JOIN OVER 100,000 PROS! Join Log In | Member Benefits

STICKYMINDS

Agile Testing DevOps Security AI

Building For Success ARTICLE

By Steve Berczuk , Brad Appleton Robert Cowham | December 5, 2005

SUMMARY

This article addresses detailed scenarios and the associated problems with the build process. We look at the usually activities and work patterns of a developer and relate those to build practices.

Parallel Development and Codeline Merging

Fred works in his Private Workspace and his team is using the Mainline pattern of development. Having created the workspace based on the mainline code from the repository, his standard pattern of working:

1. Implement desired bug fix or new feature by editing, adding or deleting files
2. Build locally (Private System Build)
3. Test locally (Unit Tests)
4. Commit globally (Task-Level Commit)—if it works, then check-in (commit) changes to the repository, otherwise make some more changes

Any changes made need to be tested before being released (surely you've never made a change that was "so small and obvious it doesn't need testing?!..."), and you can't test them without building them. If the build fails, then you need to fix it first. Only if the test succeeds does the commit happen.

What the above doesn't take into account, is what happens when other team members are also making changes. Fred's lifecycle needs to include the extra step of merging in other people's changes with his own.

Fred performs a Catchup or Rebase. The problem, as shown by Figure 2—Include Changes From Others, is that this extra step brings with it the need for a sub-cycle of Reconcile/Rebuild/Retest/Resolve. This introduces an extra element of risk since Fred's code is currently only in his workspace and has not yet been checked in or committed. Possible resolutions [1] include Private Archive, Private Branch and Task Branch.

We use cookies to ensure you get the best experience on our website.

Accept

Read More

commit—something that is saved into the repository but not visible to other users until desired which is the equivalent of a Task Branch). He normally performs a build and test before the commit, but on occasion can still commit something that only partially works since no one else will see it. Fred makes his changes visible to the rest of the project by "publishing" them, or merging them back to the Mainline. The key thing here is that just before publishing, he does a Catchup to bring in other team members changes. This does the more risky merge in his Task Branch which should make the Publish a very simple operation.

To Branch Per Task/Privately or Not?

Using Task Branches isolates the risk but at the cost of extra work required to perform the catchup/publish. Interestingly, working directly on the Mainline can be fine if the risk of developers changing the same modules is low, and it is surprising how often this is the case. Indeed, "collisions" are often restricted to a small subset of modules in the system, and judicious refactoring of these can reduce collisions dramatically. That said, there are often some files that are widely used and shared and frequently updated such as makefiles or some global system definitions.

Team Velocity

In "The Illusion of Control" [2] we talked about how introducing checks to ensure that builds aren't broken and that bad changes aren't released into the code base can end by decreasing the velocity of the team. This contrasts with the Continuous Updates pattern described in "Codeline Merging and Locking" [1] where what appears to be more work turns out to give greater velocity.

Checks are enforced by:

- Build (Private System Build and Integration Builds)
- Smoke Tests

Each team has to work out the appropriateness of the checks for each stage. Let's look first at some considerations for improving build velocity.

Greater Build Velocity

A build needs to be reliable, repeatable and consistent, all of which engender trust in the result. As we discussed last month, manual build processes tend not to have these properties—all it takes is a loss of concentration at a crucial moment, or something dodgy in the environment, and the build produced is not reliable.

Thus we would always recommend a "one button build"—an automatic way of doing the whole thing by executing a single command. This is nice and simple in some environments, particularly where a clean build from scratch takes a small amount of time, e.g. in the order of minutes. As soon as builds start taking tens of minutes or hours,

We use cookies to ensure you get the best experience on our website.

cause problems. One of the fundamental practices of agile development (and SCM) is that fast and frequent feedback gives major benefits. Problems swept under the carpet that surface later can be exponentially more difficult to deal with at that time.

So what other factors affect build velocity and what can we do to improve it?

