2009

PACIFIC NW

SOFTWARE QUALITY CONFERENCE

MOVING QUALITY FORWARD

October 27-28, 2009

Conference Paper Excerpt

From the

CONFERENCE PROCEEDINGS

Moving Software Quality Upstream: The Positive Impact of Lightweight Peer Code Review

Julian G. Ratcliffe
Senior Member of Technical Staff
Advanced Micro Devices
julian.ratcliffe@amd.com

Abstract

As schedules tighten and product launch deadlines remain fixed, software developers have to rise to the challenge without compromising quality. Managing software development and maintaining quality necessitates the development of processes and tools to facilitate efficiency. Software development teams, with a wide range of experience and expertise, are often spread across several continents. Coordinating an effort across such a workforce brings enormous challenges, not only because of the ever increasing complexity of silicon designs, but also the need to make excruciatingly efficient use of software development resources.

At Advanced Micro Devices (AMD) we have measurably improved software quality and built a culture of defect prevention by integrating a lightweight peer code review process into the development workflow. This paper describes the practical implementation and quantitative quality improvements. Additionally, the approach to process transition was effective in cultivating cultural acceptance, leading to developers adopting the changes almost voluntarily, something that surprised many of those involved!

Code review styles vary widely, ranging from "over the cube wall" to rigorous inspection, but many software development teams end up with a review process that is either chaotic or cumbersome. The addition of a lightweight peer code review process to coordinate and track the reviews introduces a powerful new weapon into the battle against bugs. Further, by closely integrating the code review, revision control and bug tracking, it is possible to design a process that can help and not hinder development.

Biography

Julian G. Ratcliffe is a Senior Member of Technical Staff at AMD. He has a background in parallel computing and holds a BSc. Hons. in Electronic Engineering from the University of Southampton, United Kingdom. His software experience spans more than two decades and in that time he has dealt with all processors great and small. He moved to Portland, Oregon in 1996 intending to spend a couple of years "getting the American thing out of his system", but forgot to leave.

Introduction

Imagine for a moment, a river of code bubbling up from its source within the consciousness of a development team on its journey to the ocean of customers. Bugs* in the code are present as pollution and the art of clean programming is the effort of maintaining the clarity of the water. Impurities are inevitably introduced and the process of removing the contaminants becomes more expensive and time consuming as the river flows increasingly voluminously towards the ocean. It is therefore imperative to maintain the quality of the river of code by filtering it through a code review process as far upstream as possible.

Mention code review to most software development teams and the first thing that springs to mind are multiple meetings, reams of printouts and a lot of time that could be better spent coding. While a very rigid Fagan code inspection process may have been appropriate in the mid-70s, a significant amount of time and effort is required to collate review material and coordinate its distribution and review. [1] Even less formal methods with less overhead, such as over-the-shoulder reviews or pair-programming still rely on authors and reviewers being able to meet, either in the same location or using teleconferences. But today development teams are not only distributed across multiple locations but also multiple time zones, so reviews can often only be performed asynchronously. A new better process is needed to efficiently conduct code review.

Examining the pros and cons of many traditional and modern code review techniques highlights a number of key factors that describe what the ideal process might look like. In order to be acceptable to both developers and reviewers a code review process needs to be easy to use and implement. It needs to make efficient use of their valuable time and not require explicit scheduling for either individuals or groups. Close integration with source code management and bug tracking systems is essential. Most importantly, from a quality perspective, the process should create a verifiable record of defect resolution and provide process enforcement. The process has to be effective at exposing defects and encouraging knowledge-transfer, naturally facilitating collaborative engagement across multiple sites and time-zones.

There are many factors to consider when modifying the workflow of a software development team, but the primary challenge is introducing change without disrupting existing commitments to schedules or deliverables. A productive team that is working to meet deadlines does not want to be distracted by process improvements that could be perceived as restrictive. Therefore the challenge is to introduce a lightweight process that does not impede current priorities.

The Lightweight Peer Code Review Tool

We chose the Code Collaborator code review tool from Smart Bear Software for our implementation.† The tool met our criteria for enabling the internationally distributed development team to perform code reviews in a repeatable and verifiable manner. Tasks such as gathering review material, notifying reviewers and coordinating review feedback are simple and intuitive.

---

* bug, n: An elusive creature living in a program that makes it incorrect. The activity of "debugging", or removing bugs from a program, ends when people get tired of doing it, not when the bugs are removed.

