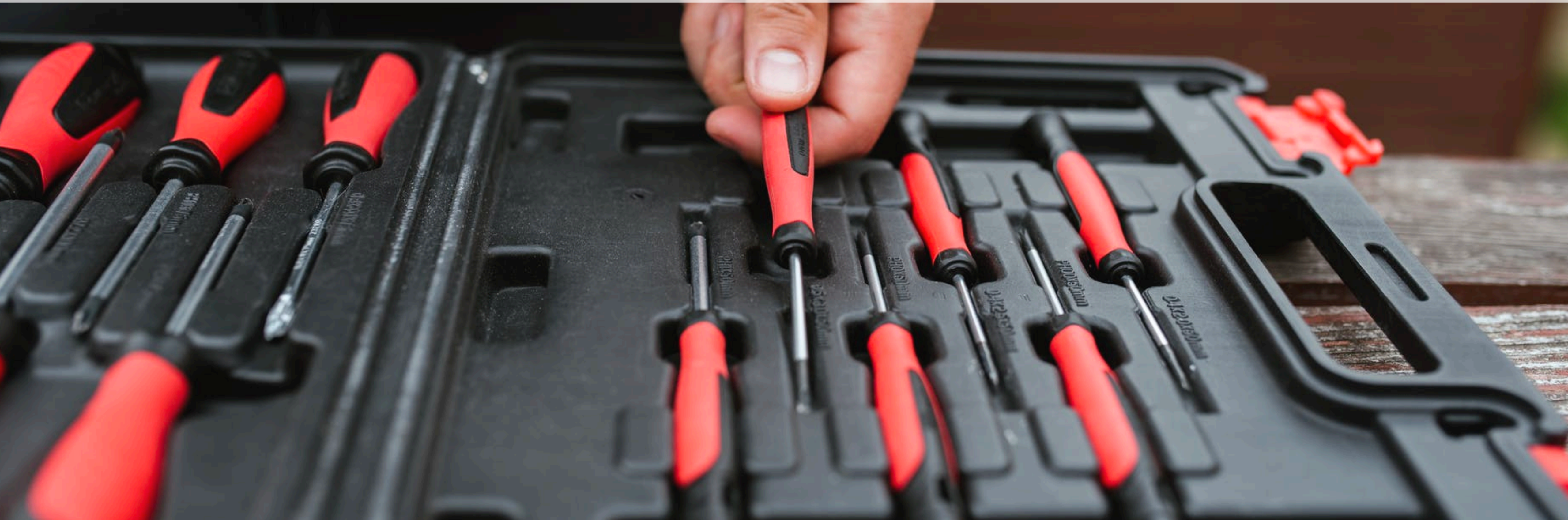
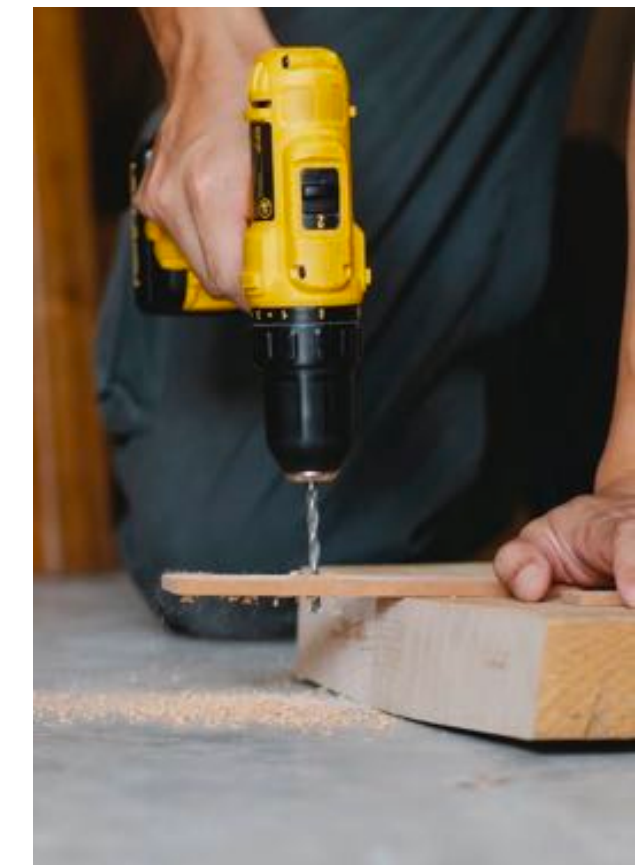
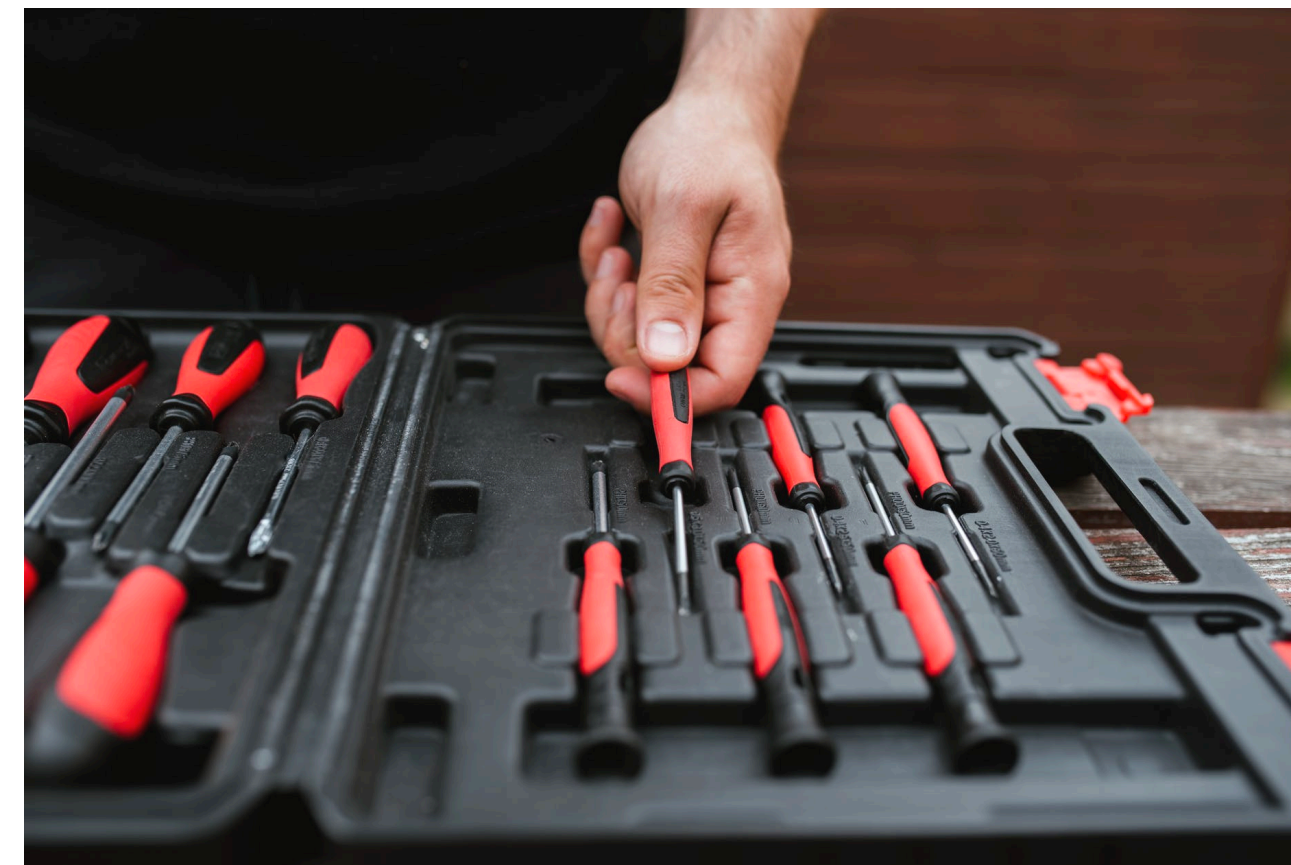


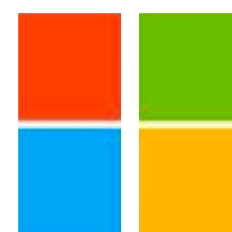
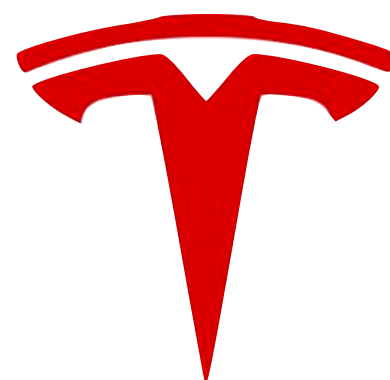
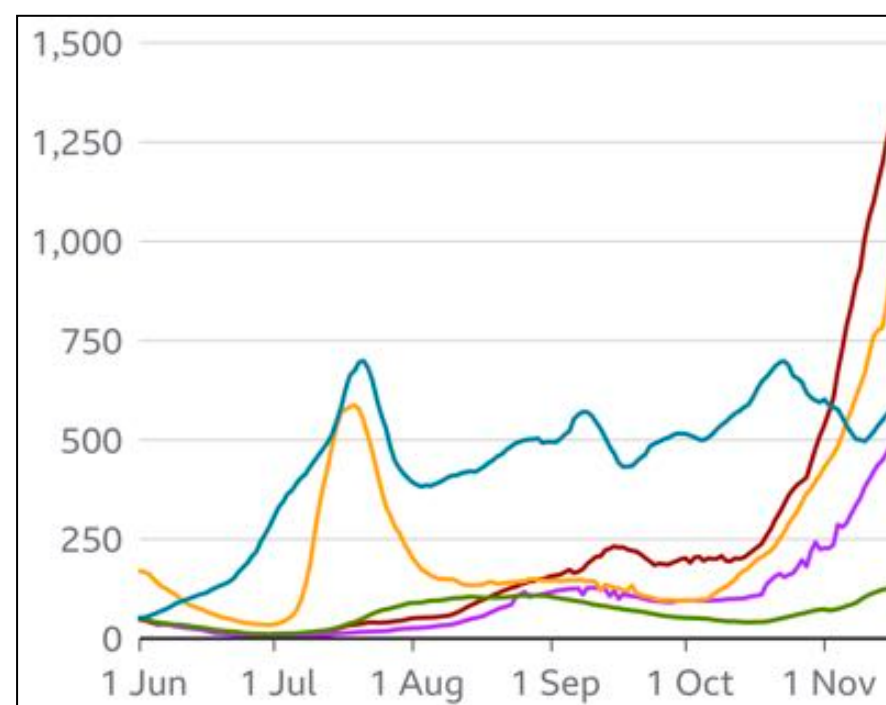
Improving the Usability of Refactoring Tools for Software Change Tasks



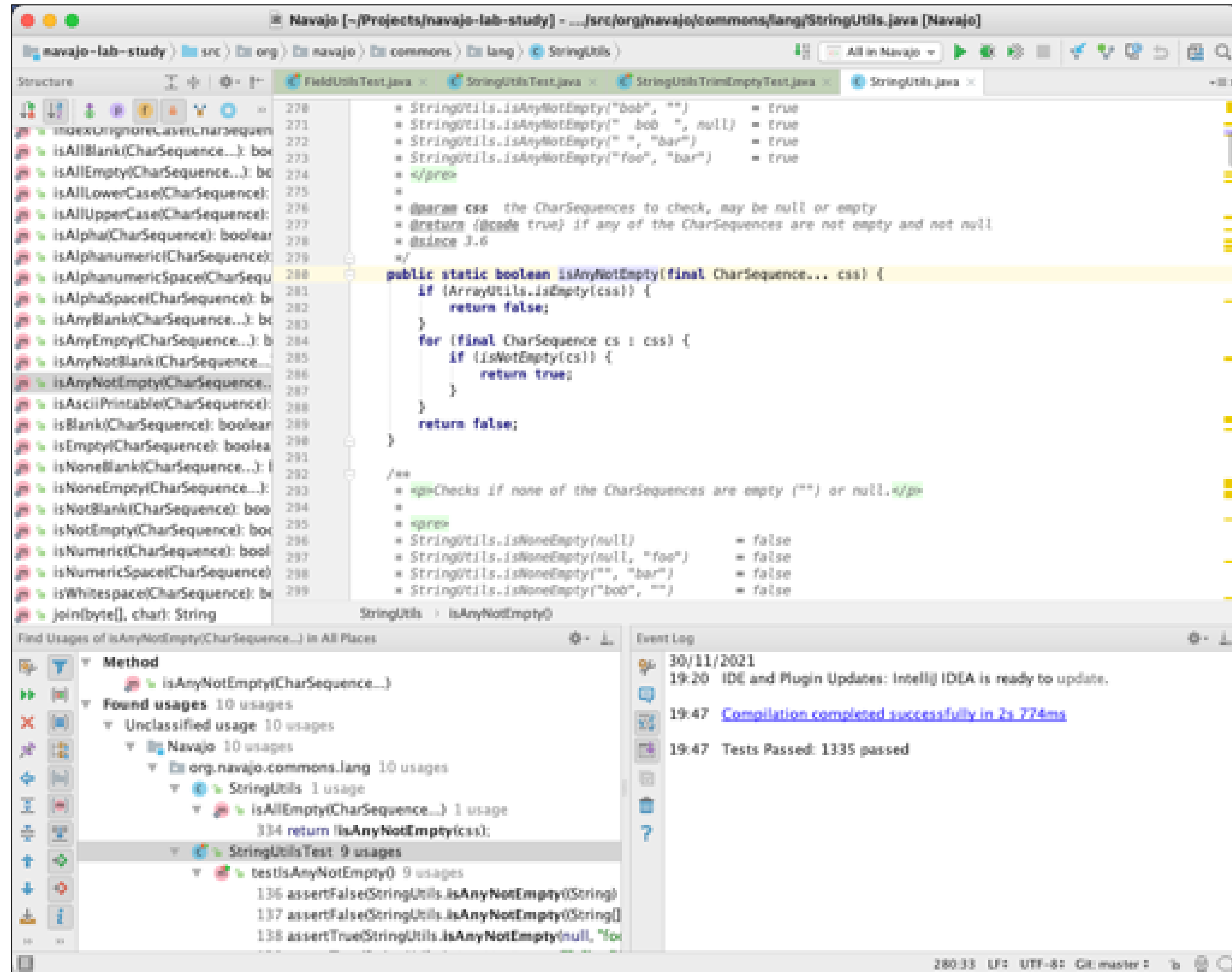
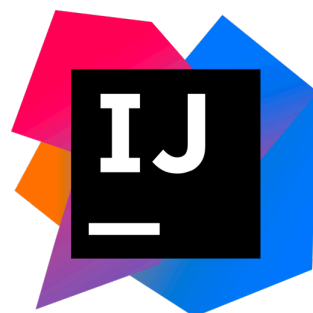
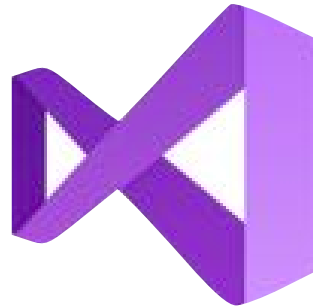


Tool support

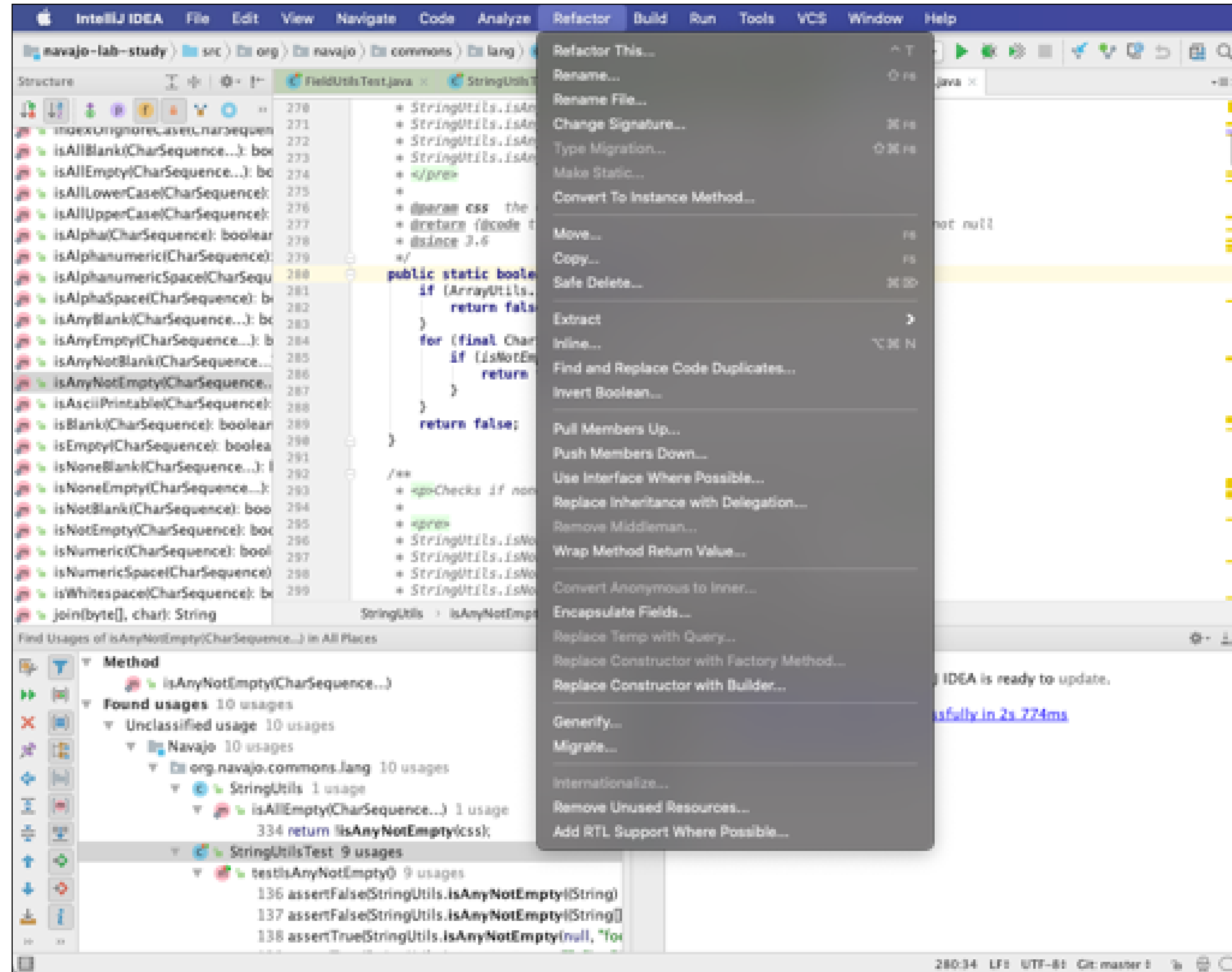
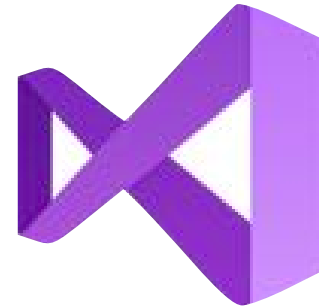




