal component life-cycle, to identify opportunities for reuse both within component design and component deployment phases. In his view components exist within a component repository during the design phase. Components can be composed with other components to form larger components stored during design or component assemblies during deployment. In the final run-time phase, a run-time infrastructure executes the constructed component assemblies. We consider any testing during this final phase as *integration testing*, so we restrict our attention to the type of testing one might carry out during component design and component development.

The components in the component repository must be independently tested using a unit testing strategy. However, this requirement is challenged by the inter-component dependencies that invariably exist within systems decomposed from components. One must be a bit more careful during design and when developing components, as we discuss in the paper.

1.3 Requirements

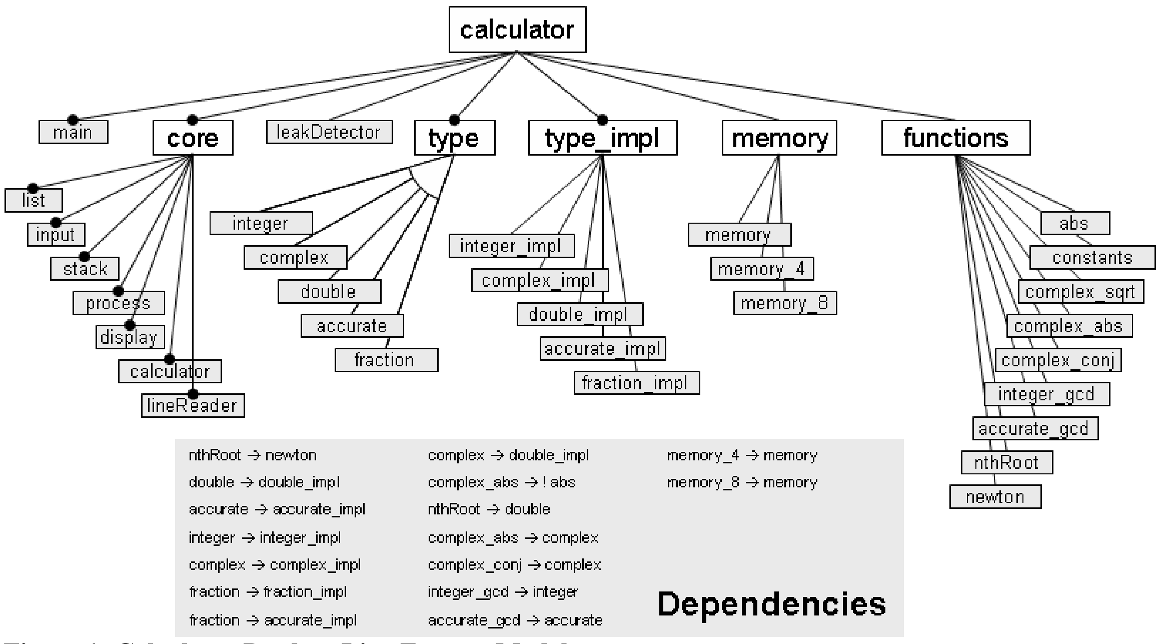
Because we had in mind two separate case studies, with different technologies, we defined a set of requirements to guide our effort so we could normalize our results.

- Test cases must be defined separately from the component under test – without such separation, one would be required to repackage and re-deploy software components whenever new test cases.
- A testing framework must be defined separately from the test cases – we must be able to support different testing frameworks, such as JUnit (http://www.junit.org), or home-brewed techniques.
- Testing an individual component must not depend on having all components for the final application – it must be possible to truly test each component in isolation from other components; where necessary, *mock components* are to be written to substitute for an interface dependency.

Our solutions must also reduce as much as possible the manual human element of testing and support the greatest amount of automation. Clearly there is more work to be done to support this principle; in this paper we focus our attention on the "bottom-up" issues faced by component unit testers.

