Beautiful Builds

Growing Readable, Maintainable Automated Build Processes

ROY OSHEROVE

Beautiful Builds

Patterns for Growing Readable, Maintainable Automated Build Processes

Roy Osherove

This book is for sale at http://leanpub.com/build

This version was published on 2013-02-07

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Roy Osherove by spreading the word about this book on Twitter!

The suggested tweet for this book is:

AUTOMATE ALL THE THINGS. Just got #beautifulbuilds book. http://beautifulbuilds.com https://leanpub.com/build

The suggested hashtag for this book is #beautifulbuilds.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#beautifulbuilds

Also By Roy Osherove

Notes to a Software Team Leader

Contents

About the Author

Roy Osherove is the author of “The Art Of Unit Testing” and “Notes to a Software Team Leader”, and has been in leadership roles for most of his professional life, acting as team lead, CTO and architect in many places.

Roy currently works as the chief sceitntist at Bouvet.no, where is regularly is available for consulting and training.

He’s had many failures to learn from but also some great successes, that he likes to share by doing training courses and mentoring. You can read his blog at 5whys.com and osherove.com You can reach roy at roy@osherove.com

1 Basic Terms

- A Build Script is a file that contains script that runs automated actions. This file sits in your source control. It is executed by Continuous Integration systems as part of their Build Configurations.
- A Build Configuration (BC) is part of a Continuous Integration System. A build Configuration has a name that makes it unique (such as “nightly build”), and is composed of two basic elements: Build Configuration Context, and Build Configuration Actions.
- A BC Context is the combination of any physical files on the system, or from source control, or from artifacts of other build Configurations, and user entered information, such as environment variables to set, when to run (for example, after the successful finish of a different build configuration), and what artifacts to publish.
- BC Actions are the set of actions to perform with the given BC Context. For example, execute the build script file that is part of the current source control snapshot.
- BC Artifacts: A set of files published by a specific run of a build Configuration, and saved for later use by other build Configurations.

2 Pattern: Build Script Injection

Other Names:

Inversion of Build Control, Self-Builder, Versionable-Build-Script

2.1 Problem:

You have a continuous integration process running nicely, but sometimes you need to be able to build older versions of your product. Unfortunately, in older versions of your product in source control, the structure of files is different than the one that your build actions are set up to use. For example, a set of directories that exists in the latest version in source control, and is used in deployment, does not exist in the source control version from 3 months ago. So your build fails because it expects certain files or directories to be there, which did not exist in that source control version.

2.2 Forces:

You want the set of automated actions in your build to match exactly the current version and structure of your product files in source control.

2.3 Solution:

Separate your build script actions into two parts:

- The CI side script, which lives in the continuous integration system.
- The source control side script.

Source control side script

one or more script files that are inside the file structure of your product in source control. These scripts change based on the current product version. It is important that this is the same branch that is used within the CI system to build the product, so that the CI side scripts have access to the source control side scripts.

Developers should have full access to the source control side scripts.

Source Control side scripts contain the knowledge about the current structure of the files, and which actions are relevant for the current product version. So they get to be changed with every product version. They will usually use relative path because they will be executed by CI side scripts on a remote build server.

CI Side Scripts/Actions

These actions act as very simple “dumb” agents. They get the latest version of the source control scripts (and possibly all other product files if needed), and trigger the build scripts as a command line action.

2.4 Summary

By separating the version aware knowledge to a script inside source control, and triggering it via a CI side script that contains only parameters and other “context” data to invoke the source control scripts correctly, we “inject” the version aware knowledge of what the build script should do, into a higher level CI process trigger, that does not care about product file structure, but still gives us the advantages with had before with a CI process.

3 Pattern: Shipping Skeleton

Other Names:

"Hello World", Walking Skeleton (based on XP[1]), Tracer Bullet Build,

3.1 Problem:

Remember when you had a working product, but you could not ship it, or shipping and deploying it took a long time, or could not be estimated?. By that time, however, your product was too big and your free time was too short to start creating a working automated build and deploy process. As time went on, shipping became even more and more of a nightmarish manual task, and automating it became more and more of a problem.