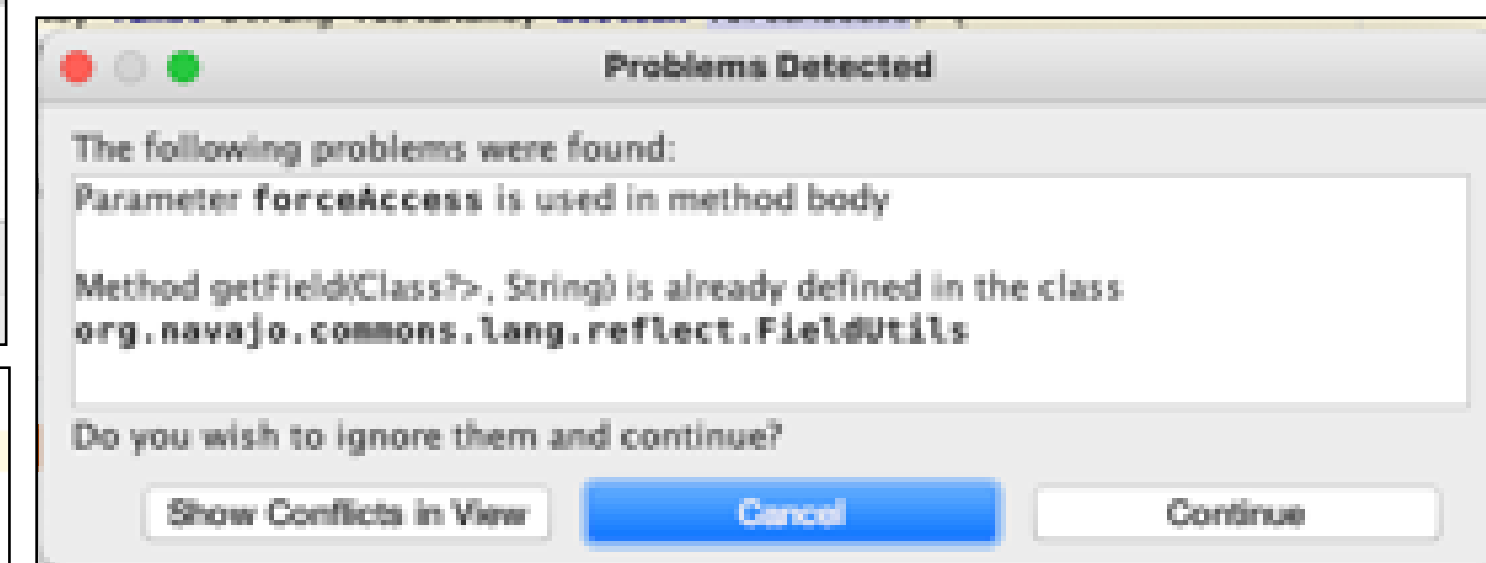
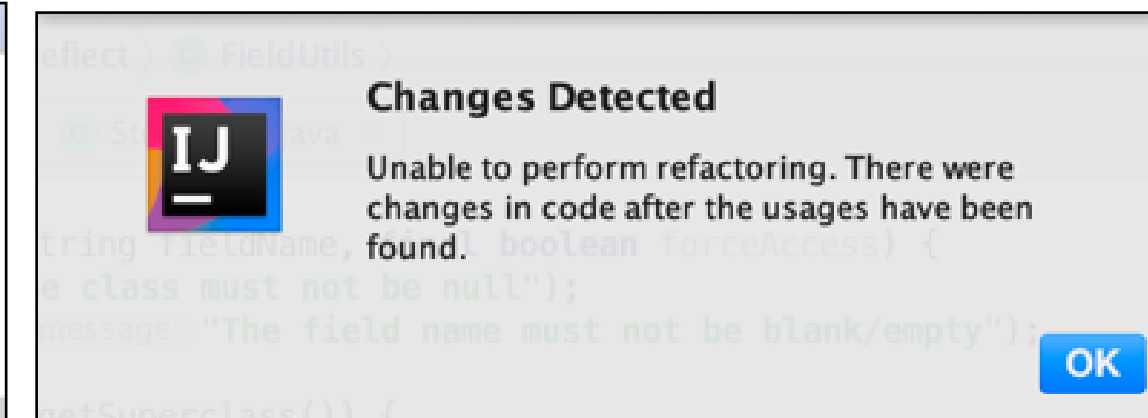
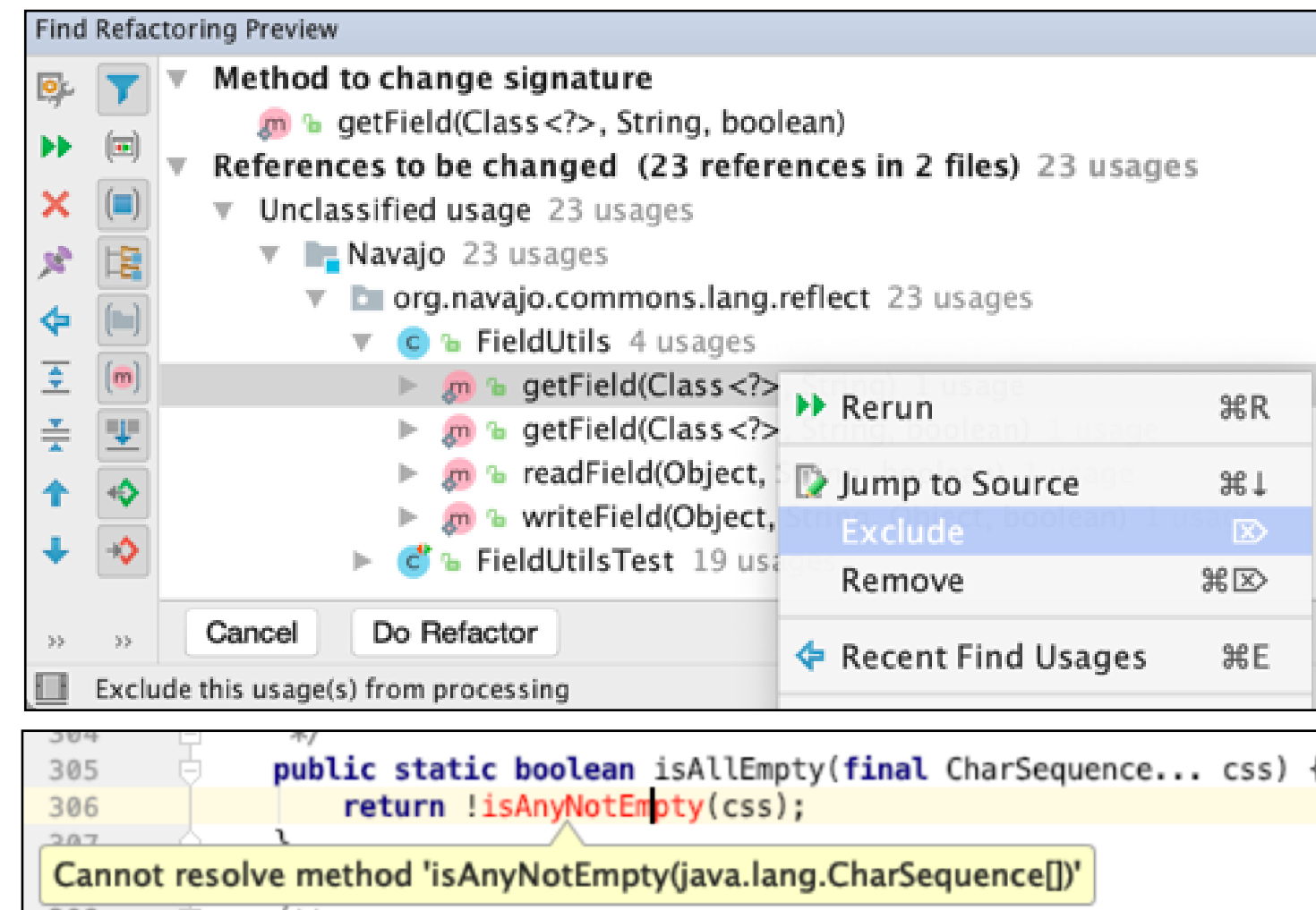
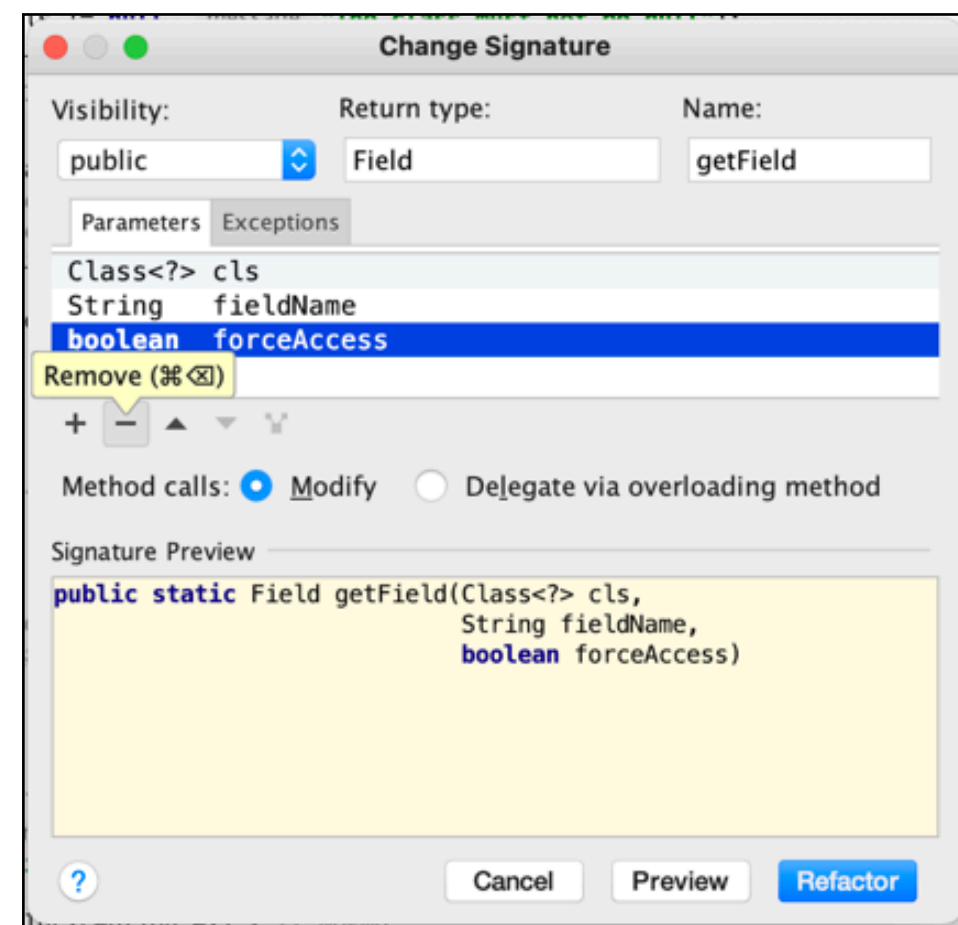
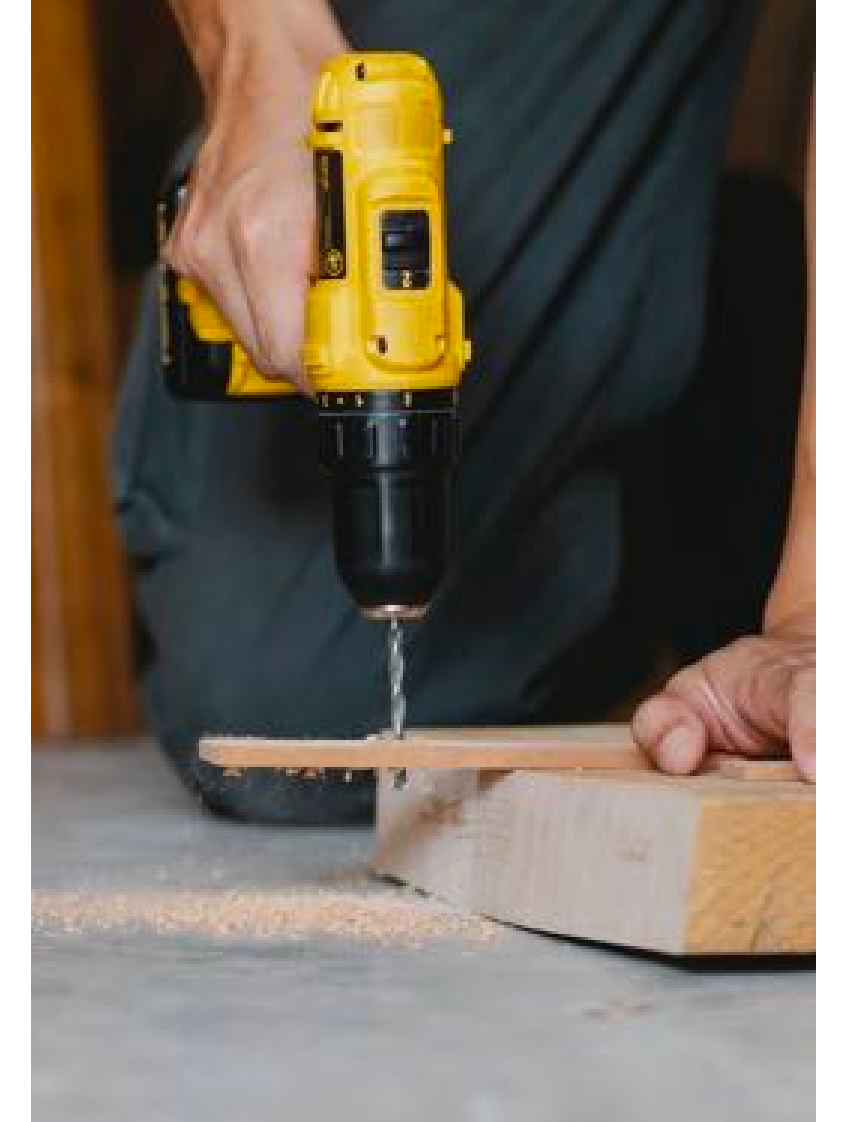
Tool support



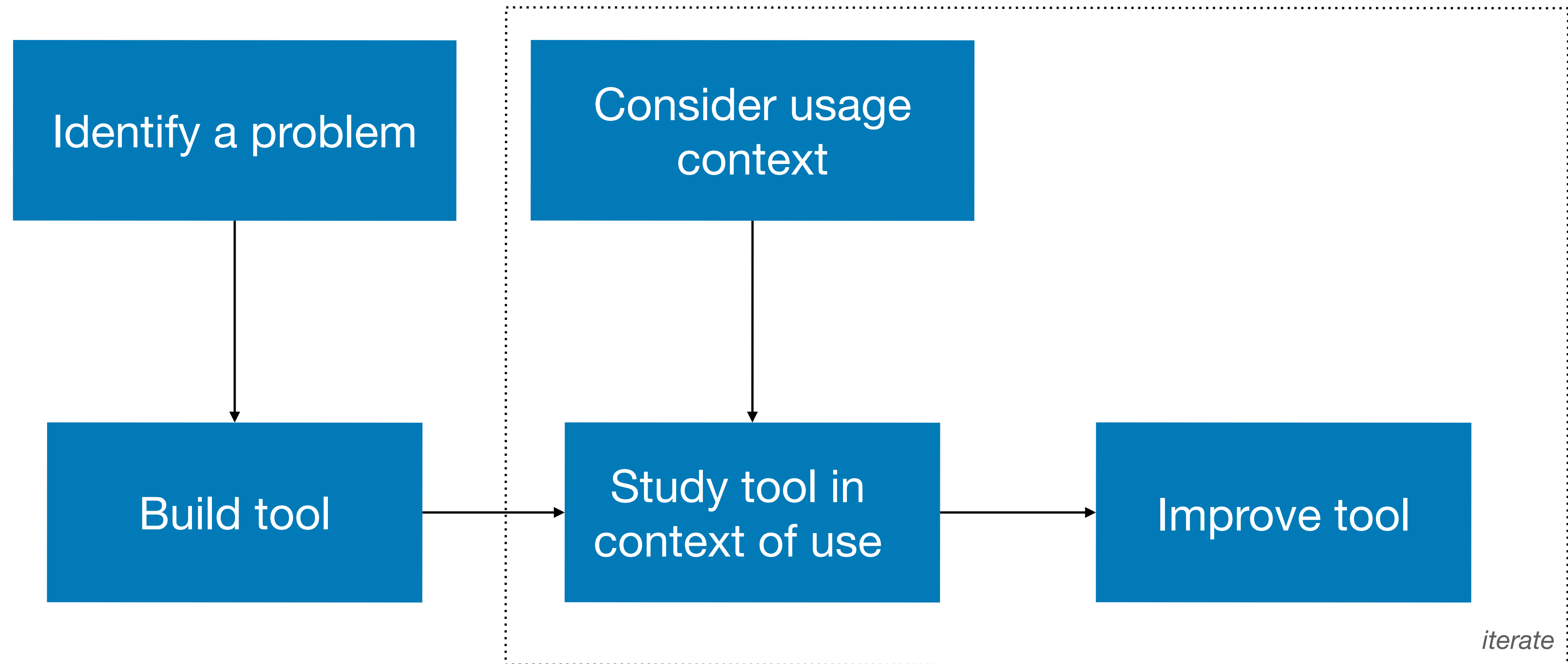
Tool support

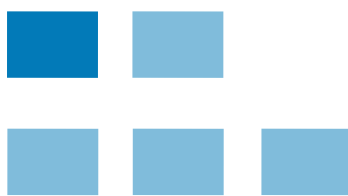


Tool support



Tool design



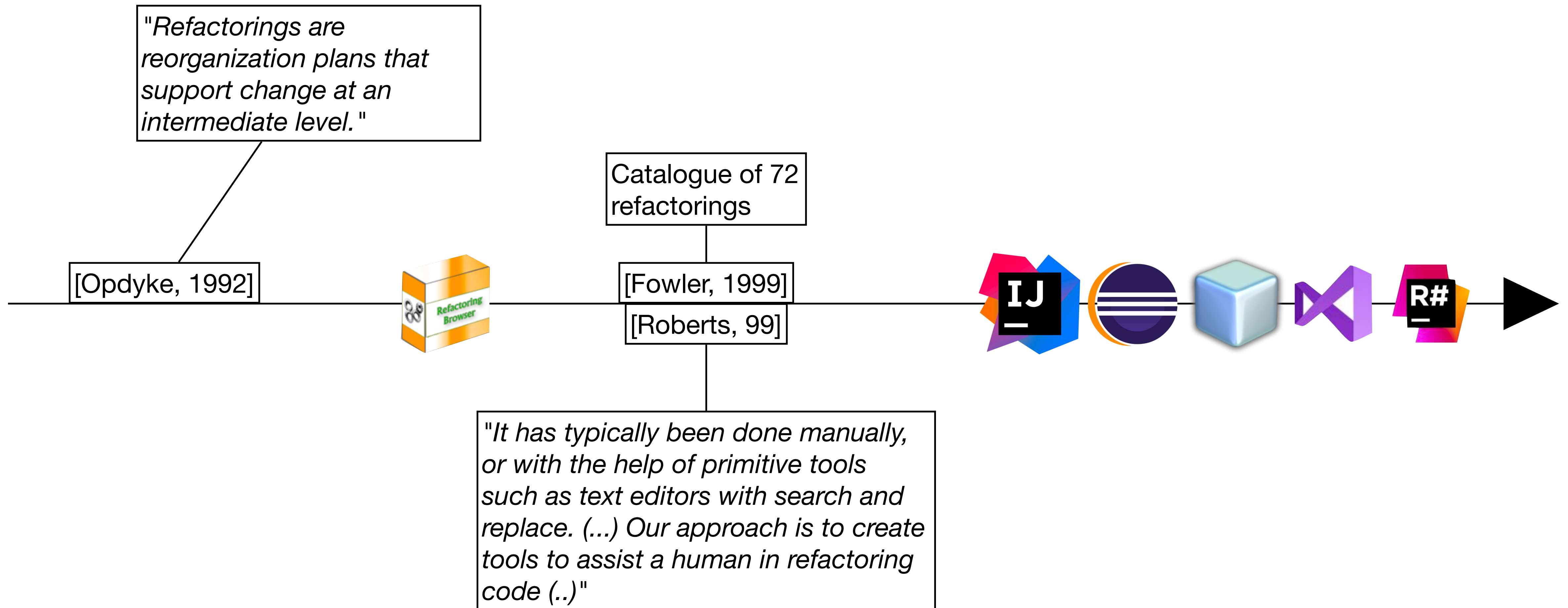


Refactoring tools

[Griswold, 1991]

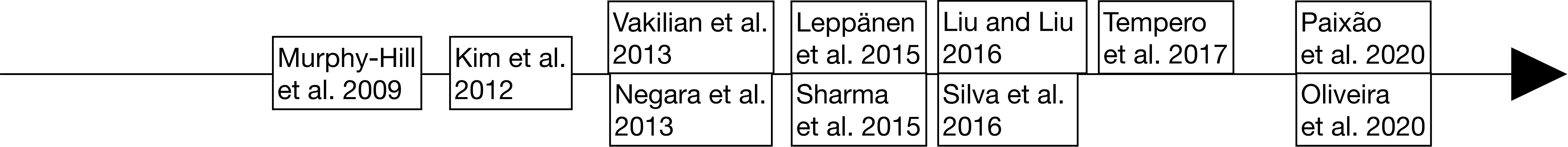
"Maintenance tends to degrade the structure of software, ultimately making maintenance more costly. At times, then, it is worthwhile to manipulate the structure of a system to make changes easier. However, it is shown that manual restructuring is an error-prone and expensive activity."

