Untangling the Knot: Enabling Rapid Software Evolution

James Ivers and Ipek Ozkaya

September 10, 2020

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

DM20-0761

CMU SEI is a DoD Federally Funded Research and Development Center

Our mission: Engineering and securing software

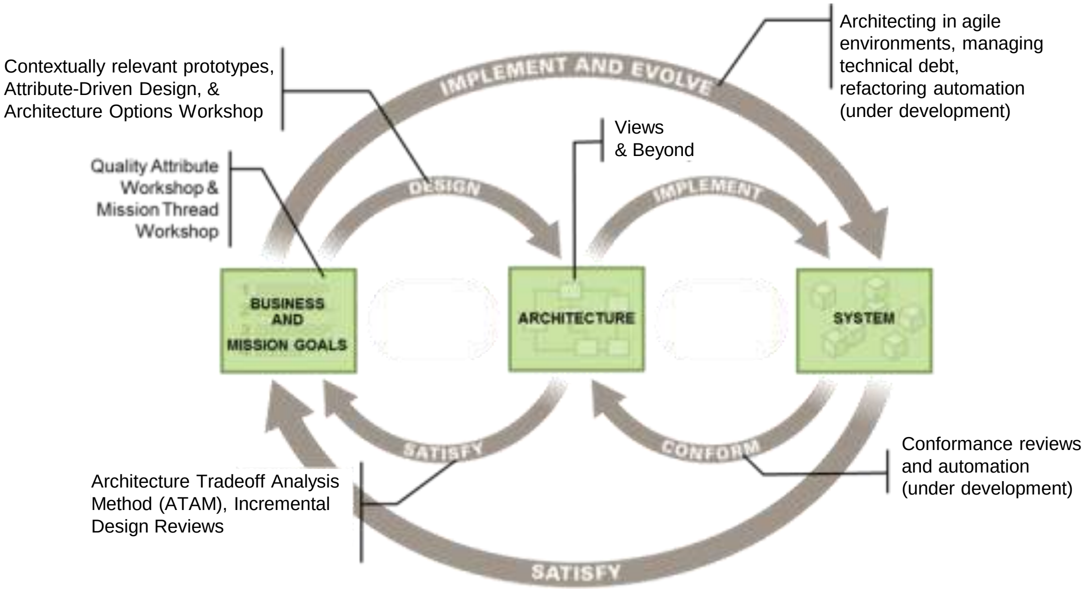
Established in 1984 at Carnegie Mellon University

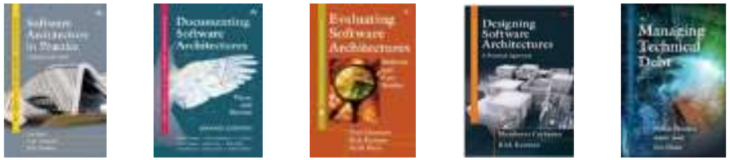
~635 employees

Offices in Pittsburgh and DC, with locations near customer facilities in MA, MD, TX, and CA

~$145M in annual funding (~$20M USD(R&E) 6.2 and 6.3 Line funding)

Establishing a Discipline of Software Architecture





Range of methods and practices applicable at different points in the development lifecycle

- Domains of expertise include IT, C2, tactical, and health informatics
- Technology expertise includes IoT, big data, digital twin, cloud, and machine learning

10+ courses, available in a mix of public, on-site, and eLearning options

3 professional certificates

A collection of books for wide dissemination

Software Is Never Done

Software is an essential building material for any system today.

The ability to efficiently evolve software depends on its architecture (structure)

- Architectures that are well aligned with needs allow faster changes with greater confidence
- Misaligned architectures slow change, if not outright preventing it

Software Is Never Done

Change is inevitable

- Requirements change
- Business priorities change
- Programming languages change
- Deployment environments change
- Technologies and platforms change
- Interacting systems change
- ...

To adapt to such changes, we need to periodically improve software structure to match today's needs.

