Focusing Test Efforts While Servicing Large Systems

Jacek Czerwonka
Microsoft Corp.
Redmond, WA 98052
jacekcz@microsoft.com

ABSTRACT

Building large software systems is hard. Maintaining large systems is equally difficult. Making post-release changes requires not only thorough understanding of the architecture of a component about to be changed but also its dependencies and interactions with other components in the system. Testing such changes in reasonable time is a tough problem on its own as infinitely many test cases can be executed for any modification. What if you had to make such modifications daily, maintain reasonable turnaround time, and you had a few hundred million users intolerant of even the smallest mistakes? This is the challenge Microsoft's Windows Serviceability group faces each day.

One way of battling complexity of software re-testing is to ensure that appropriate information is collected and used to guide and focus test efforts. Data needs to be modification-specific but should allow testers to understand system-wide implications of the change and risks it involves. The Windows Serviceability team uses a set of software metrics that are based on both static analysis of source code and dynamic analysis of tests running on the system. Both aspects come together and deliver data used to predict risk and impact of a change and to guide re-testing of modified code by answering the following questions: Which parts of the change are the riskiest? Which existing test cases should be executed to maximize the chances of finding defects? Which parts of the change will not be covered by existing tests? What dependent pieces of code need to be re-tested?

This paper presents an analysis of the challenges behind servicing Windows operating system, ideas behind our system of collecting and reporting change-related data, implementation details, and some of the results we were able to achieve with it. Above all, this paper will show how appropriately collected and applied data exposes an additional layer of detail which supplements testers' own skills and experiences and allows for better understanding of testing required in the context of a specific code change.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—complexity measures, product metrics.

Keywords

Software maintenance, software metrics, large systems, risk and impact evaluation

1. MAINTENANCE TESTING

Software maintenance is a set of activities associated with changes to software after it has been delivered to end-users. The IEEE Standard for Software Maintenance (1) defines three main types of maintenance activities:

1. **Corrective** maintenance: reactive modification of a software product to correct discovered faults, which includes:
   a. **Emergency** maintenance: unscheduled corrective maintenance performed to keep a system operational.
2. **Adaptive** maintenance: modification performed to keep a computer program usable in changed or changing environment.
3. **Perfective** maintenance: modification to improve performance or maintainability.

The amount of effort going into each of these categories will vary depending on the nature of the software considered, its intended purpose, size and characteristics of its current user base. However, the software maintenance phase exhibits attributes that are common across different software products:

1. Software maintenance is expensive. It is generally accepted that the maintenance phase consume the majority of resources required to take a software product throughout its lifecycle, from inception until end-of-life. The total cost of maintenance is estimated to comprise 50% or more of total life-cycle costs. (2)
2. Maintenance work is typically done by people who had not created the system. Unless the effective lifetime of a software product is relatively short, it should be expected that original designers, developers and testers are no longer involved in changes to the product. Reverse engineering might be necessary in absence of good documentation and institutional knowledge.
3. The maintenance staff is typically much smaller in size than the development staff required to create the product in the first place.
4. Changes in deployed software carry a high risk due to possibility of introducing unwanted behavior. Customer's tolerance to such breaking changes is low.

All of these characteristics influence testing during software maintenance phase. Testing of changes to deployed software is necessarily a different activity from software testing before release. Even though software maintenance testing deals with fewer changes, it typically needs to happen in very limited time and often with limited resources. On the other hand, increased risk and cost of making mistakes might warrant expanded test scope. These two competing forces create a challenging environment.

2. WINDOWS SERVICING LANDSCAPE

2.1 Customers

Customers interested in receiving updates to Windows can be very broadly divided onto four groups. **Home users** predominantly use the client version of the operating system like Windows Vista, Windows XP, or Windows MediaCenter and want to keep their PCs in good working condition. **Small and medium-size businesses**, as well as **enterprises** use both Windows server and client releases and want to protect their IP, avoid work stoppage, keep their maintenance costs low and want their investments in IT infrastructure working reliably. **Original equipment manufacturers (OEMs)** and **independent hardware vendors (IHVs)** produce PCs and devices working with Windows and want the operating system to support the newest hardware and help their customers obtain the best possible experience. **Independent software vendors (ISVs)** produce applications for Windows and need compatibility between different Windows versions and to have all the application features supported.

During the post-release phase any of these groups might be the primary target of a fix. All of our customers however expect two things when a Windows fix is requested: quality, defined as seamless integration and lack of change in behavior[1], and reasonable turnaround time between a problem is reported and having a fix ready for deployment.

