Improving the software quality - an educational approach

Violeta Bozhikova, Mariana Stoeva, Bozhidar Georgiev and Dimitrichka Nikolaeva
Department of Software and Internet Technologies, Faculty of Computer Sciences and Automation,
Technical University of Varna, 1 Studentska str., 9010 Varna, Bulgaria
vbojikova2000@yahoo.com, mariana_stoeva@abv.bg, bojidar_g@mail.bg, dima.nikolaeva@abv.bg

Abstract – **The term "quality software" refers to software that is easy to maintain and evolve. The presence of Anti-Patterns and Patterns is recognized as one of the effective ways to measure the quality of modern software systems. The paper presents an approach which supports the software analysis, development and maintenance, using techniques that generate the structure of Software Design Patterns, find Anti-Patterns in the code and perform Code Refactoring. The proposed approach is implemented in a software tool, which could support the real phases of software development and could be used for educational purposes, to support "Advanced Software Engineering" course.**

Keywords – **Software Engineering, Software Design Patterns, Software Anti-Patterns, Software Refactoring**

I. Introduction

The presence of Anti-Patterns and Patterns is recognized as one of the effective ways to measure the quality of modern software systems. Patterns and Anti-patterns are related [1]. The history of software production shows that Patterns can become Anti-patterns. It depends on the context in which a Pattern is used: when the context become inappropriate or become out of date than the Pattern becomes Anti-pattern. For example, procedural programming, which was Pattern at the beginning of software production activity, with advances in software technology gradually turned into Anti-pattern. When a software solution becomes Anti-pattern, methods are necessary for its evolution into a better one. Refactoring is a general way for software evolution to a better version. This is a process of source code restructuring with the goal to improve its quality characteristics without changing its external behavior. In refactoring we replace one software solution with another one that provides greater benefits: code maintainability and extensibility are improved, code complexity is reduced.

The paper proposes an approach (Fig.1) that improves the software quality. The approach supports software analysis, software implementation and maintenance. It provides techniques that generate the structure of Software Design Patterns (DP), find Anti-Patterns in the code and perform Code Refactoring. As stated in [4] refactoring can be applied in different contexts; "improving the software quality" it is the high level context. This paper focuses on two specific refactoring contexts: "refactoring, to remove bad code constructions" and "refactoring, to implement design patterns".

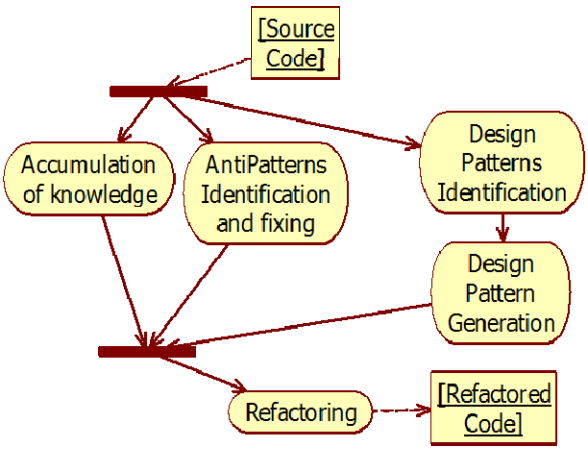

Fig.1 The general structure of the approach

The proposed approach is realized in a software tool that covers real phases of software development (Fig.2) and could be used by software engineers in their real work. The main purpose of the developed tool, however, is - to be an educational Instrument within the disciplines "Software Engineering" and "Software design Patterns" in "Software and Internet technologies" Department of the Technical University in Varna.

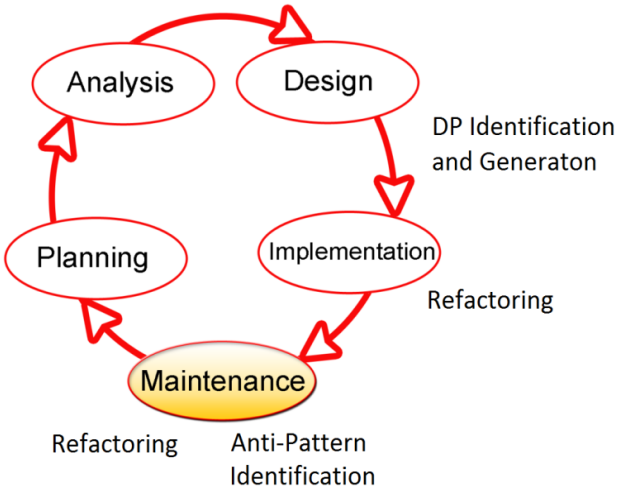

Fig.2 The approach supports real phases of software development

978-1-5386-1753-3 /17/$31.00 ©2017 IEEE

Recent studies show that the development of refactoring tools is an active research ([1]÷[10]). At the same time, such tools are not enough used in practice. One possible raison is that most existing refactoring tools focus only on improving some quality metrics. Case technology leads to significant improvement of the software process, but not all aspects of software production could be easy automated. It is because software development is creative and team-based activity; large projects waste a lot of time for interaction between participants. These aspects of software production cannot be automated.

