Continuous Integration

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. This article is a quick overview of Continuous Integration summarizing the technique and its current usage.

01 May 2006

Martin Fowler

Translations: Portuguese · Chinese · Korean · French · Chinese · Czech ·

Find similar articles to this by looking at these tags: popular · agile · delivery · extreme programming · continuous integration

For more information on this, and related topics, take a look at my guide page for delivery.

ThoughtWorks, my employer, offers consulting and support around Continuous Integration. The open source CruiseControl, the first continuous integration server, was originally created at ThoughtWorks. Recently ThoughtWorks Studios, our products group, has released Go - a new server for continuous integration and delivery. It supports parallel, deployment pipelines with multiple projects, a snazzy looking dashboard, and support for automated deployments. It's a commercial tool, but is free to use for small setups.

Contents

I vividly remember one of my first sightings of a large software project. I was taking a summer internship at a large English electronics company. My manager, part of the QA group, gave me a tour of a site and we entered a huge depressing warehouse stacked full with cubes. I was told that this project had been in development for a couple of years and was currently integrating, and had been integrating for several months. My guide told me that nobody really knew how long it would take to finish integrating. From this I learned a common story of software projects: integration is a long and unpredictable process.

But this needn't be the way. Most projects done by my colleagues at ThoughtWorks,

and by many others around the world, treat integration as a non-event. Any individual developer's work is only a few hours away from a shared project state and can be integrated back into that state in minutes. Any integration errors are found rapidly and can be fixed rapidly.

This contrast isn't the result of an expensive and complex tool. The essence of it lies in the simple practice of everyone on the team integrating frequently, usually daily, against a controlled source code repository.

When I've described this practice to people, I commonly find two reactions: "it can't work (here)" and "doing it won't make much difference". What people find out as they try it is that it's much easier than it sounds, and that it makes a huge difference to development. Thus the third common reaction is "yes we do that - how could you live without it?"

The original article on Continuous Integration describes our experiences as Matt helped put together continuous integration on a ThoughtWorks project in 2000.

The term 'Continuous Integration' originated with the Extreme Programming development process, as one of its original twelve practices. When I started at ThoughtWorks, as a consultant, I encouraged the project I was working with to use the technique. Matthew Foemmel turned my vague exhortations into solid action and we saw the project go from rare and complex integrations to the non-event I described. Matthew and I wrote up our experience in the original version of this paper, which has been one of the most popular papers on my site.

Although Continuous Integration is a practice that requires no particular tooling to deploy, we've found that it is useful to use a Continuous Integration server. The best known such server is CruiseControl, an open source tool originally built by several people at ThoughtWorks and now maintained by a wide community. Since then several other CI servers have appeared, both open source and commercial - including Cruise from ThoughtWorks Studios.

Building a Feature with Continuous Integration

The easiest way for me to explain what CI is and how it works is to show a quick example of how it works with the development of a small feature. Let's assume I have to do something to a piece of software, it doesn't really matter what the task is, for the moment I'll assume it's small and can be done in a few hours. (We'll explore longer tasks, and other issues later on.)

I begin by taking a copy of the current integrated source onto my local development machine. I do this by using a source code management system by checking out a working copy from the mainline.

The above paragraph will make sense to people who use source code control systems, but be gibberish to those who don't. So let me quickly explain that for the latter. A source code control system keeps all of a project's source code in a repository. The current state of the system is usually referred to as the 'mainline'. At any time a developer can make a controlled copy of the mainline onto their own machine, this is called 'checking out'. The copy on the developer's machine is called a 'working copy'. (Most of the time you actually update your working copy to the mainline - in practice it's the same thing.)

Now I take my working copy and do whatever I need to do to complete my task. This will consist of both altering the production code, and also adding or changing

automated tests. Continuous Integration assumes a high degree of tests which are automated into the software: a facility I call self-testing code. Often these use a version of the popular XUnit testing frameworks.

Once I'm done (and usually at various points when I'm working) I carry out an automated build on my development machine. This takes the source code in my working copy, compiles and links it into an executable, and runs the automated tests. Only if it all builds and tests without errors is the overall build considered to be good.

