Microsoft

Customising CruiseControl.NET

Going Beyond the Box | Craig Sutherland

Table of Contents

Introduction

What is CruiseControl.NET?

CruiseControl.NET was primarily designed as a Continuous Integration (CI) server. A CI server is a piece of software that will continuously monitor one or more source control repositories and automatically trigger a build if any changes are detected.

But CruiseControl.NET is so much more than just a CI server – at its core it is a highly customisable scheduling engine. While CruiseControl.NET provides for continuous integration out of the box, it can be customised to perform any type of automated processes.

What Does This Document Cover?

This document covers how CruiseControl.NET can be customised. There are two components of CruiseControl.NET that can be customised: the server and the dashboard.

Most of the customisations for the server are at the project level. A new item can be added and configured – CruiseControl.NET will pick it up automatically. The following project items can be added to the server:

- Tasks/publishers
- Triggers
- Source control blocks
- Labellers
- State managers

Additionally, in CruiseControl.NET it is possible to build server extensions that use a deeper level of integration. These extensions allow changes to the core functionality of the server.

The dashboard is mainly customised by either changing the existing templates or adding new plug-ins. Plug-ins are mainly used to display additional information, but can also be developed to interact with the server.

This document starts with a high-level overview of the components that make up CruiseControl.NET and how they interact. While this is not necessary, it does help to show how everything fits together and how the individual customisations will have an effect.

The subsequent sections then delve into the different customisations that can be done. These start with the various project items, move onto server extensions and finally finish with the dashboard customisations.

Then this document moves with details on some of the common components. These can be used in many different parts of the system, but are not necessary.

Finally some tips and tricks are covered for developing customisations.

Versions

This document is written for CruiseControl.NET 1.5. However many of the underlying principals are still relevant for version 1.4. Where a feature has changed or is new, it will be noted.

About the Author

The author, Craig Sutherland, is one of the developers on the CruiseControl.NET project. He has been involved with the project since 2008 and has contributed several major enhancements to the codebase (including security, server extensions and parameters).

OVERVIEW

THE BIG PICTURE

CruiseControl.NET does not consist of a single component – instead there are multiple components that work together to make the entire system. To additionally complicate things, these components can be on different machines.

At a high-level there are four parts to the system:

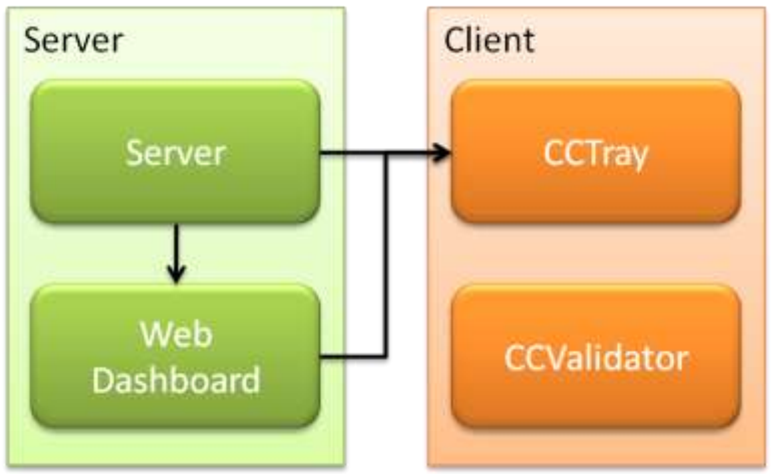

Figure 1: High Level Overview

The server and web dashboard both reside on a "server" machine. These are both automated components that do not require direct user interaction. In contrast the "client" components, CCTray and CCValidator, are designed to be used directly by a user.

The server provides the core of CruiseControl.NET. This is the actual workhorse of the system.

The web dashboard is an ASP.NET application that exposes server information to the external world. It enables a user to see what is happening on the server and drill down into the results.

CCTray is a client-side component that allows a user to monitor one or more servers. It is a WinForms application that has very few extension points. As such it is not covered in this document.

CCValidator is a tool for checking the validity of a configuration file. As such it is not directly part of the system, but it is a very useful tool for checking configuration files. It uses the same underlying binaries as the server, so it will also detect any new project items. However it does not have any direct extension points, so it will not be covered in this document either.

THE SERVER

Internally the server consists of a number of parts that work together. At a high level these parts are:

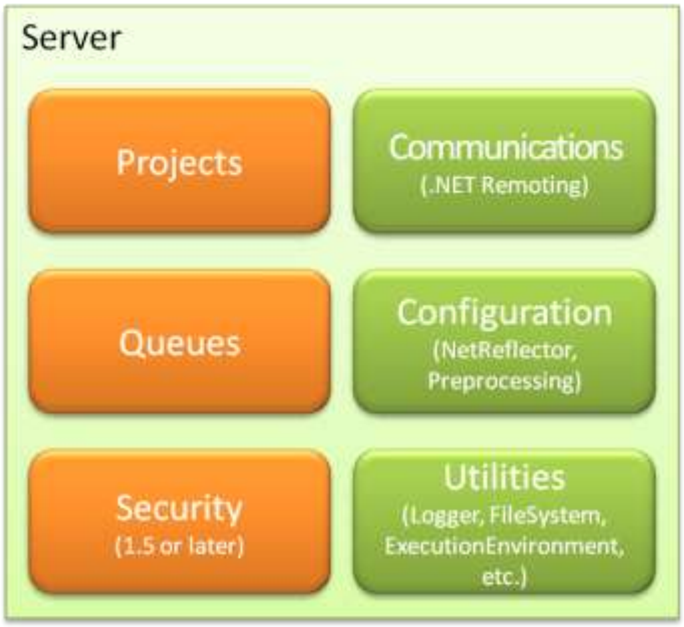
Server
Projects
Communications (.NET Remoting)
Queues
Configuration (NetReflector, Preprocessing)
Security (1.5 or later)
Utilities (Logger, FileSystem, ExecutionEnvironment, etc.)

Figure 2: Server Components

The components in orange are directly loaded from configuration – these are typically the items that can be extended.

The green components are the extra bits and pieces that interface with the external world. These generally cannot be extended, but they can be used by any extensions.

Natively the server only supports one communications protocol – .NET Remoting. External clients (e.g. CCTray) appear to connect via HTTP, but this is routed via the dashboard and not handled natively by the server. In 1.5 or later it is also possible to add a new communications protocol via a server extension.

Internally the server uses one thread for the application (the server instance), plus one thread per started project (see below). Additionally, any incoming communications requests are handled on a new thread.

Each project that runs must be associated with a queue. If the project is not explicitly associated with a queue, then an implicit queue will be created for the project.

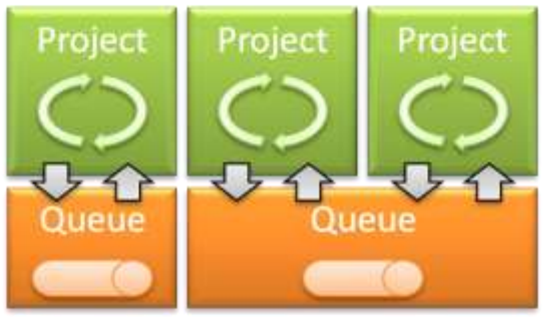

Figure 3: Project-queue interactions

The project has an internal check to see when it should run. When a build is triggered for a project (see The Project below) the project sends a message to the queue saying it is ready to start. The queue receives this request, adds it to its internal store. If there are no requests pending it sends a commence build message to the project. If there is a project already building, then the queue does not send a message, instead it just leaves the new request in its data store.

When a project finishes building it sends another message to the queue saying it has finished. The queue removes the request from its data store and checks if there are any other projects pending. If there are pending projects, it sends a commence build message to the next project in the queue.

If a project request for the project already exists on the queue, it will ignore the request.

In 1.5, a security system was added to the server. This works in a slightly different way from the queues. Security fits in-between the projects and the communications layer:

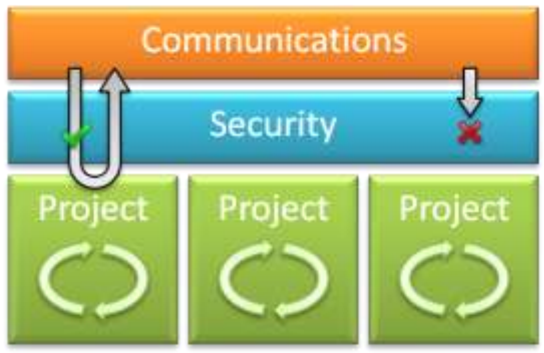

Figure 4: Project-security interactions

When a request comes in from a client (via the communications channel), the security check validates the security credentials. If the credentials are valid, then it checks if the user has the required permission to perform the action. If so, then the request is passed onto the project. Otherwise the request is denied.

If the server does not have any security, then both the credential and permission checks are still made, but they are automatically passed.

The Project

Loading

The projects themselves are defined entirely in the configuration. When the server starts it loads the configuration into memory. This is a two-step process involving first a pre-processor phase, and then using an external library called NetReflector.

The pre-processor phase converts the XML format into an expanded XML format. As this cannot be customised, it is not covered in this document.

The NetReflector processor phase converts the XML format into POCO (Plain Old Class Object) instances. The processor uses attributes within the class definitions to define how the configuration is converted. Once the configuration has been loaded, it is then validated – see below for details.

When the project is loaded it is held in memory – but nothing actually happens until the project is started. When the project is started a new thread is started and the project triggers are polled every second.

BUILD SEQUENCE

At a high level a build (called an integration internally) consists of seven sequential phases:

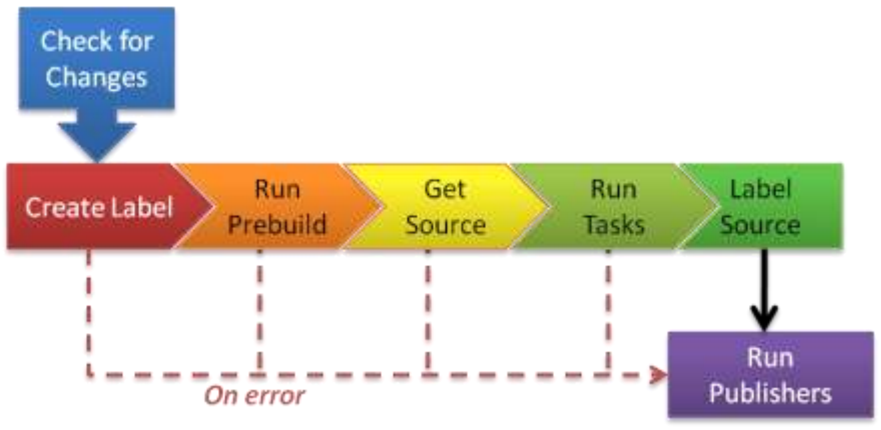

Figure 5: Build phases

At the start of every build is a check for any changes. If there are changes, or a force build has been triggered, the build runs through five phases in sequence: create label, run prebuild, get source, run tasks and label source. Each of the phases only occurs if the previous phase was successful, otherwise the sequence is terminated. The final step of a build, regardless of any errors, is to run the publishers.

In more detail, an actual build is a complex set of interactions between three main classes:

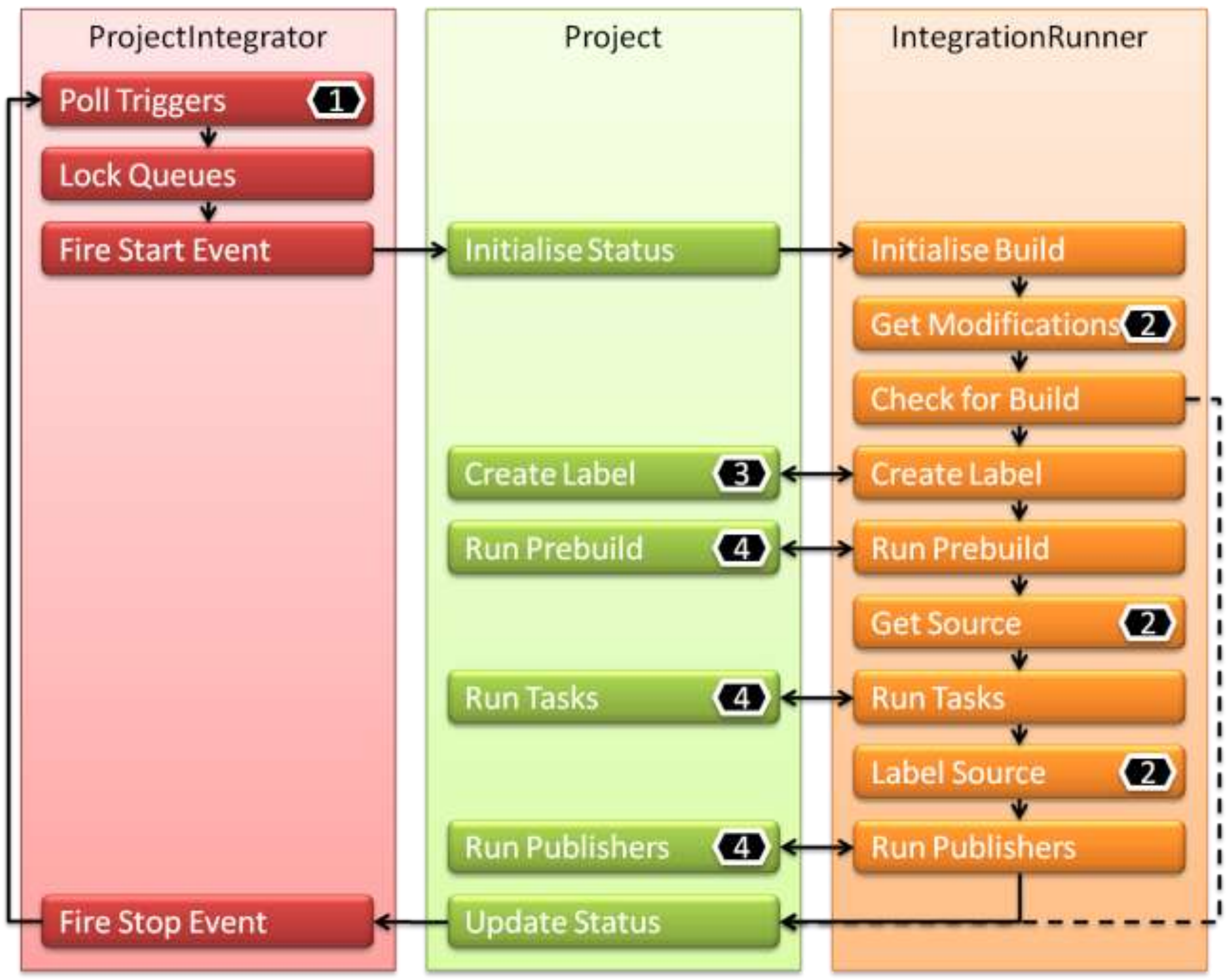

Figure 6: Detailed build sequence

As well as the three main classes, there are a number of external classes that are also used to perform specific tasks. These classes are:

1. Triggers (`ITrigger`)
2. Source control block (`ISourceControl`)
3. Labellers (`ILabeller`)
4. Task/publishers (`ITask`)

The `ProjectIntegrator` holds the thread for the project. It handles starting and stopping, plus kicks off the actual build. As such it is the starting point for each build.

Every second the polling thread checks the triggers. It does this by iterating through all the triggers (external #1) and sees if any of them have been set on. If none of them are set, then the process is cancelled and it waits for another second.

Once a trigger has been set, it negotiates with the queues to see when it is allowed to run. This involved both checking that there are no other projects ahead of it in the queue (if there are it will pause until there are not) and locking any other queues.

Once the project is approved to run, it will fire off a build starting event. This is an extension hook that server extensions can intercept to pause or cancel a build (if required). Finally it is now ready to hand over to the project for the build.

However, even at this point a "build" has not officially started. The `Project` initialises the status (both for the project and the individual items within the project) and then hands over to the `IntegrationRunner` for the actual build co-ordination.

The `IntegrationRunner` then performs the final initialisation for a build (starts the required status objects, creates folders, etc.) Then it asks the source control block (external #2) for a list of any modifications that have occurred since the last build.

Once the modifications have been retrieved, the runner checks to see if the conditions necessary for a build have been met. These are either a trigger has fired a force build or there are new modifications. If either of these conditions is met, then a "build" officially starts. Otherwise everything is cancelled (dashed line).

Once the "build" is official, the runner and the project co-ordinate their way through a series of steps:

1. Create a label (via external #3)
2. Run any pre-build tasks (via external #4)
3. Retrieve the source code (via external #2)
4. Run the build tasks (via external #4)
5. Label the source code (via external #2)
6. Run the publishers (via external #4)

Each of these steps is performed by an external rather than the runner or the project directly, but both the runner and the project provide the infrastructure around the externals (logging, error handling etc.)

If an error occurs in steps 1 to 5, the remainder of these steps are cancelled and the publishers are run directly. The publishers (step 6) will always run once a build is official – no matter what has happened (unless there is a catastrophe failure that crashes the entire server!)

Finally, when all the publishers have run, the project updates the status and returns control to the integrator. This fires an event to let any server extensions know that the build has finished and returns to the polling cycle.