- Fast machines!
- Shared build servers
- Incremental builds and different build tools
- IDEs
- Share library problems

Fast Machines/Build Servers
It is amazing how much difference a fast machine can make to a build. By their nature, builds are compute intensive tasks (though can also be I/O intensive for things like dependency checking) and a PC suitable for standard office work will have developers twiddling thumbs and drowning in coffee.

Like any system, decisions to not provide fast machines are very seldom taken "with malice aforethought" to purposely make developer's lives difficult. It is usually lack of knowledge and not looking at the whole process, leading to unintentional side effects.

The trend for outsourcing is often a source of problems. What used to be a simple request to upgrade a machine or similar and could be solved by someone from the IT department coming round in an hour or two, may require raising a request to some remote support centre and some huge rigmarole of a process which requires incantations and the blood of your first born! A recent experience of one of the authors was where the need to delete some corrupted files from a proxy server, the work of a few minutes, was estimated to take 2 weeks to wend its merry way through the system. Meanwhile, 20 developers had to work around the problem!

At the very least, a shared (and fast!) build server which can turn around builds quickly can be a great boon. This concept of course can be extended to include build farms for parallel builds—for which various commercial and open source solutions exist—well worth investigating. In itself a potentially deep topic which we will look at in the future (or maybe covered elsewhere in CMCrossroads).

Incremental Builds
As mentioned above, the gold standard build is a clean build from scratch. For complete system builds, it is always going to be the preferred solution.

However, if you have just changed a few modules since the last build and

We use cookies to ensure you get the best experience on our website.

thing, the incremental build is always going to look very attractive. If done well, it is an excellent tool to have in your team's repertoire.

The key to is having accurate dependency checking in your build system.

The art of writing build scripts and using tools to automate this is something the most developers are not particularly interested in and indeed can't work out how to do properly. It is well worth someone investing the time to learn the intricacies of your particular build tool and learning how to write this, or alternatively to bring in an experience external consultant. The granddaddy of build tools is Make and for many years it pretty much had the field to itself. Some pretty arcane syntax and the dreaded tab/space problem have caused much tearing of hair over the years. Enhanced versions such as Gnu Make are quite a bit nicer to use and there are some excellent articles showing the power of this by John Graham-Cumming at CMCrossroads.

Other alternatives that have surfaced over the years include Jam, Scons and more recently the XML based Ant and its .Net equivalent Nant. Vendor specific solutions include OpenMake, ClearCase Make with its wink-in technology to avoid rebuilding what someone else has recently done, and Microsoft's upcoming MSBuild—another XML based tool. There are a number of discussions as to the suitability of XML as a human-readable description of such systems, and it will be interesting to see how this area develops in the future.

Incremental Build Case Study

One of the authors implemented a system based around Jam which has worked very well since first being implemented 8 years ago. While not exactly being an agile environment, the principles apply!

The system consisted of over a hundred executables that were running on a VAX/VMS system. Subsequently this was migrated to Alpha/VMS and then flavours of Unix. The basic build scripts (which the developers needed to maintain), were very simple and just listed source files and libraries for each executable. These changed very little during the migrations between operating systems. The nitty gritty rules as to how to build files and do dependency checking (including operating system specifics) were maintained by 1 or 2 people—others didn't need to know.

Prior to this system, a hard coded set of batch files were used to build things. This included no dependency checking at all—for incremental builds people just need to know what they wanted and ensure that everything was built in the appropriate order (the build worked OK for the full system builds, though changes in dependency ordering caused

We use cookies to ensure you get the best experience on our website.

Since developers typically worked on 1 or 2 executables at a time, with a few source files, they hacked up batch scripts just to build the individual executables. The process was time consuming and error prone, particularly the first time you wanted to build one you hadn't done before.

The Jam based system did full automatic dependency checking and used the same scripts that the full system build used in order to build individual executables, just supplying the name of the required executable as a command line parameter. The trick was to use the equivalent of search paths so that any local source code was used if present, otherwise a central shared read-only snapshot was used (so that people didn't have to have the full source code present in their workspaces if they didn't need it). The advantage of this was that there was only one place that people needed to maintain the contents of individual executables—the system build script. Various libraries were constructed as necessary and linked in to the appropriate executable.