With a good build, I can then think about committing my changes into the repository. The twist, of course, is that other people may, and usually have, made changes to the mainline before I get chance to commit. So first I update my working copy with their changes and rebuild. If their changes clash with my changes, it will manifest as a failure either in the compilation or in the tests. In this case it's my responsibility to fix this and repeat until I can build a working copy that is properly synchronized with the mainline.

Once I have made my own build of a properly synchronized working copy I can then finally commit my changes into the mainline, which then updates the repository.

However my commit doesn't finish my work. At this point we build again, but this time on an integration machine based on the mainline code. Only when this build succeeds can we say that my changes are done. There is always a chance that I missed something on my machine and the repository wasn't properly updated. Only when my committed changes build successfully on the integration is my job done. This integration build can be executed manually by me, or done automatically by Cruise.

If a clash occurs between two developers, it is usually caught when the second developer to commit builds their updated working copy. If not the integration build should fail. Either way the error is detected rapidly. At this point the most important task is to fix it, and get the build working properly again. In a Continuous Integration environment you should never have a failed integration build stay failed for long. A good team should have many correct builds a day. Bad builds do occur from time to time, but should be quickly fixed.

The result of doing this is that there is a stable piece of software that works properly and contains few bugs. Everybody develops off that shared stable base and never gets so far away from that base that it takes very long to integrate back with it. Less time is spent trying to find bugs because they show up quickly.

Practices of Continuous Integration

The story above is the overview of CI and how it works in daily life. Getting all this to work smoothly is obviously rather more than that. I'll focus now on the key practices that make up effective CI.

Maintain a Single Source Repository.

Software projects involve lots of files that need to be orchestrated together to build a product. Keeping track of all of these is a major effort, particularly when there's multiple people involved. So it's not surprising that over the years software development teams have built tools to manage all this. These tools - called Source Code Management tools, configuration management, version control systems, repositories, or various other names - are an integral part of most development

projects. The sad and surprising thing is that they aren't part of *all* projects. It is rare, but I do run into projects that don't use such a system and use some messy combination of local and shared drives.

So as a simple basis make sure you get a decent source code management system. Cost isn't an issue as good quality open-source tools are available. The current open source repository of choice is Subversion. (The older open-source tool CVS is still widely used, and is much better than nothing, but Subversion is the modern choice.) Interestingly as I talk to developers I know most commercial source code management tools are liked less than Subversion. The only tool I've consistently heard people say is worth paying for is Perforce.

Once you get a source code management system, make sure it is the well known place for everyone to go get source code. Nobody should ever ask "where is the foo-whiffle file?" Everything should be in the repository.

Although many teams use repositories a common mistake I see is that they don't put everything in the repository. If people use one they'll put code in there, but everything you need to do a build should be in there including: test scripts, properties files, database schema, install scripts, and third party libraries. I've known projects that check their compilers into the repository (important in the early days of flaky C++ compilers). The basic rule of thumb is that you should be able to walk up to the project with a virgin machine, do a checkout, and be able to fully build the system. Only a minimal amount of things should be on the virgin machine - usually things that are large, complicated to install, and stable. An operating system, Java development environment, or base database system are typical examples.

You must put everything required for a build in the source control system, however you may also put other stuff that people generally work with in there too. IDE configurations are good to put in there because that way it's easy for people to share the same IDE setups.

One of the features of version control systems is that they allow you to create multiple branches, to handle different streams of development. This is a useful, nay essential, feature - but it's frequently overused and gets people into trouble. Keep your use of branches to a minimum. In particular have a **mainline**: a single branch of the project currently under development. Pretty much everyone should work off this mainline most of the time. (Reasonable branches are bug fixes of prior production releases and temporary experiments.)

In general you should store in source control everything you need to build anything, but nothing that you actually build. Some people do keep the build products in source control, but I consider that to be a smell - an indication of a deeper problem, usually an inability to reliably recreate builds.