THE DASHBOARD

THE LIBRARIES

Finally, to round off the big picture, here are the libraries that make up the CruiseControl.NET system:

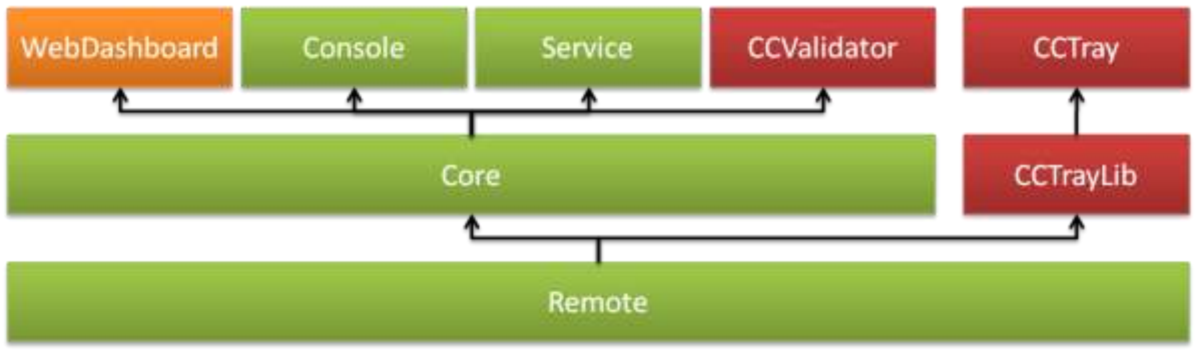

Figure 7: The libraries of CruiseControl.NET

The base for everything is Remote. This library provides the communications framework for the various parts to communicate together. It also contains a number of common interface definitions and the exceptions.

CCTrayLib and CCTray form their own un-related branch. This is functionality that is specific to the client and has nothing to do with the server code.

On the server side, Core forms the base for both the actual applications (Console and Service) as well as for the WebDashboard. This provides most of the functionality for the service, including the pre-defined tasks/publishers, triggers, source control blocks, etc.

Console is a command-line version of the server that can be run from a console, while Service is a Windows NT service version. Both of these are shells, they provide the infrastructure necessary to run as either a command-line or service and all the actual functionality is handled by Core.

WebDashboard is the ASP.NET application for exposing information via a web interface. It shares some of the functionality from Core to reduce the amount of duplication.

Finally CCValidator shares Core so it can access all the definitions for the configuration. It does not perform any server functionality.

Project Item Basics

Required Libraries

There are three assemblies that are required to build a custom item. These libraries are:

- NetReflector.dll
- ThoughtWorks.CruiseControl.Core.dll
- ThoughtWorks.CruiseControl.Remote.dll

NetReflector.dll is used for converting from XML to POCO instances. It contains a number of attributes that are used to mark classes and properties so they can be read. Basically it is used to define what can be configured.

ThoughtWorks.CruiseControl.Remote.dll provides some base classes and enumerations. These are typically used by the custom items.

ThoughtWorks.CruiseControl.Core.dll provides the definitions and base classes for the project items. Additionally it also has a number of utility classes that can be used.

Assembly Name

In order for CruiseControl.NET to detect the plug-in the assembly needs to have a name with a specific format. The format is "ccnet.*yourname*.plugin.dll", where *yourname* is the name of your plug-in.

For example, all the demos for this document are in an assembly called "CCNet.CSharpDemos.Plugin.dll".

This assembly needs to be in the same folder as the server assemblies.

NetReflector

Basics

NetReflector is a third party component that is used by CruiseControl.NET to handle the configuration serialisation. Basically it will convert an XML document into POCO instances and vice versa. This is what is used to load the configuration.

NetReflector works in a similar way to XML serialisation in .NET by using attributes to drive the process. However NetReflector uses a form of dynamic discovery to load the items (in XML serialisation the types need to be hard-coded!) This means the new items can be added to the configuration without requiring any modifications to the existing code.

However there is one down side with this approach – each item name must be unique within CruiseControl.NET. If there are two tasks called "fooTask" then NetReflector will throw an exception!

CruiseControl.NET uses two of the attributes available from NetReflector – `ReflectorTypeAttribute` and `ReflectorPropertyAttribute`. There are some other attributes available, but most of them are just duplicates of

ReflectorPropertyAttribute and so do not need to be used (the main exception is ReflectionPreprocessorAttribute but that will be covered later.)

Attribute Usage

The ReflectorTypeAttribute attribute is used to register a type with NetReflector. This is applied at the class level and has a string parameter – the name of the item. As noted previously, each name must be unique. Attempting to register the same name twice, even if it is in a different assembly, will throw an exception.

The ReflectorPropertyAttribute attribute is used to expose a property of a type. NetReflector will only load properties that have been marked with this attribute – all other properties will be ignored. Like the ReflectorTypeAttribute attribute it needs a name – this is the name of the item in the XML.

For example, if we have the following configuration:

```xml
<demoTask name="A Demo">
  <author>Craig</author>
  <items>
    <demoTask>
      <name>Inner Demo</name>
      <isNonsense>true</isNonsense>
    </demoTask>
  </items>
</demoTask>
```

Code 1: Example XML file for NetReflector

We would need to have the following attributes to load it:

```csharp
[ReflectorType("demoTask")]
public class DemoTask
{
    [ReflectorProperty("name")]
    public string Name { get; set; }

    [ReflectorProperty("author", Required = false)]
    public string Author { get; set; }

    [ReflectorProperty("age", Required = false)]
    public int Age { get; set; }

    [ReflectorProperty("isNonsense", Required = false)]
    public bool IsNonsense { get; set; }

    [ReflectorProperty("items", Required = false)]
    public DemoTask[] InnerItems { get; set; }
}
```

Code 2: Class to load example configuration

NetReflector can handle a wide variety of types – value types (including enumerations), classes, arrays, lists, etc. NetReflector does not care whether value types and strings are stored as attributes or elements. However all other types must be stored as elements.

In the example, the name is stored as an attribute on the root element, but an element below – either is perfectly valid as NetReflector does not care. We also see that it handle both strings, integers and boolean values.

The example also has an array of items. When NetReflector attempts to load the configuration it will generate an internal array of values. It does this by looking up the element name (e.g. demoTask in the example) and then generates an instance of that class. This means if we were to derive another class from demoTask (say betterDemoTask), NetReflector would know how to load it.

Configuration Options

As well as the name, the property attributes have a few other properties that can set. The most common property is the `Required` property. By default all reflected properties are required – if the configuration does not have it an exception will be thrown. Adding a `Required = false` property to the attribute makes the element optional.

The other properties available that influence loading are `InstanceType` and `InstanceTypeKey`. `InstanceType` is used for setting a default type on a singleton value. This would be used for the following type of configuration:

```
<demoTask name="A Demo">
  <author>Craig</author>
  <child name="Child Demo">
    <age>11</age>
  </child>
</demoTask>
```

Code 3: InstanceType XML example

`InstanceTypeKey` is also used for singleton values, but where the type can be configured by the user. For example:

```
<demoTask name="Demo #3">
  <author>Craig</author>
  <typedChild name="Child Demo" type="demoTask">
    <age>11</age>
  </typedChild>
</demoTask>
```

Code 4: InstanceTypeKey XML example

Pre-processing

The other attribute that is sometimes used is `ReflectionPreprocessorAttribute`. This is used to allow a class to pre-process the XML before NetReflector tries to load it. This can only be placed on a method with the following signature:

```
XmlNode PreprocessParameters(NetReflectorTypeTable typeTable, XmlNode
inputNode)
```

Code 5: Pre-processor signature

When NetReflector tries to load a class it will first check if there is a method with this signature. If there is it will call the method and pass in the node that it is trying to parse. In return it expects another node in return that contains the node that it will then parse.

This is typically used in situations where the XML needs to be transformed from one form to another. The most common scenario is when using parameter values – these have a non-XML syntax, so CruiseControl.NET needs to transform them into a syntax the parser can understand.

For example we could allow the user to enter a date of birth and pre-process it into an age. Giving the following XML:

```
<demoTask name="A Demo">
  <author>Craig</author>
  <dob>18-Jul-1982</dob>
</demoTask>
```

Code 6: Example XML with a date of birth

We could convert it into:

```
<demoTask name="A Demo">
  <author>Craig</author>
  <age>28</age>
</demoTask>
```

Code 7: Converted example XML

With the following pre-processor:

```
[ReflectionPreprocessor]
public XmlNode PreprocessParameters(NetReflectorTypeTable typeTable,
XmlNode inputNode)
{
    var dobNode = (from node in inputNode.ChildNodes
                       .OfType<XmlNode>()
                   where node.Name == "dob"
                   select node).SingleOrDefault();
    if (dobNode != null)
    {
        var dob = DateTime.Parse(dobNode.InnerText);
        inputNode.RemoveChild(dobNode);
        var ageNode = inputNode.OwnerDocument.CreateElement("age");
        ageNode.InnerText = Convert.ToInt32(
            (DateTime.Now - dob).TotalDays / 365)
            .ToString();
        inputNode.AppendChild(ageNode);
    }

    return inputNode;
}
```

Code 8: Pre-processor method to convert DOB to age

Of course this is a very simple example, but it shows the basic principal. The pre-processing can be as simple or as complex as desired.

Note: the LINQ statement in the above example is used to get around any issues with namespaces. The `SelectNodes()` and `SelectSingleNode()` methods have issues when namespaces are used.

Tasks/Publishers

Overview

Tasks are the basic unit of work within CruiseControl.NET. They perform most of the actions required to produce a build.

At its most simple a task does only one thing – it executes something. However a task can be expanded to provide additional interactions with the system – these are covered below.

Historical Note

As an aside, originally in CruiseControl.NET tasks and publishers were two completely separate concepts (and interfaces). Over time it was realised that they are actually the same thing and have been merged into one.

However some tasks are still classified as "publishers". This is more a conceptual classification rather than a physical one – they can still be run in any of the task blocks (prebuild, tasks or publishers.) However some publishers require the XML log that is produced by xmlLogger and can have unpredictable outside of the publishers block.

The Basics

The simplest task is built by implementing the `ITask` interface. This interface has the following definition:

```
public interface ITask
{
	void Run(IIntegrationResult result);
}
```

Code 9: ITask Definition

This interface provides a single method – Run(). This takes in an `IIntegrationResult`; this is used to pass information into the task and to return results (see Getting Information and Returning Results below). As a bare minimum the task needs to return a result code – whether it was successful or not.

A simple HelloWorldTask would be:

```
namespace Customisations.Demos
{
    using System;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Remote;

    public class HelloWorldTask
        : ITask
    {
        public void Run(IIntegrationResult result)
        {
            Console.WriteLine("Hello");
            result.Status = IntegrationStatus.Success;
        }
    }
}
```

Code 10: Basic HelloWorldTask

At this point, while the task will run, it won't be detected by the configuration reader. To do this a reflector attribute needs to be added:

```
namespace CCNet.CSharpDemos.Plugin
{
    using System;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("helloWorld")]
    public class HelloWorldTask
        : ITask
    {
        public void Run(IIntegrationResult result)
        {
            Console.WriteLine("Hello");
            result.Status = IntegrationStatus.Success;
        }
    }
}
```

Code 11: HelloWorldTask with reflection

This defines a task called helloWorld. When the task runs it will display "Hello" in the console.

This can then be configured in a project:

```
<cruisecontrol xmlns="http://thoughtworks.org/ccnet/1/5">
  <project name="DemoTest">
    <tasks>
      <helloWorld/>
    </tasks>
  </project>
</cruisecontrol>
```

Code 12: Example configuration for the HelloWorldTask

This can then be run and tested – although not much will be seen (unless the console window is open – then you will see the message).

Note: Each reflector name must be unique – if there are any duplicate names NetReflector will throw an exception!

GETTING INFORMATION

Information from the current build is passed in via the `IIntegrationResult`. This provides some basic details on the project, plus information about the build. The section on IIntegrationResult: Integration Results in Common Components contains the full details on this class, but this section will cover some of the more useful properties.

Probably the most commonly used property is the Status. This contains the current status of the build – it can be used to detect what state the build is in. Valid values are:

- Success – everything has run successfully prior to the task

- Failure – a previous task has failed
- Exception – there has been an unhandled exception in the build
- Unknown – the status has not been set yet (typically this will be the status for the first task in a build)
- Cancelled – the task has been cancelled

The result also contains the following information on a project:

- Name
- Working folder
- Artefact folder
- Project URL (when set)

And the following information on the build:

- Any parameters
- The build condition (IfModificationExists or ForceBuild)
- The label
- Start/end times
- Any modifications
- Any current results

Plus some details on the last build:

- Status
- Label
- Last run date/time

For example, we can modify our HelloWorldTask to contain the project name and when the build started:

```
namespace CCNet.CSharpDemos.Plugin
{
    using System;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("helloWorld")]
    public class HelloWorldTask
        : ITask
    {
        public void Run(IIntegrationResult result)
        {
            Console.WriteLine("Hello from " + result.ProjectName +
                "(build started " + result.StartTime.ToString() + ")");
            result.Status = IntegrationStatus.Success;
        }
    }
}
```

Code 13: HelloWorldTask using result information

Again this can be tested, although not much happens still. The next task is to add some results that will be added to the log.

Returning Results

To pass results back to the build the result is again used. We have already seen how the Status can be used to set the outcome of the task. It is also possible to add information to the log file using the result.

To do this there is a method on `IIntegrationResult` called `AddTaskResult()`. This method has two overrides – one that accepts a plain string and the other that accepts an `ITaskResult` (see Advanced Results below).

To add a string to the results is as simple as calling AddTaskResult with the string. For example, instead of writing to the console we can write to the log instead:

```
namespace CCNet.CSharpDemos.Plugin
{
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("helloWorld")]
    public class HelloWorldTask
        : ITask
    {
        public void Run(IIntegrationResult result)
        {
            result.AddTaskResult("Hello from " + result.ProjectName +
                "(build started " + result.StartTime.ToString() + ")");
            result.Status = IntegrationStatus.Success;
        }
    }
}
```

Code 14: Adding to the result log from HelloWorldTask

Again, running this does not do much, but if you open the log file you will see the message in the log.

Adding an external file to the log is a similar process, but using an `ITaskResult` instead. For example, to add a file:

```
result.AddTaskResult(new FileTaskResult(fileName));
```

Code 15: Adding a file result

See Advanced Results below for further details on adding files.

Passing Configuration Options

The next step is to add some configuration options to the task. This is done by adding some properties and marking them with reflector attributes.

The following code shows how to add a person's name to the task:

```
namespace CCNet.CSharpDemos.Plugin
{
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("helloWorld")]
    public class HelloWorldTask
```

```
        : ITask
    {
        [ReflectorProperty("name")]
        public string PersonsName { get; set; }

        public void Run(IIntegrationResult result)
        {
            result.AddTaskResult("Hello " + this.PersonsName +
                " from " + result.ProjectName +
                "(build started " + result.StartTime.ToString() + ")");
            result.Status = IntegrationStatus.Success;
        }
    }
}
```

Code 16: Adding a configuration property

This will add a property called name to the configuration. If we try to run this new task without changing the configuration we will get an error:

```
Unable to instantiate CruiseControl projects from
configuration document.
Configuration document is likely missing Xml nodes required
for properly populating CruiseControl configuration.
Missing Xml node (name) for required member
(CCNet.CSharpDemos.Plugin.HelloWorldTask.PersonsName).
Xml: <helloWorld xmlns="http://thoughtworks.org/ccnet/1/5" />
 Conflicting project data : <project name="DemoTest"
xmlns="http://thoughtworks.org/ccnet/1/5"><tasks><helloWorld
/></tasks></project>
```

This is because by default properties are added as required. When the configuration is loaded it will attempt to validate the configuration to ensure that all required information is set.

So if we modify our configuration to the following it will now work:

```
<cruisecontrol xmlns="http://thoughtworks.org/ccnet/1/5">
  <project name="DemoTest">
    <tasks>
      <helloWorld name="John Doe"/>
    </tasks>
  </project>
</cruisecontrol>
```

Code 17: Configuration with property

So what if we want to add an optional parameter to the task? This is done by setting the required property on the attribute. For example if we added a repeat property to the HelloWorldTask:

```
namespace CCNet.CSharpDemos.Plugin
{
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("helloWorld")]
    public class HelloWorldTask
        : ITask
    {
        public HelloWorldTask()
        {
            this.RepeatCount = 1;
        }

        [ReflectorProperty("name")]
        public string PersonsName { get; set; }

        [ReflectorProperty("count", Required = false)]
        public int RepeatCount { get; set; }

        public void Run(IIntegrationResult result)
        {
            for (var loop = 0; loop < this.RepeatCount; loop++)
            {
                result.AddTaskResult("Hello " + this.PersonsName +
                    " from " + result.ProjectName +
                    "(build started " + result.StartTime.ToString() + ")");
            }

            result.Status = IntegrationStatus.Success;
        }
    }
}
```

Code 18: An optional configuration property

Since this is an optional property a constructor has also been added to set the default value.

Attempting to run this task without changing the configuration will still work. Setting the property will then cause the greeting to be repeated multiple times.

Build Progress Information

As well as the log information it is also helpful to return a status message when the task is running. This shows the user that something is actually happening within the build.

