Introducing Static Analysis in a Mature Codebase

Author
John Ruberto
john@ruberto.com

Abstract

This paper tells the story of how our organization introduced static analysis into its software development process. The static analysis process automatically finds bugs, assigns them to the engineer that most likely introduced the bug, and then automatically verifies the fix.

This project began two years ago, when we started recording the root cause of bugs that were fixed. Once we had several hundred bugs analyzed, we found that the largest number of bugs was caused by coding errors. The good news, there are many best practices in industry to improve code quality. One survey of software quality practices (Jones 2011) shows that three practices, taken together, are capable of removing 97% of the defects. Those three practices are code review, static analysis, and unit testing.

Our organization already had healthy processes for code review and unit testing, but we had spotty history with static analysis. So, we took a fresh look at static analysis. When engineering the process for static analysis, we took care to avoid some of the issues of the past; namely false positives, excessive effort required to manage the bug list (triage false positives, manage assignments, etc.), and the demotivation of analyzing a large backlog of defects.

When we designed the new process, we decided to focus on automating the bug identification, assignment, and retest processes – eliminating a lot of the effort that got in the way of previous attempts. We also focused on keeping new code clean, which is much easier to justify than clearing old bugs from the backlog. The process also was designed to provide fast feedback, directly to the developer that created the bug.

Paying attention to new bugs has resulted in high levels of adoption by the developers, as compared to focus on backlog bugs. The engineers are fixing the new bugs that come in, plus have made a significant dent in the backlog.

Biography

John Ruberto has been developing software in a variety of roles for over 25 years. Currently, he is the Quality Leader for QuickBooks Online, a web application that helps small business owners manage their finances.

John has a B.S. in Computer and Electrical Engineering from Purdue University, an M.S. in Computer Science from Washington University, and an MBA from San Jose State University.

1 Introduction

My personal experience with using static analysis for established projects has been very spotty. These projects, which have a considerable amount of pre-existing code, are successful in their respective markets. Running static analysis tools on these projects usually produces an overwhelming volume of potential defects. Since the product is successful, the static analysis results are doubted.

In one example where I introduced the used of a static analysis tool, I was the test leader for a 5 million line C++ system with a client/server architecture. The project had been in place for seven years, and was successful in the market. I attempted to introduce static analysis into our software development process, with the goal of reducing defects found in the system test phase.

The static analysis tool that we chose found over 400K errors in the first pass. This seemed like a great tool and would really help us, until we tried to get developers to look at the result. The vast majority of the bugs did not affect functionality, but were related to odd code structure. These bugs could have caused future maintenance issues, but were not clearly bugs that needed to be addressed today. A majority of the errors that could affect functionality were false positives. A false positive bug is where the tool indicates a bug is present, but actually is the intended implementation and is not a bug after all.

Also, most of the errors found were in the legacy code, which was running in production and was successful in the market. These factors made it difficult to engage engineers to invest time to examine the bugs, much less fix them.

So, when introducing static analysis in my current company and project, I was wary about introducing this process, only to have the results ignored. This paper describes the journey that our team took to successfully introduce static analysis into an ongoing successful project.

2 Why Static Analysis?

We track the root cause for bugs found in system test, and through Pareto analysis (Bose 2010) noted that “coding errors” were the most frequent type of errors. Figure 1 shows a pie chart depicting the various causes of bugs found in system test. Coding errors were the top contributors.

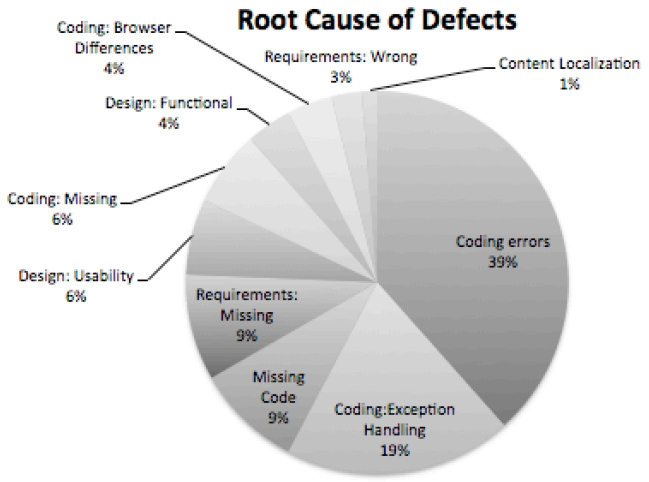

Figure 1: Pie chart of the causes of bugs in our application. This chart shows that coding errors are the most frequent cause of bugs.