2. Case Study: Product Line Structure

We created a calculator product line composed of features that one might envision having in a hand-held calculator. The Feature Model shown in Figure 1 captures the various features of this product line using the Czarnecki notation (Czarnecki and Wasowski, 2007).



nthRoot → newton
double → double_impl
accurate → accurate_impl
integer → integer_impl
complex → complex_impl
fraction → fraction_impl
fraction → accurate_impl
complex → double_impl
complex_abs → ! abs
nthRoot → double
complex_abs → complex
complex_conj → complex
integer_gcd → integer
accurate_gcd → accurate
memory_4 → memory
memory_8 → memory

Figure 1: Calculator Product Line Feature Model

Each gray box represents a feature that is encapsulated as a *feature component* (or component for short). That is, each component has its own source files and can be independently compiled. Features with a black dot at the top are mandatory. The larger white boxes represent feature "families", some of which are also mandatory. For example, the **type** family is mandatory, specifying that one of its child features must be selected. The white "arc" emanating from the **type** node declares that only one of the children is to be included (i.e., XOR functionality) in a product line member. Optional features have no black dot at the top of their box. Dependencies between features are captured declaratively and are provided in the large gray box at the bottom of the figure. For example, the **nthRoot** feature depends upon having **newton**'s method available and that the selected **type** for the calculator is **double**.

The product line was implemented in C. The primary goal of this case study was to demonstrate that one could selectively mix and match desired features in a product line by simply declaring the desired features. Because the C programming language offered no capabilities to support a product line, we engineered a set of constructs and processes to make this happen. This somewhat-academic exercise was intended as a proof of concept, to show that one could devise component models even when the underlying programming language offered no support.

```
include MakefileCommon
MODULE  = integer_gcd
SRCS    = integer_gcd.c
DEP_MOD = integer_type.c integer_type_impl.c

all: $(MODULE).a

$(MODULE).a: $(SRCS:.c=.o)
        rm -f $(MODULE).a
        ar rv $(MODULE).a $(SRCS:.c=.o)

CORE = input.c process.c calculator.c display.c lineReader.c list.c stack.c
```

```
# run tests: Must pre-link in required features (if exist)
test: $(SRCS:.c=.test.o)
        ./ut $(TEST_REMOVE) $(SRCS:.c=.test.c) $(SRCS) $(CORE) $(DEP_MOD) >> $(PL_TEST)

# invoked by product line architecture
initialization:
        @echo $(LIBS)              >> $(FINAL_LIBS)
        @echo "$(MODULE)_init(); " >> $(INITIALIZATION)

# clean away the code (including test files)
clean:
        rm -f $(MODULE).a $(SRCS:.c=.o)
        rm -f $(SRCS:.c=.test.gcov) $(SRCS:.c=.test.gcda)
        rm -f $(SRCS:.c=.test.gcno) $(SRCS:.c=.c.gcov)
        rm -f $(DEP_MOD.c=.test.gcov) $(DEP_MOD:.c=.test.gcda)
        rm -f $(DEP_MOD:.c=.gcno) $(DEP_MOD.c=.c.gcov)
```

Figure 2: integer_gcd.Makefile

2.1 integer_gcd component

It is instructive to show the full details of one of the simpler components. The full case study can be retrieved via the URL provided at the end of this paper. Each feature is implemented in its own separate set of C files and is compiled and built by its own Makefile. We intentionally chose to use Makefile specifications rather than design a separate language that captures the same information (such as an XML representation). Indeed, the intent of the Makefile is to produce a single executable `integer_gcd.a` file which can be independently deployed during application assembly. The **integer_gcd** Makefile shown in Figure 2 defines how to build the component, which simply involves compiling the `integer_gcd.c` source file. However, this Makefile specification also shows how to execute the unit tests for this component (the target "test" in the Makefile).

A collection of Makefiles are used to build individual features as well as to build an entire product line application member. In this example, the core set of features is defined as the baseline application. That is, no such application member in this product line can be constructed without this base. To describe a potential product line member, then, one need only specify the set of features within the global Makefile. Figure 3 shows some sample product line specifications.

