2009 International Conference on Software Testing Verification and Validation

Practicing Testability in the Real World

Vishal Chowdhary
Microsoft Corporation
vishalc@microsoft.com

Abstract

There have been studies both about the concept of testability and how to implement it, for example using test hooks. In spite of its obvious benefits, we still find low adoption of testability concepts. For example, we do not see a lot of bugs requesting a change in a feature because it will become simpler to test it. In this article we present our experiences while applying testability concepts and lay down guidelines to ensure testability consideration during feature planning and design.

We lay the groundwork by briefly going over testability concepts. We talk about the typical thought process in a test developer's mind and present key insights into why practicing testability is hard.

We then present real life examples that we have encountered in our career which exhibit how testability considerations could have made our testing simpler. We discuss the impact of testability on design. Building from these examples, we present a checklist for each of the testability principles. Going over this testability checklist helps ensure we think about the core testability principles when testing a particular feature and ultimately simplifies testing and reducing overall costs.

We finally present an exercise where we can apply our checklist to a complex event processing component and realize the benefits of applying testability concepts.

1. Introduction

Testability is a core concept in testing as fundamental as code coverage and white box testing. It is defined as the degree to which components and systems are designed and implemented such that it is relatively easy to test them and increase the efficiency of the testing process which in turn makes defects more discoverable for test automation and doing this in a cost-efficient manner. Put simply, it helps to make testing components simpler.

There have been past research that helps understand the testability concept and tools (e.g. test hooks [4], [9] in product code, TypeMock [5]) that help implement testability concepts. This article is neither about testability concepts, (although it does go over them briefly), nor is it about how to implement testability. This article focuses on why it is so difficult to practice testability in the real world and what can be done about it.

The rest of the article is organized as follows. We start with summarizing basic testability concepts and its benefits in Section 2. In Section 3, we discuss implementing testability and the challenges in doing so. In Section 4, with the aid of real life testing examples, we present a testability checklist for the testability concepts. In Section 5 we cover advanced testability considerations and in Section 6, we discuss the impact of testability on design. We apply the testability checklists to our motivating example in section 7, present an exercise to the reader in Section 8 and conclude the article in Section 9.

2. Testability: Concepts and Benefits.

Testability is a measure of the degree to which a system or component facilitates the execution of tests against it. Numerous studies [1], [2] have explained the concepts in detail with various examples. In [3], the author summarizes the most fundamental testability concepts through the SOCK model. In this section we briefly go over testability concepts using the SOCK model for testability as a reference. The key benefits of considering testability are also listed.

2.1 SOCK Model

Testability can be broken down into several individual principles. The SOCK model invented by Dave Catlett [3] is a good way to think about this. The basic principles of the SOCK model are:

978-0-7695-3601-9/09 $25.00 © 2009 IEEE
DOI 10.1109/ICST.2009.53

1. Simplicity: The simpler a component, the less expensive it is to test
2. Observability: Exposing state (visibility and transparency)
3. Control: Can you exercise every nook and cranny of the component?
4. Knowledge of expected results: Is the observed behavior correct?

Improving each principle of SOCK improves testability. The SOCK model ensures consideration of all aspects of testability.

2.2 Testability Benefits

Testing is expensive in terms of people and equipment. We always seem to be short on test resources. While developing tests for large scale commercial products we quite often find ourselves writing a parallel product to test another. Manual testing doesn't scale and hence we need to write a lot of automation. For companies like Microsoft where we need to support a product for 7~10 years, this automation needs to last the lifetime (Sustained Engineering) of the product and hence needs to be reliable and maintainable. It's this automation that gets run during a Quick Fix Engineering (QFE is a bug fix to address a specific customer request) or Service Pack (SP is typically a collection of QFEs and enhancements) so that the product team is not disrupted when it is working on the next version of the same product. The following are some of the key benefits of testability [3]:

1. Reduces the cost of testing in terms of time and resources
2. Reduces the time to diagnose unexpected behavior
3. Improves manageability and supportability of the software
4. Provides the building blocks for self-diagnosing and self-correcting software

3. Implementing Testability

The testability concept has been around for a decade and it has been gaining a lot of momentum in the last couple of years. When you read about it, you absolutely feel that it should be the first thing you should be thinking about when coming up with the test plan for a particular feature. But what we have observed is that people seldom think about it. We do not think too often about how to write simpler tests. That is a question we do not ask very often. Why is that the case? Why is it when we are asked to do a rough analysis of the test versus development cost for a feature, we usually estimate it takes 50 to 100% more time to test? How many times have we encountered a scenario where a feature needs to be cut or scaled down since it is too expensive to test? You wonder why we never think about making our tests simpler thus reducing the test costs. Note the intention is not making tests simple but simpler! For example the CLR JIT compiler (Microsoft .NET Just in time compiler) or the SQL Server optimization engine are complicated products hence those tests are going to be complicated as well. We do put in effort in making our tests robust given their longevity requirements. However we tend to spend more time on writing code to test the product than actually testing the products!

We need to write tests that are 'relatively' simple and thus cheap to implement. Note that the development cost of implementing a feature may increase when you put additional testability requirements. For example, you may want to make the internal state of a component more observable by writing to a trace file. This requires more product code to trace the details of the internal state. There are several considerations one needs to think about when asking for a change:

1. This may indirectly assist in the debug-ability and maintainability of the software.
2. We always need to think of the overall costs of the feature team (Feature team = comprised of all members that are part of engineering the component, typically developers, testers and program managers). Let's say the total cost for engineering a feature is 10 developer days and 20 test days for a total of 30 days. If by spending two more developer days you can actually eliminate five test days then you will see an overall reduction of three engineering days for the entire feature.
3. There are scenarios where you do not wish to design for testability; for example, security and performance are often at odds with testability. Section 6 talks about this in more detail.

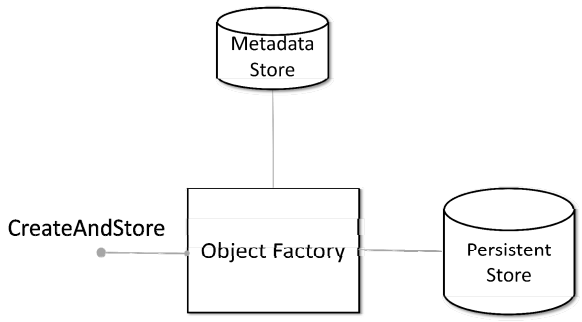
3.1 The Test Developer Role

To answer the questions we raised in the previous section, it is important to talk about and understand the psyche of a test developer. Very few schools and colleges offer any testing courses. Thus none of us usually have any formal testing education or training when we start working as test developers. We have been taught to design and implement complex software projects. Testing has usually been something we do at the end of the project to ensure that our software passes the professor's tests in order to get the necessary

grades. Frankly speaking, testing has usually been an afterthought in our minds. For almost all of us, we have never even heard of the test developer role. Hence our basic thought process is more often than not inclined towards a developer mentality. Typically a complex feature (in terms of functionality) leads to us developing a complex test framework. We tend to like the challenge in automating tests for components which are difficult to test. We may not consider that this testing challenge may actually be due to poor design [6].

In the following section we present a concrete example to explain this insight. Similar to other examples later in this article, this scenario was encountered while testing one of the features.