II. Our Approach

The general structure of our approach is presented in figure 1. It takes as input the software source code that has to be refactored. The output is refactored code. The approach relies on accumulation of knowledge about the best practices in programming so "Accumulation of Knowledge" is one of the processes that are performed in parallel with other processes. The refactored code is result of "Anti-Patterns Identification and fixing" and "Design Patterns Generation". Before generate Design Patterns it is necessary to analyze the code with the goal to find Design Patterns candidates. The proposed approach comprises the following main components:

- Accumulation of Knowledge: aims to provide information on design pat-terns and anti-patterns. It contains information about creational, behavioral and structural design patterns and software anti-patterns in software development and software architecture. Describe the problems that each design pattern solves, the advantages that it provides and the situations in which it is used. For the anti-patterns - the nature of the problems and possible options for their avoidance are described.
- Design Pattern Generation: provides functionality to generate sample implementations of the design patterns structures; basic elements and relationships between them are generated, it's not implementation of the solution of specific problem. To generate a template user must select appropriate names for key elements. Appropriate names for the key elements must be given by the user, in order to generate a pattern.
- Refactoring Component: provides methods for automatic code refactoring. The code is supplied as input of any method, as for inputs are accepted only properly constructed classes. Each method performs the appropriate changes and returns the modified code as a result.
- Anti-Patterns Identification and fixing: provides methods for code analysis. The code is supplied as input of a particular method and as result of code analysis the poorly constructed sections of code are colored. The colored code should be rewritten in order to increase its readability and maintenance.
- Design Pattern Identification: provides methods to examine source code and to identify candidates for design patterns. This component is still un-der development. Our detection strategy is based on the code inspection. Extensive research has to be conducted to develop techniques to automatically detect candidates of DP in the code.

III. Implementation

Based on the approach proposed a web system was developed. The system architecture is based on Model-View-Controller (MVC) pattern. Block "controllers" (Fig.3) includes five controllers managing the different sections of the application. „HomeController" manages the Main Page; „WikiController" has a function to display the different pages of the educating section Encyclopedia; „IdentificationController" includes methods for code analyzing and detects poorly parts of the analyzed code; „RefactoringController" contains methods and logic for code refactoring. „TemplatesController" includes methods for input data processing and for the construction of various patterns. Block "Views" contains views for each of the controllers and a general view section that contains common pieces of code for the controller's views.

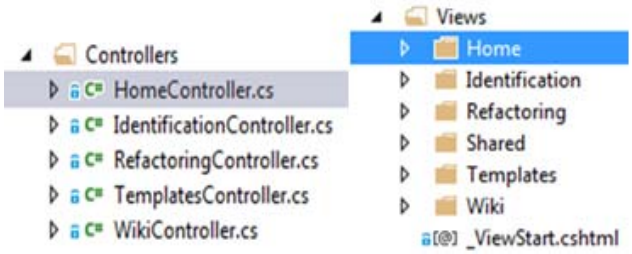

Fig.3 Controllers and Views

All views use Razor technology of Microsoft, which allows insertion of C# code in html pages. Upon receipt of a request for a page, the server compiles C # code and in response sends html content - result of the compilation. The main sections of the web system are presented in Fig.4.

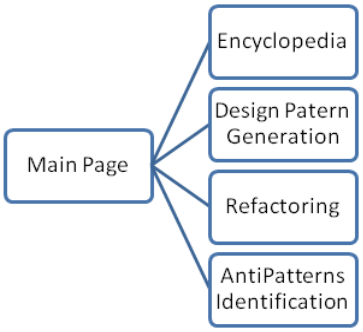

Fig. 4 Basic structural elements of the web system

- Main Page: it aims to present the different sections of the system with a short description and redirect the user to any of them.
- Encyclopedia: it aims to provide information on design patterns and anti-patterns. It consists of two parts: menu type accordion and informative part. The user can select from the menu a concrete DP or anti-pattern. When you choose a concrete pattern then the information about it is dis-played in the informative part. The section describes the problems that each design pattern solves, the advantages it provides and the situations in which it is used. For anti-patterns – their nature and options to be avoided are described. This section is realized as one page

with dynamic content that is changed through asynchronous AJAX requests to the server.