In a new (clean) workspace, the first time you tried to build an executable all the appropriate libraries would be built and stored locally and then linked in. This build of libraries took a little time, but typically only happened once. Subsequent incremental builds picked up the libraries, checked their dependencies just in case, but since they hadn't changed didn't rebuild them. Thus incremental builds typically took only a few minutes.

The benefits of automatic dependency checking and ensuring that the correct software was built and tested and no build related problems surfaced subsequently were immense.

IDEs

These days, a lot of development is done using Integrated Development Environments (IDEs) ranging from Eclipse to Visual Studio in its various forms.

IDEs have many benefits in terms of time saving conveniences for developers. The ability to check stuff out and back in can be a great boon. Another classic feature is the ability to compile and build within the IDE, showing errors and a double-click taking you to the line where a compile error occurred. Also the ability to debug easily, set break points, etc. All these are very useful features and make developers much more effective.

There have been more than a few problems in the past when vendors have created a wonderful IDE but haven't thought emough about interfaces to other tools, and in particular have provided very limited automation possibilities. For example, the information which drives the build process may be stored in proprietary format rather

We use cookies to ensure you get the best experience on our website.

Now it is very convenient and saves a lot of time when developing to be able to hit a button and incrementally rebuild the application. How do you repeat the build for the integration or system build? You want to use the same compiler and linker as your developers. It is very seldom that the compiler and linker or equivalents used by the IDE are not callable from the command line or a build script. The problem comes when the IDE wraps up options that are not easily replicated from a build script, or which require maintenance in 2 places—once within the IDE and once in the global build script. Fortunately, most IDEs support being called from the command line so that it is possible, although sometimes more work than should perhaps be the case, to wrap it up.

Installers

Any system that is going to be deployed needs to be installable. Particularly on Windows this has caused some major issues. In a similar manner to IDEs mentioned above, installers have not always been good citizens in terms of providing automation facilities so that an external build tool can build the complete installer with no manual steps. In our opinion this is a vital feature, and would automatically exclude an installer from the shortlist if not appropriately supported.

Shared Library Problems

The architectural decision to popularise component based development via DLLs (Dynamic Link Libraries—the equivalent concept exists on Unix and other operating systems) had good intentions but turned out to be flawed. It's nice to think that if many applications share a single DLL and there is a bug in it, that a single upgrade fixes problems in all of the applications. The reality was that an upgrade of a DLL often broke or destabilised existing applications due to lack of backwards compatibility or just not being tested properly. People started programming defensively and putting in start-up checks to ensure that only compatible versions of DLLs were acceptable to their application.

This has been a particular problem on Windows. Some of the reasons for DLLs were:

- Saved disk space (now irrelevant)
- A single upgrade fixed many components (but no testing and masses of incompatibilities in practice)
- COM model—Single point of registration in the registry (and single point of failure)

Having recognised the problems that this caused, Microsoft introduced .Net with its xcopy deployment promise. They have also changed their tune with regards to use of the registry, now recommending that applications store user settings in files under the

We use cookies to ensure you get the best experience on our website.

Of course .Net doesn't solve all problems, and there are indeed several different versions of the .Net runtime available which users need to have installed on their machines, or get them to download. As a result, COM is still the most widely used technology for mass market applications.

The problem with COM is that you need to register components in order to be able to use them. If you have multiple copies of a COM DLL on your system then the last registered one is the one that will be used. This can cause major problems when developing and testing, and then subsequently installing—are you sure you are using the right one? (Hint—process explorer from www.sysinternals.com will guarantee to tell you which process has which DLL loaded—and it's free!).

It can also cause major problems when building an application. If there are dependencies between COM DLLs in your application it is obviously vital to build them in the correct order. If you miss building (and registering) a DLL then you may pick up some dodgy previous version that happened to be lying around on your machine. Many Windows developments had major problems in this area.

