Building Refactoring Tools: Insights from RefactorEx

Gabriel Philippe Pereira, Lucas Francisco da Matta Vegi and Vladimir Di Iorio

Department of Computer Science – Universidade Federal de Viçosa (UFV)
Viçosa, MG – Brazil

pereira7346@gmail.com, {lucas.vegi, vladimir}@ufv.br

Abstract. *Refactoring plays a crucial role in software maintenance and evolution, yet emerging programming languages often lack the tooling to automatically and confidently perform the refactoring tasks. This paper presents insights gained from developing RefactorEx, a refactoring tool for the Elixir functional language. We discuss architectural considerations, design patterns and technical challenges encountered while implementing automated code transformations. The presented patterns and lessons aim to provide some guidance for researchers and developers seeking to build similar tooling.*

1. Introduction

Code refactoring—the process of restructuring existing code without changing its external behavior—has become a fundamental practice in software development [Fowler 2018]. While established languages, such as Java [JetBrains 2023] and Python [Rudi 2006], already possess mature refactoring tools for most editors, emerging languages often lack this support. This gap becomes particularly noticeable in functional programming languages like Elixir[1], which, despite gaining popularity for high-performance distributed systems, had no mature and widely adopted refactoring tool [Vegi and Valente 2025].

This paper presents key aspects of the development process of REFACTOREX, a refactoring tool for Elixir that addresses the lack of automated refactoring support in this language. It provides small-scale refactorings that integrate into daily workflows, enabling continuous code improvement and reducing the need for major restructuring. However, instead of focusing on implementation details specific to REFACTOREX, we extract general principles that can guide the development of other code transformation tools. By highlighting these transferable insights, this paper offers a practical reference that future tool developers can apply regardless of their target programming language.

The remainder of this paper is organized as follows. Section 2 reviews background and related work. Sections 3 to 5 detail architectural and implementation insights. Section 6 discusses adoption strategies and Section 7 outlines conclusions and future work.

2. Background and Related Work

Refactoring tools automate the process of code transformation, reducing both the effort required and the potential for introducing errors [Murphy-Hill and Black 2008]. Most of these tools follow a similar workflow structure: they parse the source code into an internal representation, analyze it to validate the refactoring's preconditions, transform the representation and then convert it back to source code [Li et al. 2003].

[1] https://elixir-lang.org/

Some refactoring tools have been created for functional languages, including REFACTORERL [Horpácsi 2013] and WRANGLER [Li et al. 2008] for Erlang, HARE [Li and Thompson 2006] for Haskell and ROTOR [Rowe et al. 2019] for OCaml. However, adapting their functionality to different languages presents unique challenges due to language-specific features and idiomatic styles.

More recently, [Vegi and Valente 2025] proposed a catalog of refactorings specifically for Elixir, which formed the basis for the REFACTOREX implementation. This catalog bridges the gap between refactoring theory and language-specific practices, being a valuable resource for the community. REFACTOREX can automatically apply 29 transformations, with 23 drawn from the aforementioned catalog. The full list of supported refactorings is available in the documentation on our GitHub repository (`https://github.com/gp-pereira/refactorex`).

3. Architectural Insights

This section explores the architectural decisions behind REFACTOREX, demonstrating how a thoughtful design can enhance extensibility and robustness in language tooling.

3.1. Editor Independence with LSP

A key insight from our experience was centering our tool's architecture around a standardized protocol like the Language Server Protocol (LSP).[2] The LSP separates the language support logic from editor-specific implementations, creating a design particularly well-suited for the scattered editors landscape. Its architecture establishes a clear division of responsibilities: the client (implemented by the IDE) handles UI interactions like code selection and option menus, while the language server exposes the actual functionality.

This separation also allows the server component to be developed in the target language itself, creating a more idiomatic implementation that leverages the language's own parsing capabilities. REFACTOREX demonstrates the benefits of this approach—initially implemented as a VS CODE plugin, it was later ported to NEOVIM by the Elixir community[3] with minimal changes, proving the architecture's versatility and adaptability.

3.2. Leveraging Open Source

Another significant advantage in our architectural approach was leveraging open source libraries as foundational building blocks. Two critical Elixir libraries enabled us to focus on refactoring-specific challenges rather than rebuilding infrastructure: **SOURCEROR**[4] provides bidirectional conversion between Elixir source code and Abstract Syntax Trees, along with powerful tree traversal and manipulation capabilities, essential for precise code transformations while preserving formatting; and **GENLSP**[5] abstracts the complexities of implementing an LSP server by handling connection management and message parsing, significantly reducing the effort required to build a standard-compliant language server.

