An Interactive Interface for Refactoring Using Source Transformation

Scott Grant, James R. Cordy

School of Computing
Queen's University, Kingston, Canada
{grant,cordy}@cs.queensu.ca

Abstract

In this paper we present RUST, an environment that assists the programmer in locating potential code improvements by searching for a set of predetermined patterns, identifying these areas as code smells, and providing a means to automatically implement refactorings. Results and changes are presented using an interface that allows the user to examine the affected code, try out and see the results of applying potential refactorings, and accept or back out of changes after performing them. This reversible system allows for interactive programmer-driven modifications to the source code in which programmers can test out refactorings before committing to them.

1. Introduction

Code smells are metaphors for areas of code that may benefit from refactoring. Traditionally, the task of identifying these smells has fallen to the programmer, who must manually search through source code looking for potential problem areas. Thus far, it appears that no set of metrics can rival informed human intuition [1]. However, it is clear that large programs with hundreds of thousands of lines of code would benefit from some automated support for detecting and refactoring these code smells.

In this paper, we present RUST, which stands for Refactoring Using Source Transformation. RUST is an environment that assists the programmer in locating potential problems by performing transformations directly on the source code and attaching information embedded in XML tags. By specifying a general smell using a pattern matching rule, our tool can help to isolate code smells and bring them to the attention of the programmer, who can then determine whether or not they are valid concerns. We provide a point-and-click way to automatically perform the suggested refactoring in the environment itself using similar source transformation methods. In addition, the interface allows the user to back out of refactorings after performing them, allowing for interactive programmer-driven modifications to the source code. This reversible refactoring implementation allows programmers to test out refactorings before implementing them. We also provide a means to summarize the suggestions, which greatly benefits large projects.

In our previous work [2] we noted the need for leaving decisions on software maintenance in the hands of the programmer, not the automation, in order to help overcome practical barriers to industrial adoption of maintenance automation. To this end, we have developed a prototype interactive interface for refactoring C++ code in which the programmer is presented with refactoring opportunities automatically identified using code smell patterns as a set of possibilities highlighted using colour and summarized in a concise list. The programmer can then choose to implement any of the refactorings by clicking on the highlighted section, which causes the system to automatically implement the refactoring and present the result. While all steps are fully automated using source transformation patterns and rules, all decisions are left in the hands of the programmer, who may choose which refactoring possibilities to act upon, try them out, and undo them as they choose. In this way the system assists, rather than prescribes, the maintenance task.

RUST is comprised of a collection of HTML web pages with PHP and Perl scripting using TXL for source transformation. TXL [3, 4] is a programming language specifically designed to support computer software analysis and source transformation tasks. C++ was chosen as it is a common language in wide use today, with a valid and well tested TXL grammar already available. However, this process generalizes to other input languages or transformation tools without much difficulty. The environment is built with PHP scripting under the Apache Web Server. Perl is used to generate custom transformations when necessary. TXL is used on the server-side to perform the transformations. Our choices for the scripting languages lie mainly in their convenience. PHP and Perl are only used for their basic advantages, and can be replaced with other languages if necessary.

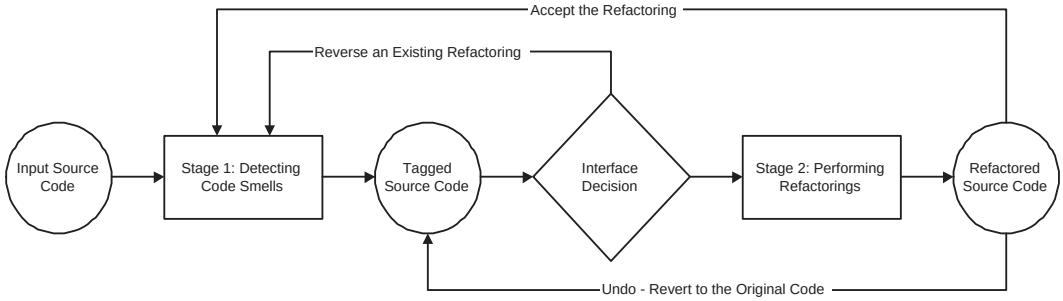

Figure 1. The Flow of Data

Our refactoring process has two primary stages. We are first presented with input source code, which is parsed and analyzed for code smells in the first stage. If any are found, we mark them up using XML tags, resulting in tagged source code. This tagged source is presented to the interface, from which point refactoring suggestions can be made. The tagged source can be passed to the second stage, which converts the tagged portions of the code into refactored versions. Output source code can then either be used again as input source code for the first stage if the refactoring is accepted, or the code can be reverted to the previous tagged source code.

