Untangling the Knot: Recommending Component Refactorings

James Ivers
Copyright 2019 Carnegie Mellon University.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.


DM19-1114
Software Structure Enables Our Ability to Innovate

Quickly delivering new capabilities and taking advantage of new technology depend on an ability to evolve software efficiently. The structure of legacy software, however, often fails to support this goal.

A recent anecdote from a DoD contractor: The estimate for isolating a mission capability from the underlying hardware platform was 14,000 staff hours (development only).

This is representative of a class of changes that involve isolating a specific software capability from its context. Other examples include

- migrating a capability to the cloud
- harvesting a component for reuse
- replacing a proprietary component

Our project will allow the same work to be done in one-third of the time.
Even modest systems are hard to comprehend, and harder to modify.

- A modest application with only 68K lines of code (LOC) contains more than 10K nodes and 50K relations.
- Making a "simple" change, like isolating the code for deployment as a service, can require reasoning about hundreds of dependencies.

A 2018 survey found that more than 40% of an average developer work week was spent on "maintenance (i.e., dealing with bad code/errors, debugging, refactoring, modifying)."

https://stripe.com/reports/developer-coefficient-2018
SEI Goal: Create an Automated Refactoring Assistant

Refactoring is a technique for improving the structure of software, but it is typically a labor-intensive process in which developers must:

- figure out where changes are needed
- figure out which refactoring(s) to use
- implement refactorings by rewriting code

Our goal is to create an automated assistant for developers that recommends refactorings to isolate software, allowing capabilities to be harvested or replaced in 1/3 of the time it takes to do so manually.

- Uses a semi-automated approach
- Addresses all three labor-intensive activities

In perspective, our work would reduce the cost in the earlier example from 14,000 staff hours to 4,500 staff hours—saving the cost of 9,500 hours of development.
Building on Search-Based Software Engineering

By framing software engineering problems as optimization problems, we can use metaheuristic search techniques to automatically find solutions.

• Encouraging work in refactoring focuses on improving general quality metrics\(^1,2,3\)
• Limited but growing interaction with users to capture preferences

Our innovation

• Focus on isolating software
• Start with user preference
• Define criteria to guide search to practical solutions

---

Our Approach

We are adapting search-based optimization algorithms to recommend refactorings that isolate software to support harvesting or replacing capabilities.
Problem Framing

Basis: Only certain software dependencies interfere with the goal.

Approach: Focus search on solutions that reduce those dependencies.

• Counting those dependencies is an **objective** basis for fitness.
• Reducing scope of search (by 1 to 4 orders of magnitude) promotes **scalability**.
Problematic Couplings

Problematic couplings are those software dependencies that interfere with achieving a specific goal.

Our prototype automatically identifies these, and we are using this data to drive the research.

Application: Sizing the work to isolate software for a range of scenarios
- Prioritizing software for migration
- Providing input to cost analysis
We use a static code analysis tool to extract structural information from C# source code.

Sample graph sizes

- **Duplicati:**
  - 68K code lines
  - 10,194 nodes and 49,620 relations

- **MissionPlanner:**
  - 756K code lines
  - 81,790 nodes and 587,542 relations
Formalized Refactorings

Refactorings are the operations that the search algorithms use to explore changes to the graph.

Refactorings involve changes like
- moving, copying, or removing code
- extracting portions of code
- introducing interfaces or intermediaries

We formalize each in terms of a precondition and transformation over the graph.

FY19 – initial set; FY20 – scale up

Of 19,720 problematic couplings in our open source case studies,
- 74.1% can be resolved by at least one refactoring
- 14.0% can be resolved by more than one refactoring
Multi-objective Search and Fitness Functions

Multi-objective genetic algorithms like NSGA-II allow us to employ multiple fitness functions and generate Pareto-optimal solutions.

We are exploring fitness functions to find a combination that yields recommendations that developers will accept.

Candidates include

• solving the core problem – minimizing problematic couplings
• reducing work – minimizing code changes and unrealized interfaces
• maintainable code – improving a range of code quality metrics
• understandable code – maximizing semantic coherence
Automated search finds sequences of refactorings that collectively solve as much of the project-specific goal as possible.

FY19 – local search; FY20 – global search (genetic algorithms)
Looking Ahead

**FY19**
- Build out infrastructure: representation, refactorings, fitness functions, and local search
- Assemble open source data and initial analyses
- **Ready to pilot** the ability to size problems for C# software

**FY20**
- Broaden the palette: more refactorings and fitness functions
- Implement global search using multi-objective genetic algorithms

**FY21**
- Fine-tune search
- Validate with experienced developers
- **Ready to pilot** generation of refactorings for C# software
Software Is Constantly Changing over Its Lifetime
What We Want Software to Do Also Changes
Over Time, Gaps Emerge and Grow

When software structure differs significantly from what is needed, the pace of change and innovation slows down.
Vision: AI for Software Engineering
Automation Can Bring Projects Back into Alignment
Vision: AI for Software Engineering
Automation Can Keep Software Aligned with Needs