2.2 Support complexity

A Windows fix is a packaged set of binaries. When installed, these new binaries will replace (or if completely new, be added to) the binaries existing on the machine. Let's look at the complexities behind producing a fix for Windows in the context of number of variations that need to be built and tested.

When a new Windows OS version is released to manufacturing (RTM), the servicing team assumes ownership of all the source code used to build that version of Windows and will use it to produce *hotfixes*. Post-RTM code changes are cumulative. When a fix to binary FOO.EXE is made for the first time, the resulting package will contain only change a_1. When a later change a_2 is made to the same binary, that new hotfix will contain binary FOO.EXE with both changes a_1 and a_2. This approach simplifies code maintenance but as time goes on and more fixes get implemented, the likelihood of the resulting hotfix containing more than just one change increases. Consequently, if any of the changes are faulty and the failure is not discovered early, quality of all subsequent hotfixes will be negatively affected.

To minimize this problem, the servicing team maintains two copies (*branches*) of the source code. They are identical at RTM. One branch is used for Global Distribution Releases (GDR) and contains only changes that are to be released very broadly e.g. through the *Windows Update* service. Another, used for Limited Distribution Releases (LDR), contains all changes regardless of the scope of distribution.

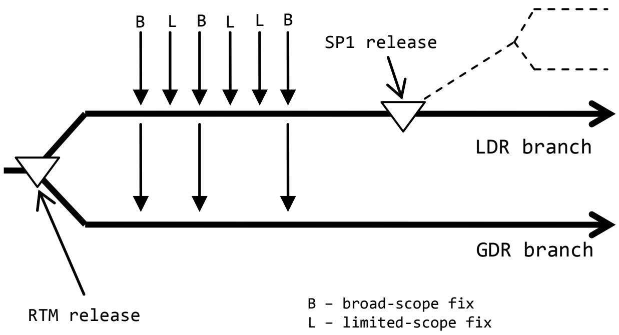


Figure 1. Servicing branch structure

The reason for having the GDR branch is to minimize the accumulation of changes in widely-applicable hotfixes in case any of them is faulty and causes the OS to misbehave. Whenever we can we use the binaries coming from the less-changing GDR branch and only use the LDR binaries if absolutely necessary.[2] See Figure 1.

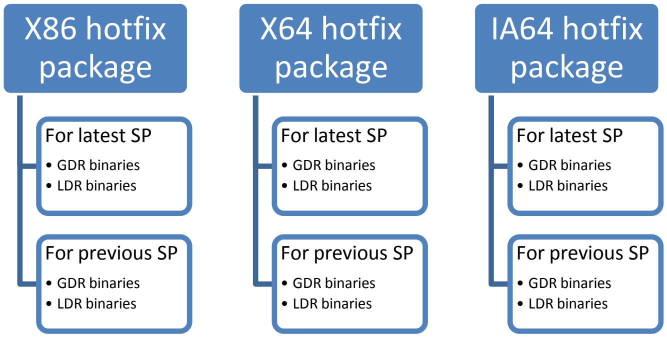


Figure 2. Eight possible payloads for one fix

At certain points in time all changes made to the LDR branch are rolled into a Service Pack. Apart from LDR changes, a Service Pack might contain additional fixes (perfective maintenance). Its

[1] If a change in behavior was intended, it should be backwards compatible.

[2] One reason to use LDR branch is because the fix was not intended to be very widely distributed and resides only in the LDR code. Also, when a binary on the system is already from the LDR branch, any new versions of that same binary have to come from the LDR branch; otherwise fixes already on the machine might be lost when a newer GDR binary is installed.

release signifies a start of another set of branches, which will be used for fixes intended to be installed on computers with that Service Pack version already deployed.

In practical terms a hotfix is an installable package that includes a set of new binaries. In the case of Windows, there is typically more than one package released for each fix. For example, there might be one package intended for Windows running on 32-bit (X86) and two for 64-bit (X64 and IA64) Windows. Before Windows Vista, each supported language was also packaged separately. Each hotfix package contains up to four sets of binaries. Separate GDR and LDR versions constitute one dimension. We support installing packages on the last two Service Packs therefore we create a separate set of binaries for each. Figure 2 shows all 12 basic package variations produced for each fix.

In the context of testing the differentiation does not stop there. Windows comes in a wide range of editions. For example Windows Vista has five major flavors: Home Basic, Home Premium, Business, Enterprise, and Ultimate (3). All Windows versions contain the same core set of binaries but their behavior or functionality limitations might be different between editions. A change to a binary which is subject to such different limitations, will likely trigger testing on more than one edition.

