ReLE – A REFACTORIZING SUPPORTING TOOL

Stanimir Stoyanov, Asya Stoyanova-Doycheva, Ivan Popchev*, Mincho Sandalski

(Submitted on February 15, 2011)

Abstract

The paper describes a Refactoring eLearning Environment (ReLE), which is intended to analyze and assess programming code, based on refactoring rules. The Refactoring Learning Environment architecture includes an intelligent assistant – Refactoring Agent, which is responsible for the analysis and assessment of the code, written by students in real time by using a set of refactoring methods. According to the situation and based on the refactoring method, which should be applied, the agent could react in different ways. The ReLE architecture has been presented as well. The operation of the environment is demonstrated by an example, typical of its application.

Key words: Intelligent Agents, Refactoring, eLearning Environments, Software Engineering, Agent-Oriented Architectures

2000 Mathematics Subject Classification: 97C80 Technological tools and other materials in teaching and learning

Introduction. The existence of a large number of legacy systems and the necessity of their improvement for the purposes and needs of their users gives rise to the creation of a specific process in software engineering, called reengineering. Reengineering is demanded in different situations when the software needs to undergo evolution [1]. Here are some examples for the necessity of reengineering: the division of monolithic software into separate modules with the objective of their easier management, enhancement of the software productivity or portability, the use of new technologies or a change in the clients’ requirements. When the software constantly adapts or modifies, its source code becomes more complicated.

The authors would like to acknowledge the support of the Bulgarian Research Fund (Project Ref. No. ДО02-149/2008).
and its initial architecture and structure lose their identity. For this reason, the basic part of the software development price is focused on their support [2]. In the application of well-known and effective software development approaches such as the iterative, evolutionary and others, a solution has not been found in connection with the code complexity. This is so because in these approaches the software engineers’ efforts concentrate on developing new requirements while at the same time the software has to be supported [3]. For the solution of the code complexity problem, within the reengineering there, the special techniques of restructuring [4] and refactoring [5] emerged.

At the Faculty of Mathematics and Informatics of the University of Plovdiv a Distributed eLearning Center (DeLC) has been created [6, 7]. DeLC includes an eLearning environment for provision of teaching material and education services in an interactive, proactive and personalized way. A tool, extending the environment with special techniques applied in the Master’s software engineering programmes, is presented in the paper. The tool, called Refactoring eLearning Environment (ReLE), assists the acquisition of the special technique of refactoring. An overview of the environment has been presented in [8]. The rest of the paper is organized as follows: Section 2 briefly presents the refactoring technique and reviews existing supporting tools, Section 3 describes the ReLE architecture, Section 4 presents an intelligent agent, called RA, which is the kernel of the environment, in Section 5 some implementation issues are discussed, and finally Section 6 concludes the paper.

Refactoring. The main goal of refactoring is to improve the design of existing code. In [5] the refactoring is presented as a sequence of small transformations of the source code in a software system, as they preserve the external behaviour of the software system and lead to considerable source code restructuring. Each transformation can be presented as a pattern that is called “refactoring”. An example for refactoring named ‘Extract Class’ has been described as:

- **Condition:** You have one class doing work that should be done by two classes.
- **Motivation:** A class has many methods and quite a lot of data. A class is too big to be understood easily.
- **Action:** Create a new class and move the relevant fields and methods from the old class into the new one.

The activities connected with the source code transformation are defined as follows [5]:

- Localization of the places in the source code that have to be refactored.
- Choosing the applicable refactoring method from a repository of refactoring pattern according to the source code.
- Guaranteeing that the applied refactoring will preserve the behaviour of the software system. For this purpose we create a unit test about this place in the source code.

S. Stoyanov, A. Stoyanova-Doycheva, I. Popchev et al.
• Applying refactoring.
• Executing unit tests after refactoring.
• Determining the influence of refactoring on the software quality characteristics (complexity, intelligibility, support possibilities) or on the process (productivity, price, efforts).
• Coordination support between the code, to which refactoring is applied, and other software artifacts (project documents, requirements specification, tests, etc.)

