{Coders}
Kitchen

Baseline Testing

The key to reducing technical debt in legacy code bases

Table of Contents

1 Introduction

Software was once seen as "the holy grail" in the sense that it should be virtually maintenance-free; a product that could be written once and used many times. Unlike mechanical components, which after a certain point need to be refreshed and replaced, an organization shouldn't have to worry about software wearing out or breaking down. After all, it's software right? Perhaps in theory.

But problems began to appear, ultimately caused by the continual development of software without the correct quality control processes in place, typically due to the tremendous business pressure to release new products on a specific date. This has resulted in legacy software applications carrying an enormous amount of technical debt, a metaphor for latent defects introduced during system architecture, design or development.

The key to reducing technical debt is to refactor components (the process of restructuring application components without changing its external behavior/API) over time, but developers are often hesitant to do so for fear of breaking existing functionality. One of the biggest impediments to refactoring is the lack of tests that formalize existing behavior, or an easy to run testing environment that demonstrates the correct behavior of the component. Without refactoring, an application's code becomes overly complicated and difficult to maintain. As new features and bug fixes are bolted onto existing functionality, the original design often loses its simplicity.

As a result, many organizations are finding that legacy software typically has a waning lifespan. Companies are forced to decide whether to throw an application away and start again from scratch, or try to salvage it. In most cases, a substantial financial investment has been made in the software, and there is tremendous pressure to reuse it.

2 What to do next

Many large companies are now discovering the technical debt that has accumulated in the software that is currently deployed. This is a growing problem as these deployed/fielded applications are based on legacy code that doesn't have the proper regression test suite to ensure application correctness. According to a Gartner study, "a lack of repeatable test cases limits an organization's ability to demonstrate functional equivalence in an objective, measurable way[1].

The reason companies are facing the technical debt issue in the first place is that often they want to deploy legacy software in a new platform/product, and the test suite to validate the software doesn't exist. There is a massive quality gap that needs to be addressed along with seemingly insurmountable test work ahead – and companies usually don't know where to start or don't have the resources needed to address the problem.

3 Compliance with the ISO 26262 standard

If a developer needs to make updates to certain components of a legacy code base, they have to be very careful not to break any of the existing capabilities. However, if an existing capability is broken, the developer needs to understand if it was because the original software was written incorrectly or due to a requirement that wasn't adequately captured in the original implementation.

Often the lack of sufficient tests means that a software application cannot be easily modified since, unfortunately, changes frequently break existing functionality. So, what can be done to allow legacy code bases with inadequate test cases to be used with confidence? One approach would be to unit test the entire application. This sounds good in theory but is often impractical due to the amount of time and resources required. But what if you could create these "Baseline Tests" automatically? Baseline testing, also known as characterization testing, can be used to capture the current behavior of software.

For an application that has been "in service" for a number of years it is not unreasonable to assume that the application has some level of stability. Recognizing this level of stability, one might look at the source code as a basis for defining test scenarios. The software is analyzed to determine what the test vectors should be (input values). Those input values are used to drive the code down different paths with the intent of building all of the tests necessary for a high level of code coverage. By defining those test scenarios and executing the code, the expected values are captured by looking at the actual values

[1] Gartner Group, Monitor Key Milestones When Migrating Legacy Applications, May 18, 2015

that were produced. The test case is not derived from the requirements in baseline testing; it is derived from the actual source code, which has been deemed to be functionally correct.

By using automatic test case generation to quickly provide this baseline set of tests that capture existing behavior, testing completeness of legacy applications is improved, and refactoring can be done with confidence that application behavior has not regressed.

4 Developer's dilemma

I am going to modify an application's source code to add new functionality to the next revision of a software product. When I modify that source code, I may change the behavior of it because I look at something and determine that it's not right - it should be behaving this way or that, or the requirements have changed - so I delete a block of code and add new capabilities.

