Refactoring Techniques and Automated Approaches Through Tool Support

Lisa Maria Kritzinger
Johannes Kepler University Linz
1255353
Email: kritzinger@gmx.net

Peter Feichtinger
Johannes Kepler University Linz
1056451
Email: shippo@gmx.at

Abstract

Refactoring, the restructuring of a software system without changing its semantics, is essential in software evolution. The notion of refactoring has been embraced by many object oriented software developers as a way to accommodate changing requirements. If applied well, refactoring improves the maintainability of software, but manual refactoring can be time-consuming and error-prone, so tool support is desirable when making large changes. In this article, we will explore a number of publications on automating different refactoring tasks, from just making code more compact to introducing objects into a C codebase.

Additionally, we will provide comparisons which will show the advantages and disadvantages of different approaches, regarding the performance of the software after refactoring, the applicability of a certain approach, or the simplicity of the application of an approach.

Keywords

Software restructuring, automatic refactoring, tool support, software evolution.

1. Introduction

Refactoring is the process of restructuring a software system without changing its semantics. It is used to increase readability and maintainability of software, reduce its complexity, or change the architecture of a system. Refactoring is essential in software evolution, because as a system is adapted to new requirements it inevitably becomes more complex and drifts away from its original design. This makes maintenance more difficult and reduces the software quality. Refactoring can then help to bring the system back to its original design and into a more maintainable state, improving code quality in the process.

However, manual refactoring without any tool support can be error-prone and time-consuming. There are various tools available for supporting elementary refactorings like renaming a variable or introducing an additional parameter to a function, often built into the used *Integrated Development Environment* (IDE) itself. But even with tool support, manual refactoring can still be too complicated or just tedious, and tool support for automating refactoring tasks is desirable in a number of cases.

In this article we're going to highlight some recent and not-so-recent contributions in the field of automatic refactoring, ranging from simple tasks like making code more compact [1], to more complicated tasks like introducing object-orientation into a C codebase [2].

2. Automated Refactoring Approaches

There are many refactoring tools available, most of which focus on performing specific refactorings as requested by the developer, other tools are based on search-based refactoring. Although there are already many tools for software refactoring, there are still techniques without tool support. The following section will give an overview of some papers which describe some techniques and tools by summarizing the main points.

2.1. Restructuring Legacy C Code into C++

This is an older paper from 1999 by Richard Fanta and Václav Rajlich of *Wayne State University* in Detroit, MI, USA [2]. They did a case study on the Mosaic browser, an early web browser implemented in C. Their approach uses a number of discrete refactorings. By combining those, a C `struct` or a number of related variables can be transformed into a C++ class, with related functions becoming member functions of that class.

The following two sections will briefly explain the implemented refactorings and their use in the whole restructuring process, respectively.

2.1.1. Refactoring Tools. This section briefly summarizes the implemented tools used in the restructuring process. Each tool has specific restrictions placed on when it can be applied, which simplifies the tool and makes sure the code is in a consistent state after its application.

1) The **variable insertion** tool inserts a selected variable into a class as a static or non-static member. The programmer needs to specify the variable which should be inserted, as well as the class—for a static variable—or the instance—for a non-static variable—it should be inserted into.
2) Another tool **makes access to a non-local variable explicit** by introducing an explicit parameter for the accessed variable, redirecting all accesses inside the function to that parameter, and finally adding an actual parameter for the variable at every call of the function.

The same is also possible in reverse to **make access implicit**.
3) To **add a parameter** to a function, another tool is used that adds the parameter to the formal parameter list. After selecting the instance to be passed as the new parameter for each call, the tool inserts that instance as a parameter to the call.
4) Finally there is a tool for **changing the access specifier of a class member**, which just checks that a certain change doesn't make the code inconsistent.

2.1.2. Restructuring Scenario. The refactoring tools described in the previous section are used at various steps in the whole process. The complete restructuring process thus involves both actions by a human as well as use of the tools, and is divided into three phases.

1) Data-only classes are created from a number of variables, if necessary. In case there is already a C `struct`, this step can be skipped.
2) After creating the desired classes, possible clones are removed. Clones may result from the same domain concept being implemented at various places in the code.
3) Lastly, the user specifies functions which should be added to a class as a member function. These may be functions that have the target class as a parameter, access it through a global variable, or have individual members of the target class as parameters.

2.1.3. Results. They selected a subsystem of 3000 lines from the Mosaic codebase for their tests. Of these 3000 lines they were able to encapsulate about 60% of the code into 12 classes. As an example where clone removal was necessary the give the URL class, which was extracted separately at five different locations.

2.2. Conditionals vs. Polymorphism

The argument that the cost of refactoring cannot be afforded is often made by programmers, because refactoring usually has a negative impact on the performance.