Automate the Build

Getting the sources turned into a running system can often be a complicated process involving compilation, moving files around, loading schemas into the databases, and so on. However like most tasks in this part of software development it can be automated - and as a result should be automated. Asking people to type in strange commands or clicking through dialog boxes is a waste of time and a breeding ground for mistakes.

Automated environments for builds are a common feature of systems. The Unix world has had make for decades, the Java community developed Ant, the .NET

community has had Nant and now has MSBuild. Make sure you can build and launch your system using these scripts using a single command.

A common mistake is not to include everything in the automated build. The build should include getting the database schema out of the repository and firing it up in the execution environment. I'll elaborate my earlier rule of thumb: anyone should be able to bring in a virgin machine, check the sources out of the repository, issue a single command, and have a running system on their machine.

Build scripts come in various flavors and are often particular to a platform or community, but they don't have to be. Although most of our Java projects use Ant, some have used Ruby (the Ruby Rake system is a very nice build script tool). We got a lot of value from automating an early Microsoft COM project with Ant.

A big build often takes time, you don't want to do all of these steps if you've only made a small change. So a good build tool analyzes what needs to be changed as part of the process. The common way to do this is to check the dates of the source and object files and only compile if the source date is later. Dependencies then get tricky: if one object file changes those that depend on it may also need to be rebuilt. Compilers may handle this kind of thing, or they may not.

Depending on what you need, you may need different kinds of things to be built. You can build a system with or without test code, or with different sets of tests. Some components can be built stand-alone. A build script should allow you to build alternative targets for different cases.

Many of us use IDEs, and most IDEs have some kind of build management process within them. However these files are always proprietary to the IDE and often fragile. Furthermore they need the IDE to work. It's okay for IDE users set up their own project files and use them for individual development. However it's essential to have a master build that is usable on a server and runnable from other scripts. So on a Java project we're okay with having developers build in their IDE, but the master build uses Ant to ensure it can be run on the development server.

Make Your Build Self-Testing

Traditionally a build means compiling, linking, and all the additional stuff required to get a program to execute. A program may run, but that doesn't mean it does the right thing. Modern statically typed languages can catch many bugs, but far more slip through that net.

A good way to catch bugs more quickly and efficiently is to include automated tests in the build process. Testing isn't perfect, of course, but it can catch a lot of bugs - enough to be useful. In particular the rise of Extreme Programming (XP) and Test Driven Development (TDD) have done a great deal to popularize self-testing code and as a result many people have seen the value of the technique.

Regular readers of my work will know that I'm a big fan of both TDD and XP, however I want to stress that neither of these approaches are necessary to gain the benefits of self-testing code. Both of these approaches make a point of writing tests before you write the code that makes them pass - in this mode the tests are as much about exploring the design of the system as they are about bug catching. This is a Good Thing, but it's not necessary for the purposes of Continuous Integration, where we have the weaker requirement of self-testing code. (Although TDD is my preferred way of producing self-testing code.)

For self-testing code you need a suite of automated tests that can check a large part

of the code base for bugs. The tests need to be able to be kicked off from a simple command and to be self-checking. The result of running the test suite should indicate if any tests failed. For a build to be self-testing the failure of a test should cause the build to fail.

Over the last few years the rise of TDD has popularized the XUnit family of open-source tools which are ideal for this kind of testing. XUnit tools have proved very valuable to us at ThoughtWorks and I always suggest to people that they use them. These tools, pioneered by Kent Beck, make it very easy for you to set up a fully self-testing environment.

XUnit tools are certainly the starting point for making your code self-testing. You should also look out for other tools that focus on more end-to-end testing, there's quite a range of these out there at the moment including FIT, Selenium, Sahi, Watir, FITnesse, and plenty of others that I'm not trying to comprehensively list here.

Of course you can't count on tests to find everything. As it's often been said: tests don't prove the absence of bugs. However perfection isn't the only point at which you get payback for a self-testing build. Imperfect tests, run frequently, are much better than perfect tests that are never written at all.

Everyone Commits To the Mainline Every Day

Integration is primarily about communication. Integration allows developers to tell other developers about the changes they have made. Frequent communication allows people to know quickly as changes develop.