---

2. Refactoring by Source Transformation

2.1. Detecting Code Smells

As seen in Figure 1, our approach works in two stages. In the first stage, we are interested in searching for code smells residing within the input source code. By writing a collection of rules to identify predetermined code smells, it becomes possible to automatically search for the features in code that are traditionally tracked down manually by programmers.

Emden and Moonen [5] differentiate between *primitive smell aspects* that can be observed directly in the code, and *derived smell aspects* that are inferred from other aspects. Primitive aspects allow very clear and unambiguous transformation rules to be written. Derived aspects are certainly more complicated in any approach, but code smells like those that can be extracted from fact bases are equally possible by reiterating the analysis. TXL has been used in several fact extraction projects [6, 7], and experience has shown that the necessary parsing and fact generating transformations are fast and easy to perform.

Refactoring by source transformation lends itself very well to primitive smell aspects. Since we are constructing a parse tree, we need only specify the pattern that we are looking for, perhaps using simple metrics, in order to identify the area of interest. Derived smell aspects require a slightly different approach. The source transformation approach to detecting code smells requires two features to be identified: the pattern that we are looking for, whether it is a block of code, or features found within the code itself; and the way that the code block should be marked up using XML tags. Performing additional steps such as fact extraction can provide more information that can be used in the smell detection for derived smell aspects.

2.2. Performing Refactorings

In order to perform our refactoring transformations, we must start by searching for the code smell markups found in the first stage. These will take the form of blocks of code surrounded by appropriate XML tags, as described in Section 2.1. Once a code smell has been identified, we can offer the option of automatically performing the suggested refactoring.

The general layout of a rule to perform a refactoring in our implementation will need two main features. First, we need to roughly specify what the format of the previously identified smell will look like. Since we have already marked up the section with XML tags, this task is very easy. Figure 2 shows two possible examples that we could match. Secondly, we need to specify the actual change that must be made to the source code in order to perform the refactoring. One of the major benefits of XML is that it allows us to implement an essentially transparent markup of refactoring information in the interface. It is possible to tag code smells and refactored code using XML tags, and replace them with HTML colour tags directly within RUST. An example can be seen in Figure 3, where indicates a highlighted markup.

The act of performing a refactoring on a block of code has been explored for over ten years now [8]. Automatic source code transformations that perform these refactorings at a user's request have also been in development for a num-

```
int ReallyBigMethod ()
<longMethod> {
    // A lot of code omitted
} </longMethod>

<stv id="1" line="1">relevantStatement1;</stv>
irrelevantStatement;
<stv id="1" line="2">relevantStatement2;</stv>
```

Figure 2. Sample XML markups

By wrapping the interesting area in XML blocks, we indicate to future stages of our environment that we have detected an area that may benefit from refactoring. The first shows a method body wrapped in XML tags that indicate a long method. The second shows a Split Temporary Variable refactoring, spanning several locations in the code. We can indicate each particular stv instance with a unique id, and each relevant line in the refactoring by a line parameter.

---

ber of years (for example, [9]), and many refactorings are now performed by programs rather than by programmers. Source transformation, as shown above, can clearly be used to perform many refactorings. The benefit to our refactoring technique is that the transformations are reversible; if it is unclear which refactoring solution will work for a given problem, RUST can be used to try refactorings, back out of undesirable ones, and start anew from any point.

3. Presenting Information Effectively

A fundamental design feature of RUST is the efficient presentation of information. In order to best convey our results, we decided to use framed hypertext pages. Figure 3 shows the general layout. Initially, the user is presented with a framed window with the source input on one side and the output on the other. The input source code always resides on the left side of the screen, while the parsed and transformed code with summary information is presented on the right.

Section is the source code listing. While we currently use a text box, allowing for very easy editing and cut-and-paste operations, we have also successfully experimented with a directory navigation frame, allowing full source files to be sent directly to the parser. is the parsed and tagged source output, with code smell highlighting and method markups; each method is tagged with a hypertext link to examine specific methods directly. is a code smell that has been detected and highlighted in the source output. The font colour has been altered, and a notification comment has been inserted in the code. Adding the refactoring suggestion as a comment retains the ability of the source output to be compiled. is the summary listing, providing a concise overview of the refactoring suggestions with hypertext links to the code smell location in the right frame. In this example, the link points to the code smell located at .

Once the code has been marked up with XML tags, we use a transform to remove the source code and only keep the core refactoring recommendations. This information becomes our summary, which can then be displayed at the top of the page (④ in Figure 3). In order to make this summary truly useful, we attach links to each summary line that point to unique HTML anchors in the source output. This allows the user to click on the summary line, and be immediately directed to the relevant refactoring. By providing links to these points in the summary, we allow the user to click on any summary line that seems interesting and be shown the smell without having to scroll through the entire source itself. This method works well for large projects. If a project summary is generated, links to both the specific source file and the anchor will provide a clear identification of any detected code smells.