Now you're at the start of a new project, and you want to avoid all that pain in your new project.

3.2 Forces:

You want to avoid the pain of automating the build and deploy cycle. But you also don't want to spend too much time working on it.

3.3 Solution:

One solution is to avoid this from happening at the start of the product cycle.

Before starting development on the actual product, **start with a shipping skeleton** – An empty solution with nothing but a "hello world" output, that has the basic automated build and deploy cycle working. You should be able to ship this empty hello world project with the click of a button, in a matter of minutes.

once the shipping skeleton is in place, you can start filling out the product with features, and growing the build scripts in small increments alongside the product.

Basic Shipping Skeleton Structure:

1. A Build script (in source control) for compiling the current source
2. A Continuous Integration Server that has a "CI" build configuration, triggered by code check-in, that invokes build script from the previous bullet (also see Build Script injection[2] for more on this pattern)

[1] http://alistair.cockburn.us/Walking+skeleton

[2] http://osherove.com/blog/2012/12/30/build-patterns-script-injection.html

3. Another build configuration on the CI server for "**Deploy**". This can be either deploy to test, or deploy to production. Usually you want at least a "deploy to test" succeeding before having a "deploy to production" CI build Configuration. This "Deploy to test" gets invoked automatically upon "CI" configuration passing. Later on, as the build process matures, you can change this, but for now, knowing that the product is deployable as soon as possible, while it changes so much, is important for quick feedback cycles.
4. If there isn't a "Deploy to production", create it. This deploys the product to a production machine. If it is a web site, it deploys the website to a web server. if it is an installer, it deploys and runs the installer on a machine that will act as a "production" machine. either mimicking a user machine, or or an enterprise machine where the product will be installed. for web servers, make them as real as possible, all the way, even by making them public (although possibly password protected). This gives you the chance to make the web server become real production by the switch of a dns setting.

It is important to note that the whole point of this structure is to "ship" your product. If you end up deploying to a simple test server, or to anything that is not "touchable" by a real customer, you are simply delaying this problem to a later time, a time when your pressure is much higher, and the stress levels rise accodingly. Setting up production servers, with all the necessary IT approvals in some companies, can take weeks. Waiting weeks to deploy a product that is ready now, under stress, is not a recommended strategy for a thriving, happy, work environment.

Get it out of the way now, and have a coffee when it's time to show a demo. When the coffee is ready, so will your demo.

3.4 Summary

By starting with a shipping skeleton, you give yourself several benefits:

- You can ship at will
- Adding features to the build and deploy cycle is a continuous action that takes a few minutes each day, if at all.
- You can receive very quick feedback on features or mockup-features you are building into your product.

4 Build Pattern: Location Agnostic Script

4.1 Symptoms:

- Running the build script on a machine where it was never run before requires extra pre-work to make the machine ready, such as mapping drives, creating a special build script folder etc.

4.2 Problem:

- The build script uses hardcoded file and directory paths to find its dependent source files. For example, it searches for a solution file in Z:/SolutionFileName . That means the build script has specific requirements from its environment before running, which causes lots of menial, boring work to do before being able to run it.

4.3 Solution:

- Have the script use only relative paths to find files.
- Make the script part of source control, or deploy it into the root of the source control branch to build, so that it has access to all the files it needs

5 Build Pattern: Fill In the Blanks

5.1 Other names:

- DRY Build Scripts (Don't Repeat Yourself)
- Infrastructure agnostic scripts
- Parameterized Scripts
- Fill In the Blanks

5.2 Symptoms:

- You find yourself having to change the script in many places, because a single "thing" has changed.
- For example, the name of the remote machine you are deploying to has changed, and that name has to be changed in several places in the build script.
- Another example: you want to run the same set of actions both on the debug source directory, and on the release source directory.

5.3 Problem:

- The script is breaking the "Don't repeat yourself" rule, that many programmers know and love (sometimes they love to hate it).There is not "Single source of truth" about specific knowledge in the script. For example, there is no single location that defines what is the deployment target machine name. Instead, that machine name appears in many different places in the script.

