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
- The client source tree includes a built version of the server module

The next step is to add an Enterprise Continuous Integration (ECI) process that will attempt to build the client module with the latest binary version of the server. This is **in addition** to the existing CI process that just builds what is specified by the client's build script.

3.1.1 Specifying dependency versions

To setup such an ECI process we need a way of varying which version of the server module the client is to build against. The first step to implementing this is to **publish** the built versions (or distributions) of the server module to a file server. It needs a structured directory layout, including the ability to locate distributions by both version number, and *latest* logical tag.

We can now update our client build to fetch a specified version of the server module from the build server before building, rather than keeping a fixed copy within the version control tree. The version of the client build script checked into the source tree would always default to use a *last-known-good* version of the server that we have successfully integrated with the client. However, the ECI process **overrides the server version to *latest*.**

We'll see concrete examples of all of this later on.

3.1.2 When to integrate

The next question is when do we integrate? With normal CI, we perform an integration run whenever the source code changes, since that is the only changing input of our integration process. However, our client build now has a 'latest server build' that can also change, so we should perform an ECI run whenever there is a change in either the source code we are integrating, or the binary dependencies upon which the source code depends.

3.1.3 What to do on a successful ECI build

The client's standard CI build is already responsible for producing a release-ready distribution and corresponding source label, so what can you usefully do with a successful ECI build? It's always good to know when everything is working together, so marking the client source with an appropriate label is a good practice. You can

automate it so it's zero effort, and in most modern Source Control systems labeling is a cheap, and fast, operation.

However, you know that the client build now passes all of its tests against a new version of the server, so it's also useful to **automatically update** the client's *last-known-good* server version so that developers, and future client builds, are up-to-date with the server version.

3.1.4 What to do on a failed ECI build

There are 2 possible causes of an ECI build break:

- The source module (the client module in our example) is internally broken.
- There is a discrepancy between the source module and the latest versions of the dependencies.

The first of these should also be picked up by standard CI processes. If an ECI build fails in this way we should check that the standard CI process has failed in the same way.

Breakages of the second kind are the feedback that ECI provides beyond single-module CI. There are various reasons why such a situation can have occurred:

- A compilation error may indicate a change in the interface of the server module. In this event, the development team could consider using deprecation cycles to avoid breaks between modules.
- A breaking test could indicate that the client code was relying on 'accidental behavior' of the server code. In this case the client code should be updated.
- A breaking test could also expose an untested part of the server code. In this case the server module would need updating, preferably including a new test that would simulate how the client code had broken the old code.

3.2 Versioning

So far we have made a few assumptions with respect to versioning:

- We do not need to worry about the versions of *chained* dependencies (e.g. the dependencies of the server module itself.)
- Versions of the server module increase linearly, with no branching of versions.
- If the server module is branched, it is always appropriate for the client to build against the *trunk* version of the server, rather than against a stable branch.

The first of these is a complicated area beyond the scope of this paper. A solution to it would allow us to perform binary dependency-based ECI for scenarios where we'd like any module in a complex dependency tree to cause an integration attempt for all dependent modules.

The second two points do not require assessment if dependency modules are never branched, but if they are we have some decisions to make. We'll have an introductory look at this area in the rest of this section.

3.2.1 Aside: Continuous Integration & Branching

Extreme Programming steers towards a model of continual release, and source tree branching is not required in such an ideal model. However, due to business concerns many agile development projects can't release to the actual customer at the end of every iteration (especially if iterations are 1 or 2 weeks long.) Typically a development team will construct a release branch for fixing any bugs that may appear in the release, but still be able to carry on continual development on the trunk.

In such a case, it is worth using the same CI process on the release branch that is used on the trunk, e.g. to use the same automated build, testing, and distribution techniques. However, if the CI process publishes distributables and performs labeling, how do we perform CI for both the trunk and the branch in a non-conflicting manner?

A good answer is to do the following:

- Use different CI instances for **each** code line.
- Use an appropriate **version numbering scheme** so that the distributables and labels produced by each CI instance are distinguishable from each other.