Description	**Definition**
Simplest calculator supporting just basic *, –, + and ÷ over doubles	`double_type.c` `double_type_impl.c`
Calculator using complex numbers and supporting small set of complex operators (conjugate, absolute value) as well as a bank of 4 memory registers	`complex_type.c` `complex_type_impl.c` `double_type_impl.c` `complex_sqrt.c` `complex_abs.c` `complex_conj.c memory.c` `memory_4.c`
Calculator supporting arbitrary-precision accurate integer arithmetic, a bank of memory registers (defaults to 8), some pre-defined constants, and the greatest common divisor function	`memory.c constants.c` `accurate_type.c` `accurate_type_impl.c` `accurate_gcd.c`

Figure 3: Sample Product Line Member Applications

We were successful in this effort. We then wondered how we could add unit testing to the underlying development process. Since each component was implemented with its own files, we had to clearly declare its dependencies within its Makefile (note the `DEP_MOD` variable in Figure 2). Since the intent is to construct an executable whose purpose is to execute the test cases specified within `integer_gcd.test.c`, we must be able to construct an executable, so all concrete dependencies for the **integer_gcd** component are realized. In this example we use the actual components themselves but this could easily have been rewritten to use mock components.

The result is that each component can be independently built (using `make -f component.Makefile`) and independently tested (using `make -f component.Makefile test`). We crafted a unit test utility, **ut,** (referred to in the Makefile) to carry out the unit tests by replicating much of the functionality as specified by JUnit. The `integer_gcd.test.c` source file contains test cases as shown in Figure 4. While some of the details are unnecessary, one can readily see the use of `testXXX()` functions to represent test cases and `setUp()` and `tearDown()` functions as supported by JUnit. **ut** generates the requisite driver code that launches the four test cases as defined, bracketing these invocations with calls to set up and tear down resources as required.

```c
#include "calculator.h"
#include "process.h"
#include "integer_gcd.h"
#include "integer_type.h"

#include "ut.h"

/** Useful variables for test cases. */
static CALCULATOR_PTR calc;
static TYPE_PTR       at;
static TYPE_PTR       bt;
static INTEGER_PTR    ai;
static INTEGER_PTR    bi;

/** Useful test macro */
#define localCheck(expected,tp)     \
{                                   \
  assertTrue ((tp) != NULL);        \
  assertEquals ((expected),         \
((INTEGER_PTR)(tp)->inner)->n);     \
  freeType ((tp));                  \
}

/** allocate resources for each test. */
void setUp() {
  calc = constructCalc();

  at = newType();
  ai = at->inner;
  bt = newType();
  bi = bt->inner;

  /* initialize module under test. */
  integer_gcd_init();
}
```

```c
/** release resources. */
void tearDown() {
  freeCalc (calc);
  freeType (at);
  freeType (bt);
}

/* this is now a binary operator */
void testisOperator() {
  assertEquals (1, isBinary("gcd"));
}

/* test application */
void testGCD() {
  ai->n = 117;
  bi->n = 13;
  localCheck (
     13, applyGCD (calc, "gcd", at, bt));
}

void testGCD2() {
  ai->n = 1;
  bi->n = 1;
  localCheck (
      1, applyGCD (calc, "gcd", at, bt));
}

void testGCD3() {
  ai->n = 14;
  bi->n = 0;
  localCheck (
     14, applyGCD (calc, "gcd", at, bt));

  localCheck (
     14, applyGCD (calc, "gcd", bt, at));
}
```

Figure 4: Sample Test cases for integer_gcd

To complete this case study, the primary Makefile for assembling product line application members was modified to also test the features used within the product line by repeatedly invoking `make -f feature.Makefile test` on all of the selected features. **ut** uses `gprof` (the Unix utility for call graph profile data) and `gcov` (the Unix coverage testing tool) to generate reports showing the code coverage of the test cases, as well as identifying those which failed.

One of the lessons learned from this C-based case study is that the testing of individual features did depend upon having a fully working base. There was no easy way to eliminate the dependency that a feature component has on the base. A corollary of this lesson was the observation that the base had to be tested as a single unit because of the deep interconnections between the requisite C files that made up the base. See Muccini and van der Hoek (2003) for ideas on testing product lines. Nonetheless, each feature can be tested independently by identifying the dependencies of the feature in its Makefile. Another lesson learned was that the unit testing was actually quite effective when using the actual components themselves, rather than stub or mock objects. The reason was the structure of the product line specified a clear tree-like set of dependencies between the feature components, thus it was possible to test small subsets of features first before expanding up to unit test features that depended upon larger collections of features.