Build progress information is sent to the server from the task via the `BuildProgressInformation` property on `IIntegrationResult`. Typically when a task starts it will signal that it has started. The following example shows how this is done:

```
namespace CCNet.CSharpDemos.Plugin
{
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("helloWorld")]
    public class HelloWorldTask
        : ITask
    {
        public HelloWorldTask()
        {
            this.RepeatCount = 1;
        }

        [ReflectorProperty("name")]
        public string PersonsName { get; private set; }
        [ReflectorProperty("count", Required = false)]
        public int RepeatCount { get; set; }

        public void Run(IIntegrationResult result)
        {
            result.BuildProgressInformation
                .SignalStartRunTask("Sending a hello world greeting");
            for (var loop = 0; loop < this.RepeatCount; loop++)
            {
                result.AddTaskResult("Hello " + this.PersonsName +
                    " from " + result.ProjectName +
                    "(build started " + result.StartTime.ToString() + ")");
            }

            result.Status = IntegrationStatus.Success;
        }
    }
}
```

Code 19: Sending a build progress report

This call passes the message into the build status – when a client queries the server these messages are included in the status details.

Additional Interfaces

As well as the basic `ITask` there are some additional interfaces that can be used to extend the functionality of the task. These are all covered in Common Components below, but this section will cover how they apply to tasks.

For tasks that require validation beyond required/optional it is possible to use the `IConfigurationValidation` interface. This will be triggered after the configuration is loaded. Any validation issues can either be flagged as a warning (the build will still run) or as an error (the build will not run.)

For example, if we wanted to modify HelloWorldTask to display a warning if the count is zero or less we could do the following:

```
namespace CCNet.CSharpDemos.Plugin
{
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Core.Config;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("helloWorld")]
    public class HelloWorldTask
        : ITask, IConfigurationValidation
    {
        public HelloWorldTask()
        {
            this.RepeatCount = 1;
        }

        [ReflectorProperty("name")]
        public string PersonsName { get; private set; }

        [ReflectorProperty("count", Required = false)]
        public int RepeatCount { get; set; }
        public void Run(IIntegrationResult result)
        {
            result.BuildProgressInformation
                .SignalStartRunTask("Sending a hello world greeting");
            for (var loop = 0; loop < this.RepeatCount; loop++)
            {
                result.AddTaskResult("Hello " + this.PersonsName +
                    " from " + result.ProjectName +
                    "(build started " + result.StartTime.ToString() + ")");
            }

            result.Status = IntegrationStatus.Success;
        }

        public void Validate(IConfiguration configuration,
ConfigurationTrace parent, IConfigurationErrorProcesser errorProcesser)
        {
            if (this.RepeatCount <= 0)
            {
                errorProcesser.ProcessWarning("count is less than 1!");
            }
        }
    }
}
```

Code 20: Adding validation to HelloWorldTask

This will display the following warning message:

```
[CCNet Server:WARN] count is less than 1!
```

As you can see only the message gets displayed - so making the warning as precise as possible is very helpful!

The `IStatusSnapshotGenerator` allows a task to generate any status report. This is used in the dashboard and CCTray to show the internal workings of a project. In 1.5 a new interface has been added called `IParamatisedItem`. This allows a task to apply any parameter values.

These last two interfaces provide some very common functionality, so in 1.5 some new classes were added to simplify using these interfaces.

Base Classes

In the 1.5 release some new abstract classes were added to simplify the development of tasks. These abstract classes provide the common functionality that is used in different tasks. These classes are shown in Figure 8:

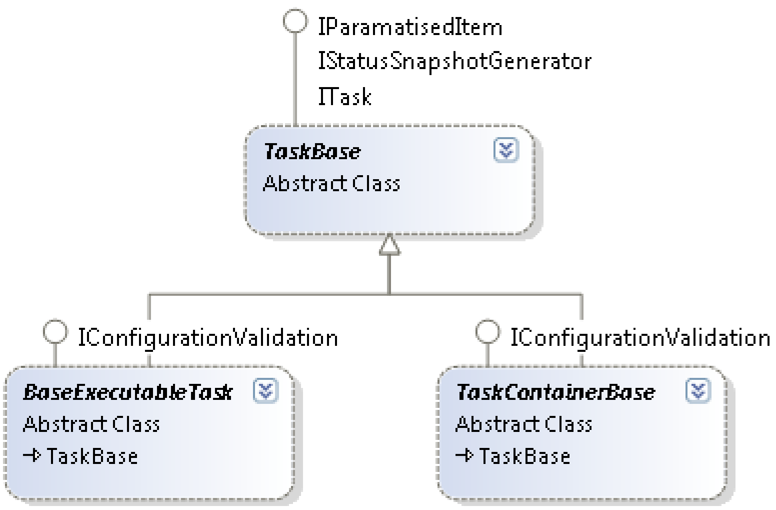

Figure 8: Abstract task classes

TaskBase implements the functionality required for IParamatisedItem and IStatusSnapshotGenerator. New tasks can now get this functionality without needing to re-code it every time. It also added a Description property to be used in the task reporting.

BaseExecutableTask provides the infrastructure for executing external applications from within CruiseControl.NET. This includes error monitoring and handling time-outs, stopping tasks mid-run, etc.

TaskContainerBase provides functionality for tasks that contain other tasks as sub-tasks. This is mainly an extension of the functionality in TaskBase so it can handle the sub-tasks as well.

These abstract classes all provide a default implementation of `Run()`. Any derived classes need to override the `Execute()` method instead. This method is very similar to the `Run()` method, but it returns a `bool` value. This value is then used to set whether the task was successful or not.

So, the HelloWorldTask can be rewritten to use these base classes:

```
namespace CCNet.CSharpDemos.Plugin
{
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Core.Config;
    using ThoughtWorks.CruiseControl.Core.Tasks;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("helloWorld2")]
    public class HelloWorldTask2
        : TaskBase, IConfigurationValidation
    {
        public HelloWorldTask2()
        {
            this.RepeatCount = 1;
        }

        [ReflectorProperty("name")]
        public string PersonsName { get; private set; }
        [ReflectorProperty("count", Required = false)]
        public int RepeatCount { get; set; }

        protected override bool Execute(IIntegrationResult result)
        {
            result.BuildProgressInformation
                .SignalStartRunTask("Sending a hello world greeting");
            for (var loop = 0; loop < this.RepeatCount; loop++)
            {
                result.AddTaskResult("Hello " + this.PersonsName +
                    " from " + result.ProjectName +
                    "(build started " + result.StartTime.ToString() + ")");
            }

            return true;
        }

        public void Validate(IConfiguration configuration,
ConfigurationTrace parent, IConfigurationErrorProcesser errorProcesser)
        {
            if (this.RepeatCount <= 0)
            {
                errorProcesser.ProcessWarning("count is less than 1!");
            }
        }
    }
}
```

Code 21: HelloWorldTask from TaskBase

This still has all the old functionality, but it now reports its status and can use parameters.

Advanced Results

In the above demos we looked at two versions of `AddTaskResult()`. One took a `string` as the parameter, while the other used a `FileTaskResult` instance. Why is this of importance?

Internally the `string` version converts generates a new `DataTaskResult` instance and stores the `string`. This then gets stored for writing to the log. So internally the log results are all stored as `ITaskResult` instances. This provides an additional extension point for storing and writing results.

Out of the box, CruiseControl.NET comes with four implementations of `ITaskResult`:

- `DataTaskResult` – this is the most basic implementation, it simply stores a string to write to the log.
- `FileTaskResult` – stores a reference to a file to import into the log. When the log is written the contents of the file are loaded into memory, converted into XML and then written to the log.
- `ProcessTaskResult` – holds the results of an external application call. This is typically the standard error and output strings, plus the result code. These are concatenated and written to the log on saving.

- `XmlTaskResult` – used for building up an XML document in memory and then writes it to the log on saving (added in 1.5).

These can all be instantiated directly and then stored as a result for the log. However it is also possible to write new implementations.

The definition of `ITaskResult` is very simple:

```
public interface ITaskResult
{
    string Data { get; }
    bool CheckIfSuccess();
}
```

Code 22: ITaskResult definition

The `Data` property is called when the log is being written. This just returns a string, in any format! As the log is XML the log writer will automatically ensure that the data is a valid XML string (it will enclose it in a CDATA block if the string is not XML.)

The `CheckIfSuccess()` method is a short-cut path for setting the `Status` of `IIntegrationResult`. If this method returns `true`, then will be set to `Success`, otherwise it will be set to `Failure`.

For example, we can build a custom `ITaskResult` for our HelloWorldTask. Rather than storing the entire string, we might just want to store the important details:

```
namespace CCNet.CSharpDemos.Plugin
{
    using ThoughtWorks.CruiseControl.Core;

    public class HelloWorldTaskResult
        : ITaskResult
    {
        private readonly string personName;
        private readonly IIntegrationResult result;

        public HelloWorldTaskResult(string personName, IIntegrationResult 
result)
        {
            this.personName = personName;
            this.result = result;
        }

        public string Data
        {
            get
            {
                return "Hello " + this.personName +
                    " from " + this.result.ProjectName +
                    "(build started " + this.result.StartTime.ToString() + 
")";
            }
        }

        public bool CheckIfSuccess()
        {
            return true;
        }
    }
}
```

Code 23: Custom ITaskResult for HelloWorldTask

This simply stores the person's name and the result details. When the log is written it will generate the greeting and return it, exactly the same as the old version.

Then the task needs to be changed to use the new implementation:

```
namespace CCNet.CSharpDemos.Plugin
{
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Core.Config;
    using ThoughtWorks.CruiseControl.Core.Tasks;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("helloWorld2")]
    public class HelloWorldTask2
        : TaskBase, IConfigurationValidation
    {
        public HelloWorldTask2()
        {
            this.RepeatCount = 1;
        }

        [ReflectorProperty("name")]
        public string PersonsName { get; private set; }

        [ReflectorProperty("count", Required = false)]
        public int RepeatCount { get; set; }

        protected override bool Execute(IIntegrationResult result)
        {
            result.BuildProgressInformation
                .SignalStartRunTask("Sending a hello world greeting");
            for (var loop = 0; loop < this.RepeatCount; loop++)
            {
                result.AddTaskResult(
                    new HelloWorldTaskResult(this.PersonsName, result));
            }

            return true;
        }

        public void Validate(IConfiguration configuration,
ConfigurationTrace parent, IConfigurationErrorProcesser errorProcesser)
        {
            if (this.RepeatCount <= 0)
            {
                errorProcesser.ProcessWarning("count is less than 1!");
            }
        }
    }
}
```

Code 24: Using the custom ITaskResult

Of course this is a very contrived example, but it does show how easy it is to add a new type of result (perhaps encrypted or compressed in memory – the possibilities are endless!)

EXECUTING EXTERNAL APPLICATIONS

One common pattern in CruiseControl.NET is a task that calls an external application. This pattern is so common that in 1.4 the `BaseExecutableTask` abstract task was added. The class was changed slightly in 1.5 to inherit from, but it still works in the same way (the `Execute()` method needs to be implemented instead of the `Run()` method.)

If HelloWorld is an external application, we could use this abstract task to build the following class:

```
namespace CCNet.CSharpDemos.Plugin
{
    using System.Diagnostics;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Core.Tasks;

    [ReflectorType("callHelloWorld")]
    public class CallHelloWorldTask
        : BaseExecutableTask
    {
        [ReflectorProperty("name")]
        public string PersonsName { get; private set; }

        [ReflectorProperty("executable")]
        public string Executable { get; private set; }

        protected override string GetProcessFilename()
        {
            return string.IsNullOrEmpty(this.Executable) ?
                "HelloWorldApp" :
                this.Executable;
        }

        protected override string GetProcessArguments(IIntegrationResult
result)
        {
            return "\"" + this.PersonsName + "\"";
        }

        protected override string
GetProcessBaseDirectory(IIntegrationResult result)
        {
            return result.WorkingDirectory;
        }

        protected override ProcessPriorityClass GetProcessPriorityClass()
        {
            return ProcessPriorityClass.Normal;
        }

        protected override int GetProcessTimeout()
        {
            return 300;
        }

        protected override bool Execute(IIntegrationResult result)
        {
            var processResult = this.TryToRun(
                this.CreateProcessInfo(result),
                result);
            result.AddTaskResult(new ProcessTaskResult(processResult));
            return !processResult.Failed;
        }
    }
}
```

Code 25: Deriving from BaseExecutableTask

The two public properties are not strictly needed – they just provide configurable options for the task. However all the protected methods are required.

`GetProcessFilename()` sets the name of the application to execute. Since this is a configurable property we also want to set a default – otherwise when the task tries to run it will throw an exception. Also note the missing extension – it is generally good practise to omit the extension on the default. This is to allow cross platform compatibility with Mono-based installs.

`GetProcessArguments()` allows the task to build the list of arguments to pass into the application. This method just returns a string.

`GetProcessBaseDirectory()` sets the working directory for the application. If the process filename is a relative path then the path is relative to this directory.

`GetProcessPriorityClass()` and `GetProcessTimeout()` provide some additional settings for the execution.

Finally there is the standard `Execute()` (or `Run()`) method. Internally this method calls `this.TryToRun(this.CreateProcessInfo(result), result)`. This call builds up the process information (defines the application to call, its arguments, etc.) and then calls it. A `ProcessResult` is returned containing the results of the call.

This is a very minimal example, it is also possible to do additional processing, e.g. import external result files, pre-/post-process data, etc.

Triggers

Overview

A trigger is a special type of task that can start a build. It does not run inside the normal build process, instead it is called by the project integrator directly. This means it does not have access to any of the information that a normal task has.

A trigger simply checks to see if its condition is fulfilled and then returns either a request to start a build or `null`. The project integrator is responsible for checking the outcome and then scheduling a build.

If a request is returned, then a build is started. This must be an instance of `IntegrationRequest`. If `null` is returned, then nothing will happen.

The Basics

A trigger implements the `ITrigger` interface. This interface has the following definition:

```
public interface ITrigger
{
	IntegrationRequest Fire();
	DateTime NextBuild { get; }
	void IntegrationCompleted();
}
```

Code 26: ITrigger definition

The `Fire()` method is called by the poll loop. This means that it will be called approximately once every second when the project is pending a build, and not called at all when a build is running. If a trigger takes a long time to run then it will slow down all other triggers for a project!

The `NextBuild` property is for informational purposes. It is used for generating the project status report. When a report is requested each trigger is queried to see when it next expects to either trigger a build, or to check whether a build should be triggered. If there is no next time, then this property should return `DateTime.MaxValue`.

Note: triggering a build does not necessarily mean a build will run. If the trigger condition is `IfModificationExists`, then this will initiate a source control check. Only if this check detects changes, or the condition is `ForceBuild`, will an actual build start.

Finally the `IntegrationCompleted()` method is a way for the trigger to get feedback on when a build has completed. This can be useful for resetting the trigger after a build has completed.

Basic Example

For an example, we will build a trigger that triggers a build when a file changes.

To start with, this trigger must implement the `ITrigger` interface:

```
namespace CCNet.CSharpDemos.Plugin
```

```
{
    using System;
    using System.IO;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("fileChangedTrigger")]
    public class FileChangedTrigger
        : ITrigger
    {
        private DateTime? lastChanged;

        public DateTime NextBuild
        {
            get { return DateTime.MaxValue; }
        }

        public void IntegrationCompleted()
        {
        }

        public IntegrationRequest Fire()
        {
            IntegrationRequest request = null;
            var changeTime = File.GetLastWriteTime(@"C:\monitor.txt");

            if (this.lastChanged.HasValue)
            {
                if (changeTime > this.lastChanged.Value)
                {
                    request = new IntegrationRequest(
                        BuildCondition.ForceBuild,
                        this.GetType().Name,
                        null);
                    this.lastChanged = changeTime;
                }
            }
            else
            {
                this.lastChanged = changeTime;
            }

            return request;
        }
    }
}
```

Code 27: Basic FileChangedTrigger

This trigger has a private field to store the date the file was last changed. The first time the trigger is checked it will store the date/time the file was last changed. On subsequent checks this field is checked against the file date/time to see if anything has changed.

If the date/time has changed then a new `IntegrationRequest` instance is started. This instance contains the condition, the name of the trigger and the name of the user who triggered the changed.

By convention the trigger name defaults to the class name of the actual trigger, but a trigger can change this if desired. The user name defaults to `null` for automatic triggers – this is normally set when a build is forced by a client.

This implementation returns `DateTime.MaxValue` for the `NextBuild` property. Since we don't know when the file will change, we just tell the system we don't know.

And that's all there is to a basic trigger. The condition for a build can be any check desired – the only requirement is the trigger returns an `IntegrationRequest` when a build is to be triggered.

To configure this trigger in a project is a simple matter of adding the trigger to the triggers block of a project. Using our last example we can configure the project like this:

```
<cruisecontrol xmlns="http://thoughtworks.org/ccnet/1/5">
  <project name="DemoTest">
    <triggers>
      <fileChangedTrigger/>
    </triggers>
    <tasks>
      <helloWorld name="John Doe"/>
    </tasks>
  </project>
</cruisecontrol>
```

Code 28: Project configuration with the fileChangedTrigger

This will monitor a file called monitor.txt in C:\. When this file is changed a build will be triggered.

