2009

PACIFIC NW

SOFTWARE QUALITY CONFERENCE

MOVING QUALITY FORWARD

October 27-28, 2009

Conference Paper Excerpt

From the

CONFERENCE PROCEEDINGS

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commecial use.

Leveraging Code Coverage Data to Improve Test Suite Efficiency and Effectiveness

Jean Hartmann
Test Architect, Developer Division Engineering

Microsoft Corp.
Redmond, WA 98052

Tel: 425 705 9062
Email: jeanhar@microsoft.com

Biography

Currently, a Principal Test Architect in Microsoft's Developer Division with previous experience as Test Architect for Internet Explorer. My main responsibility includes driving the concept of software quality throughout the product development lifecycle. Spent twelve years at Siemens Corporate Research as Manager for Software Quality having earned a Ph.D. in Computer Science in 1993, while researching the topic of selective regression test strategies.

Abstract

During each release of Visual Studio, substantial time and resources are expended in test case development, execution and verification. Thousands of new tests are added to existing test suites without any kind of review regarding their unique contribution to test suite effectiveness or impact on test suite efficiency. In the past, such unbridled growth in test collateral was sustainable without significantly impacting product release, offset by simply increasing machine and staff resources. With the growing number of test configurations in which these tests need to be run, this is no longer feasible – it is time to clean up!

In this paper, we describe how we leverage existing code coverage data, together with reduction techniques, to help each test team analyze its test suite and guide them in improving its effectiveness and efficiency. The analysis focuses on identifying groups of tests cases given specific tactical goals – for example, increasing current test suite stability and reliability, assisting with test migrations, reducing test suite execution time and reducing test suite redundancy. The guidance focuses on a set of best practices that teams can adopt to achieve those goals. The paper reflects on some of the benefits and challenges we faced as part of this case study. It also outlines the tools that were developed to conduct the analysis and support the best practices. We use examples and data taken from the case study to illustrate and emphasize key points.

1. Introduction

For each new release of a major Microsoft product, thousands of new test cases are added to existing test suites. Over time, these tests suites have evolved into huge legacy test beds that consume large amounts of resources for test maintenance, execution and failure analysis.

It appears that a key contributor to our test suite redundancy issue is the turnover and off-shoring of testing staff. The original author of the test code has moved on and the test code may be insufficiently documented or complex. New testers needing to maintain that code are either unfamiliar with it (structure and purpose) or may lack sufficient experience to update it correctly. Instead, they opt to simply copy and slightly modify the test case or write new tests that essentially perform the same purpose as the old test code.

Gaps in the testing process appear to be another contributor. Reviews of test plans and test code are not rigorous enough yet to identify and filter out potential (test case) duplicates in the various stages of test development based on their effectiveness and contributions. For example, reviews do not identify those scenario tests that are effectively subsets of larger end-to-end scenario tests. Rather than deprecating those simpler tests, they remain in the test suite for years. Also, tests are not repeatedly evaluated based on their priority during release or from release to release. Tests that were high priority during the early stages of a development cycle or previous release may now no longer be relevant or as important for product validation.

As a result, test agility is being severely impacted. Test teams that in previous releases were able to run their 'nightlies' overnight can now no longer do so, even with extra resources. Instead, teams are finding that their 'nightlies' are taking on the order of days to execute (and even longer to analyze). Those teams are seeking ways to retain/regain their test agility with original goals of providing timely data concerning the quality of their builds, e.g. nightly runs – 8 hours, weekly runs – 24 hours, full runs – 48 hours. This problem is further compounded with the rising number of test configurations that require these suites need to be run against – think simply of the combinations of product versions, platforms and SKUs that need to be validated during the product development cycle.

With product functionality changing significantly, there is also the maintenance cost associated with updating the growing number of tests and preventing them from becoming 'stale'. That cost constitutes an increasing portion of testers' time. This leads to a situation wherein testers are spending a major portion of their time maintaining and analyzing failures from existing tests, rather than creating new test automation to exercise new product functionality.

Test reliability also appears to be impacted as trade-offs are increasingly being made between writing new test automation and maintaining existing test code. As a result, there are large numbers of test case failures that are being less frequently analyzed by testers. Such situations can in turn have undesirable consequences - true product failures/bugs are

‘lost’ in the thousands of unanalyzed test failures being reported by the test case management/execution system.

The following sections describe the initial phase of a long-term test initiative aimed at regaining our test agility, while maintaining our standards regarding high quality products. In this phase, we aim to characterize and optimize test suites owned by different product units within Developer Division based on two key goals:

- **Test effectiveness** - expressed in terms of broadest possible product coverage and code execution based on the hypothesis that the more code a test case exercises, the more likely it is to find a bug.