3.2 Motivating Example



We take the example of a scenario where we need to write a functional test for an asynchronous object serialization API (application programming interface) that saves the object to a persistent store. The API exposed for this operation takes in parameter values that control the object creation and then serializes the object to the data store. It gets the persistent store information from the metadata store. The metadata store is created during software installation. The following is a simple functional test case that creates the object for a set of parameters and validates the persistent store.

```
bool validated = false;
//create object based on these parameters
int a = 2, b = -1;
CreateAndStore(a, b);
validated = ValidatePersistentStore();
if(validated)
   Console.Write("Test passed");
else
   Console.Write("Test failed");
```

There are a couple of problems with the API design that are apparent from the test case code:

1. Weak cohesion in the `CreateAndStore` procedure since it performs two separate and unrelated (create and store operations) in the same method call.
2. In case of actual failure, there is no way to find out whether the creation or the storage of the object failed.
3. The procedure name `CreateAndStore` does not indicate it is asynchronous.
4. The store operation is asynchronously invoked on a separate thread and hence may not have completed before the test verification.
5. Which database data store, the metadata or persistent store, is the object getting serialized into?

Let's try to tackle the problem of validating the asynchronous object serialization code. If you simply invoke the command and try to check whether the object has been serialized, the test may fail since the object creation and serialization operation may take some time. A simple solution is to introduce a "dumb sleep" proportionate to the database write operation. The test invokes the command, sleeps for five seconds, and then verifies the object serialization has finished successfully.

```
bool validated = false;
//create object based on these parameters
int a = 2, b = -1;
CreateAndStore(a, b);
Thread.Sleep(5);
validated = ValidatePersistentStore();
if(validated)
   Console.Write("Test passed");
else
   Console.Write("Test failed");
```

This is also not a guaranteed solution since you may test this functionality on a machine with low CPU and memory resources where the data store operation takes significantly more time. Most of us will then implement a "smart sleep", a technique whereby the test does the following:

```
int maxTimeout = 60;
int currentTimeout = 0;
bool validated = false;
//create object based on these parameters
int a = 2, b = -1;
CreateAndStore(a, b);
while (currentTimeout < maxTimeout)
{
   validated = ValidatePersistentStore();
   if (!validated)
   {
      Thread.Sleep(5);
      //retry the validation after 5 seconds
```

```
        currentTimeout += 5;
    }
    else
    {
        break;
    }
}
if(validated)
    Console.Write("Test passed");
else
    Console.Write("Test failed");
```

Here the test sleeps for regular intervals of 5 seconds checking whether the object serialization operation has completed. This solution, although being more resilient, is still not ideal. Five years down the line the team may decide to optimize machine resources by running all tests in a virtualized environment. The low resources with virtualization make the operation take more than a minute. The test case timeout is usually external and is governed by the test case manager running the test and can be adjusted depending on the platform and is independent of the actual test case. The test owner may try to increase the timeout of all the test cases through the test case manager on this platform, but the internal timeout value of 60 seconds for this particular test case will prevent that increase and the test may start failing.

In the ideal case, the functional test is independent of the time taken to complete the serialization operation. The test simply waits for notification when the CreateAndStore has been completed and then does the validation. The notification could be an event or trace that is emitted by the object serialization code.

```
//create object based on these parameters
int a = 2, b = -1;
CreateAndStore(a, b);
BeginValidatePersistentStore(OnValidatePersist
entStore);
void OnValidatePersistentStore ()
{
    if (ValidatePersistentStore())
        Console.Write("Test passed");
    else
        Console.Write("Test failed");
}
```

This is a lot of work to get around bad design. It forces the test code to be more complex than it should be. This illustrates a key point that during the design reviews for the CreateAndStore API, the simple question "How are we going to test this?" should have come up. It has been noticed when the dev team is writing their own unit tests, the feature tends to be more testable. In this way, testing clearly influences design. We talk about this more in Section 6.

The goal should be making test case writing simpler, which leads to the bigger goal of improving the quality of the product in a cost-effective manner.

4. Testability Checklists

In this section, we present checklists for testability principles. Going over these checklists during product design and implementation will ensure that basic testability principles have been taken into consideration. For each of the SOCK principles, we present real life situations that exhibit how testability considerations could have made our testing simpler. Building from these examples, we present the checklist for that particular SOCK principle.

