Scaling Refactoring

James Ivers
jivers@sei.cmu.edu

October 14, 2022

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

About Me

25+ years experience as a software architect at the intersection of industry, government, and academia.

Since 2010, I've been leading the software architecture team at the Carnegie Mellon Software Engineering Institute (SEI)

- Works with organizations with large architecture challenges
  - Analysis, evolution, modernization, etc.
- Research in technical debt and refactoring

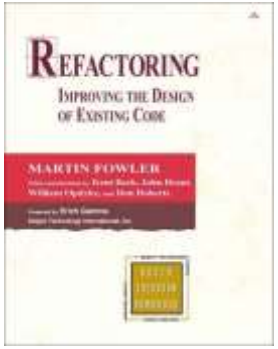
My industry experience includes

- Building commercial code analysis tools for a large organization
- Just about every role you can think of at a tiny mobile computing startup

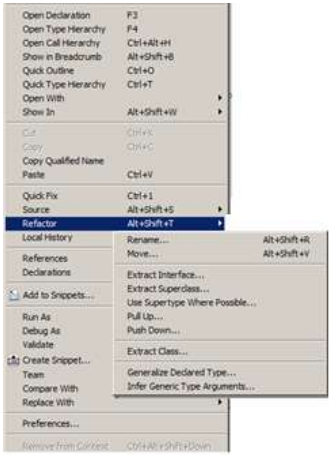
Refactoring is a Familiar Concept in Industry

Martin Fowler helped popularize refactoring in the 90s as

> "... the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."



Modern IDEs commonly provide support for some types of refactorings, though adoption by developers varies significantly.



Refactoring Tactics

Floss Refactoring

- Intended to maintain healthy code
- Frequent, short bursts intermingled with other code changes
- Can by studied by monitoring IDE use or mining repositories for commits containing known refactorings

Root-Canal Refactoring

- Intended to correct unhealthy code
- Infrequent, protracted efforts that perform few other code changes
- Harder to study in practice

Emerson Murphy-Hill and Andrew P. Black. **Refactoring Tools: Fitness for Purpose**. *IEEE Software* 25, 5 (2008), 38–44.

Scaling Refactoring

My focus is on scaling refactoring to make substantial changes to a system that typically cannot be accomplished (in reasonable time) by one or even a few developers.

Examples include

- Moving from a monolithic to a more modular architecture (e.g., micro-services)
- Re-organizing software to a more configurable or reusable form (e.g., product lines)
- Moving existing functionality from an unsupported platform (e.g., mainframes) or framework
- Removing pervasive technical debt accrued when prematurely moving a prototype to production (e.g., decoupling elements)

How well do these fit our understanding of refactoring?

"... the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."

A Personal Example: Improving Portability of a Product

Starting Point: Commercial organization has a successful product (a profiler) that runs on a single OS. Extension to a second OS would increase sales.

Challenges:

- Project is written in multiple languages and assembly doesn't port well.
- Some libraries the product relies on are dated.
- By the way, adopt a new UI framework to unify on corporate look and feel.

Elements of the solution:

- Rewrite assembly code in C.
- Optimize the new C libraries to avoid performance degradation.
- Replace all references to outdated libraries with references to newer versions of the same libraries.
- Adopting new UI framework (C++) lead to restructuring significant portions of C code to C++ classes.
- Re-test everything, incrementally and often.
- Estimated effort:

  4-5 developers x 4-5 months.

Scaling Refactoring

Insights from Industry

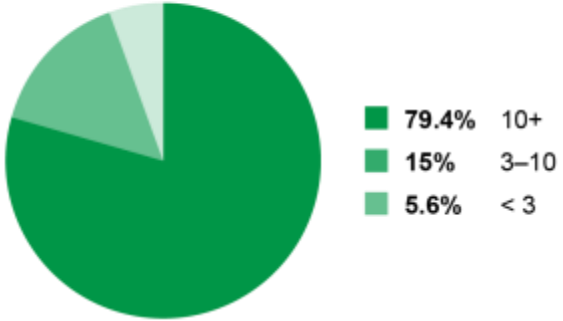
A Survey on Large-Scale Refactoring (LSR)

We surveyed practitioners to understand how large-scale refactoring is performed today

- How common is it?
- What motivates it?
- What makes it hard?
- What tools are used? Needed?

107 responses, 96% of whom worked as software engineers or architects.

How many years of experience did respondents have?



J. Ivers, R. Nord, I. Ozkaya, C. Seifried, C. Timperley, M. Kessentini. **Industry Experiences with Large-Scale Refactoring.** *Foundations of Software Engineering: Software Engineering in Practice* (ESEC/FSE). November 2022.

J. Ivers, R. Nord, I. Ozkaya, C. Seifried, C. Timperley, M. Kessentini. **Industry's Cry for Tools That Support Large-Scale Refactoring.** *Intl. Conference on Software Engineering: Software Engineering in Practice* (ICSE-SEIP). May 2022.

