Note 28

maximkazakov2005@gmail.com

Contents

1 A Comprehensive Survey of Software Refactoring

Authors: Tom Mens, Member, IEEE, and Tom Tourwe´
Date: February 2004

Abstract

This paper provides an extensive overview of existing research in the field of software refactoring. The research is compared and discussed based on several criteria: the refactoring activities supported, the specific techniques and formalisms used for these activities, the types of software artifacts being refactored, critical issues to consider when building refactoring tool support, and the effects of refactoring on the software development process. A running example is utilized throughout to elucidate and illustrate the main concepts.

Keywords

- Coding tools and techniques
- Programming environments/construction tools
- Restructuring
- Reverse engineering
- Reengineering

2 Introduction

In today's dynamic programming landscape, software must constantly evolve to meet new requirements and adapt to changing conditions. This evolution, however, often leads to increased complexity and a deviation from the software's original design, ultimately degrading its quality. Studies indicate that a significant portion of software development costs is allocated to maintenance, emphasizing the need for effective strategies to manage this complexity (Coleman et al., 1994; Guimaraes, 1983; Lientz & Swanson, 1980).

The challenge lies in the realization that enhanced methods and tools for software development tend to accommodate new features within the same time constraints, thus perpetuating the complexity spiral (Glass, 1998). To combat this, there is a pressing demand for techniques that incrementally improve the internal quality of software without altering its external behavior. This domain, referred to as restructuring or more specifically refactoring in the context of object-oriented software, seeks to achieve this goal (Chikofsky & Cross, 1990; Opdyke, 1992).

Refactoring is defined as the transformation of an object-oriented software system to improve its internal structure while preserving its external behavior (Fowler, 1999). The fundamental objective is to reorganize classes, variables, and methods within the class hierarchy to facilitate future adaptations and enhancements. Extending beyond just restructuring, refactoring also plays a crucial role in reengineering—analyzing and altering a software system to reconstitute it in a new form (Demeyer et al., 2002).

2.1 Structure of the Paper

This paper is structured as follows:

1. **Running Example**: Introduction of a running example to illustrate concepts.
2. **Refactoring Activities**: Identification and explanation of various refactoring activities.
3. **Techniques and Formalisms**: Overview of techniques supporting refactoring activities.
4. **Software Artifacts**: Different types of software artifacts that can be refactored.
5. **Issues in Tool Development**: Essential considerations when creating refactoring tools.
6. **Refactoring and the Software Development Process**: How refactoring integrates into the software development framework.
7. **Conclusion**: A summary of key findings and future directions.

3 Running Example

To ground our discussion, we present a running example that illustrates a typical nontrivial refactoring of an object-oriented design.

3.1 Initial Design

The initial class hierarchy, depicted in **Figure 1**, illustrates a `Document` class that is divided into three subclasses: `ASCIIDoc`, `PSDoc`, and `PDFDoc`. Each document provides preview and print functionalities, realized through methods in corresponding `Previewer` and `Printer` classes. However, each subclass implements preprocessing or conversion differently, leading to complexity and redundancy.

3.2 Need for Refactoring

This design is not optimal. For instance, adding new functionality, such as a text search or spell checker, necessitates updating all subclasses, which increases complexity and diminishes clarity. The design lacks explicit relationships between helper classes, complicating maintenance and development.

3.3 Refactored Design

To address these issues, we introduce the **Visitor Design Pattern**, as shown in **Figure 2**. This design consolidates helper classes and provides a common interface for them via a `Visitor` class hierarchy, improving understandability and modularity.

3.3.1 Implementation Steps

1. **Move Methods**: Move the `print` and `preview` methods from each subclass to the `Printer` and `Previewer` classes, respectively, using the **MoveMethod** refactoring.

2. **Rename Methods**: Rename these methods to appropriate visit methods to avoid name conflicts.

3. **Introduce Visitor Class**: Create an abstract `Visitor` class that establishes a superclass for `Printer` and `Previewer`, ensuring a cohesive design.

4. **Add Accept Method**: Implement an `accept` method in all document subclasses to facilitate interaction with the `Visitor` classes.

5. **Refactor Calls**: Modify the original print and preview methods to call the new `accept` method, thus centralizing functionality.

The above example illustrates the need for over twenty primitive refactorings, which serve as the foundational steps for creating composite refactorings that encapsulate more complex behavior.

4 Refactoring Activities

The refactoring process encompasses several distinct activities:

1. **Identifying Refactoring Candidates**: Determine where in the code refactoring is necessary.

2. **Selecting Refactorings**: Choose the appropriate refactoring techniques for identified areas.

