An Exploratory Study of Component Reliability Using Unit Testing

R. Torkar, S. Mankefors, K. Hansson and A. Jonsson
Dept. of Informatics and Mathematics
University of Trollhättan/Uddevalla
P.O. Box 957, SE-461 29 Trollhättan, Sweden
{richard.torkar, stefan.mankefors}@htu.se

Abstract

Using basic unit testing techniques we found 25 faults in a core component within a larger component oriented framework after the component had already started to be reused. We found that, even though this particular component had been subject to subsystem and system testing and used for some time, several faults were discovered which seriously would have affected applications using it, especially in terms of reliability. This study clearly indicates the need of a new approach to testing and verification within component-based development and reuse.

1. Introduction

The use of Commercial-Off-The-Shelf (COTS) software components has increased over the years. The continued success of COTS components, however, is highly dependent on the reliability of the software at hand. In a survey recently made [3], one of the key findings was that developers reuse components, but they seldom test software before incorporating it in the implementations, especially unit testing is seldom used. At the same time the majority of the developers did not test the components during original development [3], hence leading to a paradox of un-tested software being used again and again.

We do not believe that components and code in general are tested well enough. This makes, in some respect, component-based development (CBD) a potential nightmare. According to Parnas [1], every software product should be evaluated before being used at any later stage in the development process, something that is only partly done. If we are to really succeed in component-based software engineering and reuse in general, we must make sure that developers test [5] their code even more than they do currently. This to ensure that any faults in the product are detected as early as possible, and more important, is not "inherited" with the use of COTS components; Boehm [2] pointed out already 20 years ago that the longer a fault stays in a software system the more expensive it is to remove.

In this paper we report on an explorative case study on a component already in use in the software community, applying unit testing to it as a third party developer would (should) have, before incorporating it. By doing this, we want to, in a practical case, investigate the reliability of an actual component already in use. In order to try to choose a component that is relatively representative of "high reliability components" in terms of expected frequent reuse we tested a core class component (System.Convert) in the Mono:: framework [8]. Since the framework will be a foundation for potentially tens of thousands of applications using it in the open source world, any undetected fault will have very severe repercussions. The component at hand was furthermore already to some extent subsystem and system tested, deemed reliable and reused. No unit tests had to our knowledge been applied, however.

Different persons from the ones actually implementing the class or method, closely mimicking the situation of a developing team testing a COTS component before re-using, wrote all tests. Using a straightforward unit test approach we tested all available methods in the class, finding a total of 25 faults.

We find that even applying a straightforward basic suite of tests to a component before re-using it is of interest to the developers, as well as extra test cases performed after the formal development of the software. The remaining parts of this paper are devoted to the technical background, results, analysis and the broader scope and impact of our findings.

2. Background

Software verification and validation (V & V) intends to answer two basic questions. Are we building the product right and are we building the right product? In our case: is the product being built, conforming to the European Computer Manufacturers Association

Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)
1071-9458/03 $ 17.00 © 2003 IEEE

specifications 334 [6] and 335 [7] ECMA-334? The former being the C# Language Specification and ECMA-335 being the Common Language Specification, as submitted to the ECMA standardization body by Microsoft, Intel, Hewlett-Packard and Fujitsu Software in December 2001.

These two standards are likely to have a great impact on COTS, CBD and reuse in general the next couple of years. Thus a need to make sure that the foundation whereas several thousands or even tens of thousands of application will be built upon is stable. ECMA-334 is further considered to be a standard, which has clear component-based aspects in it and combined with ECMA-335 in conjunction with the framework library gives the future developer a platform with which (s)he can reuse large parts. The framework is in other words a large collection of components that can and will be reused. Hence the reliability of these fundamental components must be extremely high.

The components that tested in this study came from the Mono:: project [8]. Mono:: is an open source version of .NET [18], which is hosted by Ximian Incorporation. The goal for Mono:: is to provide several pieces of components for building new software, most notably a virtual machine, class library and compiler for the C# language.

2.1. Unit testing

Unit testing is a well-known technique [9] and has increasingly been used in the last couple of years, especially since the arrival and success of object-oriented languages, such as Java, C++ and more recently C#. Lately also development processes such as XP has made unit testing a closely integrated part of the development. Furthermore a recent study [3] shows that unit testing is one of the most common test technique used by software developers today.