The good news is that there are many best practices in industry to improve code quality. One survey of software quality practices (Jones 2011) shows that three practices, taken together, are capable of removing 97% of the defects. Those three practices are code review, static analysis, and unit testing.

Our organization already had healthy processes for code review and unit testing, but we had spotty history with static analysis. So, we took a fresh look at static analysis. When engineering the process for static analysis, we took care to avoid some of the issues of the past namely false positives, excessive effort required to manage the bug list (triage false positives, manage assignments, etc.), and the demotivation of naming a large backlog of defects.

2.1 What is Static Analysis?

According to (Black, 2011), static analysis is "*Analysis of software artifacts, e.g., requirements or code, carried out without execution of these software development artifacts. Static analysis is usually carried out by means of a supporting tool.*"

Figure 2 shows an example of a defect found through static analysis. The tool used for this example was Coverity® Static Analysis for Java. In this example, the tool was able to detect that an object was being dereferenced before it was being checked to see if it had a value equal to null. If the object was null coming in, this code would have generated a null pointer exception, and the effort to create a check for the null condition would not be effective.

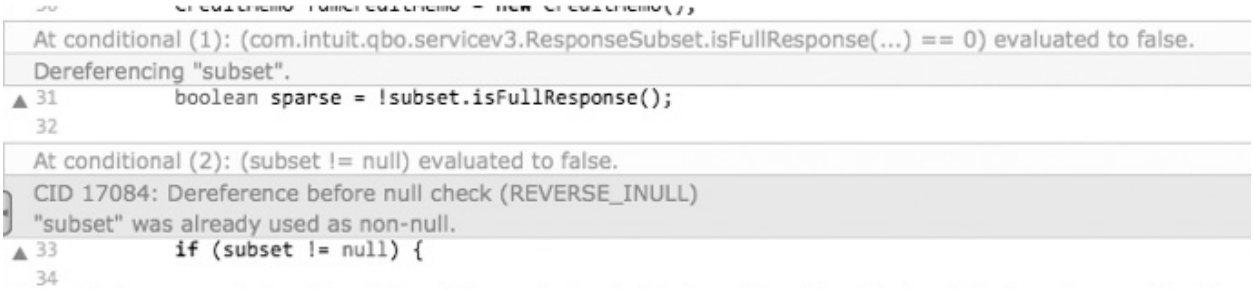

Figure 2: An example bug found by static analysis. In this bug, the object "subset" is dereferenced in line 31, and checked to see if its value is null in line 33. If the object was indeed null, this code would have thrown a "null pointer exception" before the null check happened.

This is a good example of the power of static analysis; the tool can find common coding errors like this automatically. Finding this type of error through test may be difficult if it is difficult to create the conditions where the 'subset' is null.

3 Process Design

The team formed to design and implement this process was comprised of quality engineers and developers. Before jumping into tool selection, the team designed the process. The process design illuminated requirements for the tool selection.

Using Design For Delight (Ruberto 2011), the team identified important factors for the static analysis process. The Design For Delight method involves identifying the key pain points for customers, and designing a process to solve those pain points.

In this context, developers in our department were the “customers”. The developers were very interested in delivering bug free code, and less concerned with fixing old code that was already in production. So, we designed the static code analysis process around identifying newly introduced bugs.

Plus, the new bugs were shown to be quicker and easier to fix, since the developer that introduced the bug was still assigned to the project, and the code was still fresh in their minds.

3.1 Initial Static Analysis Process

We initially designed the process with the following factors in mind:

- Find new defects as close to creation/injection as possible
- Alert the developer about the defects, and allow developers to fix defects before checking in code, or before code review.
- Track defect status and provide status reports
- Efficient management of the defect management process. Automatically assign defects to developers, and automatically close defects once they are fixed.

Figure 3 depicts the process that was designed. The static analysis happens before check-in, and ideally all defects are fixed before the code is committed to source control.

