Automation for the people: Continuous Integration anti-patterns

Make your life with CI easier by learning what not to do

Paul Duvall

December 04, 2007

While Continuous Integration (CI) can be extremely effective at reducing risks on a project, it requires a greater emphasis on your day-to-day activities related to coding. In Part 1 of a two-part article in the Automation for the people series, automation expert and co-author of Continuous Integration: Improving Software Quality and Reducing Risk, Paul Duvall, lays out a series of CI anti-patterns, and more importantly, shows how to avoid them.

View more content in this series

Throughout my career, I've found that I learn more by discovering what doesn't work for a particular situation rather than what works. For example, early in my career, in an effort to quickly release software, I skipped unit testing as I believed the effort wasn't worth the cost. Luckily, I learned that pushing untested code into production didn't work; consequently, I began to write unit tests.

It seems that our industry largely agrees with my style of learning; in fact, we've even created our own word for capturing practices that don't work in a particular context: anti-patterns. At a high level, anti-patterns are solutions that appear to be beneficial, but, in the end, they tend to produce adverse effects.

False evidence appearing real

Sadly, I've also found that when inexperienced teams attempt to introduce the practice of CI, they have a high chance of mistakenly introducing a number of anti-patterns, which ultimately can lead to a lot of frustration over what's supposed to be a boon. Unfortunately, the term CI itself is often misappropriated and thus, I often hear statements like "CI doesn't work for large projects" or "our project is so unique that CI won't work" when, in fact, CI isn't the issue at all — it's the ineffective application, or nonexistence, of certain practices that have led to frustration.

About this series

As developers, we work to automate processes for end-users; yet, many of us overlook opportunities to automate our own development processes. To that end, Automation for the
I'm going to set the record straight for the sake of CI and detail six anti-patterns in this article:

- **Infrequent check-ins**, which lead to delayed integrations
- **Broken builds**, which prevent teams from moving on to other tasks
- **Minimal feedback**, which prevents action from occurring
- Receiving spam feedback, which causes people to ignore messages
- **Possessing a slow machine**, which delays feedback
- Relying on a bloated build, which reduces rapid feedback

If you do CI for long enough, it's a near certainty you'll experience the effects of these anti-patterns. That's OK, but if they happen too frequently, it will limit the manifold benefits of CI. Therefore, if you want to limit the occurrence and negative impact of these anti-patterns, this article is for you.

### Delayed integration due to infrequent check-ins

**Name:** Infrequent Check-in

**Anti-pattern:** Source files stay checked out of a repository for long periods of time due to the amount of changes required for tasking.

**Solution:** Commit smaller chunks of code frequently.

The premise of CI is that teams can receive speedy feedback on the status of code under development; what's more, this frequent style of software integration reduces the time (and corresponding pain) of more traditional late style "big bang" integration efforts. Effective CI, however, is predicated on the concept that changes are occurring on a regular basis (so that builds can occur frequently!). If code lives on desktops (as opposed to repositories) for long intervals, bad things will happen because other changes are occurring to different parts of the system.

### A commit a day keeps the integration woes away

A common rule of thumb is to check in code at least once a day. An effective technique I use: If I feel the need to take a break, I first see if I'm at a point where I can run a local build and then commit my code. Then I take that break.

In essence, by not committing changes frequently, you are delaying integration and the longer that delay, the more effort will be required to sort out adverse effects (such as someone else's changes affecting your code). On a project that uses CI, I recommend developers commit their code at least once a day, but I believe it's best to check in code many times a day.

### Make life easier with smaller tasking

I can hear the ire of some developers now as they inevitably complain that it's difficult to check in code daily when they are busy modifying so many files. Essentially, this statement makes my point — to commit source changes daily, you need to think smaller. In fact, you need to break down coding tasks into succinct pieces of work so that your changes are small.
Rather than implementing all the features on a business object in one large effort, for example, like coding the archetypical `read()`, `write()`, `update()`, and `delete()` methods, you can alternatively first code the `read()` method (and a corresponding test, right?) and then check in the class, such that the entire code base is then integrated. Next, you can implement another method and follow the same practice until you are finished with the entire task. That way, you are maximizing the benefits of CI and giving yourself a huge boost of confidence that your code is working with everyone else's code all the while.

Remember, even if you and your team are performing many CI practices correctly, if team members aren’t committing source changes at least daily, your team will receive minimal benefit from CI. And more often than not, that leads to a perception that CI doesn’t work, which couldn’t be farther from the truth.