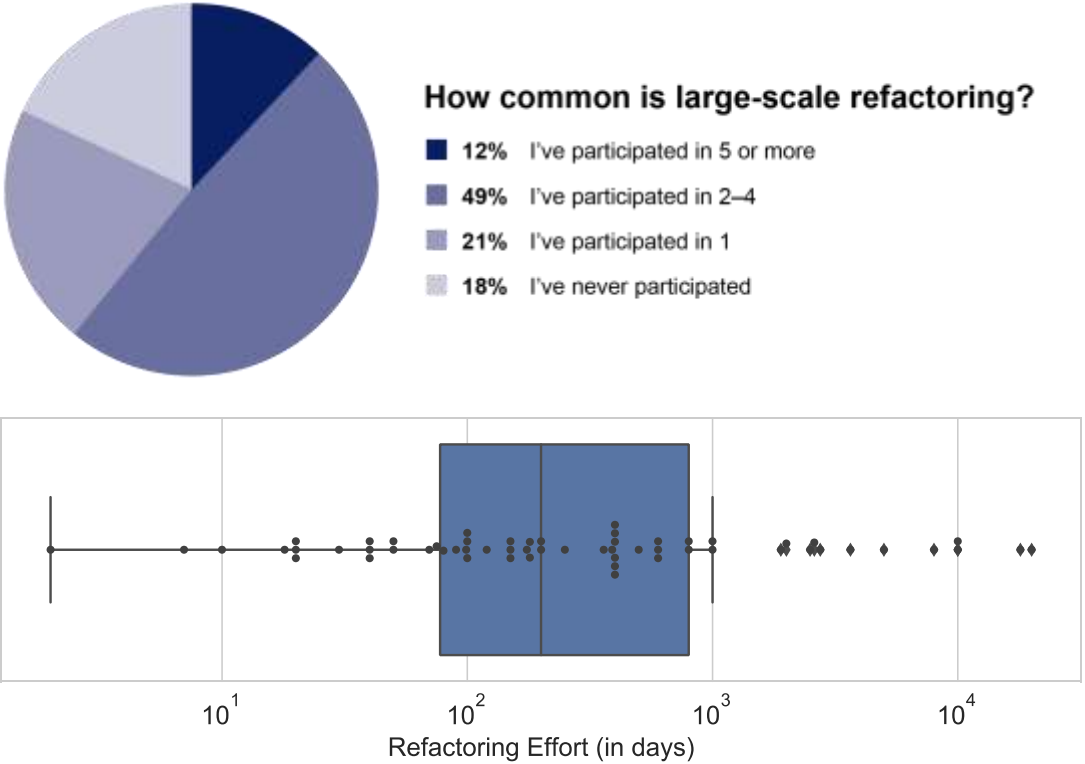
Survey Design

Survey questions were grounded in the experience(s) of each participant.

Topics addressed include

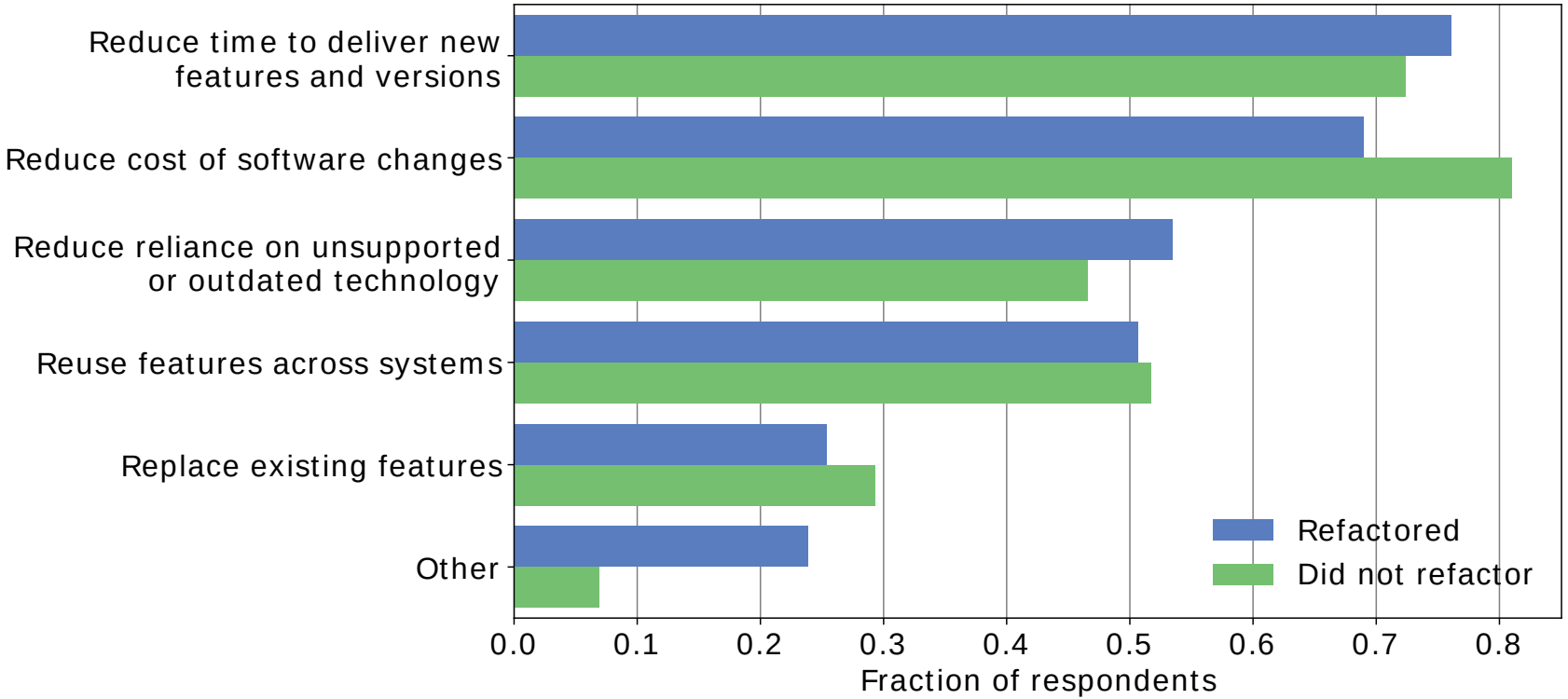
- Instances in which respondents had ***performed large-scale refactoring***
  - How often and how much effort?
  - What tools were used?
  - How did time spent, challenges encountered, and tool use vary with activities?
- Instances in which respondents wanted to but ***did not perform large-scale refactoring***
  - How often and why not?
  - What consequences were observed after foregoing refactoring?
- Respondents thoughts on kinds of ***tools that would improve large-scale refactoring***