In unit testing [10] the smallest piece of code (unit) is checked to ensure that it conforms to its requirements. These tests are written once and used many times, during the development cycle, to ensure that code regression is kept at a minimum. Usually the tests are written in the same programming language, which is used to build the software itself. Unit testing should not be used to test relationships or coupling in an object-oriented framework. If that is what one would like to test, then sub-system testing techniques do exist [11].

3. Methodology

Taking the starting point in the difficulty of a developer reusing a software component from a third party, we apply a straightforward unit testing scenario. We also assume the software development taking place within the Mono:: framework, the open source implementation of .NET, supposedly being one of the most component and reuse oriented platforms today.

As mentioned in the introduction we needed a fairly large class to use in our study. We evaluated several and finally chose the Convert [12] class in the System namespace. The main reason for choosing this class was its significance and its large number of methods, which would be in need of testing before incorporation of the component into an application. The class provides all standard methods for converting a base data type to another base data type in the framework. This is a typical task delegated to a framework or library in most applications, handling e.g. conversions between hexadecimal and decimal numbers or integers to strings. Hence possible failure in this class would affect base functionality in a vast number of applications employing the Mono:: framework. The namespace System also indicates that this class is a core part of the framework.

Assuming the typical limited resources [3] allocated for testing in software developing projects we chose to only implement a basic suite of test cases. We did not strive, in any way, towards completeness in test coverage, the reason being that we set out to show that even a very basic suite of tests still could find faults in a widely used part of a framework. The basic test cases we are referring to in this case consisted of testing the boundary conditions and off-nominal cases in which this component should degrade gracefully, without loss of data. Finally some random input was also carried out on each method being tested.

Since the tests in our case derived (as for a general software developer) from the knowledge of the specification and structure of the class(es), a pure structural approach known as white-box testing [13], was used. The tests written had only one objective in mind and that was to find flaws in the implementation according to the specification.

Several tools are available to a developer when performing unit tests of the type mentioned above. Most notable is JUnit [14], which is described by Gamma and Beck as being a *regression testing framework* and is Open Source [15]. Since JUnit is open source, other developers can port it to different languages. NUnit [16], by Philip Craig, is such a port.

Several programming languages are supported by NUnit, but in our case the C# programming language was the most important. The NUnit framework consists of several classes. These classes contain methods, which the developer uses when constructing test cases.

To compare resulting values, which is very often the case, different Assert [17] methods were used in this study. Especially the AssertEquals method was used extensively, since if used correctly it generated a message that makes it easier for the developer to establish exactly which test case failed.

```
AssertEquals("ID", expectedObject, receivedObject);
```

In the above case, when the expected object is not the same as the received object, an exception is thrown. The exception includes the value of the expected/received objects and the test ID, so that the developer can easily see where and why it failed.

An example of an error message can be seen below:

```
AssertEquals("#A00",(int)2,(short)2);
TestChangeType(MonoTests.System.ConvertTest) :
                #A00 expected:<2> but was:<2>
```

The reason the above test failed was that even though the value was equal, the type was not. Notice how the method ChangeType is being tested by the method TestChangeType. A Test prefix is added to a test method so that it will automatically be included into the testing framework when being run the next time.

It is not uncommon to write several tests that manipulate the same or similar objects. To be able to do this in a controlled environment a common base must be established. This base, also known as the fixture, makes sure that the tests are run against a known and well-established foundation. The next step is to create a subclass of TestCase (see below), add an instance variable for each part of the fixture, override SetUp() to initialize the variables and finally use TearDown() to release the resources you allocated in SetUp().

```
public class ConvertTest : TestCase {
  bool boolTrue;
  bool boolFalse;
  [...]
  protected override void SetUp() {
    boolTrue = true;
    boolFalse = false;
    [...]} [...]}
```

Once the above fixture is in place the developer can write many tests manipulating the same units. If the developer wants to run several tests at the same time the NUnit framework provides the developer with the object TestSuite which can execute any numbers of test cases together.

3.1. Unit testing of System.Convert

As already mentioned previously we selected the class Convert in the System namespace, for a number of reasons. The System.Convert class consisted of one public field and 22 public methods, all in all 2463 lines of code (LOC). Furthermore, each overridden method should be tested to ensure progressive reliability.