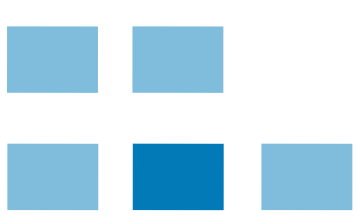
Refactoring tools



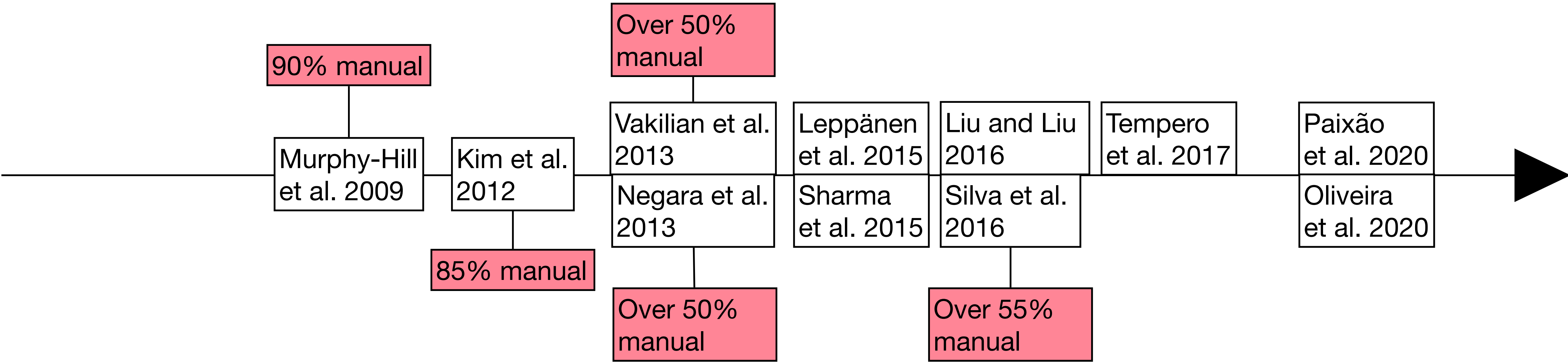


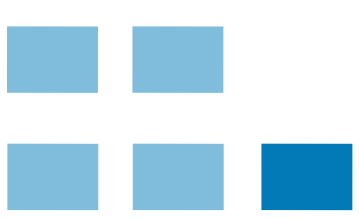
Refactoring tools



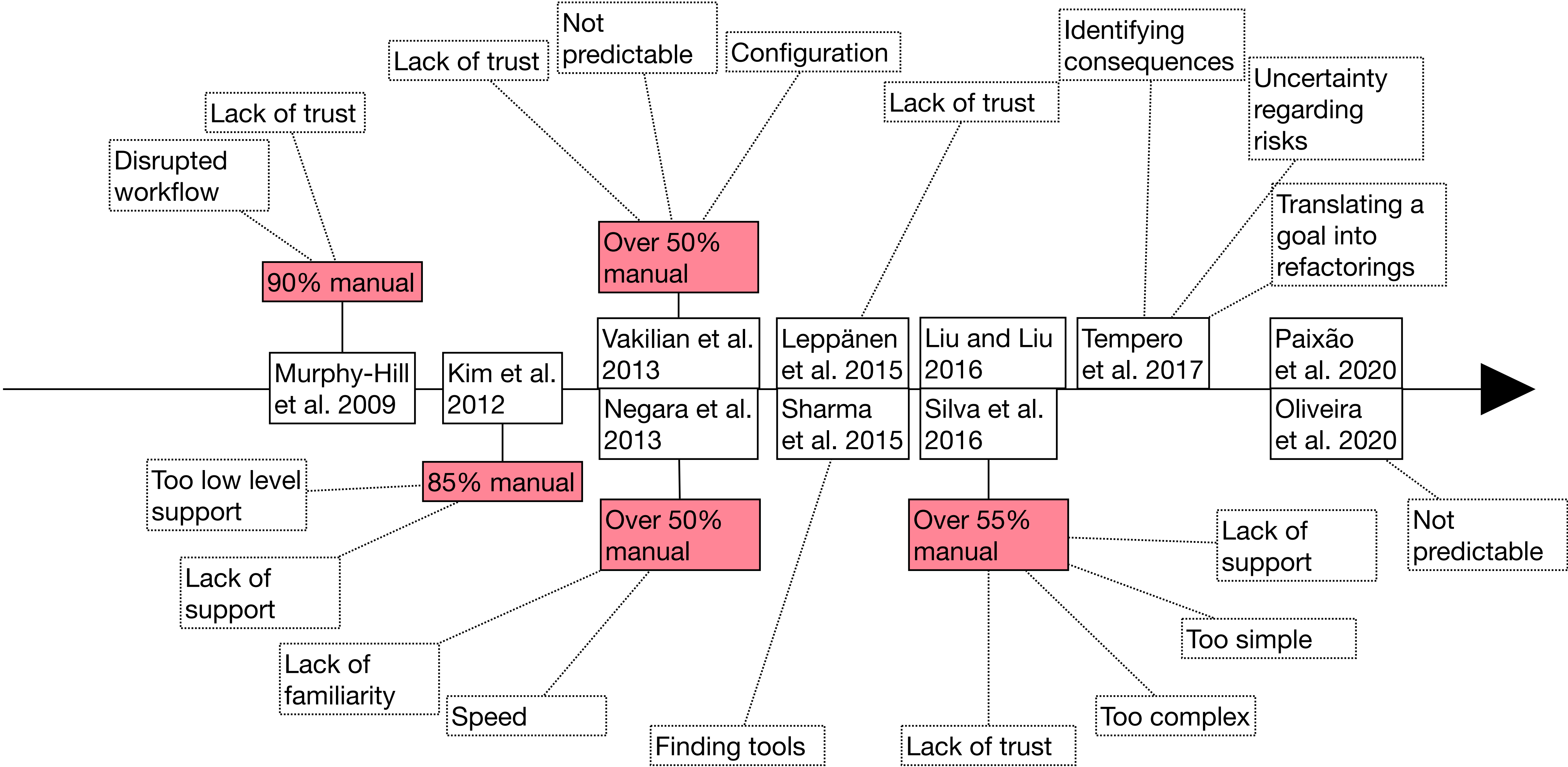


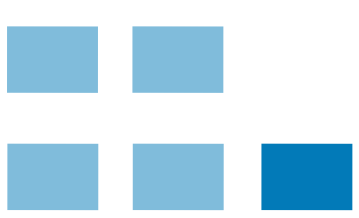
Refactoring tools



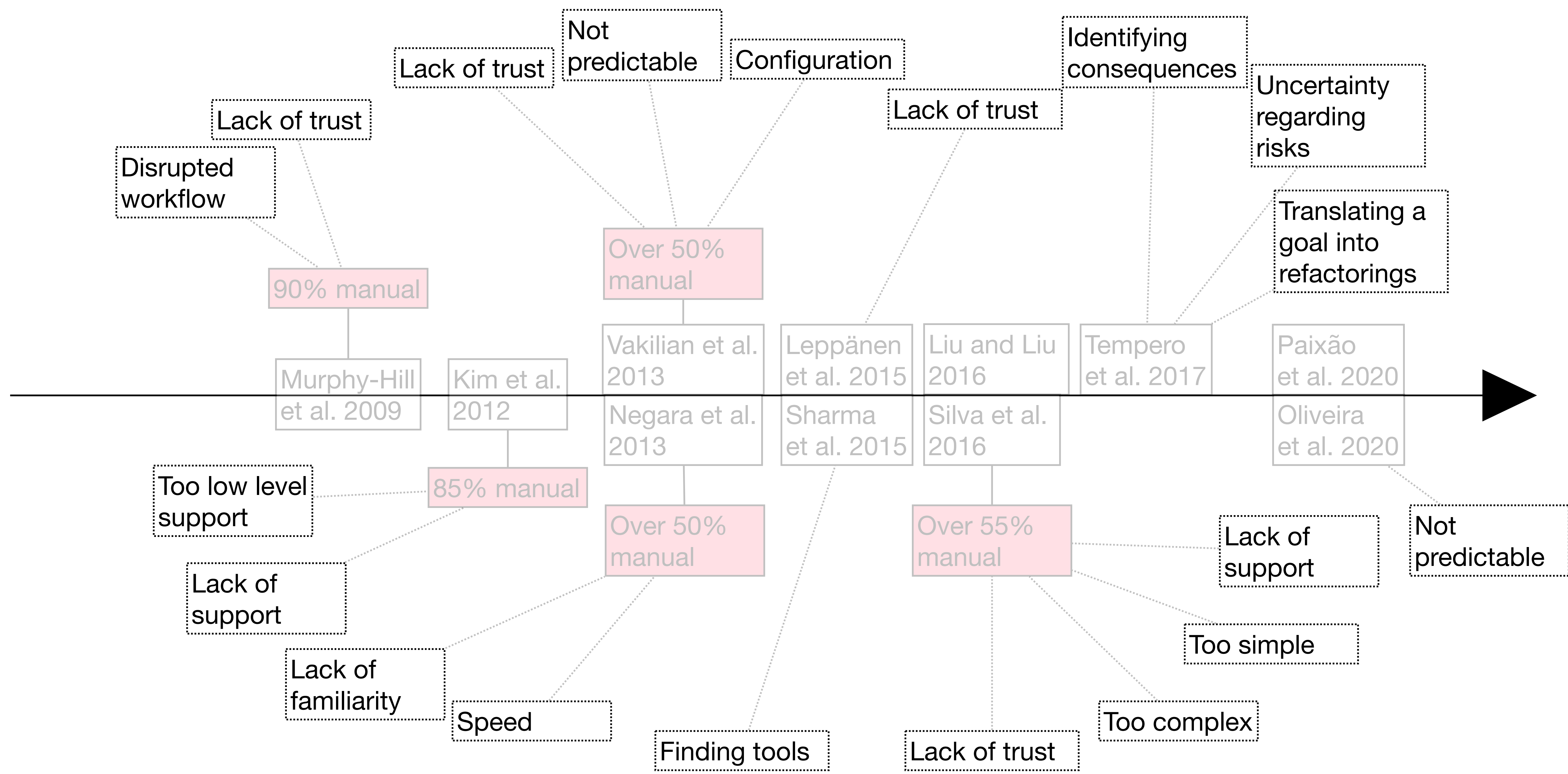


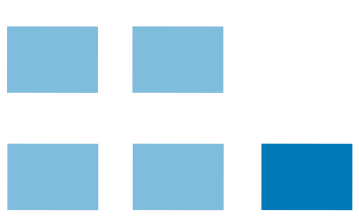
Refactoring tools



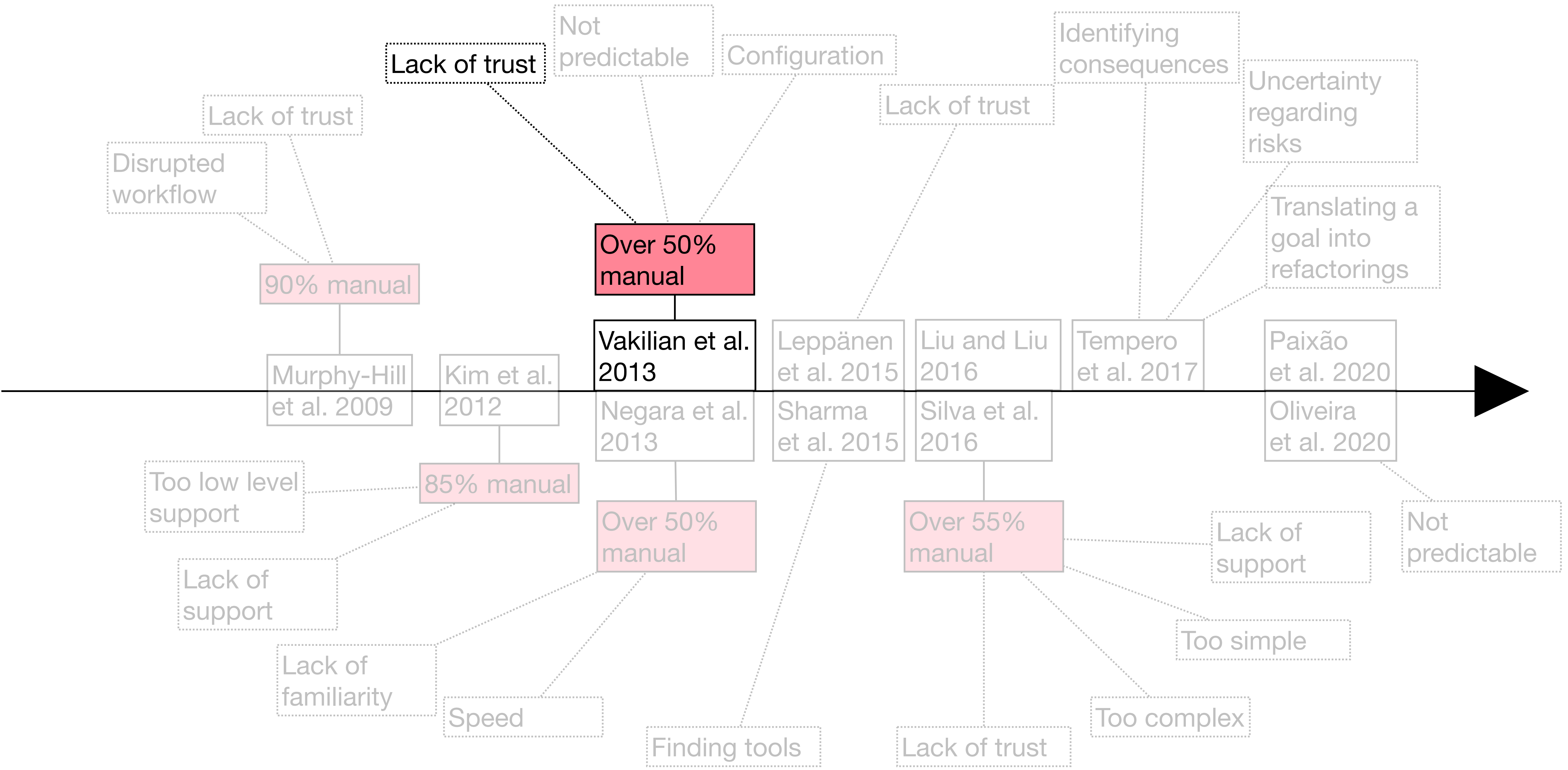


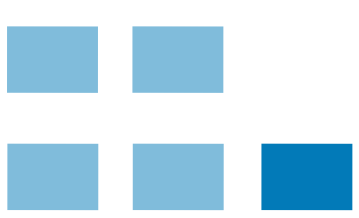
Refactoring tools



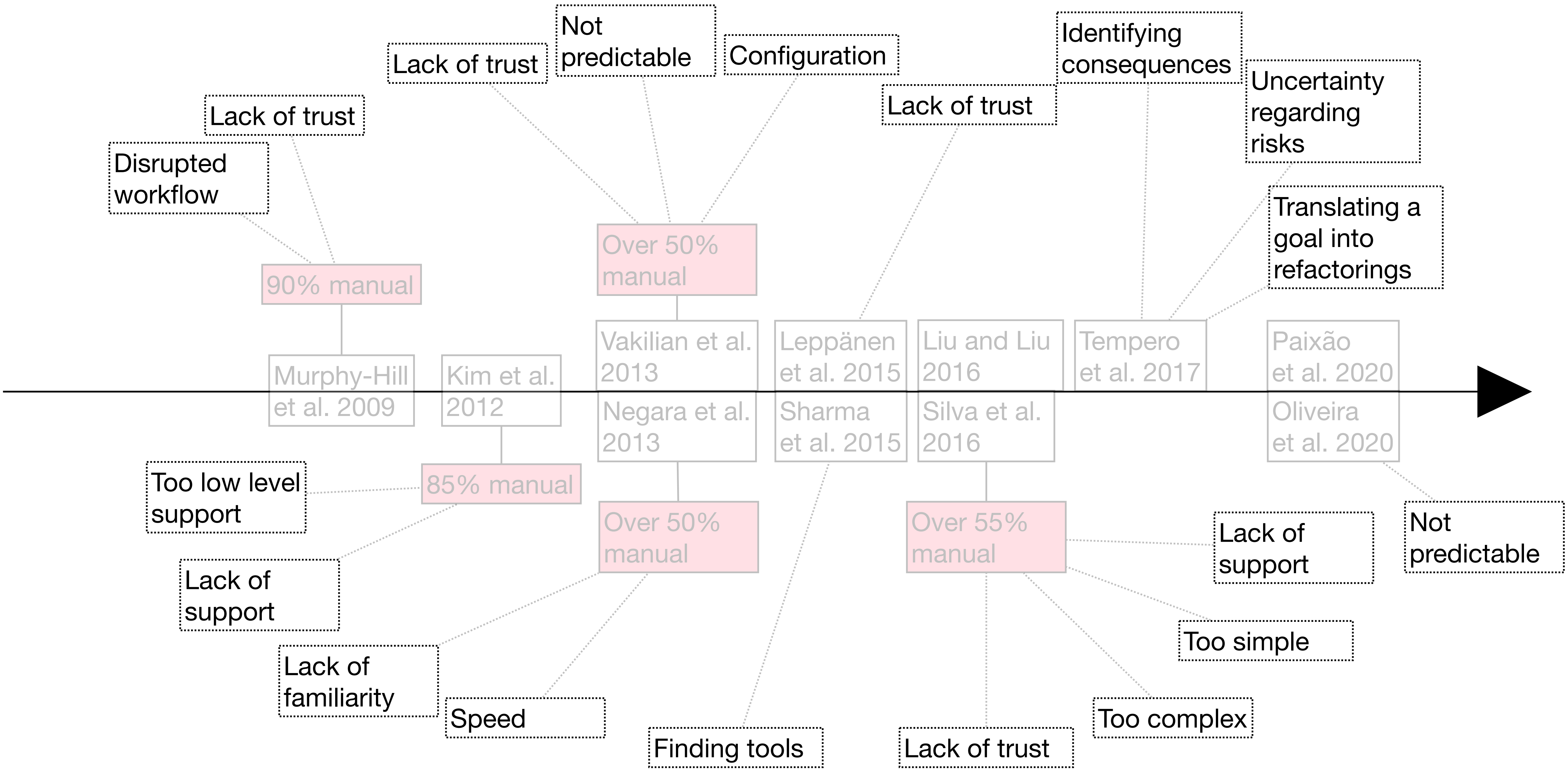


Refactoring tools

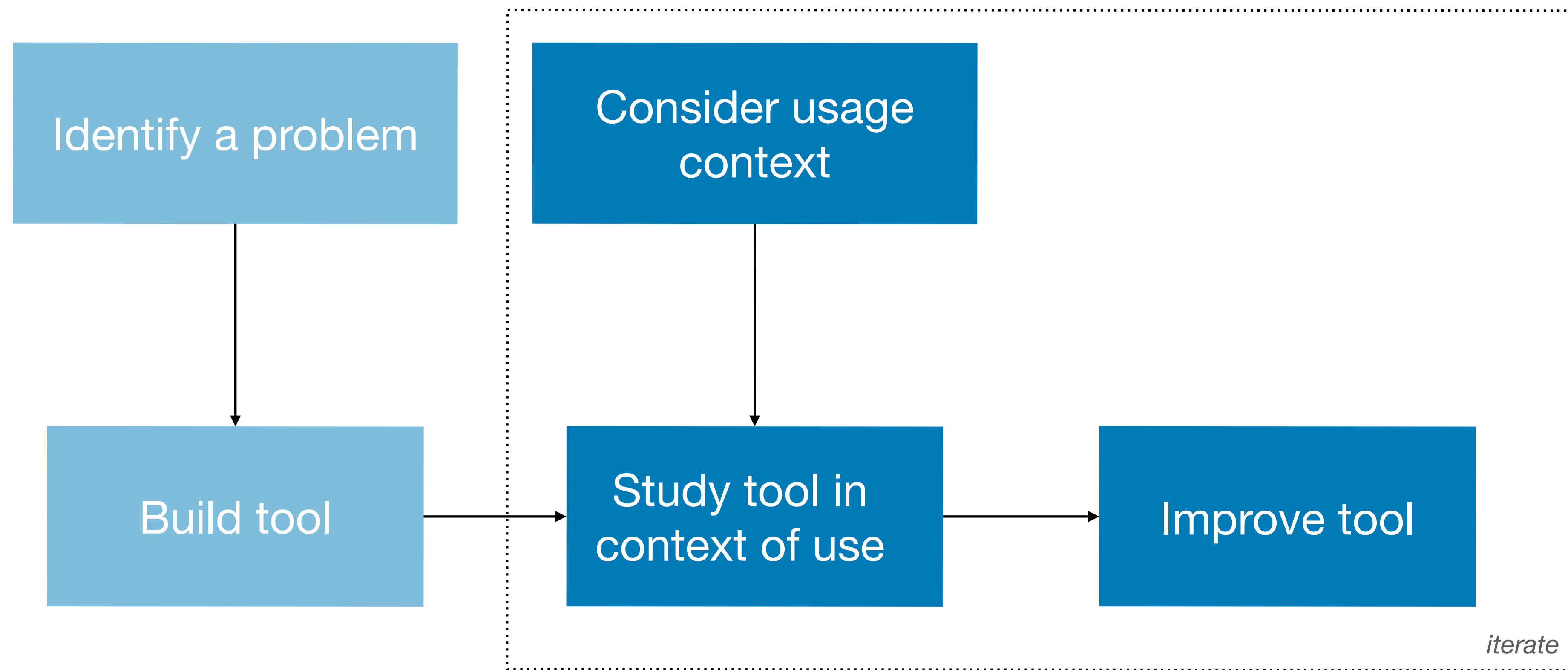




Refactoring tools



Improving the Usability of Refactoring Tools for Software Change Tasks



Improving the Usability of Refactoring Tools for Software Change Tasks

Paper	Pages
I A Study of Refactorings During Software Change Tasks	31 - 71
II Refactoring Tool Usability	73 - 88
III Usability Profiles of Refactoring Tools	91 - 114
IV Stepwise Refactoring Tools	115 - 140

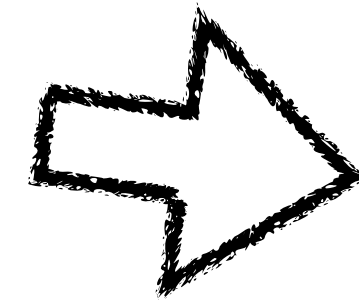
Improving the Usability of Refactoring Tools for Software Change Tasks

Paper	Pages
I A Study of Refactorings During Software Change Tasks	31 - 71
II Refactoring Tool Usability	73 - 88
III Usability Profiles of Refactoring Tools	91 - 114
IV Stepwise Refactoring Tools	115 - 140

Navajo

78K lines (Java version 1.8)

1335 JUnit tests



**Move
Method**



1

**Inline
Method**

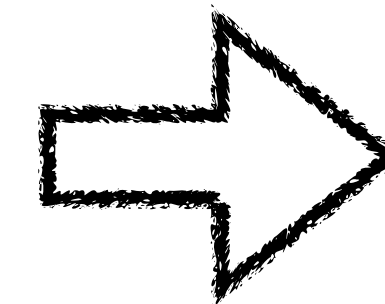


2

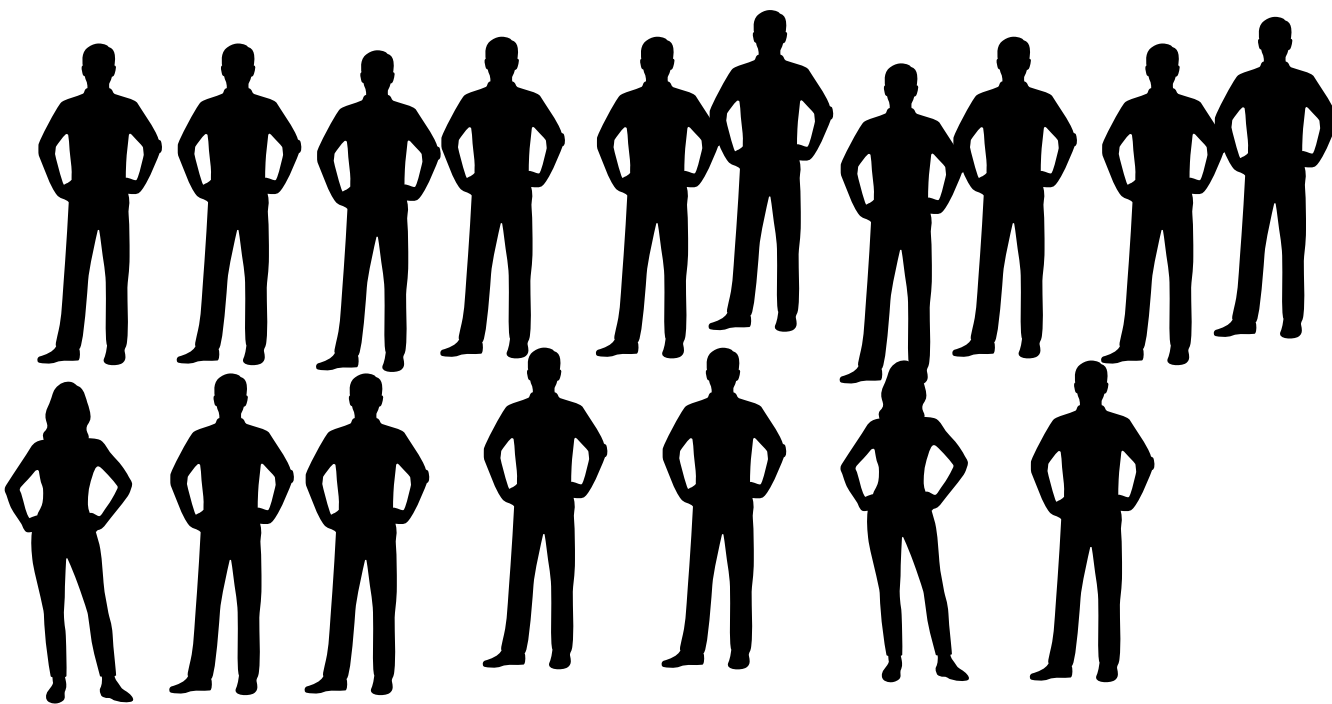
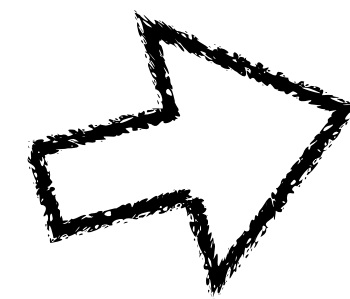
**Remove
Parameter**



3

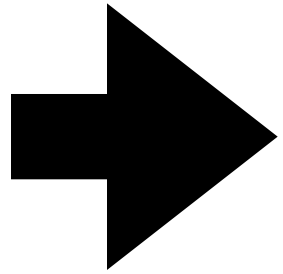


- 32 hours recording
- 17 transcripts
- 134,947 words
- 100 refactoring tool invocations
- 385 detected refactorings (manual&tool)



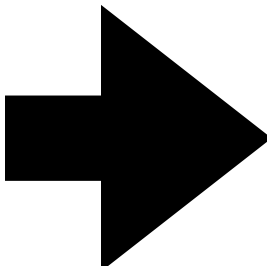
Intent	"My long-term goal is that I want to inline this method. So by keeping this kind of stub implementation of it - stub isn't the right word - I'm preserving it as long as possible so any tests calling it [are] still working while I do my refactoring. What I want to do is move the implementation into the method I am going to keep and ensure that the tests are all passing and then remove the method I want to remove."
Approach Summary	Forms plan upfront that includes Inline Method. Changes plan when encountering tool error. Isolate changes and keeps tests running by introducing a shared dependency as an intermediate step. Manual inline, keep old code in method to validate (compare). Validates change by running tests and comparing changes in git. Guides steps by Find Usages.
Refactoring Tools	
Mentioned:	Inline Method
Mentioned prompted:	
Tried:	Inline Method
Tried prompted:	
Used to solve task:	None
Other Tools:	Find Usages
Guide changes:	Find Usages, Avoid compiler errors
Verify:	Dynamic after each change, git diff

Figure 3.1: Task analysis schema for P_9 on Task-2.



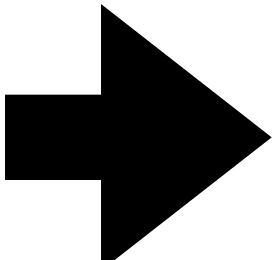
Intent	"My long-term goal is that I want to inline this method. So by keeping this kind of stub implementation of it - stub isn't the right word - I'm preserving it as long as possible so any tests calling it [are] still working while I do my refactoring. What I want to do is move the implementation into the method I am going to keep and ensure that the tests are all passing and then remove the method I want to remove."
Approach Summary	Forms plan upfront that includes Inline Method. Changes plan when encountering tool error. Isolate changes and keeps tests running by introducing a shared dependency as an intermediate step. Manual inline, keep old code in method to validate (compare). Validates change by running tests and comparing changes in git. Guides steps by Find Usages.
Refactoring Tools	
Mentioned:	Inline Method
Mentioned prompted:	
Tried:	Inline Method
Tried prompted:	
Used to solve task:	None
Other Tools:	Find Usages
