McCabe SOFTWARE

Case Study:

Developers and Testers in Head-On Collision?
You May Need a “Mechanic”

Introduction

As a contractor for a major financial player in Germany, SOBEGE, a German-based consultancy specializing in embedded IT and web services, was faced with the challenge of analyzing the quality of various legacy software artifacts.

SOBEGE's client, utilizing in-house and outsourced development, used old legacy systems based on **COBOL** on mainframes, and source code which was created in the 1970s and updated since then. The client began to use MS Windows on the employee's machines in 1990, and as such, required a user friendly GUI written in **C**, **C++** and/or **Visual Basic**. They also required an interface for transferring data back and forth from the mainframes. As time went on, they adopted web technologies and newer databases which led to the adoption of **Java**.

All of the systems are presently in use - so the biggest challenge was to find a way to compare the code quality of the legacy systems against that of the Java systems.

Furthermore, the Developers wanted to know whether all of their code was used, and when or how often a line of code was tested. In addition, the Testers wanted to know whether their functional tests thoroughly covered the source code, or whether critical areas were missed. How did they avoid the inevitable Developers/Testers head-on collision? They needed a "mechanic".

Key Challenges

- **Multiple Source Code Languages** - Comparing legacy systems (COBOL and VB prior to .NET), which follow a procedural programming paradigm, to Java, which is object oriented.
- **Static Quality Gates For Various Languages** - Determine how and what to measure. Are the measured values comparable? Repeatable? Reproducible?
- **Code Coverage Supporting Both Programming and Testing Teams** - Quantifying code coverage to developers. Explaining coverage data to testers who do not have a programming background.

The Challenges and Solutions

Challenge and Solution 1: Multiple Source Code Languages

After considerable research, there was only one solution capable of analyzing source code quality and providing code coverage on such a disparate array of software development languages in play at the client site. Only McCabe IQ could provide support for COBOL, VB6, and Java (along with a dozen other languages not utilized by the client).

Additionally, McCabe metrics are calculated in a strictly mathematical and language independent way and therefore can be used to compare code of different languages. (Of course, some other empirical metrics such as Halstead metrics are also calculated by the tool and could be used as a "side dish".)

Using "the McCabe metrics", one can compare source code of different programming languages because the metrics are calculated the same way – a prerequisite for any real metric.

Challenge and Solution 2: Static Quality Gates for Various Languages

Another challenge was to find appropriate values for the metrics to be used as static quality gates for code acceptance. That is, what "standard" threshold values should be used, especially given that there are several very different programming languages?

Because the client already had many COBOL applications, SOBEGE began to collect 3 items: the source code, a dozen metrics (e.g. v(G), ev(G), iv(G), SLOC, CLOC, ...), and - most important - a "rating" for the application. Applications or source code with a very long lifecycle (in this case 30 years) are maintained over and over, which enables developers to rate their code as "very stable, running years without errors", "last error in coding found 3 years ago", "coding is easy to understand", "very well documented and commented", "horrible coding, cannot understand", and so on.

So SOBEGE took the metric values, calculated the mean values for the lower bounds where "good code" becomes "bad code" (based on the developer ratings), and the quality gates were born.

A similar thing occurred when addressing VB6, where there were various projects with developers' comments scaling from "worst I have ever seen" to "running like a charm and easy to maintain".

The next language, Java, proved to be a bit more challenging. There were not many long-term Java projects and the developers were unable to rate the coding. So "reference implementations" were taken, i.e. sample Java JDK 1.2 projects from Sun ("good coding") together with a few discontinued Open Source Java projects which had been reported as "bad coding". Moreover, the set of metrics had to be changed to reflect the OO paradigm of Java. McCabe IQ supports this, with their OO metrics (per definitions by Chidamber and Kamerer), therefore SOBEGE collected several McCabe metrics, several line count metrics, and the entire set of OO metrics. The client now has 3 sets of metrics with their corresponding quality gate values. One for COBOL, one for VB, and one for Java. Because quality assurance is an ongoing process, SOBEGE continues to update the metrics database with current projects. Over time, the Java metrics mean values have changed slightly.

Every project member, no matter what source code language they use, can compare the metric values of his code with the "top 3rd average" of "good code".

When contractors deliver code, the code should pass through the quality gates. If code does not pass and is still accepted, there must be a short explanation why the code was accepted, e.g. "Code generated by Rational Rose".

Challenge and Solution 3: Code Coverage Supporting Both Programming and Testing Teams

The third challenge was the implementation of dynamic analysis – also known as test or code coverage in a manner supporting both programming and testing teams.

