

# Long Build Trouble Shooting Guide

Jonathan Rasmusson

ThoughtWorks Canada Corporation, 805-10<sup>th</sup> Ave SW, 3<sup>rd</sup> floor  
Calgary, Alberta, Canada T2R 0B4  
jrasmusson@thoughtworks.com

**Abstract.** Excessively long build times severely reduce a team's ability to apply the XP practice of Continuous Integration. Long build times have a severe impact on team morale, productivity, and project ROI. With the application of techniques described in this Long Build Trouble Shooting Guide, teams will be able to keep their builds from exceeding 10 minutes in length. By keeping builds short, teams will be able to minimize the cost of integration, thereby freeing them to focus on other critical project areas.

## 1 Introduction

### 1.1 Why long builds are problematic

Feedback is a wonderful thing. There is a certain feeling of satisfaction and joy that comes in software development when we are able to make changes to a system and receive immediate feedback as to whether our change left the system in a better state than we found it. That's great feedback.

In fact, we are so addicted to feedback that anytime something increases the amount of time it takes us to receive feedback, we feel pain.

Long builds are painful because they decrease the speed at which we obtain feedback. We do not like the state of uncertainty - not knowing if our changes have been successfully integrated into the system.

Further, integration becomes more difficult the longer developers code without merging their changes. Integrating changes early and often, helps minimize the opportunity and impact of multiple parties working on a common code base.

Allowing the feedback duration to grow unchecked eventually impacts the team productivity. Instead of working on new features, a long build process forces developers to idly wait while the build runs. Or they go on and start adding new functionality to code that may not have yet already been checked in (thus further increasing the amount of integration to be done later).

One might suggest that there are many other things that developers could be doing while their build is running (helping others, catching up on email, surfing the web, or think about upcoming problems).

While this sounds good in theory, I find this does not happen in practice. Writing quality code is a thought intensive process. Every time a developer is distracted or pulled out of their problem domain, it takes a considerable period of time before they can return to their previous level of productivity. Anything that takes time away from coding and receiving rapid feedback from the system is counter productive.

Another consequence of long builds is that developers may be tempted to take shortcuts. For instance, they may skip running the build locally before checking in – confident that their changes will not break the system. Or they may skip running all the tests and only check the compilation before checking in their changes.

Finally, the cumulative effect of long build times can have a negative effect on team morale. Nothing is more demoralizing then waiting an hour for your build to complete, only to find in the 59<sup>th</sup> minute that you forgot to uncomment a section of code critical to your bug fix. Fixing small incremental changes on long running builds can consume the better part of a team's day. Conversely, a solid, repeatable, fast build can give a team great confidence and speed.

All these negatives ultimately affect the project's bottom line and have a direct impact on the project's Return-On-Investment (ROI).

## **2 Background**

### **2.1 Build/Continuous Integration Defined**

Continuous Integration (CI) is the act of continuously integrating and merging people's code together into one common source repository. This is usually done with a fully automated build and test process that allows the team to test their software many times a day.

A core part of CI is compiling and executing of all the project code and their corresponding tests – or what we commonly refer to as the build. For the purposes of this article we will define a successful build as one whereby:

- All the latest sources are checked out of the configuration management system
- Every file is compiled from scratch
- The resulting object files are linked and appropriately packaged
- The system is started and a suite of tests are run against the system (all tests must pass 100%)
- All of these steps pass without error or human intervention

(<http://www.martinfowler.com/articles/continuousIntegration.html>)

The build is also the tools and artifacts used to actually do what we just described above. Because we are big fans of automated builds, we usually have an automated process running our builds (like CruiseControl) which execute our build scripts (Ant for J2EE or NAnt for .NET) and report back to us if there were any problems.

Before checking in any changes, developers are expected to run the build scripts against their local versions of the code. If the build passes, they are free to integrate their changes into the main repository.