Guide changes:	Find Usages, Avoid compiler errors
Verify:	Dynamic after each change, git diff

Figure 3.1: Task analysis schema for P_9 on Task-2.



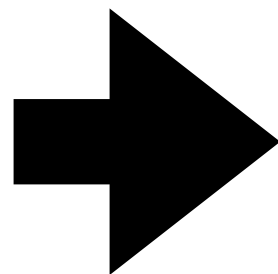
Intent	"My long-term goal is that I want to inline this method. So by keeping this kind of stub implementation of it - stub isn't the right word - I'm preserving it as long as possible so any tests calling it [are] still working while I do my refactoring. What I want to do is move the implementation into the method I am going to keep and ensure that the tests are all passing and then remove the method I want to remove."
Approach Summary	Forms plan upfront that includes Inline Method. Changes plan when encountering tool error. Isolate changes and keeps tests running by introducing a shared dependency as an intermediate step. Manual inline, keep old code in method to validate (compare). Validates change by running tests and comparing changes in git. Guides steps by Find Usages.
Refactoring Tools	
Mentioned:	Inline Method
Mentioned prompted:	
Tried:	Inline Method
Tried prompted:	
Used to solve task:	None
Other Tools:	Find Usages
Guide changes:	Find Usages, Avoid compiler errors
Verify:	Dynamic after each change, git diff

Figure 3.1: Task analysis schema for P_9 on Task-2.



Intent	"My long-term goal is that I want to inline this method. So by keeping this kind of stub implementation of it - stub isn't the right word - I'm preserving it as long as possible so any tests calling it [are] still working while I do my refactoring. What I want to do is move the implementation into the method I am going to keep and ensure that the tests are all passing and then remove the method I want to remove."
Approach Summary	Forms plan upfront that includes Inline Method. Changes plan when encountering tool error. Isolate changes and keeps tests running by introducing a shared dependency as an intermediate step. Manual inline, keep old code in method to validate (compare). Validates change by running tests and comparing changes in git. Guides steps by Find Usages.
Refactoring Tools	
Mentioned:	Inline Method
Mentioned prompted:	
Tried:	Inline Method
Tried prompted:	
Used to solve task:	None
Other Tools:	Find Usages
Guide changes:	Find Usages, Avoid compiler errors
Verify:	Dynamic after each change, git diff

Figure 3.1: Task analysis schema for P_9 on Task-2.

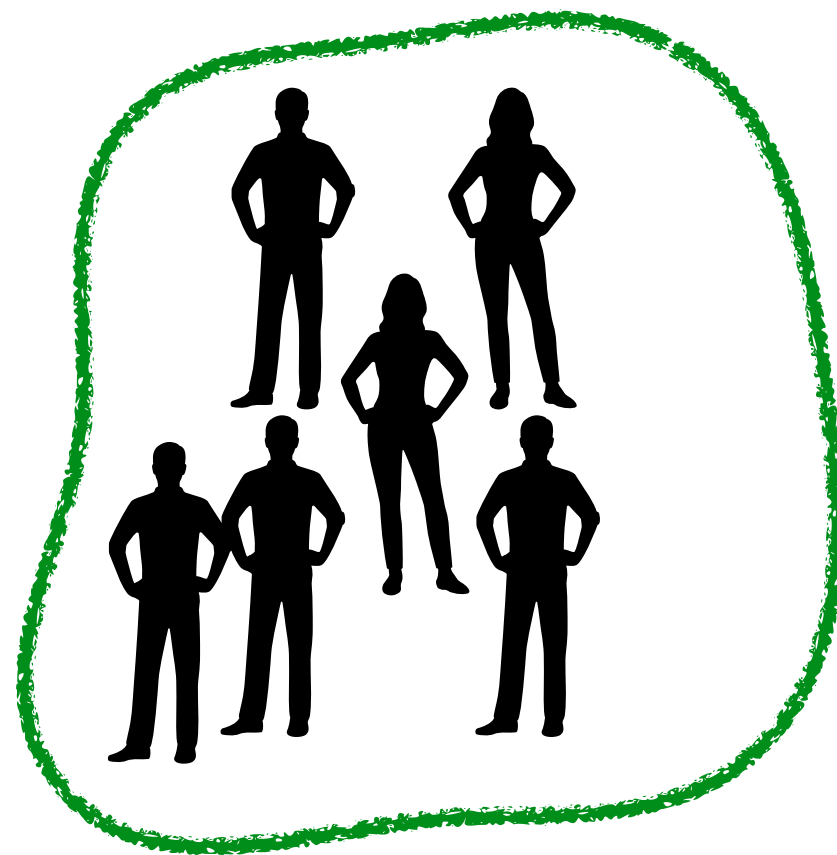


Intent	"My long-term goal is that I want to inline this method. So by keeping this kind of stub implementation of it - stub isn't the right word - I'm preserving it as long as possible so any tests calling it [are] still working while I do my refactoring. What I want to do is move the implementation into the method I am going to keep and ensure that the tests are all passing and then remove the method I want to remove."
Approach Summary	Forms plan upfront that includes Inline Method. Changes plan when encountering tool error. Isolate changes and keeps tests running by introducing a shared dependency as an intermediate step. Manual inline, keep old code in method to validate (compare). Validates change by running tests and comparing changes in git. Guides steps by Find Usages.
Refactoring Tools	
Mentioned:	Inline Method
Mentioned prompted:	
Tried:	Inline Method
Tried prompted:	
Used to solve task:	None
Other Tools:	Find Usages
Guide changes:	Find Usages, Avoid compiler errors
Verify:	Dynamic after each change, git diff

Figure 3.1: Task analysis schema for P_9 on Task-2.

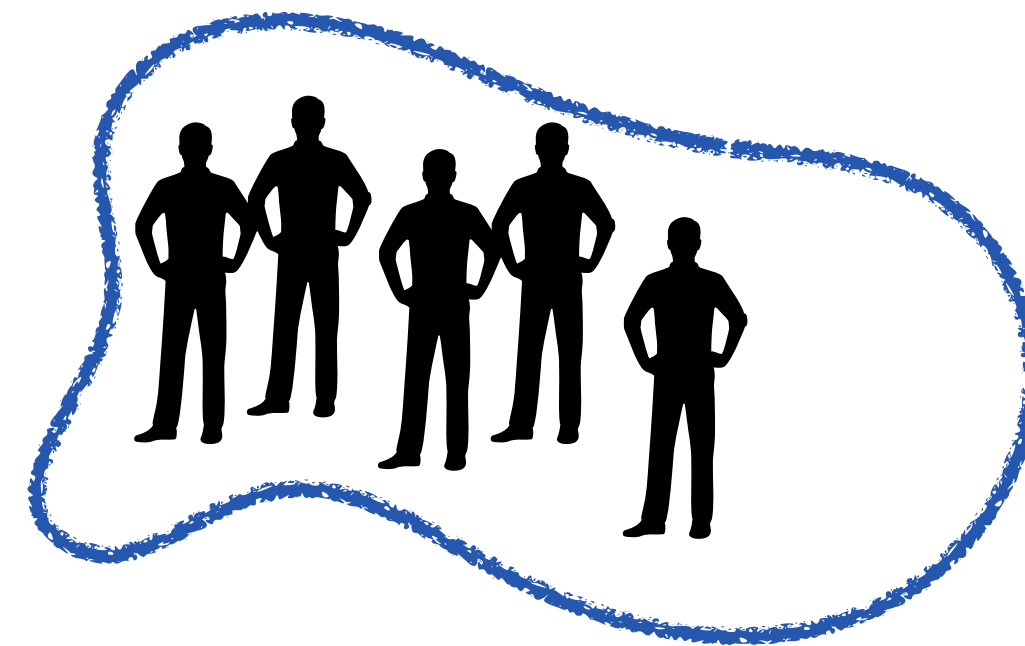
Local

“So what I’m going to do now is rely on my friendly *compiler* to do this. So now things are going to *break*”



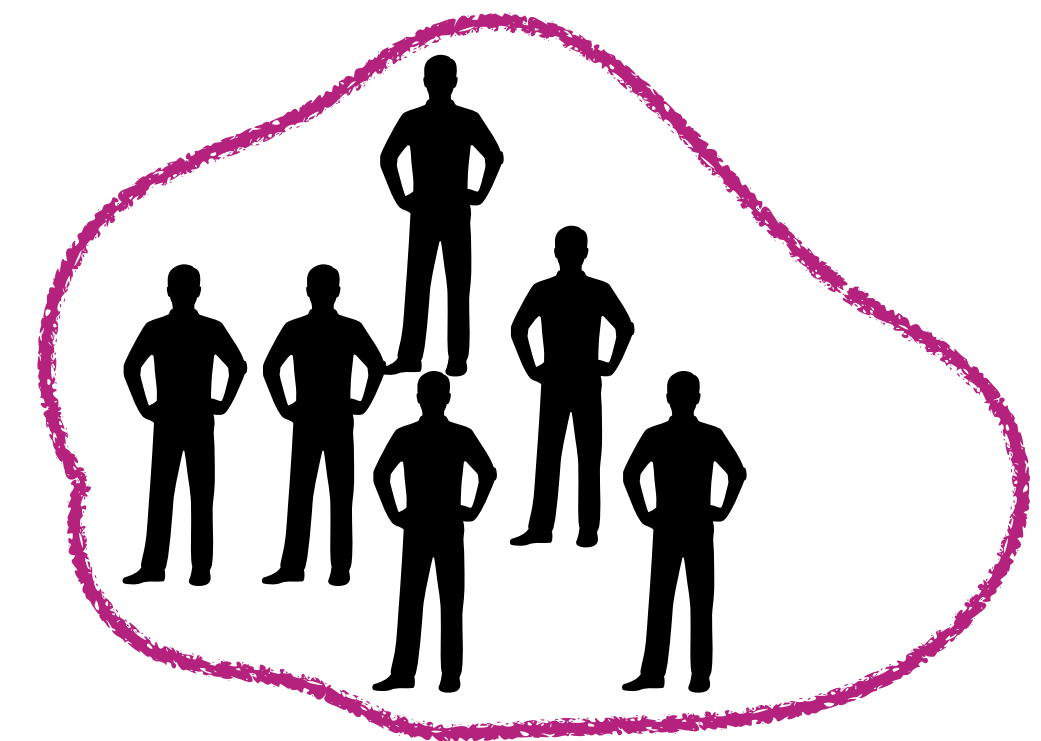
Structure

“Any *invocations* would need to be dealt with. So what I’d like to do, is look in StringUtils, look *where these are used* (...)”