In that situation, the existing software might be exercising the path that is being updated, and that path may be reliant on errant behavior to do what it is supposed to do. Perhaps by modifying the source code, when an existing path is executed in some of the unmodified software that runs throughout this area, the software may take another path other than what it was supposed to. In this case, I inadvertently discover that I just broke something that was working before!

If we don't have the test cases to verify all of these paths, someone can make a one-line fix and put it in the integration build and things just don't work anymore. Then everyone is trying to debug it, and it's a mess. A week later, we discover that it's actually a bug that was written five years ago that was just uncovered now, accidentally, as a result of changing this one line.

5 Approach for paying off technical debt

Baseline testing can be accomplished by using technology that looks at the application's source code and analyzes it to build test cases with the goal of having 100% code coverage. During the execution of tests, the results are captured and intelligently stored as expected results for use during future regression testing or during refactoring. An important point to make is, not only are the test cases created automatically, but the test environments needed to execute these low-level tests are also created automatically. These two key test elements can be deployed in a highly automated manner to baseline the behavior of software without any prior knowledge of the environment – and with minimal human intervention involved in the process.

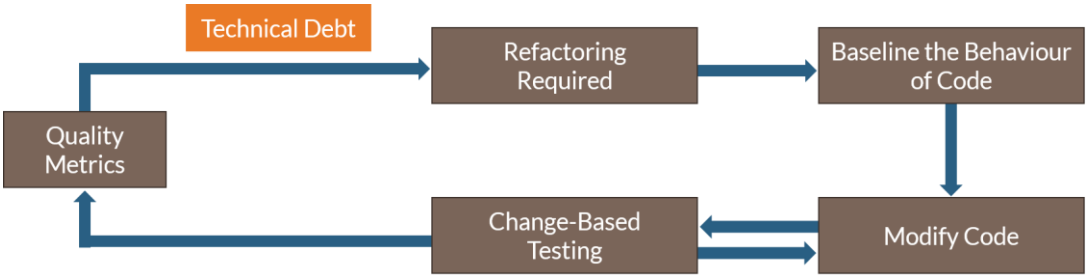

Figure 1: An approach to paying off technical debt using baseline testing of legacy code

Once the behavior of the software has been characterized, or baselined, the developer now has a complete regression test suite available that can be run as needed on a daily basis to ensure that updates and modifications to the code doesn't break existing functionality.

To further automate the continuous integration and testing process, impact analysis in the form of change-based testing can be utilized to run the subset of test cases needed to show what effect code changes have on the integrity of the whole system. It is not unusual for a project to have test scenarios that run for hours or days. With change-based testing, a developer can make a one-line change and get feedback on its impact to the entire application within a few minutes.

As a result, developers are able to make quick incremental changes on the code while knowing that they have the test cases needed to capture the existing behavior of the software. They are also able to do further analysis if something is broken to work out if a bug has been introduced, a capability has been removed that actually should be there, or if there is a bug that should be addressed because it may have other ramifications.

6 Analytics: A view of risk

As a key part of the baselining process, sophisticated analytics tools enable organizations to drill down to analyze particular areas in the software that need to be addressed. This provides a "view of risk" (risk assessment) that includes:

6.1 Complexity risk

An analysis of where the highest amount of complexity exists in the software in terms of the number of decisions. Naturally, the extremely complex areas equal the highest amount of risk. For example, if code coverage is being used to determine how well an application has been tested, analytics tools can determine that one of those functions hasn't been tested at all (0% code coverage) another one has only been tested up to 25% code coverage, and another one has been tested up to 50-75%.

