2008

PACIFIC NW SOFTWARE QUALITY CONFERENCE

Collaborative Quality

October 13-15, 2008

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 commercial use.

Ensuring Software Quality for Large Maintenance Releases

Jean Hartmann
Test Architect
Developer Division Engineering

Microsoft Corp.
Redmond, WA 98052

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

Abstract

Like many divisions within Microsoft, Developer Division with its two flagship products - Visual Studio and Visual Studio Team System - allocates a significant portion of its time and resources to ensure the highest possible quality of its maintenance releases. Product teams within the division face the challenge of having to execute increasing numbers of automated tests against a growing code base in ever shorter development test cycles.

In a previous PNSQC paper[1], we described an approach that focused on leveraging so-called selective revalidation techniques using code coverage data, which provided a significant reduction in the number of regression tests that needed to be rerun for a given code change and better prioritized test suites. Thus, the focus of this paper is to describe and discuss how we deployed and evaluated these techniques as part of a recent maintenance test cycle.

Our guiding principles for a successful deployment/evaluation of this technology were:

- **Redefine the test exit criteria** to force a change in *test focus* from validating the entire product to validating only the modified portions of that product (aka code churn)
- **Enhance the test process and tool infrastructure** to support the collection, analysis and reporting of relevant data, so that test teams could gauge progress and declare success respective to those new test exit criteria
- **Enable testers to gain a deeper understanding of the code changes** and their impact on the product as well as requiring them to better quantify the (re)testing effort

This paper outlines the testing process associated with our maintenance releases. We reflect on some of the challenges that we faced in enhancing this process and tool infrastructure. Finally, to demonstrate the potential of these selective revalidation techniques, we use examples and data from this recent maintenance test cycle.

[1] J. Hartmann, "*Applying Selective Revalidation Techniques at Microsoft*", PNSQC 2007, pp. 255-65.

1. Introduction

Maintenance work on Microsoft products is characterized by scope and frequency. Within a division this work is either conducted by the product team during new product development work or by a dedicated Servicing Team that focuses on maintenance. For example, a team working on a smaller product, such as the C# compiler, would conduct its own maintenance, while the Windows organization would have a dedicated servicing team for maintenance for operating systems such as Windows XP/Server 2003.

Maintenance covers a wide range of activities in terms of required code changes and testing effort. In this paper, the emphasis is on large-scale maintenance releases, known as Service Packs (SPs), where the changes are much more substantial and less frequent in nature compared with smaller-scale patching efforts. These releases require collaboration and coordination from many teams within an organization. They touch many features in different ways with resulting code changes ranging from new binaries being added to existing code being re-factored.

The nature of such large-scale maintenance releases has the potential for introducing not only bugs within the individual Visual Studio products, for example, the various language compilers, but more importantly bugs resulting from the integration of these individual products into the overall Visual Studio product due to numerous dependencies. Thus, the motivation and goal for improving our regression testing strategy is to thoroughly exercise the individual products as well as eliminate system integration issues without having to rerun all our test suites repeatedly.

In a previous paper [1], a methodology for performing selective revalidation was discussed. In that paper, we focused on how to effectively and efficiently select a subset of regression tests based only on the code that has changed or churned. Here, we explain how this methodology benefited large maintenance releases within Developer Division.

2. Selective Revalidation Techniques – Recap

While selective revalidation techniques were discussed in detail in a previous paper [1], it is important to briefly recap the key facts concerning these techniques and highlight those aspects that contribute to the discussion in this paper.

Selective revalidation techniques are a class of testing techniques that guide testers in the selection of suitable subsets of regression tests or assist in prioritizing tests in response to code changes or churn. The techniques described in the previous paper and applied here use code coverage data which was collected at a previous product release milestone, plus linear programming algorithms known as 'greedy algorithms' [2], to determine a minimal set of tests to rerun for the upcoming maintenance milestone.