Although the refactoring process could be realized by hand, the possibility of applying automatic tools is of great importance. At present, a number of such tools are available, where the aspect and degree of the process automation vary depending on the particular tool and the maintenance it supports. Tools such as the Refactoring Browser \cite{9}, XRefactory \cite{10}, and jFactor \cite{11} apply a semi-automatic approach, after which the place and type of refactoring are chosen by the user. Completely automatic refactoring, according to some scientific researches, is also an acceptable approach. Guru, for example, belongs to this category and is used for restructuring hierarchies of successors and methods for refactoring in SELF programmes \cite{12}. Some other approaches for automatic refactoring are presented in \cite{13-16}. A current tendency in this field consists in the integration of refactoring tools in powerful, industrial environments for software development. Such is the case with Smalltalk Visual Works from v7, Eclipse from v2, Together Control Center from v6, IntelliJ IDEA from v3, Borland JBuilder from v7, etc. All these tools focus on applying refactoring in compliance with the user requirements. Another group of tools, which are less in number by comparison with the previous ones, afford the opportunity to define when and where to apply refactoring. In \cite{17} an approach is presented after which the implementation is realized via metrics, whereas in \cite{18} the possibility of automation via invariants by means of the tool Daikon is described. The latter approach is based on a dynamic analysis of the behaviour of the run-time of the system and its most proper application is as a complement to the other approaches.

The proposed environment differs from the existing tools in several aspects:
• The environment is a prototype and is intended, first of all, for teaching students.
• The code analysis is done in real time, i.e. already during the development of the code the students could be assisted in improving its quality.
• An agent-oriented implementation is realized.

**ReLE Architecture.** The ReLE architecture consists of two components presented in Fig. 1. Front-end FE(JADE) is the environment used by the students for the development, compilation and testing of the source code. The kernel of Back-end BE(IDE) is the Refactoring Agent (RA), an intelligent agent that assists the students during the code development. RA is an autonomous software application that continuously analyzes and assesses the code developed in
FE(JADE). So, from the RA’s point of view, FE(JADE) is its operational environment. RA communicates with its environment by means of its sensors and effectors. Via the sensors RA accesses the complete source code. This implies not only the files, being edited, but also the completed ones that were not opened in FE for editing. In this way, the agent could make a detailed analysis and give an adequate assessment for the required changes on the basis of all the code, and not only of the part that is currently modified. The sensors also provide some basic metric information to the agent, which is used for initial filtering of the possible refactoring methods that can be further evaluated. The possible metrics are LOC (Line of Code) per class/method, number of methods/attributes per class, and so on. The role of the effectors is to raise different events that assist the students by the accomplishment of their tasks in FE(JADE), where they are working. Such events could be:

- Emphasizing particular parts of the code by highlighting with an appropriate colour.
- Displaying messages in dialog boxes, tooltips, etc.
- Emitting sound-signals, vocal messages.
- “Materializing” the agent in the form of animation to exalt the effect.

The collaboration of the sensors and effectors is coordinated by the Local Control of the agent, which is based both on the information, incoming from the sensors, and the refactoring rules, stored in the *Refactoring Knowledge Base* (RKB) of the agent. The analysis of the source code, written by the student in FES, is made by the RAnalyzer. Before the RAnalyzer starts its work, the RParser parses the source code and creates a tree structure from it. This tree structure can be analyzed by the RAnalyzer. The RKB consists of a set of rules together with a set of classes, which build a consistent knowledge base.
Each rule describes in a common form the conditions, which allow a particular refactoring method to enter the “short list”, based upon some metrics. For example, a possible rule for choosing the “Extract class” refactoring method could be LOC\_by\_class $\leq$ predefined\_value, which actually means that the refactoring method will be fired when the class becomes too large (depending on the predefined value). In this way, the rules are used by the RAnalyzer in order to make the initial filtering of the refactoring methods, which should be shortlisted at the next step.