5.4 Solution:

There are many ways to solve this problem with build scripts, but they all involve using variables inside the build script, and then "fill in the blanks" just once per script:

- **Script variables** as the single source of truth: You can define variables in your script. (for example, a variable that stands for the machine name to deploy to, that gets replaced in the script with its underlying value in all places it is used). The use these variables everywhere you see repetition.
- **Script Configuration File**: Assuming you have the ability to define script variables, you can read the value of these variables from a configuration file before running the script. This gives an added bonus of not having to muck around with the script itself when your IT admin suddenly changes your deployment machine.

- **Environment Variables**: Much like the previous solution, environment variables have an added bonus – they are usually very easy to set from within your CI server (such as TeamCity), so that there is now a clear separation between the script knowing about the source files, and the CI build configuration knowing about the IT structure and requirements. In my own builds I use this technique a lot, and might have 10 or 20 environment variables set by the build configuration in the CI server, that are used by the build script that is in source control. That way you can have the same deploy script be used to deploy to a test, staging or production machine, with a simple environment variable change.
- **Script Parameters**: This is also useful when using a CI server – instead of setting env. variables, the ci build configuration calls the build script with parameters in the command line call. these are then used to set values for variables inside the build script.

5.5 Possible Side Effects:

- **Unclear Script Requirements to run**. It can be hard to find all the possible needed environment variables or parameters needed to run the script successfully. It is up to the script creator to maintain and document the list of needed parameters as much as possible.
- **Hard to Debug**.To be able to run the build script on your local machine you will have to provide all the needed env. variables or parameters locally. If the build system supports variables that can be overridden by environment variables, but that have a default value if there are no environment variables, this can ease this pain.

6 Pattern: Extract Script

This is a variation of [Fill in the blanks][1] and will become part of the [Builds Book.][2]

6.1 Symptoms:

You have multiple build scripts that contain parts that execute the same set of complicated actions, differing only by path names, files names or other values used in the script. Changing these actions among many scripts is a big hassle (a.k.a "Shotgun Surgery").

6.2 Problem:

Your build scripts are breaking the DRY rule. Don't repeat yourself.

6.3 Solution

Extract the common set of actions into a new short build script, that receives as parameters, or environment variables the values that different between the various scripts. Then have the other scripts call the new script with the correct parameter values.

6.4 Example:

In one of my projects, as part of the deployment of a site, I also wanted to add a set of actions that makes sure the initial web page of the site was up after deploy. if not, the build should fail. It turns out that sometimes it takes the server a few seconds after reset or file change to get back to working shape, and meanwhile it returns various different codes like 403. So the script had to do the following:

- loop
- get the web page or response and write it to a file
- read the file result into a variable
- if result was 404 or unknown to break the build
- if the result was 403 then continue looping
- wait one second
- if the result contained the right text $text_to_find then build passes

This set of actions was needed when deploying to the test server, staging server, and production server.

I extracted only these actions to a new script named "check"_site" and made the deploy scripts (there were different deploy scripts for production and test) execute the check_site script as part of their work.

6.5 Possible Side Effects

you might end up with many small scripts, without knowing in which order they are supposed to be executed. To mitigate this you might want to only have one MAIN script call all the other scripts, instead of various small sub scripts calling other sub scripts. If only one main script is in charge of sub script call flow, the build can be readable by looking at the main build script.

7 Build Pattern: Cumulative Build

7.1 Other Names:

- Distilled Build
- Adopt/Inherit/Share/Import Artifacts
- Borrow Build Artifacts
- Pre-Chewed Artifacts
- Single Responsibility Build Configuration

7.2 Symptoms:

- Your build configuration on the CI server is taking a long time to complete.
- Each Build Configuration down the build chain takes longer and longer than the build configuration up the chain

7.3 Problem:

Your build config might be doing too many things.

7.4 Forces:

You want your build time to take less, but you do not want to remove any actions from the build because all the build actions are important.

7.5 Solution:

First, Have each build configuration do one thing, and then “share” the resulting artifacts, such as built binaries, so that other build configurations can use them.

Then, for each build down the pipeline, have the build import the artifacts from the build config up the line, and simply do one more action on the artifacts, and share them with the next build up the pipeline.