- **Test efficiency** - expressed in terms of a reduced number of tests to save on test execution and more importantly, failure analysis time, but also maintain failure detection rates of the original suite.

In Section 2, we describe how we characterized sample test suites and interpreted factors contributing to their test effectiveness and efficiency. Based on that analysis, Section 3 outlines the techniques and tools that were applied to ‘extract value’ out of specific classes of tests, while Section 4 then describes how we ‘squeezed’ the suites to optimize for efficiency. Section 5 highlights some early results by identifying the test case reductions that were achieved, lessons learned and describing key benefits that the teams experienced. Finally, Section 6 outlines the future work that we intend to conduct in subsequent phases of this initiative.

2. Characterizing Test Suites

In a first step, we attempt to characterize various test suites based on their *unique* and *cumulative* contributions to code coverage. Results are aggregated across the product binaries owned by each team. **In this exercise, we use code coverage not so much as an indicator of test adequacy, but focus more on its ability to demonstrate product coverage (or lack thereof)**.

The data that we collected is expressed as *block* coverage[1] and is grouped according to test priority. These criteria were selected as the data was readily available at Microsoft - most test teams regularly collect and analyze code coverage data on a per binary basis. They also structure their testing efforts and schedules around running test cases of specific priorities. Thus, mapping coverage data against test priorities helped the dozen teams involved in the study to better understand the results. A typical MS test schedule relies on running P0 or priority 0 tests on a daily basis with P1 tests being run weekly and P2, P3 and P4 tests being run during select test passes or major milestones. P0 tests, often known as *Build Verification Tests*, validate the basic operations of a product, such as

[1] *Block coverage* is a control flow-based coverage criterion that is essentially the same as statement coverage except the unit of code being measured is a sequence of non-branching statements.

installation and start-up. P1 tests concentrate on exercising key customer scenarios and features and so forth.

2.1 Individual Contributions

We assess the individual or independent coverage contributions of each set of tests of a given priority to see, if those tests are able to effectively exercise the product code. Figure 1 compares three Visual Studio product test suites whose tests have been categorized by test priority and charted against % block coverage (y-axis) across product binaries. Column 1 represents P0 tests, column 2 depicts P1 tests and so forth.

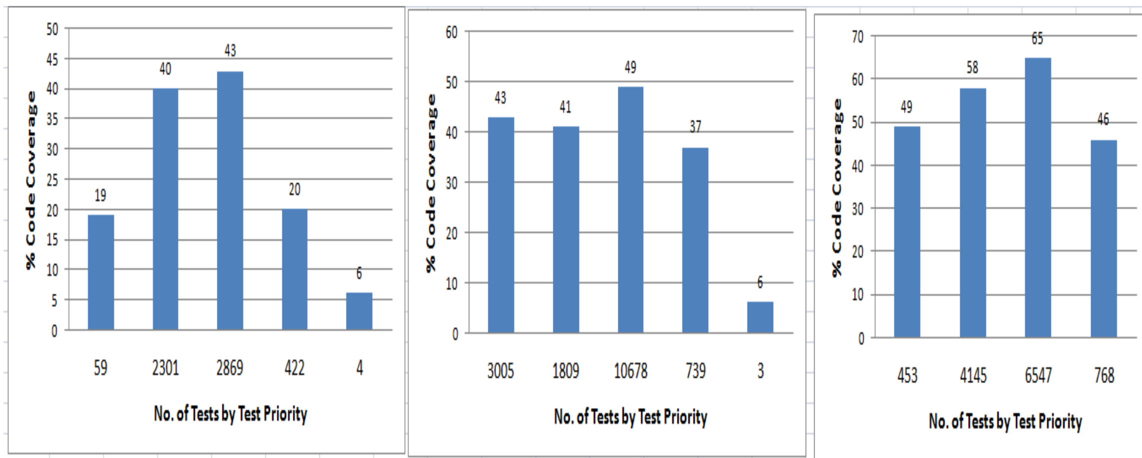

Figure 1: Individual Contributions to Code Coverage by Test Priority

We expect to see (and want to drive towards) an early peak in coverage, so that higher priority P0/P1 tests are executing the major customer scenarios as frequently as possible in a typical test schedule. Subsequent test sets (P2 and beyond) can ideally start to trend downwards in terms of coverage attained, indicating less independent coverage of the product, yet increasing overall cumulative coverage (Figure 2).

We also examined the "yield" per test case (%coverage/total no. of tests) to give a *rough* measure of how efficient a particular test category is in achieving its product coverage. Any significant peaks are worth investigating further. By calculating the yield using the data in Figure 1, for example, it became clear that some teams exhibited a wide variation of yields with test suites exhibiting significant peaks with their high (P0) and low priority (P4) tests. The former indicates that some teams have taken time in carefully defining their high priority test sets to broadly exercise the product. The latter typically represent higher priority tests from earlier product releases; these still represent effective regression tests and need to be further examined and leveraged better.

