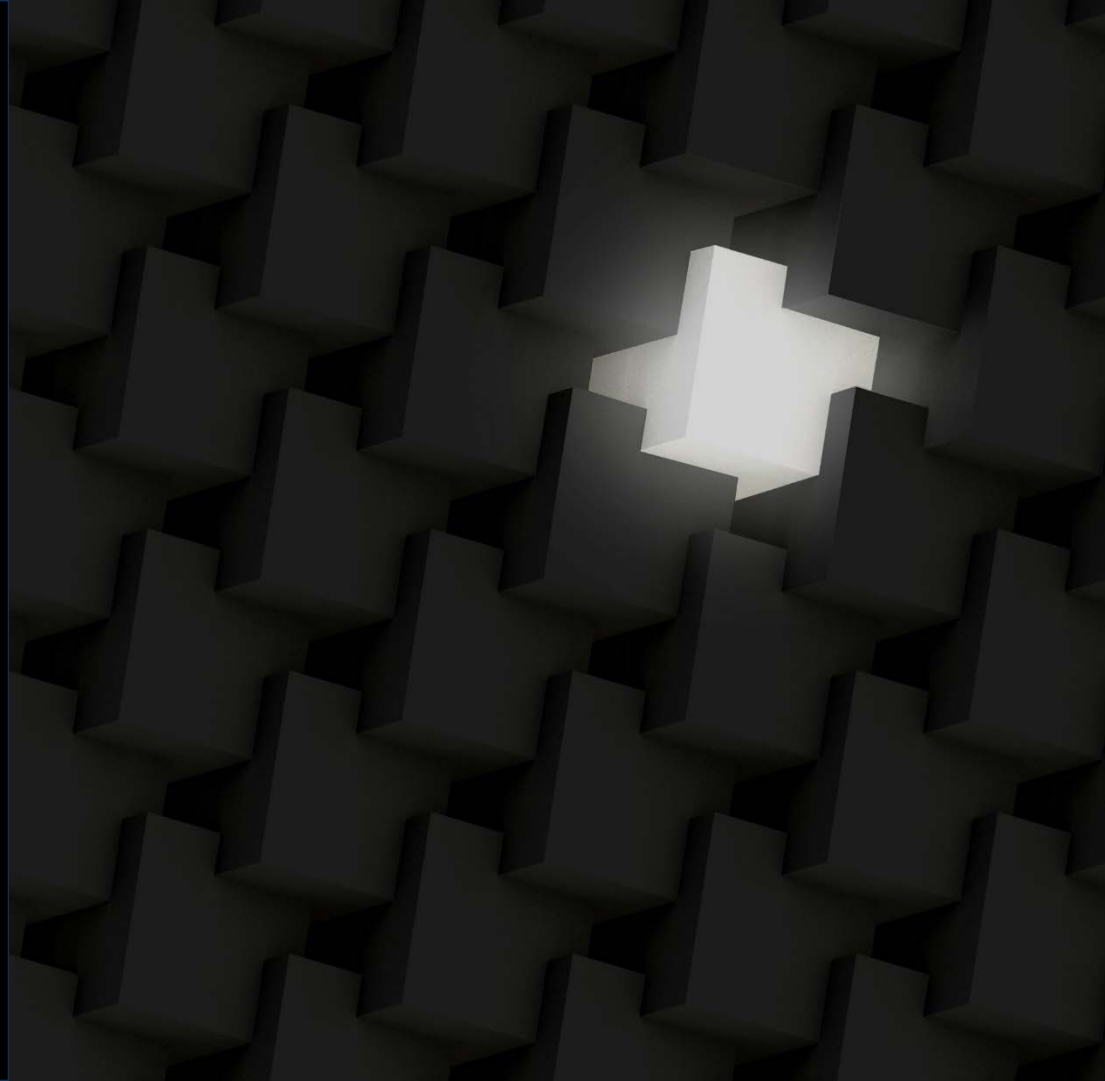
Carnegie Mellon University
Software Engineering Institute

RESEARCH REVIEW 2020

Untangling the Knot: Enabling Rapid Software Evolution

James Ivers



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.

