International Journal of Research in Computer Science
eISSN 2249-8265 Volume 3 Issue 5 (2013) pp. 19 - 23
www.ijorcs.org, A Unit of White Globe Publications
doi: 10.7815/ijorcs.35.2013.071

AN ENHANCEMENT TO CLEANROOM SOFTWARE ENGINEERING USING CSUNIT TOOL

Pooja Manocha[1], Arun Jain[2]

[1]*Research Scholar, Department of Computer Science & Engineering, Haryana College of Technology & Management, Kaithal, Haryana, INDIA*
Email: poojamanochamca10@gmail.com

[2]*Assistant Professor, Department of Computer Science & Engineering, Haryana College of Technology & Management, Kaithal, Haryana, INDIA*

Abstract: *Clean-room software engineering is a technical and managerial process for the development of high-quality software for zero defects with certified reliability, which combines the correctness, verifications and statistical usage testing to certify the software fitness. This paper describes the clean-room software engineering along with its technology, processes, tools support and its comparison with other traditional approaches, advantages and disadvantages. In this paper is presenting the detail description of Clean-room Software Engineering and Object Oriented Technology and also the experimental study of collaborative engineering and mutation testing performed through csUnit tool integrated with Visual studio 2008, which have been introduced to enhanced or improved Clean-room Software Engineering.*

Keywords: *Clean-room, Box Object Oriented Technology, Collaborative Engineering and Mutation Testing*

I. INTRODUCTION

Clean-room software methodology was developed by Harlan Mills and his colleagues from IBM. The main goal with the clean-room is to construct software with no defects during development. Clean-room software has an 80-20 life cycle means 80 percent of the total time is dedicated to design and only 20 percent is dedicated to code. Correctness is built by the development team through formal specification, design and verification. Certification team is responsible for certifying the quality of the software with respect to its specification instead of testing in quality. The main idea in the clean-room software engineering is to develop the right product with the high quality. Clean-room development begins with requirement specification. Development team then creates the functional specification which defines the required external system behaviour and certification team creates the usage specification which defines usage scenarios. Sometimes for larger software projects functional specification is also created by a separate specification team. Testing in clean-room is fundamentally different than conventional testing methods. After the correctness verification phase, software increments are delivered to the certification team for first execution. Clean-room testing focus on showing statistical amount of use cases that functions correctly. Testing team generates test cases such that every test case represents a possible use of the software as defined by the models. Appropriate inputs and events are chosen based on these use cases. Certification team calculates probability distribution of these use-cases depending on customer interviews, usage scenarios and a general understanding of the product. There are various software tools that can play an important role in the Clean-room development approach. These tools are useful in constructing and manipulating the system design and source code. The use of these tools facilitates the process of reviewing the system design and source code prior to submission for testing by the independent group. Below are categories of tools which are necessary according to the stage of life cycle.

Decomposition tool: This tool is useful in requirement phase to structuring the system hierarchy. It is also useful to plan the composition and number of increments to form the software milestones.

Design tools: These tools are useful for producing box structure design and also for prototyping which allow early evaluation of the user interfaces.

Statistical testing tools: These tools generate set of test cases and mainly useful for software testing. IBM clean-room certification assistant is an example of a tool which performs this function.

II. RELATED WORK

Papers and articles about Clean-room usually emphasize the mathematical soundness of Clean-room verification and often this causes readers to believe that Clean-room is an all-or-nothing approach: if one is not doing full mathematical verification, then it will not be a Cleanroom. However, the actual experience of projects shows this not to be the case [3]. Instead, Clean-room projects combine all forms of peer review from walk-throughs and informal "buddy" inspections to formal reviews that include correctness proof.