The last thing to consider is the time for which support is extended to customers. Depending on the version of Windows it, support will either be 5 or 10 years (4). During this time, th Windows Serviceability group will investigate problems, modify source code, build and test fixes.

All of these aspects bring to light the requirement for the Windows maintenance testing process to be very efficient. It needs to have the ability to identify with high accuracy the areas of software directly or indirectly affected by the fix and to bring up and allow analysis of differences between hotfix variations.

2.3 Hotfix testing process

2.3.1 Objectives

The primary objective of the hotfix testing process is to minimize the number of regressions, defined as unintended changes in the system's behavior, found in released hotfixes. Secondarily, we should be able to release fixes efficiently, within days or weeks rather than months.

2.3.2 Process

All problem reports related to Windows go through one of the product support channels. A small portion of these support cases turns out to be true code defects (corrective maintenance) or requests to change behavior (adaptive maintenance). The Windows Serviceability team assumes responsibility for implementing necessary changes in both cases and starts the hotfix request process (Figure 3).

Each hotfix request begins with *triage* which involves representatives of business management, development, testing, product support, and the customer in a brainstorming session on available and feasible workarounds, potential methods of fixing, and risks and efforts required from development and testing.

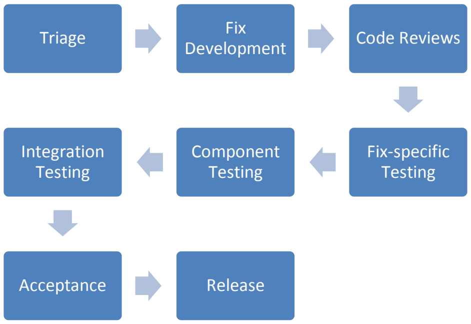


Figure 3. Hotfix release process

Assuming the fix is approved by all parties[3], developers then implement the fix and get it code reviewed. Testers at the same time, prepare and carry out their test plans.

Hotfix testing is typically carried out in three sequential phases:

1. **Fix-specific testing or unit testing**. This set of tests makes sure the fix itself is in fact correcting the original problem as expected.

2. **Component integration testing.** Internally, Windows is well componentized and all binaries have a place within some component area; interfaces between these areas serve as boundaries of component testing. The objective is to find and remove any regression in behavior within the component itself as well as at each of its interfaces.

3. **System integration testing**. The changed component might have dependencies i.e. other components that use it through its interfaces. Each such dependent component might require re-testing, at least in places where it calls into changed code.

When reasonable confidence in the quality of the fix is gained, it is sent to the customer for final acceptance. Upon approval, the hotfix is released.

3. TESTING CHALLENGES

Testing in general is an open-ended activity and more tests can always be developed for any non-trivial software. In the context of software maintenance, the challenge is to find the smallest set of tests that maximizes the defect-finding power of its members. Re-running all tests in our collateral (millions) on each fix would give us a good understanding of the level of quality[4] but is not

[3] Fixes might not turn into a code change if the customer accepts a feasible workaround or the triage team concludes the risks are too high for the expected benefits.

[4] At least to the extent of the resulting system being of the same quality as the original release.

feasible from either the turnaround time point of view or cost incurred. On the other hand, not doing testing at all provides terrific turnaround time but we learn nothing about the quality of changed software. Some middle-ground must exist where enough testing is done to mitigate risks and enough information is gathered to assess the level of quality.

Knowing how much testing is needed involves having deep understanding of the software under test as well as the change itself and its effects, and selecting from existing or adding just enough new tests to uncover side-effects of the change. The main idea behind the work described in this paper is that additional information about the system and the change itself will help a tester do her job more efficiently and effectively. More specifically, in the context of the three aforementioned categories of tests we want to get answers to the following questions:

For the fix-specific testing:

1. Which parts of changed code will we reach with our existing tests?
2. For changes already covered with existing tests, are they enough?
3. For which parts of changed code do we not currently have any tests?
4. Which parts of the changed code carry the most risk of regressing existing behavior?

For the component integration testing:

1. Which tests from our component-specific test collateral should we run?
2. In which order should we run them so that we maximize our chances of finding problems early?

For the system integration testing:

1. Which other components will be affected by this change and with what level of indirection?
2. How exactly are other components affected (e.g. function calls of shared data)?
3. What are the paths that will lead to executing the dependency link between that component and the changed component?