The techniques address two issues, both with the same end goal of streamlining the regression testing process. One technique is known as *regression test selection*, the other as *test set optimization (or test prioritization)*. The key distinction is that the former utilizes code churn data in addition to code coverage data to determine a minimal solution that focuses on tests traversing modified code. The latter simply identifies a minimal and thus prioritized set of tests from the test suite without regard for where code changes have occurred.

The previous paper [1] also describes the various scenarios in which these selective revalidation techniques could be used. This paper is focusing on *post-verification testing*. In post-verification testing, code changes have already been checked-in and tested by the individual product teams. Now test passes are conducted on a larger scale to validate overall product functionality and in this case, to specifically satisfy a given exit criterion - code coverage – for the SP maintenance release.

3. Enhance the Test Process

While the previous paper [1] focused on describing the test methodology itself, this paper intends to focus on the next steps that we are taking to leverage the methodology and reap the benefits of resulting test efficiencies and effectiveness.

3.1 Redefine the Exit Criterion

Maintenance releases are subject to the same set of exit criteria as major product releases, which requires teams to satisfy a code coverage criterion by reaching 70% block coverage[2] for each executable that is part of the product[3]. This quality bar applies to both existing and new executables for a maintenance release. In the past, such a criterion led to adequate testing of the new code (executables) being introduced, but did not necessarily ensure that code modified in the existing code was adequately retested, if that code had reached the 70% bar for the previous product release. Regression suites were being rerun in their entirety with inadequate attention being paid to how many of those tests actually traversed the modified code segments. There was no means of assessing test thoroughness.

Therefore, the first enhancement that we made to the existing test process was to redefine this exit criterion subtly by stating that the quality bar was now to be applied to the *coverage of churned code*. This shift in test focus has the benefit that testers were forced to examine and better understand the code changes as well as quantify and plan the test effort required for those changes. As a result, testing a change was not simply a binary statement of whether or not the change had been tested, but a quantifiable measure of how thoroughly the change had been tested – a big step forward.

[2] A basic block is a sequence of binary instructions without any jumps or jump targets. In other words, a basic block always (in the absence of exceptions) executes as one atomic unit (if it is entered at all). Because several source lines can be in the same basic block, for efficiency reasons at execution time it makes more sense to keep track of basic blocks rather than individual lines

[3] This coverage adequacy criterion and its specific value of 70% is a corporate-wide standard.

	Coverage by Churned Code Block		Total Coverage	
Binary Name	Affected Code Blocks	% Blocks Covered	Total Blocks	% Total Blocks Covered
binary 1	23	82.60%	2085	69.10%
binary 2	78	91.00%	4907	78.20%
binary 3	206	71.80%	15795	67.80%
binary 4	533	79.20%	4563	74.10%
binary 5	70	75.70%	320	82.80%
binary 6	15	100.00%	5738	99.30%
binary 7	33	97.00%	41183	74.90%
binary 8	107	83.20%	6957	78.00%
binary 9	119	95.00%	2694	76.40%
binary 10	10	100.00%	955	66.20%
binary 11	26	76.90%	12515	74.50%
binary 12	22	72.70%	7643	67.20%
binary 13	47	85.10%	239	77.40%
binary 14	63	79.40%	1029	61.10%
binary 15	46	84.80%	264	80.30%

Figure 1: Code Coverage Statistics for Selected Binaries

Figure 1 above shows the coverage metrics collected by a Developer Division team as they adopted this redefined criterion. Under the old coverage criterion of 70% block coverage per binary, several binaries, such as binaries 10 and 14, would have required additional test effort to meet the quality bar. This would have resulted in additional tests being created without knowledge of whether those tests actually traversed the churned code and ensured the quality of that churned code. By applying the new criterion to those same binaries, we can clearly see that the changes have not only met the quality bar, but have exceeded it. Availability of such data enables test managers to better monitor test progress and its completion with respect to code changes - they are no longer simply relying on the assurances of their staff.