• Design Pattern Generation: this section offers functionality to generate sample implementations of the structure of the design patterns (Fig.5). The user chooses a type of pattern for example “Prototype”, inputs its parameters and click button "Generate". The generated code is displayed below the form.

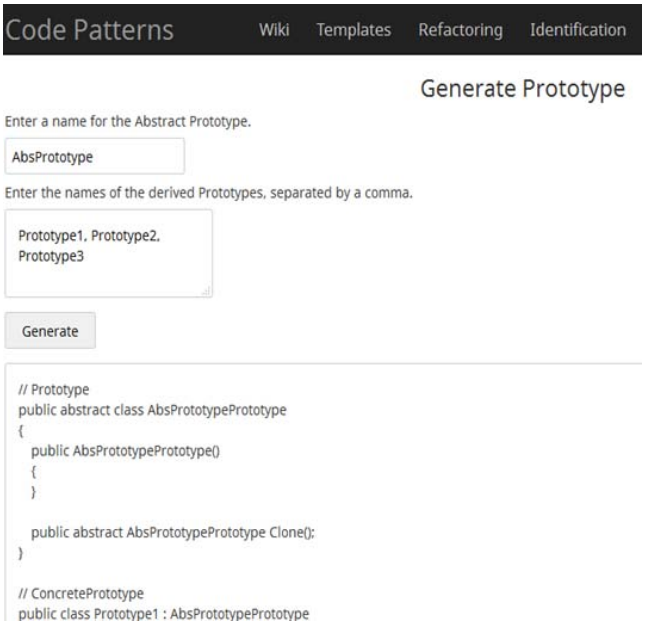

Fig.5 Prototype Design Pattern Generation

Refactoring: this section provides methods for automatic code refactoring. Each method performs the appropriate changes of the code and the modified code is returned as a result. In the left section of the refactoring window, the user puts the code, which must undergo refactoring. After entering the necessary parameters the user has to press button "Refactor". The refactoring code is displayed in the right pane (Fig.6).

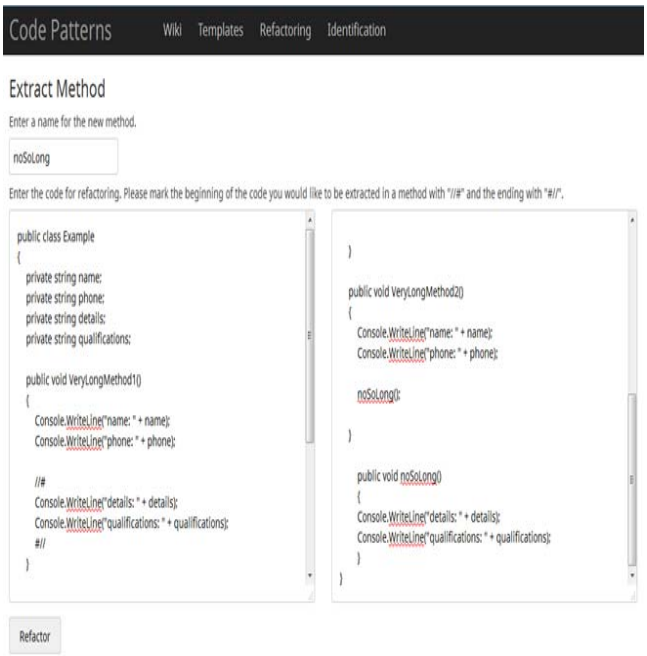

Fig.6 Code Refactoring

• Anti-pattern Identification: this section offers methods for code analysis with the goal to detect anti-patterns. The code is supplied as input to each method, which analyzes and paints the poorly constructed code sections. Then poorly constructed pieces of code must be rewritten to improve code readability and maintenance. Selecting a method (for example “Complicated If’s” – Fig.7) a page for entering the code for analysis is visualized. After entering the necessary input data the user must press the button "Identify". The program will process the code and will paint the problematic code parts in red (the result is shown in the right section): Design of libraries of components for re-use;