The general term used in Clean-room is "review", but Dyer's term [2] "verification based inspection" is also used. However, all of the terms include an extensive range of practice. In the next few sections we will look at some of the different kinds of Clean-room reviews and how they would fit into the elevator process. The general guide to selecting a point on the spectrum of rigor depends on the risk of a problem escaping detection versus the cost of additional rigor. Informal inspections are often used at the very end of a review cycle for an object, when the final review identifies a small number of insignificant changes [9]. Here, the reason of review is to ensure that, there is no typographic or other small mistakes are made. A "buddy" system will often be used in such a case, where two developers agree to undertake such a final inspection of each other's work. The buddy system is less resource-consuming than convening a full review by the entire team, but admits the possibility that a minor change is the prior review's suggestions were not correctly noted. Walk-throughs often form the basis for the earliest development reviews of an evolving design. Often a quick reaction is needed on a particular idea or an immediate feedback is requisite to ensure continues making progresses. Any informal walk through may be carried out without extensive preparations by the team members before-hand and materials may be handed out at the review or may even be sketched at the meeting on a whiteboard. The results and decisions must always be captured and committed to the evolving documentation (specification or design) so that they can be formally reviewed later. In an effort to make official the review process, some teams [4] have agreed to adopt elements of the inspection protocol recommended by Fagan [5]. These elements include a separation between reader and designer, the use of a trained launch pad or and the appointment of an entity to record the decisions [9]. Most Clean-room teams that adopt this approach will surely augment Fagan's checklist-based inspection approach (where teams seem for bugs or 'issues') with verification based inspections (where teams apply the verification using, like, correctness questions etc.). Bug hunting alone might be susceptible to a form of Beizer's "pesticide paradox" developers are trained over time to not make those particular mistakes, but will make (and overlook) different ones. Design-quality questions are not off limits in verification, as they may be in code inspection. Many Clean-room teams have combined or distributed the inspection roles. For example, some have adopted the practice of having each team member record design decisions and use the review-preparation period to compare the actual changes with records. The decision on exact review techniques must be made by an individual project [9]. Groupware tools which includes electronic mail and shared electronic documents, can reduce the amount of face-to-face review time that is needed and can be very helpful when meeting space is tight or when individual work schedules are not conducive to frequent team reviews [6,7]. Care should be taken when using these tools to ensure that the primary goal of review is not compromise rather than to add "correctness shown" to the checklists for design and code inspections. In this way, the organization learns from past mistakes and can apply specific design-quality guidelines.

Further, in an organization that is not yet ready for correctness verification, checklist-based inspections are still better than alternative of unrestrained, ad-hoc unit debugging. A process which employs the kind of hierarchical designs has described in the previous section, which uses traditional inspections as the Clean-room "review" techniques, would be the Clean-room process though not essentially an optimal one.

Review: A Summary

There are many techniques for demonstrating correctness and considerable variation in the 'standard of proof' required. To use a legal correspondence, the team designing life critical systems might be asked for correctness to be demonstrated "beyond a reasonable doubt" whereas a team designing a prototype word processor might only demand that the weight of the evidence points toward correctness. It is the author's opinion that, for some projects and some objects, inspection methods sufficiently meets the requirements of a "Clean-room" review.

III. RESULTS & DISCUSSION

CsUnit: A .NET class library with classes for writing test cases will be installed in the global assembly cache (GAC). Furthermore it will be displayed in the "Add Reference" dialog box of Visual Studio.

CsUnit Interfaces: This assembly contains the interface, delegate and event definitions needed for internal communication of the csUnit framework.

A. The Traditional Approach

Let us assume that our task is to implement a stack class. This is usually a no-brainer as stacks are typically part of class libraries. The code might look like this:

```
public class Stack {
   public Stack() {
   }
   public void Push(object obj) {
   _ar.Add(obj);
   }
   public object Pop() {
      object top = _ar[_ar.Count – 1];
      _ar.Remove(top);
      return top;
   }
   public int Count {
      get {
         return _ar.Count;
      }
   }
   private ArrayList _ar = new ArrayList();
}
```

Now let's assume that the implementation is considered to be finished. Which tests are required in order to make sure this class works? There are quite a few:

- Create a new stack, the count must be zero
- Add one item to the stack, the count must be one
- Add one item to the stack, remove one item to the stack, the count must be zero

These are probably the obvious ones. Though, there are lots of more:

- Add two, the count must be two
- Add a null reference, the stack must throw an `InvalidArgumentException`
- Call `Pop` on an empty stack, the return value must be a null reference and the invocation of `Pop` on an empty stack should not crash.

