Agile Address Book Report

Robert Ryterski

# Coding Standard

I used a number of guides for C# coding conventions provided by Microsoft. The different sections are available on the MSDN website:

<http://msdn.microsoft.com/en-us/library/ff926074(v=VS.100).aspx>

<http://msdn.microsoft.com/en-us/library/8a3x2b7f(v=vs.100).aspx>

<http://msdn.microsoft.com/en-us/library/ms229042(v=vs.100).aspx>

## Examples

### Layout

* Four space indents
* One statement per line
* At least one blank line between method and property definitions

#### Code

<https://github.com/RobertRyterski/AgileAddressBook/blob/master/AgileAddressBook/AgileAddressBook/Contact.cs#L34>

// Contact.cs, line 34

public string LastName

{

get

{

return this.\_lastName;

}

set

{

this.\_lastName = value;

this.OnPropertyChanged("LastName");

this.OnPropertyChanged("FullName");

}

}

public long Phone

{

get

{

return this.\_phone;

}

set

{

this.\_phone = value;

this.OnPropertyChanged("Phone");

}

}

### Capitalization

* Use Pascal case for classes, enumerations, events, properties, ...
* Use Camel case for parameters

#### Code

<https://github.com/RobertRyterski/AgileAddressBook/blob/master/AgileAddressBook/AgileAddressBook/MainWindow.xaml.cs#L43>

// MainWindow.xaml.cs, Line 43

private void editButton\_Click(object sender, RoutedEventArgs e) ...

<https://github.com/RobertRyterski/AgileAddressBook/blob/master/AgileAddressBook/AgileAddressBook/Contact.cs#L20>

// Contact.cs, Line 20

public string FirstName ...

public string LastName ...

### Namespace Naming

* Use Pascal casing
* Don’t use version names in namespace
* Don’t use the same name for a namespace and a type in that namespace

#### Code

<https://github.com/RobertRyterski/AgileAddressBook/blob/master/AgileAddressBook/AgileAddressBook/Contact.cs#L7>

// Contact.cs, Line 7

namespace AgileAddressBook

<https://github.com/RobertRyterski/AgileAddressBook/blob/master/AgileAddressBook/AgileAddressBook.Test/ContactTest.cs#L6>

// ContactTest.cs, Line 6

namespace AgileAddressBook.Test

### Field Naming

* Use Pascal case
* Use noun or noun phrases
* Don’t use prefixes for static vs. non-static

#### Code

<https://github.com/RobertRyterski/AgileAddressBook/blob/master/AgileAddressBook/AgileAddressBook/Contact.cs#L12>

// Contact.cs, Line 12

private string \_firstName, \_lastName, \_address, \_city, \_state;

private long \_phone;

private int \_zip;

### Use Lazily Evaluated Conditions

* Use && and || instead of of & and |

#### Code

<https://github.com/RobertRyterski/AgileAddressBook/blob/master/AgileAddressBook/AgileAddressBook/ContactWindow.xaml.cs#L72>

// ContactWindow.xaml.cs

if(\_mode.Equals("add") && context.FirstName != null)

{

\_contacts.Add(context);

}

### Short String Concatenation

* Use the + operator for short string concatenation

Code

<https://github.com/RobertRyterski/AgileAddressBook/blob/master/AgileAddressBook/AgileAddressBook/Contact.cs#L122>

// Contact.cs, Line 114

public string FullName

{

get

{

return FirstName + " " + LastName;

}

}

# Refactoring

I predominately used the list of refactoring rules from <http://www.refactoring.com/catalog/index.html>, but I included C# specifics from Resharper <http://www.jetbrains.com/resharper/features/code_refactoring.html>.

Naturally, this is not an exhaustive list of all changes that occurred to the source, but instead are the major changes that were documented refactoring techniques.