The routine for constructing the test method was easily established. First, the specification was read carefully; secondly, boundary, “off-by-one” and at least one legal input value test was written for each method belonging to System.Convert, and finally the tests were run. This process was repeated several times until all methods had tests written for them that covered all contingencies. To ensure the test’s integrity we implemented and executed them under the .NET framework [18] before applying the test cases within the Mono:: framework.

A unit test made for FromBase64CharArray, a method which converts the specified subset of an array of Unicode characters consisting of base 64 digits to an equivalent array of 8-bit unsigned integers, will illustrate the principles of the general methodology. The method takes three arguments, the *inArray*, the *offset* (a position within the array) and the *length* (num of elements that should be converted). The array inArray is only allowed to consist of the letters ‘A’ to ‘Z’, ‘a’ to ‘z’, numbers ‘0’ to ‘9’ and ‘+’,’/’. The equal sign ‘=’ is used to fill empty space. To make sure that the conversion was correctly made the result and the expected result, both arrays, must be looped through and compared. This can easily be done through e.g.:

```
for(int i=0; i<result.length; i++)
  AssertEquals("#U0" + i,expectedByteArr[i],result[i]);
```

The next two examples are test methods for ToBoolean. ToBoolean is overridden 18 times in System.Convert, one for each built-in type, twice for Object, twice for String and once for DateTime. Since the different examples are quite similar only Int16 and Char will be covered. Below Int16 is tested; if it is anything but zero it will be converted to true.

```
AssertEquals("#D05, true,
Convert.ToBoolean(tryInt16));
```

Next the Char example, shows that testing exceptions is just as easy. Since a conversion from char to bool is not allowed an exception should be thrown i.e. InvalidCastException.

```
try {
Convert.ToBoolean(tryChar);
} catch (Exception e) {
  AssertEquals("#D20",
                typeof(InvalidCastException),
                e.getType()); }
```

The test cases written are thus fairly straightforward and test every method's input and output, the specification deciding the legality of the outcome of each test.

4. Results

By using the described unit testing approach all in all 25 flaws were discovered. The test case consisted of 2734 LOC while the tested class holds 2463 LOC. This is more or less a 1:1 ratio between class LOC and test LOC, which is considered as being the default in XP [19].

This result in itself clearly indicates the severe problem of reliability in reusable components. That the findings occur in a core class in a framework makes this point even more severe. Virtually any type of fault in such a class could be expected to lead to failures occurring in a wide range of applications. Hence all the found faults have a very high degree of severability. Because of the nature of the class at hand, i.e. being a core component in a globally used framework, the relative reliability is extensively impaired by even a single or a few faults.

Turning to the technical details of the test cases, we cover a few examples fully, before continuing with a summary, in order to keep focus on the general component reliability rather than the individual fault.

Some of the failures detected were clearly the result of a misinterpretation as can be seen below.

```
short tryInt16 = 1234;
Convert.ToString(tryInt16, 8);
```

The above code snippet should, according to the specification, convert the short value '1234' to an octal string, i.e. '2322'. What really happened was that the value '1234' got parsed as an octal value and converted to '668'. This could easily be proved by changing tryInt16 to '1239', since the octal number system does not allow the number 9. The result in Mono:: was now '673', clearly wrong since a FormatException should have been thrown. We find it a bit strange that no developer had reported run-time failures of this kind when using the Convert class.

Yet another test case discovered a flaw in how hex values were treated.

```
Convert.ToByte("3F3", 16);
```

This line should convert '3F3', which is a hex value, to the byte equivalence. Since, in this case, there really is no byte equivalence, '3F3' is 1011 in the decimal number system and the byte type is only allowed to contain values between 0 - 255, an OverFlowException should be thrown. This was not the case in the current implementation, instead the method returned the value '243'. So the converter started over from '0', thus leading to 1011 - 3 * 256 = 243.

As can be seen from these two simple cases, all the test cases tested a minimum of two things, crossing over the maximum and minimum values for a given method or type, simply by using maxValue + 1 and minValue - 1. This is something that should have been tested during the implementation since it is considered to be one of the standard practices [20].

The above two underlying faults, which were uncovered in the implementation, would naturally lead to strange behavior in an application using System.Convert. Probably the only reason why this was not discovered earlier was that the above methods were not exercised in a similar way [as in this survey] by other developers.

As already mentioned, in total 25 faults were found in the Convert class. These faults were mainly of two types (Table 1, next page) that caused exception failures, e.g. OverflowException or FormatException, and secondarily, misinterpretation of the specification when the component was created, as we already saw previously, i.e. Convert.ToString(tryInt16,8).