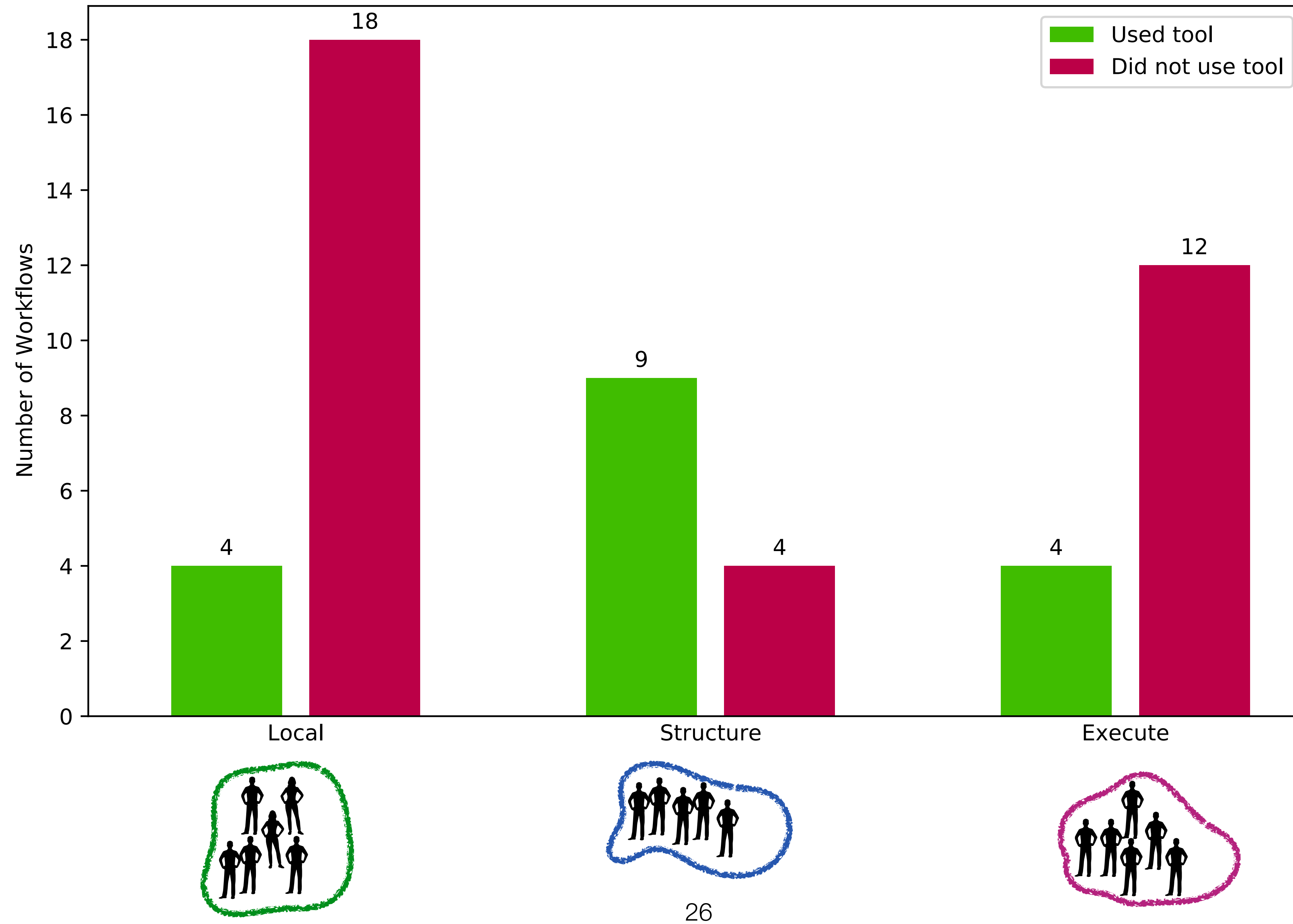


Execute

“(...) change the internal definition of the methods first so that it always *executes* as if it was false. That will reveal the *test cases* that I need to look at more closely”.



Refactoring tool use



- An experimental system that can be reused by other researchers to study realistic software change tasks that contain steps amenable to refactoring tools.
- A dataset comprising 32 hours of screen recordings from 17 experienced practitioners with time stamped transcripts totaling 130k words and 100 refactoring invocations.
- Three strategies that were observed to be used by developers to make progress in functional software change tasks: Local, Structure, and Execute.

Improving the Usability of Refactoring Tools for Software Change Tasks

Paper	Pages
I A Study of Refactorings During Software Change Tasks	31 - 71
II Refactoring Tool Usability	73 - 88
III Usability Profiles of Refactoring Tools	91 - 114
IV Stepwise Refactoring Tools	115 - 140

"extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use"

(ISO 9241-11:2018, Part 11, 3.1.1)

Software developers employ refactoring tools to help prepare or complete functional changes to a software system. Software developers seek these tools to be reasonable to locate and apply for a desired intent (effective), reasonable in terms of time and effort required to use the tool (efficient), and to add value to the development process (satisfying).

Codebook	
Label	Pattern
Effective	User experiences the tool as successfully performing a desired result.
Efficient	User experiences the tool as being fast or increasing productivity.
Satisfactory	User experiences having their needs or wants met by the use of the tool.
Trust	User experiences the tool as trustworthy, safe or reliable.
Predictable	User understands up-front what will happen by using the tool or indicates a lack of surprise when using the tool.



I: So the refactoring did break the code but it was what you expected nonetheless.

Yes it was more helpful for the refactoring to result in code that did not compile than to say I can not do that.

I: how come?

It was more in line with the change I would have done manually. I would make the change, see that there was an in-collision and then resolve the collision. Asking me to modify unrelated code in order to delete the unrelated code feels like an extra step.

- P9, T3



I: So the refactoring did break the code but it was what you expected nonetheless.

Yes it was more helpful for the refactoring to result in code that did not compile than to say I can not do that.

Predictable

I: how come?

It was more in line with the change I would have done manually. I would make the change, see that there was an in-collision and then resolve the collision. Asking me to modify unrelated code in order to delete the unrelated code feels like an extra step.

- P9, T3



I: So the refactoring did break the code but it was what you expected nonetheless.

Yes it was more helpful for the refactoring to result in code that did not compile than to say I can not do that.

Predictable
Satisfaction

I: how come?

It was more in line with the change I would have done manually. I would make the change, see that there was an in-collision and then resolve the collision. Asking me to modify unrelated code in order to delete the unrelated code feels like an extra step.

- P9, T3



I: So the refactoring did break the code but it was what you expected nonetheless.

Yes it was more helpful for the refactoring to result in code that did not compile than to say I can not do that.

Predictable Satisfaction

I: how come?

It was more in line with the change I would have done manually. I would make the change, see that there was an in-collision and then resolve the collision. Asking me to modify unrelated code in order to delete the unrelated code feels like an extra step.

Efficient (“Asking me to modify unrelated code in order to delete the unrelated code feels like an extra step.”)

- P9, T3



256 Cards

Predictable

I: So the refactoring did break the code but it was what you expected nonetheless.

Yes it was more helpful for the refactoring to result in code that did not compile than to say I can not do that.

Satisfaction

I: So the refactoring did break the code but it was what you expected nonetheless.

Yes it was more helpful for the refactoring to result in code that did not compile than to say I can not do that.

Efficient

I: So the refactoring did break the code
Yes it was more helpful for the refactoring to result in code that did not compile than to say I
I: how come?
It was more in line with the change I would have done manually. I would make the change,

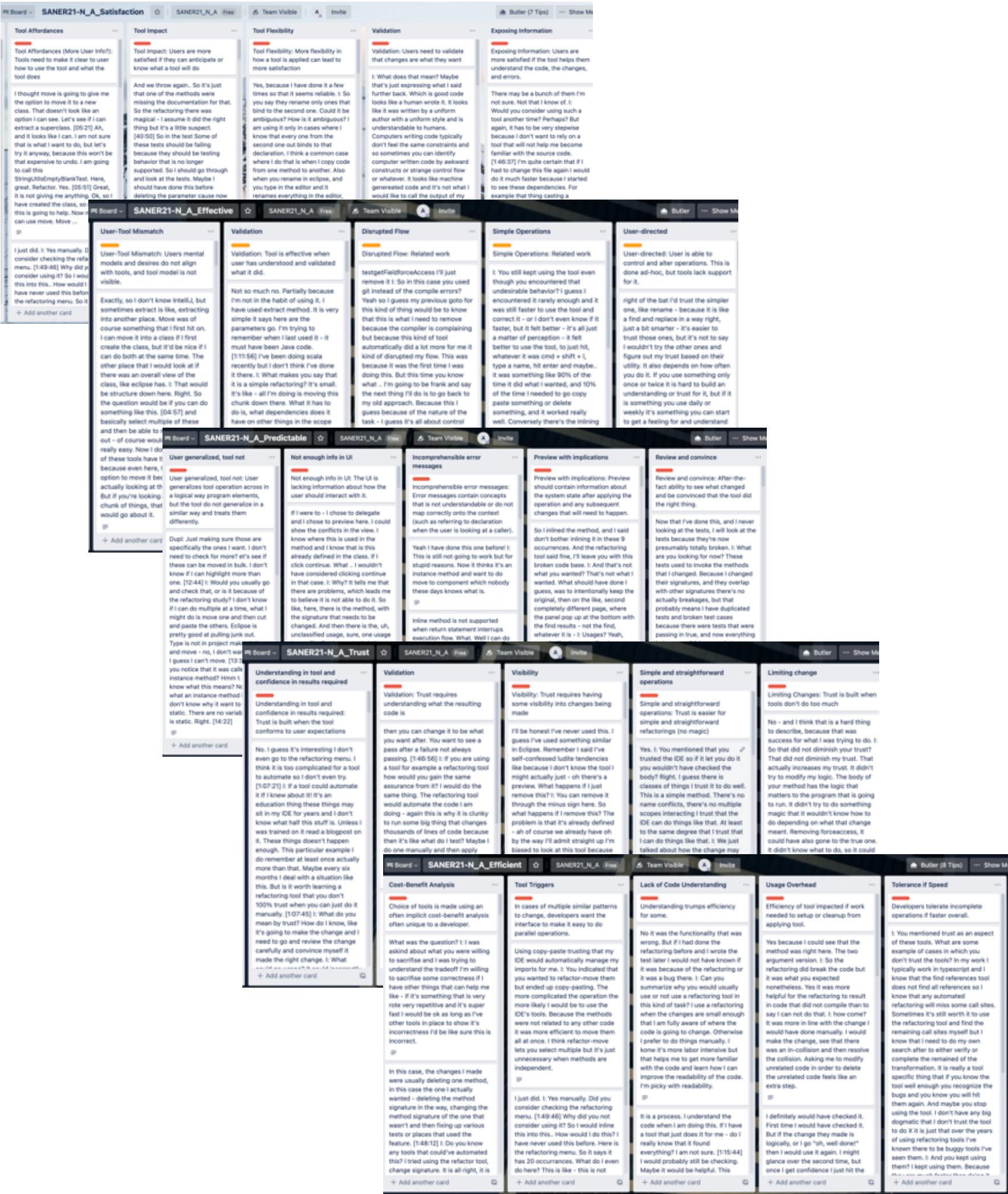
Paper II

Card sort

Cards: 256

Themes: 28

Predictable	79
Effective	53
Satisfaction	51
Efficient	42
Trust	31



The screenshot displays a Meta Card Sort interface with five columns of cards. The first column, titled 'Start', contains five cards: 'Trust' (green bar), 'Effective' (orange bar), 'Efficient' (purple bar), 'Satisfaction' (blue bar), and 'Predictable' (red bar). The other four columns are labeled with their respective card counts: 'User Cost-Benefit Analysis (44)', 'User Control of Operation (26)', 'Tool Communicating The Change (77)', and 'Tool Communicating Its Capabilities (17)'. Each column contains several cards with titles and descriptions, and a color-coded progress bar at the top. The interface includes a top navigation bar with 'SANER21_META' and 'SANER21_N_A' tabs, and a bottom bar with 'Add another card' buttons.

Start

- Trust
- Effective
- Efficient
- Satisfaction
- Predictable

User Cost-Benefit Analysis (44)

- (21) Cost-Benefit Analysis: Choice of tools is made using an often implicit cost-benefit analysis often unique to a developer.
- (4) Usage Overhead: Efficiency of tool impacted if work needed to setup or cleanup from applying tool.
- (1) Tolerance if Speed: Developers tolerate incomplete operations if faster overall.
- (1) Cost of Use: Up-front warning about the effort required.
- (1) No Alternatives: The tool offers an error message or problem, potentially together with a single alternative. The user wants to know what they can do to proceed.
- (16) Preview with Implications: Preview should contain information

User Control of Operation (26)

- (2) Tool Triggers: In cases of multiple similar patterns to change, developers want the interface to make it easy to do parallel operations.
- (8) User-directed: User is able to control and alter operations. This is done ad-hoc, but tools lack support for it.
- (1) Limiting Changes: Trust is built when tools don't do too much
- (2) User generalized, tool not: User generalizes tool operation across in a logical way program elements, but the tool do not generalize in a similar way and treats them differently.
- (11) Adjust Changes: Step through view of the change, including opportunities to view and adjust the changes in the context they happen.

Tool Communicating The Change (77)

- (12) User-Tool Mismatch: Users mental models and desires do not align with tools, and tool model is not visible.
- (2) Simple Operations: Related work
- (7) Validation: Tool is effective when user has understood and validated what it did.
- (16) Review and convince: After-the-fact ability to see what changed and be convinced that the tool did the right thing.
- (12) Understanding in tool and confidence in results required: Trust is built when the tool conforms to user expectations
- (2) Lack of Code Understanding: Understanding trumps efficiency for some.

Tool Communicating Its Capabilities (17)

- (9) Not enough info in UI: The UI is lacking information about how the user should interact with it.
- (6) Incomprehensible error messages: Error messages contain concepts that is not understandable or do not map correctly onto the context (such as referring to declaration when the user is looking at a caller).
- (6) Tool Affordances (More User info?): Tools need to make it clear to user how to use the tool and what the tool does

Theme	Description	P R D	E F E	S A T	E F I	T R S
Tool communicates capabilities	A tool’s user interface must communicate clearly and directly to a developer in terms familiar to a developer. The tool should guide a developer in its use, including providing intelligible error messages.	✓		✓		
Tool communicates change	A tool should make the changes it will make to code clear before the tool is executed. It should be possible for a developer to inspect the changes that the tool has made.	✓	✓	✓		✓
Developer guides tool	Developers wish to guide how a tool executes in several ways: applying an operation to many elements easily, excluding the application to some locations of code, and altering how particular code is changed.	✓	✓	✓	✓	✓
Developer cost-benefit analysis	Developers assess the cost of applying a tool before they invoke the tool. Developers want to assess whether it is better to proceed with a tool or manually at the start of considering using a tool. The tool should help them with this assessment.	✓	✓		✓	

Theme	Description	P R D	E F E	S A T	E F I	T R S
Tool communicates capabilities	A tool's user interface must communicate clearly and directly to a developer in terms familiar to a developer. The tool should guide a developer in its use, including providing intelligible error messages.	✓		✓		
Tool communicates change	A tool should make the changes it will make to code clear before the tool is executed. It should be possible for a developer to inspect the changes that the tool has made.	✓	✓	✓		✓
Developer guides tool	Developers wish to guide how a tool executes in several ways: applying an operation to many elements easily, excluding the application to some locations of code, and altering how particular code is changed.	✓	✓	✓	✓	✓
Developer cost-benefit analysis	Developers assess the cost of applying a tool before they invoke the tool. Developers want to assess whether it is better to proceed with a tool or manually at the start of considering using a tool. The tool should help them with this assessment.	✓	✓		✓	

Software developers employ refactoring tools to help prepare or complete functional changes to a software system.

→ Software developers seek these tools to be reasonable to locate, and for the tools to help them assess the efficiency of the tool, in terms of the costs and benefits of the tool, before its use.

→ To enable effective use in multiple situations, software developers seek to guide how a tool changes the source code for a system; this ability to tailor how a tool works can improve the efficiency of the tool for the developer.

→ Software developers also seek refactoring tools to explain their impact to source code so that the software developer can understand the effectiveness of the tool.

→ Software developers also expect tools to communicate clearly and directly in terms that match how software developers perceive refactoring operations.

These characteristics in a refactoring tool increase the satisfaction of the software developer using a refactoring tool.

- A theory of usability for refactoring tools.
- Four usability themes that illustrate problem areas of refactoring tools in the context of software change tasks.

