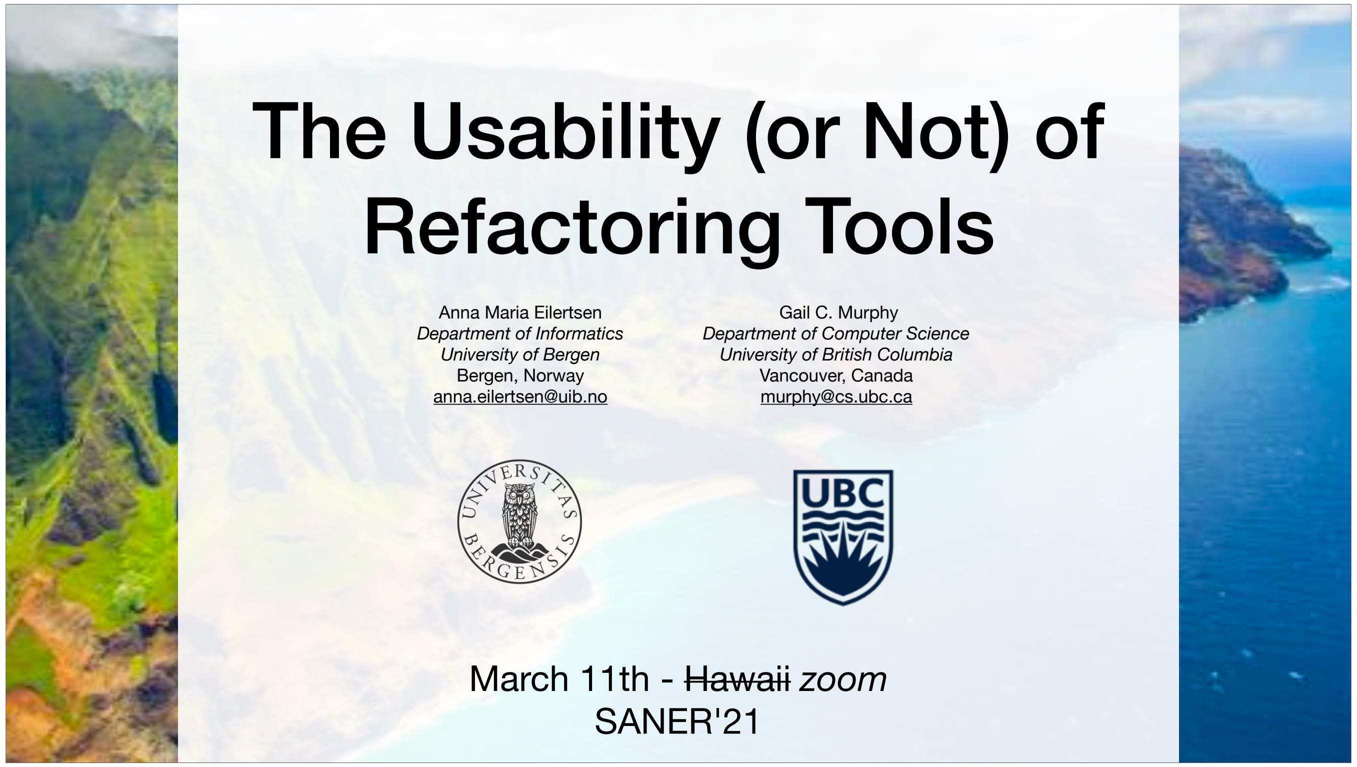


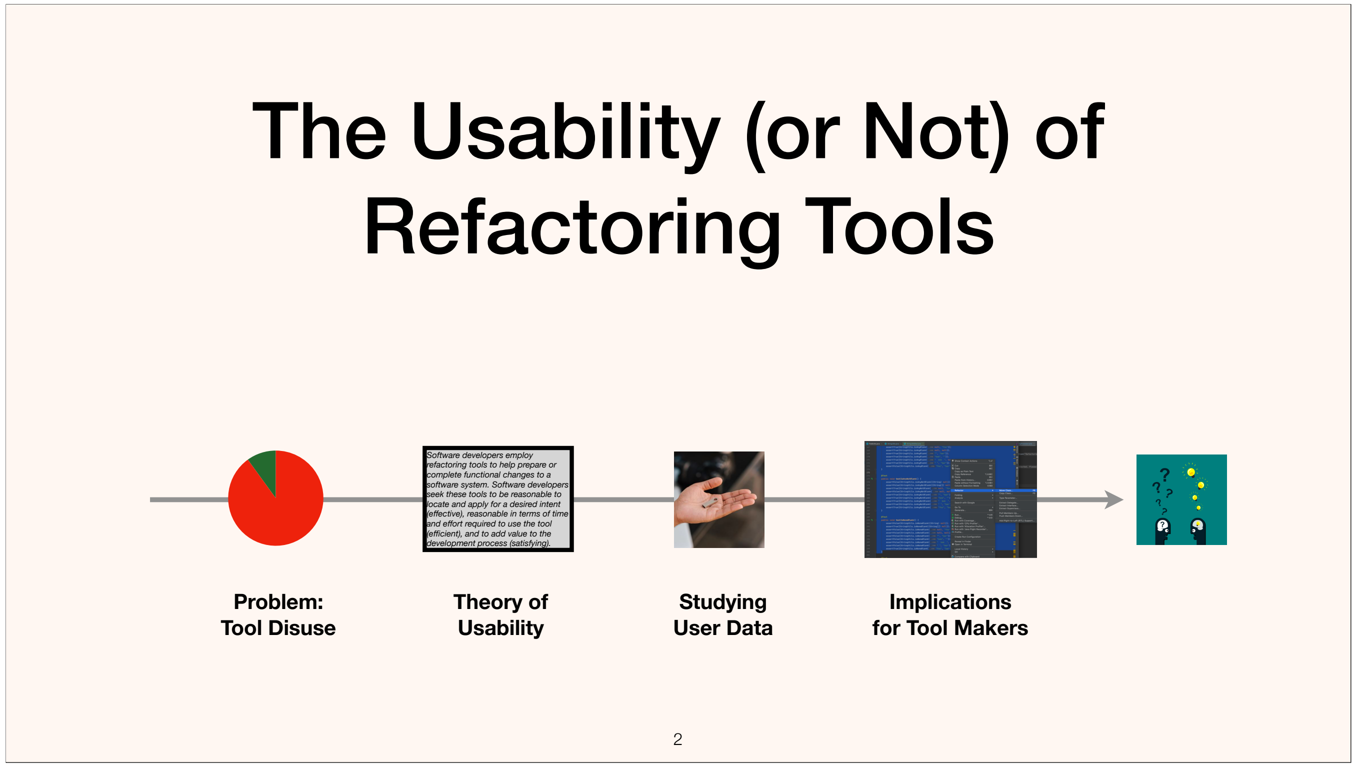
The Usability (or Not) of Refactoring Tools

Anna Maria Eilertsen
Department of Informatics
University of Bergen
Bergen, Norway
anna.eilertsen@uib.no

Gail C. Murphy
Department of Computer Science
University of British Columbia
Vancouver, Canada
murphy@cs.ubc.ca

March 11th - ~~Hawaii~~ *zoom*
SANER'21

Hello everyone, thank you for attending my talk. My name is Anna Eilertsen from University of Bergen in Norway and I will be presenting our work on Refactoring tool Usability.