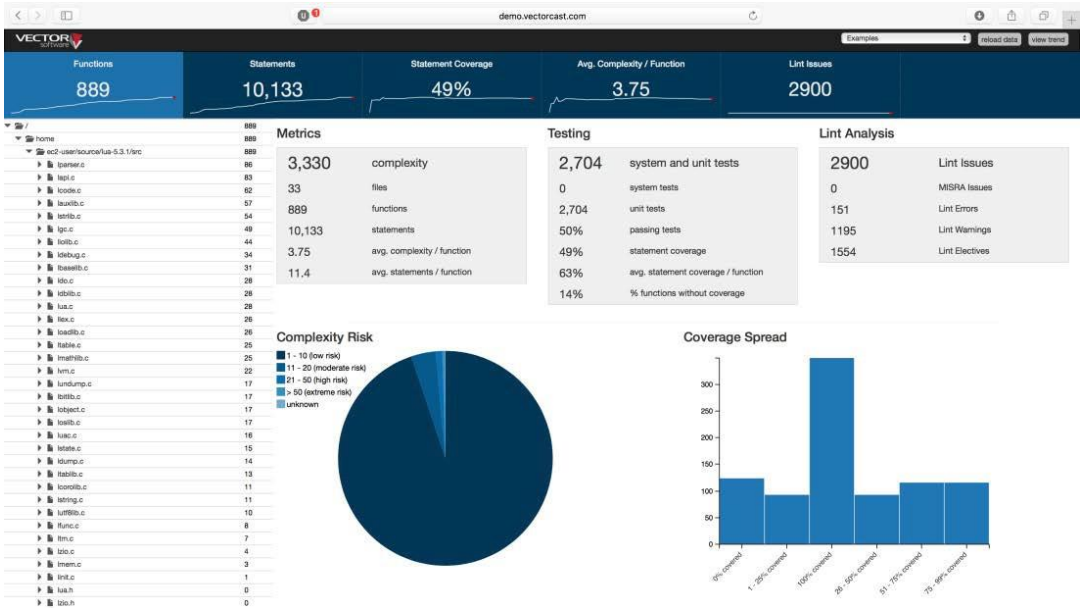

Figure 2: Analysis of complexity risk

If a manager who is trying to improve the quality of software is reviewing this data, they would know the areas to focus more attention on by being aware of the functions that have a limited amount of testing. Analytics tools enable a further breakdown to see what the specific function is.

Code coverage analysis of functions

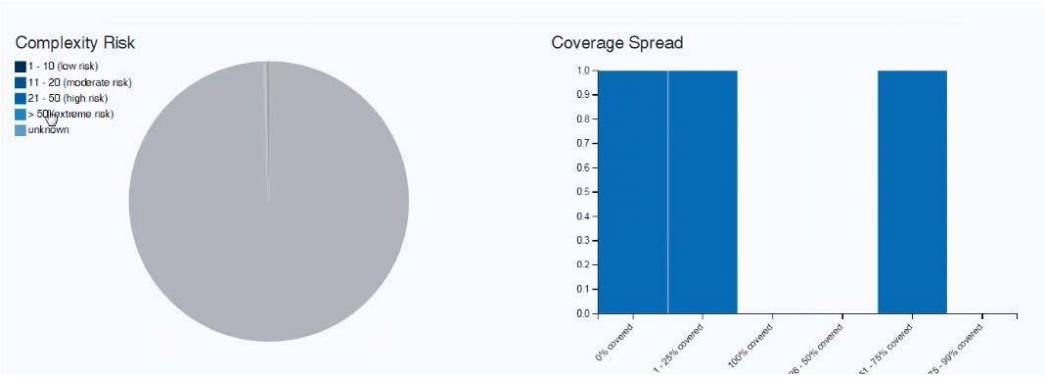

Figure 3: Breakdown of code coverage analysis by function complexity

Figure 4: Figure 4: Breakdown of code coverage analysis by function complexity

6.2 Trend information on improving software quality

A primary goal of a baseline testing effort is to create a path toward incremental improvement while increasing the maintenance life of the code base. Analytics tools can provide a view of coverage over time that shows quality is improving so that managers, developers and others can see the progress that is being made in the software.