3. Case Study: Component-Based Structure

In our second case study, we create a small CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) utility that involved a client/server system. On the client-side, the user is challenged to identify words in a moving image, and the server processes the messages sent by the user; should the words match, a new user account would be created for the user in a database.

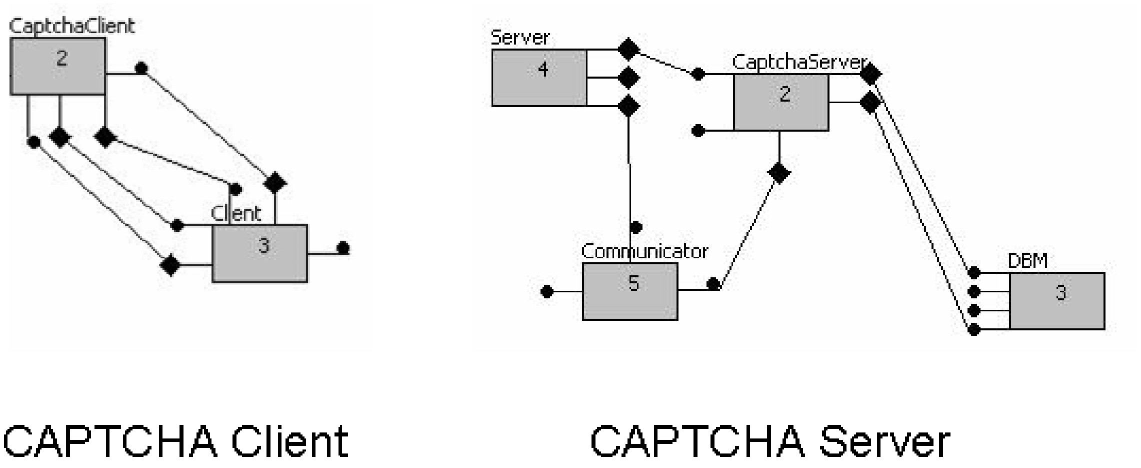


Figure 5: CAPTCHA application

To build the application, we used the open source CompUnit (Heineman, 2009) component model, which has been developed to properly teach issues regarding CBSE at both the undergraduate and graduate level. All components are written in Java and conform to an interaction standard where each component is able to interact with other components only through well-defined interfaces. In short, a component can provide (or otherwise *implement*) an interface and that component can be connected to another component that requires the functionality as defined by that interface. CompUnit components are assembled into applications by connecting components to each other using these defined interfaces. There is a set of tools to help developers package their CompUnit components into stand-alone JAR files that contain the encapsulated implementation; one can also assemble applications using a graphical editor. An application consisting of CompUnit components executes within a run-time environment container called Foundation.

Each component in Figure 5 is represented by a rectangle. A component may provide a set of services (identified by the "lollipop" handles emanating from the components) and may require services (identified by the lines with diamonds). Components can communicate directly with other components only through such interfaces. The primary modeling novelty of CompUnit is that each component must clearly identify (with meta data) the interfaces which it requires to perform its functionality. CompUnit assumes that each interface, once published, becomes immutable, which ensures the long-term interoperability of components that require and provide the same interface.

Each component is independently built, packaged and installed into a CompUnit environment and then an application is defined by assembling the components together; in Figure 5, there are two applications. The challenge for the unit tester is to find some way to test the **CaptchaServer** component even though it has three dependent interfaces (one on the **Communicator** and two on the **DBM** component).

We approach this concrete dependency by constructing a mock component to aid the effort. The challenge, naturally, is for the tester to be able to execute the **CaptchaServer** component. Towards this end, we developed a **SuiteRunner** component, whose purpose in life was to manage the JUnit test cases that were to be separately written. The final application assembly is shown in Figure 6.