LSR Consumes Significant Industry Resources

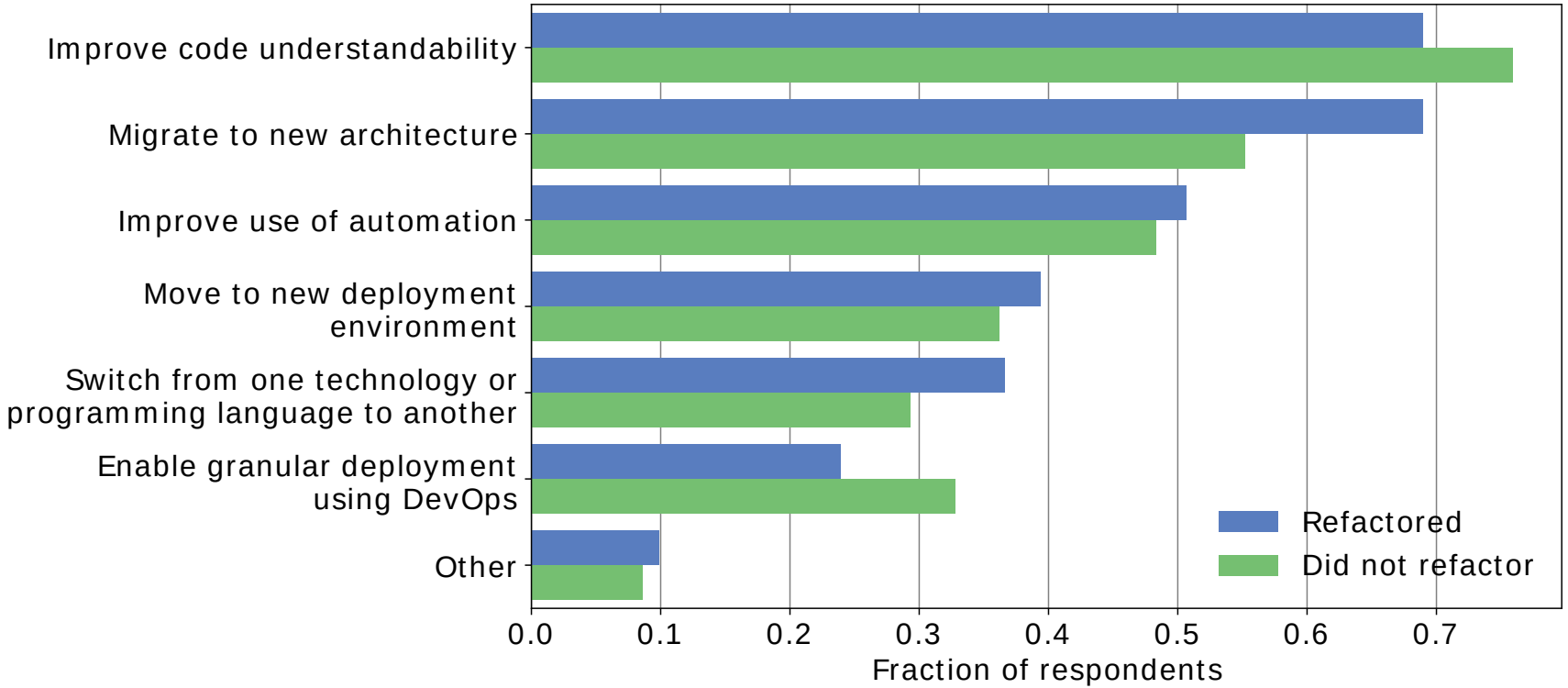


- Most respondents had performed LSR multiple times.
- Most systems on which they had performed LSR had undergone LSR multiple times.
- 86% of LSR estimates were in range of **months to years of effort** (a mean of 1,500 days)