Another interesting point is that the P2 test cases in each product suite appear to provide the best independent coverage compared with the P0 or P1 tests and yet they are only run during major test passes or milestones. This leads us to examine the cumulative coverage of the suites to, for example, see if those same P2 tests also contribute *unique* coverage. In these three cases, they do.

2.2 Cumulative Contributions

Another perspective regarding product coverage is to examine the *cumulative* coverage contributions across those same test suites. We hope to see early peaks in cumulative product coverage using a combination of P0 and P1 tests with coverage trailing off thereafter. This would ensure broadest product coverage during daily and weekly test runs.

The plots in Figure 2 represent cumulative % block coverage against a cumulative count of test cases, e.g. with column 1 depicting P0 tests, column2 showing P0 and P1 tests and so forth. These plots clearly show that the maximum product coverage is only achieved after contributions from P2 tests – the quandary being that these tests are being too infrequently executed. Moreover, P3 and P4 tests provide no increase in cumulative coverage thereafter, yet they represent thousands of additional and *potentially* redundant tests being executed and requiring analysis.

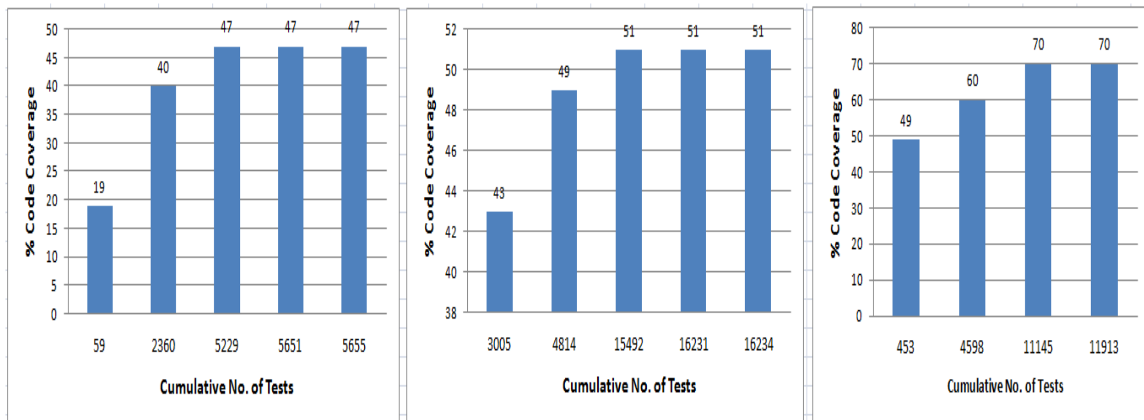

Figure 2: Cumulative Contributions to Code Coverage by Test Priority

3. Addressing Test Effectiveness

In the previous section, we examined different product test suites and attempted to characterize them according to their individual and cumulative coverage contributions. We also reasoned that the P2, P3 or P4 tests provide unique value or test effectiveness to each suite. Thus, in this section, we aim to 'extract' and assess that unique value in order to then either upgrade test priority or archive the test case(s).

To achieve this goal in a timely manner, we had to leverage automation that was able to interrogate the code coverage database and determine the differences in traces[2] between specific sets of tests. Figure 3 illustrates a typical results file showing the sorted, unique coverage contributions of P3 tests in comparison to P0/P1 tests for one of the sample product suites. Tests could be prepared for review based on the ratio of unique blocks

[2] Traces are the execution traces captured as a result of executing a given test case and expressed in terms of the code blocks traversed.

covered vs. total blocks covered – the closer the ratio is to 1, the more unique the value of that test case. For reference, we have also added the total block count. Full details are given in terms classes/functions covered, so that testers can investigate the code traversed by those tests in depth. At this time, this data is not used for the purposes of collecting metrics or triggering follow-up actions, but could be in future.

Class	Function	Unique Blocks Covered	Total Blocks Covered	Total Blocks	Test Case Name
Diagram	OnScroll	48	56	79	Branch Root.qa.md.\
VSWCFService	ImportWCFM	35	94	108	Branch Root.qa.md.\
SLClientGener	UpdateErrorLi	33	33	84	Branch Root.qa.md.\
MyApplication	FireChangeNc	28	28	35	Branch Root.qa.md.\
CDaVinciQuer	VSetForClaus	26	26	31	Branch Root.qa.md.\
AdoDotNetCo	ToDisplayStri	26	26	38	Branch Root.qa.md.\
DataTableAnd	OnSplitterMo	23	23	23	Branch Root.qa.md.\
DataConnectic	RemovePrope	22	22	22	Branch Root.qa.md.\
DataObjectIde	FindIdToIdMa	21	21	23	Branch Root.qa.md.\
DataConnectic	ModifyProper	20	20	20	Branch Root.qa.md.\
ResourceStrin	WndProc	18	30	31	Branch Root.qa.md.\
Diagram	OnMouseWhe	16	16	25	Branch Root.qa.md.\
ChannelEndpc	GetEndpoints	16	19	23	Branch Root.qa.md.\

