Understanding the Value of Program Analysis Tools

Ciera Christopher Jaspan
Carnegie Mellon University
cchristo@cs.cmu.edu

I-Chin Chen
eBay
ichen@ebay.com

Anoop Sharma
eBay
ansharma@ebay.com

Abstract

It is difficult to determine the cost effectiveness of program analysis tools because we cannot evaluate them in the same environment where we will be using the tool. Tool evaluations are usually run on mature, stable code after it has passed developer testing. However, program analysis tools are usually run on unstable code, and some tools are meant to run right after compilation. Naturally, the results of the evaluation are not comparable to the true contribution of the tool. This leaves program analysis tool evaluations being very subjective and usually dependent on the evaluators intuition. While we could not solve this problem, we suggest techniques to make the evaluations more objective. We started by making *enforcement-based customizations* of the tool to be evaluated. When we evaluate a tool, we used a *comparative evaluation* technique to make the ROI analysis more objective. We also show how to use *coverage models* to select several tools when they each find different kinds of issues. Finally, we suggest that the tool vendors include features that assist us with a *continuous evaluation* of the tool as it runs in our software process.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics—product metrics

General Terms Economics, Experimentation, Measurement

Keywords program analysis, economics, value-based software

1. Introduction

Recently, eBay started an initiative to use program analysis tools to improve the quality of their products. This was not an entirely new endeavor for eBay; several internal teams had independently tried program analysis tools without successful results. In these prior evaluations, it was not clear that the tools contained value. It is not enough to show that a tool finds important defects; it must find enough defects to offset the costs of purchase, maintenance, enforcement, and handling of false positives. In other words, the tools must have a good return on investment. The previous informal explorations had not shown that the tools contained a good ROI, but these teams also did not put as much effort into the investigation. Most of the previous evaluation teams ran an off the shelf tool, viewed the results, and then made a subjective assessment of the tool. However, this methodology was not enough to justify a large purchase decision. eBay wanted to know whether the tool would provide a reasonable ROI for their products.

It is inherently difficult to make a return-on-investment analysis for program analysis tools since we can not evaluate them in the same environment they are used. Many of these tools are meant to be run on developer's desktops or nightly builds, and they are integrated into the development process. They provide value by catching defects early in the process on recent code changes. Ideally, we would investigate the ROI of a program analysis tool running under this environment. For most companies, this is not an option; we would have to halt production and inconvenience developers. Since we cannot evaluate the tool under our ideal environment, we have to run our evaluation under conditions that are not like those in development.

In previous tool evaluations at eBay, the evaluation team ran a tool on a snapshot of code mature code and then reported on the tool's output. As this output was not from a development environment, it resulted in an inconclusive analysis of the tool. Management, developers, and quality engineers were rightfully suspicious that these tools would not be have value for eBay. They were only given sample data from an environment that was arguably very different from the true environment. Decisions on whether to use a particular tool were based entirely on the intuition of the evaluator.

Our goal was to take a more thorough and objective look at these tools and to overcome these inherent difficulties as much as possible. We first want to evaluate each tool individually to determine its cost effectiveness. As program analysis tools find different kinds of defects, we would also like to choose a suite of tools for use, so we have to evaluate

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-786-5/07/0010.

the tools as a set. Finally, we would like to continuously evaluate on our selected program analysis tools while they run in our software process.

It is not the goal of this paper to report on how tools performed, as we feel these results will be different for each company and each project. Many existing papers cover evaluations of the tools we discuss and give examples of the kinds of defects they discover; these evaluations also show the value of the tool by itself [11, 12, 9] Instead, our goal is to show how we analyzed the value of these tools in our environment, given that we planned to use multiple tools together. We present the methodology we used in a general way which is repeatable by other companies.

In this paper, we have several contributions for how to evaluate these tools:

- *enforcement-based customization* Many of the program analysis tools can be customized for a particular domain or project. To aid with the customization process, we created an enforcement policy for each issue priority level. This prevented developers from getting too ambitious in what issues should be high priority, a problem which occurred during previous evaluations.
- *comparative evaluation* We can not avoid the inherent problems with evaluating these tools in the wrong environment. However, we can use a comparative evaluation in order to make the numbers easier to manage and discuss. When a tool find 200 defects on our false environment, it is difficult to understand how this works in a production environment. Instead of extrapolating these numbers to a production environment, we ask "Do we believe this tool will provide a compelling value proposition compared to an existing technique?"
- *coverage models* Each tool has different strengths for finding defects, and no tool finds all the defects which a company might be interested in. We would like to select several tools that will compliment each other and provide the best value as a group. To understand these interactions and select a set of tools, we created a coverage model of the defects we were most interested in and fit the tools into our model.
- *continuous evaluation* There is a need for us to be able to evaluate the tools even after we have inserted them into our process. However, it is difficult to retrieve all the data we need to be able to do this kind of evaluation. In particular, we need to know the number of true positives being found and fixed by developers, and this is difficult without tool support. In order to allow us to run continuous evaluations, we suggest features that vendors could add to their program analysis tools.

Throughout this paper, we will display numbers for various costs and numbers of defects found. These numbers have all been changed, but the order of magnitude remains. Most of the examples come from the open-source static analysis tool FindBugs, [5] as it was the only tool to receive a complete evaluation and also begin integration into eBay's process. Several other tools were also evaluated, but they have not yet been added to the software process.

In Section 2, we discuss eBay's goals for using program analysis tools in their process, including their need to create custom analyses and gather metrics from the tools. Section 3 discusses how we evaluate a particular analysis tool for a particular project. In particular, we discuss customization and using comparative evaluation to evaluate a tool. In Section 4, we take the results of our individual evaluations to choose a suite of tools that will best fulfill a project's needs. Before concluding, we also discuss the need to run a continuous evaluation in Section 5 and suggest features that tool vendors could add to help meet this need.

2. eBay's goals

EBay is interested in using program analysis tools for two purposes: finding defects earlier in the lifecycle and finding defects that are difficult to detect through other quality assurance techniques. By finding defects automatically, they hope to save time in the later QA process so that quality engineers can focus their time on other issues. They also would like to catch defects that are hard to detect using traditional QA techniques, such as security, performance, and concurrency defects.

EBay plans to run these tools as early in the development process as feasible. The company would like to have developers running faster tools on their desktops, and they would like slower tools to run on the team build system. They plan to enforce the issues these tools find by running the tools on the build system. It must be easy to override the tool in case of emergencies, however, any defects which are not fixed need to be logged and reviewed later.

EBay does not have the resources to create their own program analysis tools, so they are looking for off-the-shelf products. The company can dedicate a few people towards making these tools better and customizing them for eBay. This team of people would be in charge of maintaining the infrastructure for the tool and for creating new analyses that look for project-specific defects. Since eBay is interested in creating their own customized analyses, they would like a tool which is extensible and customizable.

EBay would also like to continuously evaluate how these tools are performing. This fits well with the company culture; the software process is continually monitored to assure that quality software is being produced smoothly. This culture was derived out of necessity; eBay has a fast-paced software lifecycle and distributed engineering teams. They keep a close eye on their software process to ensure that they make their bi-weekly releases, and they keep a close eye on their quality assurance techniques to ensure that all their employees around the world are maintaining the same level of quality. Any tool that is used company wide needs to allow continual monitoring by the central team responsible for maintaining the tool.

The Whitebox QA team has been heading the program analysis initiative for eBay Marketplaces division. We are responsible for evaluation, adoption, evangelism, customization, and maintenance for automated quality assurance tools. In addition to program analysis tools, the Whitebox team is responsible for regression and coverage tools.

3. Evaluating an analysis tool

There are two problems that we found with the previous evaluations done at eBay. The first problem was that the evaluators (and tool vendors) did not recognize the need for customizing the tool to the product and domain. The second problem was that it is inherently difficult to determine value because the evaluation environment is not the same as the production environment. In this section, we provide a methodologies for customization and determining a tool's value within the evaluation environment.

3.1 Customizing

Customization is probably the most important step of determining a tools value. Many of the previous evaluations done at eBay did not go through this step. The evaluators either did not know it was possible to customize these tools or they did not realize the importance of customization. Part of the reason for this is because the tool vendors typically do not advertise that the tools should be customized; the tools are generally marketed as being general purpose, off the shelf products.

For a company that is willing to spend the hours to customize it though, the tool can perform significantly better. On the project website, FindBugs advertises a 50% false positive rate, which is very good for a scalable static analysis tool. At eBay, customizing FindBugs dropped 75% of the original issues found. The developers at eBay considered these dropped issues to be mostly false positives, so it drastically lowed the false positive rate for FindBugs. On the customized version of FindBugs, we estimated only 10% false positives.