† A full description of the features or alternative tools is beyond the scope of this discussion.

The code review cycle starts with preparation for the review. Review material is attached with optional annotation and reviewers are assigned. Each review participant has a predefined role either as an author, reviewer or observer. Typically, the author allocates reviewers and observers from a list, but another mechanism allows an individual to use one of two subscription methods. One subscription method selects reviews by author for following changes made by a colleague. The other subscription method defines a file-grouping selection criteria used to watch changes in a specific part of the code tree.

Before the review begins the author has an opportunity to annotate the review material by adding notes describing the code changes. These notes are not a replacement for good use of comments within the code but are intended as a prologue for the reviewers examining the code. It's important to emphasize that this additional preparation represents a significant value to the reviewers in understanding the context of the review.

During the inspection and rework phases, the review is conducted using a web-based interface with an integrated chat-room style interface for comment and defect logging on a line by line basis. Conversations can occur in real-time or asynchronously. The customizable review templates define a set of roles, custom fields, and other options that determine the behavior and rules of a review.

The code review cycle is not complete until all of the reviewers are satisfied that the defects have been addressed and the review can close. The review is archived along with all of the associated activity and can be inspected or reopened at a later date. Metrics are gathered automatically as the reviews proceed and a configurable report generation capability provides a way to profile the data collected.

Integration with Issue Tracking and Version Control Tools

The introduction of a lightweight peer code review tool into the development process is only part of the solution. The integration of issue tracking and version control is essential in the successful execution of the overall code review process, in order to be tangibly effective in a large organization. Coordination with these other tools through their respective interfaces and hooks represents a significantly beneficial element of the implementation.

If you consider the three tools working together, the issue tracking tool defines the granularity of changes to the code base, the version control tool handles the change management and the code review tool introduces upstream quality inspection. Information from one tool can be used by the next, if at each stage of the process the continuity of the issue is maintained. By linking the tools we can verify the integrity of the entire workflow.

The issue tracking life-cycle begins when an issue is entered and assigned for assessment. This assessment may result in a task being generated and assigned to a developer, who in turn uses the version control tool to make code changes. The task forms the basic unit of work required to resolve the issue. The term "issue" is used not only to describe problems found in the code after the core development is complete but any work that is done on the code base at each stage of

development, debug, and maintenance. Using the issue tracking tool to manage the task granularity creates a verifiable record of the workflow that can be used to forecast future development effort. Each task remains active in the issue tracking tool until the code changes have been completed, reviewed and integrated into the code base, at which point the task can be closed.

Changes are commonly managed using task branching, which creates an explicit short term branch from the main trunk. This provides an isolated "sandbox" for the developer to work on any given task. The developer is then free to make as many iterative changes to the code as they require, testing the changes as they progress. The obvious and most convenient place to introduce the code review is just before the task branch is ready to be merged back into the main trunk.

By integrating the tools through their respective interfaces and creating a direct association with the task granularity, there are numerous opportunities to add automation and therefore reduce manual errors and tedious steps. A simple example is the use of hyperlinks, constructed using the task identifier, to quickly cross-reference information between each of the tools with a single click. Further examples might be the automatic attachment of review material from the version control system or the inclusion of the originator of a task from the issue tracker on a review.

An additional benefit we've been able to leverage from improved tool integration is a mechanism that more effectively generates release notes. In the past the release notes would be harvested from the issue tracking database and were often inconsistently completed. This was performed using a manual method immediately before the release was due and presented a significant challenge. Instead of entering the release notes into the issue tracking database a release note file is generated in the task branch and included in the review. Once the review is complete it can be pushed to the issue tracker interface using one of the configurable triggers provided by Code Collaborator. By tying the release note information to the review process for the code changes we were able to ensure that the review notes had been reviewed by several people and the details were entered while they were fresh in the minds of the developers.

The Importance of Checklists

When developers are reviewing code they will normally use a checklist to ensure standardization and identification of common errors. These lists can get very long as most are created to be comprehensive. Unfortunately, it soon becomes a tedious and unmanageable task when a couple of hundred lines of code are cross referenced against a list with twenty or more items. Even if the list is divided into categories, most people with average attention spans would not be able to maintain a focus on all of the items across all of the code. We need to keep the code review process sufficiently lightweight and efficient to ensure that we are gaining maximal benefit from a small number of reviewers.