Figure 3: Report Indicating Unique Block Coverage by Test Priority (P3 Tests)

4. Addressing Test Efficiency

Now that the unique contributions of a particular test or set of tests of given priority have been analyzed and the appropriate action has been taken, it is time to address our second objective of reducing the overall size of the test suite and thus, achieving better test efficiency.

For this purpose, we again deploy automation that examines the code coverage data and determines a 'core' set of tests with the same level of product coverage as the original test set, but significantly smaller in size. As a result, we have a core set of tests that we aim to keep and build upon and a large set of remaining tests that constitute potentially redundant or duplicates ('dupes'). These need to be reviewed and then either included in the core set or deprecated.

4.1 Selecting a Core Set of Tests

To select this core set of tests, we developed automation that was previously described in [1, 2]. This tool embodies two algorithms - one is capable of determining a truly minimal set of tests using a traditional greedy algorithm [3], the other an algorithm that focuses on maximizing product coverage. In the future, we will be examining the benefits of the first algorithm; this paper focuses on applying the second algorithm. As a result, our core set of tests was slightly larger than the one that would have resulted from applying the

greedy algorithm, but it was therefore also ‘safer’ and had potentially better fault detection capabilities.

Figure 4 illustrates this algorithm at work. For a code coverage matrix that maps tests T1 through T6 against code blocks B1 through B8, the shaded squares depict a test case executing a particular code block. Each of these tests represents a unique customer scenario being independently executed and the corresponding set of code blocks being traversed. For example, scenario test T1 exercises B1, B2, B3 B5, B6 and B8. The algorithm now traverses the matrix, that is, each code block from B1 through B8 and selects the test case with the largest number of blocks traversing not only itself, but also all other blocks in the product binary. By doing so, it is maximizing the product coverage, while also maintaining the original level of product coverage. When the algorithm has finally traversed all blocks, the final set of tests comprises T1, T4 and T6.

Earlier on, we mentioned that this core set was not minimal – in the example below, the application of a greedy algorithm would result in T1 and T2 being picked as the final set. This may not appear to be a big difference in the context of this example, but when test matrices include thousands of tests, this difference is significant. Also, one of the refinements that we are currently implementing is to run this algorithm repeatedly over the previous set of identified ‘dupes’, thereby ordering them and making further review and analysis easier.

