Free software development resources on programming, software testing and project management

Rich Internet Application Monitoring

What's on your wallboard?
Win an HDTV, Mac Mini & more »

Ottawa Coupons
1 ridiculously huge coupon a day. Like doing Ottawa at 90% off!
www.Groupon.com/Ott

Ottawa Agile Training
Budget getting tighter? Get Agile Software Development training now.
www.WestboroSystem:

Click here to view the complete list of archived articles

This article was originally published in the Fall 2010 issue of Methods & Tools

Delivering Working Code through Automation and Collaboration

Todd Landry, Klocwork, www.klocwork.com

To keep pace with ever-increasing customer demands on software functionality and time-to-market expectations, software developers have had to evolve the way they develop code. The result is the emergence and rapid adoption of the Agile software development methodology.

By embracing the 'early and often' mantra of Agile, individual software teams and even entire software development organizations can deliver more technology, to more satisfied customers, earlier and more frequently than ever before. However, the reality for many teams trying to successfully execute on the Agile principles is a backlog of software defects interfering with their ability to deliver working software in short iterations.

This article will demonstrate that implementing a repeatable process for ensuring code is as bug-free as possible is key to fully realizing several of the core principals of the Agile methodology. The approach discussed is the use of automated source code analysis (SCA) technology to help developers find and fix areas of weakness in their software source code as they're coding. After providing a brief overview of Agile and discussing some of the barriers to achieving agility, this paper discusses how key capabilities of SCA technology can enhance the Agile development processes and empower Agile teams.

Defining Agile Development

Simply put, Agile software development is an approach that provides flexibility to accommodate continuous change throughout the software development cycle. It stresses rapid delivery of working software, empowerment of developers, and emphasizes collaboration and communication between developers and the rest of the team, including business people.

Agile contrasts with the still-popular Waterfall development approach, which is front-end loaded with comprehensive scope and requirements definitions, and which employs clear, consecutive hand-offs from requirements definition to design to coding and then to quality assurance. In contrast, Agile incorporates a continuous stream of requirements gathering that flows throughout the development process. Business people are involved throughout the release cycle, ensuring that the software being developed meets the true needs of both the end-user and the business. Change to the requirements and to the overall feature set is expected to occur as outside opportunities or threats arise.

In short, Agile fully embraces change and Agile teams are structured in such a way that they can receive and act on constant feedback provided by the build process, by other developers, from QA, and from business stakeholders.

Agile is based upon a number of guiding principles that all Agile teams follow. For the purposes of this discussion, four principles - or values - are of particular interest:

- Quality software development
- Iterative flexibility
- Continuous improvement
- Collaboration and communication

Quality Software Development

The primary focus of the Agile methodology is to enable the development of quality software that satisfies a customer need - i.e. provides a functioning feature or capability - within a specific period of time (typically no more than a few weeks) called an "iteration" or "sprint" in an Scrum context. In theory, a product developed in an Agile environment could be market-ready after each iteration.

Delivering a series of market-ready products, each in just weeks, demands that a rigorous quality process be built into the Agile development cycle. Deliverables in each iteration must be fully developed - meaning tested, defect-free, and complete with documentation.

Iterative Flexibility

With a focus on speed and nimbleness, Agile is open to changes that inevitably arise throughout the development cycle. The iterative process is flexible, based on an understanding that original requirements may (or will likely) need to change due to customer demand, market conditions, or other reasons. Because business users are involved throughout the process, and because each iteration is short, new requirements can be introduced and prioritized very quickly.

Continuous Improvement

An Agile environment provides developers with an opportunity to learn new skills and to exercise greater autonomy to do their jobs. The iterative framework is empowering because it enables continuous improvement, with testing/quality assurance occurring as part of the process, rather than only periodically or at the end of a long process when it is often difficult or not cost effective to fix coding defects or to incorporate lessons learned along the way. Agile also makes the

testing and QA process transparent to the developers who originate the code, further contributing to their learning and facilitating future improvements and coding efficiencies.