**Broken builds slow the cadence of development**

**Name:** Broken Build

**Anti-pattern:** Builds stay broken for long periods of time, thus preventing developers from checking out functioning code.

**Solution:** Developers are immediately notified upon a build breakage and make it a top priority to fix a broken build.

Believe it or not, builds become increasingly more troublesome to fix the longer they stay broken. This is because there are more files, more changes, and more dependencies that make the isolation of the defect difficult. Accordingly, when someone is notified that the build is broken (through e-mail, RSS, or other mechanisms), it should be their top priority to fix the offending aspect; otherwise, the longer the build stays broken (especially with teams that make frequent changes), the harder it is going to be to rectify the situation once someone decides to take action.

Broken builds aren’t always a bad thing. In fact, broken builds enable you to learn quickly if there’s a problem with software. Broken builds become a problem when the build is broken often or when it stays broken for too long. You never want to leave the office for the day if the build is broken.

**Builds that never break**

Not so fast. I’ve heard some developers say that “your build should never ever break!” This is poor advice. You want to prevent many of the common build errors, such as missing files or broken tests; however, builds that don’t break may be telling you something as well. The build might not being doing all that much (maybe just a compile and a few unit tests). I call this “Continuous Ignorance” and this can sometimes be worse than having a high frequency of broken builds.

**Private builds reduce broken ones**

One of the more useful techniques that prevents a broken build is what's known as running a private build prior to committing code into a repository. The steps to executing a private build, at a high level, are as follows:
1. Check-out code from a repository.
2. Make code modifications locally.
3. Perform an update with the repository to integrate changes from other developers.
4. Run a local build.
5. Once the build is successful, commit changes into the repository.

Figure 1 demonstrates this practice in action. Note how the work flow stresses frequent synchronization with a repository, thus permitting periodic check-ins and limiting broken builds — that's hitting two birds with one stone!

**Figure 1. Run a private build to reduce broken integration builds**

By performing private builds before you check in source code (which, of course, is happening frequently, right?), you can prevent many of the typical errors that end up putting builds in a broken state; thus, you save time and headaches.

**Impeding action with minimal feedback**

**Name:** Minimal Feedback

**Anti-pattern:** Teams choose not to send build status notifications to team members; thus, people aren't aware a build has failed.

**Solution:** Use various feedback mechanisms to relate build status information.

Often, when setting up a CI system, teams decide that receiving e-mails is tantamount to spam; thus, they decide that "for the time being" no notifications will go out. You can't take action, however, if you're not receiving any feedback from a build. In fact, feedback is one of the most crucial aspects of CI; having said that, it's also crucial that the feedback be effective.

If you want to expand the mechanisms by which build status information is relayed to team members, the use of visual and sound devices can be quite useful, especially for co-located teams. Devices like the Ambient Orb are effective in providing near real-time visibility of a build's status. For example, when a build happens to fail, the orb could turn red and when the build passes, the orb turns green. What's more, orbs can be useful for disseminating information such
as whether the cyclomatic complexity of a code base is increasing or decreasing by changing different colors (like blue for good, yellow for bad).

**Let your creative juices flow**

Setting up an Ambient Orb couldn't be easier. Listing 1 demonstrates how to set up an Ambient Orb in Ant by using Quality Lab's open source OrbTask:

**Listing 1. Using the Ambient Orb Ant task**

```xml
<target name="notifyOrb">
  <taskdef classname="org.qualitylabs.ambientorb.ant.OrbTask"
    name="orb" classpathref="orb.class.path"/>
  <orb query="http://myambient.com:8080/java/my_devices/submitdata.jsp"
    deviceId="AAA-9A9-AA9"
    colorPass="green"
    colorFail="red"
    commentFail="Code+Duplication+Threshold+Exceeded" />
</target>
```

In Listing 1, the task is configured to turn the orb green on a pass status and red on a failure status. Figure 2 illustrates a green orb which, presumably, means the most recent build status was successful:

**Figure 2. Successful build!**

By being creative, your team can utilize various feedback mechanisms so that team members don't begin ignoring build status messages. What's more, these lively techniques make it fun to get into the CI groove; plus, they make it easy to notice when there's a problem that requires action when necessary.