## **2.2 How long is long?**

So what constitutes a long build? For my colleagues who build digital switches in the telecom industry, it is not unheard of for some builds to take days. Fortunately however, most applications we deal with at ThoughtWorks are not of this size. They are usually complex enterprise business applications – usually written in Sun Microsystem's J2EE or Microsoft .NET platform. Using these technologies, I have seen builds that run anywhere from a couple of minutes to a couple of hours.

Depending on the size of the project, teams will often have various stages of builds. A full system rebuild may include a complete re-building of the code base, execution of unit and acceptance tests, a database rebuild, and deployment to a remote server. A developer build may consist of rebuilding the code base, and execution of all unit tests.

When I say that all builds can be kept under 10 minutes in duration, I am referring to the build developers perform before checking in. This usually includes a complete recompile, and execution of all tests (unit tests at a minimum and potentially acceptance tests depending on how long they take to run). This would not include tasks that are not typically performed on a continuous basis (like a full system database rebuild).

Ten minutes is about all I can take before I begin to feel like I am being unproductive. In the early stages of the project, the build will often be much shorter (less than a minute). This is a real sweet spot because now team members can confidently make very small changes and check-in multiple times per hour.

## **2.3 About The Trouble Shooting Guide**

The principle that drove much of the discovery of the long build issues could best be described as high level profiling. When we profile a build, we measure how long certain build tasks take, and then investigate why they take so long. For those sections that do take a long time we have two options: make the operations themselves faster or do them less often.

From my experience, the number one culprit of long running builds is the accumulation of long running automated tests. Working for a company that puts a strong emphasis on automated testing, projects with hundreds (sometimes thousands) of unit and acceptance tests are not uncommon. Initially I did not find this to be a great concern as I would much rather have too many automated tests than none at all. Over time however, the impact of long running builds became too great to ignore.

It is with this focus on automated tests that much of the trouble shooting guide content is directed.

### 3 The Long Build Trouble Shooting Guide

Below is the summarized form of the Long Build Trouble Shooting Guide. Root causes of long running builds are listed on the left, with one or more potential solutions listed on the right.

**Table 1.** Long Build Trouble Shooting Guide

<b>Root Cause</b>	<b>Solution</b>
Slow hardware	Acquire faster hardware
Poorly written test	Re-visit original intent of test
Testing at the wrong architectural layer	Write tests at proper architectural layer
Network Intensive Tests	Stubs and mocks
Large code base	All the above Break application into sub domains Run tests in parallel Serialize the check-in process

The following section summarizes the root causes of long running builds and their respective solutions.

Running a long build on slow hardware does not aid the team in achieving continuous integration nirvana. *Acquire faster hardware* explains how improving your computer hardware can deliver great bang for the buck.

Applying the 80/20 rule, we can greatly improve our build time by focusing on the longest running tests first. *Re-visit original intent of the test* looks at how tackling these few troublesome tests first can go a long way to rapidly lowering our build times.

Periodically when writing tests the team will inadvertently write them at the wrong architectural layer. *Write tests at the proper architectural layer* discusses the importance of ensuring tests are written at the appropriate architectural layer and the impact not following this rule can have on our builds.

A distributed remote call across the network to another computer can be several orders of magnitude greater than a local one. When we have many unit tests that make distributed calls, a large portion of our build time is spent communicating across the network to other machines. *Stubs and mocks* show us how, in some circumstances, it is desirable for us to make the network disappear and more directly focus on the original intent of our test.

Large code bases may be a fact of life, but that should not automatically translate into long running builds. *Break application into sub domains* describes how large code bases may be broken into smaller independent ones. *Run tests in parallel* describes how running groups of unit tests simultaneously can save build time.

*Serialize the check-in process* does not actually reduce our build time, but it can prove handy when the consequences of breaking the build are too great to chance.

### **3.1 Acquire Faster Hardware**

Taking advantage of Moore's Law can be the fastest way of reducing your build time. Because of the importance in getting rapid integration feedback, make the build box run on the fastest machine you can obtain. Newer computers with faster CPU's and disk IO can have drastically reduce your build time with little relative upfront investment. While this will not solve all your problems, minutes can be rapidly shaved off the build time with very little effort.