An Automated Refactoring Assistant

We have developed an automated refactoring assistant for developers that improves software structure for several common forms of change that involve feature isolation:

- Solves project-specific problems
- Uses a semi-automated approach
- Addresses all three labor-intensive activities
- Allows refactoring to be completed in less than 1/3 of the time required by manual approaches

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

Category 1: Input to Funding Decisions

Most organizations have an appetite for more software changes than their budgets can support. Data-driven decision making benefits from richer forms of data.

Sample scenarios:

- A program office wants to rehost a capability on a new platform. A contractor estimates that isolating the capability from the original platform will require 14,000 staff hours (development only). Is this reasonable?
- An organization is prioritizing funding requests as part of an annual portfolio analysis. Multiple projects require refactoring to achieve their goals. Has each reasonably accounted for the effort required to refactor?

Our refactoring assistant gathers important data useful as inputs to a cost estimate:

- specific to the project goal
- localized to the entities affected by the proposed change
- traceable to specific lines of code

Category 2: Comparing Refactoring Options

As part of many rearchitecting or modernization activities, development teams have options on how to restructure software. However, they often lack good tools for determining the relative merits of different proposed solutions.

Sample scenario:

- An organization is breaking a monolithic application into independent services or microservices. There are multiple ways to split the software into pieces. What is the relative difficulty of each proposed restructuring?

Our refactoring assistant can be used to compare the difficulty of alternate scenarios:

- sketch multiple ways of partitioning the monolith
- use the tool's initial analysis to estimate difficulty for each option
- use the tool's refactoring recommendation features to assess ripple effects and down stream challenges

Category 3: Automating Refactoring

Restructuring software is often a necessary first step to take advantage of new technologies (e.g., cloud or DevOps) or add new capabilities.

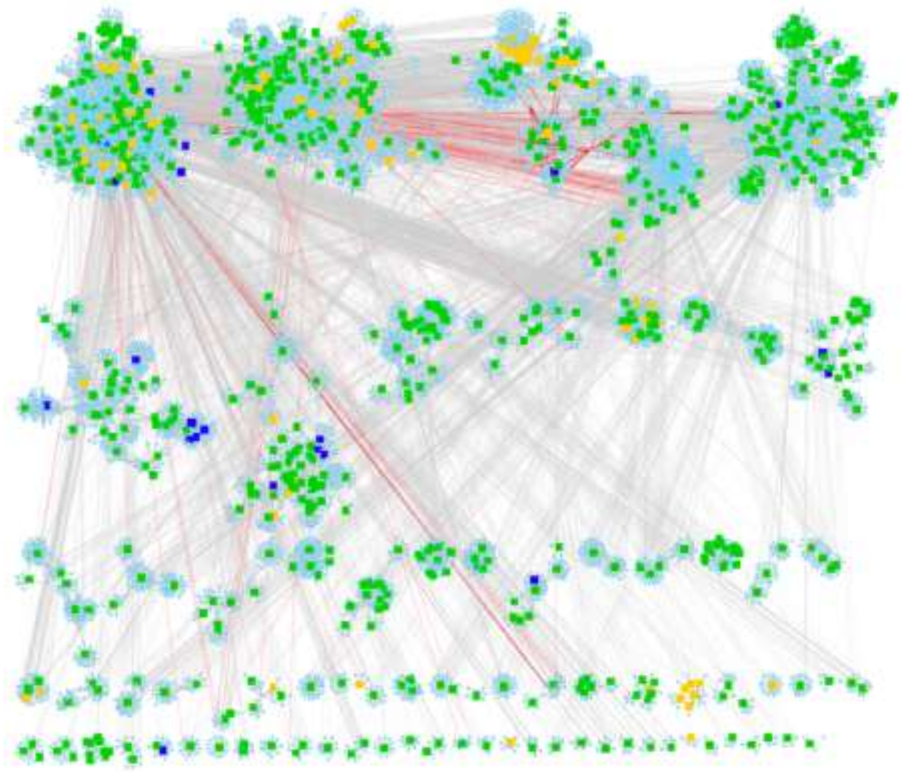
What software changes are needed to enable each of the following scenarios?