## Extract Clone Method

<https://github.com/RobertRyterski/AgileAddressBook/commit/95585e540128bd3f276b178b28cba4737883a18d>

### Rule

<http://www.refactoring.com/catalog/extractMethod.html>

### From

// ContactWindow.xaml.cs

// copy properties of selected contact for local edits

// global change will occur when OK is hit

context.FirstName = \_original.FirstName;

context.LastName = \_original.LastName;

context.Phone = \_original.Phone;

context.Address = \_original.Address;

context.City = \_original.City;

context.State = \_original.State;

context.Zip = \_original.Zip;

### To

// ContactWindow.xaml.cs

context = \_original.Clone();

// Contact.cs

public Contact Clone()

{

return new Contact(FirstName, LastName, Phone, Address, City, State, Zip);

}

## Extract Copy Method

<https://github.com/RobertRyterski/AgileAddressBook/commit/95585e540128bd3f276b178b28cba4737883a18d>

### Rule

<http://www.refactoring.com/catalog/extractMethod.html>

### From

// ContactWindow.xaml.cs

else if (\_mode.Equals("edit"))

{

\_original.FirstName = context.FirstName;

\_original.LastName = context.LastName;

\_original.Phone = context.Phone;

\_original.Address = context.Address;

\_original.City = context.City;

\_original.State = context.State;

\_original.Zip = context.Zip;

}

### To

// ContactWindow.xaml.cs

else if (\_mode.Equals("edit"))

{

\_original.Copy(context);

}

// Contact.cs

public void Copy(Contact other)

{

FirstName = other.FirstName;

LastName = other.LastName;

Phone = other.Phone;

Address = other.Address;

City = other.City;

State = other.State;

Zip = other.Zip;

}

## GetPhoneAreaCode to Property

<https://github.com/RobertRyterski/AgileAddressBook/commit/69819a3c0db9c17e74b8f15ea0a87d0825fffb9f>

### Rule

<http://www.jetbrains.com/resharper/features/code_refactoring.html#Convert_Method_to_Property>

### From

public int GetPhoneAreaCode()

{

return (int)(Phone / 10000000);

}

### To

public int PhoneAreaCode

{

get

{

return (int)(Phone / 10000000);

}

}

## GetPhoneExtension to Property

<https://github.com/RobertRyterski/AgileAddressBook/commit/69819a3c0db9c17e74b8f15ea0a87d0825fffb9f>

### Note

I also corrected the definition of extension to be the last four digits on the phone number.

### Rule

<http://www.jetbrains.com/resharper/features/code_refactoring.html#Convert_Method_to_Property>

### From

public int GetPhoneExtension()

{

return (int)(Phone % 10000000);

}

### To

public int PhoneExtention

{

get

{

int four = (int)((\_phone % 10000000) % 10000);

return four;

}

}

## Self Encapsulate Fields in PhoneOffice

<https://github.com/RobertRyterski/AgileAddressBook/commit/643c281c3fa5d8698e9024118e132995d4c5f6b6>

### Rule

<http://www.refactoring.com/catalog/selfEncapsulateField.html>

### From

public int PhoneOffice

{

get

{

int three = (int)((\_phone % 10000000) / 10000);

return three;

}

}

### To

public int PhoneOffice

{

get

{

int three = (int)((Phone % 10000000) / 10000);

return three;

}

}

## Self Encapsulate Fields in PhoneExtension

<https://github.com/RobertRyterski/AgileAddressBook/commit/643c281c3fa5d8698e9024118e132995d4c5f6b6>

### Rule

<http://www.refactoring.com/catalog/selfEncapsulateField.html>

### From

public int PhoneExtention

{

get

{

int four = (int)((\_phone % 10000000) % 10000);

return four;

}

}

### To

public int PhoneExtension

{

get

{

int four = (int)((Phone % 10000000) % 10000);

return four;

}

}

## Use Inline Temp in PhoneOffice

<https://github.com/RobertRyterski/AgileAddressBook/commit/bc8f75093124c6a7f42aacebc15c6ec9e62a7860>

### Rule

<http://refactoring.com/catalog/inlineTemp.html>

### From

public int PhoneOffice

{

get

{

int three = (int)((Phone % 10000000) / 10000);

return three;

}

}

### To

public int PhoneOffice

{

get

{

return (int)(Phone % 10000000 / 10000);

}

}

## Use Inline Temp in PhoneExtension

<https://github.com/RobertRyterski/AgileAddressBook/commit/bc8f75093124c6a7f42aacebc15c6ec9e62a7860>

### Rule

<http://refactoring.com/catalog/inlineTemp.html>

### From

public int PhoneExtension

{

get

{

int four = (int)((Phone % 10000000) % 10000);

return four;

}

}

### To

public int PhoneExtension

{

get

{

return (int)(Phone % 10000);

}

}

# Test Driven Development

I used the unit testing tools built into Visual Studio. The test cases can be run directly from within Visual Studio by navigating to Test > Run > All Tests in Solution. They can also be run by the command line tool MSTest provided with Visual Studio.