LOC class	2463
LOC test	2734
Misc. exception failures	15 (1)
Logic fault	*4*
Incorrect control flow	*2*
Signed/Unsigned fault	*6*
Data/range overflow/underflow	*3*
Misinterpretation	9
Unknown	1
Total num of faults	25

Table 1. Overview of results. Faults in italic belong to the misc. exception category.

One unknown failure was found where we could not pinpoint the exact reason. The Convert.ToString method below should have returned the expected result, but instead it returned '-1097262572'.

```
long tryInt64=123456789012;
AssertEquals("#O40", "123456789012",
            Convert.ToString(tryInt64,10));
```

Clearly this is a case of overflow, but no exception was raised, which should have been the case.

What then, could the uncovered faults lead to? In the case of reliability ISO-9126 [21] mentions maturity, fault tolerance and recoverability. Clearly several of the flaws we found showed relatively immature aspects, e.g. faults that should have been uncovered if the framework had been used more extensively by developers. These faults probably would have been uncovered over time when the framework had been used more. But as we have already mentioned, Boehm [2] has pointed out the need for uncovering faults early in the development process for several reasons.

Fault tolerance in a framework, such as this, should be able to identify a failure, isolate that failure and provide a means of recovery. This was not the case with several of the exception failures we uncovered. Identification and isolation of a failure could in several of these cases be implemented by examining the input for validity, isolate non-valid input and notify the developer of the fault.

Finally, when an exception is thrown because of a fault in a core component, a developer would have problems recovering, since a stable foundation is expected. On the other hand, an overflow occurring without an exception being thrown would cause a very strange behavior in the application using the method as well as a severe problem to debug. If it is possible to differ between severability of faults in a core class in a framework such as Mono:: - all faults being of a very serious nature - a fault that *does not* cast an exception holds even a higher severability than the other faults.

5. Conclusion

CBD is often promoted as one of the great trends within software development. A fundamental problem, however, is the degree of reliability of the individual components, something clearly indicated by our current study.

Mimicking the situation of a third party developer, we chose to apply straightforward unit testing to a core component from the Mono:: framework, being the open source implementation of .NET. Employing "off-by-one", boundary testing and certain legal input for each method we were able to uncover in total 25 faults in the implementation of the class at hand. Although always extremely serious when it comes to a core class like System.Convert, some failures did not result in any exception being thrown. A fact that must be – if possible – considered even more severe.

This component had already been subject to certain sub-system and system testing and makes up one of the core parts in the framework. The fact that the component already was in reuse, clearly shows the seriousness of the reliability problem of CBD. Combined with the non-systematic evaluation of components from third parties by software developers [3] (i.e. lack of testing before usage as opposed to what was done in this study) the reliability not only of the components but a wide range of resulting applications is jeopardized.

Based on our findings we propose that some sort of low level testing of components should be a foundation for further testing methodologies, more or less without exception. Trusting the foundation, when adding module and sub-system tests, is vital. It is, to put it bluntly, better to add low level testing after implementation or even usage of a piece of software, than not doing it at all. In the specific case at hand a third party developer performing the test cases in this study would have avoided failures late in the development time-line and at the same time aided the CBD community. Sooner of later one will experience failures if testing is not performed properly. The question is; can you afford trusting the origin of a component? Since no de facto certification is widely used today [3], we believe the answer is no to that question.

One important thing must be stressed throughout any software project - if a developer finds a fault, they should immediately write a test case for it. That way the fault will show up again, if the present code deteriorates. This

could be considered as a best practice and somehow forced upon the developers during check-in of new or changed source code. If this practice had been followed in the project then some of the faults we found would probably have been found much earlier.

6. Future work

Programmers, in general, want fully automated tests, which find faults instantly. This is of particular interest in a CBD context where reusing is en extensive part of software development. Only automatization will allow for easy checks of software before incorporation into new software projects.

Even though there today exist some on-the-fly error checking techniques, e.g. syntax checking and lexical checking during compilation, there is still a need, to extensively improve and expand the current methods. Ideally, the tests should be created and performed constantly in the background and give feedback to the programmer immediately when manipulating code [19].

A different approach could involve test suites being run automatically when a developer checks in a change to the project on a configuration management system. By doing this, the developers could be notified when their submission deteriorates the existing and hopefully working code base. This has, to some extent, already been implemented, but a need for a more formalized approach still exists [22].