3. **Behavior Preservation**: Ensure that the applied refactorings do not alter the software's external behavior.

4. **Implementation**: Apply the identified refactorings.

5. **Assessing Impact**: Evaluate the effects of refactoring on software quality (e.g., complexity, maintainability).

6. **Consistency Maintenance**: Achieve consistency between the refactored code and other software artifacts such as documentation and tests.

4.1 Identifying Refactoring Opportunities

Identifying where to apply refactorings can be done through various techniques, including:

- **Bad Smells**: Code structures that indicate a need for refactoring, as highlighted by Martin Fowler (1999).

- **Clone Analysis**: Tools that detect duplicated code, suggesting areas where refactorings may be beneficial (Balazinska et al., 2000).

4.2 Ensuring Behavior Preservation

Behavior preservation is critical when refactoring. It can be ensured through:

- **Testing**: Comprehensive test suites that validate against expected behavior.
- **Static Analysis**: Tools that check for compliance with preconditions and invariants before and after refactoring.

4.3 Assessing Quality Impact

Refactorings can be classified based on their effects on quality attributes like robustness, extensibility, maintainability, and performance. For instance, refactoring can enhance maintainability by reducing code duplication.

4.4 Consistency Maintenance

Maintaining consistency across software artifacts is vital. Changes in one artifact must reflect in others to avoid discrepancies. Approaches include using logic rules for automated consistency checks (Rajlich, 1997).

5 Techniques and Formalisms

Various techniques and formalisms facilitate the refactoring process:

5.1 Invariants and Preconditions

Refactorings often include invariants that must remain satisfied. Preconditions are established to ensure that behavior is preserved during the transformation process.

5.2 Graph Transformation

Refactorings can be conceptualized as graph transformations, where software artifacts are represented as graphs. This formalism aids in verifying properties such as behavior preservation and consistency (Mens et al., 2002).

5.3 Additional Techniques

- **Program Slicing**: Extracts relevant code segments that may influence specific behavior, aiding in behavior preservation during refactoring (Weiser, 1984).
- **Software Metrics**: Numerical measures that help evaluate software quality before and after refactoring (Demeyer et al., 2000).

6 Types of Software Artifacts

Refactoring is not limited to source code; it can also be applied to various software artifacts, including:

6.1 Programs

Refactoring techniques differ across programming languages. Object-oriented languages often provide more straightforward restructuring opportunities due to their encapsulated nature.

6.2 Designs

Refactoring at the design level, such as UML models, allows for higher abstraction and facilitates better structural understanding (Boger et al., 2002).

6.3 Software Requirements

Refactoring can also apply to requirements specifications, breaking them down into structured viewpoints, enhancing clarity and manageability (Russo et al., 1998).

7 Tool Support

Effective refactoring requires robust tool support. Various characteristics impact tool usability:

7.1 Automation

Automation levels vary among tools, with some offering full automation while others provide semi-automated support.

7.2 Reliability

Reliability hinges on the tool's ability to guarantee behavior-preserving transformations. A robust undo mechanism is essential for reverting undesired changes.

7.3 Configurability

A configurable tool allows users to adapt refactorings to their specific needs. Open extensibility mechanisms enable the addition of new refactorings and specifications.

7.4 Coverage and Scalability

Ideally, a refactoring tool should cover a wide range of activities. Composite refactorings can enhance scalability by encapsulating multiple primitive refactorings into a single operation.

7.5 Language Independence

A language-independent framework for refactoring can streamline the application of techniques across different programming languages (La¨mmel, 2002).

8 Process Support

Refactoring integrates seamlessly into various software development processes, including:

8.1 Software Reengineering

Refactoring plays a key role in the reengineering of legacy software, providing a systematic approach to restructuring while addressing significant challenges.

8.2 Agile Software Development

In an agile context, refactoring becomes a continuous practice, enabling developers to improve software incrementally while maintaining flexibility.

8.3 Framework-Based Development

In frameworks and product lines, refactoring may result in evolution conflicts, necessitating careful coordination and resolution of changes across multiple systems.

9 Conclusion

This paper offers a comprehensive survey of research in software refactoring, categorized by multiple criteria. It highlights the need for continued exploration of formalisms, processes, methods, and tools that can enhance the consistency, scalability, and flexibility of refactoring practices. While commercial tools have proliferated, ongoing research is essential to address the remaining challenges and shortcomings in the field.

Acknowledgments

This research was funded by the FWO Project G.0452.03 “A formal foundation for software refactoring.” We express our gratitude to Jean-Marc Je´ze´quel and the anonymous reviewers for their insightful feedback that greatly improved the quality of this paper.

References

(References as per the original text)