The one prerequisite for a developer committing to the mainline is that they can correctly build their code. This, of course, includes passing the build tests. As with any commit cycle the developer first updates their working copy to match the mainline, resolves any conflicts with the mainline, then builds on their local machine. If the build passes, then they are free to commit to the mainline.

By doing this frequently, developers quickly find out if there's a conflict between two developers. The key to fixing problems quickly is finding them quickly. With developers committing every few hours a conflict can be detected within a few hours of it occurring, at that point not much has happened and it's easy to resolve. Conflicts that stay undetected for weeks can be very hard to resolve.

The fact that you build when you update your working copy means that you detect compilation conflicts as well as textual conflicts. Since the build is self-testing, you also detect conflicts in the running of the code. The latter conflicts are particularly awkward bugs to find if they sit for a long time undetected in the code. Since there's only a few hours of changes between commits, there's only so many places where the problem could be hiding. Furthermore since not much has changed you can use diff-debugging to help you find the bug.

My general rule of thumb is that every developer should commit to the repository every day. In practice it's often useful if developers commit more frequently than that. The more frequently you commit, the less places you have to look for conflict errors, and the more rapidly you fix conflicts.

Frequent commits encourage developers to break down their work into small chunks of a few hours each. This helps track progress and provides a sense of progress. Often people initially feel they can't do something meaningful in just a few hours, but we've found that mentoring and practice helps them learn.

Every Commit Should Build the Mainline on an Integration Machine

Using daily commits, a team gets frequent tested builds. This ought to mean that the mainline stays in a healthy state. In practice, however, things still do go wrong. One reason is discipline, people not doing an update and build before they commit. Another is environmental differences between developers' machines.

As a result you should ensure that regular builds happen on an integration machine and only if this integration build succeeds should the commit be considered to be done. Since the developer who commits is responsible for this, that developer needs to monitor the mainline build so they can fix it if it breaks. A corollary of this is that you shouldn't go home until the mainline build has passed with any commits you've added late in the day.

There are two main ways I've seen to ensure this: using a manual build or a continuous integration server.

The manual build approach is the simplest one to describe. Essentially it's a similar thing to the local build that a developer does before the commit into the repository. The developer goes to the integration machine, checks out the head of the mainline (which now houses his last commit) and kicks off the integration build. He keeps an eye on its progress, and if the build succeeds he's done with his commit. (Also see Jim Shore's description.)

A continuous integration server acts as a monitor to the repository. Every time a commit against the repository finishes the server automatically checks out the sources onto the integration machine, initiates a build, and notifies the committer of the result of the build. The committer isn't done until she gets the notification - usually an email.

At ThoughtWorks, we're big fans of continuous integration servers - indeed we led the original development of CruiseControl and CruiseControl.NET, the widely used open-source CI servers. Since then we've also built the commercial Cruise CI server. We use a CI server on nearly every project we do and have been very happy with the results.

Not everyone prefers to use a CI server. Jim Shore gave a well argued description of why he prefers the manual approach. I agree with him that CI is much more than just installing some software. All the practices here need to be in play to do Continuous Integration effectively. But equally many teams who do CI well find a CI server to be a helpful tool.

Many organizations do regular builds on a timed schedule, such as every night. This is not the same thing as a continuous build and isn't enough for continuous integration. The whole point of continuous integration is to find problems as soon as you can. Nightly builds mean that bugs lie undetected for a whole day before anyone discovers them. Once they are in the system that long, it takes a long time to find and remove them.

A key part of doing a continuous build is that if the mainline build fails, it needs to be fixed right away. The whole point of working with CI is that you're always developing on a known stable base. It's not a bad thing for the mainline build to break, although if it's happening all the time it suggests people aren't being careful enough about updating and building locally before a commit. When the mainline build does break, however, it's important that it gets fixed fast. To help avoid breaking the mainline you might consider using a pending head.

When teams are introducing CI, often this is one of the hardest things to sort out.