The output of the test cases can be found in Jenkins, on a per-build basis, at the end of the build log (“console output”).

## Test Cases

1. ContactConstructorAllArgumentsTest
2. ContactConstructorNameArgumentsTest
3. ContactConstructorNoArgumentsTest
4. ToStringTest
5. FullNameTest
6. PhoneAreaCodeTest
7. PhoneOfficeTest
8. PhoneExtensionTest
9. PhoneStringTest
10. CloneTest
11. CopyTest
12. FirstNameNotifyTest
13. LastNameNotifyTest
14. PhoneNotifyTest
15. AddressNotifyTest
16. CityNotifyTest
17. StateNotifyTest
18. ZipNotifyTest

# Configuration Management

I used Git and GitHub for my configuration management.

Git ([http://git-scm.com](http://git-scm.com/)) is a distributed source control solution developed by Linus Torvalds for the Linux kernel. The collection of software is primarily used through a command line interface, although GUI programs have been developed on top of it. Note that there is no official “checking out” of files with git. The distributed model git uses does not explicitly lock files for modification. Anyone can create a local copy of the project repository by using git clone work on the source as they please. User controls, branching, and other features all play a part, but are beyond the scope of this report.

GitHub (<https://github.com>) is a project hosting service built around git. They provide features like bug tracking and wiki systems for each project. There is also a Windows Github application that integrates with the website, provides a git client, and GUI frontend for that client.

## Example

This example is for the typical command line interaction with git. GUI clients may vary in implementation, but maintain the same concepts. For the Github Windows application specifically, a shell can be launched from the tools section where these commands can then be executed.

## clone the project repository

$ git clone https://github.com/RobertRyterski/AgileAddressBook.git

## change to project folder

$ cd AgileAddressBook

## check status, no changes

$ git status

# On branch master

nothing to commit, working directory clean

## modify a file

$ your\_favorite\_editor AgileAddressBook/AgileAddressBook/Contact.cs

## check status, changes

$ git status

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: AgileAddressBook/AgileAddressBook/Contact.cs

#

no changes added to commit (use "git add" and/or "git commit -a")

## add changes to be committed

$ git add AgileAddressBook/AgileAddressBook/Contact.cs

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: AgileAddressBook/AgileAddressBook/Contact.cs

#

## commit those changes

$ git commit -m "A message"

## git output about the insertions and deletions goes here

## push changes to remote server

$ git push origin master

## git output about transmission to server goes here

# Continuous Integration

I used the Jenkins continuous integration system (<http://jenkins-ci.org/>) with the following plugins:

* git <https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin>
* GitHub API <https://wiki.jenkins-ci.org/display/JENKINS/GitHub+API+Plugin>
* GitHub <https://wiki.jenkins-ci.org/display/JENKINS/Github+Plugin>
* MSBuild <https://wiki.jenkins-ci.org/display/JENKINS/MSBuild+Plugin>
* MSTestRunner <https://wiki.jenkins-ci.org/display/JENKINS/MSTestRunner+Plugin>