3.2 Improve Tool Infrastructure

Due to the scale of our current testing processes, adoption of new test processes and technology is only viable when supported by reliable tools. This support must be defined and deployed in such way that it has minimum impact on the existing test infrastructure and effectively complements it. Figure 2 illustrates the new and existing tools and their flow; their usage will be explained in more detail in the next section.

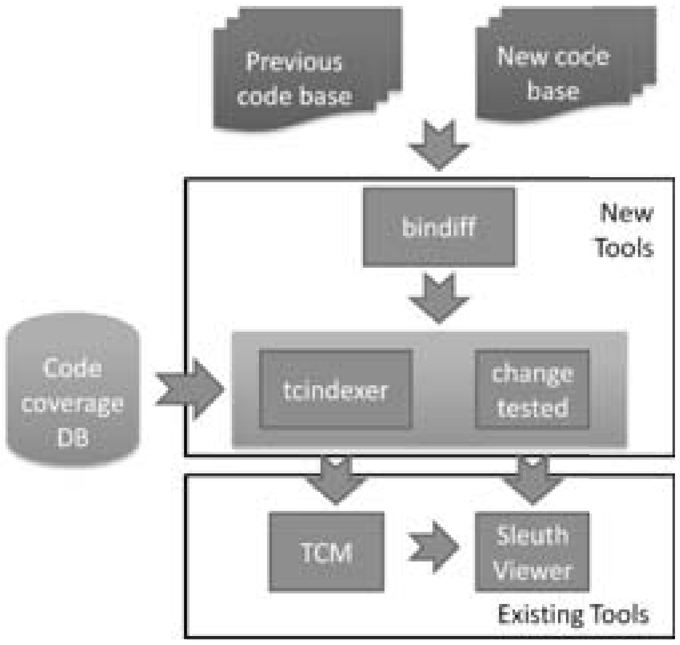

Figure 2: Tools and Flow

In prior testing cycles, the environment consisted of the code coverage database, the *Test Case Management (TCM) tool* and the *Sleuth Code Coverage Viewer* tool. In order to measure code coverage during the maintenance test pass, the code base was compiled and the resulting binaries instrumented for block coverage. The tests stored in the TCM were executed and the testers ensured that they had met the quality bar by viewing the resulting coverage data in the Sleuth code coverage viewer tool. These processes leveraged the Magellan toolset [3], a suite of tools for instrumenting compiled code binaries and then collecting, analyzing and viewing the resulting code coverage data.

The new tool flow adds a binary differencing (*bindiff*) tool that compares each binary in the previous (baseline) and current product builds and generates an XML file detailing the differences between builds on code block basis. This file is then consumed by two tools, *TCIndexer* and *ChangeTested*, together with code coverage data associated with the previous and current builds, respectively. This tool is also part of the Magellan toolset.

TCIndexer, the selective revalidation tool, leverages the code churn data for the two builds and the code coverage data from the baseline build to compute a regression subset or prioritized test set that can be rerun to exercise the current build and collect new code coverage data. *TCIndexer's* XML-based results are delivered with each new instrumented product build during a test pass and can be consumed and filtered by the various test teams for their relevant test cases as the solution contains tests (contributions) from other test teams. Its output file can also be consumed by the TCM tool, so that the process of rerunning could be fully automated, if the teams wanted to do so. *TCindexer* is built on top of yet another tool in the Magellan toolset.

While *TCIndexer* provides guidance for test selection, the *ChangeTested* tool can be regarded as feedback for the teams on how much testing remains before the churned code has been adequately tested. Again, the tool leverages the code churn data for the two builds and code coverage data. However, this coverage data is reflected in a newly created and then forward-merged coverage database that coincides with the start of the maintenance test pass. Metaphorically speaking, you are at base camp respective coverage of the churned code, looking up the mountain that you need to conquer and getting feedback with each new product build (step) as to how you are progressing with testing the churn.