Early on a team can struggle to get into the regular habit of working mainline builds, particularly if they are working on an existing code base. Patience and steady application does seem to regularly do the trick, so don't get discouraged.

Keep the Build Fast

The whole point of Continuous Integration is to provide rapid feedback. Nothing sucks the blood of a CI activity more than a build that takes a long time. Here I must admit a certain crotchety old guy amusement at what's considered to be a long build. Most of my colleagues consider a build that takes an hour to be totally unreasonable. I remember teams dreaming that they could get it so fast - and occasionally we still run into cases where it's very hard to get builds to that speed.

For most projects, however, the XP guideline of a ten minute build is perfectly within reason. Most of our modern projects achieve this. It's worth putting in concentrated effort to make it happen, because every minute you reduce off the build time is a minute saved for each developer every time they commit. Since CI demands frequent commits, this adds up to a lot of time.

If you're staring at a one hour build time, then getting to a faster build may seem like a daunting prospect. It can even be daunting to work on a new project and think about how to keep things fast. For enterprise applications, at least, we've found the usual bottleneck is testing - particularly tests that involve external services such as a database.

Probably the most crucial step is to start working on setting up a deployment pipeline. The idea behind a **deployment pipeline** (also known as **build pipeline** or **staged build**) is that there are in fact multiple builds done in sequence. The commit to the mainline triggers the first build - what I call the commit build. The **commit build** is the build that's needed when someone commits to the mainline. The commit build is the one that has to be done quickly, as a result it will take a number of shortcuts that will reduce the ability to detect bugs. The trick is to balance the needs of bug finding and speed so that a good commit build is stable enough for other people to work on.

Once the commit build is good then other people can work on the code with confidence. However there are further, slower, tests that you can start to do. Additional machines can run further testing routines on the build that take longer to do.

Jez Humble and Dave Farley extended these ideas into the topic of Continuous Delivery, with more details on the concept of deployment pipelines. Their book, Continuous Delivery, rightly won the Jolt excellence award in 2011.

A simple example of this is a two stage deployment pipeline. The first stage would do the compilation and run tests that are more localized unit tests with the database completely stubbed out. Such tests can run very fast, keeping within the ten minute guideline. However any bugs that involve larger scale interactions, particularly those involving the real database, won't be found. The second stage build runs a different suite of tests that do hit the real database and involve more end-to-end behavior. This suite might take a couple of hours to run.

In this scenario people use the first stage as the commit build and use this as their main CI cycle. The second-stage build runs when it can, picking up the executable from the latest good commit build for further testing. If this secondary build fails, then this may not have the same 'stop everything' quality, but the team does aim to fix such bugs as rapidly as possible, while keeping the commit build running. As in this example, later builds are often pure tests since these days it's usually tests that cause the slowness.

If the secondary build detects a bug, that's a sign that the commit build could do with another test. As much as possible you want to ensure that any later-stage failure leads to new tests in the commit build that would have caught the bug, so the bug stays fixed in the commit build. This way the commit tests are strengthened whenever something gets past them. There are cases where there's no way to build a fast-running test that exposes the bug, so you may decide to only test for that condition in the secondary build. Most of time, fortunately, you can add suitable tests to the commit build.

This example is of a two-stage pipeline, but the basic principle can be extended to any number of later stages. The builds after the commit build can also be done in parallel, so if you have two hours of secondary tests you can improve responsiveness by having two machines that run half the tests each. By using parallel secondary builds like this you can introduce all sorts of further automated testing, including performance testing, into the regular build process.

Test in a Clone of the Production Environment

The point of testing is to flush out, under controlled conditions, any problem that the system will have in production. A significant part of this is the environment within which the production system will run. If you test in a different environment, every difference results in a risk that what happens under test won't happen in production.

As a result you want to set up your test environment to be as exact a mimic of your production environment as possible. Use the same database software, with the same versions, use the same version of operating system. Put all the appropriate libraries that are in the production environment into the test environment, even if the system doesn't actually use them. Use the same IP addresses and ports, run it on the same hardware.