The Usability (or Not) of Refactoring Tools

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).

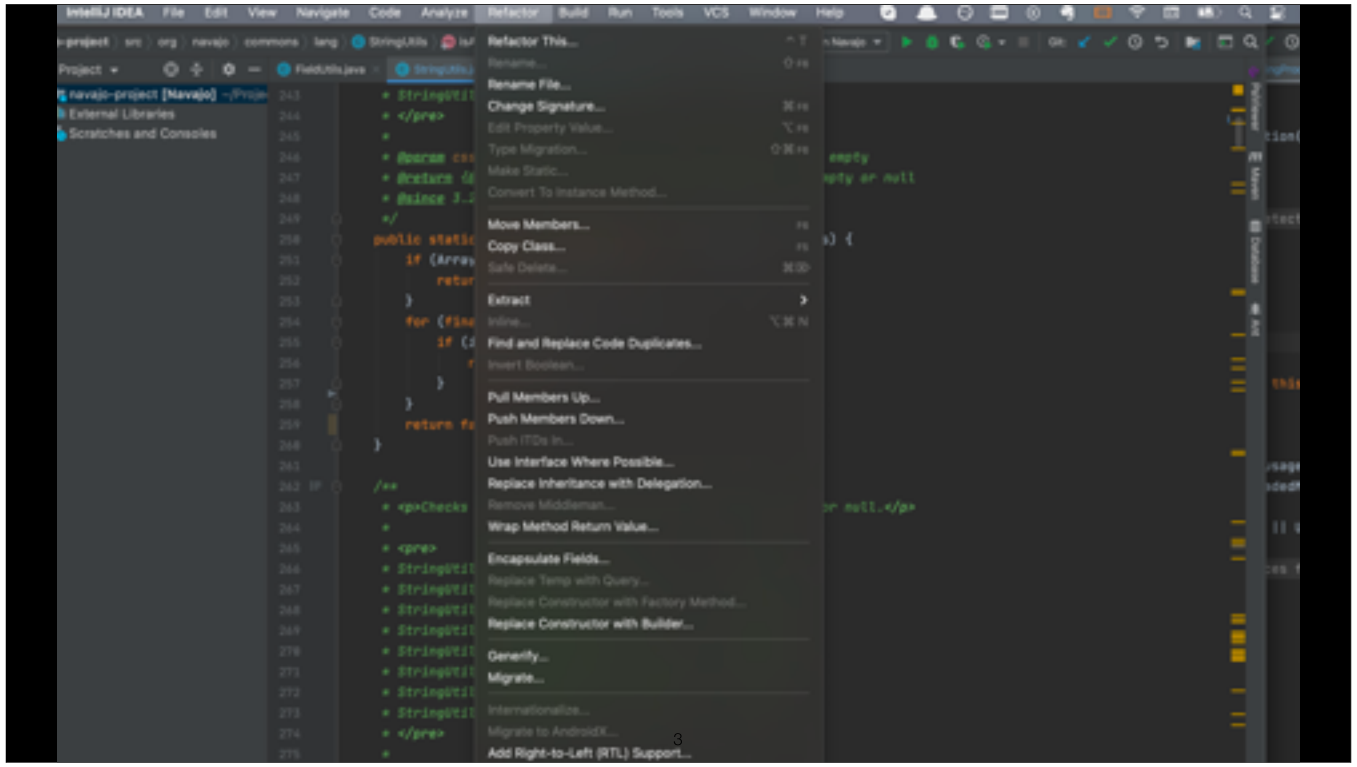
Problem: Tool Disuse

Theory of Usability

Studying User Data

Implications for Tool Makers

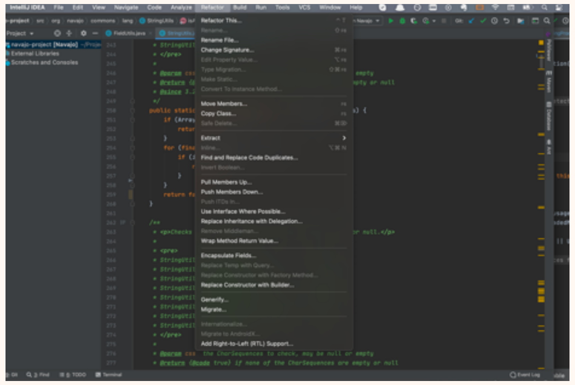
Today I will first introduce you to the problem of refactoring tool disuse. Then, I will show how we derive a definition of refactoring tool usability, which we then use to study user data from developers. I will focus on the insights that emerged from the user data because it is from this insight and its basis in the data that we refine our theory and suggest implications for toolmakers, which is well-described in the paper.



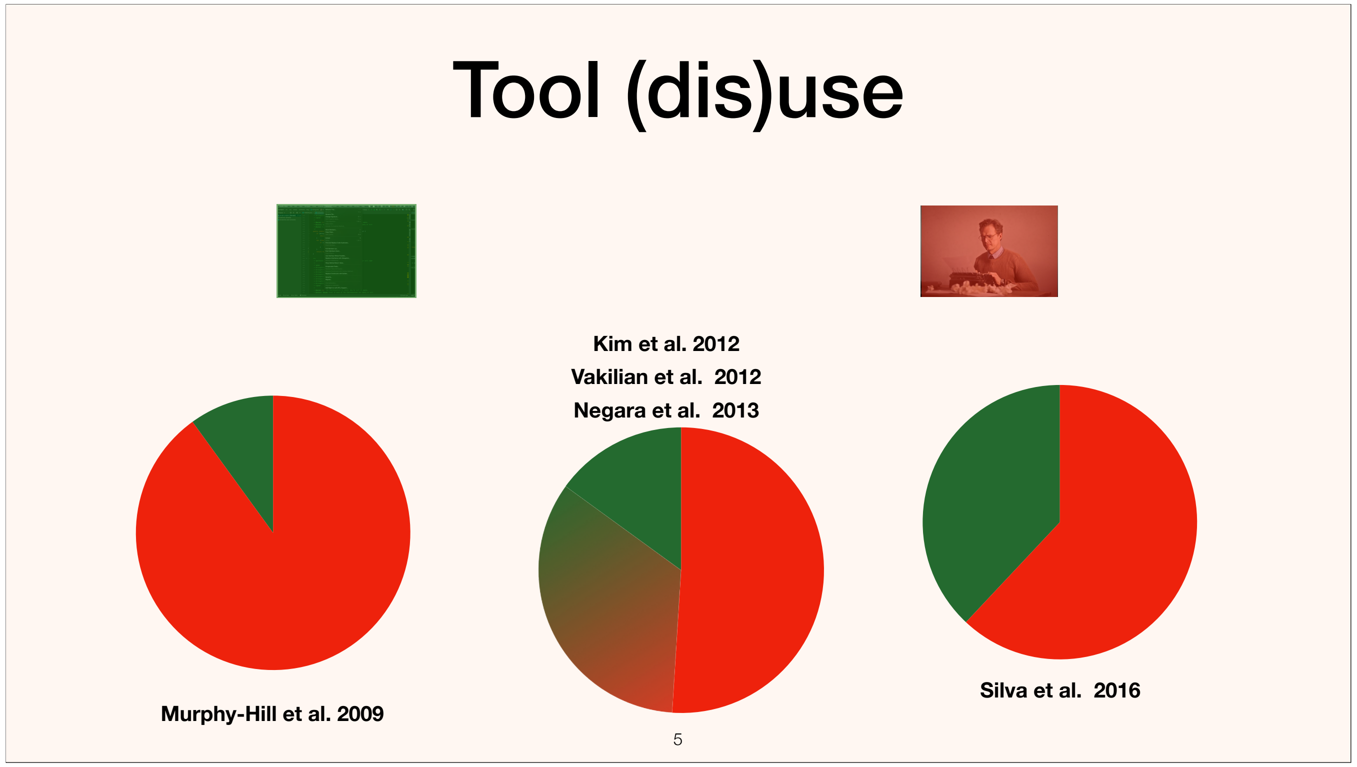
We focus here on refactoring tools that are available in mainstream IDEs for object-oriented languages like Java. Here is the refactoring menu in IntelliJ, which, similarly to Eclipse, ships with support for over 40 refactoring operations.