4.1 Simplicity (S)

A simple design will reduce the test setup time and ensure correct cleanup for the tests. This greatly enhances testability of the software. The software should be architected to ensure that it follows basic good design principles – low redundancy, loose coupling and strong cohesion, and follow established design patterns (Bridge, Composite and others [6]). The number and type of dependencies also can greatly increase the complexity of testing as well. The main criterion for judging the simplicity quotient is "How simple is it to execute the actual test for this particular component?"

4.1.1. Real Life Experiences (S)

A. Thick UI

Consider a product which consists of a local client application to access data from a database. In the first version, the developer designed a thick client with the database access code embedded in the presentation layer. For the next release, the user has demanded a programmatic set of APIs to access the same data. Due to lack of time and resource constraints, the developer ends up copying the application logic code from the thick UI client from the first version and wraps it in a set of APIs. The product now has two copies of the same code. The code-base has thus become difficult to maintain since code fixes will need to be applied to two places. The test cost has also doubled. Furthermore, it violates the Model-View-Controller design pattern which stresses separation of the business logic from the UI and the data store.

B. Weak Cohesion

Consider a component that creates a representation of an object Foo from a certain set of input parameters. The developer realizes that this object Foo will be eventually saved to a database. Hence, he writes the database interaction logic in the same method that creates the object. Now, to test the component that is responsible for creating the object Foo, you also need

to create and query a database to ensure that the creation logic is correct. For the next release, the user has requested to save the object to a file instead of a database. This unnecessarily causes the original set of tests which intended to test the Foo object creation logic to be changed since they now need to read a file instead of a database to validate the Foo object creation.

C. FileStoreAccess()

The user's requirements ask for a new set of APIs for programmatic interface to the objects that are serialized and stored in the software's proprietary file store (*FileStoreAccess* motivating example). To initialize a new instance of the *FileStoreAccess* class the user needs to pass in the location of the file store to its constructor. The API is designed such that for the default constructor that takes in no arguments, it reads this location information either from the SQL management database or from the application's configuration file. Which one would you prefer assuming all of them are equally secure? Consider we chose to store this information in the SQL management database. In order to test the *FileStoreAccess()* APIs we need to setup the SQL management database so the file storage location can be retrieved. Thus there is additional cost to have SQL installed on the test automation platform, setup the management database, insert the data and simulate error conditions that the *FileStoreAccess()* constructor may have to handle during database access. Consider this approach versus storing the location of the file store in the registry or a simple application configuration file wherein the test case setup simplifies a lot. It's easier to update the configuration file or the repository than to setup a database server and the management database. Thus, the test case setup can avoid tedious setup of the SQL management database.

4.1.2. Checklist (S)

1. Does it take you a considerable amount of time and code to "setup" before testing your actual component and "cleanup" later?
2. Does your test fail intermittently due to a component external to the component that you are testing?
3. Could you make your test execution simpler and easier by directly working with the component?
4. Can your component be initialized or started in isolation by itself?
5. Can you exactly point out what failed?
6. Do you write redundant test cases for the same functionality?
7. Does your component follow established design patterns?
8. Does your component rely on complicated dependent software for data storage, network communication, setup or configuration?

4.2 Observability (O)

Observability is the degree to which one can effectively analyze the execution results on both the feature component under test and the overall system in order to accurately determine whether the test succeeded or failed. The more easily and deterministically one can programmatically query and determine the state of the system, the more accurate and resilient the analysis phase of a test will be. In addition, the cost of automating the test case decreases with the ease of observability. The main criterion for judging the observability quotient is: "Can one programmatically easily access all outputs to deterministically know whether the component passed the test?"