Windows Case Study

Interestingly the same product referred to in the previous case study had a Windows GUI implemented via Visual Basic and C++ COM DLLs which needed to be built in a particular order. Also, an installer needed to be created.

The procedure to produce a complete build of the system was quite well documented but ran to 30+ steps! This involved things like opening a project in Visual Basic and resetting the references (to other DLLs) before building. Results then needed to be manually copied to a different directory structure and the installer IDE used to create the installer. All told, a build took nearly 2 hours, and there was one person whose chief job it was to do this! If they were away or busy, the whole team had problems.

In contrast one of the authors automated a different system—rather smaller but also including Visual Basic and C++ DLLs and an installer. This was done using Scons, which since it is written in Python is very easy to customise if necessary. Dependencies were automatically extracted from project files and DLLs were built using the IDEs with command line parameters.

Conclusion

Since builds are so fundamental to the development of software, it is vital that they be done well. Often, all it requires is the knowledge that such things are desirable and indeed vital—the implementation is not usually that difficult. The sad thing is how many

We use cookies to ensure you get the best experience on our website.

The advantage of agile methods is that a good build process is recognised for the fundamental enabler that it is, and thus a corner stone for all agile development.

References

[1] **Codeline Merging and Locking: Continuous Updates and Two-Phased Commits,** Configuration Management Journal, Nov 2003

[2] **The Illusion of Control –** Configuration Management Journal, Nov 2003

[3] The **Agile Difference for SCM –** Configuration Management Journal, Oct 2004

Topics: change management, configuration management,

We use cookies to ensure you get the best experience on our website.

ABOUT THE AUTHOR

Steve Berczuk

Steve Berczuk is a Principal Software Engineer with experience as a manager, Scrum Master and technical lead in Boston, MA. The author of *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*, he is a recognized expert in software configuration management and agile software development. Steve is passionate about helping teams work effectively to produce quality software. He has an M.S. in operations research from Stanford University and an S.B. in Electrical Engineering from MIT, and is a Certified ScrumMaster. Contact Steve at steve@berczuk.com or visit berczuk.com and follow his blog at blog.berczuk.com.

Brad Appleton

Brad Appleton is an Enterprise Agile+DevOps leader, coach & manager, and seasoned DevOps/ALM/CM solution architect at a large Fortune 100 company. Currently he helps organizations and teams scale, adopt and apply lean/agile development methods and DevOps/ALM/CM practices and tools. He is co-author of *Software Configuration Management Patterns*, the Streamed-Lines branching patterns, and a former section editor for *The C++ Report*. You can read Brad's blog at blog.bradapp.net.

Robert Cowham

Robert Cowham has long been interested in software configuration management while retaining the attitude of a generalist with experience and skills in many aspects of software development. A regular presenter at conferences, he authored the Agile SCM column within the CM Journal together with Brad Appleton and Steve Berczuk. His day job is as Services Director for Square Mile Systems whose main focus is on skills and techniques for infrastructure configuration management and DCIM (Data Center Infrastructure Management) - applying configuration management principles to hardware documentation and implementation as well as mapping ITIL services to the underlying layers.

UPCOMING WEBINAR

We use cookies to ensure you get the best experience on our website.

LETS HANG!

FEATURED RESOURCES

SECURITY

Shifting Security Left: DevSecOps eGuide

White Paper | StickyMinds

Log In or Register to post comments

USER COMMENTS

0 comments

YOU MAY ALSO LIKE

SECURITY

Agentic AI Opens Up New Security Flaws

Nahla Davies | August 27, 2025

We use cookies to ensure you get the best experience on our website.

SECURITY

Countering the Threat of Ransomware-as-a-Service (RaaS)

Nahla Davies | August 22, 2025

SECURITY

Building Trust in Blockchain: The Role of Smart Contract Testing

Ekaterina Egorova | November 18, 2024

Join the crew for FREE! Access exclusive content, contribute, and more.

See All Member Benefits

We use cookies to ensure you get the best experience on our website.