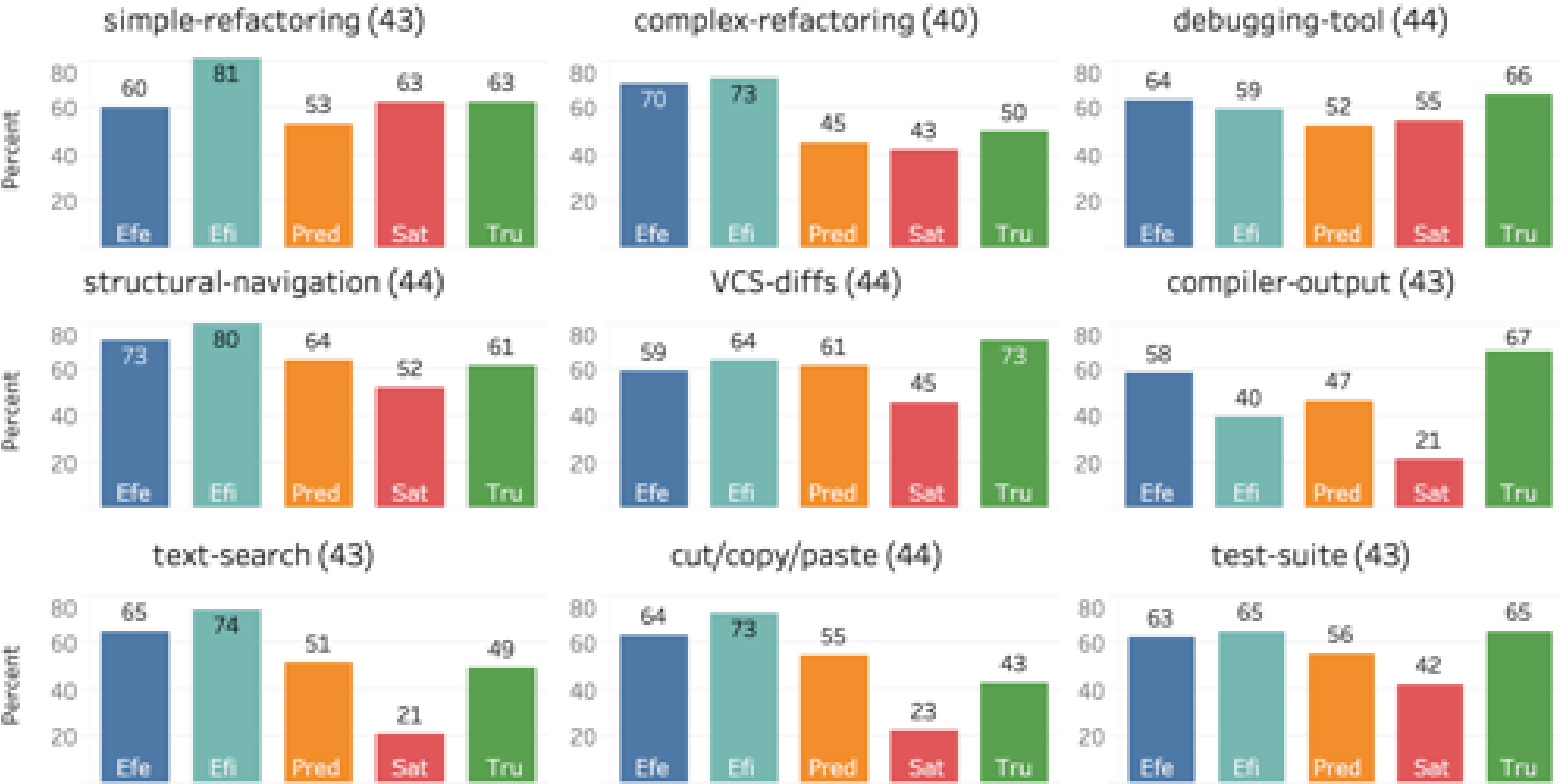
Improving the Usability of Refactoring Tools for Software Change Tasks

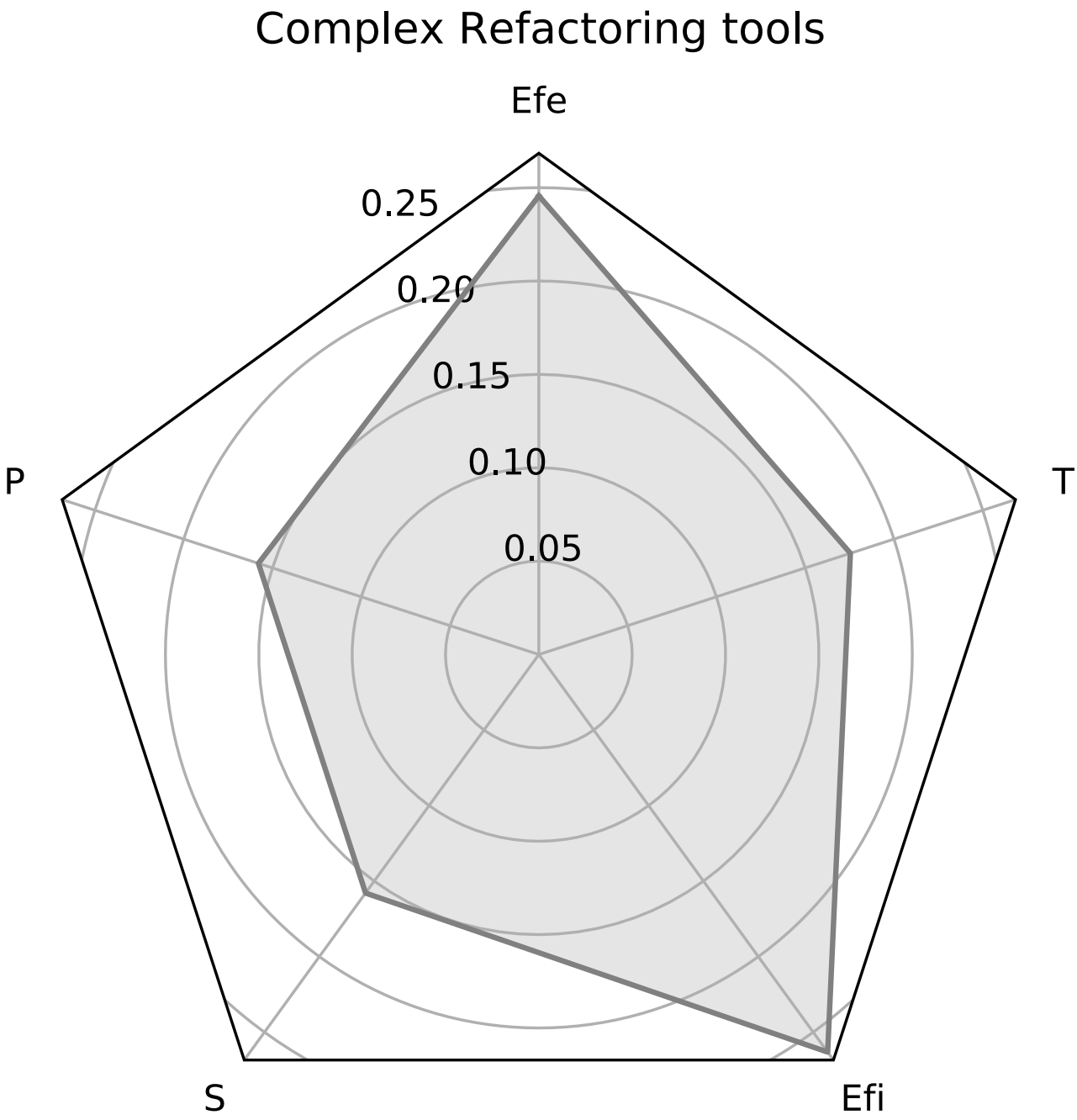
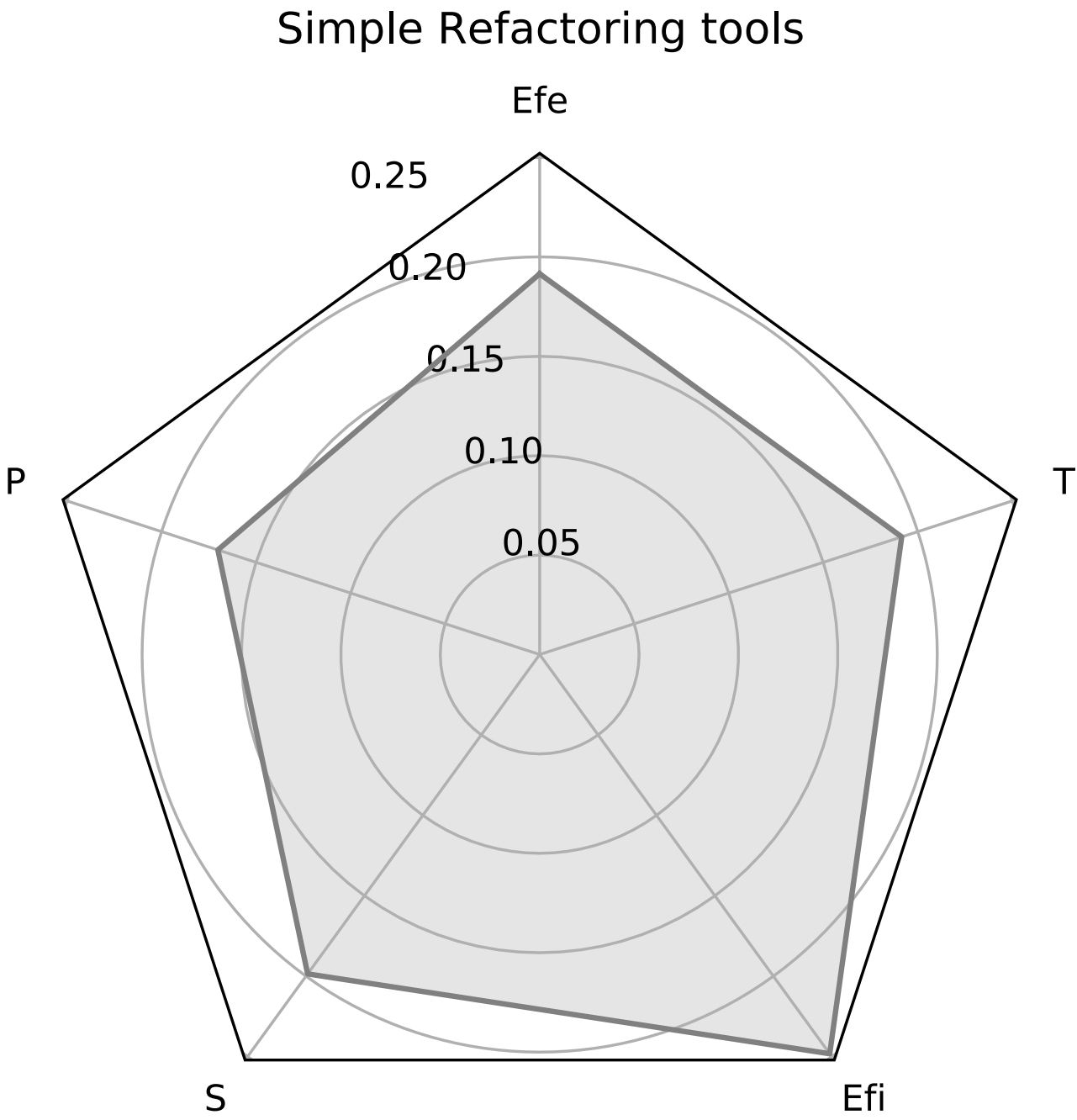
Paper	Pages
I A Study of Refactorings During Software Change Tasks	31 - 71
II Refactoring Tool Usability	73 - 88
III Usability Profiles of Refactoring Tools	91 - 114
IV Stepwise Refactoring Tools	115 - 140

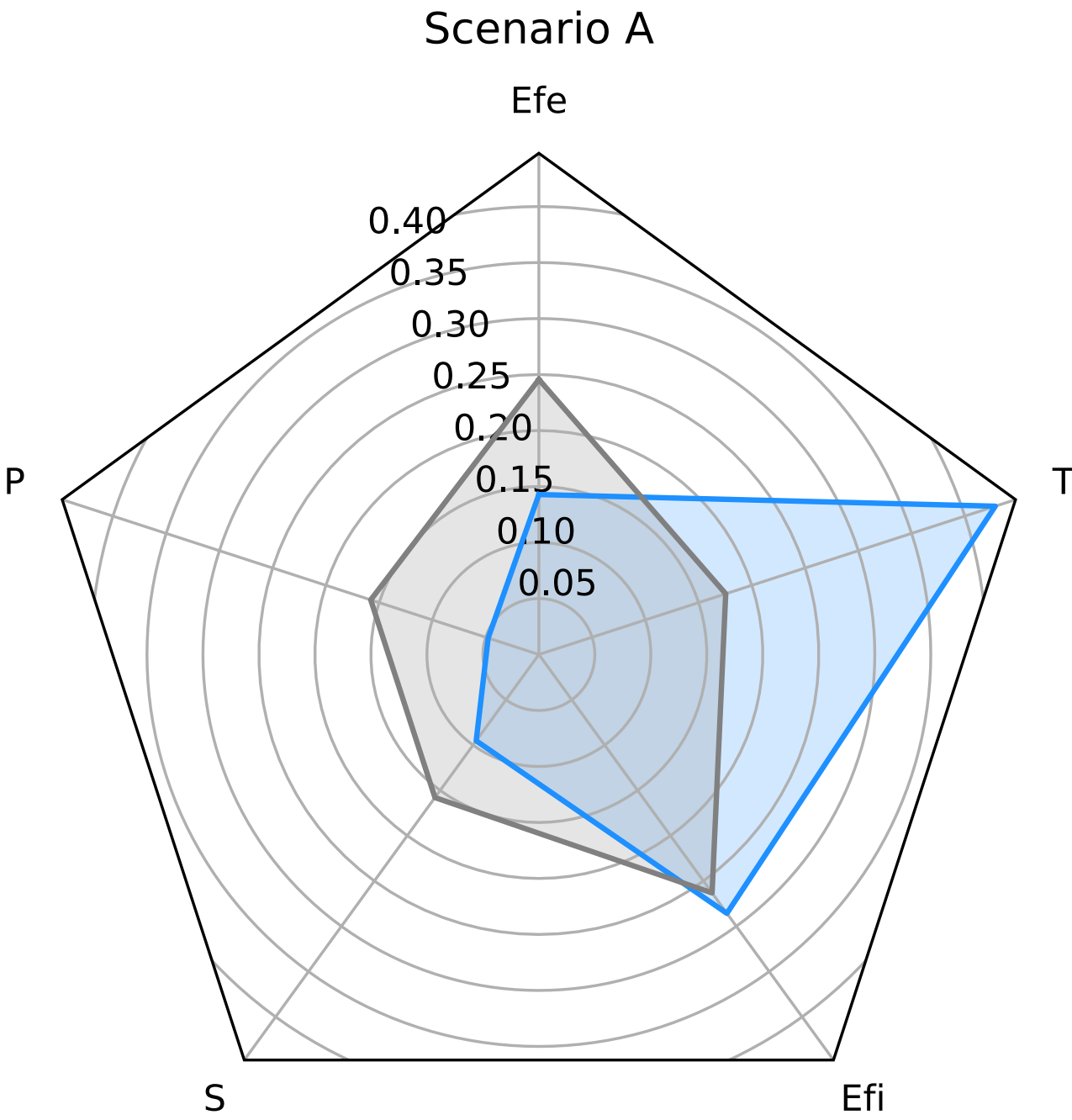
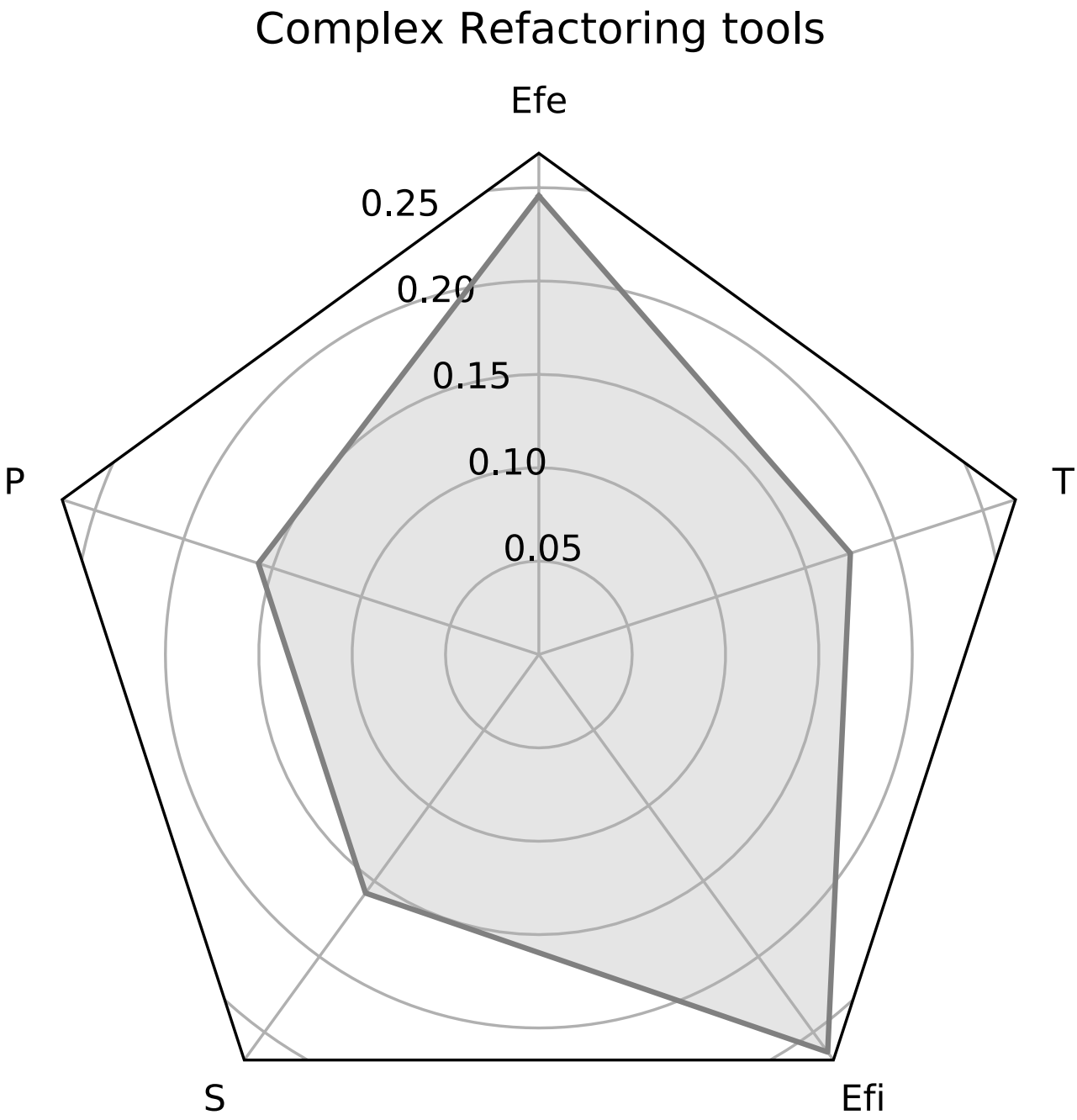
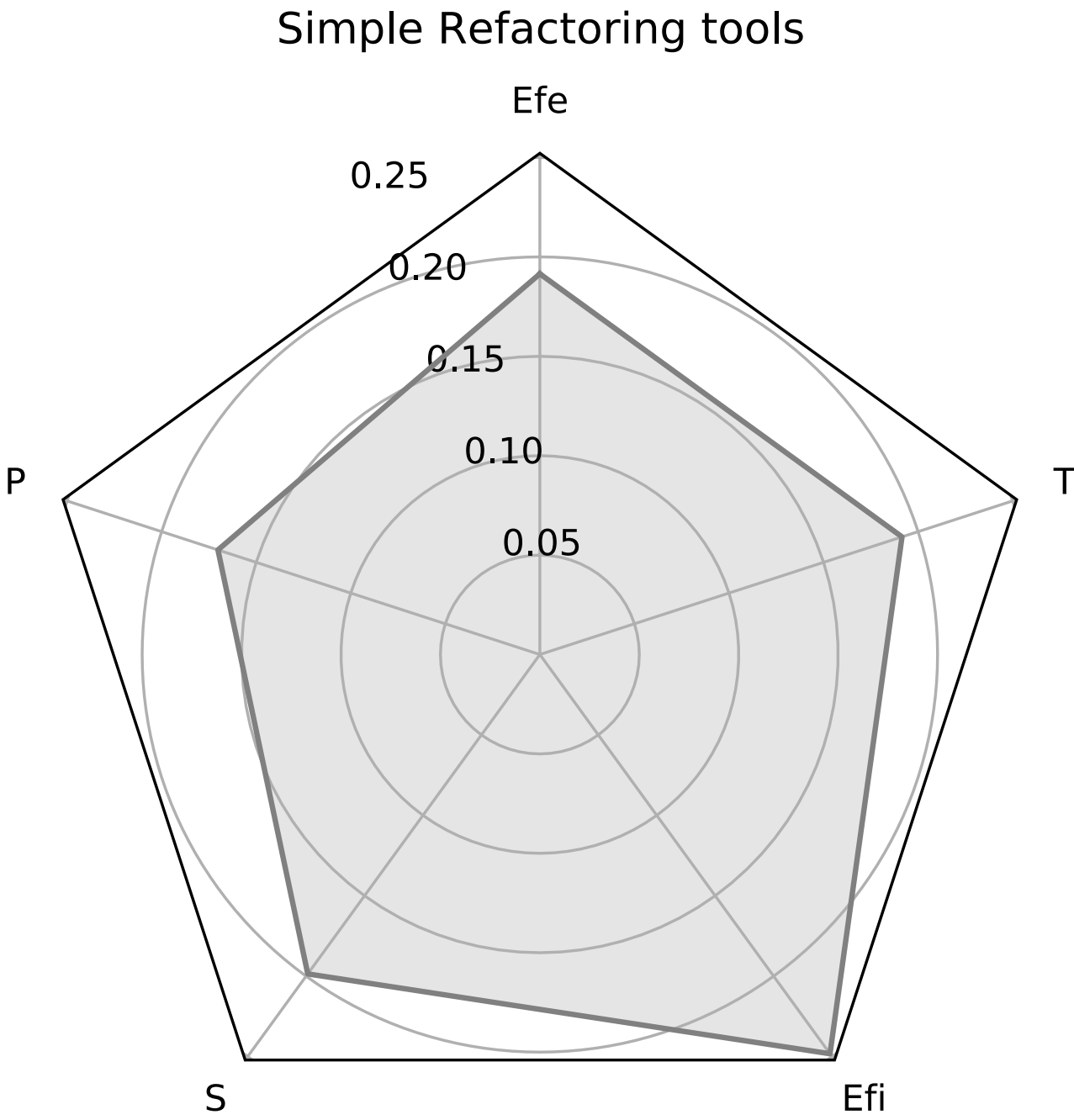
Paper III