Each refactoring class contains the code for the particular refactoring method, as well as a code for the final evaluation of the possibility of applying this refactoring method. The refactoring methods filtered by the RAnalyzer are then examined by using the evaluation part of each refactoring class. In this way, the agent takes a final decision about the applicable refactoring method and location of application. At the last step, the refactoring is applied by using the actual refactoring class after a negotiation with the user – as described in the next Section.

**Refactoring Agent (RA).** As it was mentioned, the kernel of ReLE is an intelligent agent providing the students with assistance in refactoring. Its main task consists in checking the code, which is being developed by the students in FE, and appropriately displaying instructions for improving its quality, whenever necessary. Depending on the refactoring method, which should be applied, the agent could react in three different ways (Fig. 2):
Automatic refactoring – to apply automatically the method after receiving a confirmation from the user.

Refactoring proposal – to display detailed instructions, explaining to the user where and how the particular refactoring method should be applied.

Refactoring questionnaire – to ask the user additional questions in order to clarify the conditions and define the appropriate refactoring method.

In the cases, the application of a refactoring method is relatively simple, the agent could offer an automatic code update. For example, Move Method, Move Field, Extract Class, Extract Method are appropriate methods for this kind of agent’s reaction. By the Move Method the agent’s behaviour is following:

- On the basis of the refactoring rules in RRB the agent finds out that there is a method in class A, that uses resources mainly from class B, which implies the application of the Move Method.
- The agent displays a message, in which it offers the user to move this method to class B.
- In case the user agrees, the agent moves the whole method from class A to class B by:
  - correcting all references to the resources in class A, so that they are accessible from class B, and adding the required parameters to the method if needed;
  - substituting all method invocations, so that they use its new position in class B.

The intervention of the user in this process is possible in case he/she would like to correct the code of the method once it has been replaced in the other class.

In case the criteria for refactoring are simple but the application of the particular method implies a significant change in the code or structure, the agent could deliver detailed information about the located code piece and its possible improvements. Refactoring methods belonging to this category are Replace Conditional with Polymorphism, Replace Delegation with Inheritance, Replace Inheritance with Delegation. For example, the agent’s behaviour by Replace Conditional with Polymorphism is the following:

- The agent finds out, there is a situation, where the Replace Conditional with Polymorphism method can be applied.
- The agent marks the code piece and displays a tooltip pointing out required refactoring.
- If the user demands detailed situation description the agent delivers a recommendation in the form of improvement steps.

A refactoring method could be chosen after a dialog with the user as given below:

- The agent finds out that a given class contains a numeric type code, but cannot determine if this code changes the behaviour of the class.
The agent asks the user whether the code piece influences the behaviour of the host class.

If the answer is “no”, the agent offers the user to apply the Replace Type Code with Class Method.

If the answer is “yes”, the agent asks the user whether the type code attribute changes during the lifecycle of the object.

If the user’s answer is “no”, the agent offers the user to apply the Replace Type Code method with Subclass.

Otherwise, the agent offers the Replace Type Code with a State/Strategy.

ReLE Implementation. ReLE uses an integrated technology, which consists of Integrated Development Environment (IDE) Eclipse [19] and JADE [20] environments. JADE is a development environment for the creation of agent-oriented applications. Eclipse is an IDE (Integrated Development Environment), offering a powerful mechanism for interaction with external components in the form of plug-ins. Integration between the two environments is possible since they operate on the Java Virtual Machine. The agent’s sensors and effectors are realized as plug-ins in Eclipse, thus implementing a real integration between the two environments. RA has access to the source code of Java projects and to the functionality presented by Eclipse, for example graphical components and various APIs. Besides, a JADE agent can be activated for analysis and it changes the Eclipse IDE code. The agent can dynamically alter its behaviours, which realize the logic for analysis and change.