4. PATCH ANALYSIS TOOLSET

The Windows Serviceability team has created a set of tools which intend to help testers analyze changes, identify risks and answer some of the questions mentioned above. The primary objective of the toolset is to expose information in a systematic way. There are six major data points we collect and report to testers in the context of a specific change. All of them are indicative of regression risk and consequently the needed scope of testing:

1. Detailed package content.
2. List of existing tests likely to uncover defects.
3. List of dependent components.
4. List of changed lines of code for which tests don't exist.
5. Quality of the test process for the changed area.
6. Regression probability (partial support at this time).

4.1 Package content

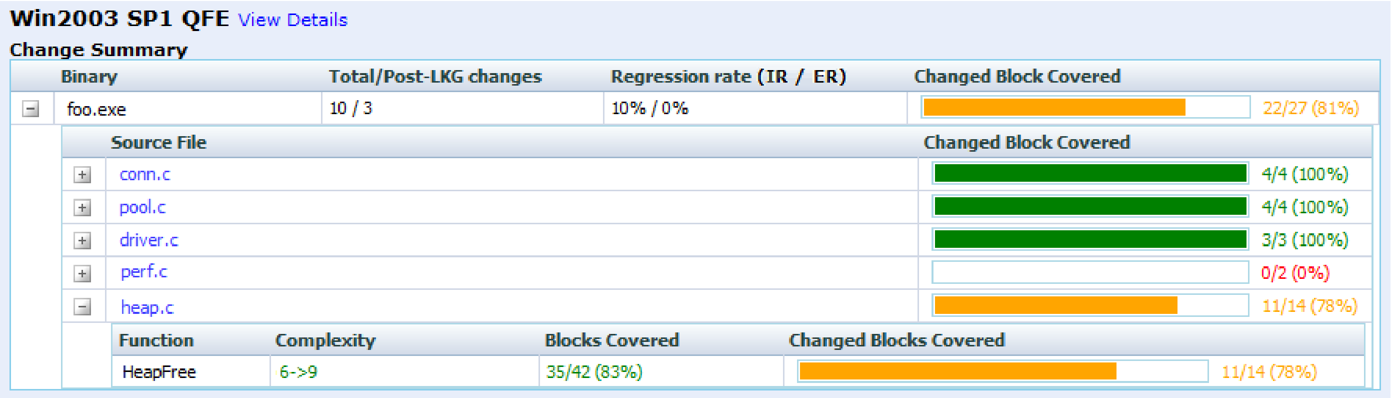
The first step to understanding impact and risk of change is to realize what exactly is being changed. It is not easy in many cases since source code might get compiled into multiple binaries and changed binaries might force inclusion of dependent binaries in the package as well. Moreover, even though testers are most often interested in perusing the last code modification, sometimes they also want to know the extent of changes done since the last broadly distributed release of said executable, for example the last Service Pack in which the binary was included. The reason is that broad releases have typically gone through a very extensive and rigorous test process and, more importantly, have already been deployed in the field and their level of quality is known.

Therefore, the first thing the Patch Analysis tools show is the number and the extent of changes done since the last known good (*LKG*) version of each binary. We show this on three levels: binary, source file, and at an individual procedure level. Figure 4 shows a sample change summary.

It is likely that the same change applies to multiple branches of code. Figure 4 only shows one branch. In the tool however we would show details for each baseline code branch affected by the fix. With that, testers would be able to look at the same change in multiple contexts and decide if full re-testing in each branch is necessary. Often similarities between code changes can be exploited to our advantage and test processes shortened.

4.2 Quality of the test process

In our test collateral, each executable has specific regression tests associated with it. These are supposed to exercise the binary's functionality in a deep way (e.g. unit tests, specific functionality tests etc.). As much as we want to have uniformity in quality and fault detecting power of such tests, that is very hard to achieve in the real world. Not only might tests vary in their comprehensiveness, which is relatively easy to correct, but also the nature of a specific binary, its design, or evolving environment will naturally make some areas more change- and regression-prone. Therefore, it is important to understand which binaries were regression prone in the past and for which binaries we have very reliable and dependable test suites.