7.6 For example

say you have the following build configurations in TeamCity before applying the solution:

- **Continuous Integration Build** : Gets from source control, compiles in debug, runs fast tests in debug
- **Nightly Build**: Compiles in debug, runs tests in debug, compiles in release, runs tests in release, runs slow tests on release, deploys to test env.
- **Release Build**: Compiles in debug, runs tests in debug, compiles in release, runs tests in release, runs slow tests on release, deploys to release env.

Here is one way we can make each build do one thing, then share the artifacts with the build up the pipeline:

- **Continuous Integration Build** : Gets from source control, compiles in debug, runs fast tests in debug, *shares source and binaries artifacts*
- **Nightly Build**:* Gets build artifacts from CI build. *Compiles in release, runs fast tests in release, runs slow tests in release.* Shares binaries as artifacts.*
- **Release Build**: *Gets build artifacts from nightly build*, deploys to release env. *Shares deployed binaries as artifacts.*

Now, each build does just one thing, on top of the pre built binaries. For example, only the first build of the bunch gets from source code. Only two builds compile. The deploy build doesn't do anything but deploy.

This also eases the build script management, each build gets a separate build script in source control, that can be reused if needed.

8 Pattern: Gated Commit

8.1 Other names:

- Pre-tested commit
- Gated Check-in
- Private Build
- Delayed Commit

8.2 Symptoms:

The build keeps breaking for trivial problems.

8.3 Problem:

Developers do not know what are the outcomes of committing their source code into the master branch. so they blindly commit and wait to see the results.

8.4 Forces:

You want to check in (commit) your changes into the master source control, but you are afraid of breaking one of the immediate build configurations on the CI server. For example, you are not sure if all the tests in the nightly build will pass, and your machine is too slow to run the nightly build locally.

You'd like to make sure, before you commit or merge into the master branch, that your changes won't break. but this is very time consuming task locally.

8.5 Solution:

A gated commit is a process that can be supported by the CI server. It combines several steps, in several variations:

- Developer Requests a gated commit **before** committing actual changes.
- the CI server gets the current snapshot of files on the developer's local machine through a local plugin to the text editor, usually. (in TFS this is called "shelving")
- The CI Server merges the current snapshot with the master source version, as a separate, temporary set of files. this is not being committed to the master branch.

- The CI server then runs the specific build configuration requested by the developer , such as a nightly build, with the current code. this might include compilation, running tests, deployment etc.
- if the build configuration passes, the temporary file set is committed to master.
- if the build configuration fails, the temporary file set is discarded, and the developer is notified of the build failure, without interrupting the main master branch builds.
- Developer now has a log file to go through and see what to fix before checking in.

8.6 Examples:

Teamcity provides a "pre-tested commit[1]" feature. Install the teamcity plugin in visual studio, and get a new button that allows you to run the current sources as a pre-tested commit. if it passes, changes are committed to source control.

8.7 Side effects:

This can lead to an overall feeling in the team that breaking the build is wrong, and is to be avoided at any cost. This might not be a healthy attitude, since the original purpose of a build process was always to find out problems. You might say that if the build breaks, then the build did it's job. This culture of "breaking the build is bad, and it's your fault" can lead to developers hiding their efforts, in an effort not to look stupid in front of their peers. in essence, a private build can lead to moving from a group effort, into individual based comparison system "who breaks the build the most?".

More problematic: if developers always get the build failing **privately**, developers will feel they have to **solve** the build privately. when the build fails publicly, it can trigger help from the rest of the team to solve the problem faster since more brains are involved.

[1] http://www.jetbrains.com/teamcity/features/delayed_commit.html

9 Tests and Builds are Made to be Broken

Many agile teams seem to be very focused on not breaking the build. I myself used to jokingly say that your team should have a "I Broke the Build" T-Shirt, to switch between team members. That was a mistake. I think that builds, much like tests, are made to be broken. Avoiding the public build is like debugging so you won't have to run your tests. Well ,that last sentence is a bit of hyperbole. It's like running your tests locally before running them on the build server. Which is actually a great thing to do, unless you can only run some of the tests locally, or it takes time to set up all the integration tests to run on your machine etc.. then it's just a waste of time. Just run them in the build process. that's what it's for.