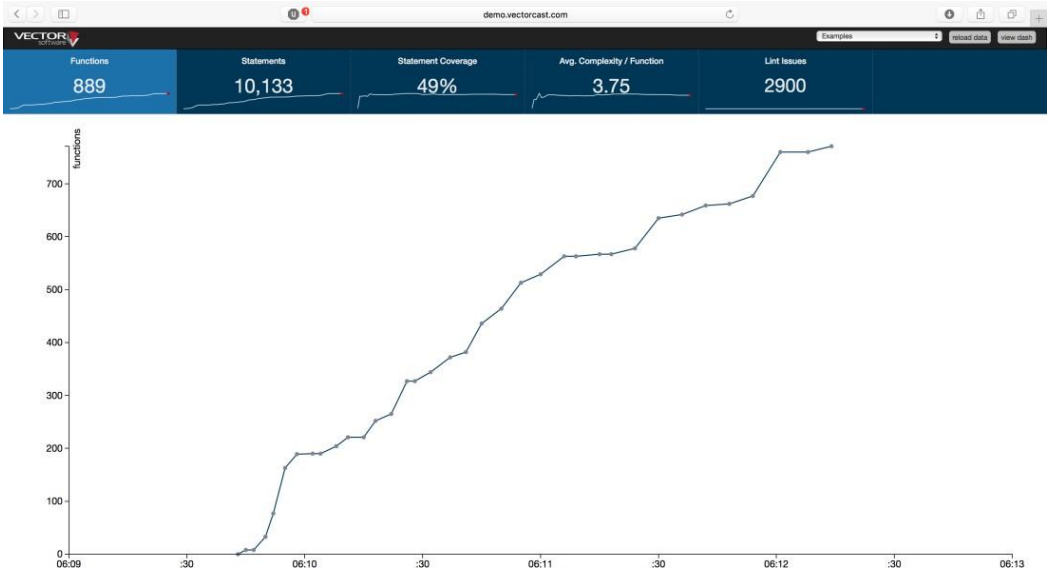

Figure 5: Analysis of statement coverage over time to show quality improvement

It is also well-known that many software development projects finish significantly behind schedule. A key contributing factor to this is poor data. This capability also enables a user to build useful schedule and trend information to help create more realistic schedules, resulting in people taking fewer shortcuts and creating less risk in the software produced. If a compromise does need to be made, the information is available to help prioritize decisions.

Analytics tools also enable users to quickly look at a project and get a bird's eye view of where the effort should be going by providing a real-time and interactive picture of what is happening in a software application. This can be very valuable when trying to make decisions on whether legacy software should be updated for the next project, or thrown away and started again.

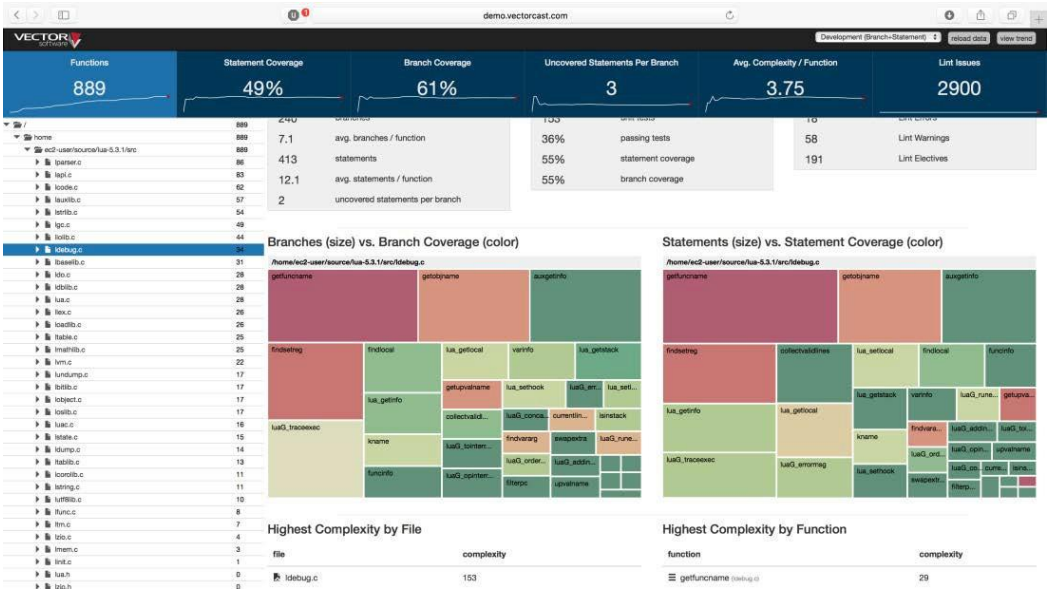

Figure 6: Bird's eye, interactive view of what is going on in a software application