By adapting and merging several specific testing technologies it will hopefully be possible to show how to make it an integrated, reliable part of software engineering with automatization as the key benefit.

Once such tool could be the control of the consistency of a common code base stored in a repository and warn developers immediately when tests fail. The aim is that only the relevant code changes should be tested. Such a tool should also, in the future, be able to create simple test cases if asked for by a developer as described already by Luo et al. [23] in the context of constructing stubs for testing.

In the longer perspective, statistics such as test coverage and code regression (e.g. test failures) could be retrieved or calculated, for the benefit of both the developers and management.

7. Acknowledgements

Prof. Claes Wohlin for giving us input on how to best present our work. European Union regional development fund for funding part of this project, and Dr. Steven Kirk for proofreading it.

8. References

[1] Parnas, D.L. and Clements, P.C., “A Rational Design Process: How and Why to Fake it”, IEEE Transactions in Software Engineering SE-12, Feb. 2 1986, pp 251-257.

[2] Boehm, B.W. *Software Engineering Economics*, Addison-Wesley, 1983.

[3] Torkar, R. & Mankefors, S. “A survey on testing and reuse”, SwSTE’03, November 2003.

[4] Karlström, D., “Introducing Extreme Programming - An Experience Report”, SERP'01, October 2001.

[5] Rosenblum, D., "Adequate testing of componentbased software", Department of Information and Computer Science, University of California, CA, Technical Report 97-34, Aug 1997, http://citeseer.nj.nec.com/rosenblum97adequate.html, 2003-02-26.

[6] ECMA-334, “C# Language Specification”, ECMA, 2001.

[7] ECMA-335, “Common Language Infrastructure (CLI)”, ECMA, 2001.

[8] Mono::, http://www.go-mono.org, 2002-07-31.

[9] Brooks, F.P., *No Silver Bullet; Essence and Accidents of Software Engineering*. IEEEComputer April 1987, pp. 10-19.

[10] IEEE 1008-1987, “Standard for Software Unit Testing, ANSI/IEEE”, IEEE, 1993.

[11] Wang, C-C. et al., “An Automatic Approach to Object-Oriented Software Testing and Metrics for C++ Inheritance Hierarchies”, International Conference on Information, Communications and Signal Processing, 1997.

[12] The Class Library Specification, http://msdn.microsoft.com/net/ecma/All.xml, Microsoft Corporation, 2002-07-31.

[13] Kamsties, E. & Lott, C. M., “An empirical evaluation of three defect-detection techniques", Proceedings of the 5h European Software Engineering Conference, Springer-Verlag, 1995, pp 362-383.

[14] Beck, K. & Gamma, E., “Test Infected: Programmers love writing tests”, Java Report, 1998, pp. 51-66.

[15] Raymond, E.S. & Young, B., *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly, 2001.

[16] NUnit, http://nunit.sourceforge.net, 2002-07-31.

[17] MacKinnon, T., "Xunit – a plea for assertEquals", http://citeseer.nj.nec.com/525137.html, Posted January 01 2002, 2003-02-20.

[18] Thai L. et al., *.NET Framework Essentials (2nd Edition)*, O'Reilly, 2002

[19] van Deursen, A., "Program Comprehension Risks and Opportunities in Extreme Programming", Proceedings 8th Working Conference on Reverse Engineering, IEEE Computer Society, 2001, pp. 176-185.

[20] Myers, G. J., *The Art of Software Testing*, John Wiley & Sons, New York, 1978.

[21] ISO (1991). *International Standard ISO/IEC 9126. Information technology -- Software product evaluation -- Quality characteristics and guidelines for their use*, International Organization for Standardization, International Electrotechnical Commission, Geneva.

[22] Bache, G. and Bache, E., "One suite of automated tests: examining the unit/functional divide", http://citeseer.nj.nec.com/526066.html, 2003-02-20.

[23] Luo, G. et al., "Approach to Constructing Software Unit Testing Tools", Software Engineering Journal 10, November 1995, pp. 245-252.

[24] Toepee, S. and Ranville, S., "Model Driven Automatic Testing Technology : Tool Architecture Introduction and Overview", 18th AIAA/IEEE/SAE Digital Avionics System Conference, October 1999.

Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)
1071-9458/03 $ 17.00 © 2003 IEEE