Win2003 SP1 QFE View Details

Change Summary

	Binary	Total/Post-LKG changes	Regression rate (IR / ER)	Changed Block Covered
⊟	foo.exe	10 / 3	10% / 0%	22/27 (81%)

	Source File	Changed Block Covered
⊞	conn.c	4/4 (100%)
⊞	pool.c	4/4 (100%)
⊞	driver.c	3/3 (100%)
⊞	perf.c	0/2 (0%)
⊟	heap.c	11/14 (78%)

Function	Complexity	Blocks Covered	Changed Blocks Covered
HeapFree	6->9	35/42 (83%)	11/14 (78%)

Figure 4. Change summary

We show two numbers with intention of exposing the regression finding power of our internal test processes, namely historical internal and external regression rates for a given binary. The **internal regression rate** (*IR*) for a binary is a number of times the binary, when changed, had a problem in it and we found it before releasing it. The **external regression rate** (*ER*) measures the number of times we did not manage to find an existing problem in a binary and it was later found in the field. Both are shown as a percentage of the total number of releases done since the branch was created for the given binary. Naturally, you would think of a 50% regression rate differently depending on whether we had observed it over 2 or 20 releases. Therefore to give context to *IR* and *ER*, we also show the total number of releases (*Total changes*).

If either *IR* or *ER* is high, we advise testers to think of that binary as regression prone i.e. they should expect defects in it. Depending on which number is higher we can point out where they should focus their efforts: running their existing tests or adding new ones. In any case we want to minimize *ER*, problems found in the field.

4.3 Test coverage and test prioritization

For test prioritization, Patch Analysis uses Microsoft's Echelon tools (5). Echelon calculates differences between two binaries and then uses previously stored code coverage information to identify tests which will trigger execution of changed parts of the binary. It prioritizes tests according to their change covering ability, with tests covering most of the changed code at the top of the list. We do not recommend that only these tests are run but rather that they are run first. (See Figure 6).

There will be cases where certain portions of changed code might be identified as not covered through existing tests. These "test holes" are an important indicator of test effort required since ideally all changed code would be executed before the release. In our tool, a source level view of changes[5] represents this information in a form of "green" (covered by existing tests) and "red" (not covered) coloring of all changed lines of code. Our recommendation is that all currently uncovered parts of code have tests developed and executed for them.

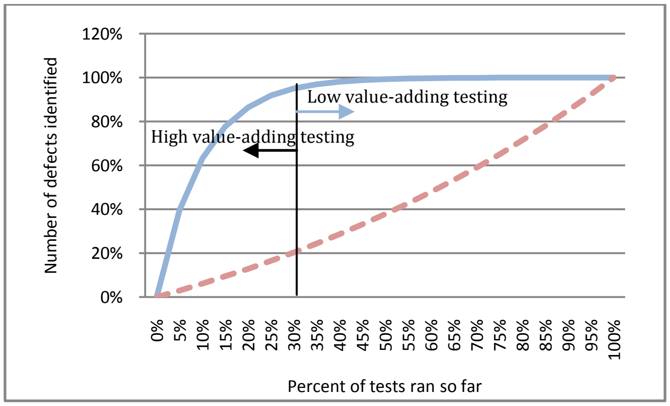


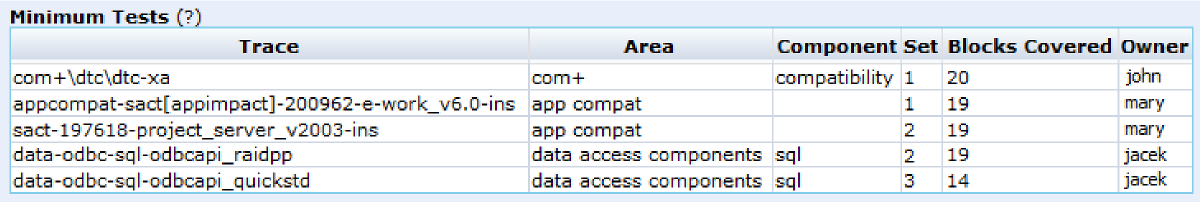
Figure 5. Well-prioritized test execution (solid line) vs. execution of tests not prioritized by effectiveness (dashed)

Test prioritization algorithms implemented in Echelon try to minimize the number of tests that cover the maximum changed lines of code. A better approach would be to try to minimize effort (man-hours, machine time or both) required to cover maximum changes. This is an area of further study.

4.4 Dependency identification

Since our code coverage database contains information on all tests developed for Windows OS, we might sometimes see that tests that were not specifically intended for the changed binary are identified as high priority tests. This is frequently an indication that some dependent binary will trigger execution of code in the changed binary. Our recommendation is that these tests should be executed as part of system integration testing. The number of such "foreign" tests is another indication of how far-reaching the change is and again an indication of the test effort required when testing.

[5] Not depicted in this paper.



Minimum Tests (?)