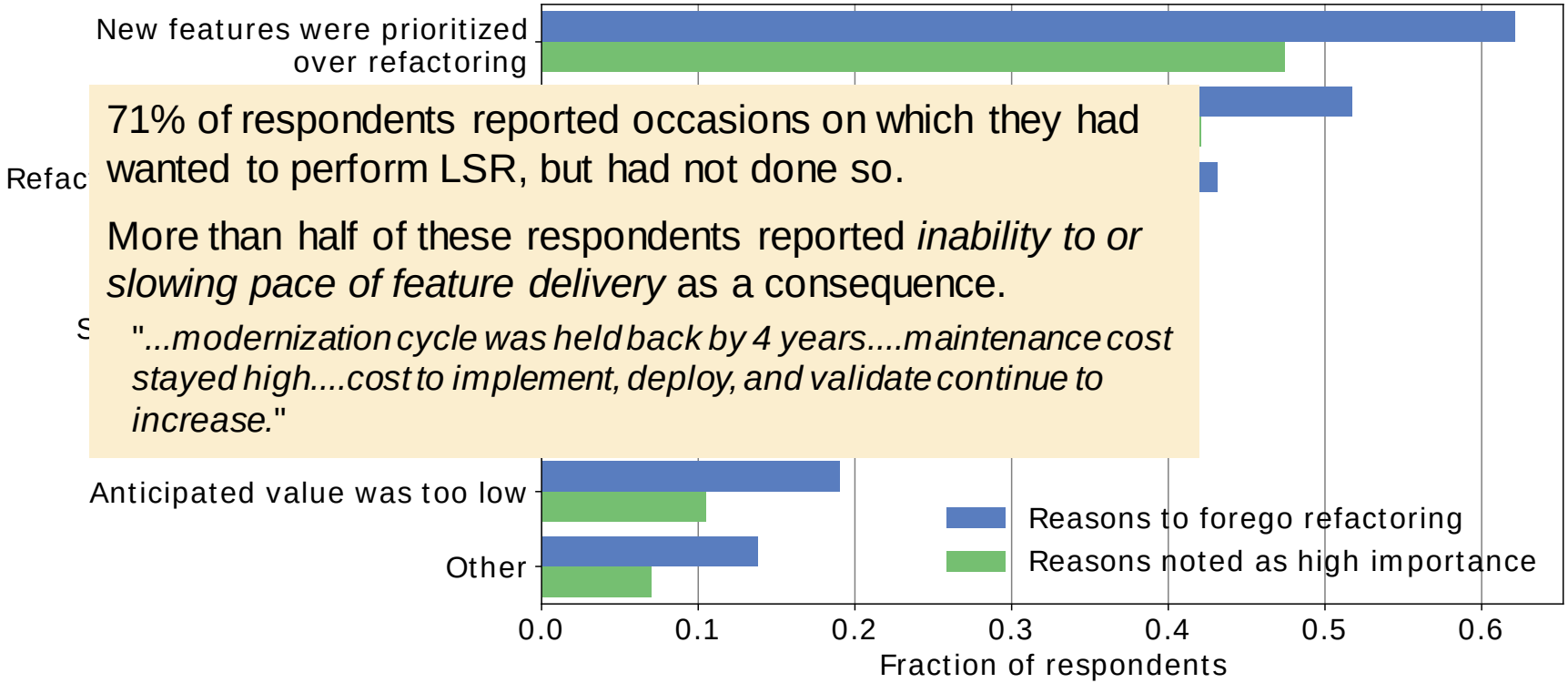
Business Reasons for LSR



Technical Reasons for LSR



Reasons for Forgoing LSR

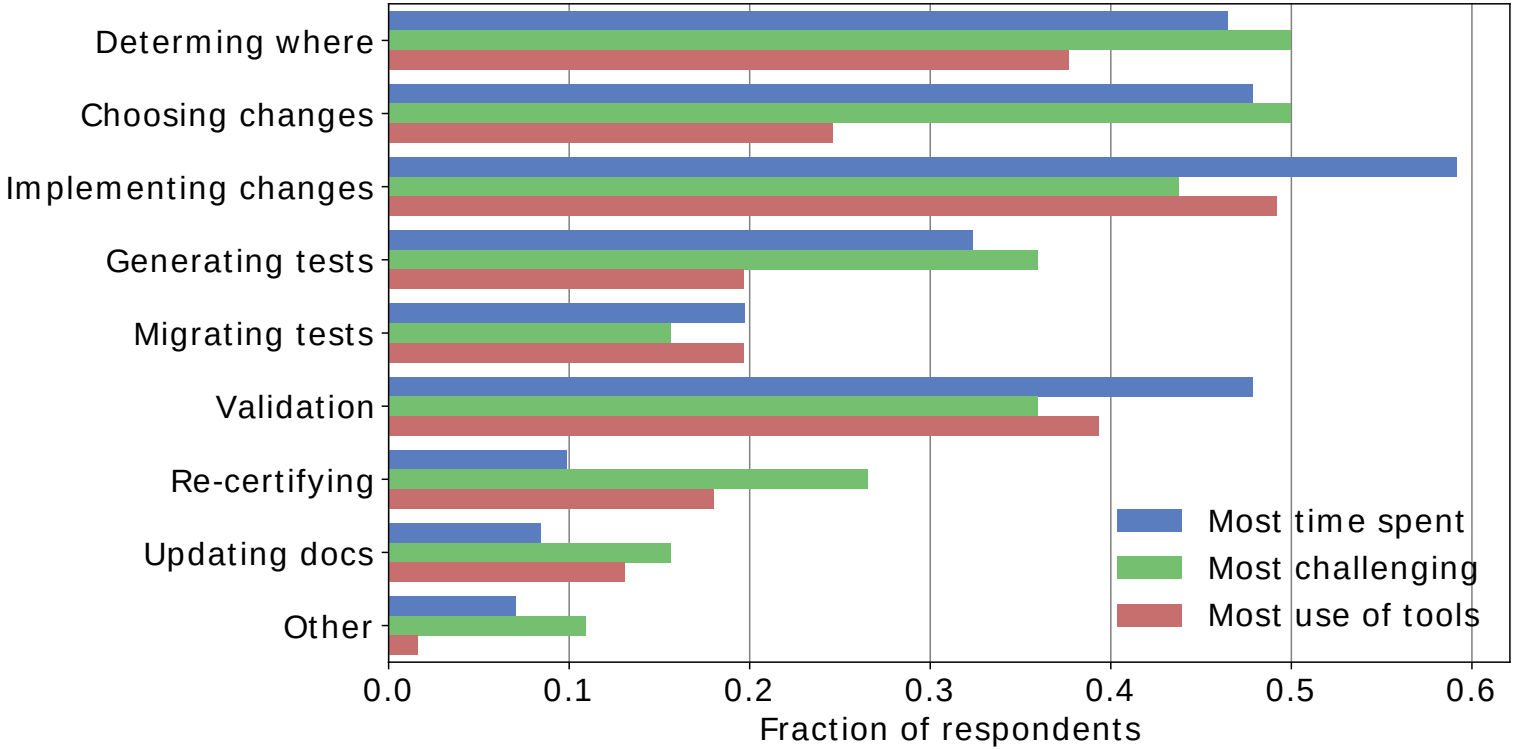


71% of respondents reported occasions on which they had wanted to perform LSR, but had not done so.