DM20-0856

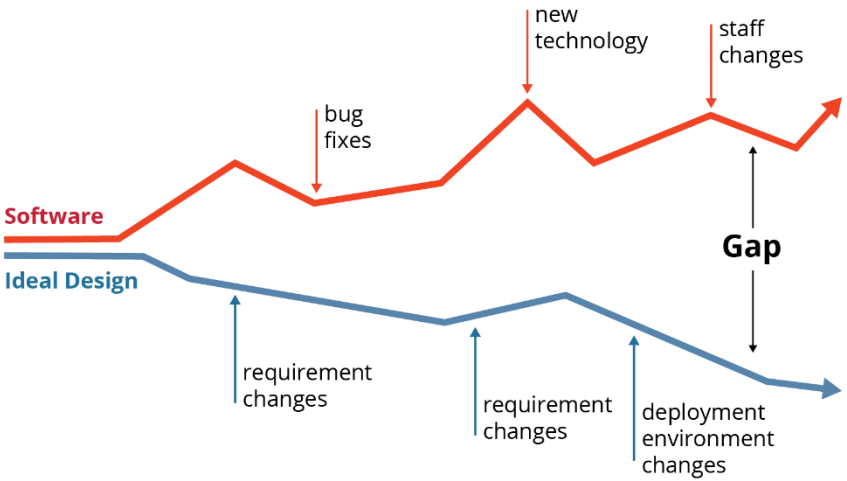
Software Is an Essential Building Material

Our ability to work with software significantly influences project cost, schedule, time to field, and other concerns.

The ability to efficiently build, change, and evolve software depends on its architecture and how that architecture is realized in code.

Architectures that are well aligned with needs allow faster changes with greater confidence.

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 (architecture) 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

J. Ivers, I. Ozkaya, R. L. Nord. **Can AI Close the Design-Code Abstraction Gap?** Software Engineering Intelligence Workshop 2019, co-located with Intl. Conference on Automated Software Engineering: 122-125.

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:

- Assess contractor cost estimates for architecture improvements
- Support portfolio analysis and prioritization activities

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:

- Compare different options for breaking a monolithic application into independent services or microservices

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 downstream challenges

Category 3: Automating Refactoring

Restructuring software is often a necessary first step to take advantage of new technologies or add new capabilities.

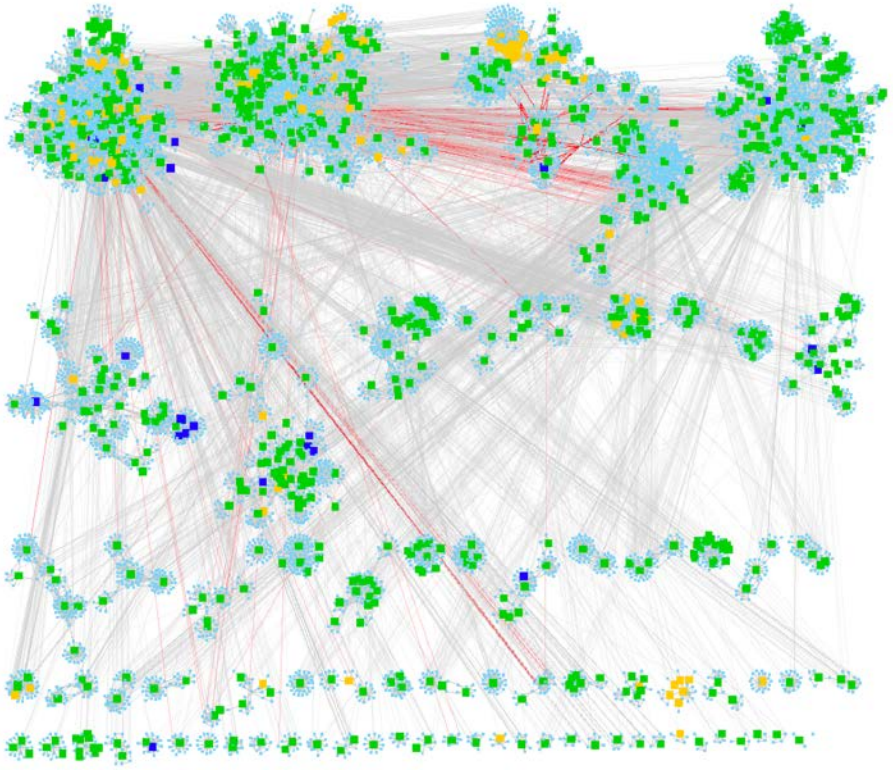
Sample scenarios:

- Migrate capabilities to the cloud
- Containerize capabilities to embrace DevOps
- Reuse a capability across multiple systems
- Replace an outdated component with a new alternative

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

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

Key Concept – Problematic Couplings



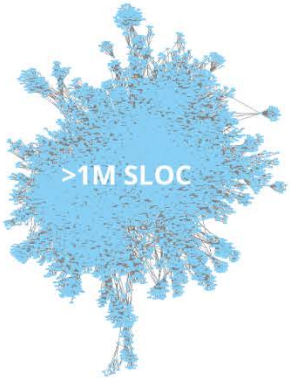
Only certain software dependencies interfere with any particular goal.

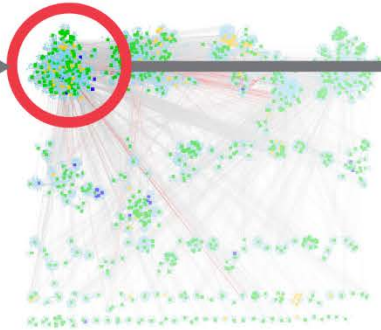
For example, if we want to harvest a feature:

- 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

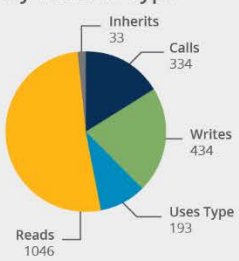
This insight enables us to apply **search-based software engineering** techniques and treat this as an **optimization problem**.

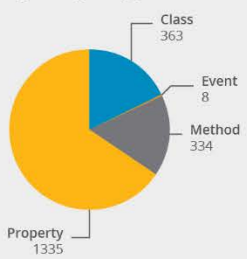
Problem Analysis





2040 Problematic Couplings

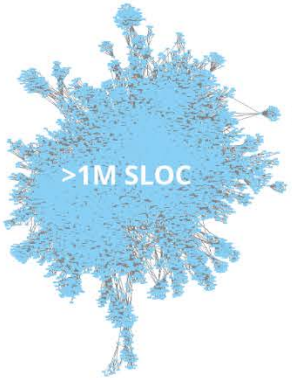


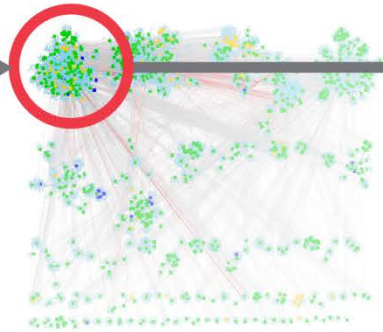


Target Type	PC Count	# Unique Targets
Class	363	15
Event	8	1
Method	334	51
Property	1335	77
	2040	**144**