Trace	Area	Component	Set	Blocks Covered	Owner
com+\dtc\dtc-xa	com+	compatibility	1	20	john
appcompat-sact[appimpact]-200962-e-work_v6.0-ins	app compat		1	19	mary
sact-197618-project_server_v2003-ins	app compat		2	19	mary
data-odbc-sql-odbcapi_raidpp	data access components	sql	2	19	jacek
data-odbc-sql-odbcapi_quickstd	data access components	sql	3	14	jacek

Figure 6. Highest priority tests

However, even though our code coverage database provides us information about existing tests, in context of the entire system it is necessarily only as comprehensive as *existing* test cases. Since code coverage data is collected at run-time during code execution, if a piece of code does not get tested at all, there is no usable information for it. As a complementary measure, we should try to analyze the system in a way that is independent of the run-time execution. The idea here is that if we can discover dependency *potential*, this information can lead to additional testing in places where no execution was done before.

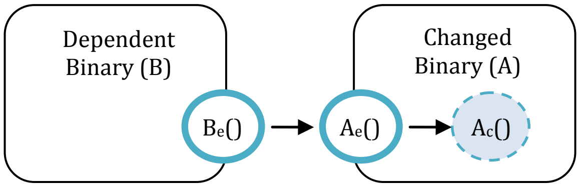


Figure 7. Static dependency analysis

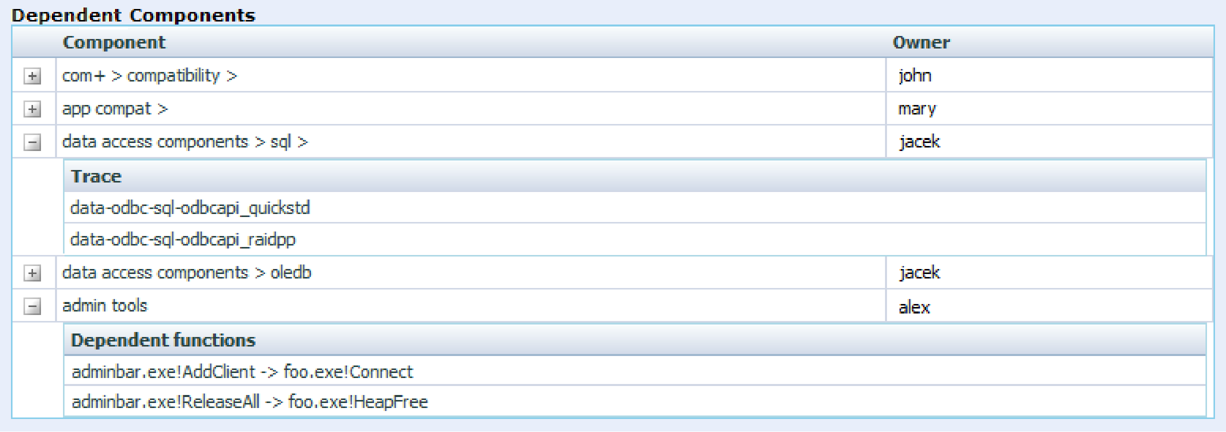
Patch Analysis use MaX tools (6) to do this analysis. First we generate a call graph for all binaries in the operating system which represents all discoverable places from which we can call any reachable procedure. If such a procedure is modified we can immediately determine all of its direct and indirect callers. Figure 7 depicts an example where a function A_c in binary A, reachable though entry-point A_e was changed. Function B_e in binary B is known to have call sites into A through A_e. Binary B is then identified as impacted by a change in A_c and will be added to the list of dependent binaries. Figure 7 depicts how, by putting together result of run-time (Echelon) and static (MaX) analysis, we are able to present a more comprehensive picture of dependent areas.

Dependency analysis methods we currently employ have their limitations. Currently, MaX is able to discover the most prevalent types of calls between procedures both in unmanaged and managed code. We currently do not identify dependencies that are triggered by data, for example if two binaries use the same file, configuration entries, or objects, we do not discover such cases. Further work is necessary in this area and we hope to add more types of dependencies over time.

4.5 Regression risk

The goal for this metric is to have a quantifiable measure of fault-proneness of a given binary. The idea is to try to understand and quantify the risk of a particular change given historical data on similar changes.