There are a few questions that we need to answer when trying to develop a good checklist. How many items should be on the list? What should be, and should not be, on the list? How do we maintain the list over time?

Firstly, let's consider that research into the capacity of the human short-term memory shows we can handle about 7 items plus or minus 2 [2]. This would imply that the checklists should contain at most 9 items in order to keep them at the forefront of our mind as we scan through the code. Any more than that and we would need to repeatedly refer to the checklist in order to maintain awareness of any subset of items on the list, breaking our focus on the actual code under review.

As with debugging, the code review ends not when all of the defects have been discovered but when we become tired of looking for them. There is a significant risk of lowering the effectiveness of reviews if the checklists get too long and several passes over the code are needed to cover all the checklist items. If there isn't time to complete a full review be sure to at least look at the code with regard to a few points on the checklist. Even a partial review is better than no review at all.

Some review items don't appear on the checklist at all. An experienced developer has cultivated an instinctive set of personal rules for coding over the years, as they have learned from their own mistakes. These items do not need to be on the checklist since they can be considered as obvious to the individual reviewer. These "obvious" items, however, are communicated as each participant acknowledges the comments and defects marked by the other reviewers. This collaborative acknowledgment is an important aspect of the shared experience of reviewing the code. It builds not only a greater understanding and respect between remote teams but also provides the additional benefit of informally mentoring less experienced developers.

Most developers are looking for mistakes in the code itself, but just as important and often harder to spot, is what is missing. There are usually a number of things that are commonly forgotten. For instance, a developer is generally focused on how to make the code work and the reviewers will be looking to verify specific functionality. What is commonly missing is the consideration of error conditions. Well designed software will gracefully handle an unexpected execution path, so do not forget to add items to the review checklist that look for good error handling. The quality of the code is dependent upon things to go right even when something has gone wrong.

Additionally, remember that code review is not a replacement for testing and verification. There is little point keeping items on the checklist if they can be detected using an automated tool. The supplementary use of a static code analysis tool to check for stylistic and common programming problems is advisable. These tools can also reveal issues that are not easy to spot using a purely visual inspection of the code. A code review process typically ensures that developers run automated static analysis tests both before submitting the code for review and again after any rework is done.

Each code base and development team has their own dynamics which cannot easily be predicted. The code review tool allowed us to customize the defect reporting dialog and ensure that each defect found was identified by severity and category. By profiling the reviews to identify the most frequently tagged defects in each category, one can empirically build a prioritized list of commonly occurring issues. By using the frequency of the various categories of defects from the first few hundred reviews a checklist can be developed that is relevant to context of the development group. Once established, a complete checklist needs to be maintained by retiring items that have become stale and replacing them with those of a higher frequency or priority.

Building a good checklist is not as straightforward as grabbing an example from somewhere else. This data driven approach of eliminating the types of defects that appear to be most common in the code allows focus on those areas that require the most urgent quality improvement. Over time as the review metrics accumulate it is important to establish quantifiable goals towards process improvement and maintaining quality.

Which Metrics are Important?

The Code Collaborator tool collects information on the number of files and lines that have been modified, the number of defects found and the amount of time spent in the various phases of each review. Using these figures, configurable templates can be used to flag reviews that are considered trivial or stalled. For example, a review may be classified as trivial if no defects were found in a review of less than 30 seconds or stalled if no activity was logged for several days.

These raw numbers are influenced by many different parameters for a given group of developers or code base and will vary widely. While analysis may not provide a definitive measure of the effectiveness of the code review it is possible to interpret the data and identify rough trends to facilitate quality improvement.

Three measurements provide the basis for further analysis; defect density, inspection rate and defect rate:

- Defect density is the number of defects per 1000 lines of code. Considering that a reviewer may inspect files that vary in size from a few lines to a few thousand lines we would expect that more defects would be found in larger files. Likewise, the number of defects found in a review may differ given the wide variation in number of lines changed. By using defect density we can normalize the number of defects with respect to the amount of code under review.

- Inspection rate is the number of lines of code reviewed per hour of review and will typically vary from approximately 100 to 500 lines of code per hour.

- Defect rate is the number of defects found per hour of review. Again, reviews will vary in length depending on the number and range of the changes so it is useful to define these rates in order to normalize against time.