We can invoke for example Move, Inline, and Change Signature, and the tool is likely to perform it faster and more correctly than a developer that manually makes the same change, through, for example, search, cutting and pasting, compiler errors and so on.

Tool (dis)use



Despite the benefits of using automated refactoring tools, developers mostly make such changes manually.



Tool (dis)use

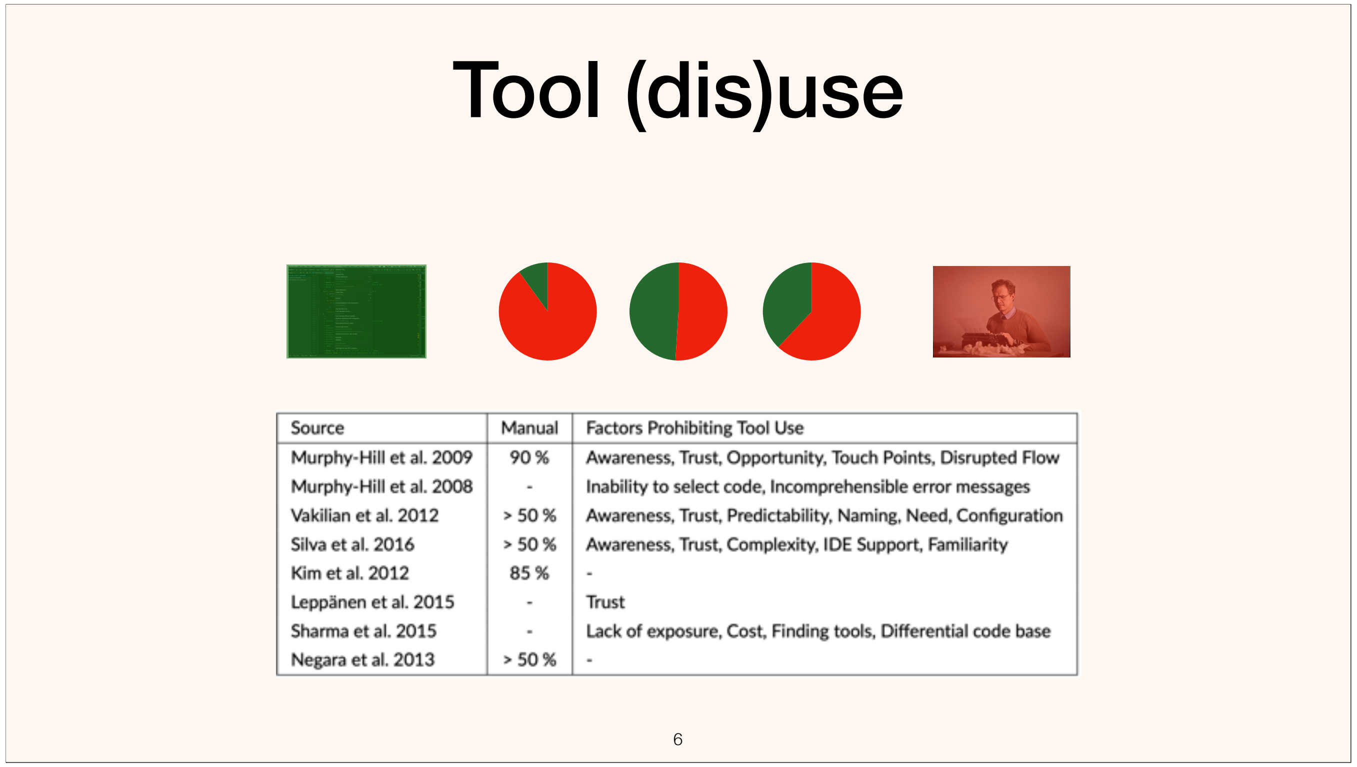
Kim et al. 2012
Vakilian et al. 2012
Negara et al. 2013

Murphy-Hill et al. 2009

Silva et al. 2016

Many studies have looked at the ratios between automated and manual refactoring applications, and across studies, a persistent finding is that:

Software developers perform more than half of refactorings manually instead of using refactoring tools.