As time goes on, I see many team members wasting precious time trying to avoid the build, by running it privately, when the whole purpose of the build was to give the indication of failure.

I want to bring forward a small comment I made next to the "Gated Commits" build pattern in this book:

This(pre-tested commits) can lead to an overall feeling in the team that breaking the build is wrong, and is to be avoided at any cost. This might not be a healthy attitude, since the original purpose of a build process was always to find out problems. You might say that if the build breaks, then the build did it's job. If the build never breaks, why have it in the first place?

The Blame Game:

This culture of "breaking the build is bad, and it's your fault" can lead to developers hiding their efforts, in an effort not to look stupid in front of their peers. in essence, a private build can lead to moving from a group effort, into individual based comparison system "who breaks the build the most?".

Lost Productivity:

More problematic: if developers always get the build failing **privately**, developers will feel they have to **solve** the build privately. when the build fails publicly, it can trigger help from the rest of the team to solve the problem faster since more brains are involved.

What about making all the other team members wait to commit until the build finally passes? Isn't that lost productivity? theoretically, yes. but it also is a great reason for the team to help out and make the build pass. It is a "Stop the line" mentality, which we like to adopt in lean circles.

10 Rolling Builds and the Plane of Confidence

When I check in code I have been working on, I feel:

1. I want to wait for the build result
2. I don't want to waste time waiting for the build results, doing nothing
3. But I fear doing something in the code until I'm sure I didn't break anything

Supposedly, this is where a CI build configuration comes into place. CI builds are supposed to be as fast as possible so we can get feedback about what we checked in as quickly as possible and get back to work.
But the trade-off is that CI builds also do fewer things so that they can become faster,
Thus leaving you with a sense of some risk, even if the build passed.

This is why I like to have "**Rolling Builds**". I like to have builds trigger each other in the Continuous Integration Server:

- A Check-in triggers the CI build
- A successful CI build triggers a nightly build.
- A successful nightly build triggers a deploy to test build.

I think of the builds now as single waves crashing on my shoreline. Each build is a slightly bigger wave. Each wave crashing on the shore brings with it a layer of confidence in the code. And because they happen serially, I can choose to just relax and watch the waves of increasing confidence crash on the shore, or I can choose to continue coding right after the first wave of confidence.

As I program, the next waves of build results hit the shoreline, and my notification tray tells me what that wave brought with it. Another "green" result wave tells me to go on about my business. A "red" wave tells me to stop and see what happened.

But I always wait for at least the first wave to come ashore and tell me what's going on. I need that little piece of information that tell me "seems legit so far" so I can feel 50@5 good about going back to coding. If my changes were big, I might wait until the next wave to see what to do.

So my confidence after check in is not a black or white result. It is a continuous plane of increasing confidence with breakability of the code that I am writing, that peaks when the code is deployed to production.

11 Fast, Dry, Single Purpose Scripts

One of the things that kept frustrating me when I was working on various types of build configurations in our CI server, was that each build in a rolling wave would take longer than the one before it.

The **CI build configuration** was the fastest, the **nightly build** was slower, because it did all the work of CI (compile, run tests) plus all the other work a nightly needed (run slow tests, create installer etc..). The **Deployment to test** build would maybe just do the work of the CI and then deploy, or, in other projects, would do the work of nightly, then deploy, because wouldn't you want to deploy something only after it has been tested?

And lastly, **production deploy builds** took the longest, because they had to be "just perfect". so they ran everything before doing anything.

That sucked, because at the time, I did not discover one of the most important features that my current CI tool had : the ability to share **artifact**s between builds. Artifacts are the output of the build actions. these could be binary compiled files, or they could be configuration files, or maybe just a log file, or just the source files that were used in the build , from source control.

Realization

Once I realized that build configurations can "publish" artifacts when they finish successfully, and that other build configurations can then "use: those artifacts in their own work, things started falling into place.