4.2.1. Real Life Experiences (O)

A. Internal memory (hashtable)

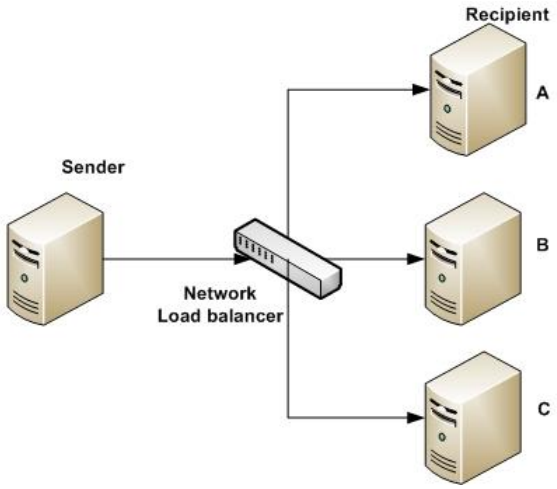
Consider a feature involving an internal memory table of a router containing unique identifiers of messages and their destinations. The key-value pairs stored in the router are of type *MessageIdentifier* and *DestinationIdentifier*. The component supports two operations, an `Update()` which lets the user update the internal routing table and a `Submit()` which lets the user submit messages to the routing component. To test the `Update()` function atomically, one needs to be able to query the internal memory table state before and after the update operation. The component can provide the querying capability by providing a test interface that let's query the routing table's state or by emitting debug traces that contain the routing table's content before and after the `Update()` operation.

B. Lossy channel

Consider that the object serialization component from Section 3 writes initially to an internal buffer and after the buffer reaches a particular object count threshold value it writes the objects to the persistent store. If the persistent store is unavailable and it fails to write to the store, then there is a maximum buffer size to which the internal buffer will grow before it drops the new objects. It is difficult to test this functionality since the test needs to verify that once the maximum buffer size is reached the serialization component drops this message and this is not written to the persistent store if the store becomes available at a later period in time. To enable the testing of this feature, the serialization component can raise an event when it drops the objects

from its buffer. The test can now subscribe to this event and accurately determine when an object has been lost from the buffer.

C. Load balancer



Consider a network load balancing (NLB) component that routes incoming HTTP web requests to one of many back-end web servers. It makes a routing decision through the load balancing algorithm which involves both configuration time and various runtime properties on the web servers which are constantly changing. If multiple web servers are connected to the NLB, when a web request comes and is routed to one of web servers, the test needs to determine whether the NLB component has taken the correct decision to route the web request. For the test to validate the decision it needs knowledge of the runtime properties that affects the load balancing algorithm's output. Since these properties are constantly changing, it's impertinent that the NLB component traces the values it used while determining the outcome of the algorithm.

4.2.2. Checklist (O)

1. Can you write simple code that easily verifies the result of your test execution?
2. Can you programmatically detect any change in the component state? Is there any component state that is internal and cannot be viewed by your tests?
3. In case of multiple options for the same input, can you easily observe which particular option has been exercised?
4. Can you capture unexpected errors, warnings and exceptions in the software?
5. Can you easily analyze the test execution result to determine whether your test has passed or failed?

4.3 Control (C)

Control can be defined as the degree to which you can control the state of the component and system such that all states (code paths) in the component can be easily reached. The easier it is to control, the less expensive and more reliable the test case will be. To measure the controllability quotient, ask the following question: "What hooks do we need to reduce the cost of programmatically controlling the component?"

4.3.1 Real Life Experiences (C)

A. DataAccessor

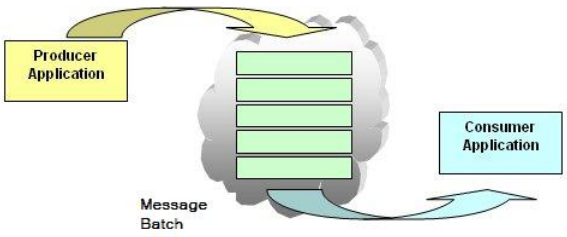
A Data Accessor component provides developers programmatic access to information that is stored in a persistent database. It has a single default constructor `DataAccessor()` that initializes a new instance of the data accessor class. It uses information from the windows registry to find out what database to connect to for retrieving the data. Consider testing this component for failure scenarios. To test the condition where the persistent database is in-accessible, the test can quickly update the database information in the registry thus easily controlling the behavior of the database accessor component.