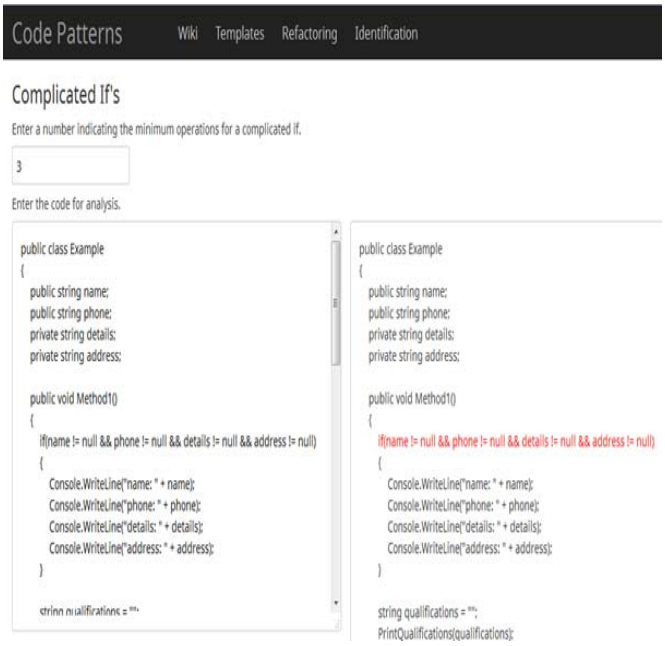

Fig.7 Anti-pattern Identification

IV. Conclusion and Future Progress

An approach that improves the software quality is proposed in the paper (Fig.1). “Refactoring” is the most important module of the approach. It is a general way for software evolution to a better version, i.e. general way to increase the software quality. Two specific contexts of refactoring are addressed by our approach: "refactoring, to remove bad code constructions" and "refactoring, to implement design patterns". The presence of Anti-Patterns and Patterns is recognized as one of the effective ways to measure the quality of modern software systems. In these contexts, the approach focuses on:

• Studying anti-patterns (code smells) to support their identification - module "Anti-pattern Identification".

• Studying patterns, to increase their effective use - module "Design pattern generation".

Actually, refactoring in the contexts addressed improves the quality of the software but more can be done in this direction. Software Refactoring includes many more contexts [5].

• Future work is needed to cover more refactoring contexts.

A Case tool (Fig.4÷Fig.7) is created that implements the approach proposed.

- Currently our DP detection strategy is not automated. Extensive research is necessary to develop techniques to automatically detect candidates of Design Patterns in the code.

We plan further:

- To add new patterns and anti-patterns in encyclopedic part.
- To add new methods for automatic refactoring and Anti-Pattern identification.
- To add support for various programming languages.

REFERENCES

[1] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, Thomas J. Mowbray, AntiPatterns. Refactoring Software, Architectures, and Projects in Crisis, John Wiley & Sons, Inc., 1998, Canada

[2] https://en.wikipedia.org/wiki/Systems_development_life_cycle

[3] https://en.wikipedia.org/wiki/Code_refactoring

[4] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Mel Ó Cinnéide, Kalyanmoy Deb, Katsuro Inoue, A Multi-Objective Refactoring Approach to Introduce Design Patterns and Fix Anti-Patterns, http://sel.ist.osaka-u.ac.jp/lab-db/betuzuri/archive/990/990.pdf

[5] Martin Drozdz, Derrick G Kourie, Bruce W Watson, Andrew Boake, Refactoring Tools and Complementary Techniques, https://pdfs.semanticscholar.org/ae2a/5ccaf697880cb386046e8882a6c268e83312.pdf

[6] Jagdish Bansiya, Automating Design-Pattern Identification, Dr. Dobb's Journal, 1998, http://www.drdobbs.com/architecture-and-design/automating-design-pattern-identification/184410578

[7] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, A comparative study of manual and automated refactorings, in 27th European Conference on Object-Oriented Programming (ECOOP), 2013, pp. 552–576

[8] M. Kim, T. Zimmermann, and N. Nagappan, A field study of refactoring challenges and benefits, in 20th International Symposium on the Foundations of Software Engineering (FSE), 2012, pp. 50:1–50:11

[9] Xi Ge and Emerson Murphy-Hill. Manual Refactoring Changes with Automated Refactoring Validation. In Proceedings of the Int. Conf. on Soft. Eng. (ICSE), 2014

[10] Ioana Verebi, A Model-Based Approach to Software Refactoring, https://www.researchgate.net/publication/281686403_A_Model-Based_Approach_to_Software_Refactoring