The output files produced by *ChangeTested* include a coverage report for the churned code, supplemented with details of the churn. In fact, the table shown in Figure 1 is derived from such a coverage report. These reports are not only consumed by the individual teams, but are rolled up to an organization-wide website displaying all exit criteria relevant to the maintenance release. Figure 4 shows the code coverage reporting page. The other file being output is a *Sleuth* coverage viewer filter that testers can use, whenever they view the coverage data superimposed on the code. The filter file focuses and guides testers as it directs the viewer to display only those code functions in which code blocks have churned. This helps testers to more quickly understand the churned code, its impact and additional testing that may be required.

4. Testing a Service Pack Release

Figure 3 outlines the enhanced regression testing process flow for a recent service pack release based on the redefined exit criterion and improved tool infrastructure described above. A step-by-step explanation of the new process is given.

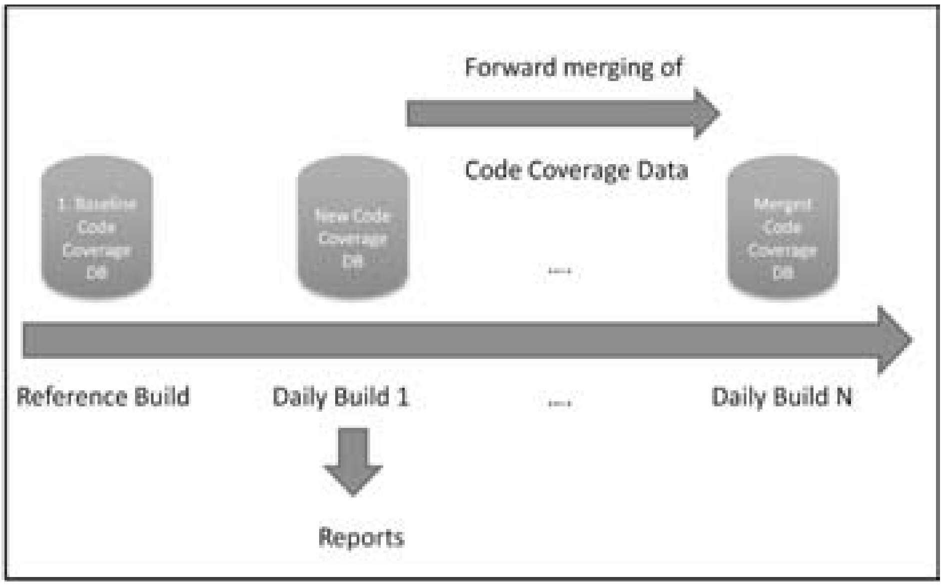

Figure 3: New Regression Testing Process Flow

Before test teams could apply this technology, process and tools, they had to collectively run their test suites against an instrumented product (reference) build for each major product release. This collected code coverage data, which then became the baseline code coverage database for subsequent maintenance releases. This baseline met the code coverage quality bar of 70% block coverage.

As the date for a service pack release grew closer, a new code coverage database was created alongside stable, instrumented daily product builds. This database was used to capture code coverage data from each team as they contributed to the testing of the service pack. The intent was that each team conducted a test pass using their regression tests (with or without guidance from *TCIndexer*) and then supplemented them, if necessary, with new tests. During and upon completion of the test pass, teams measured their progress and success against the exit criterion using *ChangeTested*.

As new product builds were released and teams executed their tests against those builds, a central team with Developer Division ensured that all coverage data was aggregated for the entire duration of the test pass. The final, merged code coverage database along with the product binaries was archived. This data now has the potential to be leveraged at a later date as baseline data for additional test passes where the above process would be repeated. The result was that test teams would reduce their regression testing workloads, the closer they got to the final release date for that service pack.