This trade-off has been investigated by Serge Demeyer in 2002 [3] by comparing the performance of a program which contains large conditionals against one where the conditionals were replaced by polymorphic method calls. This comparison showed that C++ programs refactored this way perform faster than their non-refactored counterparts.

This approach to refactor a system is mostly suitable for software which contains a large number of conditional logic and where performance matters. It is not always possible, or desirable, to replace conditional logic by polymorphism. But when the same conditional logic reappears in other pieces of code it becomes a maintenance problem, because one risks modifying one condition without adapting the other part as well.

There are three variants of complex conditional logic and ways to remove it mentioned and described in [3].

1) **Transform Client Type Checks** If a client is testing the type of a certain provider object, it can be refactored by moving code from the client to the provider. The special case that a client tests whether a provider is `null` or empty can be refactored by introducing a special *null object* [4].
2) **Transform Self Type Checks** The case that an object is testing an attribute serving as some kind of type-tag can be refactored by creating a new subclass for each leg of the conditional and moving the code as a polymorphic method into the subclass. If an object is changing its state dynamically, it can be refactored by introducing a state object [5, pp. 305–313].
3) **Transform Conditionals Into Registration** Another case are clients which test the type of a series of objects before performing a certain action. This can be refactored by a central registration mechanism, which acts as a mediator between objects providing services and clients requesting services.

Although this method is easy to use, there is no tool support to automate the process. Therefore it is difficult to motivate programmers to make use of this method.

2.3. Design Differencing

An automated refactoring approach is described in [6]. This novel refactoring approach refactors a program based both on its desired design and on its source code. The process of this method is illustrated in Figure 1. The programmer creates a desired design for the software. This design shall be based on the current software design and additionally on the understanding of how the software may be required to evolve. The software is then refactored using that design. The resulting software code has the same behavior as before the refactoring. That is, the whole algorithm is divided into a detection phase and a reification phase.

This method is supported by some tools, two of them are JDEvAn (Java Design Evolution and Analysis) [7] and Code-Imp (Combinatorial Optimization for Design Improvement) [8], [9].

2.3.1. JDEvAn. JDEvAn is developed at the University of Alberta. It is an Eclipse plugin which analyzes a software system's design-evolution history and provides information about the system's history. The plugin contains a Java fact extractor, a query-based change-pattern detection module, and a design differencing algorithm. The Java fact extractor extracts a UML logical design model from Java source code. The design differencing algorithm uses lexical and structural similarity to automatically recover differences between one version of a system and the next [10]. JDEvAn provides a refactoring-detection module which categorizes detected differences as refactoring instances [11].

The process initially extracts two UML models from the source code corresponding to two versions of a Java system. Afterwards the two models are compared and the differences

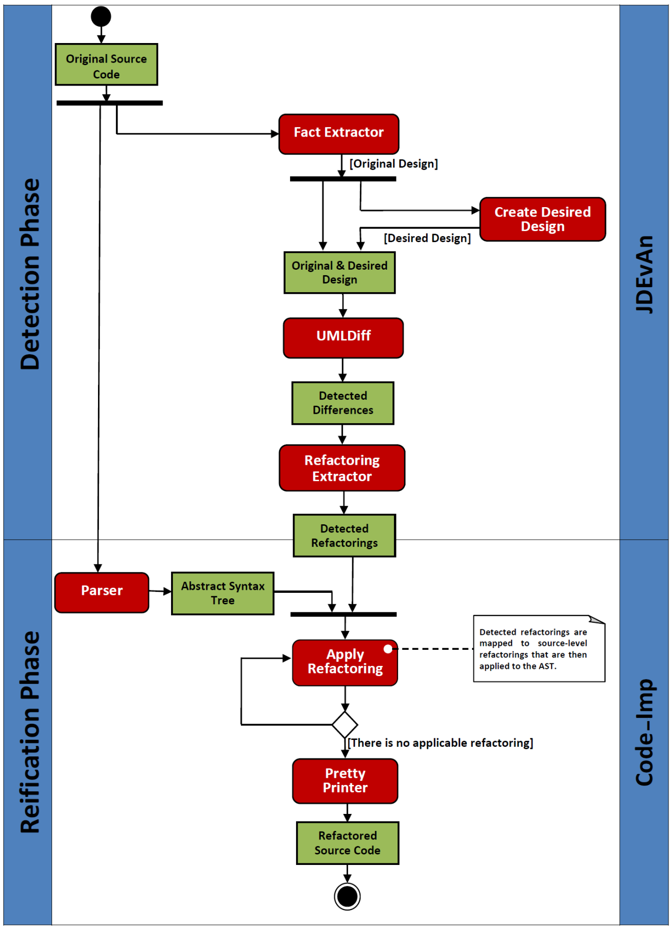

Figure 1: Automated refactoring process using Design Differencing [6]

between them are detected. Finally the detected differences are categorized as design-level refactoring instances. For example, a method which is moved from a class to a subclass in the desired design is detected as a number of *move method* differences.