With implementation first `and` tests afterwards, it might have been the case that one of the latter tests would have been overlooked.

B. Test-First Development

Enter test-first. This time we do not have an implementation. Instead we write our first test. It might look like this:

```
[TestFixture]
public class StackTests {
   [Test]
   public void EmptyTestHasCountZero() {
      Stack s = new Stack();
      Assert.Equals(0, s.Count);
   }
}
```

Obviously this test must fail. Even better, we will not be able to convince the compiler to accept this test without warnings or errors.

Therefore we implement our first version of the stack:

```
public class Stack {
   public int Count {
      return 0;
   }
}
```

This time the compiler will no longer complain. Furthermore, we will not get any error or failure when running the tests.

Could we have implemented less? In order to satisfy the compile we need the class and the method definition (which is a property in .NET speak in this case). So no, there is no simpler stack class that would make the test run at all.

So we have finished the first iteration. If you like you could add both the test fixture and the stack class to your revision control system (e.g. CVS). Although the class is not complete by far, you can rest assured that the functionality that is there is tested and will not degrade in quality.

Let us write a second test: Add one item, the Count must be one:

```
[Test]
public void CountForStackWithOneItem() {
   Stack s = new Stack();
   s.Add(5);
   Assert.Equals(1, s.Count);
}
```

We add an integer with the value 5. An integer is a value as well so we are allowed to add the 5.

Do not add any implementation code at this point. Just start the compiler and see what happens. It does not compile, as the Add() method is missing. You could say, the compiler "talks" to you. Or as some extremos say: "Don't ask me, ask the system."

Next we add the Add method:

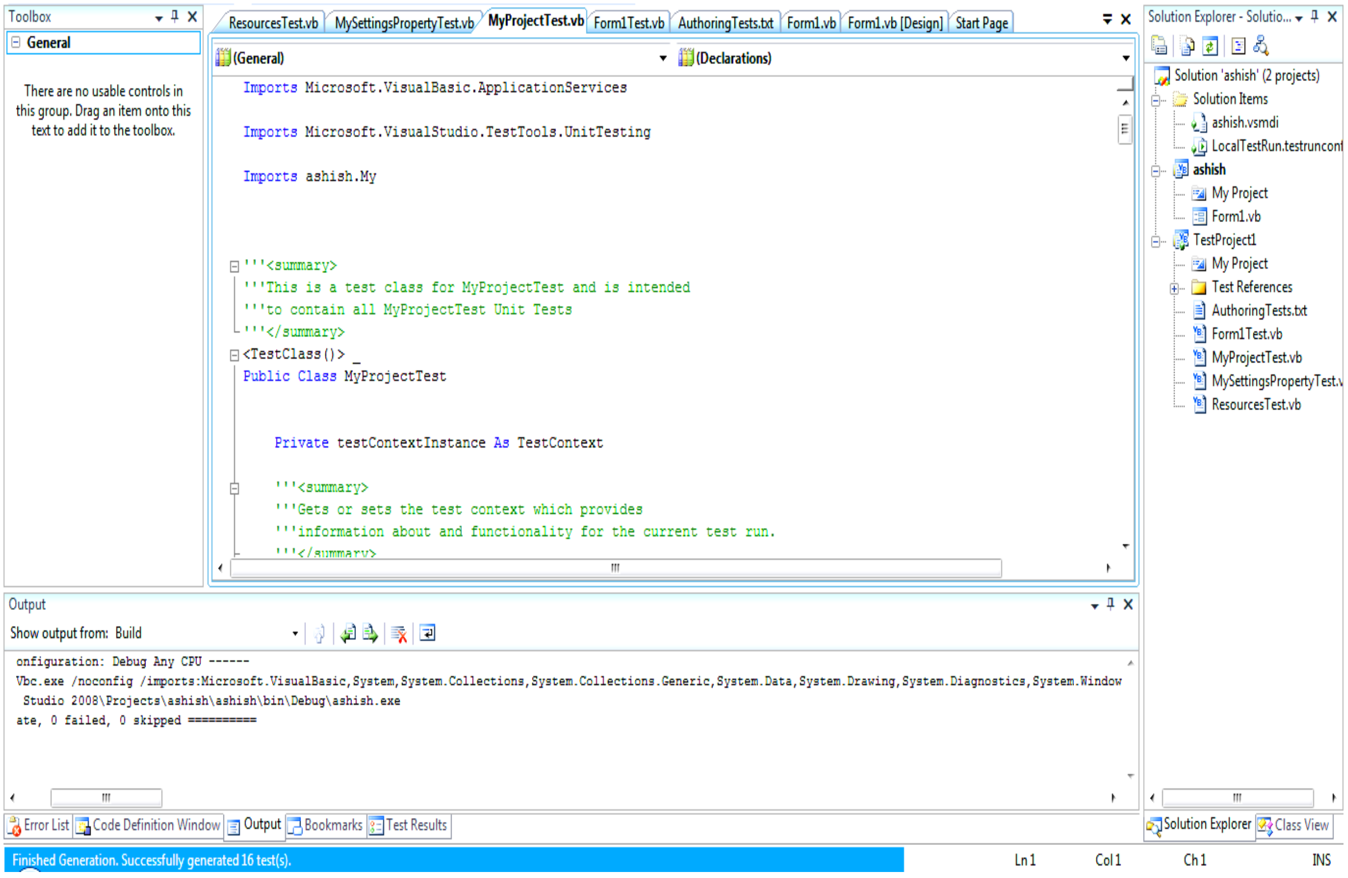
```
public class Stack {
   public int Count {
      return 0;
   }
   public void Add(object obj) {
   }
}
```

We cannot add less. The compiler does not report any more errors, so we can execute the tests. The result is that our first test will still pass; however, the second one will fail.

With the advent of agile methodologies test driven development has gained popularity again. Originating many years before the term "agile" was used for software development, some programmers have already used test driven development.

From a pragmatic perspective units are components of software. In general such components can be class libraries, database access layers, but also user interfaces. For the purposes of this manual we will focus on a class library.

Some of the traditional development approaches explicitly distinguish different development activities such as implementation and testing. These activities are even carried out by different departments in some cases. In many cases testing is a separate step that is done *after* implementation. However, this order has a major disadvantage. How do you know, when exactly you have conducted enough tests and how many tests are enough. One simply cannot know. Even if one can write hundreds of test you might have missed the single one that would have shown that there is still a critical bug in software. Let us consider the reverse order: test first, then implementation. So we write a test and let it run. If the test really codifies an additional piece of functionality (requirement) of you software, then the test will fail and we are done with the test. Next is implementation. How do we know when exactly we have implemented enough and how much code is enough for particular application. This question can answer by author. It may be done if and only if have test all written pass. How do we know, in this scenario, whether we have to written enough test or not. Then, how is this approach better than the other one. We have to make a connection between the test and some code. We write a test; if it passes, then we either test functionality that is already covered by a different test or the test covers functionality that has not been covered frequently, that is in the previous step we have implemented more that required for the previous test. Figure (1) is representing mutation testing using Cusnit. It is Deciding on which of the two scenarios is true is important and makes you think about your software.



```
Imports Microsoft.VisualBasic.ApplicationServices

Imports Microsoft.VisualStudio.TestTools.UnitTesting

Imports ashish.My



'''<summary>
'''This is a test class for MyProjectTest and is intended
'''to contain all MyProjectTest Unit Tests
'''</summary>
<TestClass()> _
Public Class MyProjectTest


    Private testContextInstance As TestContext

    '''<summary>
    '''Gets or sets the test context which provides
    '''information about and functionality for the current test run.
    '''</summary>
```

```
onfiguration: Debug Any CPU ------
Vbc.exe /noconfig /imports:Microsoft.VisualBasic,System,System.Collections,System.Collections.Generic,System.Data,System.Drawing,System.Diagnostics,System.Window
 Studio 2008\Projects\ashish\ashish\bin\Debug\ashish.exe
ate, 0 failed, 0 skipped ==========
```