Well, in reality there are limits. If you're writing desktop software it's not practicable to test in a clone of every possible desktop with all the third party software that different people are running. Similarly some production environments may be prohibitively expensive to duplicate (although I've often come across false economies by not duplicating moderately expensive environments). Despite these limits your goal should still be to duplicate the production environment as much as you can, and to understand the risks you are accepting for every difference between test and production.

If you have a pretty simple setup without many awkward communications, you may be able to run your commit build in a mimicked environment. Often, however, you need to use test doubles because systems respond slowly or intermittently. As a result it's common to have a very artificial environment for the commit tests for speed, and use a production clone for secondary testing.

I've noticed a growing interest in using virtualization to make it easy to put together test environments. Virtualized machines can be saved with all the necessary elements baked into the virtualization. It's then relatively straightforward to install the latest build and run tests. Furthermore this can allow you to run multiple tests on one machine, or simulate multiple machines in a network on a single machine. As the performance penalty of virtualization decreases, this option makes more and more sense.

Make it Easy for Anyone to Get the Latest Executable

One of the most difficult parts of software development is making sure that you build

the right software. We've found that it's very hard to specify what you want in advance and be correct; people find it much easier to see something that's not quite right and say how it needs to be changed. Agile development processes explicitly expect and take advantage of this part of human behavior.

To help make this work, anyone involved with a software project should be able to get the latest executable and be able to run it: for demonstrations, exploratory testing, or just to see what changed this week.

Doing this is pretty straightforward: make sure there's a well known place where people can find the latest executable. It may be useful to put several executables in such a store. For the very latest you should put the latest executable to pass the commit tests - such an executable should be pretty stable providing the commit suite is reasonably strong.

If you are following a process with well defined iterations, it's usually wise to also put the end of iteration builds there too. Demonstrations, in particular, need software whose features are familiar, so then it's usually worth sacrificing the very latest for something that the demonstrator knows how to operate.

Everyone can see what's happening

Continuous Integration is all about communication, so you want to ensure that everyone can easily see the state of the system and the changes that have been made to it.

One of the most important things to communicate is the state of the mainline build. If you're using Cruise there's a built in web site that will show you if there's a build in progress and what was the state of the last mainline build. Many teams like to make this even more apparent by hooking up a continuous display to the build system - lights that glow green when the build works, or red if it fails are popular. A particularly common touch is red and green lava lamps - not just do these indicate the state of the build, but also how long it's been in that state. Bubbles on a red lamp indicate the build's been broken for too long. Each team makes its own choices on these build sensors - it's good to be playful with your choice (recently I saw someone experimenting with a dancing rabbit.)

If you're using a manual CI process, this visibility is still essential. The monitor of the physical build machine can show the status of the mainline build. Often you have a build token to put on the desk of whoever's currently doing the build (again something silly like a rubber chicken is a good choice). Often people like to make a simple noise on good builds, like ringing a bell.

CI servers' web pages can carry more information than this, of course. Cruise provides an indication not just of who is building, but what changes they made. Cruise also provides a history of changes, allowing team members to get a good sense of recent activity on the project. I know team leads who like to use this to get a sense of what people have been doing and keep a sense of the changes to the system.

Another advantage of using a web site is that those that are not co-located can get a sense of the project's status. In general I prefer to have everyone actively working on a project sitting together, but often there are peripheral people who like to keep an eye on things. It's also useful for groups to aggregate together build information from multiple projects - providing a simple and automated status of different projects.

Good information displays are not only those on a computer screens. One of my favorite displays was for a project that was getting into CI. It had a long history of

being unable to make stable builds. We put a calendar on the wall that showed a full year with a small square for each day. Every day the QA group would put a green sticker on the day if they had received one stable build that passed the commit tests, otherwise a red square. Over time the calendar revealed the state of the build process showing a steady improvement until green squares were so common that the calendar disappeared - its purpose fulfilled.

Automate Deployment

To do Continuous Integration you need multiple environments, one to run commit tests, one or more to run secondary tests. Since you are moving executables between these environments multiple times a day, you'll want to do this automatically. So it's important to have scripts that will allow you to deploy the application into any environment easily.

