Benefits of the Build

- Kirk Knoernschild
- www.qwantify.com

Original article as published in March 2005 issue of Software Development Magazine

Introduction

What's the single most important activity your team can perform to get your project moving along? An automated and repeatable build! Why? Because an automated build produces the only artifact that really matters to everyone associated with the project...the executable application. Certainly there are other valuable artifacts that a team may produce, and possibly even find useful. But all other pretty documents will be quickly forgotten if you don't deliver the final product.

Defining the Build

Embodied in XP's Continuous Integration practice, an automated and repeatable build can work wonders for your development team. First and foremost, an automated and repeatable build forces you to integrate early and integrate often. You're guaranteed to always have a system that works. Of course, there are a couple of rules you have to follow. First, any compile error that ever creeps in must be immediately resolved. Since you should be using a version control system, such as CVS, this means that the CVS projects comprising your application should always be free of any compilation errors. Second, any unit test that fails must be immediately fixed. Unit tests should be run after each change you make, and if you ever find that a unit test fails, you should treat this failure with the same urgency you'd give a compile error. While pretty simple, these two rules are key elements because they are part of what defines an automated and repeatable build. So here are the steps, in order, defining the steps of an automated and repeatable build [Fowler].

- It must be a clean compile, meaning that no compiler generated syntactical errors occur. While it's acceptable to encounter certain warnings, such as those generated by deprecated method usage, you should strive to remove these warnings before future builds.
- The build should be done using the most current version of all source files. There are occasions where you'll want to compile older versions of code to revert back to a previous version. Typically, when compiling these older versions you'll use the source files consistent with the version of the application you're building.

- A clean build implies that all application source files are compiled. Conditional compiles or partial builds do not prove syntactical correctness of the entire application.
- The build should generate all files representing the units of deployment. For Java applications, this means all .jar, .war, and .ear files should be created.
- The build should verify that the application is functionally correct, meaning all unit tests for the application should be run. If all unit tests pass successfully, the build is successful.
- Upon completing a successful build, it's good practice to deploy the build to a development region so that it can be further verified by developers, and reviewed by any interested stakeholders beyond the development team.

Frequency of the Build

But how often should your team perform the build? The frequency of builds is really a product of your development team and environment. I've heard some say that the build should be done multiple times daily, which is the ideal. However, I've found that once a day proves pretty successful too. And if you decide to only build a couple times a week, that's fine as well. But don't go much longer than a few days since it's likely errors will creep in (ie. compile errors or junit failures), and you'll wind up devoting a larger part of your effort correcting bugs over growing the system.

Keep in mind that if your build ever fails due to a compilation error or a failed test case, it must become your team's immediate priority to fix that problem before adding any new functionality. So building multiple times daily is ideal since you know there won't be long periods of time between builds that allow more than just a few bugs to surface. You may be at the mercy of developers who choose to work extended periods of time without releasing their changes to a version control system. It's best if you can establish ground rules advocating that developers release early and often, preferably multiple times per day. Enforcing the rule that failed builds must become the immediate focus of the entire team establishes a peer review system that usually works well. I don't want my changes to serve as the reason why the team must quit working to correct any errors.

Build Promotes Agility

Aside from continuously proving integration though, an automated build facilitates a number of other very useful practices.

- You can perform it anytime you want. Ideally, you should schedule the build to run at certain times throughout the day or each time code is checked into a version control system. With some experimentation, you'll find what works best for your development team. Keep in mind though, that since the build is repeatable, you can also run it on demand.
- You verify consistency. Because it's automated and repeatable using a build tool such as Ant (http://ant.apache.org), you've assured yourself it will be run the same

way each time. Without an automated build, you cannot guarantee that a developer won't forget to do something important. You should rarely consider using a development tool's export utility to generate your deployable unit. Too much manual intervention is typically required, leaving the door open for problems.

- You ensure there are no compile errors. Even though most project teams are under incredible pressure to deliver, it doesn't do help anyone to pile code that doesn't work on top of code that doesn't work. It's ironic though, that I've seen this happen on numerous occasions. An automated and repeatable build adhering to the rules above ensures this won't happen.
- You verify your code's functional correctness. A key component to any automated and repeatable build is that your unit tests are run immediately after the source has been compiled successfully. Most developers learned long ago that just because something compiles, doesn't mean it works. Unfortunately, while we learned it, we don't always practice it.
- You prove your physical structure. There are different ways to perform the build. You can build the entire application with all source code in the classpath, and that's pretty useful. But you can also perform a levelized build, whereby those physical units with no dependencies are built first, followed by those components that have dependencies on previously compiled components. A levelized build is valuable since it can enforce the dependency structure of your application components. If an undesirable dependency creeps in, the build will fail.
- You can drive other important lifecycle activities. Frequent customer feedback is very necessary to ensure you give customers what they need, instead of what they ask for. If you have a system that works, it's much easier to get early and frequent feedback. A great way to garner feedback from your customers is to show them the system, and allow them to experiment by simulating their work. Frequent system demos involving customers and developers are a great way to facilitate discussion and iron out any wrinkles associated with especially tricky or complex areas of the applications. Full system demos help everyone stay in tune with the overall vision of the project, especially as some folks become detached as they drown themselves in the details of a specific use case or subsystem.
- You can use it to get objective feedback. Build tools, such as Ant, often times have tasks that allow you to analyze your code base. For instance, Jdepend (http://www.clarkware.com/software/JDepend.html) and JavaNCSS (http://www.kclee.de/clemens/java/javancss) can give you feedback that allows you to somewhat objectively analyze the quality of your source code.
- You can use it to review your code. Other build tools, such as PMD (http://pmd.sourceforge.net), can eliminate those mundane and often useless code reviews. Tools like this can scan your source code and identify potential problems, or violation of rules that you define.

- You never really change how you work. Hopefully, you'll eventually release your application in a production environment. Once you do, continue to perform the build just as you always did. This makes maintenance much easier since you don't have to radically redefine how you're doing things. In fact, every line of code you write after the first is maintenance. So don't treat maintenance as a phase of a macro lifecycle. Treat maintenance as a way to grow your application, and continually maintain the application's behavior.
- You prove the full lifecycle. You develop, test, integrate, and deploy every time you build, helping you avoid the big bang. That is, the bang you hear when your application explodes as you try to move it from a local development environment to a shared server, or attempt to integrate with the work of another developer after not doing so for a long period of time. Instead, integration is continuous and performed each time you execute the build.
- It's a grass roots approach to agility. The beauty of an automated build is that nobody really cares much about it. So it's doubtful you'll encounter much resistance from management or process purists along the way. In fact, everyone knows that at some point you have to compile and deploy the application, so nobody views it as a threat to the old school approach to software development. I've not yet run across a situation whereby someone balked at the thought of creating build scripts early in the project. But once the build is working, it's hard to deny the benefits associated with each of the points above. And all of the above points are steps toward a more agile process where you don't wait until a few weeks before the ship date to integrate or roll the application out for testing. Instead, you roll the application out early and often.

Conclusion

An automated and repeatable build is a valuable asset for any development team. It enables a number of other useful lifecycle activities. Instead of wasting valuable time debating the merits of XP, RUP, or your other favorite process, devote your efforts to creating and refining your automated and repeatable build process. Then use the build to perform other activities that make sense for your project, regardless of which process you gleaned those activities.

Bibliography

[Fowler] Fowler, Martin and Foemmel, Matthew. *Continuous Integration.* http://www.martinfowler.com/articles/continuousIntegration.html