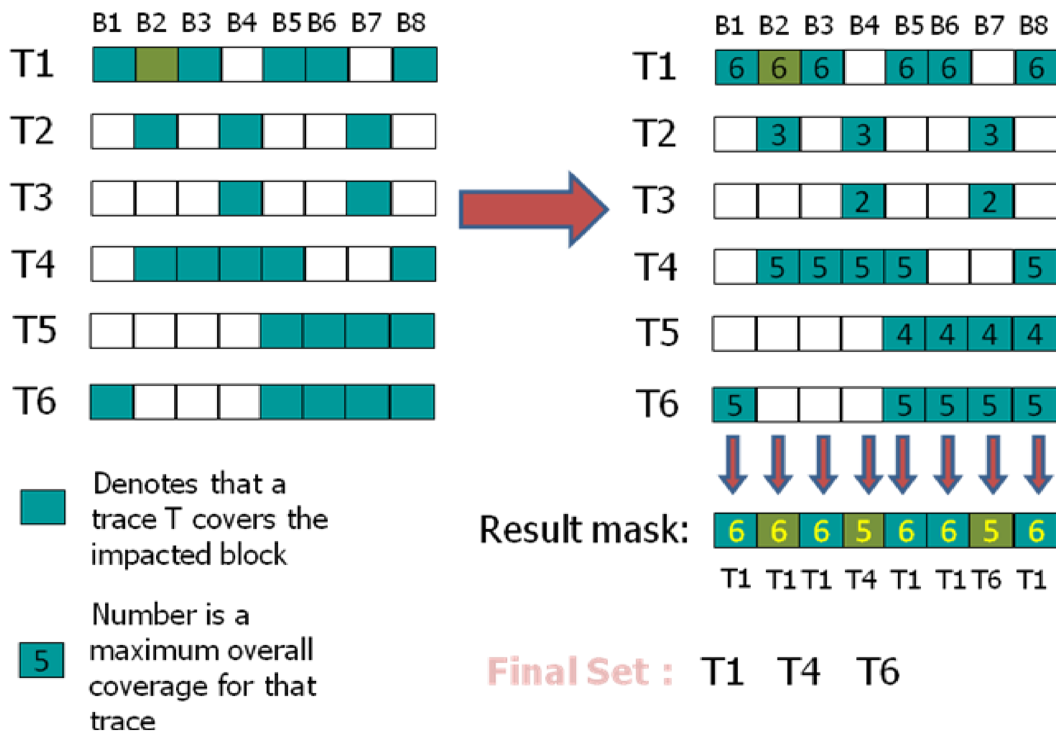

Figure 4: The Maxcoverage Algorithm at Work

It should be noted that before applying the algorithm, we have the ability to filter out any unwanted traces that pertain to failed or unwanted categories of tests, the former often contributing partial product coverage. Furthermore, we explicitly filter out code coverage

contributions from developer unit/API tests as we want to compare and select QA tests, that is, functional tests that validate our customer scenarios. Not doing so, would have led to the algorithm favoring the QA tests and implying that the developer unit/API tests were potentially redundant, which of course they are not.


```
<binary name="microsoft.data.connectionui.dialog.dll" IDComponentInfo="119" traces="249" new_blocks=
  old_blocks="7103" impacted_blocks="7103" deleted_blocks="0" total_blocks="7103">
- <sets>
  - <set number="1">
    - <traces>
        <trace IDTrace="13550" impacted_block_coverage="2455" new_impacted_block_coverage="0"
          old_impacted_block_coverage="2455" blocksaddedtoset="2455" totalcoverage="2455"
          cumulativecoverage="2455" />
        <trace IDTrace="27209" impacted_block_coverage="2299" new_impacted_block_coverage="0"
          old_impacted_block_coverage="2299" blocksaddedtoset="482" totalcoverage="2299"
          cumulativecoverage="2937" />
        <trace IDTrace="27211" impacted_block_coverage="2142" new_impacted_block_coverage="0"
          old_impacted_block_coverage="2142" blocksaddedtoset="268" totalcoverage="2142"
          cumulativecoverage="3205" />
        <trace IDTrace="27148" impacted_block_coverage="2165" new_impacted_block_coverage="0"
          old_impacted_block_coverage="2165" blocksaddedtoset="147" totalcoverage="2165"
          cumulativecoverage="3352" />
        <trace IDTrace="27213" impacted_block_coverage="2058" new_impacted_block_coverage="0"
          old_impacted_block_coverage="2058" blocksaddedtoset="43" totalcoverage="2058"
          cumulativecoverage="3395" />
        <trace IDTrace="27408" impacted_block_coverage="2236" new_impacted_block_coverage="0"
          old_impacted_block_coverage="2236" blocksaddedtoset="32" totalcoverage="2236"
          cumulativecoverage="3427" />
        <trace IDTrace="27214" impacted_block_coverage="2053" new_impacted_block_coverage="0"
          old_impacted_block_coverage="2053" blocksaddedtoset="30" totalcoverage="2053"
          cumulativecoverage="3457" />
        <trace IDTrace="27153" impacted_block_coverage="2001" new_impacted_block_coverage="0"
          old_impacted_block_coverage="2001" blocksaddedtoset="18" totalcoverage="2001"
          cumulativecoverage="3475" />
        <trace IDTrace="13548" impacted_block_coverage="2023" new_impacted_block_coverage="0"
          old_impacted_block_coverage="2023" blocksaddedtoset="10" totalcoverage="2023"
          cumulativecoverage="3485" />
        <trace IDTrace="746220" impacted_block_coverage="1533" new_impacted_block_coverage="0"
          old_impacted_block_coverage="1533" blocksaddedtoset="3" totalcoverage="1533"
          cumulativecoverage="3488" />
      </traces>
    </set>
  </sets>
- <traces_id_name_mapping>
  - <trace IDTrace="13550">
      <![CDATA[ Branch Root.qa.md.wd.Bizapp.AxPro.Data.DataSet.Designer
      [ds].Nightly.dsNightlyTableAdapterESP:1.1.385947.35669.716663.P0.Pass  ]]>
```


Figure 5: Sample output from TCIndexer - Test Selection Tool

Figure 5 shows the XML output that is generated by the tool embodying the algorithm. It generates a detailed report for each binary in the product as well as a summary report. Each detailed report indicates the number of code blocks being considered; in this case, 7103 code blocks. Furthermore, it indicates how each test case identified by a *<trace IDTrace/>* attribute contributes to overall product coverage as well as providing a *cumulative coverage* block count. As can be seen in this example, the *blocksaddedtoset* slowly decreases and the cumulative count rises correspondingly to its final value of 3488 code blocks, which is equal to the original coverage level for that binary. Also, for each test selected, the tool outputs the complete trace string, a piece of meta-data that describes

the test case and which includes information such as test priority, test case name, pass/fail execution status, etc. In Figure 5, the test cases are highlighted within the <*trace_id_name_mapping*> attribute.