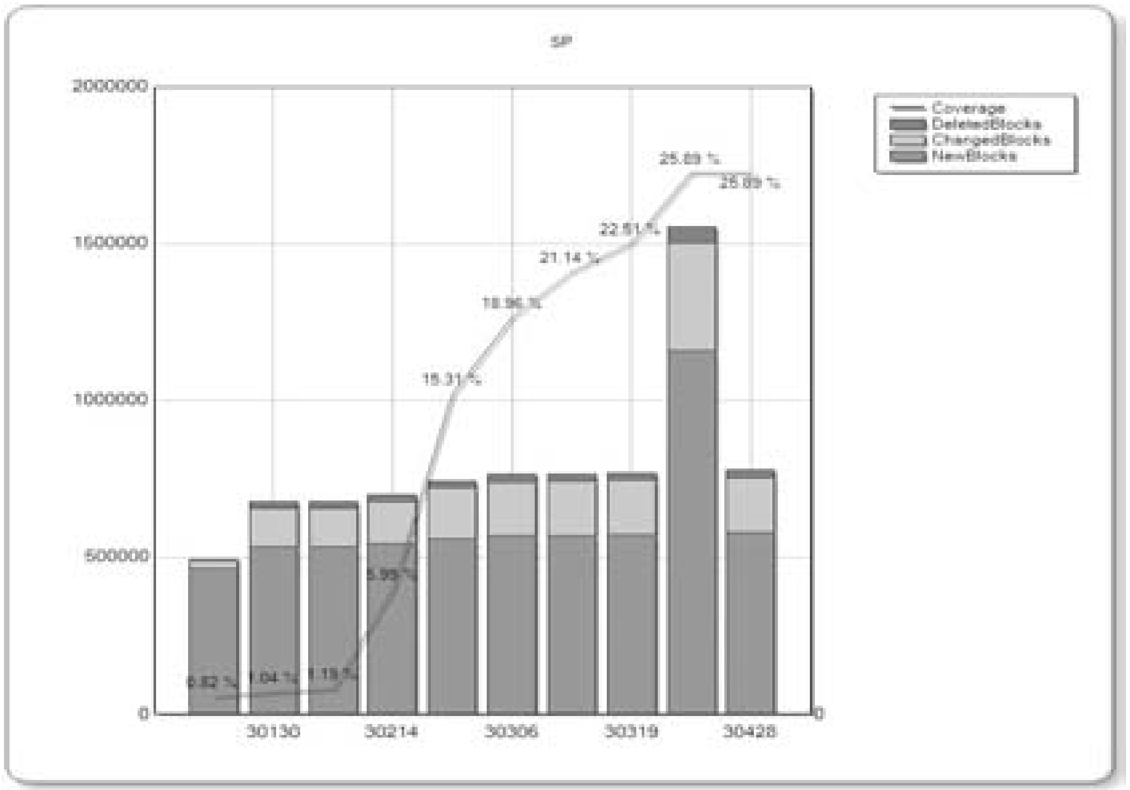

Figure 4: Code Coverage Exit Criteria Reporting

During this entire process, the output from *ChangeTested* (in the form of reports) showing % coverage of the churned code were updated and delivered to our ship room to give an indication of how close we were to meeting the coverage quality bar.

Figure 4 shows the type of data being collected during the initial stages of a test pass. The x-axis indicates the successive builds of the same product over time., while the y-axis indicates the number of code blocks being churned for each new product build compared with the baseline code binaries. The stacked bar chart indicates the contributions being made by the different types of churn – deleting, adding or modifying code blocks! This chart clearly shows the impact and timeframe of code integrations from different team branches into the main product build branch. Moreover, we superimposed the coverage data upon the code churn data via a line chart to provide a unified view.