3.2.2 Targeting a project at a branched dependency

In our ongoing example, it may be necessary to target the client code at a branched version of the server module. When branching the server, we would implement 2 standard CI processes (one for the branch and one for the head.) The branch CI process should publish a *'branch-latest'* distributable and the ECI process for the client module should be updated to use this branch-specific version, rather than the latest trunk version.

3.2.3 Ranged Versions and Published Interfaces

What we have done above is to create a **ranged version.** E.g. if the branch of the server defined the *1.2* version range of the module, we are saying that the client module should be able to build against *1.2.** (any 1.2 version) of the server.

The server trunk could now be considered the *1.3* version range. The differences between 1.2 and 1.3 may include an update of the 'published interface' [9] of the module.

4 Example - Implementing Enterprise CI in .NET

Now we have a design for ECI, how do we implement it? For Java and .NET the tools already exist since we can use standard CI and build applications. In .NET specifically we can use *CruiseControl.NET* [4] and *NAnt* [10]. There are various other

.NET build and CI tools (*Draco.NET* [11] and *Hippo.NET* [12] are alternative CI tools, and *MSBuild* is an alternative build tool to be released as part of .NET 1.2)

We will follow on with the client / server example and will assume that the client depends on a '1.2' branch of the server. We use *NAnt* and *CruiseControl.NET* as our build and CI tools.

4.1 Defining the Distribution File Server Directory Structure

We are implementing ECI using binary dependencies, so let's start off by setting up our dependency distribution file server structure. Below is a directory tree that would be created by the 3 individual 'atomic' CI instances (1 for the client, 1 for the server's *1.2* branch, and 1 for the server's *1.3* trunk).

```
\\DistributionFileServer\
+--> Server
|     +--> 1.2.455
|     |     +--> server.zip    distribution file
|     +--> 1.2.456
|     +--> 1.2.457             the last successful 1.2 build
|     +--> 1.2.latest          always the last successful 1.2 build
|     +--> 1.3.20
|     +--> 1.3.21              the last successful 1.3 build
|     +--> latest              always the last successful trunk build
|
+--> Client
      +--> 1045
      |     +--> client.zip    distribution file
      +--> 1046                the last successful client build
```

4.2 The client build script

We now setup a *NAnt* build script for the client. NAnt uses *targets* to define actions to happen during the build. Our build script needs a target to retrieve dependencies (*get-dependencies*), and a main target (*all*) that makes sure this happens before the rest of the build occurs. The *server-version* number is specified in a *property*, and this can be overridden by the environment calling the NAnt script. The *server-version* property enables us to specify exactly which server distribution file to use.

```
<project name="client" default="all">

    <property name="server-dist-location"
value="\\DistributionFileServer\Server"/>

    <property name="server-version" value="1.2.456"/>

    <property name="server-dist-name" value="server.zip"/>


    <target name="get-dependencies">

        <mkdir dir="dependencies\server"/>
```

```
        <unzip zipfile=="${server-dist-location}\${server-
  version}\${server-distname}" todir="dependencies\server" />

    </target>


    <target name="all" depends="get-dependencies, compile, test,
  deploy, dist"/>

    <!-- .. Other targets would go here.. -->

  </project>
```

4.3 The CruiseControl.NET config file for the ECI build

Now we can create a *CruiseControl.NET* instance for our ECI build. We do this by setting up a configuration file like the following. It has 2 critical sections:

1. A *sourcecontrol* section which defines where to check for modifications. We look in 2 locations – on the *filesystem* to check for server version 1.2 changes and in *cvs* to check for client changes.
2. A *build* section which defines what to build when a change is detected. It runs the NAnt build tool, and specifies the client project's build directory and build script (which configured in the previous section). Importantly it overrides the *server-version* property to always use the *latest* version of the server.