A natural consequence of this is that you should also have scripts that allow you to deploy into production with similar ease. You may not be deploying into production every day (although I've run into projects that do), but automatic deployment helps both speed up the process and reduce errors. It's also a cheap option since it just uses the same capabilities that you use to deploy into test environments.

If you deploy into production one extra automated capability you should consider is automated rollback. Bad things do happen from time to time, and if smelly brown substances hit rotating metal, it's good to be able to quickly go back to the last known good state. Being able to automatically revert also reduces a lot of the tension of deployment, encouraging people to deploy more frequently and thus get new features out to users quickly. (The Ruby on Rails community developed a tool called Capistrano that is a good example of a tool that does this sort of thing.)

Many people are concerned about how to deal with databases with frequent releases. Pramod Sadalage and I wrote this article explaining how handle this with automated refactoring and migration of databases.

In clustered environments I've seen rolling deployments where the new software is deployed to one node at a time, gradually replacing the application over the course of a few hours.

A particularly interesting variation of this that I've come across with public web application is the idea of deploying a trial build to a subset of users. The team then sees how the trial build is used before deciding whether to deploy it to the full user population. This allows you to test out new features and user-interfaces before committing to a final choice. Automated deployment, tied into good CI discipline, is essential to making this work.

Benefits of Continuous Integration

On the whole I think the greatest and most wide ranging benefit of Continuous Integration is reduced risk. My mind still floats back to that early software project I mentioned in my first paragraph. There they were at the end (they hoped) of a long project, yet with no real idea of how long it would be before they were done.

The trouble with deferred integration is that it's very hard to predict how long it will take to do, and worse it's very hard to see how far you are through the process. The result is that you are putting yourself into a complete blind spot right at one of tensest parts of a project - even if you're one of the rare cases where you aren't already late.

Continuous Integration completely finesses this problem. There's no long integration, you completely eliminate the blind spot. At all times you know where you are, what works, what doesn't, the outstanding bugs you have in your system.

Bugs - these are the nasty things that destroy confidence and mess up schedules and reputations. Bugs in deployed software make users angry with you. Bugs in work in progress get in your way, making it harder to get the rest of the software working correctly.

Continuous Integrations doesn't get rid of bugs, but it does make them dramatically easier to find and remove. In this respect it's rather like self-testing code. If you introduce a bug and detect it quickly it's far easier to get rid of. Since you've only changed a small bit of the system, you don't have far to look. Since that bit of the system is the bit you just worked with, it's fresh in your memory - again making it easier to find the bug. You can also use diff debugging - comparing the current version of the system to an earlier one that didn't have the bug.

Bugs are also cumulative. The more bugs you have, the harder it is to remove each one. This is partly because you get bug interactions, where failures show as the result of multiple faults - making each fault harder to find. It's also psychological - people have less energy to find and get rid of bugs when there are many of them - a phenomenon that the Pragmatic Programmers call the Broken Windows syndrome.

As a result projects with Continuous Integration tend to have dramatically less bugs, both in production and in process. However I should stress that the degree of this benefit is directly tied to how good your test suite is. You should find that it's not too difficult to build a test suite that makes a noticeable difference. Usually, however, it takes a while before a team really gets to the low level of bugs that they have the potential to reach. Getting there means constantly working on and improving your tests.

If you have continuous integration, it removes one of the biggest barriers to frequent deployment. Frequent deployment is valuable because it allows your users to get new features more rapidly, to give more rapid feedback on those features, and generally become more collaborative in the development cycle. This helps break down the barriers between customers and development - barriers which I believe are the biggest barriers to successful software development.

Introducing Continuous Integration

So you fancy trying out Continuous Integration - where do you start? The full set of practices I outlined above give you the full benefits - but you don't need to start with all of them.

There's no fixed recipe here - much depends on the nature of your setup and team. But here are a few things that we've learned to get things going.

One of the first steps is to get the build automated. Get everything you need into source control get it so that you can build the whole system with a single command. For many projects this is not a minor undertaking - yet it's essential for any of the other things to work. Initially you may only do build occasionally on demand, or just do an automated nightly build. While these aren't continuous integration an automated nightly build is a fine step on the way.