> *Failure-proneness is the probability that a particular software element (such as a binary) will fail in the operation in the field.* (7)



Dependent Components

	Component	Owner
⊞	com+ > compatibility >	john
⊞	app compat >	mary
⊟	data access components > sql >	jacek

Trace
data-odbc-sql-odbcapi_quickstd
data-odbc-sql-odbcapi_raidpp

	Component	Owner
⊞	data access components > oledb	jacek
⊟	admin tools	alex

Dependent functions
adminbar.exe!AddClient -> foo.exe!Connect
adminbar.exe!ReleaseAll -> foo.exe!HeapFree

Figure 8. Dependency list

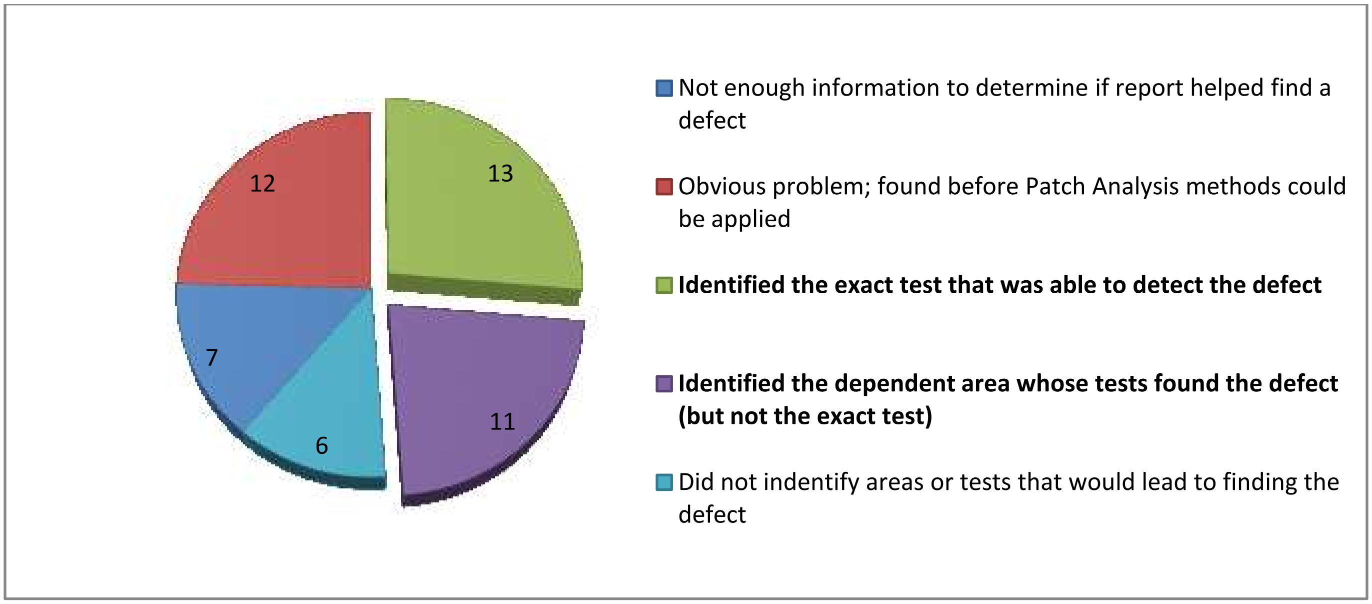


Figure 9. Effectiveness of Patch Analysis

Studies described in (7) and (8) concentrate on developing statistical models for estimating the number of defects that will be found after RTM based on pre-RTM characteristics of binaries. Attributes being considered as predictors of risk are: size of the binary, number of functions and global variables in a module, average number of lines of code per function, number of incoming and outgoing dependencies, maximum and total complexity of functions, depth of class inheritance. The aforementioned studies have shown that some of these have strong correlation with defects found after the release, specifically in context of the Windows code base.

The Windows Serviceability team is investing in repurposing methods described in (7) and (8) for our area of interest. We are currently working on recalibrating the prediction models. In the meantime, we use McCabe's cyclomatic complexity of changed functions (9) and provide their complexity before and after a change is implemented (Figure 4). Such presentation of data allows us to discover fixes where complex functions needed to be changed and fixes where complexity substantially increases. Both kinds of events should trigger greater scrutiny of the affected pieces of changed code.

5. RESULTS

Patch Analysis has been in use for about 12 months. To measure its effectiveness we have examined a sample of 49 fixes which contained defects (regressions). [6] We then went back to the reports we created for the original fix in an attempt to learn whether the reports were successful in focusing test efforts appropriately.

The results, which are shown in Figure 9, suggest that the employed methods can be useful in helping testers focus their efforts. Another analysis that would try to quantify impact of the Patch Analysis tools on the number of externally-found regressions specifically is planned. We hope to report on it in future.