The other aspect of hardware to consider with long running builds is the network. Many projects have saved considerable build times by simply moving their build boxes closer to their source code repositories. This can have a particularly significant impact if you have a large code base as the time it takes to download the code from the repository will be much less.

### **3.2 Re-visit Original Intent of Test**

When profiling long running tests, any test that consumed an abnormal amount of memory or network resources was one that garnered further investigation. Sometimes we found the test was simply poorly written in the first place. One test I remember in particular created a large collection of objects which were used in testing an overridden *equals* method on an object. When reviewing the intent of the test, it quickly became apparent that the test could be met with the same degree of confidence by using a much smaller collection. Re-writing this test alone shaved three minutes off of our one and a half hour build. A small but crucial first step was taken.

In other cases, the test itself had become redundant and obsolete. Two people were not pairing and wrote similar looking tests in different sections of the application. One could argue the root cause of this problem was more related to team communication and a longer build time was an indirect result. While true, redundant and non-essential tests still need to be removed.

Whatever the reason, the point is to revisit the test and determine how we can fix the problem that is causing the test to be long running in the first place.

### **3.3 Write Tests at Proper Architectural Layer**

Unit tests play an invaluable role on all our projects. As developers, we unit test everything our software does. As the application grows, and more tested functionality is added, we begin to accumulate a large number of tests.

Because we also have demanding customers that want proof the system is working, we also write customer or acceptance tests. These tests are different from unit tests (which are at the class method level). Customer tests are written at a higher level. They test the system as a whole, and give our customers confidence that the system is doing what it needs to do in terms they understand.

What can sometimes occur when aggressively testing our applications is that the line between a unit test and a customer test can begin to blur. In other words, we will sometimes accidentally write customer tests where a unit test would have been more appropriate.

The consequence of writing tests at a higher level than necessary is two fold. First, we lose valuable feedback at the unit level when something in our code breaks. In other words, it takes longer for us to isolate the source of the problem because we have more code to search through. Second, the customer tests typically take longer to run than their unit test equivalent. This is due to the fact that there is more code to execute (often making use of expensive network resources unnecessarily).

I was once on a project that had the most wonderful GUI testing framework. You would start the application, hit the record button on the GUI tester, and it would proceed to record all the button clicks and mouse events the user performed while using the application. Further, the framework allowed testers to make assertions about things they would like to see on the screen – like the color of a given widget, or the visibility of a given dialog. This framework was easy to use and made writing customer tests very simple and convenient.

This testing framework's greatest asset (its ease of use) was also its greatest liability. Developers stopped writing unit tests for their code because it was easier to use the GUI tester and record what they wanted to see happen on screen.

After a couple months of this, the team started to notice the build time was steadily creeping upwards. It turns out that the accumulation of all these GUI tests was having a very significant impact on the build time (each GUI test required restarting the application in a clean state). Our once light quick build was now exceeding an hour in duration.

When we write customer tests where a unit test would have been more appropriate, we are not writing tests at the appropriate architectural layer. For instance, we should avoid writing persistence tests in the presentation and domain layer. Put persistence tests where they belong – in the persistence layer. This does not mean that we never write tests that span architectural layers. It just means that we do not want the bulk of these tests outside the level they are directed towards. Having a nicely layered architecture and writing focused tests at these layers goes a long way to producing a quality product while helping us keep our build time in check.

### **3.4 Network Intensive Tests**

As Martin Fowler aptly reminds us, the first thing he recommends to clients who are building distributed applications is not to build distributed applications. Martin lists a variety of reasons for this, but the most important for the purposes of long build times is performance.

In the earlier stages of a project, when there are few tests that include network calls, the impact of calling a remote process on another machine (i.e. a database query) is relatively small. Indeed many small projects can quite regularly include network calls in their unit tests with impunity and not notice a significant impact on build times.

Large projects however, can not afford this luxury. There eventually comes a point on a large project where the impact of the unit tests chatting with remote services does begin to negatively affect the team's ability to continuously integrate.

