Continuous integration in agile development

How agile methods, continuous integration, and test-driven enhance design and development of complex systems

Martin R. Bakal
Jennifer Althouse
Paridhi Verma

August 14, 2012

This article explores how agile development, continuous integration (CI), and test-driven development (TDD) techniques can be employed in embedded software development. When applied as part of an architecture-based approach, these combined practices provide both high quality and project flexibility.

For a decade or more, software teams have benefited from agile development methods. They have adopted these iterative and incremental development practices, where solutions evolve through collaborative development. Traditional, non-agile approaches to creating software typically rely on a more regimented flow of development. An example of this is the waterfall process, where each activity of requirements, design, development, and testing is done serially.

Although waterfall development was the standard for large, complex system development for many years, it has several notable flaws. The first is that a lot of work is wasted trying to complete documents before designs and designs before code, even though it is well known that the requirements will change over time. Another is that, in holding off on testing and integration until the end of a project, issues are frequently discovered too late to be resolved without causing missed deadlines. These two factors, combined, might have been tolerable in a world that moved at a slower pace. But as the pressure to create innovative systems has increased, the ability of that approach to meet the needs of organizations has decreased.

Even though they were popularized by teams developing IT systems, agile practices can apply equally well to product development where the product consists of hardware, electronics, and software. Embedded software development differs from IT application development primarily in the limited availability of deployment target resources, such as processor performance and memory. Embedded software often performs complex real-time operations in these constrained conditions. Think about a computer-controlled system like the airbags in your car. You need them to deploy immediately, but you also need them to deploy reliably. The agile methods were originally designed for smaller, co-located project teams in unregulated industries. It took many years for them to
be stretched so that the agile approach could accommodate larger, more complex development projects.

When applied as part of an architecture-based approach, continuous integration (CI) and test-driven development (TDD) extend basic agile practices enough to provide both high quality and project flexibility. This article explores how agile methods, CI, and TDD can be employed in the context of embedded software development. It also describes the benefits of this combination.

**How continuous integration and test-driven development fit into agile practice**

Most people have heard of agile methods by now. The concepts that they brought to software development changed the way teams organize their work, adapt to changing requirements, and release software. Continuous integration (CI) was created for agile development, so the agile approach is the context for any CI discussion. It organizes development into functional *user stories*. These stories are prioritized into smaller groups of work, or *sprints*.

The idea is not to try to solve every issue up front but, instead, to focus on what you already know. So the team designs, builds, and tests what they know about the desired functionality. This creates a working product based on a subset of the complete product's requirements. Then the team moves on to the next-highest priority set of requirements and repeats the process. Of course, this is a very simplified view, and there are many variants of this process, but that's the core: Build your product incrementally, and try to improve things as you go.

According to Martin Fowler of ThoughtWorks, *continuous integration* is a software development practice that requires team members to integrate their work frequently. Every person integrates at least daily, which leads to multiple integrations each day. Integrations are verified by an automated build that runs regression tests to detect integration errors as quickly as possible. Teams find that this approach leads to significantly fewer integration problems and enables development of cohesive software more rapidly.

This leads to the final detail that makes for successful execution of a CI process. If the idea of continuously integrating is to find issues quickly, thus giving each developer feedback on their work, then there must be some way to evaluate that work quickly. Test-driven development fills that gap. With TDD, you build the test and then develop functionality until the code passes the test. As each new addition to the code is made, its test can be added to the suite of tests that are run when you build the integrated work. This ensures that new additions don't break the functioning work that came before them, and developers whose code does in fact "break the build" can be notified quickly. A typical combination of continuous integration and test-driven development is illustrated in Figure 1.
Types of projects that benefit from continuous integration

Teams of fewer than 50 people working on less complex projects were certainly the proving ground for both agile development and CI. But as products have become "smarter," there has been a significant increase in their complexity.

The amount of embedded software going into traditional products is astounding. Today, a new car is marketed less by its horsepower and more by its embedded software technology (self-park, advanced safety warnings, fuel efficiency, infotainment system, for example). The number of lines of code written to create a new car is higher than the number of lines of code written for an F16 fighter jet.

This increase in the complexity of products has come simultaneously with a hastening of time to market for new products. The prevalence of embedded software, combined with tighter deadlines, has brought agile practices and CI to the embedded developers.

Using agile methods for embedded systems development

Agile methods allow software and systems teams to respond quickly to change. The agile approach reduces the schedule risk associated with traditional software engineering, in which integration of component parts is treated as a late-phase effort. This late-phase integration causes misinterpretations of design specs to be discovered too late for teams to correct the issues and still meet their deadlines.