Passing Configuration Options

This trigger is very inflexible – it does not allow for the user to choose which file to monitor or the type of build condition. Like a task this information can be exposed to the user by adding some reflected properties:

```
namespace CCNet.CSharpDemos.Plugin
{
    using System;
    using System.IO;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("fileChangedTrigger")]
    public class FileChangedTrigger
        : ITrigger
    {
        private DateTime? lastChanged;

        public FileChangedTrigger()
        {
            this.BuildCondition = BuildCondition.IfModificationExists;
        }

        [ReflectorProperty("file")]
        public string MonitorFile { get; set; }

        [ReflectorProperty("buildCondition", Required = false)]
        public BuildCondition BuildCondition { get; set; }

        public DateTime NextBuild
        {
            get { return DateTime.MaxValue; }
        }

        public void IntegrationCompleted()
        {
        }

        public IntegrationRequest Fire()
        {
            IntegrationRequest request = null;
            var changeTime = File.GetLastWriteTime(this.MonitorFile);

            if (this.lastChanged.HasValue)
            {
                if (changeTime > this.lastChanged.Value)
                {
                    request = new IntegrationRequest(
                        this.BuildCondition,
                        this.GetType().Name,
                        null);
                    this.lastChanged = changeTime;
                }
            }
            else
            {
                this.lastChanged = changeTime;
            }

            return request;
        }
    }
}
```

Code 29: FileChangedTrigger with configurable properties

This adds two properties – an optional property called `BuildCondition` and a required property called `MonitorFile`. If the user does not set the `MonitorFile`, then the configuration will not load – making it easy to validate that a value has at least been set.

The other changes are in the `Fire()` method – it now checks the specified file instead of the hard-coded path, and then returns the specified build condition. A constructor was also added to set the default value for the condition.

Note: NetReflector does not know how to handle nullable types. Therefore it is not possible to use a nullable type, with a get default value check.

After changing the configuration, it is now possible to monitor any file.

Post Build Processing

This will happily check the file and then trigger a build whenever the file has been changed since the last check. But what happens if the file is changed during the build? During the build this trigger is not checked, but since the date/time has not been changed, the first check after a build will fire the trigger again!

As you can imagine this could lead to an endless sets of trigger-build-trigger cycles. Not a very good scenario. The way to handle this is to update the date after a build has run:

```
namespace CCNet.CSharpDemos.Plugin
{
    using System;
    using System.IO;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("fileChangedTrigger")]
    public class FileChangedTrigger
        : ITrigger
    {
        private DateTime? lastChanged;

        public FileChangedTrigger()
        {
            this.BuildCondition = BuildCondition.IfModificationExists;
        }

        [ReflectorProperty("file")]
        public string MonitorFile { get; set; }

        [ReflectorProperty("buildCondition", Required = false)]
        public BuildCondition BuildCondition { get; set; }

        public DateTime NextBuild
        {
            get { return DateTime.MaxValue; }
        }

        public void IntegrationCompleted()
        {
            this.lastChanged = File.GetLastWriteTime(this.MonitorFile);
        }

        public IntegrationRequest Fire()
        {
            IntegrationRequest request = null;
            var changeTime = File.GetLastWriteTime(this.MonitorFile);

            if (this.lastChanged.HasValue)
            {
                if (changeTime > this.lastChanged.Value)
                {
                    request = new IntegrationRequest(
                        this.BuildCondition,
                        this.GetType().Name,
                        null);
                    this.lastChanged = changeTime;
                }
            }
            else
            {
                this.lastChanged = changeTime;
            }

            return request;
        }
    }
}
```

Code 30: Post-build processing

This example updated the private field to record the date/time after the build has completed. Otherwise this check will work the same as the previous example.

Validating the Configuration

Like a task it is possible to add internal validation to a trigger. This uses the same `IConfigurationValidation` interface.

In our trigger so far, the file to monitor must be set, but it could be set to an empty string, or it could be set to monitor a file that does not exist. We can modify our trigger to check these conditions in the validation phase:

```
namespace CCNet.CSharpDemos.Plugin
{
    using System;
    using System.IO;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Core.Config;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("fileChangedTrigger")]
    public class FileChangedTrigger
        : ITrigger, IConfigurationValidation
    {
        private DateTime? lastChanged;

        public FileChangedTrigger()
        {
            this.BuildCondition = BuildCondition.IfModificationExists;
        }

        [ReflectorProperty("file")]
        public string MonitorFile { get; set; }

        [ReflectorProperty("buildCondition", Required = false)]
        public BuildCondition BuildCondition { get; set; }

        public DateTime NextBuild
        {
            get { return DateTime.MaxValue; }
        }

        public void IntegrationCompleted()
        {
            this.lastChanged = File.GetLastWriteTime(this.MonitorFile);
        }

        public IntegrationRequest Fire()
        {
            IntegrationRequest request = null;
            var changeTime = File.GetLastWriteTime(this.MonitorFile);

            if (this.lastChanged.HasValue)
            {
                if (changeTime > this.lastChanged.Value)
                {
                    request = new IntegrationRequest(
                        this.BuildCondition,
                        this.GetType().Name,
                        null);
                    this.lastChanged = changeTime;
                }
            }
            else
            {
                this.lastChanged = changeTime;
            }

            return request;
        }

        public void Validate(IConfiguration configuration,
            ConfigurationTrace parent,
            IConfigurationErrorProcesser errorProcesser)
        {
            if (string.IsNullOrEmpty(this.MonitorFile))
            {
                errorProcesser.ProcessError("File cannot be empty");
            }
            else if (!File.Exists(this.MonitorFile))
            {
                errorProcesser.ProcessWarning(
                    "File '" + this.MonitorFile + "' does not exist");
            }
        }
    }
}
```

Code 31: Internal validation for a trigger

This example adds the `IConfigurationValidation` interface and implements the method. Since there must be a file, this will file the validation if it is not set. However since the file could be generated during the build (or by another project, etc.) the file missing check just returns a warning.

Nested Triggers

While this approach works fine for a file on a local disk, for a file on the network it would mean there is network traffic every second. If the network is slow, it might take several seconds to check the file. It would be nice to give the user the option of how often to check the file.

Now we could add a second property to our trigger saying how often to check a trigger and this is a valid approach. But there is a common pattern in building triggers where a trigger can have an inner trigger. The trigger is only checked if the inner trigger has already fired.

This pattern also gives the user the choice of which inner trigger to use – what if they only wanted to check the file once a day at a set time? Or if they wanted to wait for a project to finish before performing the check? With this approach the user chooses the trigger and we do not need to change our trigger at all!

Here is how we would add an inner trigger to our trigger:

```
namespace CCNet.CSharpDemos.Plugin
{
    using System;
    using System.IO;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Core.Config;
    using ThoughtWorks.CruiseControl.Core.Triggers;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("fileChangedTrigger")]
    public class FileChangedTrigger
        : ITrigger, IConfigurationValidation
    {
        private DateTime? lastChanged;

        public FileChangedTrigger()
        {
            this.BuildCondition = BuildCondition.IfModificationExists;
            this.InnerTrigger = new IntervalTrigger { IntervalSeconds = 5
};
        }

        [ReflectorProperty("file")]
        public string MonitorFile { get; set; }

        [ReflectorProperty("buildCondition", Required = false)]
        public BuildCondition BuildCondition { get; set; }

        [ReflectorProperty("trigger", InstanceTypeKey = "type", Required =
false)]
        public ITrigger InnerTrigger { get; set; }

        public DateTime NextBuild
        {
            get { return DateTime.MaxValue; }
        }

        public void IntegrationCompleted()
        {
            this.InnerTrigger.IntegrationCompleted();
            this.lastChanged = File.GetLastWriteTime(this.MonitorFile);
        }

        public IntegrationRequest Fire()
        {
            IntegrationRequest request = null;
            if (this.lastChanged.HasValue)
            {
                if (this.InnerTrigger.Fire() != null)
                {
                    var changeTime =
File.GetLastWriteTime(this.MonitorFile);
                    if (changeTime > this.lastChanged.Value)
                    {
                        request = new IntegrationRequest(
                            this.BuildCondition,
                            this.GetType().Name,
                            null);
                        this.lastChanged = changeTime;
                    }
                }
            }
            else
            {
                this.lastChanged = File.GetLastWriteTime(this.MonitorFile);
            }

            return request;
        }

        public void Validate(IConfiguration configuration,
            ConfigurationTrace parent,
            IConfigurationErrorProcesser errorProcesser)
        {
            if (string.IsNullOrEmpty(this.MonitorFile))
            {
                errorProcesser.ProcessError("File cannot be empty");
            }
            else if (!File.Exists(this.MonitorFile))
            {
                errorProcesser.ProcessWarning(
                    "File '" + this.MonitorFile + "' does not exist");
            }
        }
    }
}
```

Code 32: Using a nested trigger

There are a few things to note with this example. First, we added a new property called `InnerTrigger`. This uses an instance type in the reflection attribute to enable NetReflector to load the type directly (see below).

Second this property is initialised to a default trigger. We could have forced the user to always define an inner trigger – but this allows the user to use the existing configuration without any changes.

Next, when the `Fire()` method is calling for second and subsequent times, it calls the inner trigger's `Fire()` method. If this returns a non-null result, then we check the file. Otherwise we leave things unchanged.

Finally, because the inner trigger may need to do something after a build is finished, we need to pass on the `IntegrationCompleted()` call to it. In theory we should also pass on validation to the inner trigger, but to save space I have not added this code.

The inner trigger uses a slightly different syntax to configure, mainly because of the way NetReflector works. To use the new inner trigger with a 30s interval trigger we would define it as follows:

```
<cruisecontrol xmlns="http://thoughtworks.org/ccnet/1/5">
  <project name="DemoTest">
    <triggers>
      <fileChangedTrigger>
        <trigger type="intervalTrigger">
          <seconds>30</seconds>
        </trigger>
        <file>C:\monitor.txt</file>
        <buildCondition>ForceBuild</buildCondition>
      </fileChangedTrigger>
    </triggers>
    <tasks>
      <helloWorld name="John Doe"/>
    </tasks>
  </project>
</cruisecontrol>
```

Code 33: Configuring an inner trigger

Other than this change the trigger is defined in exactly the same way. You can try playing around with different trigger types and see what happens.

On a side note, since the custom trigger uses the standard interface it can also be used as an inner trigger in the other trigger types that accept inner triggers (e.g. filterTrigger, parameterTrigger, etc.)

Source Control Blocks

Overview

A source control block provides a link between CruiseControl.NET and a source code repository. All interactions with the repository go through one of these blocks.

A source control block exposes three actions:

- Gets any modifications since the last check
- Retrieves the updated source code
- Applies a label to the repository

These actions are called at different times in the build process, but they use the same underlying definition.

The Basics

Like the other project items, a source control block is defined by an interface. This time the interface is called `ISourceControl`. The full definition is:

```
public interface ISourceControl
{
    Modification[] GetModifications(IIntegrationResult from,
        IIntegrationResult to);
    void LabelSourceControl(IIntegrationResult result);
    void GetSource(IIntegrationResult result);
    void Initialize(IProject project);
    void Purge(IProject project);
}
```

Code 34: ISourceControl definition

The first three methods (`GetModifications()`, `LabelSourceControl()` and `GetSource()`) provide the three actions that a source control provide. The remaining methods are no longer used within CruiseControl.NET.

When a build is triggered, it makes a call to `GetModifications()` to see if anything has changed since the last build. It makes this call irrespective of the build type – for the simple reason that it needs the list of modifications for later parts of the build process. If the build condition is `IfModificationExists`, then the build will be cancelled if there are no changes. Otherwise the build will continue to the next step (generating the label and executing any pre-build tasks).

The next step for the source code block is a call to `GetSource()` to update the source code. This will pull the code from the source code repository into the working directory. Once this has finished the tasks in the project can then use the code.

When all the tasks have completed `LabelSourceControl()` is called to apply a label to the repository. This label is generated just after the modifications check has finished.

The first action is performed every time a build is triggered, and the other two are triggered every time a build is started – the only way to bypass these actions is internally within the source control block.

A Basic Source Control Block

As an example, we will build a new file system monitor (there is already a source control block to do this, but it is the simplest type of block to implement.) As a starter here is the implementation of the block, plus the NetReflector registration:

```
namespace CCNet.CSharpDemos.Plugin
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using System.IO;

    [ReflectorType("fileSystemSource")]
    public class NewFileSystemSourceControl
        : ISourceControl
    {
        private Dictionary<string, DateTime> files = new Dictionary<string,
DateTime>();

        [ReflectorProperty("directory")]
        public string DirectoryName { get; set; }

        public Modification[] GetModifications(IIntegrationResult from,
IIntegrationResult to)
        {
            var newList = new Dictionary<string, DateTime>();
            var modifications = new List<Modification>();
            var directory = new DirectoryInfo(this.DirectoryName);
            var newFiles = directory.GetFiles();
            foreach (var file in newFiles)
            {
                var inList = this.files.ContainsKey(file.FullName);
                if (!inList ||
                    (this.files[file.FullName] < file.LastWriteTime))
                {
                    newList.Add(file.FullName, file.LastWriteTime);
                    var modification = new Modification
                    {
                        FileName = file.Name,
                        ModifiedTime = file.LastWriteTime,
                        FolderName = file.DirectoryName,
                        Type = inList ? "Added" : "Modified"
                    };
                    modifications.Add(modification);
                }

                if (inList)
                {
                    this.files.Remove(file.Name);
                }
            }

            foreach (var file in this.files.Keys)
            {
                var modification = new Modification
                {
                    FileName = Path.GetFileName(file),
                    ModifiedTime = DateTime.Now,
                    FolderName = Path.GetDirectoryName(file),
                    Type = "Deleted"
                };
                modifications.Add(modification);
            }

            this.files = newList;
            return modifications.ToArray();
        }

        public void LabelSourceControl(IIntegrationResult result)
        {
            var fileName = Path.Combine(
                this.DirectoryName,
                DateTime.Now.ToString("yyyyMMddHHmmss") + ".label");
            File.WriteAllText(fileName, result.Label);
            this.files.Add(fileName, DateTime.Now);
        }

        public void GetSource(IIntegrationResult result)
        {
            foreach (var modification in result.Modifications)
            {
                var source = Path.Combine(
                    modification.FolderName,
                    modification.FileName);
                var destination = result.BaseFromWorkingDirectory(
                    modification.FileName);
                if (File.Exists(source))
                {
                    File.Copy(source, destination, true);
                }
                else
                {
                    File.Delete(destination);
                }
            }
        }

        public void Initialize(IProject project)
        {
            // Not needed
        }

        public void Purge(IProject project)
        {
            // Not needed
        }
    }
}
```

Code 35: New file system source control block

As you can see, there is quite a bit to even a basic source control block! Especially to implement one properly (and even this implementation has some limitations.)

There is an internal store of all the files and their last modified time in this class. This is primarily to simplify things. When the modifications are checked, it gets the names of all the files in the directory and then builds up a modification list of all the changes.

Each modification is stored in a `Modification` instance. The basic properties are the file and folder names – everything else is optional (although helpful.) Additionally there are several other properties which are not available from a file system – such as the person who modified the file, the version number, etc.

If there are no modifications then this method will return an empty array – which is what the system is expecting.

The `GetSource()` method copies any new or modified files to the working folder, while deleting any old files.

Finally, the `LabelSourceControl()` method simply adds a new file to the folder containing the label. This is because a file system does not have any labelling functionality. It also adds the file to the internal cache so it is not used to start a build on the next check!

To and From Results

The `GetModifications()` method has a slightly different signature from most project items. This method takes in both a `from` and a `to` result. The question is why?

The answer lies in the type of data that this method is returning. The other methods are all for a single point in time – the time that the method is called. In contrast the `GetModifications()` method is getting data for a range. This range is from when the last build ran to the start of the current build:

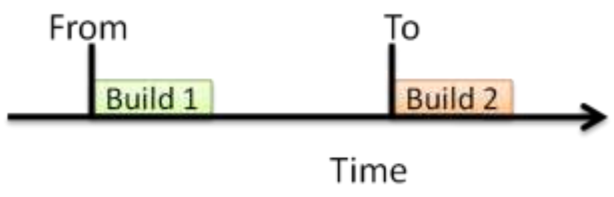

Figure 9: Timelines for from and to results

There are two possibilities for generating this range. First, the source control block can get all the modifications from the end of the last build to the start of the next:

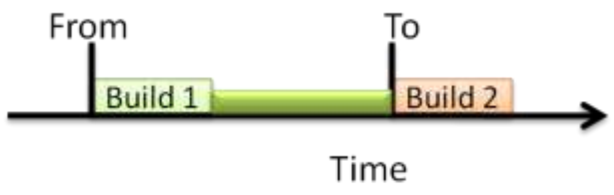

Figure 10: Selecting from end of previous build

This approach would use `from.EndTime` as the starting date for the selection and `to.StartTime` as the ending date.

The other option is to retrieve from the start of the last build to the start of the current build:

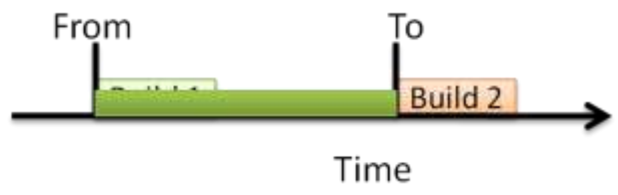

Figure 11: Selecting from start of previous build

This approach would use `from.StartTime` as the starting date for the selection and `to.StartTime` as the ending date.

Either approach is valid – it just depends on how the source control provider works. The developer can choose to use either approach or even something completely different.

Passing Data Between Calls

Sometimes a source control block might want to pass data between the various calls that are invoked. For example, the `GetModifications()` method might get the identifier of the last commit. The `GetSource()` method could then use this identifier instead of re-querying the repository.

While it is possible to store this value as a field within the block, there is no guarantee that this will be retained between the two calls. Instead there is an alternate approach that will always work – the `SourceControlData` property on `IIntegrationResult`.