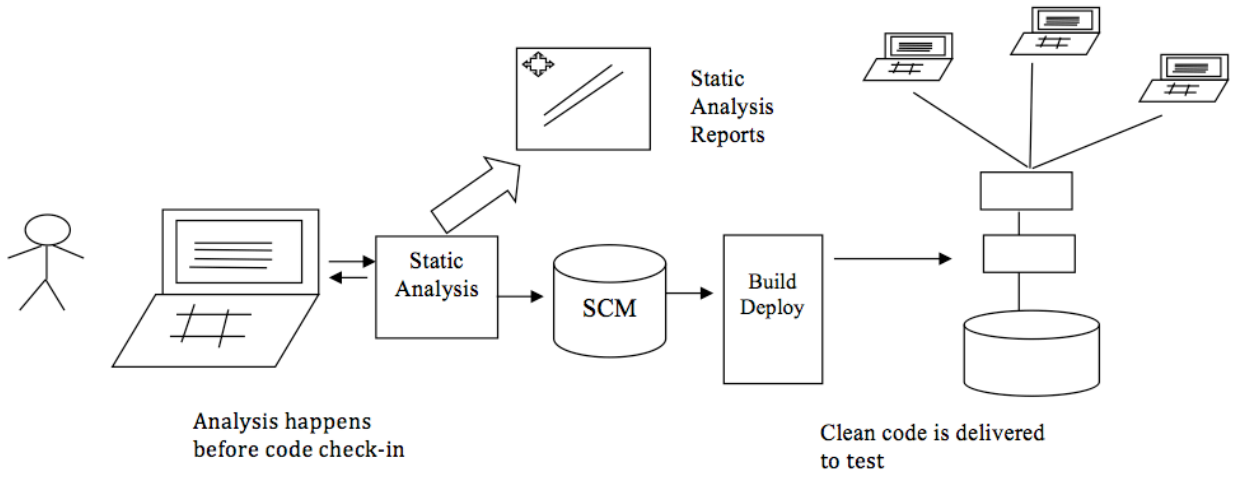

Figure 3: Incorporating Static Analysis in our build process

3.2 Process Design, Iteration 2

After surveying and evaluating the tools available, we did not find one that enabled the developer sandbox scans. The best tools actually scan the entire code base, which is only feasible in context of a full build. So, we iterated on the process, which is shown in figure 4.

This process performed the static analysis after check-in and in parallel to the build/deploy process. This process has the disadvantage of:

- Defects could affect the test, as they are detected in parallel with build/deploy
- Notification and follow-up with developers was more important, to make sure they follow up and fix the defects before moving onto other tasks.

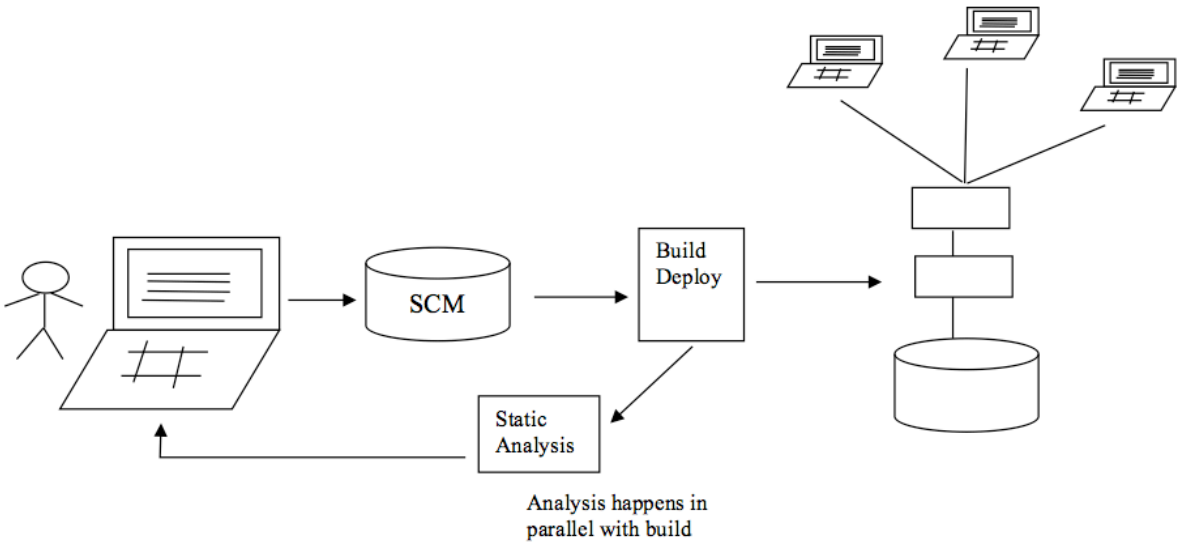

Figure 4: Modified process after learning about tool capabilities.