Collaboration and Communication

Communication and collaboration is critical in software development in general, but in an Agile development environment, its paramount. In fact, the Agile Manifesto (widely recognized as the de facto definition of Agile) emphasizes individuals and interactions as a key concept. Ultimately, it is open communication and collaboration that facilities efficiencies in the development process. Having access to the right individuals, data and feedback when needed allows the team to deliver working software in short iterations, as the Agile process demands.

Greasing Agile's Wheel

As development teams work to embrace the core principals set out in the Agile Manifesto, they are inevitably encountering roadblocks including bug debt, varying developer skill sets, and code complexity. Development teams need to acknowledge these barriers and employ effective strategies that will help 'grease the wheels' to achieving a truly Agile development process.

Managing Bug Debt

One of the development principles put forth in the Agile Manifesto states that "working software is the primary measure of progress." [1] Working software implies software that is free of issues that break builds, cause unexpected behavior, or which do not meet the product's requirements, as well as mundane programming defects (a.k.a. bugs).

This principle is not unique to Agile - many software development processes, including formal ones such as CMMI and Six Sigma, encourage the creation of bug-free code as a fundamental principle. These processes encourage in-phase bug containment - the practice of preventing bugs from being passed downstream from the phase in which they are created. Agile also implicitly emphasizes in-phase bug containment. Given its focus on short iterations, Agile processes must ensure that any potential software degradations are quickly identified and corrected so that the whole team can move on to the next iteration - all while creating functionally complete, working software.

Even within an iteration, Agile teams apply this philosophy through continuous integrations and regressions. While this practice effectively addresses defects that can break builds or regression test suites, it is not as effective in cleaning up many of the most common types of programming bugs, which generally fall into these broad categories:

- Memory and resource management
- Program data management
- Buffer overflows
- Unvalidated user input
- Vulnerable coding practices
- Concurrency violations
- A variety of longer term maintenance issues

Bug-filled code creates downstream risk both within an iteration and in subsequent iterations, in the form of bug-debt. If code flaws are not addressed within the iteration, they pass from one milestone to the next, and the bug debt accumulates downstream. Bug debt can kill Agile projects by reducing development velocity which leads to poor implementation results and fewer impactful changes being delivered per iteration. Additionally, bugs that are 'saved up' (or which go completely undetected until further downstream) are more expensive and often more difficult to fix, and can lead to products that fail in the field, disappoint customers, and damage the brand.

It is for these reasons that in-phase bug containment is vital to the Agile process. Developers must take control of the bug identification and removal process while enhancing collaboration amongst all developers to resolve bugs as early in the process as possible.

Accommodating Varying Skill Sets

Ideally, your development team is made up of the best and brightest software engineers who enjoy talking to each other and getting feedback from the customer on a frequent basis. Unfortunately, this doesn't always match reality. Typical development teams are made up of a heterogeneous talent pool, combining a mix of less experienced developers with the rock stars; natural collaborators with those that prefer anonymity, etc. Balancing out the team by providing opportunities for less experienced developers to learn from more experienced team-mates and providing ways to make communication and collaboration easier will help maximize the output of working code.

Restraining Code Complexity

Code bases are becoming increasing complex, due in part to the following factors:

- Code Base Size

  Expanding capabilities and the demand for more sophisticated features and functionality is increasing the size of code bases which leads to more variables and control paths as well as varying language constructs.

- Code Reuse

  Accepted as an industry best-practice, code re-use speeds time to market by increasing development efficiencies while minimizing the costs associated with net-new development. It also enables development organizations to leverage the lessons learned from an existing code base. However, merging that code with new systems can lead to unexpected results.