- An organization wants to deploy existing functionality in the cloud to improve scalability and reduce costs.
- An organization wants to take advantage of DevOps by containerizing capabilities as smaller, independently deployable units.
- An organization wants to reuse a successful capability across multiple systems.

Our refactoring assistant automatically recommends refactorings that speed evolution (also reducing cost):

- recommendations isolate specified software from its original context
- minimal configuration required to generate recommendations
- engineers can review all changes before application
- implementing recommendations is straightforward

Key Concept - Problematic Couplings



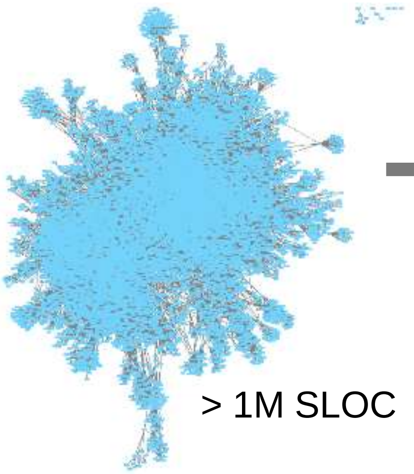
Only certain software dependencies interfere with any particular goal.

For example, if we want to harvest a feature:

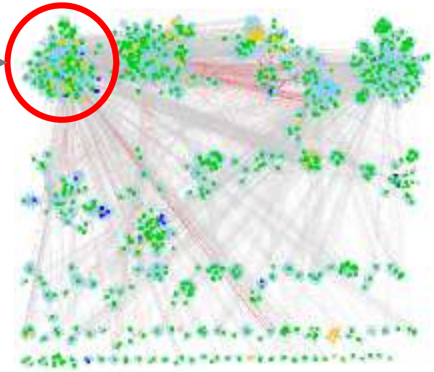
- The core problem is dependencies (red lines) from software being harvested to software that is being left behind
- All other dependencies are irrelevant to the goal, allowing us to focus our analysis and search for solutions

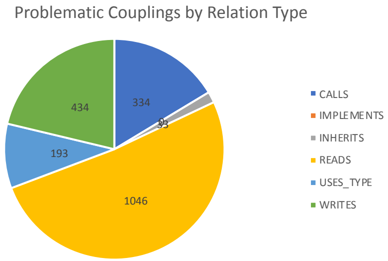
This insight enables us to apply **search-based software engineering** techniques and treat this as an **optimization problem**.

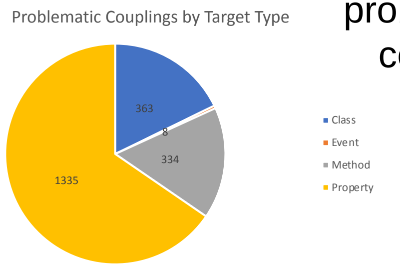
Problem Analysis



> 1M SLOC





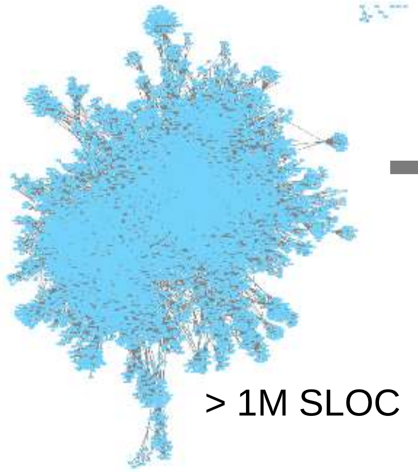


2040 problematic couplings

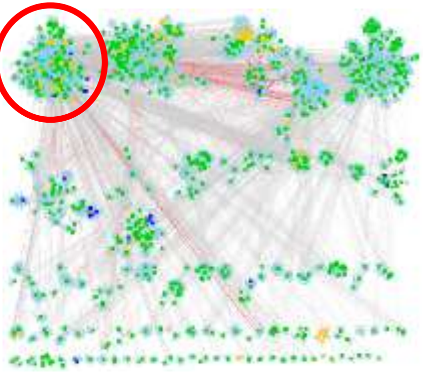
Target Type	PC Count	# Unique Targets
Class	363	15
Event	8	1
Method	334	51
Property	1335	77
	2040	**144**