This property is a list of `NameValuePair` – simply a value that has a known name. Any of the source control methods can place values in this list, and they will be available through-out the entire build.

Additionally these values are also persisted between builds. This means a source control block could store the current identifier in the `GetModifications()` method and use it in the `GetSource()` method. The next time the `GetModifications()` method is called it can use this value as its starting point, rather than the date/time properties.

Parsers

A common pattern for source control blocks is to have two classes. One performs the actual source control operations and the other parses the results from the get modifications action. This pattern is so common it has its own interface - `IHistoryParser`.

The definition for this interface is:

```
public interface IHistoryParser
{
	Modification[] Parse(TextReader history, DateTime from, DateTime to);
}
```

Code 36: IHistoryParser definition

The single method is used to parse all the modifications within a date range. The source control block will typically get the results, wrap the results in a `TextReader` and then call the interface. The `from` and `to` dates generally come from the `from` and `to` integration results that are passed to the source control block, although the source control block can pass in whatever values it wants.

For example, we could build a parser like the following:

```
namespace CCNet.CSharpDemos.Plugin.SourceControl
{
    using System;
    using System.Collections.Generic;
    using System.IO;
    using System.Xml.Linq;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Core.Sourcecontrol;

    public class IndexFileHistoryParser
        : IHistoryParser
    {
        public Modification[] Parse(TextReader history, DateTime from,
DateTime to)
        {
            var modifications = new List<Modification>();
            var document = XDocument.Load(history);
            var fromDate = from.ToString("s");
            var toDate = to.ToString("s");
            foreach (XElement change in document.Descendants("change"))
            {
                var date = DateTime.Parse(
                    change.Attribute("date").Value);
                if ((date >= from) &&
                    (date <= to))
                {
                    var modification = new Modification
                    {
                        FileName = change.Attribute("file").Value,
                        ModifiedTime = date,
                        FolderName = change.Attribute("folder").Value,
                        Type = change.Attribute("type").Value
                    };
                    modifications.Add(modification);
                }
            }

            return modifications.ToArray();
        }
    }
}
```

Code 37: History parser for retrieving entries from an index file

The parser just iterates through an index file and retrieves all the entries that have a date within the range.

To use this parser would be use the following source control block:

```
namespace CCNet.CSharpDemos.Plugin.SourceControl
{
    using System;
    using System.IO;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    [ReflectorType("indexFileSource")]
    public class IndexFileSourceControl
        : ISourceControl
    {
        [ReflectorProperty("file")]
        public string FileName { get; set; }

        public Modification[] GetModifications(IIntegrationResult from,
IIntegrationResult to)
        {
            var path = to.BaseFromWorkingDirectory(this.FileName);
            using (var reader = new StreamReader(path))
            {
                var parser = new IndexFileHistoryParser();
                return parser.Parse(reader, from.StartTime, to.StartTime);
            }
        }

        public void LabelSourceControl(IIntegrationResult result)
        {
            var fileName = Path.Combine(
                Path.GetDirectoryName(this.FileName),
                DateTime.Now.ToString("yyyyMMddHHmmss") + ".label");
            File.WriteAllText(fileName, result.Label);
        }

        public void GetSource(IIntegrationResult result)
        {
            foreach (var modification in result.Modifications)
            {
                var source = Path.Combine(
                    modification.FolderName,
                    modification.FileName);
                var destination = result.BaseFromWorkingDirectory(
                    modification.FileName);
                if (File.Exists(source))
                {
                    File.Copy(source, destination, true);
                }
                else
                {
                    File.Delete(destination);
                }
            }
        }

        public void Initialize(IProject project)
        {
            // Not needed
        }

        public void Purge(IProject project)
        {
            // Not needed
        }
    }
}
```

Code 38: Index file source control

This is very similar to the file system source control, but it reads from the index file using the parser. The parser is used to read the index file, instead of doing it directly in the source control block.

Now this may seem like overkill for something simple like this, but it means that the parser can be re-used by similar source control blocks. Additionally there are other classes that make use of this pattern to simplify development even more!

Abstract Base Classes

Source control blocks can also use the same `IParamatisedItem` interface as a task/publisher. To simplify the implementation of these interfaces there is an abstract base class that provides the basic function – `SourceControlBase`.

While this abstract class doesn't provide much in the way of additional functionality (just the ability to use parameters), there is another abstract class on top of this one that allows for executing an external source control provider – `ProcessSourceControl`. This class is similar to `BaseExecutableTask` in that it provides helper methods for executing an external application.

For example, if we have an external source control application called GetMyCode, we could write a source control block to use it like this:

```
namespace CCNet.CSharpDemos.Plugin.SourceControl
{
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Core.Sourcecontrol;
    using ThoughtWorks.CruiseControl.Core.Tasks;
    using ThoughtWorks.CruiseControl.Core.Util;

    [ReflectorType("getMyCode")]
    public class GetMyCodeSourceControl
        : ProcessSourceControl
    {
        public GetMyCodeSourceControl()
            : this(new ProcessExecutor(), new IndexFileHistoryParser())
        {
        }

        public GetMyCodeSourceControl(ProcessExecutor executor,
IHistoryParser parser)
            : base(parser, executor)
        {
        }

        [ReflectorProperty("executable", Required = false)]
        public string Executable { get; set; }

        [ReflectorProperty("source")]
        public string Source { get; set; }

        public override Modification[] GetModifications(IIntegrationResult
from, IIntegrationResult to)
        {
            var processResult = this.ExecuteCommand(to, "list");
            var modifications = this.ParseModifications(processResult,
                from.StartTime, to.StartTime);
            return modifications;
        }

        public override void LabelSourceControl(IIntegrationResult result)
        {
            if (result.Succeeded)
            {
                var processResult = this.ExecuteCommand(result,
                    "label", result.Label);
                result.AddTaskResult(
                    new ProcessTaskResult(processResult));
            }
        }

        public override void GetSource(IIntegrationResult result)
        {
            var processResult = this.ExecuteCommand(result, "get");
            result.AddTaskResult(
                new ProcessTaskResult(processResult));
        }

        private ProcessResult ExecuteCommand(IIntegrationResult result,
            string command, params string[] args)
        {
            var buffer = new PrivateArguments(command);
            buffer.Add(this.Source);
            foreach (var arg in args)
            {
                buffer.Add(string.Empty,
                    arg,
                    true);
            }

            var executable = string.IsNullOrEmpty(this.Executable) ?
                "GetMyCode" : this.Executable;
            var processInfo = new ProcessInfo(
                result.BaseFromWorkingDirectory(executable),
                buffer,
                result.WorkingDirectory);
            var processResult = this.Execute(processInfo);
            return processResult;
        }
    }
}
```

Code 39: Source control block to execute an external application

The underlying abstract class has a constructor that requires a `ProcessExecutor` and an `IHistoryParser`. Most of the time the defaults are fine (e.g. when the server needs an instance of this class), but for unit testing we might need to use some mocks instead – hence the two constructors.

Each of the three steps (get modifications, label source and get source) call through to a helper method called `ExecuteCommand()`. This method constructs the list of arguments and passes them through to a base method called `Execute()`, which is responsible for actually executing the command.

For getting the modifications there is another base method t assist with this - `ParseModifications()`. This method convert wraps the execution result in a reader and then calls the associated `IHistoryParser` to do the actual parse.

And that's pretty much all there is to it.

There are a couple of points to note on the `ExecuteCommand()` helper method. Internally this method generates an instance of `PrivateArguments`. This class is used to hide passwords and other private information from the logs and other public information. There is an associated class called `PrivateString`, which NetReflector can load. Unless specifically told, this class will hide the string value so external viewers cannot see it.

`PrivateArguments` knows about these strings and will correctly request the hidden value when needed (i.e. when passing the arguments) and the rest of the time will only show the obscured value.

As well as `PrivateArguments`, the method also uses a `ProcessInfo` instance. This instance is used to pass the command values (executable, arguments and path) to the underlying infrastructure for running the executable.

Labellers

Overview

A labeller generates a label for use within a build. This label can be in any format – CruiseControl.NET does not care – so this opens many possibilities for different types of labellers.

The generated label is not directly applied to the source code repository – instead it is stored in the `IIntegrationResult` and made available to the tasks/publishers. However it is used later by the source control block for labelling the code (after the tasks have run, but before the publishers.)

The Basics

A labeller must implement the `ILabeller` interface. This has the following definition:

```
public interface ILabeller
    : ITask
{
      string Generate(IIntegrationResult integrationResult);
}
```

Code 40: ILabeller definition

This interface also includes the interface for `ITask`. This means that a labeller can also be run as a task in one of the task blocks. The default action in this scenario is to generate a new label and store it in the result. However this is not recommended as it can screw up some other areas within CruiseControl.NET.

Basic Example

For this example we are going to build a labeller that generates a random label. The basic implementation, with reflection, looks like this:

```
namespace CCNet.CSharpDemos.Plugin
{
    using System;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;

    [ReflectorType("randomLabeller")]
    public class RandomLabeller
        : ILabeller
    {
        public string Generate(IIntegrationResult integrationResult)
        {
            var rand = new Random();
            var label = rand.Next().ToString();
            return label;
        }

        public void Run(IIntegrationResult result)
        {
            result.Label = this.Generate(result);
        }
    }
}
```

Code 41: Random labeller

This implementation provides the implementation for both interfaces – although the `Run()` method merely calls through to the `Generate()` method.

The `Generate()` method generates a string containing the label, which is returned to the caller. When the labeller is called as part of the build process the label is automatically stored in the result.

And that's all there is to a labeller. To call the labeller we would modify our configuration as follows:

```
<cruisecontrol xmlns="http://thoughtworks.org/ccnet/1/5">
  <project name="DemoTest">
    <triggers>
      <fileChangedTrigger>
        <trigger type="intervalTrigger">
          <seconds>30</seconds>
        </trigger>
        <file>C:\monitor.txt</file>
        <buildCondition>ForceBuild</buildCondition>
      </fileChangedTrigger>
    </triggers>
    <labeller type="randomLabeller"/>
    <tasks>
      <helloWorld name="John Doe"/>
    </tasks>
  </project>
</cruisecontrol>
```

Code 42: Configuration to use the randomLabeller

This configuration will generate a random number to use as the label for a project.

Additional Interfaces

A labeller can also have some of the same additional interfaces as a task. These interfaces are `IConfigurationValidation` and `IParamatisedItem`. They work in the same way as the task implementations – so we wanted to add a maximum value property to our random number labeller and validate it, we would do the following:

```
namespace CCNet.CSharpDemos.Plugin.Labellers
{
    using System;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Core.Config;

    [ReflectorType("randomLabeller")]
    public class RandomLabeller
        : ILabeller, IConfigurationValidation
    {
        public RandomLabeller()
        {
            this.MaximumValue = int.MaxValue;
        }

        [ReflectorProperty("max", Required = false)]
        public int MaximumValue { get; set; }

        public string Generate(IIntegrationResult integrationResult)
        {
            var rand = new Random();
            var label = rand.Next(this.MaximumValue).ToString();
            return label;
        }

        public void Run(IIntegrationResult result)
        {
            result.Label = this.Generate(result);
        }

        public void Validate(IConfiguration configuration,
            ConfigurationTrace parent,
            IConfigurationErrorProcesser errorProcesser)
        {
            if (this.MaximumValue <= 0)
            {
                errorProcesser.ProcessError(
                    "The maximum value must be greater than zero");
            }
        }
    }
}
```

Figure 12: Expanding the labeller to include validation

This simply adds a new property and then validates that it is greater than zero.

A Common Base

Like tasks, most of the interfaces are used for the labellers. So again a common base class has been provided to simplify development. This class is called `LabellerBase` and implements all the required functionality for the `IParamatisedItem` and `ITask` interfaces.

For example, the random labeller can be converted to use this common base:

```
namespace CCNet.CSharpDemos.Plugin.Labellers
{
    using System;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Core.Config;
    using ThoughtWorks.CruiseControl.Core.Label;

    [ReflectorType("randomLabeller2")]
    public class RandomLabeller2
        : LabellerBase, IConfigurationValidation
    {
        public RandomLabeller2()
        {
            this.MaximumValue = int.MaxValue;
        }

        [ReflectorProperty("max", Required = false)]
        public int MaximumValue { get; set; }

        public override string Generate(IIntegrationResult
integrationResult)
        {
            var rand = new Random();
            var label = rand.Next(this.MaximumValue).ToString();
            return label;
        }

        public void Validate(IConfiguration configuration,
            ConfigurationTrace parent,
            IConfigurationErrorProcesser errorProcesser)
        {
            if (this.MaximumValue <= 0)
            {
                errorProcesser.ProcessError(
                    "The maximum value must be greater than zero");
            }
        }
    }
}
```

Code 43: A labeller derived from LabellerBase

While this is not much difference in code, it also adds new functionality to the labellers – the ability for the new labeller to handle parameters. Additionally as new functionality is added to `LabellerBase` the derived classes will automatically inherit it.

State Managers

Overview

A state manager saves and loads the state of a project. As such it is used to persist the project's state between builds, allowing for the server to be stopped and started without losing the state.

The Basics

Like all other project items a state manager has an interface to inherit. This interface is `IStateManager` and is defined as:

```
public interface IStateManager
{
	IIntegrationResult LoadState(string project);

	void SaveState(IIntegrationResult result);

	bool HasPreviousState(string project);
}
```

Code 44: IStateManager interface

`LoadState()` is called the first time a project is built after the server has been started. After this it will not be called again, unless the server is restarted. It will load the last state of the project into memory as an `IIntegrationResult`.

In contrast `SaveState()` is called at the end of every build. This ensures that the state is always up-to-date. This will save the `IIntegrationResult` to whatever persistence medium is being used.

Finally, `HasPreviousState()` is used together with `LoadState()`. If `HasPreviousState()` returns `false`, then the system know not to call `LoadState()` as there is nothing to load, instead the system will generate a blank `IIntegrationResult`.

Basic Example

The default state manager in CruiseControl.NET saves the state as an XML file, we will write an alternate version that saves it as a plain text file. The basic implementation would look like this:

```
namespace CCNet.CSharpDemos.Plugin.State
{
    using System;
    using System.IO;
    using Exortech.NetReflector;
    using ThoughtWorks.CruiseControl.Core;
    using ThoughtWorks.CruiseControl.Core.State;
    using ThoughtWorks.CruiseControl.Core.Util;
    using ThoughtWorks.CruiseControl.Remote;

    [ReflectorType("plainTextState")]
    public class PlainTextStateManager
        : IStateManager
    {
        private readonly IFileSystem fileSystem;

        public PlainTextStateManager()
            : this(new SystemIoFileSystem())
        {
        }

        public PlainTextStateManager(IFileSystem fileSystem)
        {
            this.fileSystem = fileSystem;
        }

        public IIntegrationResult LoadState(string project)
        {
            var path = this.GeneratePath(project);
            using (var reader = new StreamReader(
                this.fileSystem.OpenInputStream(path)))
            {
                var status = (IntegrationStatus)
                    Enum.Parse(typeof(IntegrationStatus),
reader.ReadLine());
                var lastSummary = new IntegrationSummary(
                    status,
                    reader.ReadLine(),
                    reader.ReadLine(),
                    DateTime.Parse(reader.ReadLine()));
                var result = new IntegrationResult(
                    project,
                    reader.ReadLine(),
                    reader.ReadLine(),
                    null,
                    lastSummary);
                return result;
            }
        }

        public void SaveState(IIntegrationResult result)
        {
            var path = this.GeneratePath(result.ProjectName);
            using (var writer = new StreamWriter(
                this.fileSystem.OpenOutputStream(path)))
            {
                writer.WriteLine(result.Status);
                writer.WriteLine(result.Label);
                writer.WriteLine(result.LastSuccessfulIntegrationLabel);
                writer.WriteLine(result.StartTime.ToString("o"));
                writer.WriteLine(result.WorkingDirectory);
                writer.WriteLine(result.ArtifactDirectory);
            }
        }

        public bool HasPreviousState(string project)
        {
            var path = this.GeneratePath(project);
            return this.fileSystem.FileExists(path);
        }

        private string GeneratePath(string project)
        {
            var path = Path.Combine(Environment.CurrentDirectory,
                project + ".txt");
            return path;
        }
    }
}
```

Code 45: Plain text state manager

This example uses some of the utility classes and interfaces to assist with unit testing. The `IFileSystem` interface (and its default implementation `SystemIoFileSystem`) is used to abstract any operations with the file system. This allows a developer to write unit tests without needing to actually manipulate the file system. Since this is good practise the example uses this approach.

The `LoadState()` method is very simple – it just loads a minimal set of data from the file. It assumes that all the lines are in order (probably not a good assumption) and it only loads a subset of the full properties available (see IIntegrationResult: Integration Results below).

The `SaveState()` method is the opposite – it just writes the minimal data set to the file, with one line per property.

The `HasPreviousState()` just checks for the existence of the file – if the file is not there then the state has never been written – very simple.

These three methods all use a helper method called `GeneratePath()`. This method just generates the file name. Using the helper method means the code is not duplicated in each of the methods.

And that's all there is to writing a state manager. To use this state manager the following configuration would be needed:

```
<project name="DemoTest">
  <state type="plainTextState"/>
</project>
```

Code 46: Example configuration to use the state manager

More Details

It is possible to use NetReflector properties and configuration validation on state managers, just like any other project item.