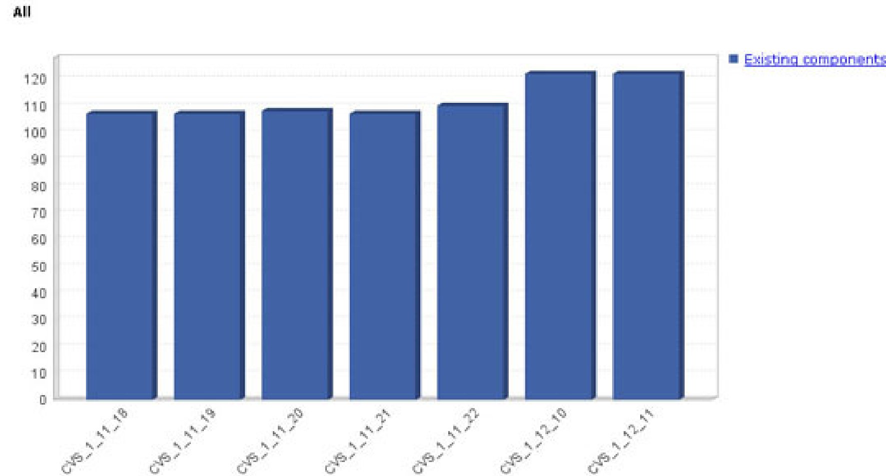
- Multiple Coders

  Code bases don't live and die within a single version or a single iteration, and in an Agile process it is quite likely that a developer will be editing code that they did not originally create. Inheriting modules means developers are unfamiliar with the original intent of the code, its variables and the language constructs used.

These factors are each contributing to the overall complexity of code bases, which in turn complicates the software development process. This complexity can lead to slipped release schedules, increased defect rates and decreased

productivity as more time is taken to accomplish simple, everyday tasks. Development teams should be vigilant in watching for these signs which indicated code complexity is becoming an issue.

With software, you can monitor the complexity of software builds using industry known metrics. For example, you can start monitoring your McCabe's Cyclometic complexity every build. This measurement shows you how "off" you are from the standard. (McCabe's complexity dictates anything over a value of 20 as very complex.) More importantly, teams should be monitoring the trend of that metric. Seeing a large spike with one version versus the previous might is evidence that complexity issues are impacting your process.



Tracking these metrics in order to identify opportunities to reduce complexity and make code more maintainable (regardless of who is working on it) should be considered. Not addressing code complexity will have a negative impact on the velocity of a software project - which can spell disaster in an Agile development environment.

Adapting Traditional Process Approaches

Traditional development processes become recognized as best practices because they bring value to the process - by speeding development, helping to produce cleaner code, etc. Unfortunately, not all of these practices are Agile-friendly. Take peer code review, for example. The value of code review is unarguable. However, in an Agile context, traditional code reviews - scheduling in-person meetings with senior development staff - are highly ineffective and time consuming. These types of traditional processes need to be adapted so that development teams can continue to benefit from them without sacrificing development velocity.

Adopting Tools and Processes to Achieve Agility

While the Agile Manifesto's principal of "individuals and interactions over processes and tools" seems to de-emphasis the need for tools, Agile teams use many tools to support their development - including tools for software configuration management, build management, requirements tracking, testing, project management, and more.

Unfortunately, most of the tools being used are for managing some aspect of the production process and are not targeted at helping developers overcome the obstacles they face during the coding phase. Deploying a properly-selected set of developer-focused tools can help ensure high-quality code is output early in the development process. The latest evolution of source code analysis technology - combining sophisticated bug detection with collaborative peer code review and automated code refactoring - can help development teams avoid bug debt and code maintainability issues, greasing the wheels to achieving an effective Agile process.

Automating Bug Detection

SCA is a bug-detection solution that requires no test cases, is fully automated, and fits well with milestones typically found in an Agile process. SCA technology has grown in popularity and is becoming a mainstream option for professional software developers to reduce the number of bugs in their code while also reducing costs and keeping software development on track.

The underlying technology associated with SCA is called static analysis and the current generation of technology solutions is capable of providing sophisticated, high-value analysis that will locate and describe areas of weakness in software source code - such as memory and resource management, program data management, buffer overflows, unvalidated user input, vulnerable coding practices, concurrency violations, and a variety of longer term maintenance issues. SCA is distinct from traditional dynamic analysis techniques, such as unit or penetration tests, because the work is performed at build-time using only the source code of the program or module in question. The results reported are therefore generated from a complete view of every possible execution path, rather than some aspect of a limited, observed runtime behavior.

Since SCA is essentially a build-time analysis, it is most effectively used as a build milestone activity when individual developers or development teams run their builds - either at the integration-build level or the developer-build level.