Out of 1M+ SLOC, changes should focus on **only 24 classes**

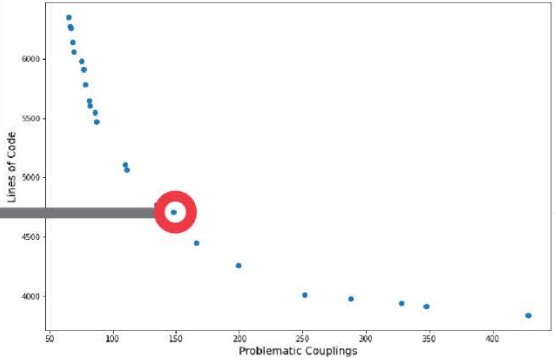
Generating Refactoring Recommendations





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

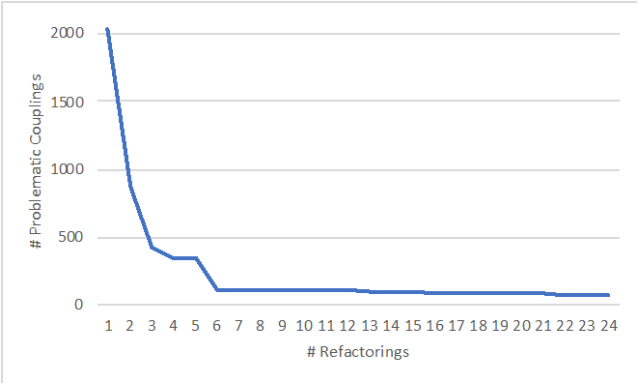
Our prototype uses a multi-objective genetic algorithm to generate a set of Pareto optimal solutions (recommendations)



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

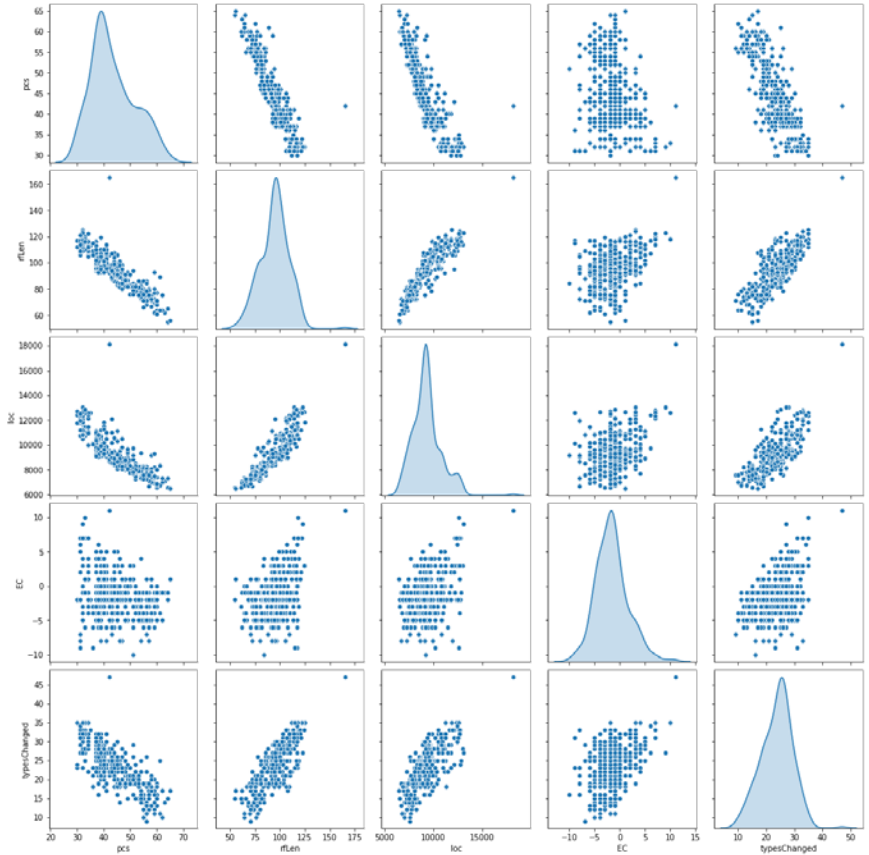
We use a combination of fitness functions to generate *recommendations that developers will accept*.

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.