Server Extensions

Overview

A server extension is an extension to the entire server, not just a single project. This allows for expanding the actual server functionality in a number of different ways.

A server extension has a very simple lifecycle. It is loaded when the server is loaded – this includes any initialisation that the extension needs to perform. When the server is started, any extensions are also started – likewise when the server is stopped, all extensions are stopped as well.

Given this simple lifecycle, what value do server extensions provide?

First, the server exposes a number of methods that an extension can call. These are useful for adding a new communications protocol (e.g. WCF), or adding a new way that the server can react to external events (e.g. a file change).

Second, the server also exposes a number of different events. An extension can listen to one (or more) of these events and then do something based on an event occurring. For example, an extension could listen to the build starting event and cancel the event if there is insufficient memory, or wait for projects to stop and send notification messages to any administrators.

The Basics

While server extensions are slightly different from project items, they still need to implement an interface. The interface to implement is called `ICruiseServerExtension` and is defined as follows:

```
public interface ICruiseServerExtension
{
    void Initialise(ICruiseServer server, ExtensionConfiguration
extensionConfig);

    void Start();

    void Stop();

    void Abort();
}
```

Code 47: ICruiseServerExtension definition

The `Initialise()` method is called when the server is first started. This provides the point where the extension can initialise itself, including loading any required configuration. The `server` argument is the server that is initialising, while the `extensionConfig` argument is for any associated configuration (see below).

The remaining methods (`Start()`, `Stop()` and `Abort()`) are pass-through methods from the server – whenever the server is starting, stopping or aborting the associated method will be called. Normally these methods will only be called once (either `Stop()` or `Abort()` will be called), but in some circumstances (i.e. restarting) they may be called multiple times.

As such, the most basic server extension would be:

```
namespace CCNet.CSharpDemos.Plugin.Extensions
{
    using System;
    using ThoughtWorks.CruiseControl.Remote;

    public class DoSomething
        : ICruiseServerExtension
    {
        private ICruiseServer server;

        public void Initialise(ICruiseServer server,
            ExtensionConfiguration extensionConfig)
        {
            this.server = server;
            Console.WriteLine("Initialise");
        }

        public void Start()
        {
            Console.WriteLine("Start");
        }

        public void Stop()
        {
            Console.WriteLine("Stop");
        }

        public void Abort()
        {
            Console.WriteLine("Abort");
        }
    }
}
```

Code 48: Server extension that does nothing

This extension does not do anything, it just writes some messages to the console whenever one of the methods is called.

Configuring an Extension

A server extension has a different configuration location than a project item. A project item is configured in ccnet.config, a server extension is configured in the application config file (e.g. ccnet.exe.config or ccservice.exe.config.)

This configuration requires two parts – a config section definition and the actual definition. The configu section definition is required for the .NET framework to read the configuration. If the default config file is used it will already include this definition, but if not the following line needs to be added to the `<configSections>` element:

```
<section name="cruiseServer"
type="ThoughtWorks.CruiseControl.Core.Config.ServerConfigurationHandler,
ThoughtWorks.CruiseControl.Core"/>
```

Code 49: Configuration section definition

Once this has been defined, a `<cruiseServer>` element can be added. This element will contain all the extension definitions. For example to call the above extension would be configured as follows:

```
<cruiseServer>
  <extension type="CCNet.CSharpDemos.Plugin.Extensions.DoSomething,
CCNet.CSharpDemos.Plugin" />
</cruiseServer>
```

Code 50: Defining the DoSomething extension

Each extension requires its own `<extension>` element, with a `type` attribute defining the extension type. This needs to be a standard .NET type definition (it can also include the public key and culture information if desired.)

Handling Events

The above example does not do anything beyond the basics. It would be nice to interact with the server and do some processing in response to events that occur. This can be done by listening to events on the `ICruiseServer` instance.

The following events are defined:

Table 1: ICruiseServer events

Event	Description
`ProjectStarting`	A project is starting – this event can be cancelled.
`ProjectStarted`	A project has been started.
`ProjectStopping`	A project is stopping – this event can be cancelled.
`ProjectStopped`	A project has been stopped.
`ForceBuildReceived`	A force build request has been received – this event can be cancelled.
`ForceBuildProcessed`	A force build has been processed. This does not mean that the force build has been completed, just that the request has been added to the queue.
`AbortBuildReceived`	An abort build request has been received – this event can be cancelled.
`AbortBuildProcessed`	An abort build request has been processed. This does not mean that the build has been aborted, just that the message has been received and the abort request made.
`SendMessageReceived`	A send message request has been received – this event can be cancelled.
`SendMessageProcessed`	A send message request has been processed and the message added to the project.
`IntegrationStarted`	An integration (build) has started – this event can be cancelled. This means that the build request is first in the queue and is ready to begin the actual build process.
`IntegrationCompleted`	An integration (build) has completed – that the build has either finished or had a fatal error and been stopped.

These events allow the extension to interact with the server and if necessary change its functionality. For example, we could expand the above extension to only allow five projects to run at the same time:

```
namespace CCNet.CSharpDemos.Plugin.Extensions
{
    using System;
    using ThoughtWorks.CruiseControl.Remote;

    public class DoSomething
        : ICruiseServerExtension
    {
        private ICruiseServer server;
        private int count = 0;

        public void Initialise(ICruiseServer server,
            ExtensionConfiguration extensionConfig)
        {
            this.server = server;
            this.server.ProjectStarting += (o, e) =>
            {
                if (this.count >= 4)
                {
                    e.Cancel = true;
                }
                else
                {
                    this.count++;
                }
            };
            this.server.ProjectStopped += (o, e) =>
            {
                this.count--;
            };
            Console.WriteLine("Initialise");
        }

        public void Start()
        {
            Console.WriteLine("Start");
        }

        public void Stop()
        {
            Console.WriteLine("Stop");
        }

        public void Abort()
        {
            Console.WriteLine("Abort");
        }
    }
}
```

Code 51: Limiting the number of running projects via an extension

The extension handles the starting and stopped events for projects. When a project is trying to start the extension will check if there is a slot available, if not it will cancel the start event.

If the stopping event was used instead there would be the possibility of more than the allowed projects running, as the time to stop a project is not always immediate (if a build is running, the build will finish before the project is stopped.) Choosing the stopped event ensures that this cannot happen.

This is a very simplistic implementation of this functionality. It does not allow for queuing starts, or concurrent starts, but it does show the basics of how events can be handled.

Passing Configuration

While the above sample works, it is not very flexible. It would be nice to allow the user to configure the extension so they can choose how many projects can run. This is done by adding XML elements to the configuration.

So, we can expand our above configuration like so:

```xml
<extension type="CCNet.CSharpDemos.Plugin.Extensions.DoSomething,
CCNet.CSharpDemos.Plugin">
  <allowedProjects>15</allowedProjects>
</extension>
```

Code 52: Passing the number of projects in the configuration

When the extension is initialised it is passed an `ExtensionConfiguration` instance. This instance contains the type of the extension, plus any XML elements that were defined. All the extension needs to do is search these elements for the required item:

```csharp
namespace CCNet.CSharpDemos.Plugin.Extensions
{
    using System;
    using System.Linq;
    using ThoughtWorks.CruiseControl.Remote;

    public class DoSomething
        : ICruiseServerExtension
    {
        private ICruiseServer server;
        private int count = 0;
        private int maxCount = 4;

        public void Initialise(ICruiseServer server,
            ExtensionConfiguration extensionConfig)
        {
            var projectsElement = extensionConfig.Items
                .SingleOrDefault(n => n.Name == "allowedProjects");
            if (projectsElement != null)
            {
                this.maxCount = int.Parse(
                    projectsElement.InnerText) - 1;
            }

            this.server = server;
            this.server.ProjectStarting += (o, e) =>
            {
                if (this.count >= this.maxCount)
                {
                    e.Cancel = true;
                }
                else
                {
                    this.count++;
                }
            };
            this.server.ProjectStopped += (o, e) =>
            {
                this.count--;
            };
            Console.WriteLine("Initialise");
        }

        public void Start()
        {
            Console.WriteLine("Start");
        }

        public void Stop()
        {
            Console.WriteLine("Stop");
        }

        public void Abort()
        {
            Console.WriteLine("Abort");
        }
    }
}
```

Code 53: Server extension with configuration.

This can be expanded to take in any parameters required, perform validation, etc. It is left up to the developer to decide what to do ad how.

COMMON COMPONENTS

IINTEGRATIONRESULT: INTEGRATION RESULTS

One of the most important interfaces for building project items is `IIntegrationResult`. This class has a number of properties and methods that allow the project items to pass and return information.

An `IIntegrationResult` is initialised at the start of a build and progressively built up over the build. It is passed into every single project item when they run, and is used to pass information around.

The properties and methods can be broken into the following categories:

- Project and build information
- Status information
- Modifications
- Historical information
- Logging
- Helpers

PROJECT AND BUILD INFORMATION

The properties in this category relate to the project and build as a whole. They are typically set by the system and should not be modified.

There are the following properties defined:

`ArtifactDirectory`	This is the location for all the project artefacts. Any file outputs from a task should be stored in this location. **Set:** this property is set when a build starts. **Change:** this property should not be changed.
`BuildCondition`	The condition for the current build. This will be either `IfModificationExists` or `ForceBuild` (there is a third condition, `NoBuild`, but it is not possible to get it during a build.) If the condition is `IfModificationExists` then it means that the build was triggered because there were changes detected to the source. **Set:** this property is set at the start of the build. If the build was initiated by a trigger, the trigger is responsible for setting this property. If the build is triggered by a client, then the client is responsible for setting this property. **Changes:** this property is read-only and cannot be changed.
`BuildLogDirectory`	This is the directory that the logs are written to. This is typically a sub-directory of the artefacts directory. **Set:** this property is set when a build starts.

	Change: this property should not be changed.
`IntegrationProperties`	Any "integration" properties that were added automatically by the system. See Integration Properties below. **Set:** this property is set automatically by the system prior to checking for any modifications. **Change:** this property is read-only and cannot be changed.
`IntegrationRequest`	The incoming request details. This includes the following properties: • `BuildCondition` – this will be the same as `BuildCondition` in the result. • `Source` – the source of the request. For triggered builds this will be the name of the trigger, for user-forced builds this will be the name of the originating machine. • `UserName` – the name of the user who started a build. This is new for 1.5. • `BuildValues` – any parameters for the build. This is new for 1.5. • `RequestTime` – the date and time the request was generated. This will typically be the date and time on the originating machine for user-forced builds. • `PublishOnSourceControlException` – whether a build should be published if there is an exception during the get modifications step of a build. This is set automatically by the system (based on the configuration) and is used to control the flow of logic within a build. **Set:** this property is set by the system when the build starts. The internal properties are set by the build source (trigger or remote machine). **Change:** this property is read-only and cannot be changed. The internal properties are also read-only.
`Parameters`	Any parameters that have been passed to the project. This includes all the user-entered parameters, plus the system level values (see below.) **Set:** these are set at the beginning of the build, and most of them do not change. However some values are updated during the build (e.g. build label.) **Change:** this property should not be changed.
`ProjectName`	The name of the project. This is the project name form the configuration. **Set:** this property is set at the start of a build. **Change:** this property is read-only and cannot be changed.

`ProjectUrl`

The URL to the project's summary page. This typically points to a web dashboard instance, but can be configured to point elsewhere.

Set: this property is set when a build starts.

Change: this property should not be changed.

`WorkingDirectory`

The working directory as configured within the project configuration. This is typically the working location for the entire project – the source code is retrieved this, the tasks and publishers all use this as their base directory, etc.

However, it is possible for the user to configure source control blocks and tasks to use different locations.

Set: this property is set when a build starts.

Change: this property should not be changed.

Status Information

These properties define information on the current status of the build.

`BuildProgressInformation`

This is a helper property for reporting progress information during a build. It is typically used to store start times for a task and any relevant information that has occurred within a task.

This property exposes the following methods:

- `SignalStartRunTask()` – initialises the start of a new project item, initialises the object for monitoring the progress.
- `AddTaskInformation()` – adds some information from the current item.
- `GetBuildProgressInformation()` – retrieves the information on the currently running item.

Set: this property is set at the start of a build.

Change: this property is read-only and cannot be changed. However the internal methods can be used to change the current build progress.

`EndTime`

The time the tasks section completed running. As such this value is not defined up to this point. Any tasks in the publishers section can use this property, as can the apply label step of the source control block.

Set: this property is set automatically by the system at the end of the tasks block.

Changes: this property is read-only and cannot be directly changed. It is changed by system using the `MarkEndTime()`

	method – this method should not be called by any project items.
`ExceptionResult`	The details of an exception that has occurred in a build. **Set:** This will be set automatically if the system detects an unhandled exception in a task or publisher. **Change:** this property is set automatically by the system and should not be changed.
`Failed`	Whether the build failed or not. This is based on the value in the `Status` property. **Set:** this is a calculated property. **Change:** this property is read-only and cannot be directly changed.
`Fixed`	Whether the build has successful fixed a prior breaking build. This checks both the `Status` property for the current build, and the `Status` property of the previous build. **Set:** this is a calculated property. **Change:** this property is read-only and cannot be directly changed.
`HasSourceControlError`	Returns `true` if there was an error in the source control block. **Set:** this is a calculated property. **Change:** this property is read-only and cannot be directly changed.
`Label`	This is the current label for the project. At the beginning of the build this value is not set – it is only set after the source code checks and the labeller has run. **Set:** this property is set automatically at the end of the generate label phase. **Change:** this property can be changed at anytime.
`SourceControlError`	The details of an exception that occurred during a source control operation. **Set:** this will be set automatically if the system detects an unhandled exception in the source control block. **Change:** this is set automatically by the system and should not be changed.
`StartTime`	The time the build started. **Set:** this is set at the start of the build.

Change: this property is set automatically and **MUST** not be changed.

`Status`

The current status of the build. This can be one of the following values:

- `Success`
- `Failure`
- `Exception`
- `Unknown`
- `Cancelled`

Prior to version 1.5 this property needed to be manually set when a task completed. It was up to the task to decide whatever logic was necessary to combine statuses (i.e. if the previous item failed what would the new status be, etc.)

Starting with 1.5 the status is now automatically set depending on the result returned from the `Execute()` method call.

The system will monitor this status at the end of every project item. If the status is set to `Failure` or `Exception`, then the current phase will be cancelled and the publishers executed.

Set: this is set to `Unknown` at the start of the build.

Change: this can be freely changed by any project item.

`Succeeded`

Whether the build is successful or not. This is based on the value in the `Status` property. This is only the status at the current item, if a later item fails the build; the previous items are not notified.

Set: this is a calculated property.

Change: this property is read-only and cannot be directly changed.

`TotalIntegrationTime`

The total time it took the build to run, from checking for modifications to the end of the tasks section.

See the description for `EndTime` for when this property is defined.

Set: this is a calculated property.

Change: this property is read-only and cannot be directly changed.

Additionally there are the following methods to update the status information:

`IsInitial()`

Whether this is the first build for a project or not. This is only be `true` if a project has never been built before. Otherwise it will be `false`, even when a project has been stopped and started.

`MarkEndTime()`	This is an internal helper method for marking the finish time of a build. It **must** not be called by any project items.
`MarkStartTime()`	This is an internal helper method for marking the start time of a build. It **must** not be called by any project items.

Modifications

These properties hold information on who modified the source code:

`LastChangeNumber`	The identifier of the last change. This will depend on the source control block. In 1.4.4 this was changed from an `int` value to a `string`. **Set:** this is a calculated property (from `Modifications`) **Change:** this property is read-only and cannot be changed directly.
`Modifications`	These are the modifications that have been detected by the source control block. When a build starts this property is undefined. **Set:** this property is set by the system after the source control block has retrieved the modifications. **Change:** this property is set automatically by the system and should not be changed.
`SourceControlData`	This is a holder property that allows a source control block to pass additional information. This is typically used to pass information between the various source control block steps (e.g. between get modifications and get source, etc.) As such the data in this property cannot be guaranteed. If a task wants to use a potential value from this property it must always check to see if the value is there first! **Set:** this property can be freely set by any project item. **Change:** this property can be freely changed by any project item.

Additionally there are the following methods that help when working with modifications:

`HasModifications()`	A helper method to check if there are any modifications for the build. If there are no modifications then this method will return `false`.
`ShouldRunBuild()`	A helper method for checking whether a build should run or not. This is an internal method that is used by the system. Internally this method checks whether there are any modifications or if the build condition is `ForceBuild`.

Historical Information

The result also holds a number of properties from the previous integration. These properties can be used to see what has happened in the project. These properties are:

`FailureUsers`	A list of any users who broke the previous build. This will only be set if the last status was `Failure` or `Exception` and there were any modifications. When a build breaks all the users who had checked in changes for that build will be added to this property. If subsequent users check in changes that do not fix the build then their names will also be added to this property. However each name will only appear in the list once. **Set:** this property is set at the start of a build. **Change:** this property is read-only and cannot be changed.
`LastBuildStatus`	The last status from a build that progressed past the get modifications step. **Set:** this property is set at the start of a build. **Change:** this property should not be modified.
`LastIntegration`	A summary of the last integration. This includes the following properties: • `Label` – the label from the last build. • `Status` – the final status of the last build. • `LastSuccessfulIntegrationLabel` – the last successful label. If the previous build failed or had an exception then this property will be different from `Label`. • `StartTime` – the date and time the last build started. • `FailureUsers` – the list of user who had modifications in a failing build. See `FailureUsers` in the result for how this property is populated. Additionally this property also exposes an `IsInitial()` method. This method is used for testing whether this is the first build for a project. This property will only be set if this is not the initial build for a project (see `IsInitial()` below). **Set:** this property is loaded automatically when the build is started. **Change:** this property is read-only and cannot be changed.
`LastIntegrationStatus`	The outcome of the last integration. If the project has never been built, this will be `Unknown`. **Set:** this property is loaded automatically when the build is started.