Most of the analysis tools are made up of several subanalyses, or *checkers*.[1] Each checker finds a different type of code issue, for example, possible null pointer exceptions, possible concurrency errors, or suspicious use of "==". Customizing the tool involves switching these checkers on and off as desired. In some tools, the issues produced can also be assigned a priority based on the checker, so the priority can be customized as well.

To customize the tools, we started by turning on all of the checkers and running it on a sample of 100-200 KLOC. We organized the results by checker and sorted them in order from the most found to least found. We also pulled a sampling of issues from each checker as examples. We provided these results to the development team whose code we ran the tool on, and we asked them to assign each checker a priority.

In order to prevent all issues from becoming "high priority" and to give some meaning to the priority levels, each priority has an *enforcement policy* behind it. The developers had to consider this enforcement policy when determining how important a type of issue was. Upfront discussion of the enforcement policy was extremely important to prevent developers and managers from becoming too ambitious in their classifications. Past evaluation teams had difficulties because managers would insist that all of the issues found were important for developers to review, regardless of how severe the issues might actually be. As a result, all the checkers would be turned on, and developers would refuse to use the tool due to high false positive counts. By providing a costly enforcement mechanism, management and developers were forced to recognize later consequences of their actions. We provided the development teams with the following priorities and enforcement policies:

- High priority: All issues of this type will be purged from the entire codebase. As eBay's codebase counts in the millions of lines of code, this becomes extremely costly. An issue must be important enough that developers are willing to wade through legacy code to fix it. A null pointer exception is an example of a high priority issue for eBay.
- Medium priority: Issues of this type are not allowed to be added to the codebase. Old issues will not be purged unless a developer is refactoring the code for other reasons. For eBay, medium priority issues were usually costly to fix throughout the codebase, but cheap to fix in new code. These issues typically signalled possible defects, but they were not defects themselves. High McCabe complexity is an example of a medium priority issue at eBay.
- Low priority: Enforcement for low priority issues is by way of a cap number. The QA team will allow low priorities in the codebase, but they are capped at how many new issues can be added in each release. These issues were usually stylistic and had little significance, but some developers felt that they were important enough to view before checking in their code.
- Tossed: Some checkers were considered to have extremely low ROI for eBay. These checkers were turned off entirely. In some cases, a checker would only be turned off for a particular package or class, if the tool allowed this fine of tuning.

After we assigned priorities to the checkers, we customized the tool and ran it on another 100-200 KLOC code-

[1] The tools each refer to these sub-analyses by different names, such as rules, detectors, and checkers. We will use the term checkers for the remainder of the paper.

base. We then organized the results, met with the developers of the new codebase, and repeated the process. The goal of these meetings was not to come to a complete consensus on priority levels. There will always be quibbles about whether a particular class of issues should be high or medium priority. Instead, we asked developers “Can you live with these enforcement policies?” For eBay, this process converged very quickly; the third development team to evaluate the results of FindBugs had no additional changes.

We consider this step vitally important to evaluating a tool. Many of the tools were not impressive when they were first run, simply because the checkers (and their priorities) did not match up with eBay’s goals. For example, the 2 FindBugs checkers that produces over half of the original issues were unimportant in eBay’s environment. These checkers were turned off to get down the false positive rate. We also discovered that some of the lower priority checkers (like the “Dead Store to Local” checker) could discover performance errors that are considered high priority in eBay’s domain. We had a similar experience with Klokwork, where we discovered that the checkers that were on by default were not interesting in eBay’s domain, but some of the checkers that were off by default found more interesting issues.

We also discovered that customization significantly affected developer adoptability. A few developers had tried these tools once before and were unimpressed with the results. They did not know that they could be customized, so they put off all static analysis tools as useless. When we were able to provide developers with customized results, they were more interested in incorporating the tool into the software process.

3.2 Evaluation environments

Many of the program analysis tools are intended to be run on the developer’s desktop before committing, or possibly run on a team server every night. The tools are run continuously in our process so that they are not only finding defects in new code, but they are also checking that the older code maintains high quality through modifications. Therefore, a complete evaluation of a program analysis tool would involve installing the tool on many developer workstations and watching it perform over a length of time. This is a high-cost method for evaluating a tool, and it undermines our reason for doing the evaluation in the first place. We can not disrupt the development process until a tool is shown to be cost effective, but we can not show it is cost effective unless it is in the development environment.