It is the check of the server distribution directory, and the override of *the server-version* property that would differentiate this from the client's normal configuration.

```
<cruisecontrol>
  <project name="ClientECI">
    <sourcecontrol type="multi">
      <sourceControls>
        <filesystem>
<repositoryRoot>\\DistributionFileServer\Server\1.2.latest</repositoryRo
ot>
        </filesystem>
        <cvs>
          <executable>c:\tools\cvs-exe\cvswithplinkrsh.bat</executable>
<workingDirectory>c:\localcvs\myproject\client</workingDirectory>
        </cvs>
      </sourceControls>
    </sourcecontrol>
    <build type="nant">
<executable>c:\localcvs\myproject\client\tools\nant\nant.exe</executable
>
      <baseDirectory>c:\localcvs\myproject\client</baseDirectory>
      <buildArgs>-D:server-version=1.2.latest</buildArgs>
      <buildFile>ccnet.build</buildFile>
      <targetList>
        <target>build</target>
      </targetList>
    </build>
```

```
    <!-- Other CCNet config would also appear as normal -->
  </project>
</cruisecontrol>
```

5 Other Solutions

5.1 Continue to use Atomic Code Lines

Our motivations for Enterprise Continuous Introduction were 2 possible issues that can occur in medium-large development projects:

- Build process too slow
- Requirements for separated delivery of different components

The best solution, if possible, may well be **not** to separate out code lines. Enterprise Continuous Integration adds extra process to your team and so if (for example) you could actually shorten your build times by reworking your tests, etc., then this would be preferable. We use several techniques for this in ThoughtWorks. One related technique is to have 2 separate CI builds for one code line: one an 'express build' that just runs unit tests to give a basic idea of the success of an integration; another a longer 'full' build that actually runs database processes, acceptance tests, deployments, etc.

5.2 Enterprise Continuous Integration using separated source code lines

A very different approach that some of my colleagues at ThoughtWorks have used successfully on large teams is to not separate out the project into binary dependencies, but instead to give different teams separated source areas (either on separated branches or in separate source control servers.) Each team has its own CI process for the code they are working on, but there are also ECI processes that attempt to integrate the entire project's code (both into and from each team's code line.)

A similar approach is Gump [13] which tries to build the latest source versions of various projects against each other.

6 Further Work

We have seen a design and corresponding implementation in .NET for Enterprise Continuous Integration which will work for many scenarios. However, we have not addressed the issue of 'chained dependencies', and specifically what happens when the versions of chained dependencies change. This area requires further work. .NET supports ranged assembly version specification, so it is possible that this may be of use in a .NET implementation.

Other areas affecting versioning that are worthy of investigation include:

- Is it worth thinking about the difference between build- and run-time dependencies?
- What is a convenient way of expressing versioned dependency requirements in build scripts and deployment artifacts?

Maven [14] includes some solutions towards these problems. It offers a way for projects to define their structure and dependencies, and from this definition 'builds' the project to produce various artifacts. It also publishes and downloads built projects using well structured, versioned repositories.

Apart from versioning, we could also address the following:

- For projects consisting of lots of separated modules, would it be worth introducing modules just for the basis of integration?
- What tests should we run in an ECI build? Can we optimize the ECI build by only running specific tests based on which dependencies have changed?

7 Summing Up

Extreme Programming defines a very useful set of practices and values that can be used throughout agile software development, including the practice of Continuous Integration. In this paper we have explored one way to solve scalability issues with Continuous Integration by splitting up a project into several modules, and then using Enterprise Continuous Integration (implemented with existing tools) to still gain the feedback that single-project CI provides.

References

1. Beck, K.: Extreme Programming Explained, Addison Wesley (2000)
2. Fowler, M., Foemmel, M., : http://martinfowler.com/articles/continuousIntegration.html
3. Hunt, A., Thomas, D.: The Pragmatic Programmer, Addison Wesley (1999)
4. CruiseControl: http://cruisecontrol.sourceforge.net/
5. CruiseControl.NET: http://ccnet.thoughtworks.net/
6. Berczuk, S., Appleton, B.: Software Configuration Management Patterns, Addison Wesley (2003)
7. Evans, E: Domain-Driven Design, Addison Wesley (2004)
8. Fowler, M.: Patterns of Enterprise Application Architecture, Addison Wesley (2003)
9. Fowler, M: http://martinfowler.com/ieeeSoftware/published.pdf
10. NAnt: http://nant.sourceforge.net/
11. Draco.NET: http://draconet.sourceforge.net/
12. Hippo.NET : http://hipponet.sourceforge.net/
13. Apache Gump : http://jakarta.apache.org/gump/
14. Apache Maven : http://maven.apache.org/