Other notification mechanisms include:
• RSS feeds
• Taskbar monitors such as CCTray (for CruiseControl)
• X10 devices (like LavaLamps)
• Instant Messages through Jabber, etc.
• SMS (Text Messages) for those who don't receive enough from friends and family

One caveat: You need to strike a balance between too much information and too little information. Your feedback mechanisms should be varied and alternate periodically based on a working environment. For instance, for co-located teams, playing a sound (like a fire whistle when builds fail) can be effective; however, other teams may prefer an Ambient Orb (which won't necessarily scare you while you're deep in thought).

The cold shoulder of spam feedback

Name: Spam Feedback

Anti-pattern: Team members quickly become inundated with build status e-mails (success and failure and everything in between) to the point where they start to ignore messages.

Solution: Feedback is succinctly targeted so that people don't receive irrelevant information.

As opposed to the anti-pattern of not receiving enough feedback, I often find teams who naively decide everyone should always receive feedback (say, an e-mail) every time a CI server does anything. Nothing screams "ignore me" louder than over saturation; too much feedback and your team will quickly start to see CI feedback as spam. Then, when something serious shows up (e.g., the build is actually broken) it may go unnoticed.

Precise targeting keeps spam at bay

In Listing 2, an example CruiseControl configuration file demonstrates the effective use of sending e-mail notifications. In this case, the technical lead always receives the e-mail whether the build is successful or not, the project manager receives an e-mail only if the build fails, and any developers who recently committed a source change to the repository will also be notified.

Listing 2. Sending email notification using CruiseControl

```xml
<project name="brewery">
    ...
    <publishers>
        <htmlemail
css="/webapps/cruisecontrol/css/cruisecontrol.css"
mailhost="localhost"
xsldir="/webapps/cruisecontrol/xsl"
returnaddress="cruisecontrol@localhost"
buildresultsurl="http://localhost:8080"
mailport="25"
defaultsuffix="@localhost" spamwhilebroken="false">
            <always address="techlead@localhost"/>
            <failure address="pm@localhost" reportWhenFixed="true"/>
        </htmlemail>
    </publishers>
    ...
```
Considering that feedback is one of the most crucial aspects of a CI system, it warrants some discussion. Although they are opposite sides of the spectrum, there’s a fine line between minimal feedback and spam feedback. When a build has broken, the feedback must be sent to the right people in a timely fashion and it must provide people with actionable information. If the build is successful, it should be sent to only a select few that either made the most recent change or are in a leadership position and thus may want this information. Blasting everyone with status messages all the time is a surefire way to limit the benefits of a CI process in short order.

Don't delay feedback with a slow machine

Name: Slow Machine

Anti-pattern: A workstation with limited resources is used as the build machine, leading to lengthy build times.

Solution: The build machine has optimal disk speed, processor, and RAM resources for speedy builds.

Years ago, I was on a relatively large project that had over one million lines of code, which took over two hours to compile. As we attempted to integrate more often, waiting for the Configuration Management team to go through this process was becoming more and more painful. Of course, two hours was the best case scenario because builds would often fail, so the process typically took several days to complete (talk about painful!). After several weeks of this monstrously slow process, it was clear that the solution was to purchase a machine with enough disk space for all of the files being checked out and generated as a result of a build, the fastest processor speed for handling many instructions, and enough RAM for running tests and other processes that were memory intensive.

Do you feel the need for speed?

With this monster box, we were able to reduce the time of that mammoth build from 2 hours to 30 minutes; thus, the benefit of putting a few extra dollars into a state of the art machine ended up saving the team considerable time and money and ultimately integrated the software more quickly (which meant we found issues sooner!).

There’s nothing wrong with starting out with an extra workstation to perform integration builds; however, the moral of this story is that you should seriously consider upgrading your build machine if you find that it’s lagging in speed or memory or you find the hard-disk thrashing; the time you save by speeding up your builds helps you to get quicker feedback, fix problems quickly, and move on to your next development task sooner.

Bloated builds delay rapid feedback

Name: Bloated Build

Anti-pattern: Throwing everything into the commit build process, such as running every type of automated inspection tool or running load tests such that feedback is delayed.
Solution: A build pipeline enables running different types of builds.

Some development teams become so enamored with all of the processes that can be added to an automated build that they forget that it takes time to perform these actions. Case in point, remember my project that took two hours to compile? Imagine if we had added executing tests as a part of that build process. With over a million lines of code, how long do you think it would take to run a static analysis tool against the code? If you think an eight-hour build process is unbelievable, think again. I run across these beasts regularly.

People tend to incrementally move toward the bloated build in a quest to provide more and more build information to team members. There is a way to strike the balance between giving the development team rapid feedback while also providing useful information from the CI build process.