2.3.2. Code-Imp. Code-Imp is developed by [8], [9] as a fully automated refactoring framework for automatically improving the design of existing programs. It takes Java source code as input and provides a refactored version of the program as output.

The refactoring process of Code-Imp is driven by a search technique. The search technique is steepest-ascent hill climbing, where the next refactoring to be applied is the one that produces the best improvement in the so called fitness function, a function which measures how good the program is.

The benefit of refactoring software using tools like JDEvAn and Code-Imp is that it enables automated refactoring towards a high-quality desired design, and hence improves maintenance productivity. The efficacy of the approach was additionally proven by [6]. Their findings were that the original program could be refactored to the desired design with an accuracy of over 90%, hence demonstrating the viability of automated refactoring using design differencing.

Listing 1: A generic class with methods generated by Eclipse

```
public class C0<T> {
  private T inner;
  public C0(T inner) {
    super();
    this.inner = inner;
  }
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result +
        ((inner == null) ? 0 : inner.hashCode());
    return result;
  }
  public boolean equals(Object obj) {
    if(this == obj)
      return true;
    if(obj == null)
      return false;
    if(getClass() != obj.getClass())
      return false;
    C0 other = (C0) obj;
    if(inner == null) {
      if(other.inner != null)
        return false;
    } else if(!inner.equals(other.inner))
      return false;
    return true;
  }
}
```

2.4. The Spartanizer

This is a recent paper by Yossi Gil and Matteo Orrù, which was presented at this year's SANER. It describes their tool called *The Spartanizer* [1], which is an Eclipse plugin for automatic refactoring of Java code. The tool is still actively being developed on GitHub[1] and has quite a number of contributors.

The supported refactorings are implemented in classes inheriting from the abstract `Tipper` class. A tipper represents a rewrite rule that can be applied to a specific type of AST node. Multiple tippers can be applied in succession to the whole project. An example is given in listings 1 and 2. Listing 1 shows a generic class with `equals` and `hashCode` methods generated by Eclipse, which are quite verbose for a class with only one member, because they are aimed at the general case of many members. The *spartanized* version of the class is shown in Listing 2, it is much more compact than the first version and has done away with much of the verbosity in the generated methods.

After each modification to a source file, it is parsed and available suggestions are displayed as info markers on the source code. From there, it is possible to apply a refactoring to a selected scope (method only, the whole file, etc).

1. https://github.com/SpartanRefactoring/Main

Listing 2: A spartanized version of the class in Listing 1

```
public class C1<T> {
  private final T inner;
  public C1(T inner) {
    this.inner = inner;
  }
  public int hashCode() {
    return 31 +
        ((inner == null) ? 0 : inner.hashCode());
  }
  public boolean equals(Object c) {
    return c == this ||
        c != null && getClass() == c.getClass() &&
        equals((C1) c);
  }
  private boolean equals(C1 c) {
    return inner == null ?
        c.inner == null : inner.equals(c.inner);
  }
}
```

This paper is unique in our list, because it describes a tool that is actively maintained and can be used in production.

3. Automated Refactoring in General

As we have already seen from just a few papers, the spectrum of automated refactoring is quite wide. Different techniques require more or less interaction from the user, and their scope ranges from localized changes (such as introducing polymorphism into a class hierarchy as seen in subsection 2.2) to changes to the whole software system (such as changing the design of the whole system as seen in subsection 2.3).

There are also different motivations for using automated refactoring, or refactoring in general. One might just want to make code more readable on the one hand, on the other hand the motivation might be that a system requires profound changes in order to make it more maintainable. Mens et al. identify three steps in the refactoring process [12] which are also illustrated in Figure 2:

1) detect when an application should be refactored;
2) identify which refactorings should be applied and where; and
3) perform the refactorings.

They also note that of those steps, only the third one is currently well supported by tools. This is still true today, more than 13 years later. We saw that some tools are able to identify where refactorings can be applied, as in the case of the *Spartanizer*, while others need to be guided by the developer.

Determining when a software system has a need for refactoring is supported by tools for the detection of *code smells*, symptoms of poor design and implementation choices [13]. Various such tools are already available. Palomba et al. use, for example, information from the change history of a software system to detect smells [14]. However, we think that combining this with the second point from above, actually identifying how a smell can be fixed by applying certain refactorings at specific parts in the code, is essential for the success of those tools.