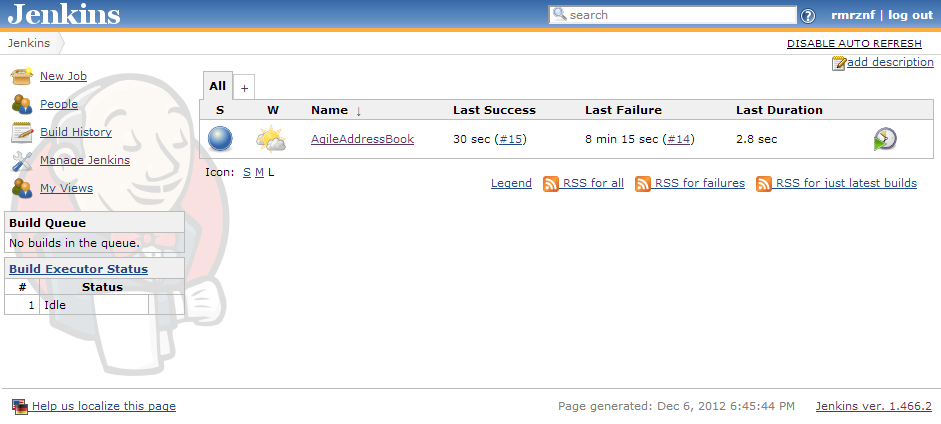
The jenkins folder in the project directory contains a skeleton configuration and the job I created for the project. The build logs and job configuration can be found inside the job folder.

The Jenkins job I created has several parts:

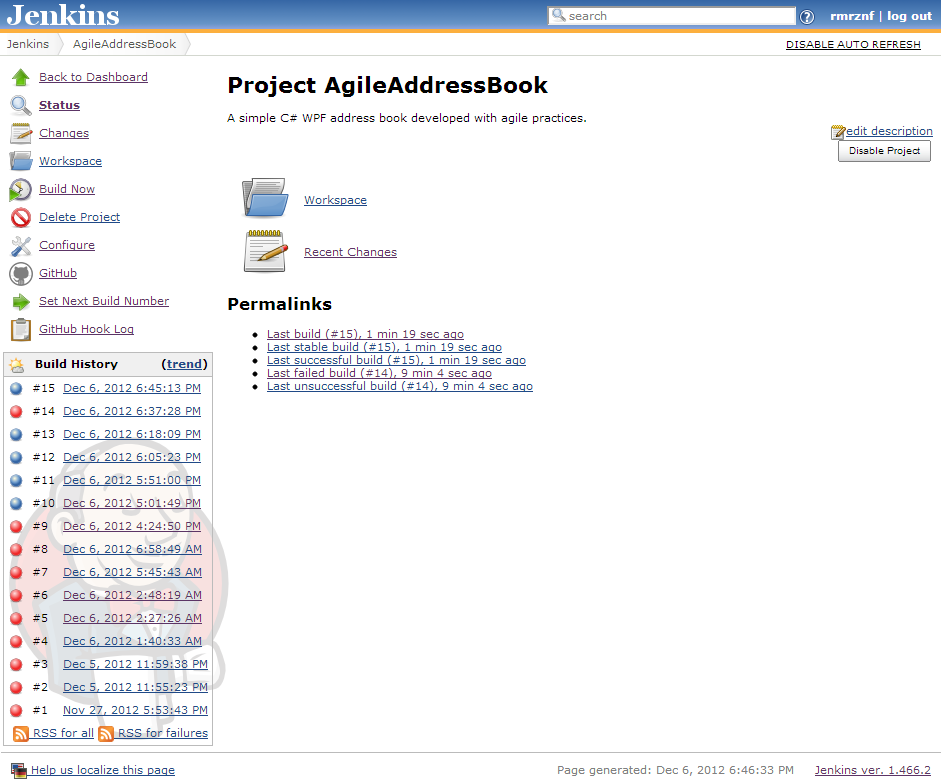
The most important part is how the project integrates with the source control system: the git plugin. This plugin handles executing the git client for Jenkins so that a local copy of the source code can be maintained. The job is configured to use the repository URL provided by GitHub.

At its core, the job defines a set of commands (build steps) to run every time a build is requested. The first step use the MSBuild plugin to compile the project with the Microsoft build program MSBuild. The second step uses MSTest (Microsoft’s test runner) to execute the tests defined in the project.

The automated part of the system comes into play with the GitHub plugins. Thanks to a feature of GitHub, the repository is set up to make a special call to the Jenkins server (via HTTP request) whenever a code change is pushed. The plugins then tell Jenkins to begin a build, as this job is configured to run on every code change.



The main Jenkins interface.



The job overview for this project.