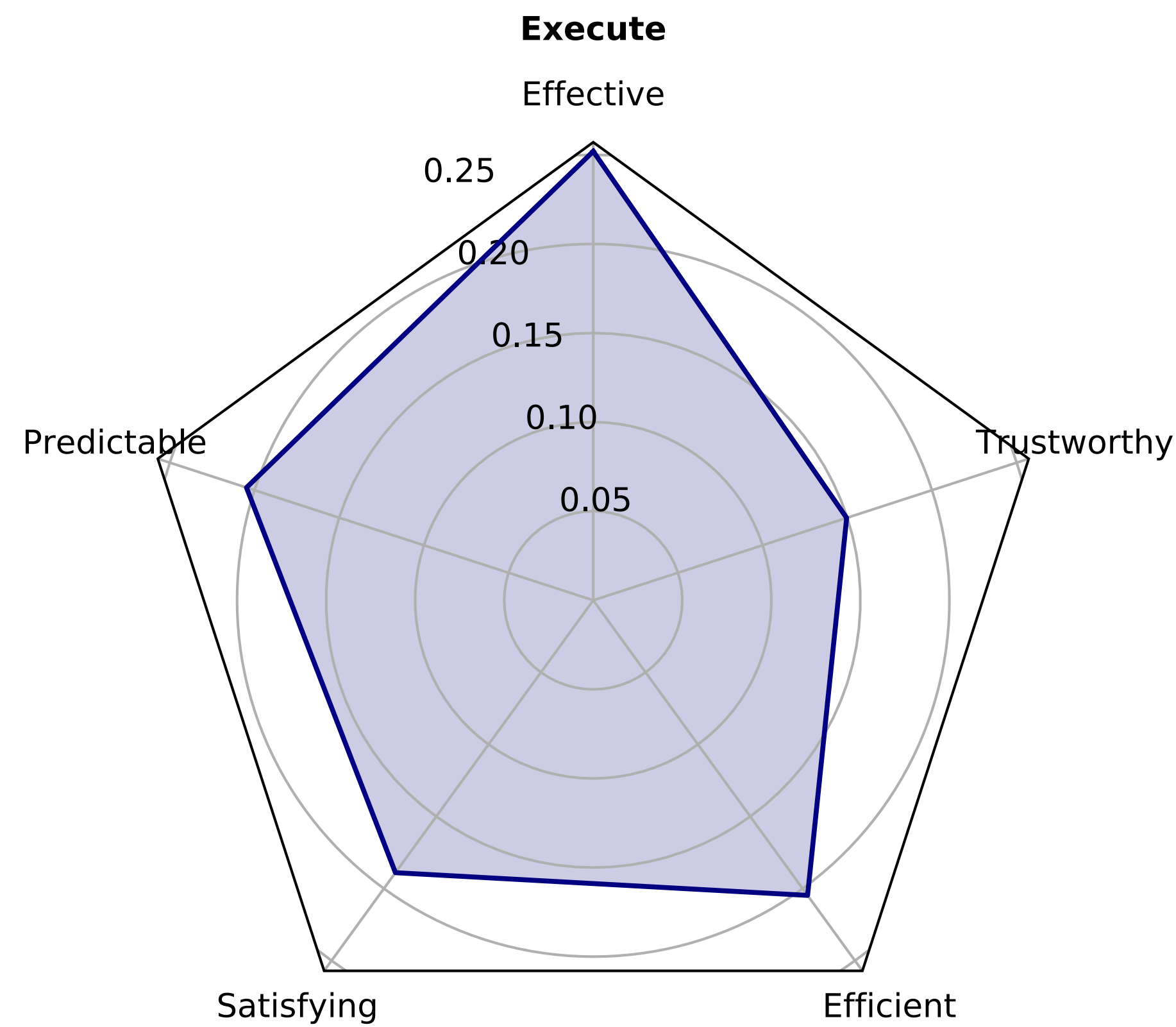
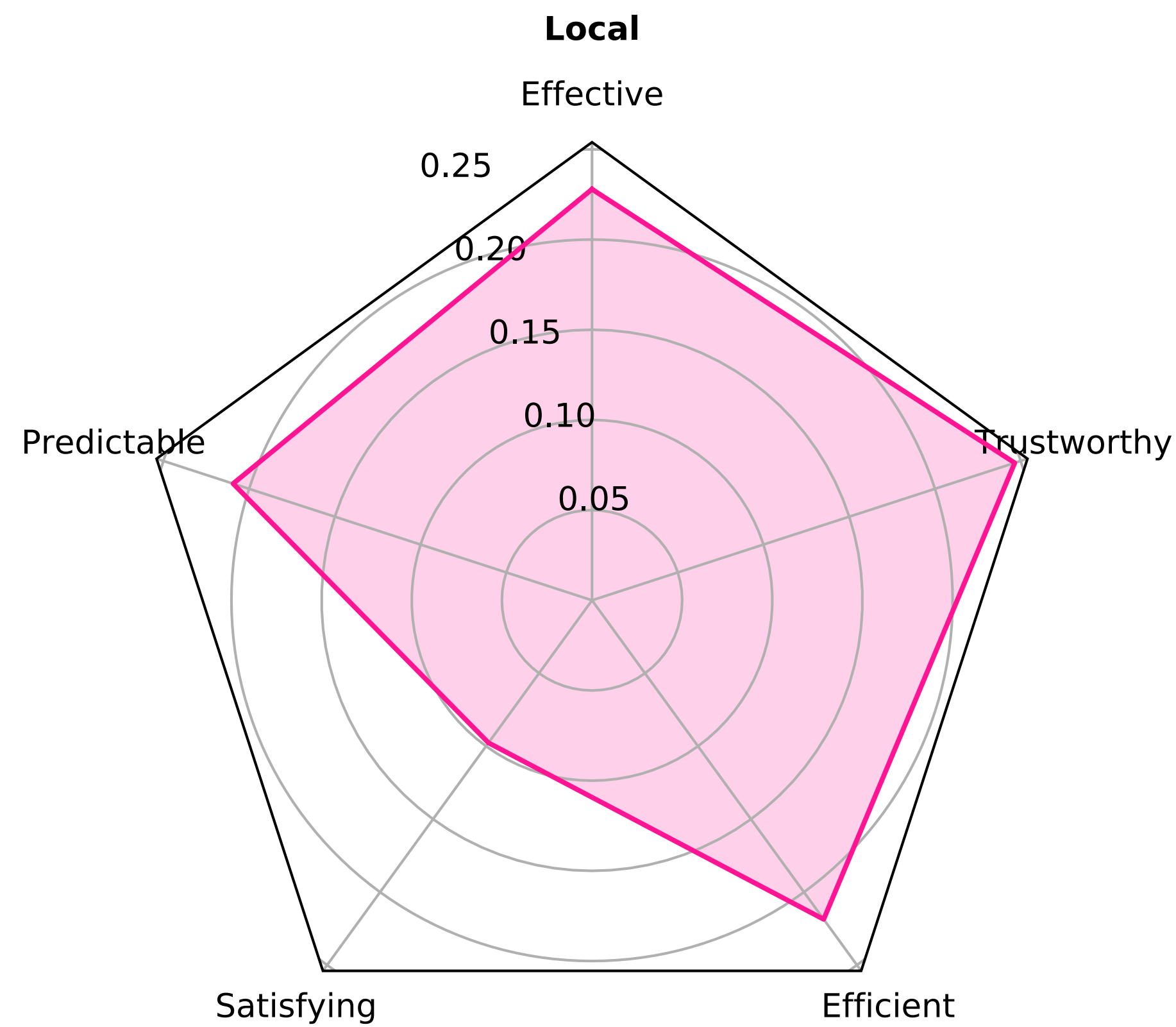
Usability factors considered necessary

Tools		The tool is effective	The tool saves time or effort	The tool is satisfying to use	The tool is trustworthy	The tool is predictable
	Complex Refactoring tools (e.g. Move Method, Extract Class, Introduce Parameter)	☐	☐	☐	☐	☐
	Compiler output (e.g. compiler errors)	☐	☐	☐	☐	☐
	Version Control Systems "diffs" (e.g. git diff, et.c.)	☐	☐	☐	☐	☐
	Structural navigation (e.g. find references, go to declaration)	☐	☐	☐	☐	☐
	Debugging tools	☐	☐	☐	☐	☐
	Copy/cut and paste code	☐	☐	☐	☐	☐
	Test suites (e.g. JUnit, test errors)	☐	☐	☐	☐	☐
	Simple Refactoring tools (e.g. Rename, Extract Constant, Inline Method)	☐	☐	☐	☐	☐
	Textual search (e.g. grep, find and replace)	☐	☐	☐	☐	☐





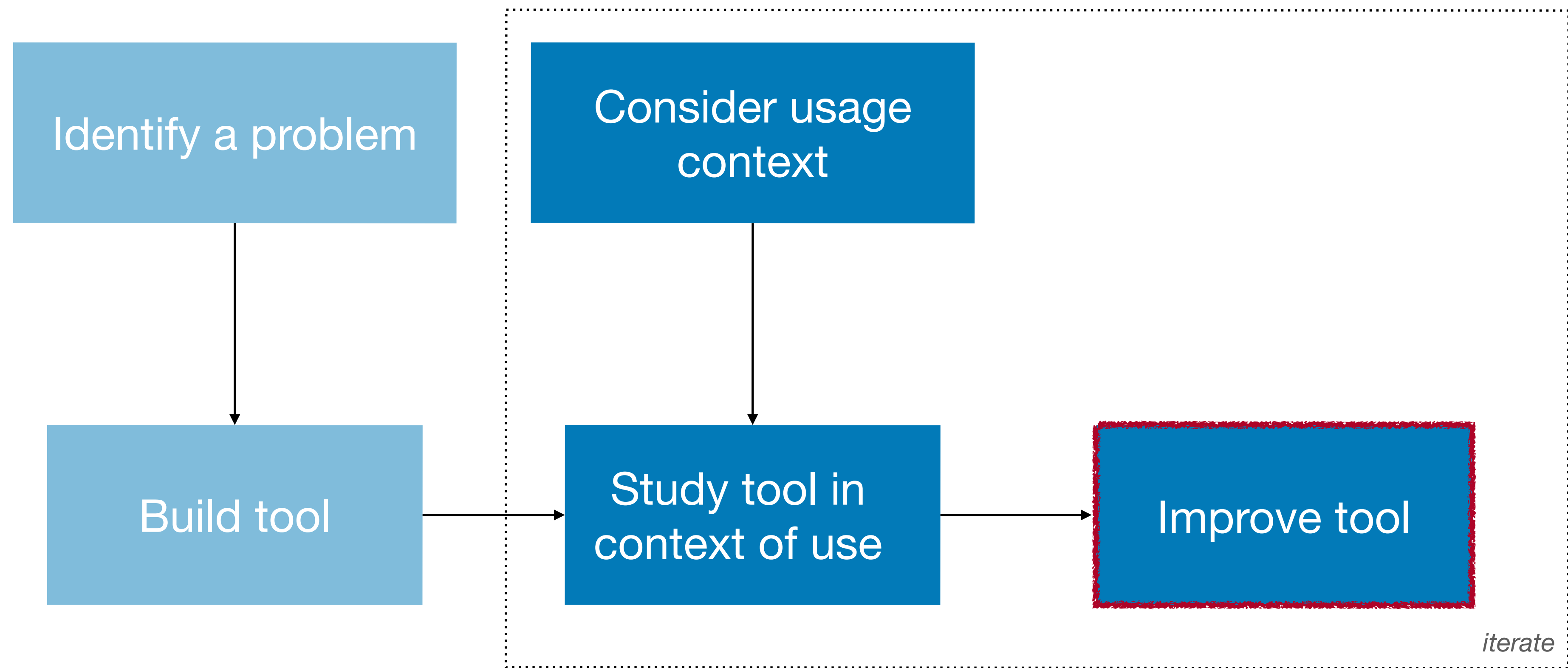


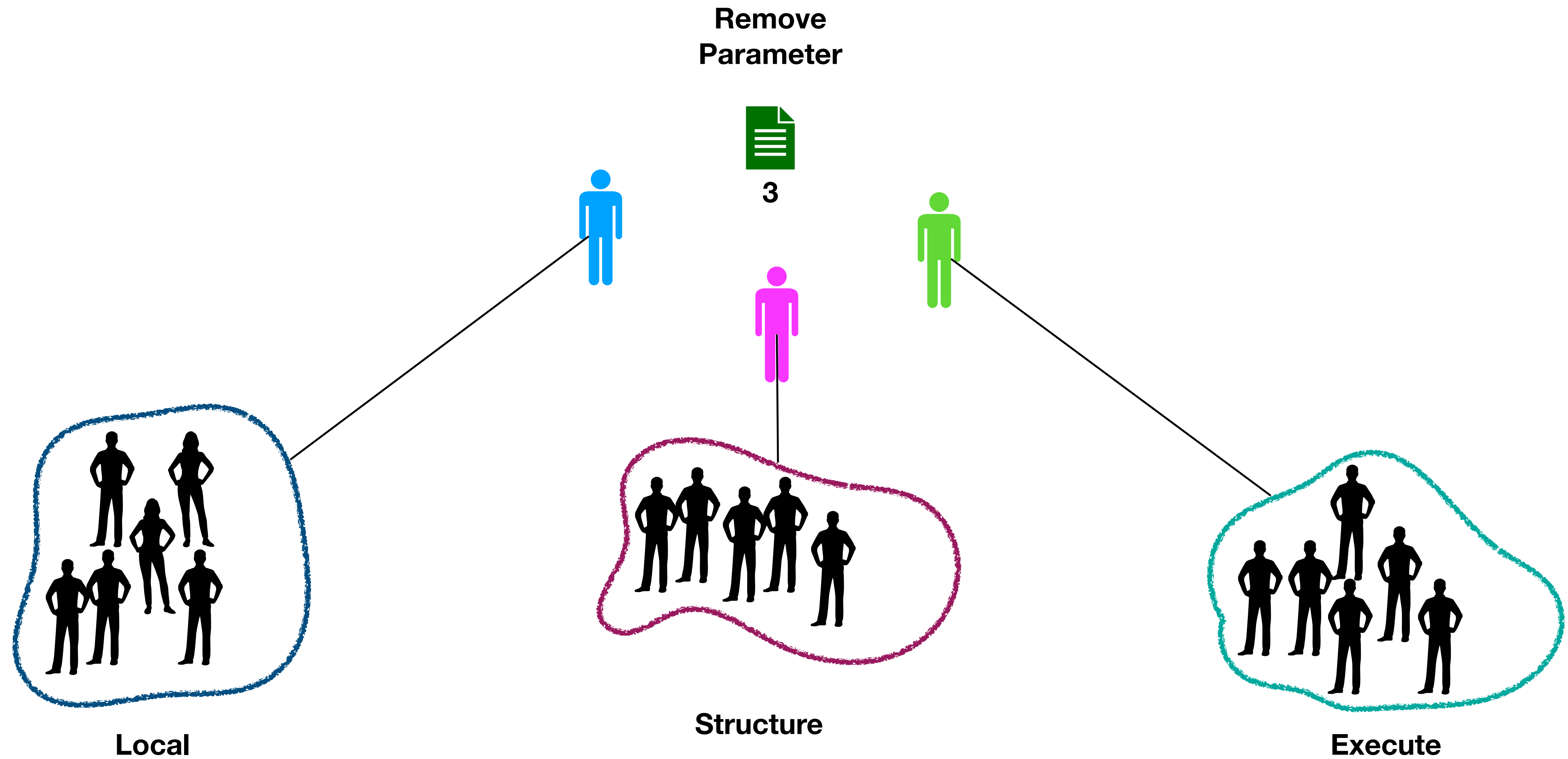


- Usability profiles of nine types of tools that developers find useful for approaching software change tasks, including simple and complex refactoring tools.

