Implementing a code analysis tool for Visual Studio 2010

Yuriy Hohan[1], Ludmila Luchianova[2]
1. Technical University of Moldova, Faculty of Computers, Informatics and Microelectronics, str. Stefan cel Mare 168, Chisinau MD-2004 Moldova
E-mail: yuriihohan@gmail.com
2. Technical University of Moldova, Faculty of Computers, Informatics and Microelectronics, str. Stefan cel Mare 168, Chisinau MD-2004 Moldova
E-mail: lluchianova@yahoo.com

***Index Terms* —** code quality, refactoring, flexibility, MVC pattern**.**

1. Introduction

1.1 The formulation of the problem

Code analysis is a very important issue nowadays. In late 80s people only tended to try to make their software qualitative. Nowadays it is a must for a professional software developer and is one of those critical criteria which separate the best from the rest. Code quality is a scientifically studied question with a big enterprise background today bringing such massive systems as NDepend or FxCop to the community table. These giants are really difficult to fight against because of their achievements but their products have their slight shortcomings too. These disadvantages can be their nonlinear curve of mastering them, their language which is far from being ubiquitous [1], the high price in the case on NDepend. There is also one more peculiarity which walks along the above mentioned: quality is a perceptual, conditional and somewhat subjective attribute and may be understood differently by different people. Anyhow we insist that there are some commonly recognized criteria for detecting bad code but that does not mean that one cannot think of his own ones. The described system tackles all these issues.

1.2 The formulation of the requirements

To solve the code criteria problem a very flexible approach was chosen – there is a possibility to load/unload assemblies describing the search of bad code into the system with the possibility to enable/disable and also set-up each search method. Of course the classes in these assemblies should correspond to certain rules – these rules are written in the interfaces which the search classes must implement (interfaces are currently the best choice and such inventions as dependency injection (DI) and inversion of control (IOC) [2] are there to support their firm positions).

A sample library was written to serve two purposes. It is both a good way to show the idea of how to implement the necessary interfaces and a proof that there are some general criteria which cannot be debated and which are well–recognized in the field. This assembly includes 9 search methods described by Martin Fowler and Kent Beck [3] with parallel hierarchies and greedy methods being one of the most interesting among them. Basically this library searches for what they call “bad smell” code. This term was chosen by Fowler and Beck while being on a trip to Europe. One night they were thinking of a proper name for such code: if something looks bad, you can close your eyes and not see it anymore, if something sounds ugly, you can close your ears but sooner or later you will have to use this code again and the new part of its smell will remind you of its quality. It is fair to mention that bad code should be refactored and having your system written in an undesired way may significantly decrease the possibility to extend it by adding new features and even lead the project to its death. Refactoring itself is a way to restructure the code without changing its behavior. Three steps are necessary to implement a refactoring: find the portion of bad code, refactor it and test the behavior of the system in order to make sure it is unchanged. The second step is really automated with all the variety of refactoring plug-ins for the majority of modern IDEs including the community – acclaimed plug-ins by JetBrains. The third step is also automated with a lot of tools like NUnit and RhinoMocks and such approaches as TDD and BDD.

2. Implementation

2.1 Chosen tools

The add-in was implemented in Visual Studio 2010 and can be distributed and installed through simple .vsix packages. The interface is built using WPF technology and is integrated into the IDE as a common tool window. The add-in was written in Visual Studio 2010 and tested using its new feature called experimental feature which completely separates the states of test and development instances. The info about the current “bad smell” registries is stored using SQLite database and is accessed ultra–fast C#-SQLite library. The parsing of the code is done either via built–in DTE features or using Roman Ivantsov’s great open-source project called Irony depending on the level of details needed. Visual Studio built-in modeling tools have been used to build the UML model.

2.2 Extension and flexibility

The add-in was written in such a way that it could correspond to the requirements of extension and flexibility to the maximum extent possible. This compartment will describe the way it was achieved in detail.

2.2.1 Adding libraries

One can add any number of libraries to the system but all the libraries should meet certain criteria.
`Criteria 1`. They should reference and use the `CommonLibrary` shipped with the add-in and containing all the necessary classes for its functionality.
`Criteria 2`. Every "bad smell" should have a model, a view and a controller attached to it. This was implemented using the MVC pattern.
1. All the controllers should be marked as `[BadSmellClass]` and should inherit from `BadSmell` class.
2. All the models should inherit from `BadSmellOptions` class.
3. All the views should implement `IOptionView`.