As we attempt to identify trends there are many factors that we need to consider. A large group will contain reviewers that are exceptionally meticulous but slow and those that are quick but more superficial. Inspection rates that are greater than 1000 lines per hour might indicate that reviewers were not inspecting code carefully but less than 10 lines per hour may be considered unproductive. Some files may be more sensitive to changes because they have algorithmically complex code or represent a reusable element that requires accurate implementation. Individual reviewers will be more or less effective at identifying defects than others based on their experience or familiarity with the code.

Even with some level of normalization we still need to be extremely careful when using these figures to draw conclusions. Trying to ascertain how fast the code should be reviewed or how many bugs per hour need to be found is not a useful goal in itself. Similarly, defect density does not reveal how buggy the code is. Abnormally low defect rates are not an indicator of high code quality and it is probable the opposite is true.

So can these metrics be meaningfully interpreted with respect to quality? As the number of reviews accumulates over time we expect to see a distribution evolving for each of the characteristics we are measuring. For instance, if we plot reviews measured by defect density we should see a distribution that we can use to determine our mean defect density. The same should be true for reviews plotted by inspection rate or defect rate. These means are important indicators which are useful for identifying the average standards of your group with normalization applied. The outlying data points may also warrant inspection though, since they may represent areas where a better understanding of the measurements is needed.

It is also important to perform periodic random audits on a small percentage of the reviews. The auditors should be an experienced team of reviewers looking for the highest level of compliance to expected quality standards. By comparing this sample against the overall statistics a determination can be made as to whether the mean values are acceptable or require action to be taken to improve them. This might be something like providing additional training, performing an in-depth review of a code tree or identifying a necessary process change. The metrics won't determine the absolute quality of the code but they will highlight issues that may require attention.

Gaining Cultural Acceptance

The introduction of a lightweight code review process is intended not only to move software quality upstream but also to create a collaborative culture of defect prevention. By creating an environment where discussion can conveniently occur not only around the water cooler but also around the world we essentially encourage an ethos of cooperative community. This may seem like an abstract concept, and difficult to measure, but it is hard to deny the benefits of creating a true sense of "team" given the realities of globally distributed developers.

So what can be done to encourage a group to enthusiastically adopt a new process? We took an approach based on consultation, prototyping, adoption and refinement. There was no "go live" date since the new infrastructure was developed alongside the existing one. The soft launch happened gradually and was expanded within the organization in stages. By presenting code review in a positive light and not as a punitive measure we aimed to convey a message of continuous improvement without highlighting an individual's productivity.

Initially we formed a small group of stakeholders in the code review process. Representatives from the software development, IT infrastructure and quality assurance organizations each had a chance to review the proposed process changes. Their feedback allowed us to tackle how the process could be designed to have a low barrier for adoption and acceptance, cause minimal disruption to the existing commitments on the schedule and scale as a wider roll-out.

As part of the prototyping a small team volunteered to be early adopters and help work out the kinks. We developed a small number of scripts to handle some of the most common operations such as the management of task branches and the attachment of review material to a review, eliminating many of the manual steps. The proof of concept was so successful that adoption spread initially by word-of-mouth and individuals began using the process voluntarily, even before they received training. We even reached some groups that we had not originally targeted, which led to further refinements in the process.

A number of concerns did surface during the prototyping that needed to be addressed. These related to ensuring that reviews did not stall and therefore impede code development. For example, one concern was how to shortcut the process when a delay to a release was not tolerable. The provision to allow developers to work around the new code review process was a useful part of the soft launch approach. The ability to skip the code review process and avoid using the task branches provided a safety net. Once the developers were familiar with the extra steps it became clear that the fear of being impeded by the process when a code change needed to be made urgently was unfounded. The process was lightweight and reliable enough to be workable at all times and ultimately the safety net was phased out.

A second concern was raised regarding how to allocate reviewers to reviews given an organization of over 100 developers. The allocation could stall the review if a particular reviewer was already overburdened, on vacation or simply inappropriately assigned. To overcome the allocation issue we implemented a concept of “reviewer pools”.

Reviewer pools are created in groups organized by technical expertise, geographical location or any other appropriate factor. The reviewer pools are implemented through an email group distribution alias. Individuals are able to subscribe or unsubscribe from the alias using an internal web page that also shows the current group memberships. When a review is initiated the reviewers can be allocated either as hand-picked individuals or opened up to one of the reviewer pools. This flexible allocation approach provides a mechanism to automatically broadcast notification of the review to the review pool subscribers.