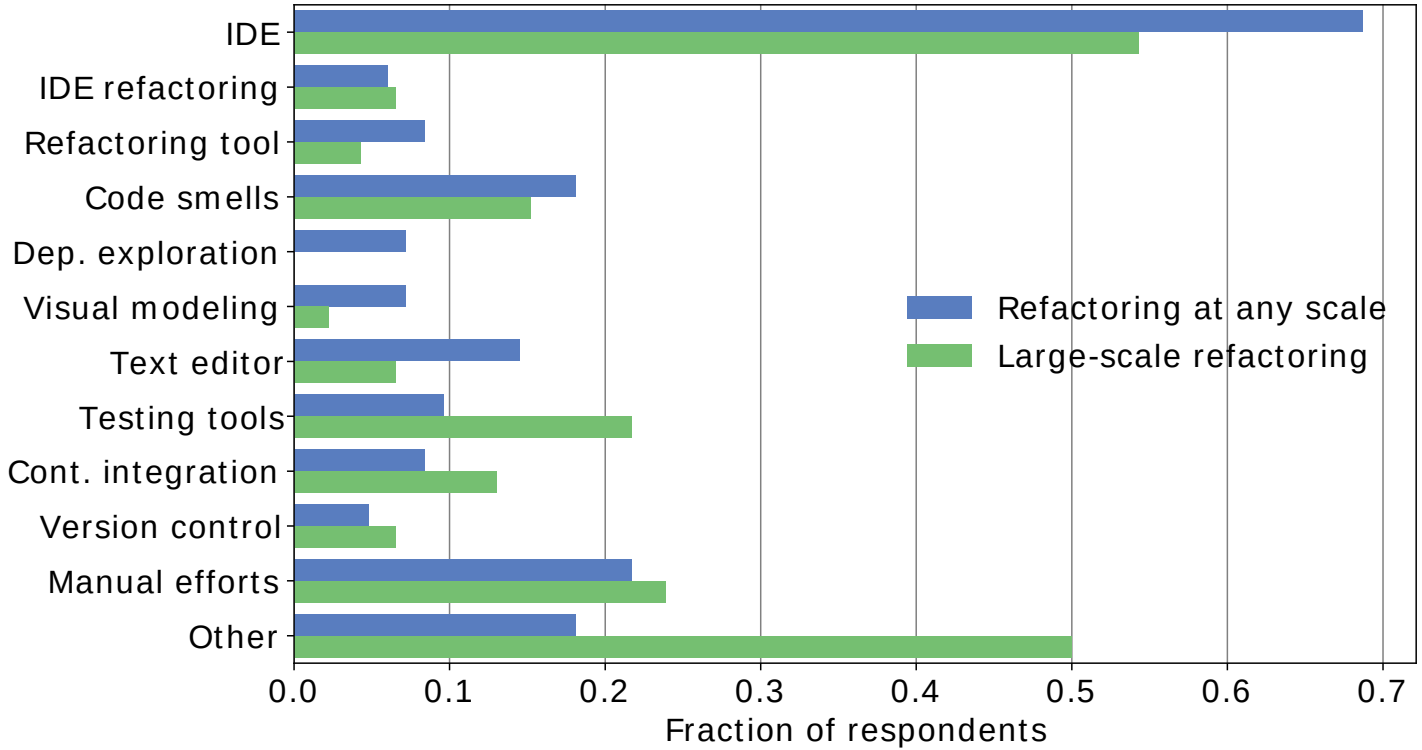
More than half of these respondents reported *inability to or slowing pace of feature delivery* as a consequence.

> "*...modernization cycle was held back by 4 years....maintenance cost stayed high....cost to implement, deploy, and validate continue to increase.*"

Activities Involved in LSR



Tools Used in Refactoring



Challenges Encountered in LSR

The top categories of responses to why LSR was challenging primarily address **pre-conditions to refactoring**

- 34% cited poor quality of the code, particularly excessive dependencies
- 26% cited difficulties understanding the code itself, as well as impact of changes
- 19% cited the lack of existing tests
- 19% cited a need to persuade management and teammates

Ancient, byzantine code that had grown organically over the course of decades. The hardest part was gaining a conceptual grasp of the overall code structure, and code flow, and understanding how one basic change – no matter how simple it appeared on on the surface – might create consequences throughout the system.

Interviewing Industry Developers

More recently, we have been interviewing industry developers to understand criteria used in deciding *how to refactor*.

Some recurring responses

- most share small-scale experiences
- many favor smaller incremental changes
- concern for changes that impact other services or teams is common
- many are skeptical of getting approval/ resources for large-scale refactoring
- few are using refactoring tools

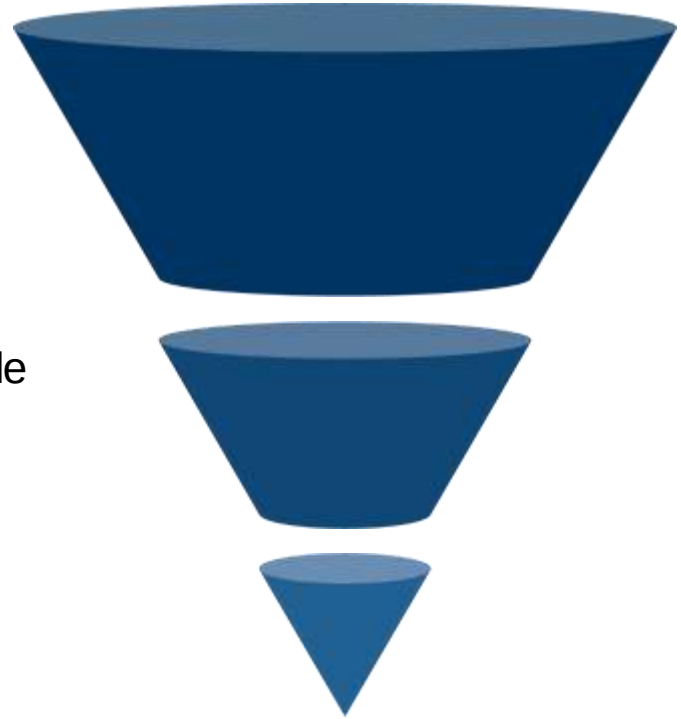
Examples of criteria that are emerging

- Change size
- Code size
- Cost/time
- Design elegance
- Ease of testing
- Extensibility
- Performance
- Propagation of changes
- Readability
- Well-defined APIs

Distilling Implications of Scaling

Scaling Refactoring

Comments from our survey and interviews paint a picture of multiple interacting factors.