Integration Build (a.k.a. system build, project build)

Today's SCA tools go well beyond the syntactical and semantic analyses commonly associated with source code analysis. Good SCA technology today can be expected to include sophisticated inter-procedural control and data-flow analysis with advanced approaches for pruning false paths, estimating the values that variables will assume, and simulating potential runtime behavior. The complexity of this type of analysis across a large system with millions of lines of code and an essentially unlimited number of potentially feasible code paths to consider is not trivial.

To make it all work, and to reduce the number of "false positives" (bugs that the tool incorrectly reports) and false negatives (bugs that the tool misses), vendors naturally provide integration with a project's system build - whether it is make, ant, Visual Studio, or other continuous integration tools such as Electric Cloud and BuildForge - to generate a complete view of the entire source code base. Of course, the downside to running SCA exclusively at the integration build level is that bugs created at the desktop are exposed to the main code stream and can impact other members of the team - both fellow developers and the QA team. When bugs are found downstream - even in a continuous integration context where integration builds are run much more frequently - an additional bug triaging process needs to be put in place to notify the developer of the error (by email, web reports, etc.). This adds more workflow and process, which is contrary to the spirit of Agile development.

Clearly, the solution is to push SCA to the developer desktop so that it can run in conjunction with a developer's build, even prior to him/her running unit tests.

Developer Build (a.k.a. personal build, sandbox build)

SCA at the developer desktop offers a big payoff for any organization adopting this technology, in particular an Agile team. If most bugs can be found prior to code check-in, organizations will achieve in-phase bug containment, reducing the number of bugs in the main code stream or integration build. This allows QA to be more efficient by focusing on being a customer advocate and ultimately producing higher quality software - earlier.

For SCA to operate at the developer desktop, it must be delivered within the developer's natural work environment (i.e. a favorite IDE, text editor or command line) and the analysis must be every bit as accurate and intelligent as the centralized analysis that benefits from a view of the entire code stream. Code check-in (or commit) is an important milestone in Agile and many organizations operating in a continuous integration context have a series of gates (smoke tests, unit tests, etc.) that the developer must pass in order for him/her to check-in code. SCA should be added to this series of pre-check-in quality gates.

Facilitating Collaborative Peer Code Review

Peer code review is a necessary evil in the software development process. Often mandated by management before code can go live, the code review process is time-consuming and usually not a pleasant experience for the person whose code is being reviewed. However, the benefits are plentiful:

- Code reviews create consistency and a culture of quality for a development team. Identifying bugs and design flaws while ensuring coding standards and best practices are met means a higher quality product goes out the door.
- Developers learn from code reviews by having more experienced team members review their code and provide feedback.

In an Agile context, these benefits are important. However, the traditional method of conducting code reviews - scheduling in-person meetings - is not effective in an Agile context. Agile teams wanting to make code review part of their process should consider SCA tools with built-in code review capabilities. These tools facilitate simple, web-based peer review that allows code to be reviewed asynchronously. This approach avoids the need to get a specific group of people in a conference room at a set time to review code. Instead, code is available to a variety of team members - regardless of geographic location, calendar availability, and title - to review and provide feedback on at their convenience. When peer code review capabilities are combined with source code analysis, coding defects are identified so that reviewers can see them, comment on them, and assign actions related to any that require action.

Simplifying Complex Refactoring

Refactoring - the process of simplifying and clarifying code without changing the program's behavior - can be a difficult and time-consuming activity, especially for those developing in often-used languages like C and C++. With a lack of tools capable of automating the refactoring process, many developers who want to refactor simply choose not to. However, implementing a process that facilitates refactoring code early and often offers some important benefits to an Agile team.

Martin Fowler, one of the original authors of the Agile Manifesto, describes refactoring as a series of small, almost negligible changes to code, that when combined, result in a significant end result. Some of the areas that refactoring impacts includes:

- Adapting to change

  Ongoing change is a fundamental aspect of Agile. By incorporating frequent refactoring into the development process, code becomes easier to work with which allows teams to adapt to change more effectively.

- Defect detection

  By simplifying code and improving its overall design, code becomes clearer. Working with clean code makes spotting quality defects and security vulnerabilities easier. And the sooner defects are found and fixed, the less impact (i.e. distracting developers, bogging down test teams, slowing down the overall process) they have on the process.