Using this flexible allocation approach, individuals can then self-select to become a reviewer thus allowing a mechanism of cooperative load balancing. As long as the pool of reviewers for any given grouping is large enough, individuals can decide whether they are available to participate in any review to which they are invited. This structure allows people to manage their own review workload and also avoids delays associated with inviting an individual that may be legitimately unavailable. The problem of evenly distributing reviewers to reviews can therefore be alleviated.

Other teams external to the development group may choose to use a review pool alias acting as a brokerage instead of group distribution. Although the review pool is defined by technical area a notification recipient may act as a broker to reallocate the review role to an appropriate individual based on their expertise or availability. In either case, the reporting functionality of Code Collaborator is able to provide details of who has actually been allocated to the reviews and provide visibility of the workload balancing.

The consultative approach and prototyping stages were crucial in establishing credibility for challenging the status quo and introducing process changes. By demonstrating a flexible approach and incorporating process changes in response to feedback we cultivated a number of early advocates. Once an initial momentum had been established, adoption occurred at a steady rate as each group moved to align themselves with their peers.

Conclusions

We are currently at an early stage of adoption and still gathering a sufficiently large set of review data to perform a detailed analysis. Our early metrics show an apparently low defect rate of less than 1 defect per person-hour, with approximately only 1 in 10 reviews registering defects. A closer look at the review archive shows that reviewers were mostly engaged in discussion, using the comment threads to fix issues instead of logging defects. Issues fixed in the discussion threads are still valuable but they circumvent the metric gathering. The challenge is to educate reviewers to critically identify defects whilst not impeding the creative and collaborative process fostered by discussion.

The initial data regarding the average amount of time spent on reviews is more consistent with our expectations. The average total time per review is about 45 minutes with the average author time about 15 minutes and the average reviewer time about 30 minutes. These measurements are useful in gauging the cost-benefit associated with upstream defect prevention in comparison to late stage or post release fixes. With the introduction of a lightweight peer code review tool we have successfully managed to detect a significant number of defects that would have been more costly to eliminate later in the workflow.

It is important to consider that while the majority of code reviews involve inspection and feedback from a peer group of developers, there are several other cases where other parties, not involved in code development, can add value to the overall improvement of code quality. For example silicon design engineers have valuable input from the point of view of how the hardware was intended to be programmed and software support engineers have experience of system integration from a customer perspective. The code review tool allows us to easily connect these associated groups and facilitate discussion between them.

It seems apparent that additional training would help reviewers structure their approach to code review in a consistent and rigorous way. The inclusion of a checklist with the review material will remind reviewers of relevant items to look for and ensure inspection is performed in a systematic and objective manner. There are many possible factors related to software quality, such as conformance, reliability, maintainability, scalability, testability, documentation etc. The tracking of defects by category and severity will identify the most commonly occurring issues and help define and prioritize our checklists.

Successfully introducing a lightweight code review process is half the battle but we have a number of things we need to tackle next. The effective use of metrics will require rigorous analysis using standardized measurements to minimize the effect of individual reviewers or reviews. Once we can establish a baseline, the next step will be to set quantifiable goals and monitor our progress.

In conclusion, quality assurance is often a misleading term used to describe a process that would be more aptly named quality insurance, given that it occurs after code is developed. True quality assurance can only occur as development is in progress. To improve software quality an organization must be prepared to establish an objective measure and take a prioritized approach that is not compromised by schedule pressure. Implementing a modern, lightweight, and verifiable peer code review process represents a big step in that direction.

References

[1] Fagan, M.E., *Design and Code inspections to reduce errors in program development*, IBM Systems Journal, Vol. 15, No 3 1976

[2] Miller, G. A., *The magical number seven, plus or minus two: Some limits on our capacity for processing information,* Psychological Review, 63, 1956

[3] Jason Cohen, *Best Kept Secrets of Peer Code Review,* Smart Bear Software

Acknowledgments

For their valuable feedback and review;

Miska Hiltunen, Qualiteers
Sara Gorsuch, Symetra Inc.
Suzanne Gillespie, Kaiser Permanente Northwest, Center for Health Research
Lisa Wells, Smart Bear Software