Tool (dis)use

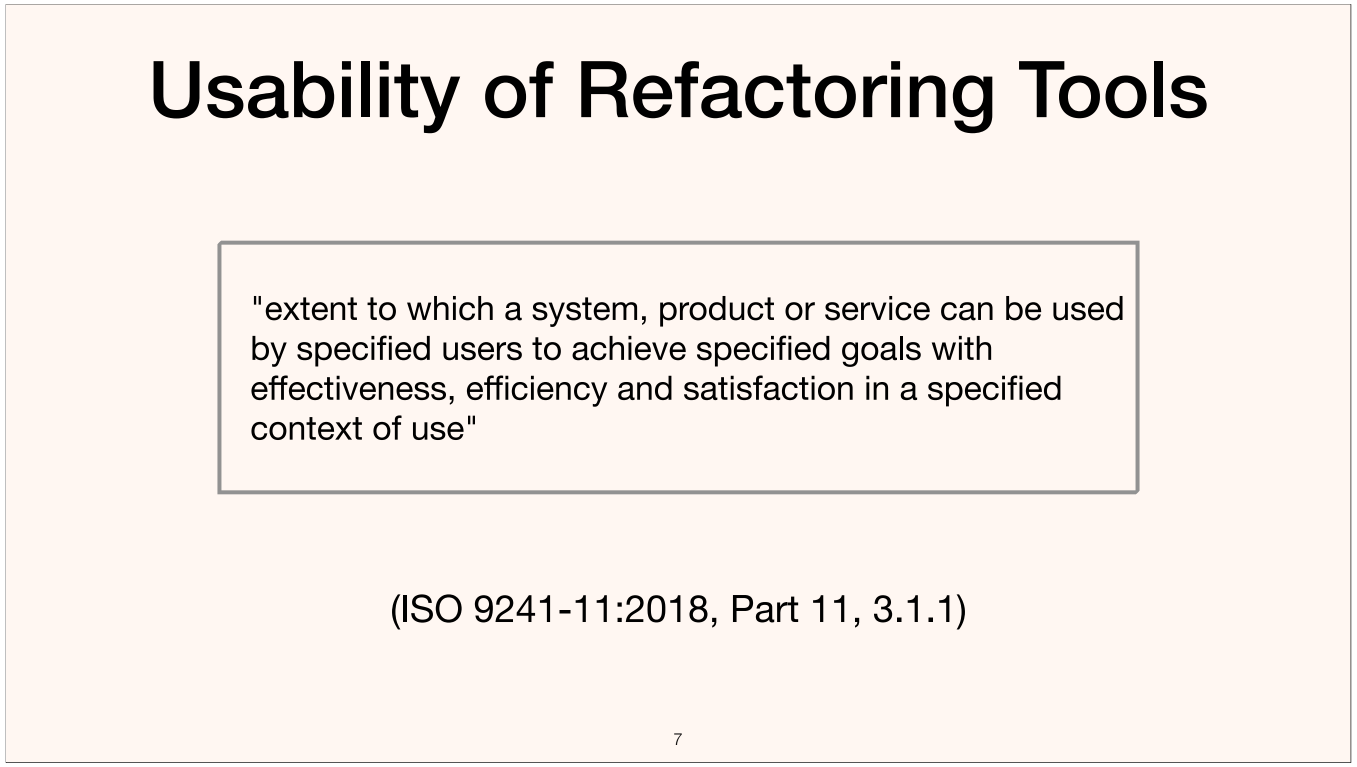
Source	Manual	Factors Prohibiting Tool Use
Murphy-Hill et al. 2009	90 %	Awareness, Trust, Opportunity, Touch Points, Disrupted Flow
Murphy-Hill et al. 2008	-	Inability to select code, Incomprehensible error messages
Vakilian et al. 2012	> 50 %	Awareness, Trust, Predictability, Naming, Need, Configuration
Silva et al. 2016	> 50 %	Awareness, Trust, Complexity, IDE Support, Familiarity
Kim et al. 2012	85 %	-
Leppänen et al. 2015	-	Trust
Sharma et al. 2015	-	Lack of exposure, Cost, Finding tools, Differential code base
Negara et al. 2013	> 50 %	-

Now, of course we want to know what we can do about that, so researchers also looked into factors prohibiting tool use. Here I list papers that we found that investigate either tool use ratios or such factors.

Of course some developers say, well, I have never heard of refactoring so I am not aware of them and cannot use them. But developers also say, I am aware of those tools, they are in my IDE but I don't want to use them, I don't trust them, they disrupt my workflow and they are unpredictable.

So, this is interesting, because there is clearly some potential for improvements here, but these factors are not well-defined and therefore they are not easy to address with technical improvements. Especially this term trust is interesting: it comes up a lot and it is sometimes conflated with predictability, but it is unclear what it means, whether it is referring to some kind of tool correctness or something else.

These works indicate that disuse is a problem worth addressing, and suggest that trust or predictability might be important terms to look for.



Usability of Refactoring Tools

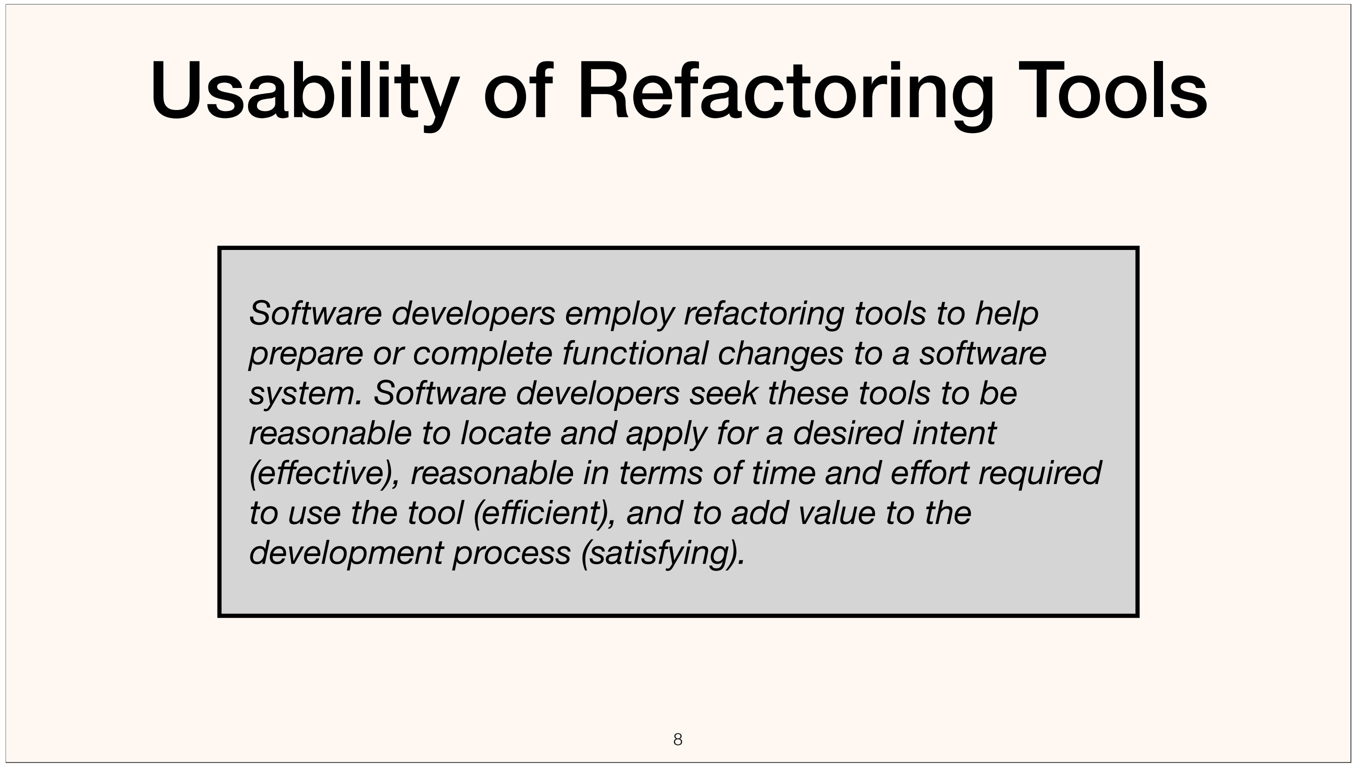
> "extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use"

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

We want to help toolmakers address the problem of refactoring tool disuse by providing them with a concise definition of usability of refactoring tools.

We start with the ISO definition of usability from their section on human-system interaction.