4 Implementation

4.1 Tool Selection

Designing the process first helped identify the requirements for tool selection. The key requirements for our search were:

- Effectiveness with our development language (Java)
- False positive rate with our code base
- Compatibility with our development environment (IDE, build system)
- Ability to uniquely identify defects
- Either a built-in defect management system, or the ability to integrate with our system through an API

Our favorite web search engine helped identify a list of potential tools. We chose a selection of industry leading tools and open source tools to actually try. It's important to try before you buy, to avoid "shelf-ware". We found a few tools that were just not compatible with our code base, even with a lot of effort and help from the vendor support team. There were also tools that worked, but produced results that we felt were not worth the effort.

Once we had scans from several tools, it was time to engage with developers. We took random samples of defects from each tool and asked developers to analyze the results. We graded several factors:

- Efficacy of the checkers (how powerful the tool is for finding interesting defects)
- False positive rate (tool indicates a defect, where there is no defect)
- False negative rate (tool doesn't find a defect, which indeed exists)

After the evaluation, we found two tools that worked well with our code base, and these tools worked well together. They had very complementary checkers (a defect missed by one tool was found by the other, and vice-versa). Table 1 shows the results of the evaluation of these two tools.

Resource Leaks	Findbugs	Coverity
OS com.intuit.qbo.bl.payrollform.EditPayrollForm.getBytesFromFile(File) may fail to close stream [line 209]	✔	✔
OS com.intuit.qbo.controller.util.AuthHeaderHelper.getResponseBodyAsString(HttpMethod) may fail to close stream [line 413]	FP	✔
com.intuit.qbo.monitor.LoadBalance.runCommand(String[]) may fail to close stream [line 214]	FP	✔
com.intuit.qbo.service.provisioning.partnerconfig.PartnerConfig.initialize() may fail to close stream [line 354]	✔	✔
OS com.intuit.qbo.ui.common.UICommonFunctions.includeUIFiles(String) may fail to close stream [line 1604]	✔	✔
OS com.intuit.qbo.ui.help.index.SiteIndex.run() may fail to close stream [line 252]	✔	FN
com.intuit.salsa.ui.login.CachedAuthCompanyInfo.loadFromFile(String) returns from routine while resource still open [line 69]	FN	✔
com.intuit.qbo.xml.xmls.XMLVendor.main(String[] args) returns from routine while resource still open [line 1104]	FN	✔
com.intuit.qbo.util.ResourceAccessGateKeeperDisableTest.touchResourceFile() returns from routine while resource still open [line 416]	FN	✔
com.intuit.qbo.util.ResourceAccessGateKeeperDisableTest.touchResourceFile() returns from routine while resource still open [line 605]	✔	✔

Possible Null Pointer Dereferences	Findbugs	Coverity
Possible null pointer dereference of defaultServiceItem in com.intuit.qbo.bl.comgmt.CreateCompany.addDefaultTimeItemToPrefs()[line 580]	FP	✔
Possible null pointer dereference of zipCode in com.intuit.qbo.bl.comgmt.QBOCompanyManager.postCreationSetup(QBOCompanyManager$CreateCompanyDataIf, boolean) [line 1953]	✔	FN
Null pointer dereference of aDB in com.intuit.qbo.bl.recur.EditRecurInfo.main(String[]) [line 917]	✔	✔
Possible null pointer dereference of response in com.intuit.qbo.bl.txns.TxnSmsNotification.invokeRequest(HttpClient, String, String) [line 244]	✔	✔
Possible null pointer dereference of checkDate in com.intuit.qbo.ui.transactions.payroll.payemployees.PayEmployeesConstants.JAVASCRIPT_SEND_PAYCHECKS(Integer, Integer, QBDate, QBDate, QBDate)	✔	FN
com.intuit.qbo.service.util.mapping.SalesReceiptMapper. mapSalesReceiptCdmType2QboType(SalesReceipt) possible null pointer dereference of detail [line 124]	FN	✔
com.intuit.qbo.ui.interviews.mini.MiniInterviewHandler.handleRequest() possible null pointer dereference of first_mip [line 135]	FN	✔
com.intuit.qbo.ui.mobile.IPhoneHandler.setLoginHelpMessage() possible null pointer dereference of locale [line 3616]	FN	✔
com.intuit.qbobilling.util.RetailRenewalUtil.send(int, boolean) dereference before null check [line 270]	FN	✔
com.intuit.qbo.ui.transactions.purch.creditcard.core.CreditCardhandler.handleRequest() dereference before null check [line 144]	FN	✔

Table 1: Matrix of defect types and score from two static analysis tools. FP = False Positive, FN = False Negative, check(✔)=valid defect (Batas 2011)

The following table lists the top fifteen most common errors detected by static analysis tools we applied to our codebase. With each error is a reference to detailed information about that error.