Improving the Usability of Refactoring Tools for Software Change Tasks

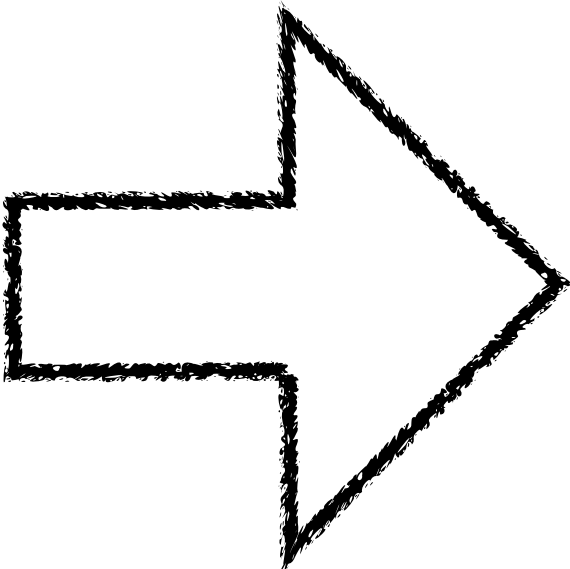
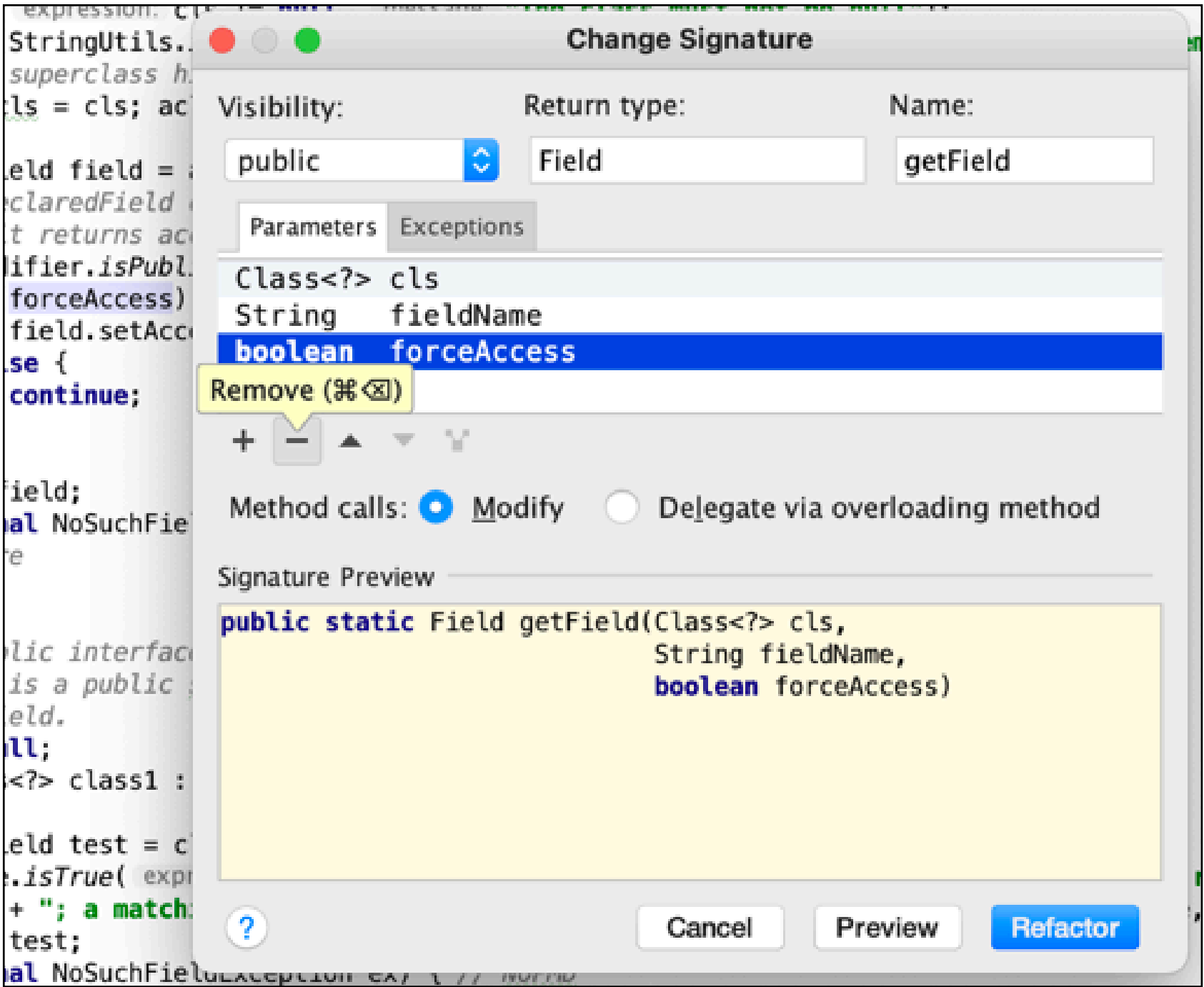
Paper	Pages
I A Study of Refactorings During Software Change Tasks	31 - 71
II Refactoring Tool Usability	73 - 88
III Usability Profiles of Refactoring Tools	91 - 114
IV Stepwise Refactoring Tools	115 - 140



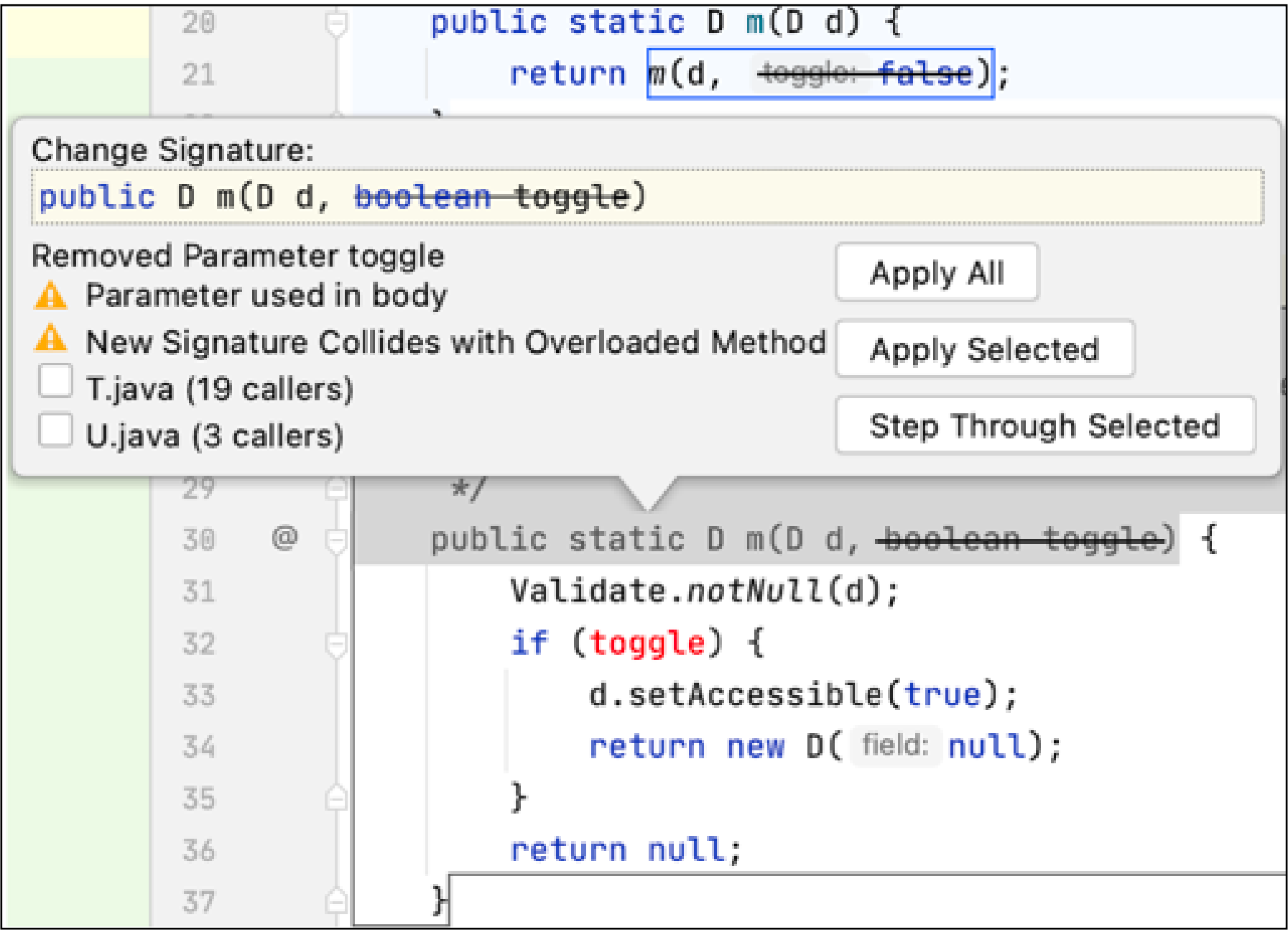




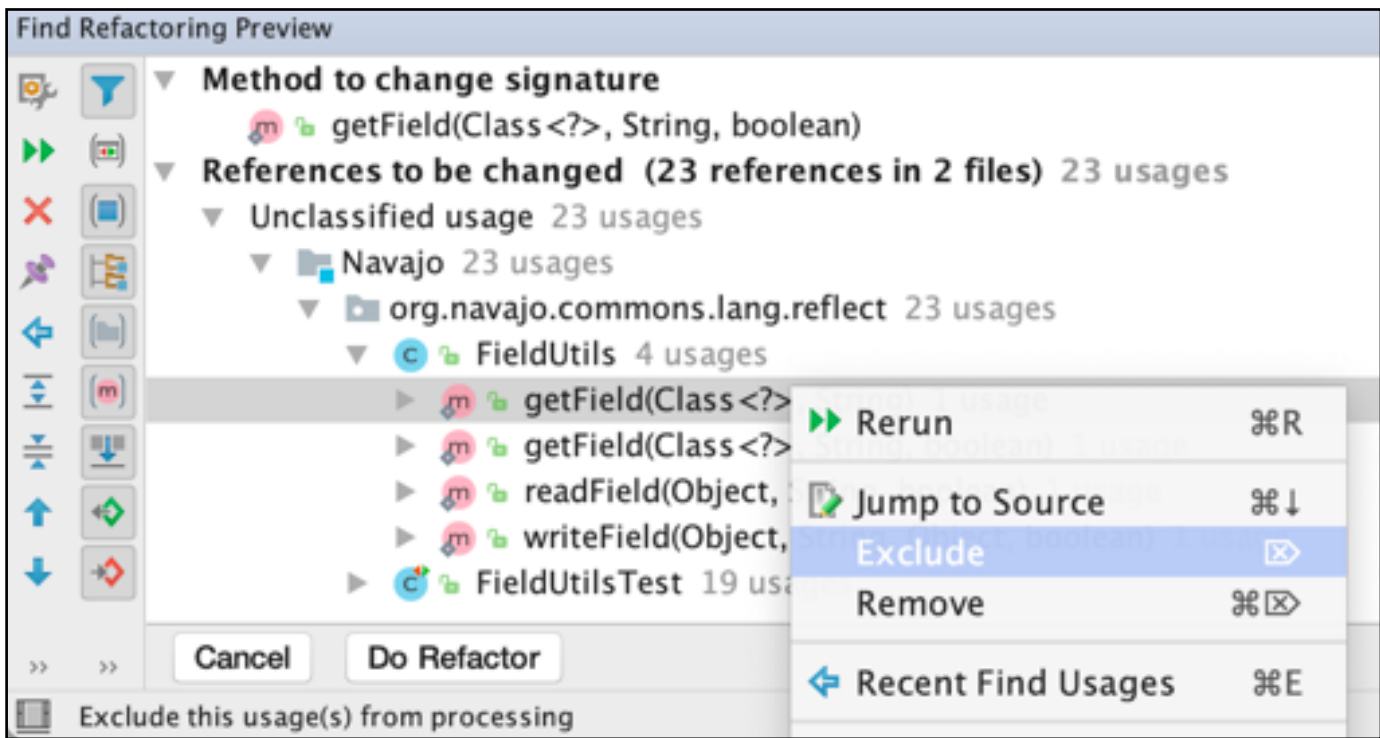
Configuration View



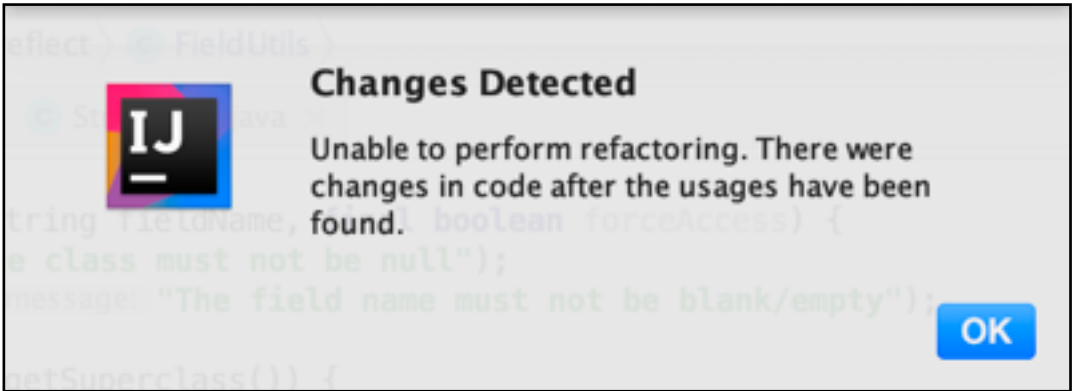
Impacts and additional changes



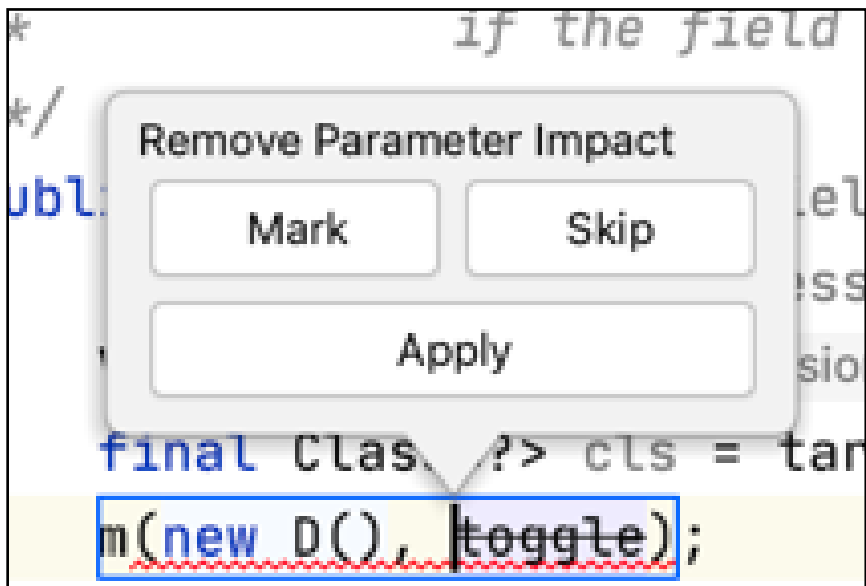
Preview View

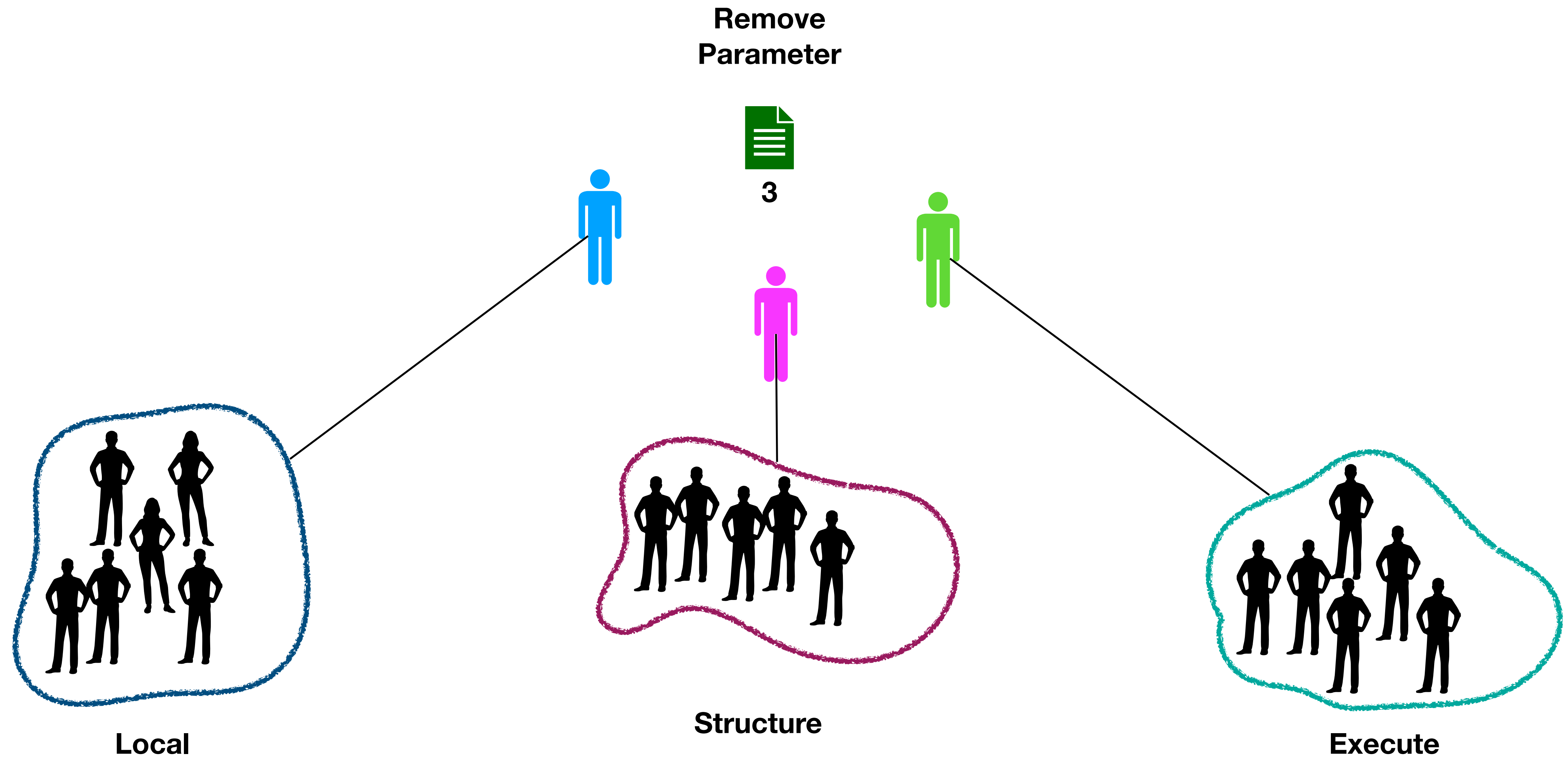


Can not change code



Ability to step through





```

14  *      match (ifcode public) fields.
15  *
16  *      Return the Field object
17  *      @throws IllegalArgumentException
18  *      if the class is (ifcode null), or the field name is blank or empty or is matched at multiple places
19  *      in the inheritance hierarchy
20  */
21
22  public static Field getField(final Class<?> cls, final String fieldName, final boolean forceAccess) {
23      Validate.isTrue(expression: cls != null, message: "The class must not be null");
24      Validate.isTrue(StringUtils.isEmpty(fieldName), message: "The field name must not be blank/empty");
25      // check up the superclass hierarchy
26      for (Class<?> acs = cls; acs != null; acs = acs.getSuperclass()) {
27          try {
28              final Field field = acs.getDeclaredField(fieldName);
29              // getDeclaredField checks for non-public scopes as well
30              // and it returns accurate results
31              if (Modifier.isPublic(field.getModifiers())) {
32                  if (forceAccess) {
33                      field.setAccessible(true);
34                  } else {
35                      continue;
36                  }
37              }
38              return field;
39          } catch (final NoSuchFieldException ex) { // NOPMD
40              // ignore
41          }
42      }
43      // check the public interface case. This must be manually searched for
44      // in case there is a public superclass field hidden by a private/package
45      // superclass field.
46      Field match = null;
47      for (final Class<?> class1 : ClassUtils.getAllInterfaces(cls)) {
48          try {
49              final Field test = class1.getField(fieldName);
50              Validate.isTrue(expression: match == null, message: "Reference to field %s is ambiguous relative to %s"
51                  + ": a matching field exists on two or more implemented interfaces.", fieldName, cls);
52          }
53      }
54      return match;
55  }

```

- An interaction model for a stepwise refactoring tool, and a prototype of such a tool, that showcases ways to address multiple of the usability issues that were previously identified

Future Work

- Is a stepwise refactoring tool more usable?
- How can a stepwise refactoring tool be integrated with other tool types, like "quick-fixes"?
- Do developers exhibit these three strategies in other software change tasks?
- Which other usability factors should be considered by toolmakers?

Summary of contributions

- Three strategies: Local, Structure, and Execute
- A theory of usability for refactoring tools
- Four usability themes
- Stepwise refactoring tool