- Developer productivity

  Frequently different people will end up working on code that they did not originally create. With refactoring, code becomes easier to inherit, which means developers won't waste iterations reviewing, interpreting and 'fixing' existing code and instead focus more time on writing new code.

Clearly, refactoring has an important role to play in helping teams achieve agility. With today's source code analysis tools which offer built-in refactoring capabilities, Agile teams can automate the refactoring process to reduce the complexity around the process while reaping the rewards it has to offer.

Achieving True Agility

By bringing source code analysis technology with its bug-detection, code review, and refactoring capabilities into the Agile development cycle, development teams can achieve in-phase bug containment while fully realizing the core Agile principals discussed at the beginning of this article.

Bug Free for Quality

Software bugs are more than a nuisance. Serious bugs can cause downstream inefficiencies, product recalls, or field disasters. Source code analysis can automatically find serious bugs in your code, such as NULL pointer dereferences and memory management issues that can lead to a system crash; or buffer overflows and unvalidated user inputs that make a system susceptible to exploit by hackers. Removing these issues prior to shipping a product is critical and the earlier they are identified the better. This prevents critical issues from being passed to QA within an iteration, or passed from iteration to iteration, both of which greatly increase the risk of shipping buggy software. In addition, if bugs are dramatically reduced before code check-in, this will ensure they never impact the main code stream and will facilitate a more efficient testing and QA process. With few bugs to find or report on, testers are able to focus instead on running functional and performance tests to ensure the product is customer- and market-ready.

Bug-Free Flexibility

To adequately test and assure the quality of software developed in Agile environments, the testing process must also be iterative and accommodate frequent change in requirements. Thus, there is little room to address programming bugs in the testing phase. If QA discovers these bugs, unacceptable drag in the development cycle is created -testers report the bug list back to the developers who must set aside their current work to shift back to the frame of mind they were in when working on the original code. By enabling developers to check-in bug-free code, this dramatically reduces or eliminates the time-consuming "rinse and repeat" cycle of check-in, find bugs, debug, and then rework. By using SCA tools in an Agile environment, developers can spend less time fixing reported bugs and more time writing new and innovative software. Within this context, developers have the flexibility to control the quality and security of the code they create during new development, before the integration build is performed.

Bug Free for Continuous Improvement

It doesn’t matter how an iteration appears to be progressing from a feature development standpoint - if the development team’s commitment to quality code cannot be measured, the project is by default accumulating significant downstream risk with each and every build and iteration. In a fast-paced and fluid development environment, Agile teams need to have strong, automated measurement capabilities in place, such as:

- Measure and track bug fix rate at the developer build;
- Identify what bugs are leaking to the integration build;
- Establish nightly build quality milestones (or any other frequency over and above the continuous build schedule); and,
- Track which teams or components are improving over time.

The answers to these questions allow Agile teams to implement targeted continuous improvement programs that quickly identify where help or training is needed so that all members of the development team can submit clean code into the integration build. Measuring and tracking quality from the bottom up is necessary to know if iteration plans are any good and if a shippable product will be available at the end of an iteration.

Bug Free for Collaboration and Communication

The short-iteration process found in Agile development requires constant communication and team collaboration in order to be successful. This helps ensure rapid resolution of issues encountered, while avoiding duplication of efforts which creates unnecessary delays. Source code analysis tools provide collaborative mitigation for all reported issues. Developers, architects and other team members can change the status of reported issues or add comments, which are automatically synchronized with other developers. Utilizing this capability, various team members can collaborate on complex issues, and developers don’t duplicate effort on the same bug. This ongoing visibility and sharing of data and feedback helps ensure mitigation is achieved early in the process.

The ROI of Source Code Analysis

Agile development teams need to be all about maximizing productivity. They need to increase - or at minimum maintain - team velocity and get as many stories completed in an iteration as possible. So how much can SCA tools help with this?

Typical ROI calculations focus on dollars saved, which is important at the senior manager and C-level. But for an Agile development team perspective, let’s look at hours that can be saved. First, we must establish a few key parameters around the Agile team. For the purposes of this paper, let’s assume:

- 10 developers that have standardized on 2 week development iterations
- Each iteration delivers approximately 5 completed stories
- Each story consists of 300 lines of code, so therefore a total of 1,500 lines of code per iteration is produced.