With data collected and processed, Figure 6 shows the results of applying the Maxcoverage algorithm to our three sample product suites. The stacked bar chart indicates the number of tests that are part of the core set (Series 1) with potential duplicate tests represented in the upper stack (Series 2). The x-axis depicts the cumulative number of tests in the test suite by test priority, while the table shows data values. So, for example, in the second bar chart, the set of P0/P1 tests contains a total of 4814 tests with 1230 being identified as part of a core set and the remaining 3584 tests being deemed potential duplicates by virtue of not being included in the core set.

The key point of interest is the ratio of core tests vs. the potential dupe sets. For example, there is significant variation in this ratio for the P0 tests, the set of tests that should be run on a nightly basis. In the second chart, the algorithm identified a set of 845 P0 tests and 2160 potential duplicates, indicating that nearly two-thirds of the tests were potentially redundant. Compare this to the first chart, where the algorithm was not able to identify any potential duplicates and thus the core set reflected the original test set! This indicated to us that this set of 59 P0 tests provides broad and unique product coverage with every single test case. In contrast, the other two suites show appreciable potential redundancies and would need to be reviewed.

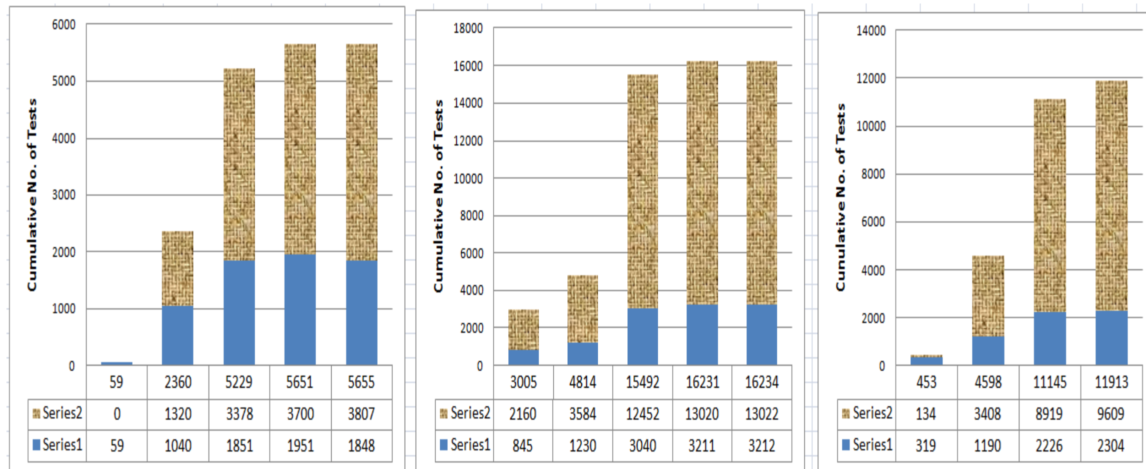

Figure 6: Ratios of Core Set vs. Potential Duplicate Tests by Test Priority

4.2 Reviewing Potential Duplicate Tests

Prior to reviewing and analyzing potential duplicate tests, it is important to realize and understand the limitations of code coverage and how to leverage the generated data discussed above. The code coverage data is being utilized here simply to indicate how each functional test in the suite has traversed the product code. However, it does *not* capture any temporal (timing) or frequency data for that test case. It does not, for example, indicate the order in which a given set of code blocks was executed or the number of times a code loop was taken. We only know that at some point, a specific code

block was traversed by a test. Thus, two trace vectors resulting from the execution of two different tests maybe identical in terms of the code blocks executed, but functionally those two tests may be targeting different customer scenarios or they could be variations of the same or similar customer scenario – we simply cannot tell. Having said that, code coverage data is readily available and the above analysis is 'cheap' enough in terms of performance and scalability.

4.2.1 Importance of Stakeholders

A first and crucial step in the review process is to ensure that the results generated above need to be disseminated to key stakeholders, that is, the domain experts within the product test teams. These stakeholders should represent senior members of each test team who are familiar with their feature areas and the tests that traverse the corresponding feature code. They also need to have worked on the team for some time and have knowledge regarding the evolution of their test set. The reason for emphasizing this point is that we found appreciable differences between offshore and in-house staff when reviewing and analyzing the potential duplicate sets and comparing them to core sets. The experienced (and thus more confident) senior testers were able to identify a larger number of *actual* duplicate tests. In contrast, offshore testers were less familiar with the tests (and thus more cautious) at deciding whether a potentially dupe was *actually* one. Thus, the number of actual dupes identified by those offshore teams can be significantly less.