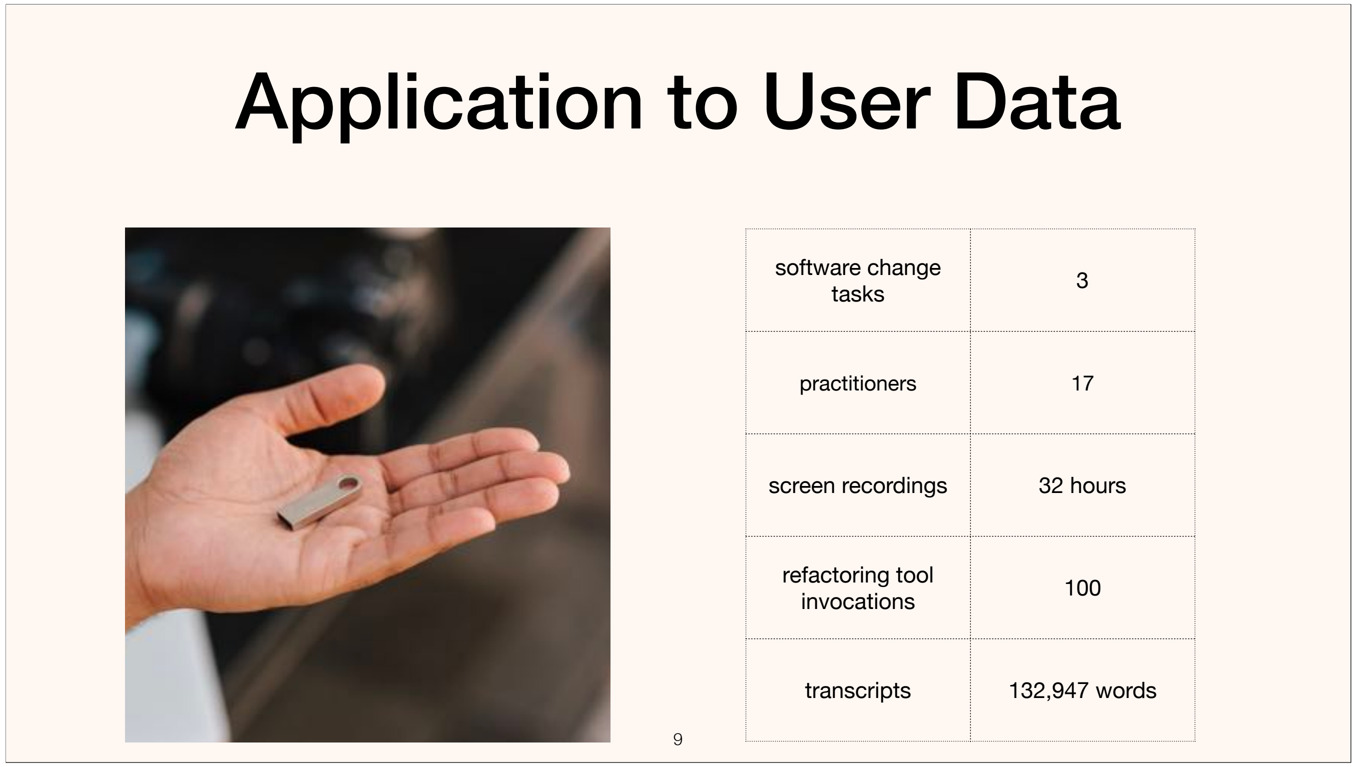
We operationalize it by, so-to-say, expanding the keywords, system, users, goals, and so on, based on recent research on refactoring tool use.



Usability of Refactoring Tools

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).

So this is the definition we achieve for now. And it has these usability factors, effective, efficient and satisfactory, that we are going to use to inspect our user data.

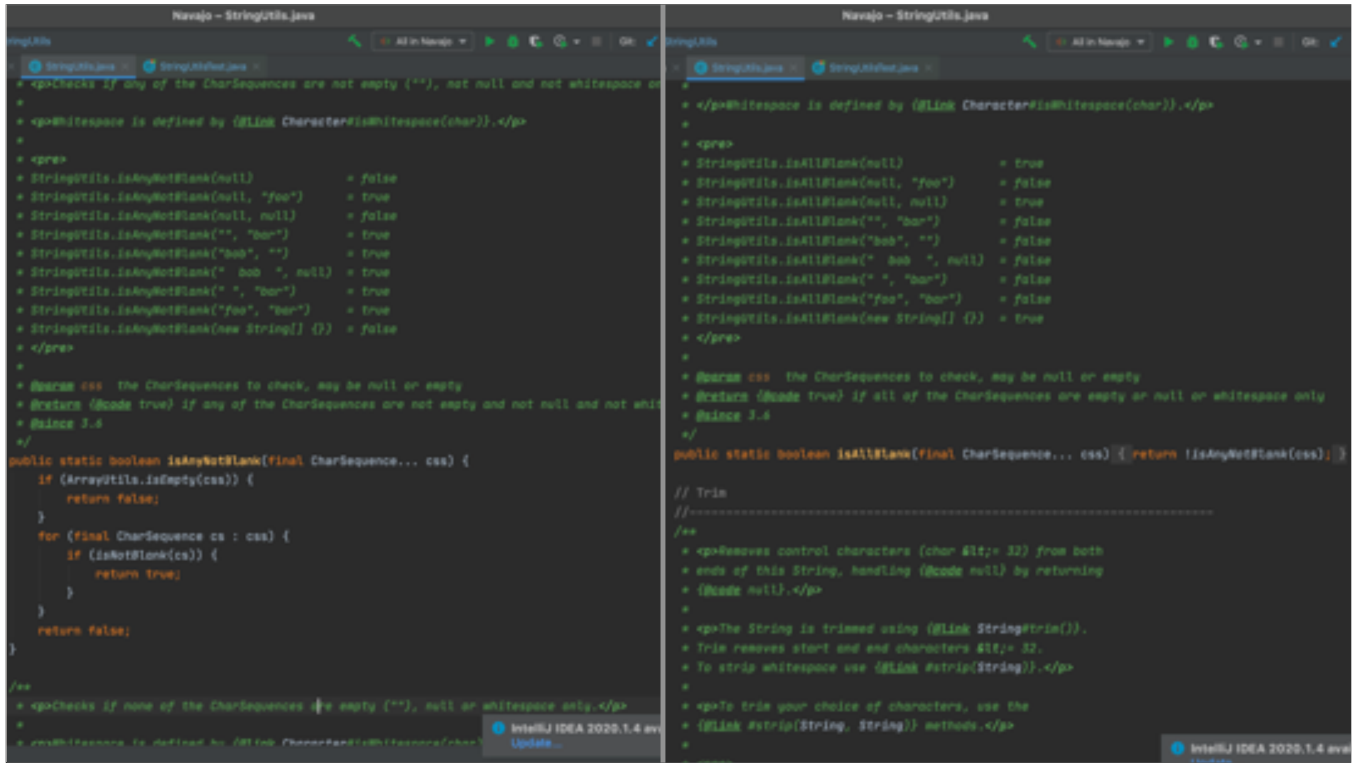


Application to User Data

software change tasks	3
practitioners	17
screen recordings	32 hours
refactoring tool invocations	100
transcripts	132,947 words

The user data we had available to us is from a user study we performed. The full study is part of a different manuscript currently undergoing review in a journal. I will briefly introduce it here. The study consists of three realistic software change tasks that are amenable to refactorings.