To demonstrate the use of ReLE, we will give an example with the application of the Inline Temp refactoring method. The purpose of the application of this method is to minimize the unnecessary use of local variables. This is achieved when all the references to a local variable are replaced by an expression providing the variable value. An essential condition for the use of RA is to install its sensors and effectors in Eclipse IDE, which is indicated by means of the presence of a corresponding button (enclosed in frame) in the Eclipse Toolbar (Fig. 3, top). When the button is first pressed, RA is activated and its further work is performed in the following manner:

- Step 1 – “Generating of a syntax tree”. The agent’s sensor scans its environment (the source code in the active Java editor) every 5 s, at which a syntax tree is generated, presenting the transitory structure of the actual class. The syntax tree is searched for local variables, which can be replaced by expressions, providing their values.
- Step 2 – “Identification of local variables”. The appropriate local variables are highlighted in order to be distinguishable to students (Fig. 3). Furthermore, in the left vertical ruler, icons show up for every row, containing a declaration or use of an identified local variable. In the right vertical ruler there appear markers, which are helpful for fast scrolling of the editor’s contents to the appropriate row.
• Step 3 – “Activating a dialog”. When selecting an icon on the left, the corresponding code is highlighted and a dialog is activated, providing various options for students (Fig. 4). The first option is the one, which is offered by the refactoring agent. When this option is selected, all instances of usage of the local variable are replaced by the corresponding expression, while at the same time its declaration is removed.

• Step 4 – “Deactivating RA”. When the button is pressed again in the Eclipse toolbar, RA is deactivated and the icons and markers on the left and on the right are removed.

The knowledge base contains the rules for determining the situations, in which the various refactoring methods can be applied. Each rule represents a separate refactoring method. Finding formalism for presenting the semantics of the rules is a serious challenge. For this reason, every rule is realized as a separate Java class, the behaviour of which demonstrates the operational semantics of the rule. For example, every class of the knowledge base contains a realization of the corresponding refactoring method and an evaluation of the possibility to apply this method (the evaluation part).

The suggested two-step mechanism for selecting an appropriate refactoring method (described in 3 above) is realized as follows:

• At the first step RAnalyzer defines the subset of refactoring methods, which have to be evaluated.
At the second step these methods are investigated by the evaluation part of the classes, which realize the corresponding refactoring methods in the knowledge base.

In this way the agent takes a final decision about which refactoring methods to use and in which places in the student’s code to apply them. The current implementation of the RA knowledge base is given in the package diagram in Fig. 5.

![Diagram of the RA knowledge base](image)

The main package of the knowledge base is called “Pattern”. It contains a common functionality for the rules and the refactoring methods. Each refactoring method is a set of classes, which inherits and extends the abstract functionality from the “pattern” package in such a way as to reach the needed refactoring effect. The set of classes for the refactoring methods is placed in separate packages for every method.

For the implementation of a new refactoring method it is necessary to create two new classes, which implement the specific behaviour. So far, we have investigated 32 refactoring methods from [5], which can be implemented by the current architecture of the refactoring agent.

**Conclusion.** The implementation of the Refactoring Agent shows that the two environments – Eclipse and JADE – can work together and that their APIs could be exploited to provide the desired behaviour:

- constant analysis of the source code on site (within the Java editor of Eclipse);
- highlighting the portions of code, which could be refactored, according to analysis results and proposing options to the student;
- changing the source code.

We are going to implement a new version of the Refactoring Agent that offers more services to the end users.
REFERENCES


*Institute of Information and Communication Technologies
Faculty of Mathematics and Informatics
University of Plovdiv
4001 Plovdiv, Bulgaria

Bulgarian Academy of Sciences
Acad. G. Bonchev Str., Bl. 2
1113 Sofia, Bulgaria

S. Stoyanov, A. Stoyanova-Doycheva, I. Popchev et al.