B. Retry on failure

Consider an order processing system that saves all incoming orders to a database. In the case of a failure to write to the database in the first attempt, the system uses built-in retry logic. It tries to write the order for a total of three times, waiting for five minutes between each retry. How will you test this retry logic? A test for this logic will need to run for at least 3 x 5 = 15 minutes. It will also need to validate that the system does exactly three retries each every five minutes. Given this is an important customer scenario; we need to have a unit test for the retry functionality needs to ensure that it works after any change in the code. This will significantly increase the overall running time of your tests thus increasing the time taken to validate any code updates. The system can read these default retry parameters from a configuration file that the test can modify to quickly and efficiently test the retry functionality.

C. Batching messages to optimize transfer.

A local messaging agent running on the same box as an application collects application traces before sending them to the central store on the same machine, or a different machine over the network. To ensure high throughput and to reduce the cost of creating and tearing down the channel, the messaging agent buffers



the traces and sends them in a single batch once 100,000 traces have been collected. We need to test whether this local agent can successfully transmit this batch message to the centralized store. For this we need to generate messages for the running application so that at least 100,000 traces get generated. How does this value affect our test?

4.3.2 Checklist (C)

1. Do you have to wait, sleep, or poll for a condition to happen in the component?
2. Can you deterministically set a condition on the component?
3. Can you manipulate internal settings easily (for example through the configuration file) to make your tests run faster?
4. Can you control the execution path, for example to choose a specific algorithm or generate a specific fault in the component under test?

4.4 Knowledge (K)

To gauge the knowledge quotient for testability, try to answer the question "What happens when …?" for scenarios against the component under test. Do you know a priori what is the outcome of the test or do you first have to run the test and then figure out the expected results? Detailed functional and design specifications, state machines, data flow diagrams and error scenarios are a key requirement to have a high degree of knowledge about the system.

4.4.1. Real Life Experiences (K)

An instance of lack of knowledge about the software component is any time one has executed a test, observed the outcome and then walked to the developer's office to chat about "Should this be the expected behavior?"

4.4.2 Checklist (K)

1. Are all error conditions defined in the functional spec or do you have to first execute and then ponder whether the actual behavior makes sense?
2. Are the results of behavior clearly documented?
3. Are all the magic numbers (thresholds, timeout values, decision formulas) documented with both value and location and how they can be manipulated?
4. Are all default values documented?
5. Are input ranges to API parameters documented?
6. Are the state transitions clearly described in the design document?

5. Extending the SOCK model

The basic intent of testability is to make software easier to test. There are some aspects of testing that are not completely captured by the SOCK model. In this section we extend the earlier checklists by discussing about some of those aspects.

1. Does the software do what it's supposed to do and much more? Is it overdesigned?
2. Does it have extensibility points not required by the user but increasing the test surface area?
3. Does the software have an option for verbose tracing that can capture the complete state during any particular execution that's helpful in case of debugging a failure when in use by the customer?
4. Can we easily pin-point the failure and add the missing test cases?

The basic expectation is to have knowledge about expected results when you run the software against a set of input parameters. However the following information significantly helps is efficient testing of the software

1. What are the critical sections of the software?
2. What scenarios are critical to the customer?
3. What are the areas with least code coverage?
4. What is the piece of code that last changed or changed the most in the last development cycle?

6. Impact of Testability on Design

We should always ask 'How are we going to test this?' during the design of any software system. In this section we discuss what should be the extent of testability's influence on design [7]. It turns out that well designed software is usually high on the testability rank. So should one design for testability?