Patch Analysis toolset has evolved considerably over the last 12 months and has seen its adoption rates improve substantially over time. We have tried to ensure its accuracy and effectiveness without sacrificing simplicity and usability. In the process, we have come to realize the following underlying principles that are of practical importance in creating data mining tools like ours:

1. **Metrics should be simple to understand, empirical, insightful.** Users need to understand the connection between a given metric and the outcome (preventing regressions in our case). Metrics should provide information that would otherwise be hidden.
2. **Metrics are project- and context- specific.** The choice of metrics is determined by the project at hand. Even metrics that can be applied universally will have project-specific thresholds above which risk is substantially larger. Statistical analysis of data helps determine these thresholds.
3. **Metrics should be non-redundant.** Few, carefully chosen data points are easier to use than a lot of numbers. Each data point should add a substantial amount of new information.
4. **Information should be actionable**. Metrics should be interpreted and users need to understand how to act based on the data presented. Some a priori assumptions are necessary (i.e. "if you see complexity >= 50 be warned") but

[6] Here we are not discriminating between how many of those defects were found internally vs. externally as we are trying to measure the overall effectiveness of the approach.

some of this knowledge can only be accumulated over time while our tools and data points are in use.

6. FUTURE WORK

Our future work is going to focus on the following areas:

1. **Accuracy and comprehensiveness of metric collection.** We are going to address the most important gaps in discovering dependencies between modules and functions. For example, we will attempt to take into account shared data stores like data files, configuration entries, and objects.
2. **Project specific risk analysis.** As mentioned in section 4.5, we are currently evaluating methods for risk prediction. We hope to make substantial progress in the next 12 months and report results then.
3. **Ensuring validity in a changing development system.** Development process is a constantly evolving social system. If our risk prediction work is completed, we should be able to make better accept/reject decisions on hotfix requests. As a result, we might see adjustments in acceptance criteria which in turn might affect what predictors we use for calculating risk and how strongly they correlate with the actual regressions found later in the field. We expect we will need to recalculate our models every few months to keep up with these changes.

7. CONCLUSIONS

Windows due to its size, complexity, diverse set of users and roles it plays in the PC ecosystem poses a unique maintenance challenge. Expectations for hotfix quality and response time are very high.

Patch Analysis is being developed to expose testers to previously hidden information with the purpose of helping them make decisions on the scope of testing required to minimize risks of further problems in changed code. Since its deployment, the tool has been able to help testers find defects by either identifying individual tests or areas of testing likely to uncover problems. Its adoption has increased significantly in recent months. Experience we have gained working on the first iteration will be applied to the next version of the tool in which we plan to make substantial further progress in our quest to have an accurate and reliable risk and test impact evaluation system.

8. ACKNOWLEDGEMENTS

I would like to thank Anand Paranjpe, a Test Manager at Microsoft, for providing support to this initiative, my colleagues: Ranajoy Sanyal, Rajiv Das and others who have contributed their time and effort to the Patch Analysis toolset, and Jari Jattu, Sonny Chinthamani and Brandy Koetters who reviewed this paper.

Special acknowledgement to all Microsoft engineers who, over the years, created the infrastructure tools used in this project.

9. BIBLIOGRAPHY

1. IEEE Standard for Software Maintenance. 1998. IEEE Std 1219-1998.

2. **Vliet, Hans Van.** *Software Engineering: Principles and Practices.* West Sussex, England : John Wiley & Sons, 2000.

3. Windows Vista Editions. [Online] Microsoft Corp., 2007. [Cited: June 2, 2007.] http://www.microsoft.com/windows/products/windowsvista/editions/.

4. Product Support Lifecycle. [Online] Microsoft Corp., July 30, 2004. [Cited: June 2, 2007.] http://support.microsoft.com/select/?target=lifecycle.

5. *Effectively Prioritizing Tests in Development Environment.* **Amitabh Srivastava, Jay Thiagarajan.** 2002. ISSTA.

6. **Amitabh Srivastava, Jay Thiagarajan, Craig Schertz.** *Efficient Integration Testing using Dependency Analysis.* s.l. : Microsoft Research, July 2005. MSR-TR-2005-94.

7. *Using Historical In-Process and Product Metrics for Early Estimation of Software Failures.* **Nachiappan Nagappan, Thomas Ball, Brendan Murphy.** Raleigh, NC : Proceedings of the International Symposium on Software Reliability Engineering, 2006.

8. *Mining Metrics to Predict Component Failures.* **Nachiappan Nagappan, Thomas Ball, Andreas Zeller.** Shanghai, China : s.n., 2006.

9. **McCabe, Tom.** A Complexity Measure. *IEEE Transactions on Software Engineering.* 1976, Vol. 2, 4.