Defect Type	Description
NULL_RETURNS	http://cwe.mitre.org/data/definitions/476.html
FORWARD_NULL	http://cwe.mitre.org/data/definitions/476.html
RESOURCE_LEAK	http://cwe.mitre.org/data/definitions/404.html
REVERSE_INULL	http://cwe.mitre.org/data/definitions/476.html
SE_BAD_FIELD	http://findbugs.sourceforge.net/bugDescriptions.html#SE_BAD_FIELD
NP_BOOLEAN_RETURN_NULL	http://findbugs.sourceforge.net/bugDescriptions.html#NP_BOOLEAN_RETURN_NULL
SE_COMPARATOR_SHOULD_BE_SERIALIZABLE	http://findbugs.sourceforge.net/bugDescriptions.html#SE_COMPARATOR_SHOULD_BE_SERIALIZABLE
EC_NULL_ARG	http://findbugs.sourceforge.net/bugDescriptions.html#EC_NULL_ARG
LI_LAZY_INIT_UPDATE_STATIC	http://findbugs.sourceforge.net/bugDescriptions.html#LI_LAZY_INIT_UPDATE_STATIC
MF_CLASS_MASKS_FIELD	http://findbugs.sourceforge.net/bugDescriptions.html#MF_CLASS_MASKS_FIELD
GUARDED_BY_VIOLATION	http://cwe.mitre.org/data/definitions/366.html
EC_UNRELATED_TYPES	http://findbugs.sourceforge.net/bugDescriptions.html#EC_UNRELATED_TYPES
XSS_REQUEST_PARAMETER_TO_SERVLET_WRITER	http://findbugs.sourceforge.net/bugDescriptions.html#XSS_REQUEST_PARAMETER_TO_SERVLET_WRITER
GC_UNRELATED_TYPES	http://findbugs.sourceforge.net/bugDescriptions.html#GC_UNRELATED_TYPES

Table 2: The top fifteen most common errors identified by the static analysis tools we applied to our codebase

4.1.1 False Positive Rate

A high rate of false positives will destroy any hope for adoption of static analysis, and for good cause. If the tool is giving more false results than true, the value is diminished. Our evaluation paid special attention to the false positive rate.

We took several random samples of defects identified and had developers evaluate each one to determine false positive rate. Any tool with lower than 50% rate was rejected. The consensus among the

development community was that a 20% false positive rate was tolerable, especially if the tool identified many serious bugs that may have otherwise gone undetected.

4.2 Automation

Most of the static analysis packages allow integration with the build system to automatically. We took advantage of this feature as well. We also wanted to automate the defect management process.

One sore point for previous attempts at using static analysis is the triage and bug assignment process. Finding someone to review the results, determine if the defects are real, find the right developer to fix the bug, and verify that the bug is actually fixed is a labor intensive task. Performing this role is usually thankless, and often doesn't get done.

We automated the bug management:

1. When a new defect is found (not already in the list), we have a script that does a lookup for the last person to check in the file that contains the bug.
2. The script then assigns the bug to that developer in the defect management system.
3. The developer is notified, via email, that they have a new bug assigned. The email contains a description of the bug, and a link into the defect management system.
4. If a defect no longer appears on the list, it is marked as fixed automatically.

With this process, the bug assignment and closure when fixed is completely automated. Being an automated process, there have been some glitches. Most notably, the script that does the bug assignment determines the engineer by the last person to check in a file. Sometimes, two or more engineers touch a file in a day, and this has introduced some errors in the assignment. These cases happen in a small percentage of the bugs found, and the developers simply reassign the bug to the right person. Overall, the cost saved by totally automating the bug management is much higher than the cost generated by wrong assignments.

4.3 Launch and Learn

The value for these tools is to actually fix the bugs. This is where the wider developer and QA community comes into play. It is important that the process is in place and for the tool to perform its job, but more important for the users to actually use it, and get value from, this process and tool. Launching the process entails more than just educating people on what to do and how to do it, but also why we are adding static analysis.

We launched this process with a presentation to the developers. In the presentation, we covered the following topics:

- Why we should introduce static analysis (Root cause analysis data from section 2, etc.)
- Results of the tool selection process
- Examples of the errors found by static analysis
- Testaments from early adopters (developers that helped with the selection process)
- Description of the process and what is expected of each developer
- Training on how to use the tools
- Feedback process for users to identify any issues with the tool or process

We then activated the process by enabling the build time scans and email notification.