Any good design keeps the basic principles (open-closed, single responsibility, etc) in mind. Design decisions involve tradeoffs, for example when deciding the balance between simplicity, maintainability, and extensibility.

You will not want to expose the usernames and passwords of all users that are logged into your website at any given time. For increasing performance, you may want to closely tie implementations with the code using them without interfaces and the Factory Pattern [8]. However, these decisions should be taken only when needed. There may be different priorities on what we optimize for code compared to what we optimize for test, but these should be explicitly called out. For example, an extensibility point may reduce readability and slightly increase complexity in some sections of the code in exchange for higher maintainability and lower cost for the tests. During database schema design, the first step is a completely normalized table structure in accordance with the database normalization principles. You may then de-normalize tables to optimize for a particular frequent query pattern. Similarly, one should always design software components in-accordance with testability concepts. Depending on a particular need (security\performance) you may change the design affecting testability but it should always be a conscious decision.

We should strive to optimize the whole software, the codebase with the tests associated with it.

7. Fixing the Motivating Example

Let's apply the testability checklist to the test case from Section 3. Following are some testability issues that become apparent:

1. (S/C) We need to setup the metadata store to test our component. We should be able to fix this if we can explicitly pass the persistent store information while initializing the component.
2. (S) We cannot point out the exact cause of the test failure since the `Create` and `Store` operations in the same method cause it to have weak cohesion. This can be fixed by separating them into separate method calls. (Note this increases the potential test area since one may create any object and try to serialize it and we may need additional validation code in the `Store` operation.)
3. (O) We need to have a mechanism to be notified when the serialization operation is complete. If this is achieved through a generic event raised by the serialization component, then it insulates us from implementing separate notification logic for different store types (database, file, etc).
4. (K) We need to know beforehand what values are supported while creating the object, important customer scenarios and failure conditions while writing to the persistent store.

8. Summary

Software components built with good design principles in mind tend to be highly testable. Although this may at times slightly increase the cost of developing the software, it leads to reducing the cost of testing and decreasing the overall complexity of the software code and tests. How simple, observable, and controllable a component is, as well as how much we know about expected behavior affects the testability quotient of the software. To assign a quantitative value to the testability of software is an area of active research.

The important point is always to remember checking the software system for the basic testability concepts. Keep testability in your mind and always ask "How am I going to test this? [10]" in the design and functional review discussions. The checklists defined in the article along with the real life scenarios should be used to focus on testability principles. At the end, it's all in your mind!

9. Acknowledgements

I am thankful to my mentor Anu Arora for her insight during the initial discussions we had about this article and her constant motivation. I am grateful to David Catlett, Arden White and Brian Rogers for their valuable feedback reviewing the article.

10. References

[1] Bret Pettichord, *Design for Testability*, Pacific Northwest Software Quality Conference, Portland, Oregon, October 2002.

[2] John A. Fodeh, *Automated Testability: The Missing Link in Test Automation*, STARWEST, 2003.

[3] David Catlett, *The Practical Guide to Defect Prevention, Chapter 6, Improving the Testability of Software*, 2007.

[4] Test Hooks == Happy Customers, *http://www.testingreflections.com/node/view/515*, 11/2004

[5] Eli Lopian, http://www.typemock.com/

[6] Alan Shalloway and James Trott, *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley Professional, 2004.

[7] Bret Pettichord, *Design for Testability Presentation*, Software Test Automation Conference, San Jose, March 2001.

[8] Eli Lopian, *Stop Designing for Testability*, Code Project, http://www.codeproject.com/KB/dotnet/StopDesign4Tests.aspx, April 2007.

[9] Lamoreaux, T. Ofori-Kyei, M. Pinone, M. , *A process for improving software testability*, Proceedings. 20th IEEE International Conference on Software Maintenance, September 2004

[10] James Bach, *Heuristics of Software Testability*, 2003.