If you are curious about how much of your build time is spent talking to network services fire up your favorite profiling tool and note which classes your build is spending most of its time in.

For those of you building J2EE applications, you may be surprised how much time is actually spent in `java.io.Socket`. On one project we discovered we spent as much as 94% of our unit test build time in this single class. This highlighted for us where most of our build time was being spent – chatting with the database across the network.

So what types of options do we have to minimize our network calls? Firstly, we can ensure that our tests are written at the appropriate architectural layer as discussed previously. Secondly, we can begin looking for ways to minimize the number of distributed network calls.

For example, a feature supported by platforms like .NET (and its respective database framework ADO.NET) is the ability to work with disconnected DataSets. ADO.NET allows us to store what looks like the result set from a database call locally on disk. This way when testing our code, we can load locally saved DataSets instead of making a remote call and fetching a new one.

With techniques like locally stored DataSets, we are avoiding the network by caching test data and results locally. Developers have been doing this for some time, although now it is nice to see the languages and frameworks providing native support for these features.

Other options include setting up databases locally on each developer's machine – thus eliminating the network call outright. If vendor specific database features are not included in the application, teams have also had success running fast in-memory databases locally and running the full production type database on their build boxes.

The rule of thumb with distributed network calls is to minimize them. I am not advocating not writing tests that communicate with databases and other remote services. When in doubt, write the test and worry about the build time later. For many smaller projects this will not even be an issue. For larger projects however, the importance of a layered architecture with properly focused tests minimizing network calls becomes critical in keeping the build time reasonable.

### **3.5 Stubs and Mocks**

Periodically, we are forced to use objects not directly related to the things we want to test. If these objects are expensive to create, their accumulated use will have a negative impact on our build time. Stubs and mocks objects can help us keep our tests focused and not necessarily rely on these expensive objects.

For the purposes of this article, I will define a stub as an object that stands in place of the real object for testing convenience (i.e. a database connection stub or message queue stub). Mocks are often used in a similar context to stubs, and there is often confusion between the two terms. One definition of a mock object is one that records state as it is being used in place of the real domain object. Testers can then query the mock and make various assertions regarding its state [Freeman].

While I have used mocks in the past, I have found stubs are more useful when removing external dependencies from my tests (which is my primary motivation when tackling long builds) and will hence forth focus on stubs.

One useful place for a stub is when the application needs to remove a problematic dependency on a service during testing. My colleague David Rice gives a nice example of a pattern solving this problem called *Service Stub* [Fowler]. In this pattern, David describes the frustration developers feel when writing tests against enterprise systems that are slow, problematic and unreliable. To remove the dependency, David recommends creating a *Separated Interface* [Fowler] so developers can have one implementation that calls the real service and one that is only a stub. Developers can switch between the two declaratively using a pattern like *Plugin* [Fowler].

Once the tests are no longer dependent on the service, continuous integration can proceed much more reliably, and the build time is reduced. The advantage of a layered architecture is that inserting stubbed out interfaces like this is relatively easy. Developers are not sitting around waiting for the external services beyond their control to come back on-line and they can avoid putting short term hacks into the code to work as temporary work a rounds.

### **3.6 Break application into sub domains**

One way of turning long running builds into faster ones is to reduce the size of the code base. This can be done by trying to see if there are any sub-systems, or *shared kernels* as Eric Evans calls them, that can be extracted from the main code base and made their own [Evans].

Not only does breaking large code bases into smaller pieces promote component reuse and module design, it reduces the amount of code and tests that need to be executed for the module under change.

Before breaking up a code base into shared domains, teams must ensure their code base is receptive to this type of extraction. Basic OO practices like encapsulation, high cohesion and loose coupling come into play here. Classes must also be correctly packaged, and not have any unnecessary external dependencies. See Bob Martin's paper on how to apply the Dependency Inversion Principle (DIP) for advice on how to structure your class packing and namespaces [Martin].

Before attempting to break the code into sub-domains, teams must also ask themselves if a shared kernel in sub-domain even exists. If a kernel is forced where one does not exist, teams will find working with the code base cumbersome and

awkward – largely because they will find themselves constantly needing to make changes to both code bases.