Most large projects at the client's site are divided into a development team and a test team. Of course the developers run their "unit tests", but most products they are coding are GUI intensive applications with a very special focus on fiscal, financial, statistical order data calculations. The test teams are "specialist teams" and most of the team members hold a BSc, MSc, or MBA in Economics or equivalent - but have little or no experience or interest in programming issues.

So, the developers do not have the time to manually test the GUI with every release, and the test team does not have the background to use McCabe IQ, a source-code based tool, to create instrumented source code. However, what they do have in common is that both groups desire to know the code coverage of the functional GUI tests.

The solution was to install at least one person as an interface between the two teams - called the "mechanic". Such a person needed to be able to "read" source code (though not necessarily required to be an excellent developer). That way the source code to be tested is now transferred to the "mechanic" who then instruments the source and compiles it to an instrumented binary. The binary is delivered to the test team and all the GUI tests are run with the instrumented version. The generated trace files are collected after all test runs and passed to the "mechanic" who imports the trace files into the McCabe project (or projects), and creates reports/graphs of various types for analysis.

For a simple overview ("How effective or how efficient are my GUI test runs?"), the "publish metric" feature of McCabe IQ is a perfect way to give the test team some values without bothering with source code. The publish feature produces an HTML dashboard that summarizes results and gives meaningful feedback, even for individuals without a strong software development background.

The developers can use the coverage values to verify whether there are any untested source files or functions (methods) which should have been tested but aren't.

On the other hand, the "mechanic" can focus his analysis on important but (partially) untested functions. For example, with GUIs, depending on the programming language, you may have many functions/methods which are called when someone clicks on a GUI item, or presses a key etc. The first step would be to ensure that all GUI elements are covered, i.e. report any untested "buttonclick", "mouseclick", "keypress" methods to the test team. The "mechanic" simply conveys the analysis into language that the tester understands (that's why the "mechanic" needs to understand the source code programming language), e.g. tell a tester to "hit the Close-button on the form xyz-123".

Note that it is impractical to attain to branch coverage of 100% with huge GUI applications and significant resource constraints. Branch coverage values of 65% and more are considered very good - higher values are rare.

If the test team has used managed tests and/or capture & replay tools, reports that "all functional requirements are tested", and there are still hundreds of completely untested functions/methods, then the developers may have coded too much. They may have developed things that were not needed, things that were intended requirements but are no longer essential, or functionality which was used in a prior release but isn't used anymore, etc. So, this may indicate that it is time for developers to do some refactoring of their source code.

The approach of using a "mechanic" should work for any software project with more than one developer. One disadvantage is that you need a person (possibly an additional employee) to focus on this responsibility. The benefit of identifying a single person is that you could share the mechanic over multiple projects. Additionally, you would not bother the developers with time consuming GUI tests, or testers with programming language issues that do not concern them.

Conclusion

SOBEGE has analyzed several million lines of COBOL, VB and Java code and understands that it is extremely difficult to install a well-defined and structured test process in most programming teams. Most, if not all, teams are pressed for time and the first thing which gets cut, in most cases, is testing at the end of a milestone cycle.

However, once the McCabe solutions are accepted, developers are able to create more understandable code and recognize that obtaining metrics on their code can lead to a higher quality application. Experience has shown that using these metrics, users are able to rate any new source code in seconds in terms of comprehensibility, maintainability or reliability.

As with most tests it is not easy to give management-enabled ROI values because "testing" means to put some destructive effort into your work, to search for incorrect behavior in the software. One can count the previously found failures, but no one can specify with certainty how many high-risk errors may still reside in the application. One can only estimate, and the McCabe solutions are a good way to verify whether the product is on the right track when it gets delivered.

McCabe solutions have been used successfully for more than 8 years at the client site, with various programming languages and project sizes. McCabe's longevity, having been around since 1977, speaks to the sound technology and track record of excellence in supporting thousands of software projects.

About SOBEGE

SOBEGE is focused on project and quality management with expertise in software metrics. The company provides services for the IT industry, specializing in embedded systems and new technologies, like Web Services with Java or the .NET platform.

Furthermore SOBEGE improves client communication by teaching technical employees to use a streamlined or more common and understandable language between technicians (developers) and management.

About McCabe Software, Inc.

McCabe Software has provided Software Quality Management and Configuration Management solutions worldwide for over 30 years. "McCabe IQ" analyzes the quality and test coverage of critical applications, utilizing a comprehensive set of software metrics including the McCabe-authored Cyclomatic Complexity metric. "McCabe CM" is the only Software Change and Configuration Management solution to utilize "integrated difference" technology. McCabe Software has offices in the United States and distribution worldwide, and can be found on the web at www.mccabe.com.