4. Related Work

Our environment provides the capability to automatically perform refactorings. This approach has been explored as early as Brant and Roberts' Smalltalk Refactoring Browser [9]. The success of this browser has prompted a number of other tools, including IDEA by IntelliJ [10] and jFactor by Instantiations [11]. These tools focus on the code transformation itself, but do not have great support for indicating when and where a particular refactoring should be applied. The real benefit of the RUST interface is the ability to interactively decide which of several refactorings may be ideal in a given situation. Since the process is reversible, users have the ability to move back and forth between refactorings. The difference between reversing a transformation using RUST and a simple undo feature is that our XML markups allow a refactoring to be undone at any point after its introduction. We simply store the information about what was changed and how, and can present the user with the ability to reverse the process at their discretion, even after several other transformations have been made.

5. Summary

We have discussed the benefits and design of RUST, a web-based environment that is based on code smell detection. We have shown how it is possible to detect code smells using source transformation, and how to automatically refactor those suggestions directly within our environment. We have also illustrated its ability to back out of refactorings that may later be undesirable. We show how this method can be extended into a powerful tool with the capability to detect a wide variety of code smells.

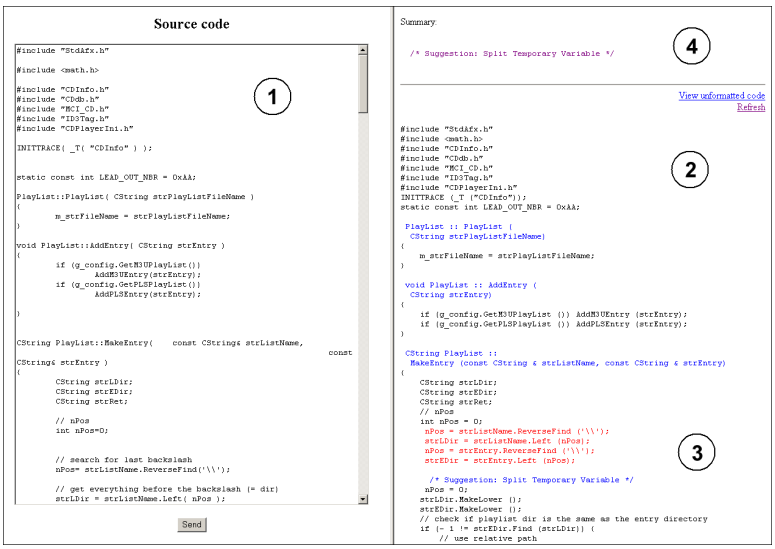

Figure 3. The RUST Environment

The layout of RUST as it appears in the browser. Section 1 is the source code listing, section 2 is the source output, section 3 is a highlighted code smell that can have a refactoring automatically performed by the environment, and section 4 is the summary of all detected code smells.

Refactoring has traditionally been a technique used by humans, as the automated detection of code smells has been ineffective compared to the careful eye of an experienced programmer. We seek to narrow the gap between the two methods by providing a means to identify code smells directly in the source. This can be accomplished by looking for features in the code. It is important to note that the patterns that we search for are not restricted to mere source code, but can also include features that can be extracted from the source code and embedded within the source in XML tags.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[2] J. R. Cordy, "Comprehending Reality: Practical Challenges to Software Maintenance Automation," *IEEE 11th International Workshop on Program Comprehension*, 2003.

[3] J. R. Cordy, C. Halpern, and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects," *Computer Languages*, vol. 16, no. 1, pp. 97–107, January 1991.

[4] J. R. Cordy, T. Dean, A. Malton, and K. Schneider, "Source Transformation in Software Engineering using the TXL Transformation System," *Journal of Information and Software Technology*, vol. 44, no. 13, pp. 827–837, October 2002.

[5] E. van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells," in *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE Computer Society Press, Oct. 2002.

[6] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider, "Source Transformation in Software Engineering using the TXL Transformation System," *Journal of Information and Software Technology*, 2002.

[7] J. R. Cordy, K. A. Schneider, T. R. Dean, and A. J. Malton, "HSML: Design Directed Source Code Hot Spots," *IEEE 9th International Workshop on Program Comprehension*, 2001.

[8] W. F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, Urbana-Champaign, IL, USA, 1992.

[9] D. Roberts, J. Brant, and R. Johnson, "A refactoring tool for Smalltalk," *Theory and Practice of Object Systems*, vol. 3, no. 4, pp. 253–263, 1997.

[10] JetBrains, *IDEA 3.0 Overview*, 2002.

[11] Instantiations, *jFactor Documentation*, 2002.