Yet systems teams that produce more than just the software component have been skeptical of certain aspects of the agile approach. Remove too much of the early planning, they say, and you end up with poor software and hardware integration. Without early-and-often checkpoints that validate progress against the architectural blueprint, a team can fail to produce a component that
can function in the broader system. Furthermore, for complex-system developers who are seeking reusability of design or scalability to larger project requirements, agile methods can appear limited.

These fears are understandable, because modeling and architecture have not been the hallmarks of agile techniques. But the CI approach to systems development offers several improvements over pure agile methods. CI helps systems development teams be agile and respond to rapid business changes, while at the same time ensuring that the actual hardware and software under development are in constant sync. CI allows team members to work effectively in their domain groups, focused on the tasks that they're best at accomplishing. At the end of each day, they know that their contributions to the project are integrated and that the component parts work together. And if something doesn't integrate, it's quickly discovered.

Let's consider some of the essential ingredients of complex system development and delivery and explore how CI helps meet the challenges.

**Architecture**

When you're building a complex system, you cannot add features continuously without a blueprint. Without a blueprint, all you will achieve with extra iterations is more rework opportunities. Whether you refer to it as a blueprint, model, or architecture, it provides a solid foundation on which to begin the iterative process.

Architecture can be helpful in smaller projects of 50 or fewer team members, but when you grow past that size, you definitely require this up-front work to understand componentization, reuse, and variability. This up-front analysis allows you to break into teams yet still release a coordinated product. The same holds when you have hardware and software developers to work together, as you do for complex systems with embedded software.

**Simulation**

By capturing the architecture in a simulation model, the team can see how the system will respond to different inputs. This form of early testing allows validation that the system performs as intended, thereby meeting the requirements. It also allows the designer to visualize any unintended consequences of the design. These unintended consequences are very difficult to see when examining code in text. They become far more obvious when viewing a model of the system, and they are even more obvious when viewing that model of the system in action.

In this way, modeling and simulation allow testing and integration to begin as soon as the design work begins, which eliminates the delays that might be experienced if the embedded hardware isn't yet available. It can save significant investment in unneeded early prototyping of architectures that aren't viable. Even when you do have the hardware available, continuous integration requires constant builds.

The sooner you need to see results, the more expensive your build environment becomes. Because the primary purpose of CI is providing results as quickly as possible, simulation allows you to test without inordinately high hardware costs. It also provides an easier way to
communicate the functionality of a component, which can be valuable for the pair programming and "code review" that is common in agile development.

**Build automation**

Continuous integration requires build automation, which is the ability to have software automatically compiled and linked into an executable. Speed is important because large builds can take a long time. Without fast, reliable builds, you lack the insight needed to resolve integration problems that arise. Conflicts between changes that were contributed by two or more developers are identified for resolution when the integration build is run. So, if an issue is found, the developer working to resolve the previous build's conflict can test the revised code through hardware simulation without delaying the other developers. But to achieve this efficiency, the integration build must occur continuously, kicking off the new build as soon as the previous one finishes. This is very different from the once per day or once per week builds that other processes employ.

Of course, this method requires build automation, because it would be impractical to assign a person the task of repeatedly starting a build, over and over, throughout the day. Moreover, the build should execute quickly, which often requires the builds to be multithreaded. A multithreaded build takes different components of the software and executes them in parallel with the build running on some other component, which speeds up the aggregate build time. It does require more hardware and a more complex script. The more complicated the script becomes, the more valuable a build management tool becomes.

**Work management**

A primary agile concept is the value of breaking down work into small, manageable chunks. This is also the basic premise behind CI: to fix your bugs early and often. This keeps them from compounding into larger, harder to resolve issues later in the project.

One of the things that this technique provides is the ability to deliver smaller, functional releases that are built and tested at many dates along the project schedule. Each delivery reduces the project risk by validating the architecture, requirements, and schedule estimates from the team. In agile methods, the work yet to be completed is called the **backlog**. As you start to assign the work to small delivery increments, called **sprints**, the work allocated to a sprint is called the **sprint backlog**. The remaining work to be allocated to future sprints is called the **project backlog**. The goal is to group only as many work items into a sprint as can be achieved in the sprint's defined timeframe.

This process is highly dependent on the collection of metrics so that a team can more accurately predict the amount of time that tasks will require and, thus, the number of tasks that can fit into a sprint's delivery. However, as valuable as these metrics are, the data collection is very tedious even for a small team. As you group these small teams together to produce a more complex product, the task is quite unwieldy to complete manually.