2.2.1.1 BadSmell class

This is the controller part. There are three methods every class looking for bad code must implement:
1) `FindSmell` - it searches for all the sequences of bad code and returns the list of found bad smells. The task which stands in front of the developer writing this method is filling the return list with the registries found by his algorithm. The reference of DTE is passed but one must not necessarily limit himself to the info from the DTE.
2) `GenerateOptions` - this method must build its model from its view. This means it must fill all the option fields of the model with the information from the view entered by the user.
3) `DefaultOptions` - must set model's options to defaults. Basically this method determines the default model which is used by the system until the user decides to change it.

2.2.1.2 BadSmellOptions class

This class represents a lot of template methods which are used for saving, serializing, etc. the model. For instance, the model class is stored in the database in binary serialized way. Anyhow its children must implement just one method:

```
public Dictionary<string, object>
GetValues().
```

This method must return all the set values wrapped in the Dictionary collection.

2.2.1.3 IOptionView interface

This is the view part. There are three methods to implement in it:

1) `Draw` - implements the view drawing logic, this method is responsible for drawing the option controls in the right place on the parent control.
2) `GetValueFromGUI` - returns the dictionary of GUI values but not the model. This is due to an attempt to completely separate the view and the model. The separation differs from one implementation of MVC pattern to another.
3) `FillGUI` - fills the GUI controls with the values taken from the abstract model which is passed in the parameters.

2.2.2 Extending help

One can extend the help just by changing the help directory in the add-in root folder. This version of add-in supports just *.txt format for files but the number of subdirectories is unlimited.

2.2.3 Further flexibility

The flexibility is achieved not only by vast extension possibilities but also through the possibility of setting-up every search method, enabling and disabling it, recommendations of the possible refactoring method which can be used, the possibility of viewing the problematic piece of code, grouping the registries and also making search using different criteria.

Extra flexibility is added by the possibility of noting a registry as AWARE. Thus it is placed in this category and viewed only if necessary. This is made because of the fact mentioned above: quality is a perceptual, conditional and somewhat subjective concept. So there is also room to ignore some coding standards due to some subjective reasons.

2.3 Imlemented algorithms

2.3.1 Greedy methods search

The method belongs to the class if it is implemented in the class it is being called from or in its base class. A borrowed method is a method which does not belong to the class it is being called from. An own method is considered to be either a method belonging to the class it is being called from or a statement in the method which is currently being evaluated.

This algorithm searches the methods to find out whether they do use many borrowed methods without adding a significant number of their own methods. The number of borrowed and owned methods is entered manually. A method is considered a borrowed method if it is not contained in this class or its base classes.

The algorithm works in two steps: during the first traverse of the parse tree it builds a dictionary of all own methods for every class, during the second traverse it counts the number of borrowed and own methods and if their number exceeds the limits, adds the bad smell code registry to the system.

2.3.2 Parallel hierarchy search

Ignoring the rules of this bad smell might lead to a “combinatorial explosion” which basically means the exponential growth of the number of classes without really adding some functionality. Such smell is certainly the result of someone's carelessness or lack of knowledge of OOP and must be avoided.

There are two variations of this algorithm: search based on naming convention and search based on methods added on the next level of hierarchy. Anyhow the first steps are the same: the class tree is divided into levels and each branch having *m* classes is compared to each other branch containing *n* classes using combinations. If derived class set minus base class set for two classes in different hierarchy branches are equal this is considered to be a parallel hierarchy. In the case of named based search the sets contain the parts of camel case class name split by the capital letters. In the case of methods based search the sets contain the methods belonging to the two classes in different hierarchy branches.

3. Conclusion. Future plans

The current system corresponds to practically all starting requirements – it is flexible, it is extendable, it is cleanly built using design patterns.

The feature of duplicated code search was not implemented yet. As it seems this feature might be one of the central in the add-in. We plan to implement the algorithm by Fei Ma [4, 5] but there is a need to set-up the current grammar to support AST tree building, and we are looking for a person which is good at building the formal grammars for this task.

The current movement tracking system could be changed to a more intelligent one because thus far this system hardly can be called a tracking one.

4. Used literature

1) Evans, E., "Domain-Driven Design - Tackling Complexity in the Heart of Software", 2004, Addison-Wesley

2) Robert C. Martin "Design Principles and Design Patterns", 2000

3) Fowler, Martin "Refactoring. Improving the Design of Existing Code", 1999, Addison-Wesley.

4) Fei Ma, Christopher W. Fraser, William S. Evans «Clone detection via structural abstraction», 2009, Department of Computer Science, University of British Columbia, Vancouver, Canada V6T1Z4.

5) F. Ma. «On the study of tree pattern matching algorithms and applications». Master's thesis, Department of Computer Science, University of British Columbia, the direct reference: https://circle.ubc.ca/bitstream/handle/2429/18318/ubc_2006-0557.pdf?sequence=1