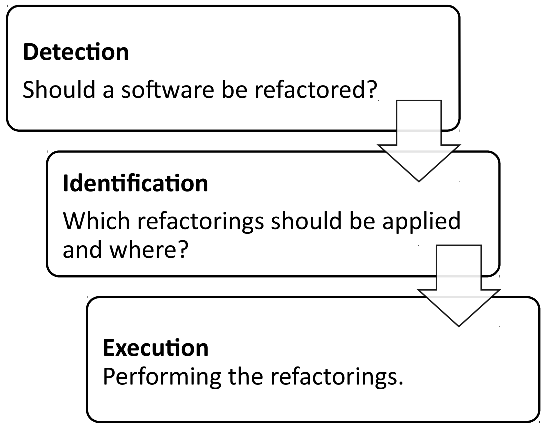

Figure 2: The refactoring process according to [12]

4. Conclusion

We showed various different approaches to automatic refactoring.

Fanta and Rajlich [2] use different refactoring tools in succession, guided by the user, to introduce classes into a C codebase. They require the user to specify which variables should be grouped into a C++ class and to specify which functions should be moved into that class.

Demeyer [3] showed that in many cases, replacing large conditionals by polymorphism to dispatch between different behaviors doesn't necessarily have an impact on performance and, if it does, then usually a positive one.

Moghadam and Ó Cinnéide [6] show how a software system can be automatically refactored based on a desired design and information extracted from the code. They use two different tools, one to extract the actual design from the software and generate necessary refactorings by comparing it to a desired design, the other to actually apply the generated refactorings to the software system.

Gil and Orrù [1] present a tool that can be used to automatically apply a large number of refactorings to (parts of) a system, with the goal to make its code more compact. The tool is integrated into the Eclipse IDE and provides opportunities for the user to apply one or more refactorings to different parts of the codebase.

In summary, we can say that a large number of automated tools are already available. These tools support developers in refactoring software, which is particularly important for the common problem of extending an existing program with new functionality. However, there are also techniques with much potential that are not supported by tools yet.

References

[1] Y. Gil and M. Orrù, "The Spartanizer: Massive automatic refactoring," in *IEEE 24th International Conference on Software Analysis, Evolution*

and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017, M. Pinzger, G. Bavota, and A. Marcus, Eds. IEEE Computer Society, 2017, pp. 477–481.

[2] R. Fanta and V. Rajlich, "Restructuring legacy C code into C++," in *1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30 - September 3, 1999*. IEEE Computer Society, 1999, pp. 77–85.

[3] S. Demeyer, "Maintainability versus performance: What's the effect of introducing polymorphism?" Lab. on Reengineering (LORE), Universiteit Antwerpe, Tech. Rep., 2002.

[4] R. C. Martin, D. Riehle, and F. Buschmann, *Pattern Languages of Program Design 3*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[6] I. H. Moghadam and M. Ó Cinnéide, "Automated refactoring using design differencing," in *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, T. Mens, A. Cleve, and R. Ferenc, Eds. IEEE Computer Society, 2012, pp. 43–52.

[7] Z. Xing. JDEvAn (java design evolution and analysis). University of Alberta. [Online]. Available: https://webdocs.cs.ualberta.ca/~stroulia/Zhenchang_Xing_Old_Home/jdevan.html

[8] M. K. O'Keeffe and M. Ó Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.

[9] I. H. Moghadam and M. Ó Cinnéide, "Code-imp: A tool for automated search-based refactoring," in *Fourth Workshop on Refactoring Tools, WRT 2011, Waikiki, Honolulu, HI, USA, May 22, 2011*, D. Dig and D. S. Batory, Eds. ACM, 2011, pp. 41–44.

[10] Z. Xing and E. Stroulia, "Differencing logical UML models," *Automated Software Engineering*, vol. 14, no. 2, pp. 215–259, 2007.

[11] ——, "Refactoring detection based on UMLDiff change-facts queries," in *13th Working Conference on Reverse Engineering, WCRE 2006, Benevento, Italy, October 23-27, 2006*. IEEE Computer Society, 2006, pp. 263–274.

[12] T. Mens, T. Tourwé, and F. Muñoz, "Beyond the refactoring browser: Advanced tool support for software refactoring," in *6th International Workshop on Principles of Software Evolution, IWPSE 2003, Helsinki, Finland, September 1-2, 2003*. IEEE Computer Society, 2003, pp. 39–44.

[13] M. Fowler, *Refactoring - Improving the Design of Existing Code*, ser. Addison Wesley Object Technology Series. Addison-Wesley, 1999.

[14] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, E. Denney, T. Bultan, and A. Zeller, Eds. IEEE, 2013, pp. 268–278.