For most companies, the best solution is to evaluate the tool in a false environment. The evaluator downloads the most recently checked-in code and runs the tool on this snapshot. The evaluator counts the number of defects found, possibly sorting by type of defect. These results are presented to management, developers, and quality engineers who make an intuitive judgement of how this data will extrapolate to our true environment.

Extrapolating this data is difficult since it may have no correlation to how the tool runs in a production environment. For example, a tool might find many defects on a snapshot, but the tool might find few on a day-to-day basis. The defects it found in the snapshot could have been from dead code or could be defects that simply accumulated over several years and were missed by other quality assurance techniques. On the other hand, a tool that performs poorly on a snapshot may be extremely cost effective in a production environment. The tool may be best tuned for finding defects that occur in the middle of development, before any other quality assurance techniques. According to [2], finding a defect during development is a cost savings of approximately 10 times over finding the defect during testing. Even if we can catch the defect with existing techniques, we would prefer to catch it early with an automated tool.

Given these problems, an evaluator has to make a subjective assessment as to whether the tool has good value for the company. The evaluator must use the data to determine some approximate number and type of defects the tool will find in the production environment. As this is based upon the evaluator’s intuition, it becomes subject to speculation and debate. Even after a tool is purchased, adoptability is difficult because there is little definitive data to show why the tool is worth using at all.

We could not find a way around the problems with the evaluation environment. However, we could take our subjective assessment and provide it with more weight so that we can agree on the overall value of the tool. In the previous approach, the evaluator must generate numbers that represents the cost and benefit for the tool over time. Instead, we provide real numbers for existing quality assurance techniques, and we ask if the tool will be better, worse, or equal to these numbers. It is much easier to say that the value will be above or below some particular threshold than to provide the value from intuition. [3] We also found that this methodology is also more likely to produce consensus among the various stakeholders.

3.3 Best effort cost analysis

Our goal for the evaluation is to determine whether a tool has enough value when run on a particular project. While it is difficult to produce these number from the evaluation, we can do a “best effort” cost analysis by comparing to an existing quality assurance technique. Presumably, since the company is already using the existing technique, they have determined that it holds some level of value for the company. Therefore, we only need to show that the tool has at least the value as the existing technique.

EBay hires hundreds of manual testers to explore the functionality of the site, and they have found this technique to be worth the high cost involved. Additionally, eBay gathers metrics on their manual testing process. We can use these metrics, along with the results from our customized tool evaluation, to produce a comparative cost analysis with the following steps:

1. Determine the yearly cost of the program analysis tool (if there is an initial cost, this must be amortized)
2. Determine the amount of the other technique's resources that have an equivalent yearly cost
3. Determine the weekly benefit for the other technique, given the resources found above
4. Evaluate the customized program analysis tool on a subset of code
5. Compare the results, and make an educated guess of whether the program analysis tool will provide at least the benefit of the other technique.

We compare the costs in terms of yearly costs because this is an easier number for stakeholders to reason about. For example, eBay knows how much a manual tester costs them every year, including salary and benefits, and eBay knows how much the yearly licenses for the tools are. We switch to using a weekly number for the benefit for the same reason. It's hard to talk about finding 500 bugs per year, but it's easier for people to understand 10 bugs per week.

When we finally get to the comparison, we still do have to take our environment problems into account. However, we now have a number to compare our evaluation against. If the numbers are significantly different, we might be able to say "this tool is at least as good as our current techniques", even if we don't know what the exact value is. Of course, there will be cases where the numbers are so close that we must rely on more subjective methods, but this technique would allow a team to make a definitive call for at least some of the tools they evaluate.

At eBay, we used this technique to evaluate the cost of the tool FindBugs. The numbers we present below are completely faked, but we have preserved the order of magnitude and the final decision about the tool.

3.3.1 Yearly cost of FindBugs

FindBugs is open source, so we have no initial cost for the tool. The initial setup cost is also not significant. Based upon our evaluation, we estimated that eBay would have to dedicate two full-time members of the Whitebox QA team to FindBugs for training developers, updating the tool, and adding infrastructure to enforce fixes and gather more metrics. For purposes of making our comparison easy, we will say each engineers will cost $200,000 per year, including salary and benefits.