4.2.2 Sample Recommendations

Once the stakeholders know the tests contained in the core set, their task of comparing these to the set of potential dupes for their feature areas begins. As this is ongoing test initiative, we are continuously evolving and refining the recommendations for other teams to leverage. Here are some samples of the prescriptive guidance provided:

- **Sub-scenario Tests** – many potential dupes exercise sub-scenarios of larger scenario tests. Our recommendation is to archive these, whenever the core set already contains a larger scenario test that 'covers' the sub-scenario or if appropriate, merge a number of sub-scenario tests into a new scenario test that can be added to the core set.
- **Regression Bug Finders** – if the potential dupe set contains tests that have found important regression test bugs, it would be wise to keep these and add them to the core set.
- **Tests with Extensive Verifications** – when considering test code, many potential dupe tests differ from their core set counterparts or other dupes by the amount of verification code that has been added at key points. Our recommendation has been to either keep all of these tests or merge them into a smaller set of tests.
- **Boundary Tests** – it is very unlikely that code coverage could distinguish these types of tests unless a significantly different execution path has been taken. Thus, our recommendation is to keep these tests.

- **Data-driven Tests** – some potential dupes represent scenarios in which Visual Studio code, for example, interfaces to SQL. Code coverage cannot reflect this interaction via traces and thus all of these tests must be retained.
- **Scenario Ordering** – as code coverage data cannot identify code execution order, scenario tests that have exercised the same code blocks, but in a different order cannot be distinguished. Thus, these potential dupes need to be examined more closely. As a result, the dupe is either archived, if its intent is to validate the equivalent customer workflow/functionality or added to the core set, if the intent is distinct from the test(s) in the core set.

5. Preliminary Results and Analysis

5.1 Goals

Before sharing some of our *preliminary* results from this ongoing study, we wanted to state its goals:

- Examine the accuracy of the Maxcoverage algorithm at picking a core set of tests
- Identify the ratio of potential vs. actual redundant tests (as determined by testers)
- Compare the fault detection capabilities of the final test set vs. the original set[3]

At the time of writing, about a dozen teams from across Developer Division are actively participating in the case study and working on various stages of analyses. These include test data preparation, test suite characterization, potential duplicate test identification and potential dupe review and deprecation.

The teams can best be characterized as representing products ‘up and down the Visual Studio stack’ from the Visual Studio platform to Visual Studio Team System. The teams own code and test suites that vary in terms of size and age. Focusing on the test aspect - some teams own only a few hundred tests that have been developed over the past one or two product release cycles; others own suites that have been around for ten years or more and include thousands of tests.

5.2 Results from Participating Teams

For this paper, we wanted to share the early results from two product teams that have completed one of the later phases, the potential dupe review phase. Neither team has undergone any kind of optimization of their test suite before. The first team maintains a true legacy test suite of nearly fifteen thousand tests, which have evolved over ten years

[3] Unfortunately, at this time, we are unable to report on the fault detection capability of the final set of tests as none of the teams has had sufficient time to complete the potential dupe review and then compare and draw conclusions concerning the fault detection capabilities of the original and final set of tests.

or more. The second team maintains a more recent suite of tests numbering several hundred tests.

The first team decided to analyze their test suite based on key features, so the results in Figure 7 represent the results for one of those key features, rather than for the entire product[4]. If we examine the data in Figure 7, we see that testers had developed approximately 1600 tests of varying test priority for this feature. The domain expert was then presented with a set of core tests (746 tests) and potential dupes (883 tests) as determined by the Maxcoverage algorithm. He then examined and compared each test case from the core set with each of the potentially redundant tests. This comparison was conducted at a rate of about 5 minutes per test comparison.

Of the 883 potential dupes, 715 tests were actually found to be redundant and could initially be disabled in the test case management system (and later on deprecated). This indicates that there was only a 10% variation (accuracy) in the number of core tests originally calculated by the algorithm and the final core set determined after the manual review phase.

CATEGORY	TOTAL	% of total
Total # of tests (P0/1/2/3)	1629	100%
MaxCoverage set identified	746	46%
Potential dupe set identified	883	54%
Actual dupe set (after review)	715	44%
Final set (MaxCoverage+(potential-actual))	914	56%

Final set lies within 10% of computed Maxcoverage set

Figure 7: Results from Team #1

For the second team, we can share the results for the entire suite as their test cases numbered in the hundreds and their domain expert had knowledge of their entire suite. Their original test suite comprised of over 650 tests with the Maxcoverage algorithm identifying approximately 200 tests as the good core set and the remainder (over 450 tests) as potentially redundant. After analysis, they too found that the algorithm had identified the core test set to within 10% accuracy. So, the final test set was just over 200 tests.