Large-Scale Refactoring

- Tends towards broad changes requiring substantial coordination and effort to achieve business goals

Refactoring Sprints

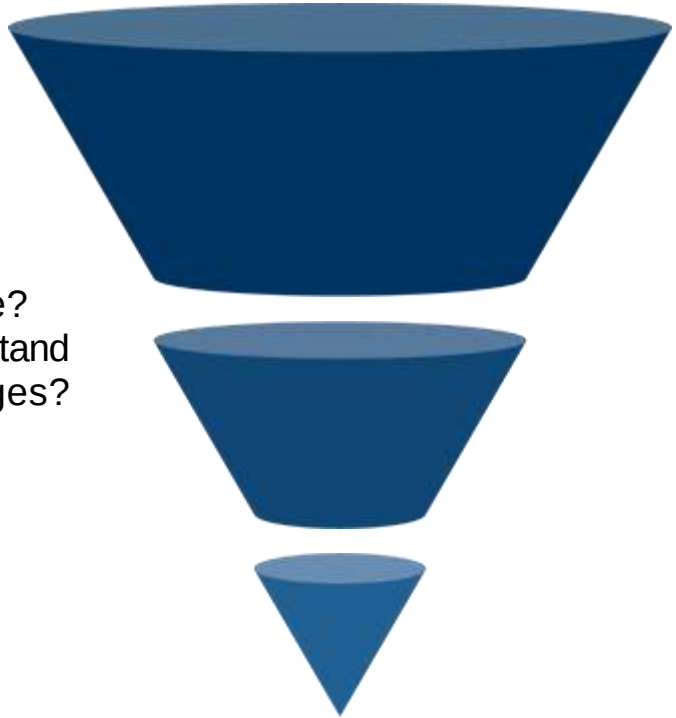
- Tends towards short bursts focused on refactoring to achieve a local, but strategic goal like removing specific technical debt

Floss Refactoring

- Tends towards continuous, but opportunistic improvements that gradually improve code health

Scaling Refactoring – Code Scope

Code Scope
- How much code to change?
- How much code to understand in order to make the changes?



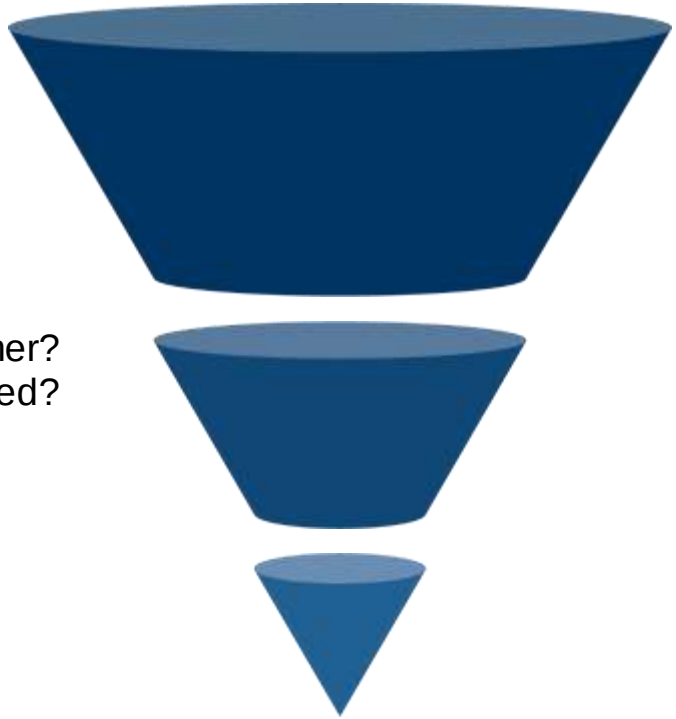
Large-Scale Refactoring
- Potentially system-wide changes
- More likely to break service contracts or change infrastructure
- Discovery and impact analysis activities get more complicated

Refactoring Sprints
- Potentially many classes, but restricted to a scope like a service

Floss Refactoring
- A few classes

Scaling Refactoring – Coordination

Coordination
- How many developers or teams need to work together?
- Who needs to be persuaded?
- How will testing and deployment be aligned?



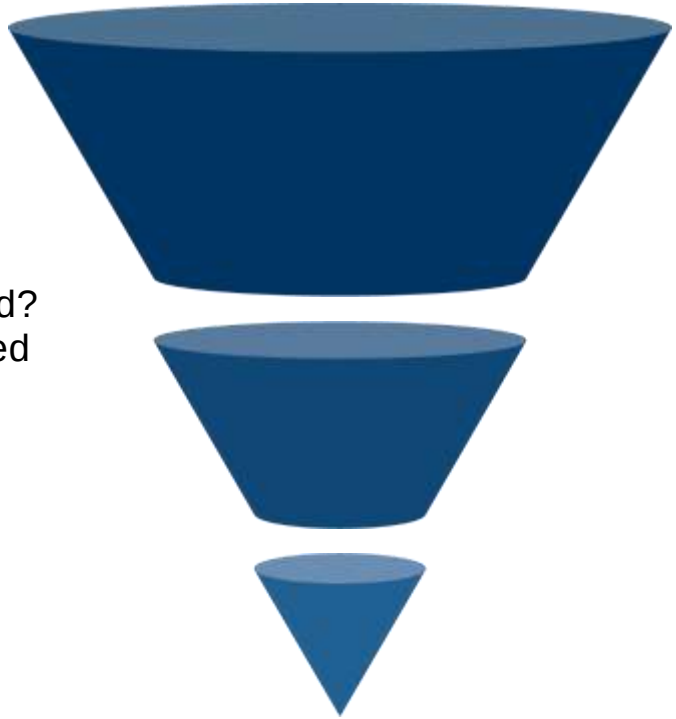
Large-Scale Refactoring
- Multiple teams need to agree
- Interface changes require coordinated test and deployment
- Schedule impact and team availability complicate timelines

Refactoring Sprints
- Multiple developers (same team)
- Testing and deployment restricted to a single service

Floss Refactoring
- All changes can be integrated in a single PR and tested locally

Scaling Refactoring – Effort

Effort
- How much work is required?
- How many developers need to be assigned?
- How long will it take?