3.3.2 Yearly cost of manual testing

Manual testing requires a lot of resources, including quality engineers and infrastructure of the test environment. For the purposes of this comparison, we will leave out the infrastructure costs, though we would include that normally. We want to choose a number of manual testers with an equivalent yearly cost to the cost of running FindBugs every year. We'll say that each manual tester costs $200,000 per year, so we will chose two manual testers as our subset.

3.3.3 Weekly benefit of manual testing

EBay gathers metrics on each of their testers on a weekly basis. We were able to average out these numbers and determine a typical number and priority of defects that a manual tester would find in a given week. We determined that a typical tester would find 10 bugs a week, and that the priority levels would be split as shown in the second column of Table 1. This means that two testers would find approximately 20 bugs per week, with the breakdown displayed in the third column of Table 1. Again, we have changed the numbers, but the order of magnitude remains.

3.3.4 Results of the tool evaluation

We took our customized version of the program analysis tool and evaluated it on a codebase of approximately 200 KLOC. We retrieved numbers that held the same order of magnitude as the fourth column in Table 1. Of course, we still have the problem that some of these defects are false positives, and we would not get these high of numbers on a maintenance level. We will take this into account in the informal comparison.

3.3.5 Informal comparison

At this point, we looked at a sample of the potential defects that FindBugs discovered to get a feel for the results. We now had to decide whether we would receive more benefit than the manual testing on a weekly basis. In the end, our decision was that FindBugs had a better cost-benefit ratio. We decided this for the following reasons:

1. While we do not expect to find 500 issues a week on 200 KLOC, we do expect to find 5-10 issues a week on that size of codebase.
2. EBay's actual codebase counted in millions of lines of code, and we expect FindBugs to scale much better to a larger codebase than manual testing.
3. FindBugs did find lower priority defects. However, we expect that it will find higher priority defects when it is run during development.
4. There will be some overlap in the bugs found by FindBugs and manual testing, and we already know that we have a 10 times cost savings if we find these defects in development.
5. While there is cost in investigating false positives in FindBugs, they are expected to be negligible in the customized version.
6. FindBugs is able to catch some defects that we could not catch at all through existing techniques, such as the

Table 1. Sample numbers of issues for a cost analysis

Priority	Percentage found by a manual tester	# issues found by 2 testers	# issues found by FindBugs
P1 (highest)	10%	2	25
P2	50%	10	25
P3	25%	15	300
P4 (lowest)	15%	3	150

performance issues found by the "Dead Store to Local" checker.

In this case, the entire team was convinced that FindBugs had an extremely compelling value proposition when compared against manual testing. [2] Arguably, it might have been easy to make the call for FindBugs without this analysis simply because the cost of FindBugs is so low. However, many of the analysis tools available have very high yearly licenses, so this methodology will make those cases more clear. There will be tools where the numbers are two close to make a definitive call, but this method will allow us to determine which tools are certainly worth the cost or certainly not.

4. Choosing a set of tools

Each program analysis tool finds different issues; for example, Fortify's SCA [7] focuses on security defects, while Agitar's TestOne [1] maintains internal invariants. We felt it would be beneficial for eBay to use several tools in order to get better defect coverage. To help us select a set of tools that will provide the best value, we used a coverage model and an incremental cost analysis.

Choosing a set of tools is not as simple as deciding which tools find the issues we are interested in. While we can categorize tools as finding different kinds of issues, they will perform differently in each category. For example, FindBugs and Fluid [6] both look for concurrency errors, but Fluid was made explicitly for difficult concurrency issues while FindBugs catches only a few simple issues. There is also a lot of overlap between what tools cover, so a tool may have good value by itself, but not when considered alongside a suite of tools. We also have to consider the tool interactions, that is, whether the tools can integrate their data and prevent overlaps. Some tools can work together as an entire suite, as is the case of Klocwork's toolset [8].

To help us with the selection process, we created the coverage model shown in Table 2. As prescribed by [4], we also gave an importance level to each kind of defect that represents eBay's quality concerns. For this model, we chose to use high level categories, though we could have drilled down to a lower granularity and made categories such as "Null Pointer Exceptions", "Dead Store to Local", and "Infinite Loops".

[2] This is not to say that manual testing should be stopped! We just have a better cost-benefit ratio for FindBugs.

When we evaluate a tool, we give it a rating on a 1-5 scale for how it performed on our product with respect to a category of defects.[3] By doing this, we can anticipate what combinations of tools will be best to cover eBay's needs. We used this to determine the ordering of evaluation and adoption of tools. For example, FindBugs appeared in every tool grouping that we considered, so we went ahead and started adoption with that tool.