Change: this property is read-only and cannot be changed.

`LastModificationDate` The date and time of the last modification. This is only set if there have been any modifications for the project.

Set: this property is loaded automatically when the build is started.

Change: this property is read-only and cannot be changed.

`LastSuccessfulIntegrationLabel` The label of the last successful build. If the previous build failed or had an exception then property will go back to the last build that was successful. If there are no successful builds then this property will be undefined.

Set: this property is loaded automatically when the build is started.

Change: this property is read-only and cannot be changed.

LOGGING

The following properties are used in conjunction with the logging:

`TaskOutput` A `string` containing the current results for the tasks.

This property should not be used as it builds the string in memory. For large results this can result in an out of memory error!

Set: this is generated automatically every time the property is read.

Change: this property is read-only and cannot be directly changed.

`TaskResults` A list of results for the tasks within the build. These results are then added to the final build log by an xmlLogger task.

There are also helper methods to simplify adding to this list – see `AddTaskResult()` below.

The following helper method is also used to add log results:

`AddTaskResult()` Used for adding results from a task to the overall results. These results will later be concatenated into the log file by an xmlLogger.

There are two overloads for this method:

Overload 1

Inputs

`string result`: a string value to add to the results. This will wrap the value in a `DataTaskResult` instance.

Overload 2

Inputs

`ITaskResult result`: an `ITaskResult` instance to add to the results.

Helper Methods

Finally there are a set of helper methods for use with results. These are primarily designed to simplify working with results in different scenarios (e.g. retrieving directories, cloning and merging, etc.)

`BaseFromArtifactsDirectory()`	This will take a relative path and offset it from the artefact directory. If the path is absolute, then the absolute path will be returned. **Inputs** `string pathToBase`: the path to offset from the artefacts directory **Output** A `string` containing the full path – either the original path if absolute or offset from the artefacts directory if relative.
`BaseFromWorkingDirectory()`	This will take a relative path and offset it from the working directory. If the path is absolute, then the absolute path will be returned. **Inputs** `string pathToBase`: the path to offset from the working directory. **Output** A `string` containing the full path – either the original path if absolute or offset from the working directory if relative.
`Clone()`	Clones the current result. This is typically used by tasks that contain sub-tasks (e.g. for running multiple tasks in parallel, etc.)
`Merge()`	Merges a result into another result. This will add all the results from the source result into the destination and update the status depending on the source result.

Integration Properties

When a build is started a number of "integration" properties are set by the system. These are typically system generated properties about the build.

The currently defined properties are:

Table 2: Integration properties

Property	Description
`CCNetArtifactDirectory`	The project artefact directory (as an absolute path).
`CCNetBuildCondition`	The condition used to trigger the build, indicating if the build was triggered by new modifications or if it was forced. Valid values are **`ForceBuild`** or **`IfModificationExists`**.
`CCNetBuildDate`	The date of the build (in yyyy-MM-dd format).
`CCNetBuildTime`	The time of the start of the build (in HH:mm:ss format).
`CCNetFailureUsers`	The list of users who have contributed modifications to a sequence of builds that has failed.
`CCNetIntegrationStatus`	The status of the current integration. Could be **`Success`**, **`Failure`**, **`Exception`**, **`Unknown`** or **`Cancelled`**.
`CCNetLabel`	The label used to identify the build. This label is generated during the labeller step.
`CCNetLastIntegrationStatus`	The status of the previous integration. Could be **`Success`**, **`Failure`**, **`Exception`**, **`Unknown`** or **`Cancelled`**.
`CCNetListenerFile`	The file used by CCNet to read the progress of external tools.
`CCNetModifyingUsers`	The list of users who have contributed to the current build only.
`CCNetNumericLabel`	Contains the label as an integer if conversion is possible, otherwise zero.
`CCNetProject`	The name of the project that is being integrated.
`CCNetProjectUrl`	The URL where the project is located.
`CCNetRequestSource`	The source of the integration request; this will generally be the name of the trigger that raised the request.
`CCNetUser`	The user who forced the build. If security is off, or the build is not forced, then this will not be set.
`CCNetWorkingDirectory`	The project working directory (as an absolute path).

Most of these properties are set when the build is started and cannot be changed.

Further Information

There are a number of sources that can be used for further information.

First and foremost, the code for CruiseControl.NET is freely available from https://ccnet.svn.sourceforge.net/svnroot/ccnet/trunk. This is a Subversion repository so a Subversion client can be used to access the code.

A number of people have blogs on CruiseControl.NET, including some of the developers on the project. Some of these blogs are:

- http://csut017.wordpress.com/ - the blog for the author
- http://rubenwillems.blogspot.com/ - one of the other core developers
- http://treyhutcheson.wordpress.com/ - an older blog, but has a number of posts on customising CruiseControl.NET
- http://www.stevetrefethen.com/blog/?tag=/Continuous+Integration – more examples of how to customise CruiseControl.Net, including how to use it in a non-CI environment

Finally, you can always try the mailing groups for CruiseControl.NET:

- http://groups.google.com/group/ccnet-devel - the developer's forum
- http://groups.google.com/group/ccnet-user - general user's form

APPENDIX 1: VISUAL BASIC EXAMPLES

INTRODUCTION

While all the examples in this document have been in C#, it is possible to write extensions in any .NET language and Visual Basic is no exception. This appendix has all of the completed examples written in Visual Basic.

TASKS

DEMOTASK

This example shows the various NetReflector configuration options:

```
Imports ThoughtWorks.CruiseControl.Core
Imports Exortech.NetReflector
Imports System.Xml

<ReflectorType("demoTask")> Public Class DemoTask
    Implements ITask

    Private myName As String
    <ReflectorProperty("name")> _
    Public Property Name() As String
        Get
            Return myName
        End Get
        Set(ByVal value As String)
            myName = value
        End Set
    End Property

    Private myAuthor As String
    <ReflectorProperty("author", Required:=False)> _
    Public Property Author() As String
        Get
            Return myAuthor
        End Get
        Set(ByVal value As String)
            myAuthor = value
        End Set
    End Property

    Private myAge As Integer
    <ReflectorProperty("age", Required:=False)> _
    Public Property Age() As Integer
        Get
            Return myAge
        End Get
        Set(ByVal value As Integer)
            myAge = value
        End Set
    End Property

    Private myIsNonsense As Boolean
    <ReflectorProperty("isNonsense", Required:=False)> _
    Public Property IsNonsense() As Boolean
        Get
            Return myIsNonsense
        End Get
        Set(ByVal value As Boolean)
            myIsNonsense = value
        End Set
    End Property

    Private myInnerItems() As DemoTask
    <ReflectorProperty("items", Required:=False)> _
    Public Property InnerItems() As DemoTask()
        Get
            Return myInnerItems
        End Get
        Set(ByVal value As DemoTask())
            myInnerItems = value
        End Set
    End Property

    Private myChild As DemoTask
    <ReflectorProperty("child", Required:=False,
InstanceType:=GetType(DemoTask))> _
        Public Property Child() As DemoTask
        Get
            Return myChild
        End Get
        Set(ByVal value As DemoTask)
            myChild = value
        End Set
    End Property

    Private myTypedChild As DemoTask
    <ReflectorProperty("typedChild", Required:=False,
InstanceTypeKey:="type")> _
    Public Property TypedChild() As DemoTask
        Get
            Return myTypedChild
        End Get
        Set(ByVal value As DemoTask)
            myTypedChild = value
        End Set
    End Property

    Public Sub Run(ByVal result As IIntegrationResult) _
    Implements ITask.Run

    End Sub

    <ReflectionPreprocessor()> _
    Public Function PreprocessParameters(ByVal typeTable As
NetReflectorTypeTable, _
                                         ByVal inputNode As XmlNode) As
XmlNode
        Dim dobNode = (From node In inputNode.ChildNodes.OfType(Of
XmlNode)() _
                       Where node.Name = "dob" _
                       Select node).SingleOrDefault()
        If dobNode IsNot Nothing Then
            Dim dob = CDate(dobNode.InnerText)
            inputNode.RemoveChild(dobNode)
            Dim ageNode = inputNode.OwnerDocument.CreateElement("age")
            ageNode.InnerText = CInt((Date.Now - dob).TotalDays /
365).ToString()
```

```
            inputNode.AppendChild(ageNode)
        End If
        Return inputNode
    End Function

End Class
```

Code 54: DemoTask in Visual Basic

HelloWorldTask

This example shows a basic task with validation:

```
Imports Exortech.NetReflector
Imports ThoughtWorks.CruiseControl.Core
Imports ThoughtWorks.CruiseControl.Core.Config
Imports ThoughtWorks.CruiseControl.Remote

<ReflectorType("helloWorld")> Public Class HelloWorldTask
    Implements ITask, IConfigurationValidation

    Private myPersonsName As String
    <ReflectorProperty("name")> _
        Public Property PersonsName() As String
        Get
            Return myPersonsName
        End Get
        Set(ByVal value As String)
            myPersonsName = value
        End Set
    End Property

    Private myRepeatCount As Integer = 1
    <ReflectorProperty("count", Required:=False)> _
    Public Property RepeatCount() As Integer
        Get
            Return myRepeatCount
        End Get
        Set(ByVal value As Integer)
            myRepeatCount = value
        End Set
    End Property

    Public Sub Run(ByVal result As IIntegrationResult) Implements ITask.Run
        result.BuildProgressInformation() _
                .SignalStartRunTask("Sending a hello world greeting")

        For i = 0 To myRepeatCount
            result.AddTaskResult("Hello " & PersonsName & " from " &
result.ProjectName & _
                "(build started " + result.StartTime.ToString() + ")")
        Next

        result.Status = IntegrationStatus.Success
    End Sub

    Public Sub Validate(ByVal configuration As IConfiguration, _
                        ByVal parent As ConfigurationTrace, _
                        ByVal errorProcesser As
IConfigurationErrorProcesser) _
                        Implements IConfigurationValidation.Validate
        If myRepeatCount <= 0 Then
```

```
            errorProcesser.ProcessWarning("count is less than 1!")
        End If
    End Sub

End Class
```

Code 55: HelloWorldTask in Visual Basic

HelloWorldTask2

This example is for a task derived from TaskBase that uses a custom task result:

```
Imports Exortech.NetReflector
Imports ThoughtWorks.CruiseControl.Core
Imports ThoughtWorks.CruiseControl.Core.Config
Imports ThoughtWorks.CruiseControl.Core.Tasks
Imports ThoughtWorks.CruiseControl.Remote

<ReflectorType("helloWorld2")> Public Class HelloWorldTask2
    Inherits TaskBase
    Implements IConfigurationValidation

    Private myPersonsName As String
    <ReflectorProperty("name")> _
        Public Property PersonsName() As String
        Get
            Return myPersonsName
        End Get
        Set(ByVal value As String)
            myPersonsName = value
        End Set
    End Property

    Private myRepeatCount As Integer = 1
    <ReflectorProperty("count", Required:=False)> _
    Public Property RepeatCount() As Integer
        Get
            Return myRepeatCount
        End Get
        Set(ByVal value As Integer)
            myRepeatCount = value
        End Set
    End Property

    Protected Overrides Function Execute(ByVal result As
IIntegrationResult) As Boolean
        result.BuildProgressInformation() _
                .SignalStartRunTask("Sending a hello world greeting")

        For i = 0 To myRepeatCount
            Dim taskresult = New HelloWorldTaskResult(myPersonsName,
result)
            result.AddTaskResult(taskresult)
        Next

        Return True
    End Function

    Public Sub Validate(ByVal configuration As IConfiguration, _
                        ByVal parent As ConfigurationTrace, _
                        ByVal errorProcesser As
IConfigurationErrorProcesser) _
```

```
                    Implements IConfigurationValidation.Validate
        If myRepeatCount <= 0 Then
            errorProcesser.ProcessWarning("count is less than 1!")
        End If
    End Sub

End Class
```

Code 56: HelloWorldTask2 in Visual Basic

```
Imports ThoughtWorks.CruiseControl.Core

Public Class HelloWorldTaskResult
    Implements ITaskResult

    Private ReadOnly myPersonName As String
    Private ReadOnly myResult As IIntegrationResult

    Public Sub New(ByVal personName As String, ByVal result As
IIntegrationResult)
        myPersonName = personName
        myResult = result
    End Sub

    Public Function CheckIfSuccess() As Boolean _
    Implements ITaskResult.CheckIfSuccess
        Return True
    End Function

    Public ReadOnly Property Data() As String _
    Implements ITaskResult.Data
        Get
            Return "Hello " & myPersonName & _
                " from " & myResult.ProjectName & _
                "(build started " & myResult.StartTime.ToString() & ")"
        End Get
    End Property
End Class
```

Code 57: HelloWorldTaskResult in Visual Basic

CallHelloWorldTask

This example shows how to call an external application via the `BaseExecutableTask` class:

```
Imports System.Diagnostics
Imports Exortech.NetReflector
Imports ThoughtWorks.CruiseControl.Core
Imports ThoughtWorks.CruiseControl.Core.Tasks

Public Class CallHelloWorldTask
    Inherits BaseExecutableTask

    Private myPersonsName As String
    <ReflectorProperty("name")> _
    Public Property PersonsName() As String
        Get
            Return myPersonsName
        End Get
        Set(ByVal value As String)
            myPersonsName = value
        End Set
    End Property

    Private myExecutable As String
    <ReflectorProperty("executable")> _
    Public Property Executable() As String
        Get
            Return myExecutable
        End Get
        Set(ByVal value As String)
            myExecutable = value
        End Set
    End Property

    Protected Overrides Function Execute(ByVal result As
IIntegrationResult) As Boolean
        Dim processResult = TryToRun(CreateProcessInfo(result), result)
        result.AddTaskResult(New ProcessTaskResult(processResult))
        Return Not processResult.Failed
    End Function

    Protected Overrides Function GetProcessArguments(ByVal result As
IIntegrationResult) As String
        Return """" & myPersonsName & """"
    End Function

    Protected Overrides Function GetProcessBaseDirectory(ByVal result As
IIntegrationResult) As String
        Return result.WorkingDirectory
    End Function

    Protected Overrides Function GetProcessFilename() As String
        If String.IsNullOrEmpty(myExecutable) Then
            Return "HelloWorldApp"
        Else
            Return myExecutable
        End If
    End Function

    Protected Overrides Function GetProcessPriorityClass() As
ProcessPriorityClass
        Return ProcessPriorityClass.Normal
    End Function

    Protected Overrides Function GetProcessTimeout() As Integer
        Return 300
    End Function

End Class
```

Code 58: CallHelloWorldTask in Visual Basic

TRIGGERS

FILECHANGEDTRIGGER

This example shows a trigger that monitors a file, but only fires after the inner trigger has fired:

```
Imports System
Imports System.IO
Imports Exortech.NetReflector
Imports ThoughtWorks.CruiseControl.Core
Imports ThoughtWorks.CruiseControl.Core.Config
Imports ThoughtWorks.CruiseControl.Core.Triggers
Imports ThoughtWorks.CruiseControl.Remote

<ReflectorType("fileChangedTrigger")> Public Class FileChangedTrigger
    Implements ITrigger, IConfigurationValidation

    Private myLastChanged As DateTime?

    Public Sub New()
        BuildCondition = BuildCondition.IfModificationExists
        Dim defaultTrigger = New IntervalTrigger()
        defaultTrigger.IntervalSeconds = 5
        InnerTrigger = defaultTrigger
    End Sub

    Private myMonitorFile As String
    <ReflectorProperty("file")> _
    Public Property MonitorFile() As String
        Get
            Return myMonitorFile
        End Get
        Set(ByVal value As String)
            myMonitorFile = value
        End Set
    End Property

    Private myBuildCondition As BuildCondition
    <ReflectorProperty("buildCondition", Required:=False)> _
    Public Property BuildCondition() As BuildCondition
        Get
            Return myBuildCondition
        End Get
        Set(ByVal value As BuildCondition)
            myBuildCondition = value
        End Set
    End Property

    Private myInnerTrigger As ITrigger
    <ReflectorProperty("trigger", InstanceTypeKey:="type",
Required:=False)> _
    Public Property InnerTrigger() As ITrigger
        Get
            Return myInnerTrigger
        End Get
        Set(ByVal value As ITrigger)
            myInnerTrigger = value
        End Set
    End Property

    Public Function Fire() As IntegrationRequest Implements ITrigger.Fire
        Dim request As IntegrationRequest = Nothing
        If myLastChanged.HasValue Then
            If myInnerTrigger.Fire() IsNot Nothing Then
                Dim changeTime = File.GetLastWriteTime(MonitorFile)
                If (changeTime > myLastChanged.Value) Then
                    request = New IntegrationRequest(BuildCondition, _
                                                     Me.GetType().Name, _
                                                     Nothing)
                    myLastChanged = changeTime
                End If
            End If
        Else
            myLastChanged = File.GetLastWriteTime(MonitorFile)
        End If

        Return request
    End Function

    Public Sub IntegrationCompleted() Implements
ITrigger.IntegrationCompleted
        InnerTrigger.IntegrationCompleted()
        myLastChanged = File.GetLastWriteTime(MonitorFile)
    End Sub

    Public ReadOnly Property NextBuild() As Date Implements
ITrigger.NextBuild
        Get
            Return DateTime.MaxValue
        End Get
    End Property

    Public Sub Validate(ByVal configuration As IConfiguration, _
                        ByVal parent As ConfigurationTrace, _
                        ByVal errorProcesser As
IConfigurationErrorProcesser) _
                        Implements IConfigurationValidation.Validate
        If (String.IsNullOrEmpty(myMonitorFile)) Then
            errorProcesser.ProcessError("File cannot be empty")
        ElseIf (Not File.Exists(MonitorFile)) Then
            errorProcesser.ProcessWarning("File '" & MonitorFile & "' does
not exist")
        End If
    End Sub
End Class
```