Identifying Good Fitness Functions

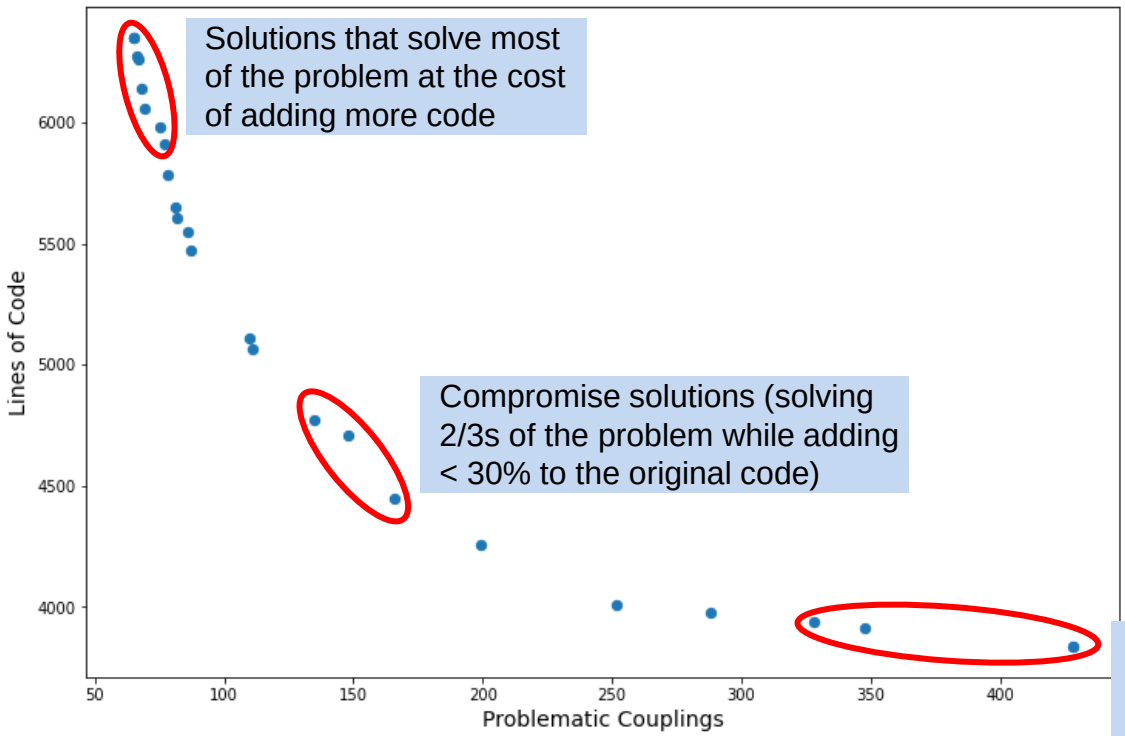


Correlation matrix
(method: Spearman)

	pcs	rfLen	loc	EC	typesChanged
pcs	1.00	-0.88	-0.85	-0.19	-0.76
rfLen	-0.88	1.00	0.89	0.28	0.75
loc	-0.85	0.89	1.00	0.27	0.69
EC	-0.19	0.28	0.27	1.00	0.37
typesChanged	-0.76	0.75	0.69	0.37	1.00