[4] Test suite characterization results for this team and their entire suite of tests can be found in the chart 3 of Figures 1, 2 and 6.

5.3 Benefits and Lessons Learned

In this section, we wanted to highlight key lessons learned so far as well as discuss benefits. While both teams agreed that performing such an exercise was time-consuming from the point of view of analyzing the potential dupes manually, the reductions being achieved in both cases, more than offset that cost.

5.3.1 Test Execution (and Failure Analysis) Time

A good example for improving test agility comes from the second team described earlier. Its suite of more than 650 automated UI tests required more than 2 days to run on a single machine with several team members needing to investigate failures. Their tests were unstable and frequently failed. After completing the above exercise, the team then focused on improving test reliability. As a result, this smaller set of tests can now be run overnight as part of a nightly test run with tests passing at a much higher rate and only one tester required for analyzing failures. Product (code) coverage has been verified as being equivalent of the original test suite. By focusing on this new core set of tests and test failures from these tests, the team (re)discovered three high-priority (P0/1) bugs in subsequent test runs. Previously, they had been swamped with test failures and had missed them.

5.3.2 Test Planning and Review

Participating in this exercise, the first team realized that they required improvements to their test case planning and test code review processes to prevent the most common mistakes being made by testers in future. One example that is driving the need for more careful code reviews is the existence of large numbers of sub-scenario tests that had not frequently enough been reviewed and merged into larger scenario tests. Another example highlighted a loophole in their test tooling/processes that resulted in testers being able to create tests that contained setup/teardown code, but no actual test case body. As a result, these ‘junk’ tests were not exercising the product, but were always passing! It turns out that appreciable numbers of such tests were wasting test resources. The exercise also helped in reviewing and prioritizing the remaining scenario tests in terms of importance and risk.

5.3.3 Test Code Knowledge and Regular Checkups

A key lesson learned by both teams has been that product coverage and value added by new (and existing) need to be assessed on a regular basis in order to maintain test effectiveness and efficiency. This also benefits team members’ knowledge of the test and product code, which given the staff turnover is very valuable. As part of this study, we are recommending that teams consider regular checkups to maintain their test agility – prior to test passes (in the test preparation phase), at major milestones (MQ or milestone quality) or on a more frequent, *per tester* basis. The latter would be the most effective model whereby testers themselves leverage the tools described in this paper and to ensure efficiency and effectiveness of their own tests.

6. Conclusions and Future Work

In this paper, we have examined techniques and tools that leverage code coverage data to help test teams focus on their test suite effectiveness and efficiency. We highlighted the application of techniques and tools to improve their test agility and the steps that need to be taken to achieve that goal. Furthermore, we illustrated this process by means of an ongoing pilot involving test teams from the Developer Division and reported preliminary results. Finally, we reflected on key improvements that teams are making as a result of participating in this exercise.

It is also important to note that at this time, the review and analysis process is completely *manual* in nature. Having said that, we are seeking and evaluating new tools that promise us better duplicate test identification as well as reduced review and analysis times. Currently, we are exploring the Spirit tool [4] that not only compares code coverage traces and analyze their differences and similarities, but also leverages data from execution profiling and static analysis techniques applied to the product binaries to accelerate this otherwise arduous manual task. We will be reporting the results and impact of this tool on our initiative in a future conference paper.

7. Acknowledgements

I would like to thank Larry Sullivan, Director in Developer Division, for his ongoing support. I also want to express my gratitude to my Microsoft colleagues, particularly Carey Brown, Xiang Zeng, Shaun Phillips, Mike Taute, Alex Teterev and Jacek Czerwonka for their contributions, support and discussions concerning this work as well as the Developer Division code coverage champions for helping me collect the necessary data. Thanks also go to my reviewers whose invaluable feedback and comments are greatly appreciated.

8. References

1. J. Hartmann, "*Applying Selective Revalidation Techniques at Microsoft*", PNSQC, pp. 255-65, Oct. 2007.
2. A. Srivastava and J. Thiagarajan, "*Effectively Prioritizing Test in Development Environment*", International Symposium in Software Testing and Analysis (ISSTA), pp. 97-106, 2002.
3. V. Chvatal, "*A Greedy Heuristic for the Set-Covering Problem*", Math. Operations Research, pp.233-5, Aug. 1979.
4. V. Vangala, J. Czerwonka and P. Thalluri, "*Test Case Comparison and Clustering using Program Profiles and Static Execution*", European Software Engineering Conference and ACM Foundations on Software Engineering, Aug. 2009.