To illustrate what we mean by that, here is an example from task two,

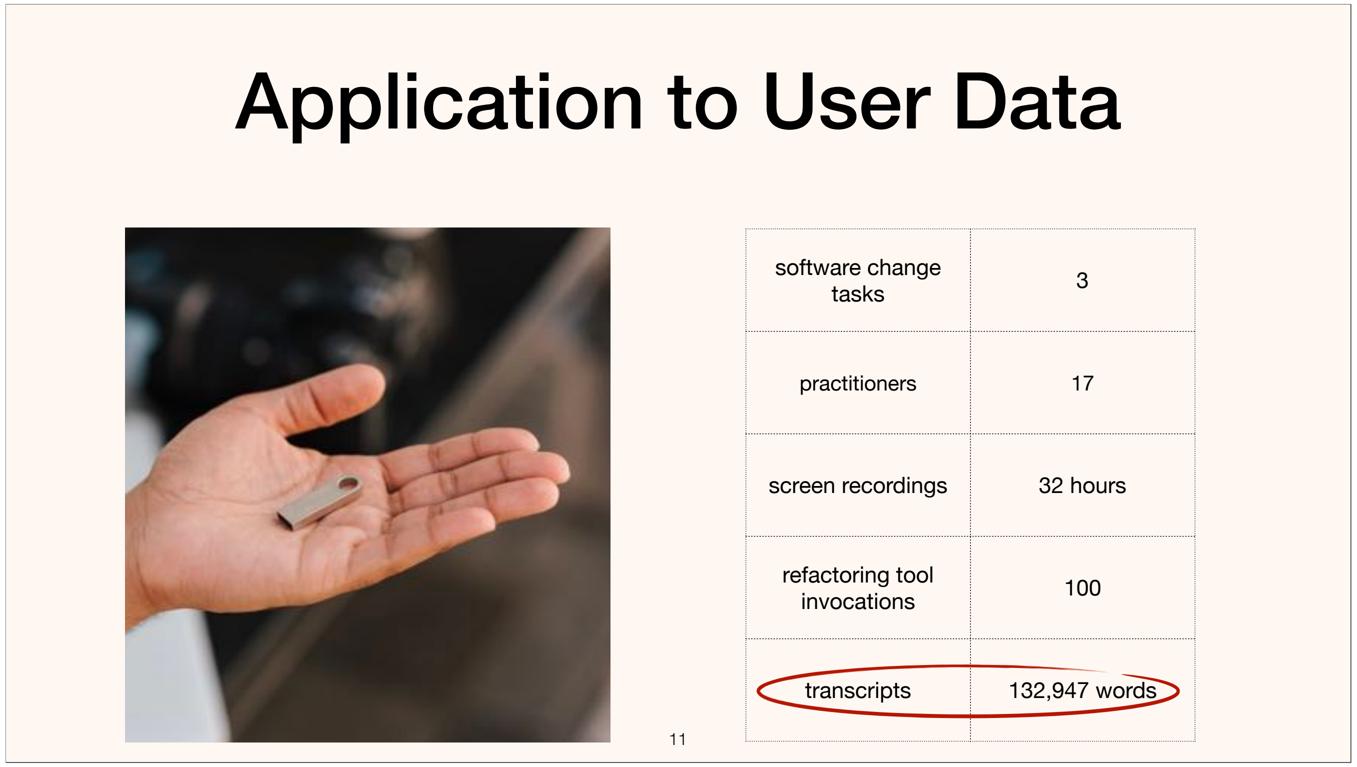


```java
 * <p>Checks if any of the CharSequences are not empty (""), not null and not whitespace on
 *
 * <p>Whitespace is defined by {@link Character#isWhitespace(char)}.</p>
 *
 * <pre>
 * StringUtils.isAnyNotBlank(null)             = false
 * StringUtils.isAnyNotBlank(null, "foo")      = true
 * StringUtils.isAnyNotBlank(null, null)       = false
 * StringUtils.isAnyNotBlank("", "bar")        = true
 * StringUtils.isAnyNotBlank("bob", "")        = true
 * StringUtils.isAnyNotBlank("  bob  ", null)  = true
 * StringUtils.isAnyNotBlank(" ", "bar")       = true
 * StringUtils.isAnyNotBlank("foo", "bar")     = true
 * StringUtils.isAnyNotBlank(new String[] {})  = false
 * </pre>
 *
 * @param css  the CharSequences to check, may be null or empty
 * @return {@code true} if any of the CharSequences are not empty and not null and not whit
 * @since 3.4
 */
public static boolean isAnyNotBlank(final CharSequence... css) {
    if (ArrayUtils.isEmpty(css)) {
        return false;
    }
    for (final CharSequence cs : css) {
        if (isNotBlank(cs)) {
            return true;
        }
    }
    return false;
}

/**
 * <p>Checks if none of the CharSequences are empty (""), null or whitespace only.</p>
 *
```

```java
 *
 * </p>Whitespace is defined by {@link Character#isWhitespace(char)}.</p>
 *
 * <pre>
 * StringUtils.isAllBlank(null)             = true
 * StringUtils.isAllBlank(null, "foo")      = false
 * StringUtils.isAllBlank(null, null)       = true
 * StringUtils.isAllBlank("", "bar")        = false
 * StringUtils.isAllBlank("bob", "")        = false
 * StringUtils.isAllBlank("  bob  ", null)  = false
 * StringUtils.isAllBlank(" ", "bar")       = false
 * StringUtils.isAllBlank("foo", "bar")     = false
 * StringUtils.isAllBlank(new String[] {})  = true
 * </pre>
 *
 * @param css  the CharSequences to check, may be null or empty
 * @return {@code true} if all of the CharSequences are empty or null or whitespace only
 * @since 3.4
 */
public static boolean isAllBlank(final CharSequence... css) { return !isAnyNotBlank(css); }

// Trim
//-----------------------------------------------------------------------
/**
 * <p>Removes control characters (char &lt;= 32) from both
 * ends of this String, handling {@code null} by returning
 * {@code null}.</p>
 *
 * <p>The String is trimmed using {@link String#trim()}.
 * Trim removes start and end characters &lt;= 32.
 * To strip whitespace use {@link #strip(String)}.</p>
 *
 * <p>To trim your choice of characters, use the
 * {@link #strip(String, String)} methods.</p>
 *
```

.. in this task, participants are told there are a number of method pairs where the methods are negations of each other. Here you see one such pair, isAllBlank and isAnyNotBlank. These pairs have been added between releases, and as we approach our next release we decided to NOT include the double-negated one, so the task is to remove isAnyNotBlank and the corresponding method in one other pair.