5. Benefits and Challenges

While we only recently started to deploy this approach for service pack testing, the test teams saw two immediate benefits regarding their test focus. They were able to quickly identify test holes corresponding to churned code and create additional scenario tests to exercise that code. They also appreciated the diminishing workload associated with code coverage test passes required at each release milestone, because they only needed to validate the churn that had occurred since that last milestone. Examples of this were discussed in Section 3.1. Unfortunately, this work is so recent that we are unable to make any kind of statement regarding the impact of this approach on defects.

The biggest challenge was technical in nature and has the potential to seriously impact this new process. When code coverage data is initially collected in a reference database, a persistent mapping results between a test case and the binary code blocks it traverses. If changes are made to the compiler used to compile source code into a binary, then it may cause different binary code to be emitted. Thus, even if no changes were made to the source code for that binary, compilation using that latest version of the compiler would cause the binary differencing tool to flag incorrect differences between the previous and new versions of the binary. *TCIndexer* would then consume those differences and attempt to compute an incorrect subset of tests for rerun, In addition, *ChangeTested* would show incorrect churn that would need to be covered[4]. In summary, compiler changes can lead to a chain reaction resulting in incorrect data being presented to test teams.

This challenge was mitigated to a certain extent by recompiling the baseline build binaries with the same compiler version used to produce the latest product binaries. That enabled us to at least monitor progress regarding coverage of the churned code via *ChangeTested*. However, it left us with fewer options regarding selective revalidation as the binary differences that were being determined could not be used to help determine a subset of tests for rerun.

[4] *Note: This situation is relatively unusual as many Microsoft product teams maintain the same compiler version for building their products over long time periods in which case, the above approach works well.*

In summary, two approaches are being discussed:

- ∉ Base the above methodology on source- rather than binary-level differencing to overcome the compiler issue. We can then fully leverage selective revalidation and retain the redefined exit criteria.

- ∉ Use *test set optimization/prioritization* to determine a subset of regression tests, effectively ignoring any code changes. This would result in larger numbers of regression tests being rerun compared with subsets selected based on code churn and we would need to revert back to the existing exit criteria of % coverage per binary.

6. Conclusions and Future Work

Regression testing remains a very important topic for companies such as Microsoft who are developing and maintaining large software platforms and applications over long time periods. Testing technologies, such as selective revalidation, have the potential to deliver huge savings in maintenance costs for the company. We are now starting to deploy these technologies, albeit with a few challenges, and gather the necessary data to make that case. This paper chronicles how we are starting to deploy selective revalidation technologies and adapting our tools as well as processes to leverage them effectively.

In future, we intend to apply the test exit criterion to the validation of the individual products before they are integrated into the Visual Studio product (suite) *as well as after*. This will contribute to the quality of our individual products as well as the overall product. We are therefore working on replicating the above methodology, processes and tools for each product test team. The result is two instances where the test exit criterion needs to be satisfied. The test teams would thoroughly test their products prior to final integration and then rerun a representative (sub)set of their tests against the overall product suite. The latter would also have the beneficial effect on contributing/improving the coverage numbers of other product teams.

While the processes and tools outlined in this paper are internal to Microsoft, the concepts described here are easily repeatable and implementable using commercial-off-the-shelf tools including code coverage analyzers, test execution harnesses and management systems, differencing tools, etc.

7. Acknowledgements

I would like to thank Mak Agashe and David Yee, Directors of Test in Developer Division Engineering for their ongoing support. I also want to express my gratitude to my Microsoft colleagues, particularly Carey Brown, Apurva Sinha and Ephraim Kam for their significant contributions, support and discussions concerning this work as well as to the Developer Division code coverage champions for their feedback as we continue to deploy and refine these technologies throughout the division. 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 2007, pp. 255-65.
2. V. Chvatal, "*A Greedy Heuristic for the Set-Covering Problem*", Math. Operations Research, pp.233-5, Aug. 1979.
3. A. Srivastava and J. Thiagarajan, "*Effectively Prioritizing Test in Development Environment*", International Symposium in Software Testing and Analysis (ISSTA), pp. 97-106, 2002.