I no longer needed to re-do all the things in all the builds. Instead, I can use the DRY[1] (Don't repeat yourself) principle in my build scripts (remember that build scripts are kept in source control, and are simply executed by a CI build configuration that provides them with context, such as environment parameters, or the artifacts from a previous build).[2]

Instead, I can make each rolling wave build[3], only do one small thing (single responsibility principle[4]), that gets added on top of the artifacts shared by the build before it.

For example:

- The CI Build Only gets the latest version from source control, compiles in debug mode, and runs the fastest tests. Then it **published the source and compiled binaries for other builds to use later on**. Takes the same amount of time as the previously mentioned CI build.
- The Nightly build gets the artifacts from the latest successful CI Build, and only: compiles in RELEASE mode, runs the slow tests, creates installers, and **publishes the installer**, **the source**, **and the binaries for later builds**. Notice how it does not even need to get anything from source control, since those files are already part of the artifacts (depending on the amount of source this might not be a good idea to publish source artifacts due to slowness, but that depends on the size of the project). takes half the time of the previously mentioned nightly build.

[1] http://en.wikipedia.org/wiki/Don't_repeat_yourself

[2] http://osherove.com/blog/2012/11/17/build-bullet-1-use-version-aware-build-scripts.html

[3] http://osherove.com/blog/2012/11/27/build-bullet-4-rolling-builds-and-the-plane-of-confidence.html

[4] http://en.wikipedia.org/wiki/Single_responsibility_principle

- The Deploy (to test) build, gets the installer from the nightly build that last passed successfully,Â and deploys it to a machine somewhere. It does not compile, or run any tests. **It publishes the Installer it used**. takes 30 seconds.
- The Deploy (to Staging) will just get the latest successful installer build artifacts from (deploy to test) builds, and deploy them to staging, and also **publish** the installer it used. Takes 30 seconds.
- The Deploy (to Production) will just get the latest successful installer build artifacts from (deploy to staging) builds, and deploy them to production, and also **publish** the installer it used. Takes 30 seconds.

Notice how with artifact reuse, I am able to reverse the time trend with builds. The more "advanced" a build is along the deployment pipeline, the faster it can become.

And Because each build in the deploy pipeline is only getting artifacts of successful builds, we can be sure that if we got all the way to this stage, then all needed steps have been taken to be able to arrive at this situation (we ran all the tests, compiled all the source…)

12 Avoid XML-As-UI

Having your build scripts, or continuous integration build configurations editable only via raw XML is one of the worst things you can do for maintainability and readability.

Treat your build scripts like you treat your source code (or, how developers **should** treat their source code) – with respect for readability and maintainability, not just for performance.

To that end, XML is a bad candidate to have. If you have any more than 15 or 20 lines of xml in a build document:

- It is hard to tell where things start, and what they depend on
- It is hard to debug the build
- It is hard to add or change things in the build logic
- XML case sensitivity sucks
- Creating, Adding and usingÂ custom actions is a chore

To avoid XML, you can start using some of the many build tools out there that are either visual, or support a domain specific language for builds that is more readable, maintainable or debuggable than XML:

- Rake[1] – Ruby Based DSL (Domain Specific Language) for builds, that is very robust and quite readable
- FinalBuilder[2] and VisualBuild[3] are two visual build scripting tools that give great readability into your build script

Image: An example of FinalBuilder– My current favorite build tool.

Do not use (for build scripts) tools such as :

- Ant, Nant, maven or msbuild (XML based editing)
- TeamCity, FinalBuilder Server, Microsoft Team Build Server (use these only for build configurations, not the scripts themselves[4])

Avoid XML editing like the plague, and you might find:

- It is easier to introduce new team members to your build script, and ask them to share in the burden of maintaining it quickly.
- It is easier to enable or disable certain actions quickly for debugging purposes
- It is easier to reuse build scripts
- It is easier to find out what went wrong and where when the build fails if you have a visual tool

[1] http://rake.rubyforge.org/
[2] http://www.finalbuilder.com
[3] http://www.kinook.com/VisBuildPro/
[4] http://osherove.com/blog/2012/11/17/build-bullet-1-use-version-aware-build-scripts.html