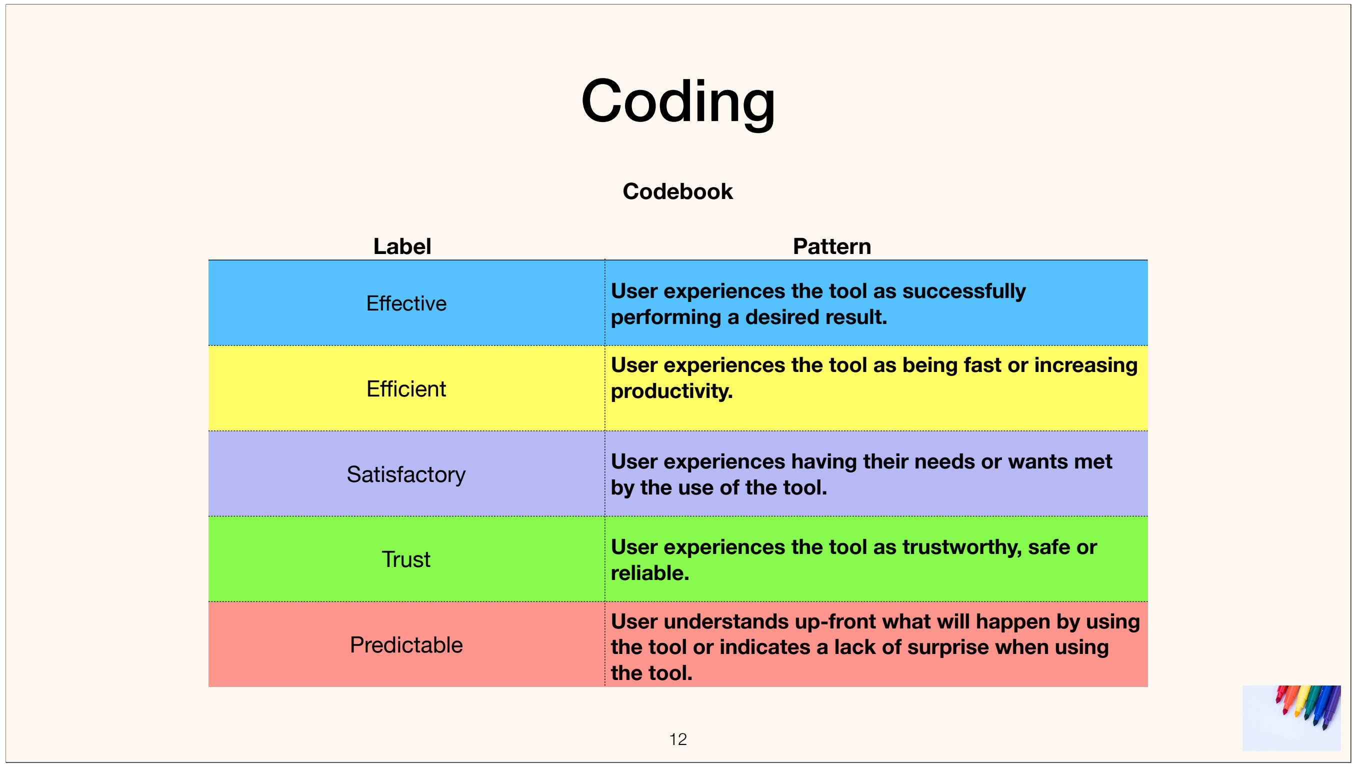
So the trick, of course, is that the double negated method contains the implementation of functionality and must be inlined into the other one but we don't mention that.



Application to User Data

software change tasks	3
practitioners	17
screen recordings	32 hours
refactoring tool invocations	100
transcripts	132,947 words

We focus on the transcripts from this study, which capture think-aloud audio recordings from their tasks and audio from subsequent interview segments. This will allow us to triangulate experiences that we find in the transcripts with the screen recordings so that if developers say they had an experience we can go and actually look at what happened.

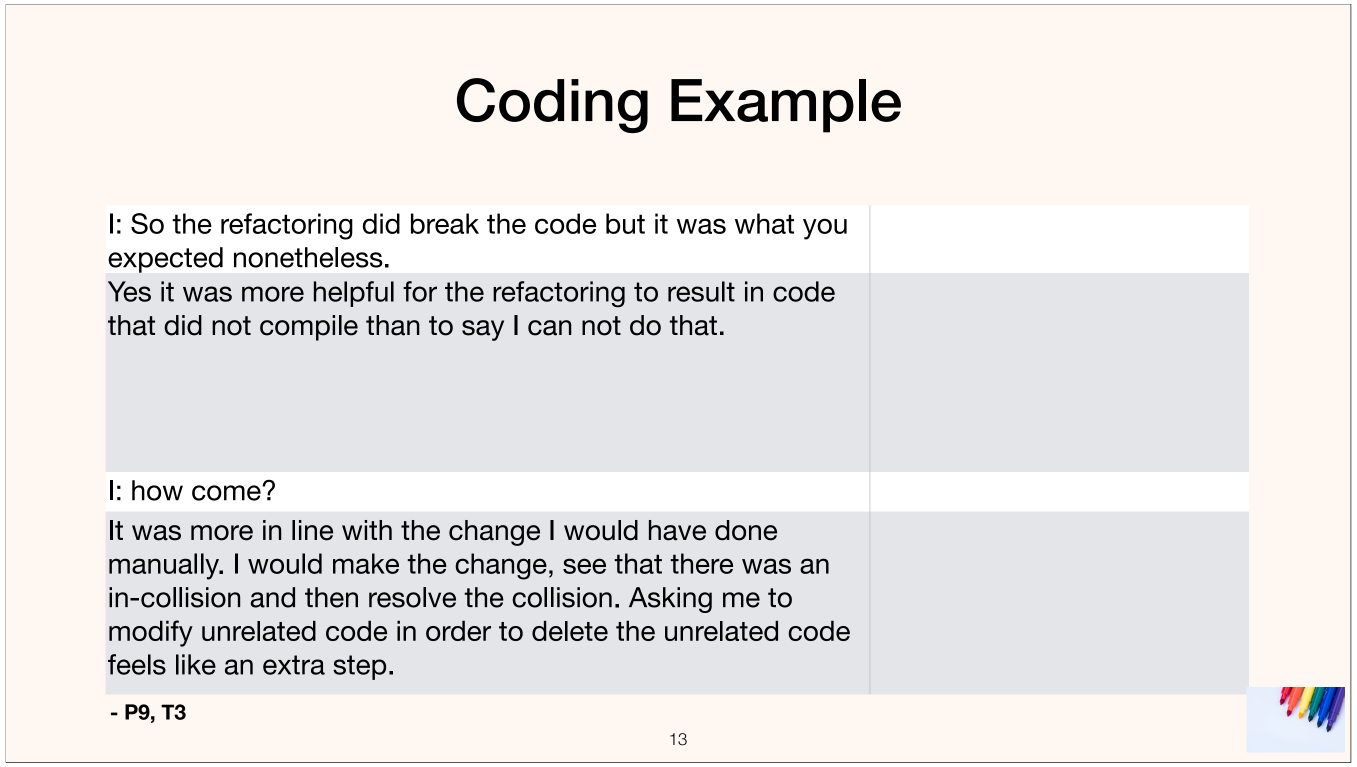


Coding

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.**

We approach these transcripts systematically by developing a codebook that contains labels for all the three usability factors from our definition, and the two experience related factors from related work. For each label, we have a pattern that we match the transcripts against, so if a statement matches a pattern we apply the corresponding label. We label the entire transcript this way.