24 unique classes

Generating Refactoring Recommendations



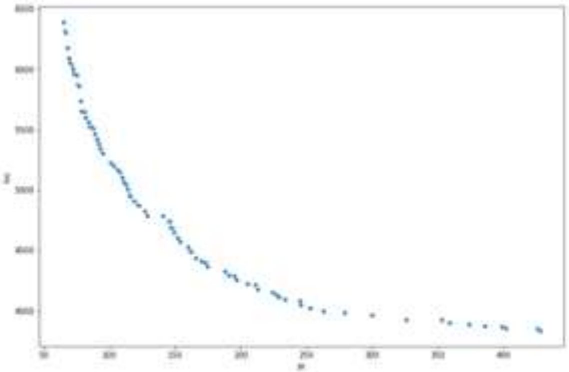
> 1M SLOC



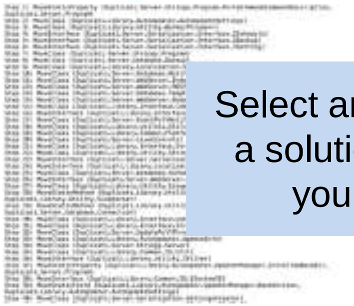
Select Objectives

- minimize problematic couplings
- minimize code changes
- maximize code quality
- ...

Use a multi-objective genetic algorithm to generate a set of Pareto optimal solutions (recommendations)