7 Branch case study: When consumer-grade systems become safety-critical

There has been an interesting shift that has resulted in a new generation of software that previously did not have safety-critical requirements but now does. Therefore, existing software usually has to be updated to accommodate new capabilities and there often isn't evidence to demonstrate how the software worked correctly in the past.

For example, in the era of self-driving cars, telematics is becoming one of the most critical in-vehicle systems. Autonomous driving capabilities have shifted this software-driven system from a consumer-grade communication application to safety-critical as certain autonomous driving capabilities are reliant on telematics information.

This telematics information is enabling potentially life-saving applications. For example, if a vehicle five cars ahead of you brakes, instead of having to wait for a visual cue from the vehicle directly in front, the vehicle five cars ahead will send a broadcast to yours over the telematics interface that it is braking. Your vehicle also needs to start braking now. In this case, faulty software has severe ramifications, so quality is no longer an option -- it is a necessity.

An example of a recent baselining exercise comes from a Vector Software automotive customer. This customer was shipping a software system to a major auto manufacturer and wanted to do a system update. The customer engaged with Vector Software, and its Global Services team went on site, analyzed the situation and determined that baseline testing services would meet the customer's requirements.

Within a six week timeframe, Vector Software's Global Services team was able to baseline test and build a complete regression suite for future testing. Without Vector Software's automatic test case generation technology and testing capabilities, this process easily could have taken six to nine months. The estimated cost savings of this was roughly $900,000 – and that is just in terms of the effort – not even taking into account quality improvements and all of the very real cost impacts associated with that.

What is more compelling is that as part of the baselining activity, Vector Software found latent bugs in the code base. These weren't critical issues in the software's current environment, but if the product were deployed in another environment, for example, those errors would have likely manifested themselves. The errors were immediately addressed and fixed in the code base. Now when developers do a new revision to the software, they can more easily make changes without worrying about new issues.

8 Conclusion

Many organizations continue to discover that a tremendous amount of technical debt has accumulated in the legacy code bases that a bulk of their currently-deployed software is built upon, which has the potential to cause substantial quality issues. This becomes an even bigger problem if they want to deploy that software in a new platform/product and they need to validate it, as these legacy code bases lack the repeatable test cases required to demonstrate functional equivalence in a measurable way.

This often prevents developers from refactoring for fear of breaking existing functionality. However, baseline testing can now be done to characterize the current behavior of existing software to address legacy code bases that have inadequate test cases. Developers will have a regression test suite available that can be run on a daily basis to ensure that updates and modifications to the code don't break the existing functionality, enabling them to successfully develop new applications or improve existing technology.

9 Legacy code baseline testing and health check services

Available through Vector , baseline testing is useful for legacy code bases that have inadequate test cases. Often the lack of sufficient tests means that the application cannot be easily modified since changes often break existing functionality. Having test cases that formalize the behavior of an existing piece of software enables developers to refactor and enhance applications with confidence.

During a Health Check, a Vector expert will conduct an unbiased view of the current state of an organization's testing infrastructure and gather key metrics such as gaps in testing, code complexity, and comment density to identify risk areas. A detailed report will be provided containing all of the gathered data along with recommendations for improvement.

For more information on VectorCAST and Vector visit **www.vector.com/vectorcast**.

Imprint

Vector Informatik GmbH
Ingersheimer Str. 24
70499 Stuttgart
Germany

Phone: +49 711-80670-0
E-mail: info@vector.com

www.coderskitchen.com