When faced with a large code base, consider breaking it into smaller pieces. If executed correctly, you may be able to reduce the build time and improve the design of your application simultaneously.

### **3.7 Run tests in parallel**

If long running unit tests are the sole cause of a team's long running build, running the tests in parallel may save the team build time. This technique involves breaking the unit tests up and running them concurrently with other unit tests.

I have seen some teams modify their builds so that all the tests run concurrently in separate threads on the same build box (they were fortunate enough to have access to a Sun E10K). Those who have access to less powerful machines may opt to break up the tests and run them on multiple build boxes concurrently.

While breaking up the build along these lines is a reasonable start for reducing the build time, this technique will only take you so far. If you still have loads of development, and the project is not near completion, I strongly recommend teams look at the root cause of why their builds are taking so long, and apply solutions described within this paper and others.

### **3.8 Serialize the check-in process**

Sometimes despite our best efforts, at the end of the day we still have builds that take longer than we would like. There is nothing worse than trying to make a high pressure deadline and not being able to check-in your code because the build is broken. Not only are you prevented from checking in your code, but everyone else on the team with changes is also held up.

One way of minimizing the chances of breaking the build and avoiding all integration conflicts, is to serialize the check-in process. By serialize I mean only one party checks in code at a time. This is different from the more normal free flowing practice whereby any developer can optimistically check-in as soon as they have run the build locally and all tests pass. By strictly controlling the manner in which team members can check-in code, we greatly reduce the possibility of the build breaking.

Note this technique does nothing to reduce our build time. Instead, it manages the impact of the long build time on the team by ensuring the build never breaks.

A list or queue is usually sufficient to keep track of who is next in line to check-in in. One team I worked with had a token they passed around indicating who had check-in in privileges. As soon as "Billy Bass" the singing fish broke into a rousing rendition of "Don't Worry Be Happy" you knew that someone had just successfully integrated their changes in to the system and the token was passed to the next in line.

While I am not a big fan of check-in serialization, it can serve as a stop gap until a more permanent solution can be applied to fixing the root cause - the long running

build itself. Slowly applying some of the techniques described earlier can eventually bring long running builds down to more reasonable levels. Each team will have to ask themselves if the loss of time due to serializing the check-in in process, outweighs the loss of time spent fixing broken builds.

## 4 Summary

Long build times prevent teams from receiving the timely feedback and should be avoided at all costs. While projects with large code bases are usually more susceptible to long builds, small projects can be negatively affected if they are not vigilant. Builds exceeding ten minutes in length must be monitored closely. By focusing on those areas of the build that take the longest (usually automated tests) and applying techniques described herein, long builds can be brought down to more manageable times.

## Acknowledgements

This article would not have been possible without the help and support of many people. I would like to thank Owen Rogers, Jason Yip, Brad Marlborough, Joe Walnes, Martin Fowler, Kerry Todyruik, and Tannis Rasmusson.

## About the Author

Jonathan Rasmusson is a Computer Engineer with ThoughtWorks Canada. He enjoys building enterprise applications, and searching for better ways to write software. Areas of interest include Agile development methodologies and exploring the very human side of software development. He received a BS in Electrical Engineering and a MS in Computer Engineering from the University of Alberta, Canada. Jonathan Rasmusson can be reached at ThoughtWorks, 805-10<sup>th</sup> Ave SW, 3<sup>rd</sup> floor, Calgary, Alberta, Canada T2R 0B4; jrasmusson@thoughtworks.com

## References

1. E. Evans, *Domain Driven Design*, Addison-Wesley, 2003.
2. M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003
3. S. Freeman, T. Mackinnon, P. Craig, *Endo-Testing : Unit Testing with Mock Objects*, Extreme Programming Examined, Addison-Wesley, 2001.
4. R. Martin, *Agile Software Development – Principles, Patterns and Practices*, Prentice Hall, 2003.
5. <http://c2.com/cgi/wiki?MooresLaw>