As mentioned earlier, bug debt can simply kill projects, so being able to find bugs as early as possible in the software development lifecycle is critical. On average, SCA tools find about 3 bugs for every 1,000 lines of code, and provide a time saving of 4 hours per bug found. Applying this to the typical iteration would lead to 4.5 bugs being detected, with a total time saving of 18 hours.

Code reviews are more and more becoming a mandatory part of the development process, and while they can be extremely effective, they do tend to take up considerable time and effort. Code reviews typically involve 4 or 5 developers sitting together in a room for an hour (on average) going through the code in question line-by-line. Doing this for every story in an iteration adds up to the tune of 20 hours. That is a significant amount of time that could be spent elsewhere. Case studies [2] show that when a developer can review code when and where they want to, they are 50% more productive than sitting around a table in a meeting room doing a review of the same code. This translates into a 10 hour saving per iteration.

Finally, starting work on a module you had never seen before can be a painstaking activity and can certainly hamper new development on that module. By performing some simple refactoring while writing the module, inheriting that code becomes slightly easier. Conservatively, let’s assume one quarter of an hour saved for every new story that work is started on because there is no need to ‘interpret’ what the previous developer produced. Working through the math, refactoring can result in a 12.5 hour time saving per iteration.

When adding up the time saving for these three activities, a total of just over 40 hours for every iteration can be realized and applied directly back to real development work.

	Time Savings/Iteration
Bug debt reduction	4.5 bugs x 4 hours = 18 hours
Peer code review	½ hour x 5 reviews x 4 developers = 10 hours
Refactoring	0.25 hours x 10 developers x 5 stories = 12.5 hours
TOTAL	40.5 hours/iteration

Time savings/iteration

Conclusion

The ubiquitous nature of software today, coupled with the pressure to rapidly develop market-ready features and products in just weeks, has led to two related phenomena:

- The widespread adoption of Agile software development principles; and,
- The adoption of various tools by Agile teams designed to help streamline and de-risk development projects.

Some of the most important types of tools that an Agile team can deploy are ones that aid in writing better-quality code. Source code analysis tools provide an automated method to detect a significant number of software bugs or security vulnerabilities right at the developer's desktop - before any code is delivered to the integration build or testing team. Combine that with asynchronous peer code review for easy collaboration and automated refactoring to ensure more maintainable code, and an Agile team can minimize project drag and run more efficiently overall. Developers can focus their time writing innovative code, while testing teams spend their time testing how the features of the project work rather than uncovering mundane code issues and retesting these again and again.

Source code analysis may be right for your Agile team, particularly if you are finding your process being impacted by quality issues or security vulnerabilities, non-Agile friendly processes, and hard to maintain code. Implementing source code analysis within your Agile environment does not have to be disruptive. You can start small and analyze only a small project or a portion of a project. Compare the results against a similar project where these tools were not used. You'll undoubtedly find opportunities to save significant time and money by using source code analysis in your Agile development process.

References

[1] Manifesto for Agile Software Development http://agilemanifesto.org/

[2] Diane Kelly, Terry Shepard, An experiment to investigate interacting versus nominal groups in software inspection, Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research, p. 122-134, October 06 -09, 2003, Toronto, Ontario, Canada.

Related Resources

SQAZone.net - Software Testing Quality Portal

SQA Planet - Software Quality Assurance Feeds Aggregator

Testing TV - Software Testing Videos and Tutorials Directory

Back to the archive list

Software Development Resources

Load Tester 4: The easiest and most complete load testing software
Oracle Magazine - Technology articles for developers and DBAs and more.
Open Source **Project management** Software Directory
Introduction to Software Development: Specific tools and techniques to develop free software
Project management TV - Tutorials and videos for the project manager
The Principles Of Project Management - Deliver projects on time and on budget, again and again

Copyright © 1995-2010 Martinig & Associates | Partners | Advertise | Contact | RSS | Twitter | LinkedIn | Facebook

Software Development Tools | Software Development Articles | Software Development Videos | Software Development Conferences | Software News
Software Development Books | Software Development Blogs | Software Development Directory | Software Development Editor's Blog