Enterprise Continuous Integration Using Binary Dependencies

Mike Roberts

ThoughtWorks, Ltd., Peek House, Eastcheap, London, UK
mroberts@thoughtworks.com

Abstract. Continuous Integration (CI) is a well-established practice which allows us as developers to experience fewer development conflicts and achieve rapid feedback on progress. CI by itself though becomes hard to scale as projects get large or have independent deliverables. Enterprise Continuous Integration (ECI) is an extension to CI that helps us regain the benefits of CI when working with separately developed, yet interdependent modules. We show how to develop an ECI process based upon binary dependencies, giving examples using existing .NET tools.

Keywords: Continuous integration, scalability, tools and techniques, .NET

1 Continuous Integration – A Review

Kent Beck defines Continuous Integration (CI) by stating 'No code sits unintegrated for more than a couple of hours. At the end of every development episode, the code is integrated with the latest release and all the tests must run at 100%.' [1]

Automated CI [2] takes much of the CI effort away by running an automated build every time a developer commits a change to version control (see ‘Ubiquitous Automation’ [3]). Automated CI is implemented by using a dedicated CI build server tool like CruiseControl [4] or CruiseControl.NET [5].

Both of these processes assume you have a single source tree which is developed as one advancing ‘code line’ [6].

Unfortunately, there can be scalability issues with this. While describing CI, Kent Beck states ‘If integration took a couple of hours, it would not be possible to work in this style ... You also need a reasonably complete test suite that runs in a few minutes' [1]. For a medium to large sized project (e.g. upwards of 5000 classes) a full build can take an hour to complete when compilation, unit testing and acceptance testing are included. This is long enough to significantly break up the development flow of a project using CI.

There can also be business concerns with forcing a large development effort onto a single source base with unified build and release timelines. Consider a client/server application that has a server layer communicating not only with the client GUI but also with other external applications. The release schedule for those external applications places requirements on the server code that do not exist for the client code. Thus, there is a need to decouple the GUI and server development efforts. However, if the client code needs the server code to compile, the client build must be able to find and reference the server code for each of its builds. Finally, you may decide to break up your application into different ‘bounded contexts’[7] when it makes sense to have semi-independent domain models within your application.
All of these issues point to the same thing - sometimes the ideal approach of developing with one tightly bound source tree doesn’t work out and we need to introduce extra processes to help.

2 Breaking up the Build by Introducing Binary Dependencies

One way to start addressing the above issues is to separate out the source tree into different modules, each with their own independent build and CI processes. Each module uses pre-built binary versions of any other modules it depends on.

We’re going to use the client/server decoupling example from section 1 as a common example thread throughout the rest of this paper. We’ll start resolving it by applying this binary separation idea.

Assuming the application has a layered architecture [8], its source code should be easily split into client, server and ‘common’ code. We can decouple the development of the client and server layers by moving the source that is specific to the client into a separate module in version control, leaving the common and server code in the original module (which we call simply the server module from now on.)

The client code requires the server code in order to compile. As a ‘first cut' implementation to get the client building, we can include a pre-built binary version of the server module in the client's version control tree. We also setup separate CI servers to build each of the 2 modules.

This technique by itself is nothing new, but we now consider how we can extend Continuous Integration techniques to such separated projects.

3 Enterprise Continuous Integration

By itself, the above separation process has a flaw. With the separated client and server modules, as soon as a developer commits code to the server module, the client module is building against an old version of the server code. In other words, the new server code has not been integrated with the client code. Despite not having a unified build and source tree, we can still apply the principles of Continuous Integration to the complete application.

We define Enterprise Continuous Integration (ECI) as the process of integrating 2 separated but dependent source trees whenever code changes in either of the 2 trees.

ECI allows us to continually integrate separated modules as if they were developed as one module.

3.1 Designing an Enterprise Continuous Integration Process

Reviewing our client/server example:

- We have 2 separated modules in version control, one for the client and one for the server
- Each module has its own CI process that builds the contents of version control and produces a versioned binary distribution