Pipelining builds for efficiency

If you find your build process is unduly time consuming and assuming that you've implemented other duration improvement techniques (such as acquiring a fast machine) and optimizing test execution times, it may become necessary to create what's known as a build pipeline. The purpose of a build pipeline is to execute longer-running processes, in essence, asynchronously, so that once someone has checked in code, they're not delayed in receiving feedback.

For instance, if a build takes more than 10 minutes to execute, a build pipeline can be established so that after someone commits code into a repository, an initial, lightweight build is run. This "commit" build consists of (hopefully) lightweight processes like compiling and running quick unit tests. Based on the success of that initial build, a secondary build can then be run, which executes longer-running tests, software inspections, and even perhaps deployment to an application server.

For example, in Listing 3, I've configured CruiseControl to check for modifications in a repository. When it discovers a change, CruiseControl then runs what's known as a delegating build, which calls the project's main build file (like build.xml, if Ant is in use). What's unique though, is that CruiseControl executes a different target, which then executes lightweight processes, like compilation and fine-grained unit tests.

Listing 3. CruiseControl configuration checking for modifications

```xml
<project name="brewery-commit">
  ...
  <modificationset quietperiod="120">
    <svn RepositoryLocation="http://brewery-ci.googlecode.com/svn/trunk"/>
  </modificationset
  ...
```

In Listing 4, CruiseControl is configured to check for modifications against the brewery-commit project (which isn't in the repository — it's actually looking at a log file). When a change is discovered, CruiseControl runs another delegating build. This build will call the same build file, but with an alternate target, which might execute longer-running processes like functional tests, software inspections, etc.
Listing 4. CruiseControl configuration executing a long running build

```xml
<project name="brewery-secondary">
  ...
  <modificationset quietperiod="120">
    <buildstatus logdir="logs/brewery-commit" />
  </modificationset>
  ...
</project>
```

The *Bloated Build* anti-pattern is the most often cited excuse for why CI doesn't work. But, as you can see, it doesn't have to be this way if you use a build pipeline. An effective build pipeline maximizes the benefits of the "80/20" rule: spend 20 percent of build time on the areas that lead to 80 percent of build errors (such as missing files, broken compilation, and test failures). After this process is complete and developers have received feedback, then run secondary builds that may take longer to run but may yield 20 percent of other build errors or relative priority.

### Anti-patterns can be fixed

CI anti-patterns can prevent teams from obtaining the most from the practice of Continuous Integration; however, the techniques I've described in this article can help prevent the frequency of these anti-patterns. You've seen that:

- Committing code often can prevent complex integrations down the road.
- Preventing the vast majority of broken builds can be as easy as running a private build prior to committing source files.
- Using a variety of feedback mechanisms can prevent stale build status information that would otherwise be ignored.
- Targeting feedback to people that can take action is one of the better ways to inform team members of build problems.
- Spending some extra money on a build machine is worth the investment in speeding up the feedback to team members.
- Creating a build pipeline is one technique that reduces build bloat.

The anti-patterns I've described in this article are the ones I see most often, but there are others, including:

- **Continuous Ignorance**, where a build consists of minimal processes, resulting in an always successful build status.
- The build *only works on your machine*, which can delay the time between when a defect is introduced and when it's fixed.
- **Bottleneck Commits**, which tend to cause broken builds and prevent team members from going home.
- Running *intermittent builds*, which delay rapid feedback.

In Part 2, I cover other CI anti-patterns that may prevent you from getting the most from Continuous Integration.
Related topics

- *Continuous Integration: Improving Software Quality and Reducing Risk* (Paul Duvall et. al, Addison-Wesley Signature Series, 2007): Learn over 40 CI practices in different languages and platforms.
- "Spot defects early with Continuous Integration" (Andrew Glover, developerWorks, 2007): In this tutorial, Andrew Glover provides a look at the fundamental aspects of Continuous Integration and walks you through setting up a CI process using best-of-breed open source technologies.
- "Continuous Integration" (Martin Fowler, martinfowler.com): Fowler’s seminal article on Continuous Integration.
- "Continuous feedback" (Paul Duvall, developerWorks, November 2006): Learn how to get immediate feedback with every source code change.
- "Is Pipelined Continuous Integration a Good Idea?" (infoq.com, September 2007): Leading CI advocates weigh in on the build pipeline.
- Ambient Orb Ant task: Change the color of your Orb, based on build status, using this Ant task.

© Copyright IBM Corporation 2007
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)