Introduce some automated testing into your build. Try to identify the major areas

where things go wrong and get automated tests to expose those failures. Particularly on an existing project it's hard to get a really good suite of tests going rapidly - it takes time to build tests up. You have to start somewhere though - all those cliches about Rome's build schedule apply.

Try to speed up the commit build. Continuous Integration on a build of a few hours is better than nothing, but getting down to that magic ten minute number is much better. This usually requires some pretty serious surgery on your code base to do as you break dependencies on slow parts of the system.

If you are starting a new project, begin with Continuous Integration from the beginning. Keep an eye on build times and take action as soon as you start going slower than the ten minute rule. By acting quickly you'll make the necessary restructurings before the code base gets so big that it becomes a major pain.

Above all get some help. Find someone who has done Continuous Integration before to help you. Like any new technique it's hard to introduce it when you don't know what the final result looks like. It may cost money to get a mentor, but you'll also pay in lost time and productivity if you don't do it. (Disclaimer / Advert - yes we at ThoughtWorks do some consultancy in this area. After all we've made most of the mistakes that there are to make.)

Final Thoughts

In the years since Matt and I wrote the original paper on this site, Continuous Integration has become a mainstream technique for software development. Hardly any ThoughtWorks projects goes without it - and we see others using CI all over the world. I've hardly ever heard negative things about the approach - unlike some of the more controversial Extreme Programming practices.

If you're not using Continuous Integration I strongly urge you give it a try. If you are, maybe there are some ideas in this article that can help you do it more effectively. We've learned a lot about Continuous Integration in the last few years, I hope there's still more to learn and improve.

Further Reading

An essay like this can only cover so much ground, but this is an important topic so I've created a guide page on my website to point you to more information.

To explore Continuous Integration in more detail I suggest taking a look at Paul Duvall's appropriately titled book on the subject (which won a Jolt award - more than I've ever managed). For more on the broader process of Continuous Delivery, take a look at Jez Humble and Dave Farley's book - which also beat me to a Jolt award.

You can also find more information on Continuous Integration on the ThoughtWorks site.

For articles on similar topics...

...take a look at the following tags:

popular agile delivery extreme programming

continuous integration

Acknowledgments

First and foremost to Kent Beck and my many colleagues on the Chrysler Comprehensive Compensation (C3) project. This was my first chance to see Continuous Integration in action with a meaningful amount of unit tests. It showed me what was possible and gave me an inspiration that led me for many years.

Thanks to Matt Foemmel, Dave Rice, and everyone else who built and maintained Continuous Integration on Atlas. That project was a sign of CI on a larger scale and showed the benefits it made to an existing project.

Paul Julius, Jason Yip, Owen Rodgers, Mike Roberts and many other open source contributors have participated in building some variant of CruiseControl. Although a CI server isn't essential, most teams find it helpful. CruiseControl and other CI servers have played a big part in popularizing and enabling software developers to use Continuous Integration.

One of the reasons I work at ThoughtWorks is to get good access to practical projects done by talented people. Nearly every project I've visited has given tasty morsels of continuous integration information.

Significant Revisions

01 May 2006: Complete rewrite of article to bring it up to date and to clarify the description of the approach.

10 September 2000: Original version published.

Guides

- Intro
- Design
- Agile
- NoSQL
- DSL
- Delivery
- About Me

Popular Articles

- New Methodology
- Dependency Injection
- Mocks aren't Stubs
- Is Design Dead?
- Continuous Integration

Books

- NoSQL Distilled
- Domain-Specific Languages
- Refactoring
- Patterns of Enterprise Application Architecture
- UML Distilled
- Analysis Patterns
- Planning Extreme Programming
- Signature Series

Site Sections

- FAQ
- Content Index
- Bliki
- Books
- DSL Catalog
- EAA Catalog
- EAA Dev
- Photos

ThoughtWorks

- Blogs
- Careers
- Mingle
- Twist
- Go