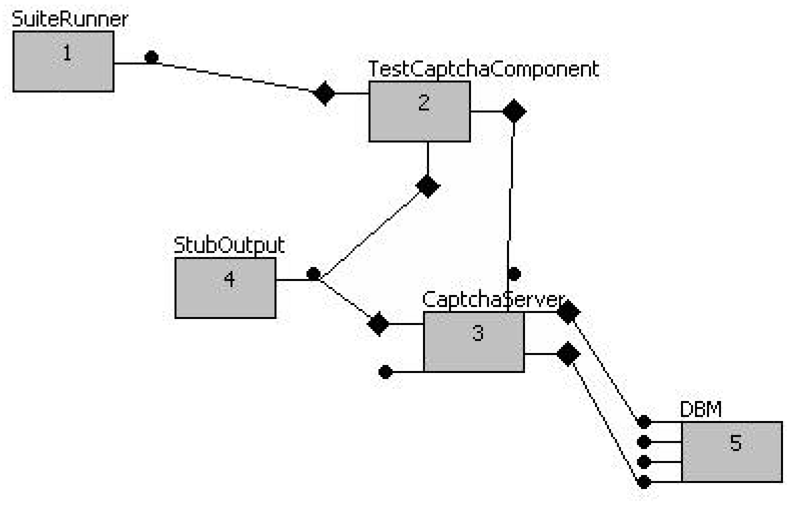


Figure 6: Assembly to test CaptchaServer

SuiteRunner takes over the responsibility of launching the JUnit test cases that are packaged within the **TestCaptchaComponent**. In this way, each component under test can have its own **TestXXX** component that is connected to the common testing infrastructure. Other helper components are written as needed, such as the **StubOutput** component whose sole purpose is to receive responses back from the **CaptchaServer** and enable the **TestCaptchaComponent** to validate the response is as expected.

Each CompUnit component is packaged into a JAR file by a CompUnit utility known as the "Packager" and then installed into a CompUnitEnvironment using the "Installer" utility. Figures 5 and 6 show screenshot captures of the CompUnit utility, "Café" that allows users to graphically construct application assemblies. Each of these CompUnit utilities is actually implemented using CompUnit components.

If one were to truly follow the TDD strategy outlined in the introduction of this paper, then each new test case would require the repackaging and redeployment of a component. While such a process would be developmentally sound, it leads to gross inefficiencies, which is why the case study was carried out entirely within Eclipse (http://www.eclipse.org).

3.1 Extended support provided by Eclipse

Many developers have grown accustomed to the powerful support that Integrated Development Environments (IDEs) such as provided by Eclipse. For example, one can develop web services without ever leaving Eclipse (Eclipse, 2009). Clearly such capabilities reduces the effort in developing these services by reducing the overhead of having to package and deploy the requisite code "natively" as required by the various Web-based protocols and Web servers. Much of the development within the CompUnit case study was performed within Eclipse, and we were able to bypass two key phases of the component-based development life-cycle. In particular:

- Installation – once a component was installed into the CompUnit container, there was no need to reinstall it whenever changes were made to the component. This was made possible because the user can set the CLASSPATH within Eclipse to include to the component under development. Thus we only needed to install the component once, and this typically was done when the first few lines of code were written for the component.
- Packaging for deployment – since the component only needed to be installed once, it meant that it only needed to be properly packaged once, just prior to this installation.

Another important productivity enhancer is the ability in Eclipse to allow developers to change code while the system is being debugged (Holzner, 2004). Under most circumstances, Eclipse is able to "rewind" the computation back to the beginning of the method (or earlier depending upon the call stack). As the components are executing, the software engineer can set breakpoints and view the step-by-step execution of the component, rewriting the code as necessary when defects are detected.

These productivity enhancers are not limited to CompUnit; indeed, it must not be the case! Fortunately, the leading IDE vendors for Java (NetBeans and Eclipse) provide various ways to productively test component technologies, such as EJB, servlets, and web services, just to name a few.

Related Work

The field of software testing is vast and cannot be captured in a single paper. We focus our attention on the most closely-related efforts for testing software components which has been explored by various researchers within the CBSE symposium series over the years (Jalote et al., 2006) (Tyler and Soundarajan, 2004) (Gao, 2000). In general, these researchers focused on specific techniques for testing, rather than the complications arising from interdepdencies.