When it is time to evaluate a new tool, we need to determine the added value it brings, not the value it has on its own. In our FindBugs cost evaluation, we compared FindBugs to eBay's manual testers. However, we don't always compare to manual testing; we need to compare to the sum of our existing QA techniques.[4] This *incremental cost analysis* ensures that we don't count defects that we can already find with an existing QA technique at the same time in the software process. When we do this, we eliminate duplicate reporting of defects and show the value the tool has in combination with our existing techniques. At eBay, once we had decided on FindBugs as our first tool, we eliminated the same errors from other tool evaluations. In order to show value, the tools had to find new defects after FindBugs had already been used on the code.

To evaluate a set of tools together, we can also use the coverage model to evaluate them as a suite. For every tool grouping we are considering, we can rate the tools as a set on our coverage model. The scores are not additive since we may have duplicate reporting of defects. For example, if Tool A receives a score of 2 for performance and Tool B receives a 3, the combined score might be 3, 4, or 5, depending on how much overlap there was in the defects found. This allows us to create a separate coverage model and value for each subset of tools, so we can handle additional benefits such as a lower cost for combined purchase or ease of integration.

To do this, we start by evaluating all of the tools we are interested in. We then select a tool to incorporate into our software process. Once the tool is incorporated into the process, we evaluate the other tools again, but this time we compare them to all of the old techniques plus our new tool. In this way, we take defect overlap into account; each tool must show that it has additional value to add to our process.

[3] We do not display our ratings here because they would not give an accurate representation for how the tool would perform on another project.

[4] In eBay's current environment, we determined that their other QA techniques would not have a significant affect on the results.

Table 2. Program analysis coverage model for eBay

Category	Importance
Performance	High
Security	High
Global Quality	High
Local Quality	Medium
API/Framework Compliance	Medium
Invariants	Medium
Concurrency	Low
Style and Readability	Low

The coverage model allows us to choose the sets of tools we are interested in and the order we will adopt them in. Notice that we might not adopt the tool that has the best value first. It's possible that, due to overlap, a tool with the best value individually might not provide us with the best value given some other tools. To take this into consideration, we must use the coverage model that we created as a guide to the order that we will add tools.

5. Continuous evaluation

Once we have selected a set of tools and incorporated them into the development process, we still believe it is important to continuously monitor the value of these tools. There are several reasons we feel this is important, even after having purchased a tool:

- The tool may not be used by the entire company. It may be used as a longer-term evaluation by a select group of teams before going company-wide.
- As our codebase and software process changes, we may find that some checkers are no longer necessary (or, conversely, that a removed checker becomes more important).
- Many of the tools provide extension points for writing domain and project-specific checkers. As we create customized checkers, we need to evaluate their value in order to tweak them.
- Some tools have a "pay-per-checker" license, so we do not want to purchase any checker that does not meet some minimum value standard.
- The metrics we gather to do this continual evaluation may also help us to discover new ways to adjust the checker and increase the tool's value.
- A continuous evaluation provides reinforcement that the tool continues to be a worthwhile investment of time and money.

This may seem like overkill to evaluate a tool which has already been determined to have value. A program analysis tool requires a great deal of faith by management that it is providing value for the company. Since the defects are caught early in the development lifecycle, we have no record of the defects that the found. We don't know what kind of defects the tool is catching, or how many it caught each day. The management must rely on the developers' and quality engineers' intuition that a tool is worth some yearly cost. It would be worth the time investment of gathering the defect data just to help maintain faith in the tool.

We saw a particular example of how the lack of data can cause a team can loose faith in a tool. In past years, eBay had been using PMD [10] as a way to create their own customized checkers. These customized checkers looked for issues that are specific to eBay's codebase. The team also created a tool that would analyze the main branch of the source using their customized version of PMD. The tool gathers the data at specified time intervals and reports the current number of PMD issues in the code. As the code grew, so did the number of current issues reported. Presumably, many of these were false positives. However, there was no way to tell whether these custom checkers were providing any value. Without any obvious value, stakeholders lost faith that the tool was worth using, even with its very low cost.

It's quite possible that the customized checkers were preventing a new bug every day, but no one on the evaluation team would ever know. FindBugs is allowing eBay to create more complex checkers than PMD allowed, and the initial evaluation results for the customized checkers looked promising. However, without any continuing evaluation, we won't know whether these checkers will continue to be useful.