Large-Scale Refactoring
- Multiple teams of developers
- Months to years of effort

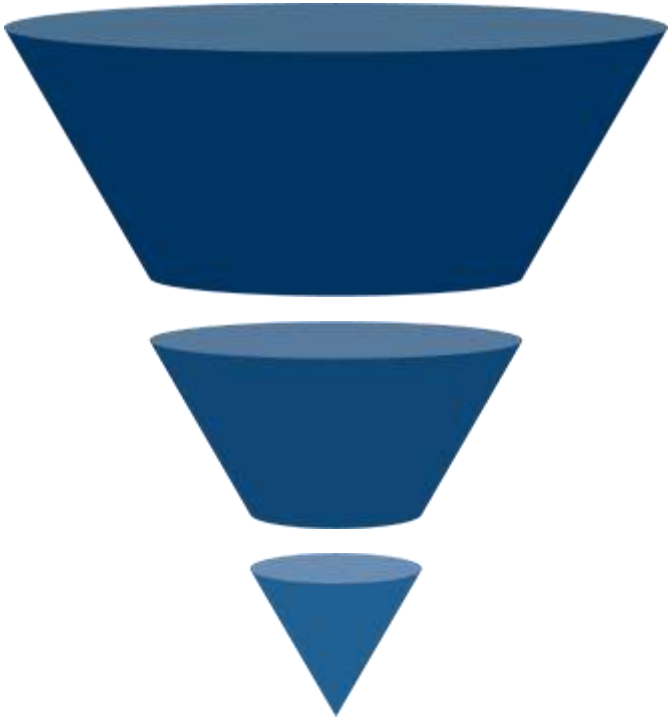
Refactoring Sprints
- A single team of developers
- Often time-boxed (e.g., a single two-week sprint)

Floss Refactoring
- A single developer
- Minutes to hours of work

Scaling Refactoring – Decision Authority

Decision Authority

- Who has the authority to approve the work?
- What kind of costs and benefits speak to the decision maker?



Large-Scale Refactoring

- May require executive-level approval Must deliver clear business value
- Demonstrating incremental value can be essential to retaining funding

Refactoring Sprints

- Team lead has authority
- Removing technical debt that impacts velocity of team's work is one goal

Floss Refactoring

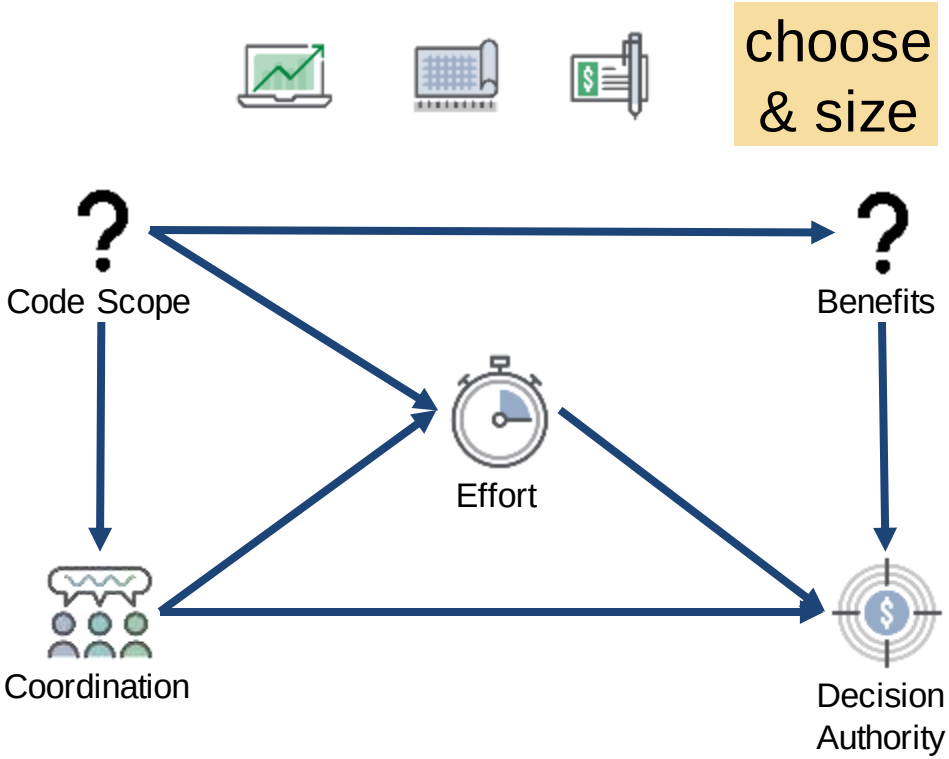
- Developer autonomy
- Making ones' own life easier can be enough (e.g., more readable code)

Factors Influencing Approval for Large-Scale Refactoring

As you aim for larger changes, you can accomplish bigger things. But, you also need to involve more people and plan changes more carefully.

Larger changes likewise take more work and have bigger schedule impacts. Non-technical factors take on greater prominence.

As the cost increases, approvals require increasingly senior staff. Business value, as perceived by less technical staff, drives decisions.



What Can We Do?

Observations

Survey responses and interviews point to challenges

- Pre-conditions for refactoring are a significant barrier
  - Availability of tests
  - Understanding how existing code works
- Tool needs increase with scale
  - A holistic tool doesn't exist yet
- Non-technical concerns increase with scale
  - Team coordination
  - Scheduling activities
  - Need to persuade others
- It's **hard to get approval** for larger refactorings

Comprehension Challenges

Many comments come from developers trying to understand code written by someone else.

- Missing or out-of-date documentation
- Highly coupled code

Legacy code, original authors moved on, original requirements were not clear or had changed significantly over long periods of time, architectural drift.

Recommendations

- Generate higher level (architecture or design) views of existing code
- Enrich views with missing dependency information
- Better support for a wider range of languages

Anything that could have given me a conceptual overview of how the components of the system fit together, and influenced one another. Code diagrams or visualizations would have been most helpful. For instance, will this function behave differently depending on the value of some given global variable?