Besides these libraries, we've also leveraged the Myers algorithm [Myers 1986] for identifying minimal changes between original and refactored code. Being natively

[2] `https://microsoft.github.io/language-server-protocol/`
[3] `https://github.com/synic/refactorex.nvim`
[4] `https://github.com/doorgan/sourceror`
[5] `https://github.com/elixir-tools/gen_lsp`

implemented in Elixir, this algorithm allows granular updates rather than full file replacements, enhancing both performance and user experience by minimizing network traffic and allowing editors to highlight only where the code changed.

This "standing on the shoulders of giants" approach reduced our development time, helped us maintain our focus on innovation by eliminating the need to revisit already solved problems and created opportunities for cross-pollination of ideas with other Elixir projects.

3.3. Resilience by Design

The inherent unpredictability of the code under analysis requires robust fault-tolerance mechanisms in refactoring tools. Even thoroughly tested operations may face edge cases in real-world scenarios due to unexpected structures, broken syntax or custom language extensions. To mitigate this, our architecture uses supervised process isolation, running each refactoring in its own process, isolating failures and preventing system-wide instability. This structure ensures that issues in one area do not disrupt the entire system, allowing individual operations to fail gracefully while maintaining overall responsiveness.

Instead of anticipating all edge cases, our architecture treats failures as expected occurrences and provides mechanisms for effective mitigation and recovery. Although REFACTOREX implements these features using Elixir's native capabilities [Jurić 2024], similar fault tolerance should be achievable with other technology stacks.

4. Implementation Strategies

REFACTOREX combines practical techniques to build an adaptable refactoring tool with minimal complexity, despite the challenges of source code transformation.

4.1. Example-Driven Development

Test-driven development [Beck and Andres 2004] with concrete examples proved especially effective for implementing our refactorings. Figure 1 illustrates a unit test for the *Extract Constant* refactoring. Similarly, each refactoring of REFACTOREX was validated through multiple test cases covering diverse code patterns. These tests specified both the input (with selection markers) and the expected output, ensuring accuracy and reliability. This pragmatic approach naturally led to a comprehensive test suite that besides verifying functionality and also documents all supported scenarios.

Another benefit of this way of building is its incrementalism. As the test suite expanded with new examples, the code evolved organically in response to actual requirements rather than anticipated ones. This discipline not only improved maintainability but also accelerated development, resulting in a more purpose-driven implementation.

4.2. Template Method for Refactorings

The design challenge of implementing multiple refactorings with shared behavior but distinct transformations was effectively addressed using a variation of the *Template Method* pattern [Gamma et al. 1994]. This pattern defines a common algorithm that handles parsing, selection localization, and the production of differential changes, while individual refactorings implement only their specific validation and transformation logic through a uniform interface. As a result, the codebase maintains a clear separation of concerns and each refactoring remains concise, averaging just 27 relevant lines, despite its complexity.

```elixir
test "extracts selection into a constant" do
  assert_refactored(
    ExtractConstant,
    """
    defmodule Shop do
    #                 v
    def discount, do: 0.25
    #                   ^
    end
    """,
    """
    defmodule Shop do
      @extracted_constant 0.25

      def discount, do: @extracted_constant
    end
    """
  )
end
```

Figure 1. Example of using TDD in RefactorEx

4.3. Composing Refactorings

Complex transformations were often built by composing simpler refactorings. For instance, *Inline Function* for a function call inside another function call required us to first *Extract Variable* for the inner call, inline the function statements and then *Inline Variable* for the return statement. This composition strategy simplified the implementation, ensured consistency and enabled new transformations to rely on established ones.

5. Underlying Machinery

Behind REFACTOREX's seemingly simple interface lies a complex machinery that transforms code with precision and reliability. The following subsections unveil the key technical foundations that power our refactoring engine.

5.1. Climbing the Abstract Syntax Tree

Relying on Abstract Syntax Trees (ASTs) rather than direct text manipulation was essential for achieving reliable refactorings in REFACTOREX. ASTs represent code as hierarchical structures that capture the relationships between elements, such as function headers and their scopes. They enable transformations that are infeasible with regex or text-based methods and support efficient tree traversal using algorithms like depth-first search.

Although Elixir natively exposes its AST—primarily for macro construction—building REFACTOREX on top of the SOURCEROR library significantly reduced implementation complexity (see Section 3.2). We also introduced higher-level abstractions to support refactorings, such as metadata-free node comparison, moving the iteration cursor to specific nodes and replacing multiple nodes at once. These abstractions enabled the creation of idiomatic AST operations, further simplifying the transformation process.