Code 59: FileChangedTrigger in Visual Basic

Source Control Blocks

NewFileSystemSourceControl

This example is for a simple source control block that monitors a file directory:

```
Imports System
Imports System.Collections.Generic
Imports System.IO
Imports Exortech.NetReflector
Imports ThoughtWorks.CruiseControl.Core

<ReflectorType("fileSystemSource")> Public Class NewFileSystemSourceControl
    Implements ISourceControl

    Private myFiles As Dictionary(Of String, Date) = New Dictionary(Of
String, Date)()

    Private myDirectoryName As String
    <ReflectorProperty("directory")> _
    Public Property DirectoryName() As String
        Get
            Return myDirectoryName
        End Get
        Set(ByVal value As String)
            myDirectoryName = value
        End Set
    End Property

    Public Function GetModifications(ByVal from As IIntegrationResult, _
                                     ByVal [to] As IIntegrationResult) As
Modification() _
                                     Implements
ISourceControl.GetModifications
        Dim newList = New Dictionary(Of String, DateTime)()
        Dim modifications = New List(Of Modification)()
        Dim directory = New DirectoryInfo(DirectoryName)
        Dim newFiles = directory.GetFiles()
        For Each file In newFiles
            Dim inList = myFiles.ContainsKey(file.FullName)
            If (Not inList Or (myFiles(file.FullName) <
file.LastWriteTime)) Then
                newList.Add(file.FullName, file.LastWriteTime)
                Dim newModification = New Modification()
                newModification.FileName = file.Name
                newModification.ModifiedTime = file.LastWriteTime
                newModification.FolderName = file.DirectoryName
                If inList Then
                    newModification.Type = "Added"
                Else
                    newModification.Type = "Modified"
                End If

                modifications.Add(newModification)
            End If

                If (inList) Then
                myFiles.Remove(file.Name)
            End If
        Next

        For Each file In myFiles.Keys
            Dim oldModification = New Modification()
            oldModification.FileName = Path.GetFileName(file)
            oldModification.ModifiedTime = Date.Now
            oldModification.FolderName = Path.GetDirectoryName(file)
            oldModification.Type = "Deleted"
            modifications.Add(oldModification)
        Next

        myFiles = newList
        Return modifications.ToArray()
    End Function

    Public Sub GetSource(ByVal result As IIntegrationResult) _
    Implements ISourceControl.GetSource
        For Each Modification In result.Modifications
            Dim source = Path.Combine(Modification.FolderName, _
                Modification.FileName)
            Dim destination =
result.BaseFromWorkingDirectory(Modification.FileName)
            If (File.Exists(source)) Then
                File.Copy(source, destination, True)
            Else
                File.Delete(destination)
            End If
        Next
    End Sub

    Public Sub Initialize(ByVal project As IProject) _
    Implements ISourceControl.Initialize

    End Sub

    Public Sub LabelSourceControl(ByVal result As IIntegrationResult) _
    Implements ISourceControl.LabelSourceControl
        Dim fileName = Path.Combine(myDirectoryName, _
                                    Date.Now.ToString("yyyyMMddHHmmss") &
".label")
        File.WriteAllText(fileName, result.Label)
        myFiles.Add(fileName, DateTime.Now)
    End Sub

    Public Sub Purge(ByVal project As IProject) _
    Implements ISourceControl.Purge

    End Sub
End Class
```

Code 60: NewFileSystemSourceControl in Visual Basic

IndexFileSourceControl

This example shows how to build a source control block that uses an history parser:

```
Imports System
Imports System.IO
Imports Exortech.NetReflector
Imports ThoughtWorks.CruiseControl.Core

<ReflectorType("indexFileSource")> Public Class IndexFileSourceControl
    Implements ISourceControl

    Private myFileName As String
    <ReflectorProperty("file")> _
    Public Property FileName() As String
        Get
            Return myFileName
        End Get
        Set(ByVal value As String)
            myFileName = value
        End Set
    End Property

    Public Function GetModifications(ByVal from As IIntegrationResult, _
                                     ByVal [to] As IIntegrationResult) As
Modification() _
                                     Implements
ISourceControl.GetModifications
        Dim path = [to].BaseFromWorkingDirectory(FileName)
        Dim reader = New StreamReader(path)
        Try
            Dim parser = New IndexFileHistoryParser()
            Return parser.Parse(reader, from.StartTime, [to].StartTime)
        Finally
            reader.Dispose()
        End Try
    End Function

    Public Sub GetSource(ByVal result As IIntegrationResult) _
    Implements ISourceControl.GetSource

    End Sub

    Public Sub Initialize(ByVal project As IProject) _
    Implements ISourceControl.Initialize

    End Sub

    Public Sub LabelSourceControl(ByVal result As IIntegrationResult) _
    Implements ISourceControl.LabelSourceControl

    End Sub

    Public Sub Purge(ByVal project As IProject) _
    Implements ISourceControl.Purge

    End Sub
End Class
```

Code 61: IndexFileSourceControl in Visual Basic

```
Imports System
Imports System.Collections.Generic
Imports System.IO
Imports System.Xml.Linq
Imports ThoughtWorks.CruiseControl.Core
Imports ThoughtWorks.CruiseControl.Core.Sourcecontrol

Public Class IndexFileHistoryParser
    Implements IHistoryParser


    Public Function Parse(ByVal history As TextReader, _
                          ByVal from As Date, ByVal [to] As Date) As
Modification() _
                          Implements IHistoryParser.Parse
        Dim modifications = New List(Of Modification)()
        Dim document = XDocument.Load(history)
        Dim fromDate = from.ToString("s")
        Dim toDate = [to].ToString("s")
        For Each change As XElement In document.Descendants("change")
            Dim changeDate = DateTime.Parse(change.Attribute("date").Value)
            If (changeDate >= from) And (changeDate <= [to]) Then
                Dim newModification = New Modification()
                newModification.FileName = change.Attribute("file").Value
                newModification.ModifiedTime = changeDate
                newModification.FolderName =
change.Attribute("folder").Value
                newModification.Type = change.Attribute("type").Value
                modifications.Add(newModification)
            End If
        Next
        Return modifications.ToArray()
    End Function
End Class
```

Code 62: IndexFileHistoryParser in Visual Basic

GetMyCodeSourceControl

This example show calling an external application in a source control block:

```
Imports Exortech.NetReflector
Imports ThoughtWorks.CruiseControl.Core
Imports ThoughtWorks.CruiseControl.Core.Sourcecontrol
Imports ThoughtWorks.CruiseControl.Core.Tasks
Imports ThoughtWorks.CruiseControl.Core.Util

<ReflectorType("getMyCode")> Public Class GetMyCodeSourceControl
    Inherits ProcessSourceControl

    Public Sub New()
        Me.new(New ProcessExecutor(), New IndexFileHistoryParser())
    End Sub

    Public Sub New(ByVal executor As ProcessExecutor, _
                   ByVal parser As IHistoryParser)
        MyBase.New(parser, executor)
    End Sub

    Private myExecutable As String
    <ReflectorProperty("executable", Required:=False)> _
    Public Property Executable() As String
        Get
            Return myExecutable
        End Get
        Set(ByVal value As String)
            myExecutable = value
        End Set
    End Property

    Private mySource As String
    <ReflectorProperty("source")> _
    Public Property Source() As String
        Get
            Return mySource
        End Get
        Set(ByVal value As String)
            mySource = value
        End Set
    End Property

    Public Overloads Overrides Function GetModifications(ByVal from As
IIntegrationResult, _
                                                          ByVal [to] As
IIntegrationResult) As Modification()
        Dim processResult = ExecuteCommand([to], "list")
        Dim modifications = ParseModifications(processResult, _
            from.StartTime, [to].StartTime)
        Return modifications
    End Function

    Public Overrides Sub LabelSourceControl(ByVal result As
IIntegrationResult)
        If result.Succeeded Then
            Dim processResult = ExecuteCommand(result, "label",
result.Label)
            result.AddTaskResult(New ProcessTaskResult(processResult))
        End If
    End Sub
    Public Overrides Sub GetSource(ByVal result As IIntegrationResult)
        Dim processResult = ExecuteCommand(result, "get")
        result.AddTaskResult(New ProcessTaskResult(processResult))
    End Sub

    Private Function ExecuteCommand(ByVal result As IIntegrationResult, _
            ByVal command As String, Optional ByVal arg As String =
Nothing)
        Dim buffer = New PrivateArguments(command)
        buffer.Add(Source)
        If arg IsNot Nothing Then
            buffer.Add(String.Empty, arg, True)
        End If
        Dim executable As String
        If String.IsNullOrEmpty(myExecutable) Then
            executable = "GetMyCode"
        Else
            executable = myExecutable
        End If
        Dim processInfo = New ProcessInfo( _
            result.BaseFromWorkingDirectory(executable), _
            buffer, _
            result.WorkingDirectory)
        Dim processResult = Execute(processInfo)
        Return processResult
    End Function
End Class
```

Code 63: ProcessSourceControl in Visual Basic

Labellers

RandomLabeller

This example shows the basics of implementing a labeller:

```
Imports System
Imports Exortech.NetReflector
Imports ThoughtWorks.CruiseControl.Core
Imports ThoughtWorks.CruiseControl.Core.Config

<ReflectorType("randomLabeller")> Public Class RandomLabeller
    Implements ILabeller, IConfigurationValidation

    Public Sub New()
        myMaximumValue = Integer.MaxValue
    End Sub

    Private myMaximumValue As Integer
    <ReflectorProperty("max", Required:=False)> _
    Public Property MaximumValue() As Integer
        Get
            Return myMaximumValue
        End Get
        Set(ByVal value As Integer)
            myMaximumValue = value
        End Set
    End Property

    Public Function Generate(ByVal integrationResult As IIntegrationResult)
As String _
```

```
    Implements ILabeller.Generate
        Dim rand = New Random()
        Dim label = rand.Next(myMaximumValue).ToString()
        Return label
    End Function

    Public Sub Run(ByVal result As IIntegrationResult) _
    Implements ThoughtWorks.CruiseControl.Core.ITask.Run
        result.Label = Generate(result)
    End Sub

    Public Sub Validate(ByVal configuration As IConfiguration, _
                        ByVal parent As ConfigurationTrace, _
                        ByVal errorProcesser As
IConfigurationErrorProcesser) _
                        Implements IConfigurationValidation.Validate
        If myMaximumValue <= 0 Then
            errorProcesser.ProcessError( _
                    "The maximum value must be greater than zero")
        End If
    End Sub
End Class
```

Code 64: RandomLabeller in Visual Basic

RandomLabeller2

This examples shows the basics of implementing a labeller from `LabellerBase`:

```
Imports System
Imports Exortech.NetReflector
Imports ThoughtWorks.CruiseControl.Core
Imports ThoughtWorks.CruiseControl.Core.Config
Imports ThoughtWorks.CruiseControl.Core.Label

<ReflectorType("randomLabeller2")> Public Class RandomLabeller2
    Inherits LabellerBase
    Implements IConfigurationValidation

    Public Sub New()
        myMaximumValue = Integer.MaxValue
    End Sub

    Private myMaximumValue As Integer
    <ReflectorProperty("max", Required:=False)> _
    Public Property MaximumValue() As Integer
        Get
            Return myMaximumValue
        End Get
        Set(ByVal value As Integer)
            myMaximumValue = value
        End Set
    End Property

    Public Overrides Function Generate(ByVal integrationResult As
IIntegrationResult) As String
        Dim rand = New Random()
        Dim label = rand.Next(myMaximumValue).ToString()
        Return label
    End Function

    Public Sub Validate(ByVal configuration As IConfiguration, _
                        ByVal parent As ConfigurationTrace, _
                        ByVal errorProcesser As
IConfigurationErrorProcesser) _
                        Implements IConfigurationValidation.Validate
        If myMaximumValue <= 0 Then
            errorProcesser.ProcessError( _
                    "The maximum value must be greater than zero")
        End If
    End Sub
End Class
```

Code 65: RandomLabeller2 in Visual Basic

State Managers

PlainTextStateManager

This examples shows how to implement a state manager that stores the state as plain text (one line per item):

```
Imports System
Imports System.IO
Imports Exortech.NetReflector
Imports ThoughtWorks.CruiseControl.Core
Imports ThoughtWorks.CruiseControl.Core.State
Imports ThoughtWorks.CruiseControl.Core.Util
Imports ThoughtWorks.CruiseControl.Remote

<ReflectorType("plainTextState")> Public Class PlainTextStateManager
    Implements IStateManager

    Private ReadOnly myFileSystem As IFileSystem

    Public Sub New()
        Me.New(New SystemIoFileSystem())
    End Sub

    Public Sub New(ByVal fileSystem As IFileSystem)
        myFileSystem = fileSystem
    End Sub

    Public Function HasPreviousState(ByVal project As String) As Boolean _
    Implements IStateManager.HasPreviousState
        Dim filePath = GeneratePath(project)
        Return myFileSystem.FileExists(filePath)
    End Function

    Public Function LoadState(ByVal project As String) As
IIntegrationResult _
    Implements IStateManager.LoadState
        Dim filePath = GeneratePath(project)
        Dim reader = New
StreamReader(myFileSystem.OpenInputStream(filePath))
        Try
            Dim statusType = GetType(IntegrationStatus)
            Dim status = CType([Enum].Parse(statusType, reader.ReadLine()),
IntegrationStatus)
            Dim lastSummary = New IntegrationSummary(status, _
                reader.ReadLine(), _
                reader.ReadLine(), _
                DateTime.Parse(reader.ReadLine()))
            Dim result = New IntegrationResult(project, _
                reader.ReadLine(), _
                reader.ReadLine(), _
                Nothing, _
                lastSummary)
            Return result
        Finally
            reader.Dispose()
        End Try
    End Function

    Public Sub SaveState(ByVal result As IIntegrationResult) _
    Implements IStateManager.SaveState
        Dim filePath = GeneratePath(result.ProjectName)
        Dim writer = New
StreamWriter(myFileSystem.OpenOutputStream(filePath))
        Try
            writer.WriteLine(result.Status)
            writer.WriteLine(result.Label)
            writer.WriteLine(result.LastSuccessfulIntegrationLabel)
            writer.WriteLine(result.StartTime.ToString("o"))
            writer.WriteLine(result.WorkingDirectory)
            writer.WriteLine(result.ArtifactDirectory)
        Finally
            writer.Dispose()
        End Try
    End Sub

    Private Function GeneratePath(ByVal project As String) As String
        Dim filePath = Path.Combine(Environment.CurrentDirectory, project +
".txt")
        Return filePath
    End Function
End Class
```

Code 66: PlainTextStateManager in Visual Basic

Server Extensions

This example shows how to build a server extension that limits the number of concurrent projects:

```
Imports System
Imports System.Linq
Imports ThoughtWorks.CruiseControl.Remote
Imports ThoughtWorks.CruiseControl.Remote.Events

Public Class DoSomething
    Implements ICruiseServerExtension

    Private myServer As ICruiseServer
    Private myCount As Integer = 0
    Private myMaxCount As Integer = 4

    Public Sub Abort() _
    Implements ICruiseServerExtension.Abort
        Console.WriteLine("Abort")
    End Sub

    Public Sub Initialise(ByVal server As ICruiseServer, _
                          ByVal extensionConfig As ExtensionConfiguration)
_
```

```
                            Implements ICruiseServerExtension.Initialise
        Dim projectsElement = extensionConfig.Items _
            .SingleOrDefault(Function(n) n.Name = "allowedProjects")
        If projectsElement IsNot Nothing Then
            myMaxCount = Integer.Parse(projectsElement.InnerText) - 1
        End If

        myServer = server
        AddHandler myServer.ProjectStarting, AddressOf Project_Starting
        AddHandler myServer.ProjectStopped, AddressOf Project_Stopped
        Console.WriteLine("Initialise")
    End Sub

    Private Sub Project_Starting(ByVal sender As Object, ByVal e As
CancelProjectEventArgs)
        If myCount >= myMaxCount Then
            e.Cancel = True
        Else
            myCount += 1
        End If
    End Sub

    Private Sub Project_Stopped(ByVal sender As Object, ByVal e As
ProjectEventArgs)
        myCount -= 1
    End Sub

    Public Sub Start() _
    Implements ICruiseServerExtension.Start
        Console.WriteLine("Start")
    End Sub

    Public Sub [Stop]() _
    Implements ICruiseServerExtension.Stop
        Console.WriteLine("Stop")
    End Sub
End Class
```

Code 67: DoSomething extension in Visual Basic

Note: this will also require a change to the configuration to point to the correct name of the extension.