4.4 Reports

Communication and follow up is important to make sure the new process sticks. We send out several reports regularly to keep the team informed about the status static analysis, and the value it provides.

Bugs are found on most days. When a bug is found, the developer assigned to fix that bug is automatically notified with the list of bugs assigned to him or her. These reports go out daily, and are private emails directly to the developer.

The quality manager sends out a weekly status report that shows the open bugs, which developers are assigned, and the severity level of the open bugs. These reports focus on the current project. Table 3 shows an example of the weekly report.

Owner	Major	Moderate	Minor	*Total*
Amit	1			*1*
Andy		1	1	*2*
David		2	1	*3*
Frank	2		1	*3*
Mary			2	*2*
Praveen	1	1		*2*
Sandeep	1	2	1	*4*
Sunil			1	*1*
Victor	1	1		*2*
Total	***6***	***7***	***7***	***20***

Table 3: Typical Weekly status report showing open bugs by developer for the current project.

The quality manager also sends out a monthly report that shows:

- Overall backlog status trend
- Status on the previous release (How many bugs were found and how many fixed)
- Highlights the top developers for fixing backlog bugs

5 Results

Figure 5 shows the defect fix rate overlaid on the incoming rate (starting from when we deployed the tool). This graph shows that the fixes are keeping up with the arrival rate, and that we've made a dent in the backlog.

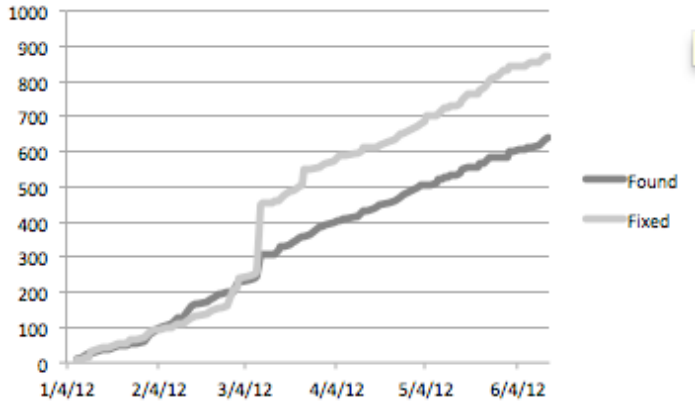

Figure 5: The defect arrival and fix rate since static analysis became active.

In the last full release, the team fixed 97% of the issues found, with 100% of the major severity defects fixed.

Since this process has been deployed, we've also reduced the number of errors and exceptions that happen in production by 20%.

Qualitatively speaking, the developers appreciate how this process has been designed. They see static analysis as a tool to help them code better.

6 Conclusions

Incorporating static analysis for this project has been a success. I believe this was successful for the following reasons:

- The process was designed to handle newly introduced bugs, rather than tackle a mountain of backlog bugs. The new bugs are the most valuable to fix since they have a high uncertainty factor and the code is still fresh.
    - New run-time errors occurring in production have declined by 20%
    - Project teams fix 97% of the identified defects prior to release
    - The bug fix rate exceeded the bug arrival rate. Over a 6-month period, approximately 600 new bugs were identified and fixed, along with 250 legacy bugs.
- Care went into the tool selection to find the best set of tools for our particular code base
    - Two tools were selected because they identified a complimentary set of defects. The chosen tools were: FindBugs, an open source utility; and the commercial Coverity® Static Analysis for Java.
    - These tools have demonstrated a false positive rate of less than 20%.
    - The low false positive rate coupled with the lightweight defect management process has improved the adoption by developers.
    - Influential developers participated in the tool selection process, which also helped to drive adoption.
- Defect management (assigning bugs, verifying closure) was completely automated, which speeds up the feedback cycle (defect injection until defect assignment), and lowers the operating cost of this process.
- Frequent reports showed the value of static analysis, which reinforced the new process

References

Batas, Brent, 2011, *Final Report on Internship Experience*

Black, Rex; Mitchell, Jamie; *Advanced Software Testing – Volume 3*, Rocky Nook, page 264

Bose, Tapan K., 2010, *Total Quality of Management*, Pearson Education India, page 343

Jones, Capers, 2011, *Software Quality in 2011, A survey of the State of the Art*.

Ruberto, John, 2011. *Design for Delight Applied to Software Process Improvement, Pacific Northwest Software Quality Conference Proceedings, 2011*, page 163