Built-in test (BIT) component capability enables the black-box testing of components through fine-grained decorator “wrappers” that enable assertions to be checked as the component executes (Edwards, 2001). Edwards describes a framework that fully automates the process of testing components, including generating the test data and the drivers that execute the components. To incorporate BIT components into a test-driven process, the developer would describe the pre- and post-conditions using the contract-based approach as popularized by Bertrand Meyer (Meyer, 1997). From these contracts, the code to execute the test cases would be generated. Nothing in the wrappers is able to address component dependencies, however.

Throughout the paper we referenced various projects (JMock, Cactus, JunitEE) whose purpose is to enable unit testing of components developed using various technologies. The unit testing supported by these projects is still complicated by the dependencies that invariably exist between components. The ideas presented in our paper can be used to guide these technologies to handle inter-component dependencies.

Conclusion

Unit testing of software components is hard enough without having to deal with the added complications of inter-component dependencies. We constructed two case studies that showed how to address the issue. When component dependencies are concrete, one strategy is to assemble component “sub-assemblies” that enable the construction of an application with the component under test. Should the dependency relationship be cyclic, then the only recourse is to develop mock components using the same component model and define assemblies with the component under test. When abstract dependencies are present, one has greater flexibility in whether to choose actual components or to develop mock components in their place. In both cases, the success of the unit testing is made possible by applying the right tool support and infrastructure to automate the code tests. The full calculator product line can be retrieved from http://web.cs.wpi.edu/~heineman/CBSE2009/CalculatorProductLine.zip.

References

Apache, Cactus Test Framework, http://jakarta.apache.org/cactus, 2009.

Beck, K., *Test Driven Development: By Example*, Addison-Wesley Longman, 2002

Czarnecki, K. and Wasowski, A.. "Feature Diagrams and Logics: There and Back Again", proceedings of the 11th International Software Product Line Conference (SPLC), 2007, pp. 23 – 34.

Eclipse Foundation, Web Tool Platform (WTP) project, http://www.eclipse.org/webtools, 2009.

Edwards, S. H., "Framework for Practical, Automated Black-Box Testing of Component-Based Software", *Software Testing, Verification and Reliability*, 11(2), 2001.

Fowler, M. "Mocks aren't stubs", January 2007, http://martinfowler.com/articles/mocksArentStubs.html

Gao, J., "Component Testability and Component Testing Challenges", Component-Based Software Engineering Workshop, 2000.

Heineman, G. "CompUnit Component model", http://sourceforge.net/projects/compunit, 2009.

Heineman, G. and Council, W, *Component-Based Software Engineering: Putting the pieces together*, Addison-Wesley, 2001.

Holzner, S., *Eclipse Cookbook*, O'Reilly Media Inc., 2004.

Jalote, P., Munshi, R., Probsting, T., "Components Have Test Buddies", Component-Based Software Engineering Symposium, June 2006, pp. 310—319.

Janzen, D., Saiedian, H., "Test-driven development concepts, taxonomy, and future direction", IEEE Computer, 38(9), 2005, pp. 43 – 50.

JUnitEE, http://www.junitee.org, 2009.

Lau, K-K. and Wang, Z., "Software component models", *IEEE Transactions on Software Engineering*, 33(10), pp. 709 – 724, October 2007.

Meyer B., *Object-Oriented Software Construction*, 2nd Edition, Prentice Hall, 1997.

Muccini, H. and van der Hoek, A., "Towards Testing Product Line Architectures", International Workshop on Test and Analysis of Component-Based Systems (TACoS), 82(6), 2003, pp. 99-109.

Muthu R., “Testing Software Components Using Boundary Value Analysis”, proceedings of the 29th EUROMICRO conference "New Waves in System Architecture", 2003.

Pavlova, I., Akerholm, M., Fredriksson, J., “Application of built-in-testing in component-based embedded systems”, ROSATEA 2006, pp. 51—52.

Tyler, B. and Soundarajan, N., “Testing Framework Components”, Component-Based Software Engineering Workshop, May 2004, pp. 138-145.