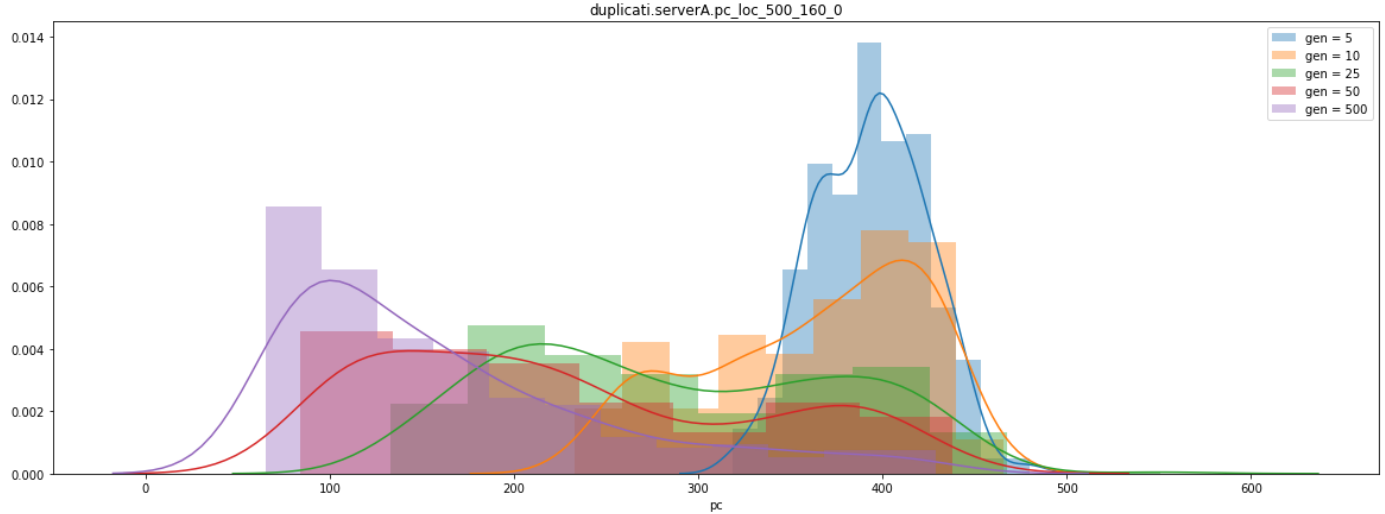
Good fitness functions measure distinct underlying phenomena.

Pareto Optimal Solutions



- Trade-offs among objectives are normal
- Pareto optimality allows us to generate a set of good solutions that represent different trade-offs
- Developers can choose from these options to suit their context

Tuning Algorithm Metaheuristics



Actively experimenting with the factors that influence search effectiveness:

- number of generations
- population size
- crossover rate
- mutation operation
- selection functions
- archive mechanics
- available refactorings
- fitness functions

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

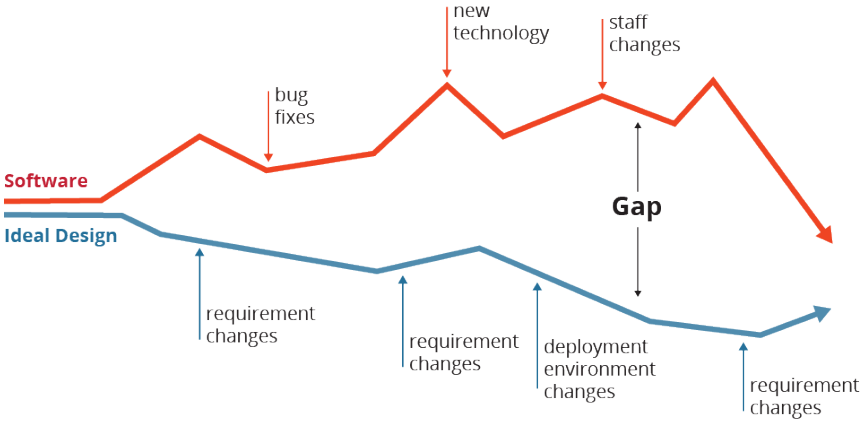
All scenarios require

- source code
- proposed isolation goal

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

Looking Ahead: Next-Generation Automation for Software Evolution



The SEI's work and vision is to

- develop automation that DoD organizations can trust to provide accurate information on the size, consequences, and resources needed for software changes
- advance the state of the art to allow developers to sketch proposed changes in the language of design and to trust that automation can realize those changes

J. Ivers, I. Ozkaya, R. L. Nord, C. Seifried. **Next Generation Automated Software Evolution: Refactoring at Scale**. 2020. *28th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20). ACM, Virtual Event, USA.*

Contact us at **info@sei.cmu.edu** if you are interested in partnering with us.