There are many products on the market that can help organize the work, track its completion, and produce the metrics associated with how much, how fast, how well, etc. the work is accomplished. When CI practices are followed, the integration errors identified must also be added quickly to
this backlog of work and moved to the top of the list as high-priority. In this respect, the best products on the market offer some level of integration between the new work items and the build management system, so that bugs identified after each build can be fixed quickly and integrated with the existing work items, escalated in priority, and routed to the right team.

**Quality management**

Quality management is the development lifecycle practice of ensuring that all of the requirements for your product have been tested. This effort needs to be organized and understood so that the correct tests can be updated when requirements change. Quality management helps project managers answer these questions:

- If my product had to release next week, which parts would present the most risk?
- Can we release without a lower-level requirement?
- Will this be a high-quality release?

In the face of market pressures that speed up delivery cycles, having quick answers to these questions can help businesses confidently release products into the marketplace. Managers better understand what resources to add, where to trim product features, and when to re-establish delivery dates for maximum advantage.

With test-driven development, the concept of testing becomes more central to the development effort. In TDD you write the test based on the requirement, and then you develop code until it passes the test. This ensures that no extra functionality is created, which would be something that development teams call "gold plating." Even if an additional function or feature seems like a good idea, when no requirement is driving the decision, the extra work can raise costs and increase time to delivery. And it might not actually increase customer satisfaction with the final product.

**Automated testing**

When you are creating multiple builds, the team is required to retest functions that were found to be working in previous releases. This process of retesting, previously known "good code," is called regression testing. It ensures that bugs are not introduced or reintroduced into previously tested code by changes that were just made. With CI, automated regression tests are scripted to run at the end of each build. This enables developers to get instant feedback regarding bugs found in the new build. It's the step that alerts the developers when new code that they've produced is (or isn't) working as required. Without regression tests, developers simply know that the build is completed. Because the tests have to be created anyway, TDD doesn't add extra work. It simply reverses the order of things by creating the tests first, and then the code.

A traditional waterfall development project might survive without any test automation. The project could be described, built, and then tested endlessly by an army of people. But as soon as you begin to release on a regular basis, problems arise with that process. It is not feasible to manually test a system that is being built many times a day.
Collaboration

The IBM® Rational® software organization has long been an advocate of collaboration as a critical ingredient for successful system development and delivery. But in CI, where both software and hardware teams are involved, collaboration includes not only efficient hand-off of artifacts from one team to the next, but also a fully orchestrated understanding of the tradeoffs among requirements, features, and deadlines.

Good architecture enables this sort of collaboration partly because people can better understand the dependencies among the various components that they’re building. With project portfolio management, you can understand features, reuse, and resource allocation. But in co-developed hardware and software projects, it’s also important to manage requirements and to make intelligent decisions about when those requirements may change and when they must not.

Such projects typically involve many stakeholders at multiple levels of decision-making. Good collaboration helps satisfy a larger percentage of stakeholders. It ensures that the right product gets created and that deviations from the broader goals are identified quickly. This produces a product that better satisfies customer demand.

Summary

From a technical perspective, CI helps teams work more efficiently. These teams can be cross-functional, creating hardware and software that works together. They can be geographically distributed, because the constant integration work will ensure that you don’t get deviating designs. People can work on a large team, because the different components of a complex system will more assuredly work together. It solves many of the early pitfalls that these nontraditional agile teams might have experienced without CI. Combining CI with test-driven development puts more people under the agile umbrella, because it allows agile methods to work more efficiently.

From a business perspective, CI offers better business results by allowing teams to have their cake and eat it too. That is, they can bring products to market faster, by finding issues when they are young and small, not waiting until they are large and more difficult to fix. They can also respond better to requirements that are introduced while the product is being development. This creates a better product for the customer, which is the real promise of agility.
Related topics

- Listen to the podcast Continuous integration comes of age. To find this podcast, search for #204.
- Watch the recorded webcast Continuous integration for agile embedded software development.
- For more information on quality management:
  - Read the paper by Moshe Cohen, Smarter quality management: The fast track to competitive advantage (IBM Software Thought Leadership White Paper, June 2011, PDF download).
  - For more about continuous integration, see the Practices of Continuous Integration section of Martin Fowler's website.
- Subscribe to the developerWorks weekly email newsletter, and choose the topics to follow.
- Evaluate other IBM software in the way that suits you best.

© Copyright IBM Corporation 2012
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)