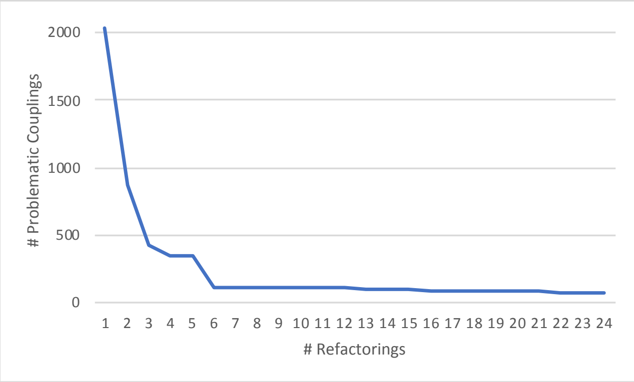


Select and implement a solution that suits your context



Refactoring Recommendations

```
Best solution:
Fitness = 33
Step 1: MoveStaticProperty (Duplicati.Server.Strings.Program.PortablemodeCommandDescription,
Duplicati.Server.Program)
Step 2: MoveClass (Duplicati.Library.AutoUpdater.AutoUpdateSettings)
Step 3: MoveClass (Duplicati.Library.Utility.WorkerThread<>)
Step 4: MoveInterface (Duplicati.Server.Serialization.Interface.ISchedule)
Step 5: MoveInterface (Duplicati.Server.Serialization.Interface.IBackup)
Step 6: MoveInterface (Duplicati.Server.Serialization.Interface.ISetting)
Step 7: MoveClass (Duplicati.Server.Strings.Program)
Step 8: MoveClass (Duplicati.Server.Database.Backup)
Step 9: MoveClass (Duplicati.Library.Localization.Short.LC)
Step 10: MoveClass (Duplicati.Server.Database.Notification)
Step 11: MoveClass (Duplicati.Server.WebServer.IndexHtmlHandler)
Step 12: MoveClass (Duplicati.Server.WebServer.RESTMethods.RequestInfo)
Step 13: MoveClass (Duplicati.Server.Database.TempFile)
Step 14: MoveClass (Duplicati.Server.WebServer.BodyWriter)
Step 15: MoveClass (Duplicati.Library.Interface.CommandLineArgument)
Step 16: MoveInterface (Duplicati.Library.Interface.ICommandLineArgument)
Step 17: MoveClass (Duplicati.Server.EventPollNotify)
Step 18: MoveClass (Duplicati.Library.Utility.Utility)
Step 19: MoveClass (Duplicati.Library.Common.Platform)
Step 20: MoveClass (Duplicati.Server.LiveControls)
Step 21: MoveClass (Duplicati.Library.Interface.Strings.DataTypes)
Step 22: MoveClass (Duplicati.Library.Utility.Strings.Utility)
Step 23: MoveInterface (Duplicati.Server.Serialization.Interface.IFilter)
Step 24: MoveInterface (Duplicati.Library.Localization.ILocalizationService)
Step 25: MoveClass (Duplicati.Server.Database.Schedule)
Step 26: MoveInterface (Duplicati.Server.WebServer.RESTMethods.IRESTMethodPOST)
Step 27: MoveClass (Duplicati.Library.Utility.Sizeparser)
Step 28: MoveStaticMethod (Duplicati.Library.Utility.Strings.Sizeparser.InvalidSizeValueError,
Duplicati.Library.Utility.Sizeparser)
Step 29: MoveStaticMethod (Duplicati.Library.Utility.Timeparser.ParseTimeSpan,
Duplicati.Server.Database.Connection)
Step 30: MoveClass (Duplicati.Library.Interface.UserInformationException)
Step 31: MoveClass (Duplicati.Library.Interface.Strings.CommandLineArgument)
Step 32: MoveClass (Duplicati.Server.UpdatePollThread)
Step 33: MoveClass (Duplicati.Library.AutoUpdater.UpdateInfo)
Step 34: MoveClass (Duplicati.Server.Strings.Server)
Step 35: MoveClass (Duplicati.Library.Common.IO.Util)
Step 36: MoveInterface (Duplicati.Library.Utility.IFilter)
Step 37: MoveStaticProperty (Duplicati.Library.AutoUpdater.UpdaterManager.InstalledBaseDir,
Duplicati.Server.Program)
Step 38: MoveInterface (Duplicati.Library.Common.IO.ISystemIO)
Step 39: MoveStaticField (Duplicati.Library.AutoUpdater.UpdaterManager.BaseVersion,
Duplicati.Library.AutoUpdater.AutoUpdateSettings)
Step 40: MoveClass (Duplicati.Server.Serialization.SettingsCreator)
```



Our prototype generates recommendations as a sequence of refactorings:

- clear directions for a developer
- independently reviewable prior to changing code
- built on refactorings supported by development environments
- future potential to automate application to code

Satisfying Multiple Criteria

We use a combination of fitness functions to generate *recommendations that developers will accept*.

Examples include

- solution to the core problem – minimizing problematic couplings
- less work – minimizing code changes and unrealized interfaces
- maintainable code – improving code quality metrics
- understandable code – maximizing semantic coherence
- secure code – minimizing public members

Our prototype uses a multi-objective genetic algorithm, based on NSGA-II, to generate Pareto optimal solutions that represent different trade-offs among objectives.

Summary

We can apply our prototype to the following scenarios:

Scenario	Maturity	Expected Results
Input to Funding Decisions	*Available now* (TRL 4)	Enumeration of problematic couplings, their locations, and types potentially impacted by proposed change as data to inform cost estimates
Comparing Refactoring Options	*Available now* (TRL 4)	Enumeration of problematic couplings, their locations, and types potentially impacted by proposed change as data to inform cost estimates
Automating Refactoring	Ready for pilot application in 3–6 months	Recommended refactorings that • enable the proposed change • address multiple criteria

All scenarios require

- source code
- proposed changes

Programming languages

- C# is ready now (tested up to 1.2M SLOC)
- Java support could be ready in 2–3 months for first two scenarios

The prototype does build on a single commercial tool.

For More Information

James Ivers

Initiative Lead
Architecture Analysis, Design, and Automation
jivers@sei.cmu.edu
412-268-7793

Ipek Ozkaya

Technical Director
Engineering Intelligent Software Systems
ozkaya@sei.cmu.edu
412-268-3551