Testing Challenges

Many comments related to testing focused on

- Lack of initial tests
- Concern for confirming behavior preservation
- Time and effort implications of missing tests

Recommendations

- Generate before and after tests for proposed changes
- Generate non-functional tests to assess quality changes

code coverage was quite low, so we worked with fear of breaking things (which we did in the end) and that made us overthink and double-triple check.

We reorganized the project from the monolith structure to microservices and changed the frontend framework. The biggest issue was to certify that no functionality was broken as there was a small number of tests in place.

Stakeholder Challenges

Non-technical concerns were more common with lager refactoring efforts

- Building support with management and *peers*
- Staging activities across teams
- Conveying value in appropriate terms

We understood cloud and dev ops. The hard part was figuring out the legacy applications and getting cooperation from the people that managed the legacy applications.

Getting the management to support the refactoring

Recommendations

- Estimation models for refactoring
- Incremental refactoring strategies
- Tracking progress and identifying stable checkpoints for incremental review, test, and deployment

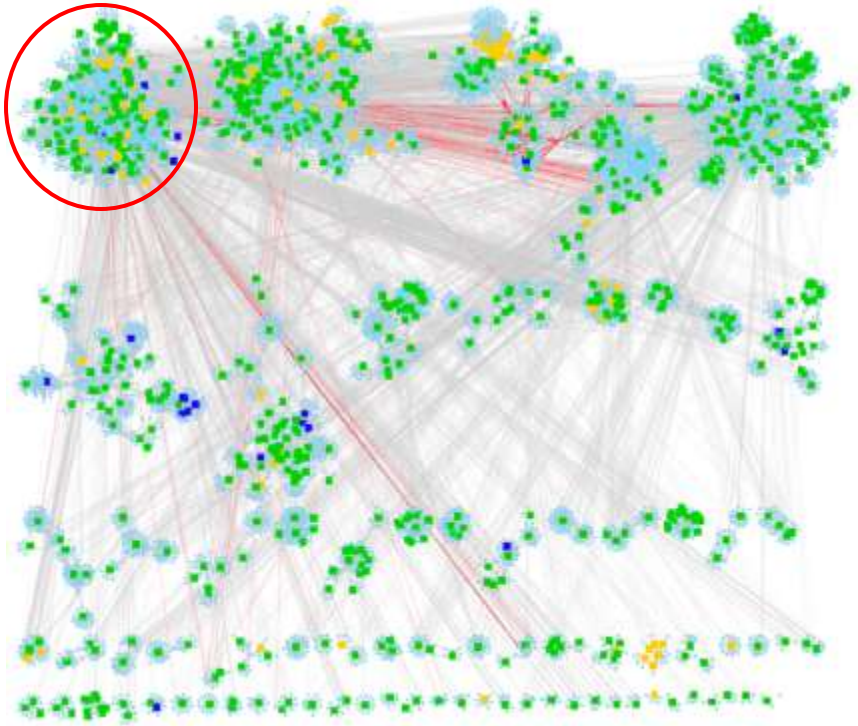
static code analyzer on an architectural level to determine the efforts and tracking refactoring activities over time

Untangling the Knot

SEI is working on a tool to help with some aspects of large-scale refactoring

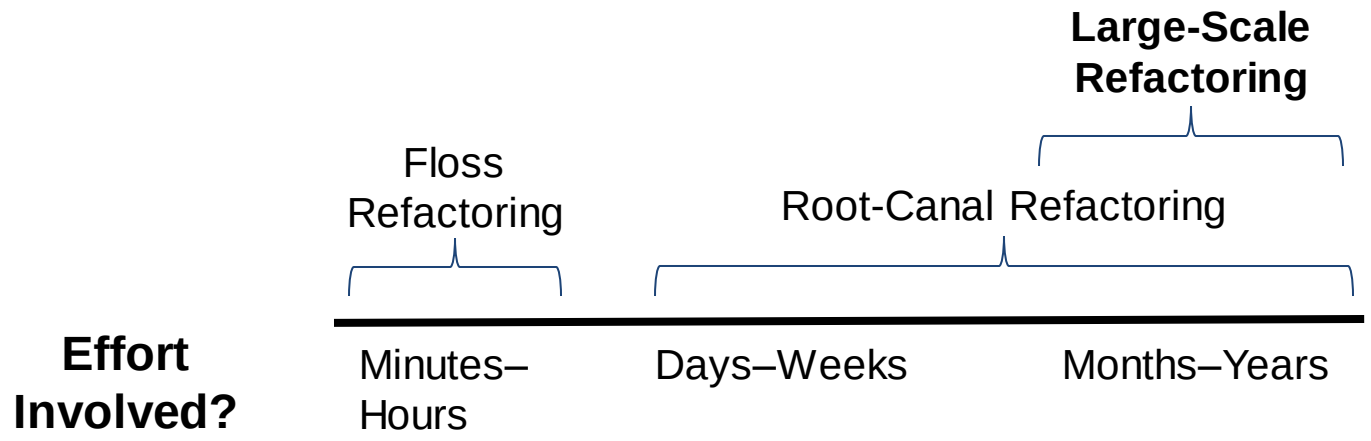
- Using search-based software engineering to generate refactoring recommendations to user-driven problems
- Focus on software isolation
- Oriented towards strategic reuse and modularization of existing code

https://www.sei.cmu.edu/go/knot



J. Ivers, C. Seifried, I. Ozkaya. **Untangling the Knot: Enabling Architecture Evolution with Search-Based Refactoring.** *19th IEEE International Conference on Software Architecture (ICSA 2022)*. March 2022.

Improving Our Understanding of Scaling Refactoring



- Effort increases for technical and non-technical reasons.
- Our tools are more oriented towards performing refactoring than enabling refactoring.
- New/better tools can improve productivity, awareness/tracking, estimation/rationale, etc., all of which are more noticeable at scale.

THANK YOU!

Document Markings

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.

DM22-0909