Finished Generation. Successfully generated 16 test(s).

Figure 1: Snapshot of mutation testing using csUnit

IV. CONCLUSION AND FUTURE WORK

In this paper, we have studied the transition phase of a software development company that wants to introduce and implement a new tool called csUnit. By this transition phase we mean from the current situation is without csUnit to the desired situation with csUnit. The motive of introducing csUnit is to improve software development. Currently, the test & integration phase, compared with the other phases, takes too much long time and makes the process which is difficult to manage. An important part of the problem is that independently developed software units do not work together flawlessly. The Cleanroom approach has both technical as well as managerial control. It is basically used for development of safer critical products and not for ordinary products. Cleanroom practitioners use tables and symbols formalism. It uses box structures for verification, so no debugging is required. But it cannot become accustomed quickly to rapidly changing requirements.

The Cleanroom Software Engineering can be improved by finding the gaps between csUnit tools defined for the project and how this can be used to improve Cleanroom Software Engineering. Dynamic analysis of software can be performed in different ways like using profilers, from its dynamic models and using object-oriented programming as this approach provides a balanced method for the dynamic analysis of programs.

V. REFERENCES

[1] Murphy P., “A Review of Clean-room Software Engineering”, Citeseer, 2007.
[2] Dyer M., “The Clean-room Approach to Quality Software Development”, Wiley, 1992.
[3] Deck, M.D., "Cleanroom Software Engineering: Quality Improvement and Cost Reduction," Proc. of Pacific Northwest Software Quality Conference, October, 1994, pp. 243-258, doi=10.1.1.104.8072.
[4] Hausler, P.A., "A Recent Clean-room Success Story: The Redwing Project," Proc. 17th Annual Software Engineering Workshop, NASA Goddard Space Flight Centre, December, 1992.
[5] Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, vol. 15, no. 3 (1976), pp. 219-249. doi: 10.1147/sj.382.0258
[6] Votta, L.G., Jr., “Does Every Inspection Need a Meeting?” Software Engineering Notes, D. Notkin, ed., Vol. 19, No. 5, pp. 107-114. doi: 10.1145/167049.167070
[7] Porter, A., H. Siy, C.A. Toman, and L.G. Votta, “An Experiment to Assess the Cost-Benefits of Code Inspections in Large-Scale Software Development,” ACM SIGSOFT 95, Washington, DC, 1995. doi: 10.1145/222124.222144
[8] Kamal Deep Kaur, “Clean-room Software Engineering: Towards High-Reliability Software, IJCST, ISSN: 0976-8491(Online) ISSN: 2229-4333(Print) Vol. 2, Issue 4, Oct - Dec. 2011.
[9] Deck M., “Cleanroom Review Techniques for Application Development”, 6th Int. Conf. On Software Quality, Ottawa, Canada, 1996.

How to cite

Pooja Manocha, Arun Jain, "An Enhancement to Cleanroom Software Engineering using csUnit Tool". *International Journal of Research in Computer Science*, 3 (5): pp. 19-23, September 2013. doi: 10.7815/ijorcs.35.2013.071