5.2. Where is the Selection?

One of the most significant challenges in developing REFACTOREX was reliably locating the user's selection within the AST, given its LSP range. After exploring several approaches, we arrived at the following solution: instead of traversing the complete AST searching for specific positions, we deleted everything but the selection from the source

code while maintaining its position. When parsed, this creates a subtree that can be readily identified within the complete AST. This technique elegantly circumvents positional mapping complexities while preserving all syntactic relationships within the selected code.

5.3. Variable Dependency Analysis

While the AST offers rich structural information, it lacks full semantic context, especially regarding variable dependencies across scopes. To address this, we implemented a recursive data flow analysis algorithm inspired by classic compiler techniques [Aho et al. 2006]. The algorithm tracks variable declarations and usages within each scope using a queue. Upon scope termination, it resolves local dependencies and passes unresolved references to the outer scope's queue. The resulting dependency graph has been essential for implementing refactorings such as *Underscore Unused Variables*, *Rename Variable* and *Extract Function*, making it crucial for any refactoring tool.

6. Fostering Adoption

REFACTOREX achieved notable adoption within the Elixir community, registering over 1,100 downloads in its first month. This early traction underscores key factors contributing to its impact: the tool addressed a well-recognized gap in the Elixir ecosystem by providing much-needed automated refactoring support; it featured comprehensive documentation with visual examples and streamlined navigation, enhancing accessibility and usability; and its open-source development model fostered community engagement, enabling contributions and facilitating integration with additional editors.

These strategies formed the foundation of our adoption approach, but were significantly amplified through some outreach efforts. A pivotal moment came when José Valim, the creator of Elixir, shared the project on Twitter/X, bringing immediate visibility to thousands of developers. Further recognition came during a talk at Code BEAM America 2025,[6] where REFACTOREX was highlighted as an important contribution to the ecosystem. These endorsements from respected community figures established credibility and trust, helping drive REFACTOREX's rapid adoption within the Elixir community.

7. Conclusion and Future Work

This paper presents the development process of the Elixir refactoring tool REFACTOREX and identifies a set of engineering practices adopted throughout its construction—including architectural planning, the application of design patterns, systematic testing strategies and attention to user experience—that may serve as valuable guidelines for developers aiming to build effective refactoring tools for other functional languages.

As future work, we plan to implement cross-file refactorings and code smell [Vegi and Valente 2023] detection capabilities into the tool. Additionally, we aim to explore property-based testing approaches for refactoring validation and integrate REFACTOREX with the official Elixir language server, which is currently under development.

Acknowledgments

The tool `https://claude.ai/` was used during the initial phase of writing to assist in transforming the first author's undergraduate thesis into a six-page paper. The entire text was subsequently reviewed and substantially modified by all authors.

[6] `https://www.youtube.com/watch?v=R1CCyszhESk`

References

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2 edition.

Beck, K. and Andres, C. (2004). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2 edition.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley, 2 edition.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Pearson Education, 1 edition.

Horpácsi, D. (2013). Extending Erlang by utilising RefactorErl. In *12h ACM SIGPLAN Workshop on Erlang*, page 63–72.

JetBrains (2023). IntelliJ IDEA: the leading Java and Kotlin IDE. Available at: `https://www.jetbrains.com/idea/`.

Jurić, S. (2024). *Elixir in action*. Manning, 3 edition.

Li, H., Reinke, C., and Thompson, S. (2003). Tool support for refactoring functional programs. In *2003 ACM SIGPLAN Workshop on Haskell*, page 27–38.

Li, H. and Thompson, S. (2006). Comparative study of refactoring Haskell and Erlang programs. In *6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 197–206.

Li, H., Thompson, S., Orosz, G., and Tóth, M. (2008). Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse. In *7th ACM SIGPLAN Workshop on ERLANG*, pages 61–72.

Murphy-Hill, E. and Black, A. P. (2008). Breaking the barriers to successful refactoring: observations and tools for extract method. In *30th International Conference on Software Engineering (ICSE)*, page 421–430.

Myers, E. W. (1986). An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1):251–266.

Rowe, R. N. S., Férée, H., Thompson, S., and Owens, S. (2019). Characterising renaming within OCaml's module system: Theory and implementation. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 950–965.

Rudi, A. G. (2006). Rope: Python refactoring library. Available at: `https://github.com/python-rope/rope`.

Vegi, L. F. M. and Valente, M. T. (2023). Understanding code smells in Elixir functional language. *Empirical Software Engineering*, 28(102):1–32.

Vegi, L. F. M. and Valente, M. T. (2025). Understanding refactorings in Elixir functional language. *Empirical Software Engineering*, 30(108):1–58.