In order to run a continuous evaluation, we need to gather data about both true and false positives. With both of these numbers, we can show that the number of false positives are balanced by the number of true positives that never make it into the system. False positives are easily tracked by counting the number of defects that get suppressed, but true positives are not tracked in a way that is useful for evaluation. Many of the commercial tools do allow us to track true positives, but they track them on the main branch of the source, not at the developer's desktops. As we are encouraging (and eventually enforcing) developers to run these tools on their desktops, we are missing all of the true positives that occur locally. As the tools are used and enforced by our process, our true positive rate on the main branch will go down, but our false positive rate will continue to rise. This would cause checkers to appear to have less value than they actually do, and the evaluation team has no quantitative data to back up their reasoning for using a particular checker.

We would like to see analysis tools add features that will track true positives from a developers desktop and anonymously report them to a central server. By doing this, the evaluators have a complete picture of how the tool is working. Additionally, this data will allow the team to identify sub-patterns of true and false positives from within a checker. If the evaluator notices that the false positives are typically coming from a particular type of defect, they might be able to split the checker into two parts to lower the false positive rate.

6. Conclusion

Program analysis tools are increasingly available to software projects as a technique for reducing the number of defects early in the lifecycle and producing high quality software. There are a lot of choices for tools now, and a company that is interested in using these tools has to evaluate each one and determine whether it will provide value for their company. EBay has developed several evaluation techniques to determine which program analysis tools will provide the best ROI for our projects. We believe that these techniques are generally useful for other companies which are interested in using many program analysis tools, but do not have the resources to create in-house tools.

Evaluating the ROI of program analysis tools is an inherently difficult task because we are usually required to evaluate the tool in an environment that is very different from the environment where we will use the tool. While we cannot completely alleviate the problems with these evaluations, we can use a more rigorous methodology, *comparative cost analysis*, to understand what the results mean for our project. In a comparative cost analysis, we gather cost and benefit numbers for an existing technique that we know has a good ROI from our company. We then attempt to compare the tool with this existing technique. By using this methodology, we can be confident that some tools will hold value for the company, even though we can't produce the exact ROI.

As many of these tools are best suited to different categories of issues, a company may want to select multiple tools to use together. In order to understand all of our choices, we created a *coverage model* of the categories of issues our company is interested in. We can then place individual tools and sets of tools into this coverage model to determine which set of tool provides us with the best ROI for our company and covers the issues we are most concerned about.

Finally, we have a need to continuously evaluate these tools while they are being used by developers. EBay would like to create their own checkers to look for project-specific issues, and we want to verify that these checkers continue to be provide a good ROI. Additionally, we believe that having this continuous evaluation would ensure that stakeholders do not loose faith in the tool simply because the false positive counts are increasing. To do this, we must gather data on how many true and false positives the checker is finding on developer desktops. We want to know every time a developer runs the tool, finds a defect, and then immediately fixes it. The current industry tools do not have features that will support gathering information from developer desktops, so large amounts of interesting data are being lost. We ask that tool vendors add in features that will assist us with gathering the information that we need.

Acknowledgments

We would like to thank Jay Singonahalli and David Pride for supporting our efforts. Thanks also to the development teams that were our "crash test dummies" through this process.

References

[1] Agitar. Agitar TestOne. http://www.agitar.com.

[2] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[3] S. A. Butler, S. Jha, and M. Shaw. When good models meet bad data: Applying quantitative economic models to qualitative engineering judgments. In *EDSER-2: Workshop on Economics-Driven Software Engineering Research*, 2000.

[4] S. A. Butler and M. Shaw. Incorporating nontechnical attributes in multi-attribute analysis for security. In *EDSER-4: Workshop on Economics-Driven Software Engineering Research*, 2002.

[5] FindBugs. FindBugs Project. http://findbugs.sourceforge.net.

[6] Fluid. Fluid Project. http://www.fluid.cs.cmu.edu:8080/Fluid.

[7] Fortify Software. Fortify SCA. http://www.fortifysoftware.com.

[8] Klocwork. Klocwork K7. http://www.klocwork.com.

[9] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: an empirical evaluation. *SIGSOFT Softw. Eng. Notes*, 27(6):11–20, 2002.

[10] PMD. PMD Project. http://pmd.sourceforge.net.

[11] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *15th International Symposium on Software Reliability Engineering*, 2004.

[12] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006.