# Overview

### About this test

We have found that an applicant's CV is not a good indicator of programming aptitude, so the eCargo development team has designed this test as a way to assess applicants.

### What we are looking at

We are after the same style and quality of code as you would write at work on production code, so correctness and readability are important. At a minimum we expect unit tests (ideally written using TDD) and source code committed using Git. Each part of your implementation should be committed separately, resulting in a sensible commit history with detailed commit messages.

You may use any testing framework. You should supply the implementation as a Visual Studio 2015 solution which we can easily run. You can use any third party libraries you require from NuGet. You need to use C#.

The requirements are deliberately open ended. You should have questions about the requirements; assume a sensible answer and document the assumptions you make.

Please try to avoid including personally identifiable information in your code so we can review your code as impartially as possible.

### How to get the test results to us

Please create a private Git repository on bitbucket.org and add hireme@ecargo.co.nz (eCargo Careers) as a user with Read permissions. Push your completed work into this repository and then email us at hireme@ecargo.co.nz to let us know you have completed the test.

We would appreciate any feedback on the test process; if you wish, please include a feedback document in the Git repository with any feedback.

### Submission checklist

1. Visual Studio 2015 solution, written in C#, with unit tests, without any personally identifiable information.
2. A document containing questions you would've asked in the real world and any assumptions made.
3. An optional suggestions / feedback document.
4. All of the above committed to the private Bitbucket Git repository.
5. hireme@ecargo.co.nz added to the repository with Read rights.
6. Email sent to hireme@ecargo.co.nz to notify us that reviewing can commence.

## The test

### Description

The test is based on writing code to first describe and then transform a collection of nodes which are arranged in a tree. The node definitions are as follows:

```
public abstract class Node
{
    public string Name { get; }
    protected Node(string name)
    {
        Name = name;
    }
}

public class NoChildrenNode : Node
{
    public NoChildrenNode(string name) : base(name)
    {
    }
}

public class SingleChildNode : Node
{
    public Node Child { get; }
    public SingleChildNode(string name, Node child) : base(name)
    {
        Child = child;
    }
}

public class TwoChildrenNode : Node
{
    public Node FirstChild { get; }
    public Node SecondChild { get; }
    public TwoChildrenNode(string name, Node first, Node second) : base(name)
    {
        FirstChild = first;
        SecondChild = second;
    }
}

public class ManyChildrenNode : Node
{
```

```
    public IEnumerable<Node> Children { get; }
    public ManyChildrenNode(string name, params Node[] children) : base(name)
    {
        Children = children;
    }
}
```

There are three additional interfaces which you will implement as part of the test. These are:

```
public interface INodeDescriber
{
    string Describe(Node node);
}

public interface INodeTransformer
{
    Node Transform(Node node);
}

public interface INodeWriter
{
    Task WriteToFileAsync(Node node, string filePath);
}
```

**Part one - C# describer**

Write a describer implementing the `INodeDescriber` interface which will output a C# description of how to create the tree of nodes. The output should have node per line, indented with four spaces per nesting level.

```
INodeDescriber implementation = ...
var testData = new SingleChildNode("root",
    new TwoChildrenNode("child1",
        new NoChildrenNode("leaf1"),
        new SingleChildNode("child2",
            new NoChildrenNode("leaf2"))));
var result = implementation.Describe(testData);
// result is:
// new SingleChildNode("root",
//     new TwoChildrenNode("child1",
//         new NoChildrenNode("leaf1"),
//         new SingleChildNode("child2",
//             new NoChildrenNode("leaf2"))))
```

**Part two - transformer**

Write a transformer which will transform a tree of nodes into a matching tree that uses the correct node types (e.g. a `ManyChildrenNode` with no children should be transformed into a `NoChildNode`).

```
INodeTransformer implementation = ...
var testData = new ManyChildrenNode("root",
    new ManyChildrenNode("child1",
        new ManyChildrenNode("leaf1"),
        new ManyChildrenNode("child2",
            new ManyChildrenNode("leaf2"))));
var result = implementation.Transform(testData);
// result is equivalent to:
// new SingleChildNode("root",
//     new TwoChildrenNode("child1",
//         new NoChildrenNode("leaf1"),
//         new SingleChildNode("child2",
//             new NoChildrenNode("leaf2"))))
```

**Part three - integration**

Write a class that implements the `INodeWriter` interface. The class should be designed to receive any `INodeDescriber`, which is used to get a string representation of the tree of nodes. This string representation is then written to the specified file. You should provide at least one integration test using an off-the-shelf IoC / DI container.

```
INodeWriter implementation = ...
var filePath = ...
var testData = new NoChildrenNode("root");
await implementation.WriteToFileAsync(testData, filePath);
var result = File.ReadAllText(filePath);
// result now contains the output from whichever
// INodeDescriber that was injected.
```