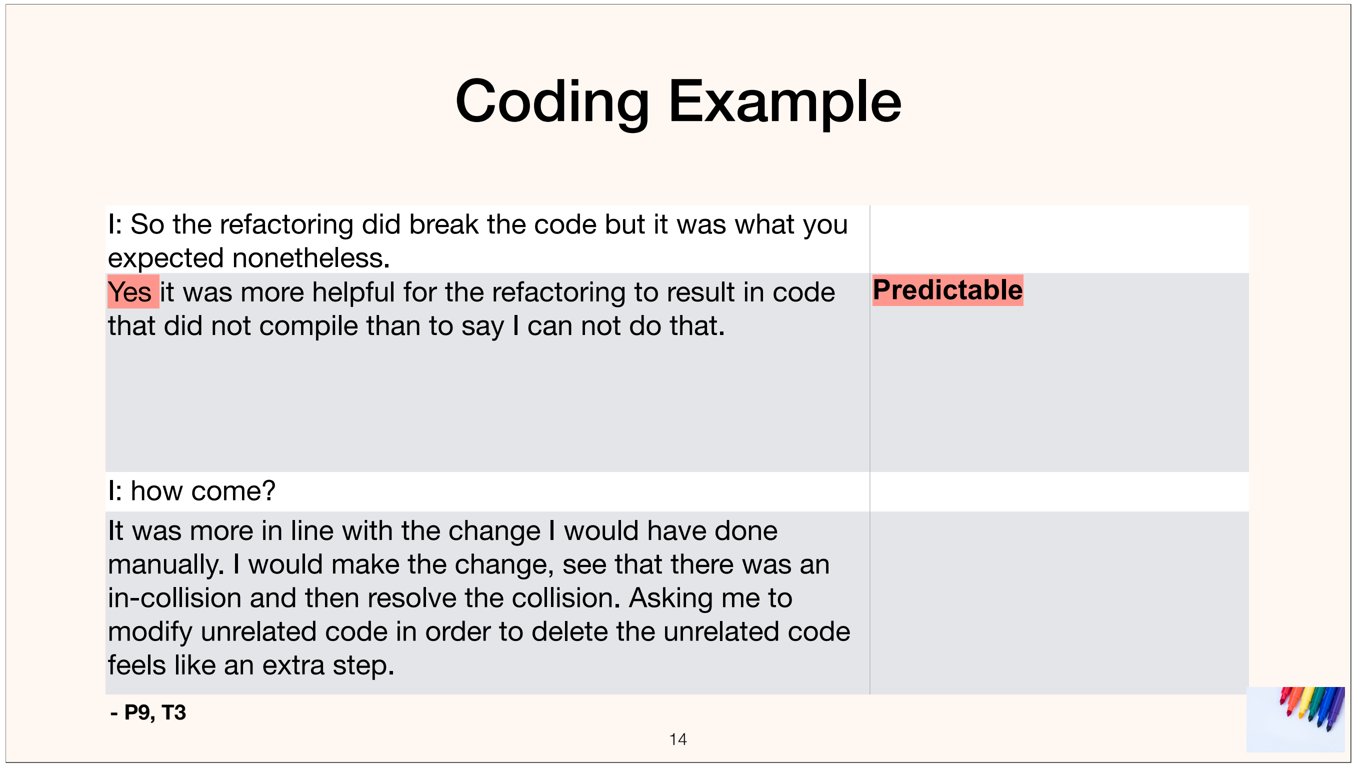
Coding Example

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

For a quick example, participant nine used Remove Parameter in task three, it introduced compiler errors but he was satisfied nonetheless, and we summarize,

“it broke the code but that’s what you expected”?
Yes.
Right, so here we mark “predictable”

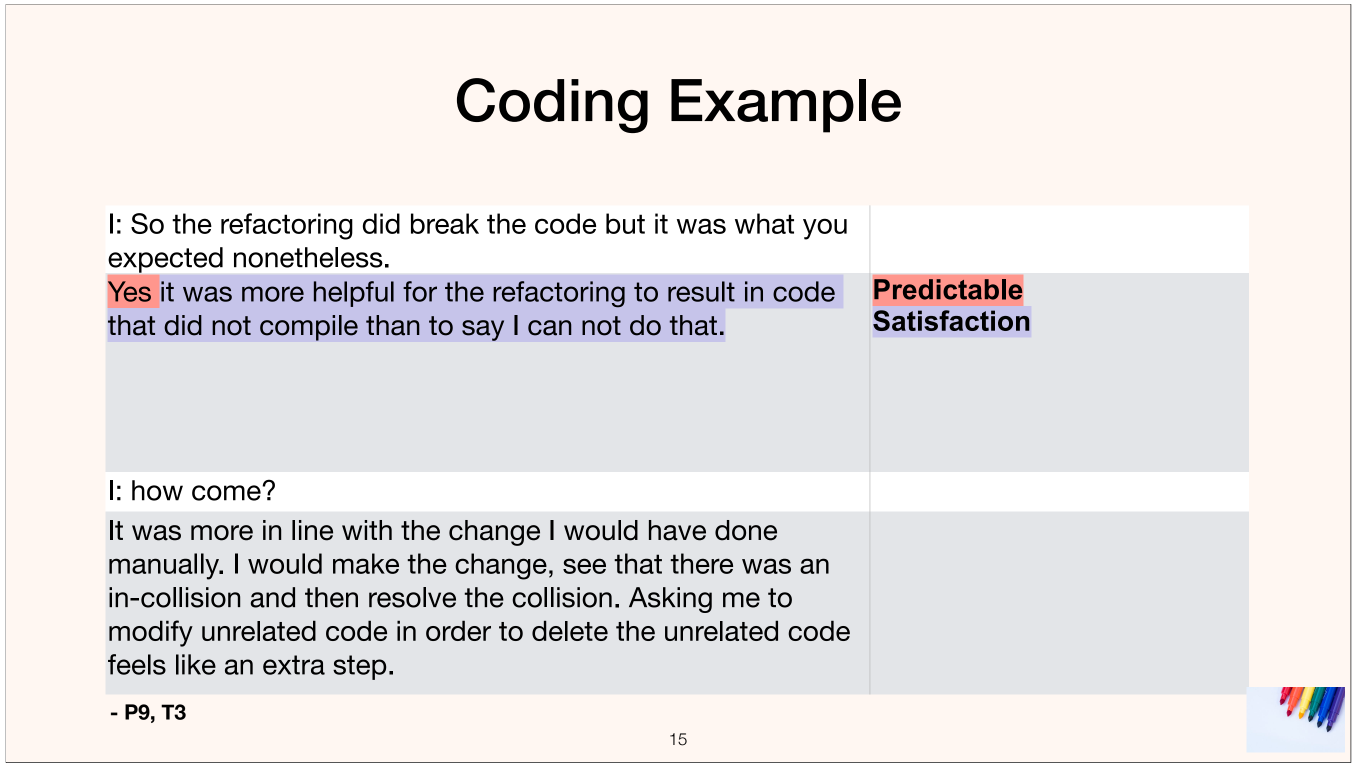


Coding Example

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

“It was more helpful than if it had rejected the invocation”
So it was helpful, so we mark it with Satisfaction.

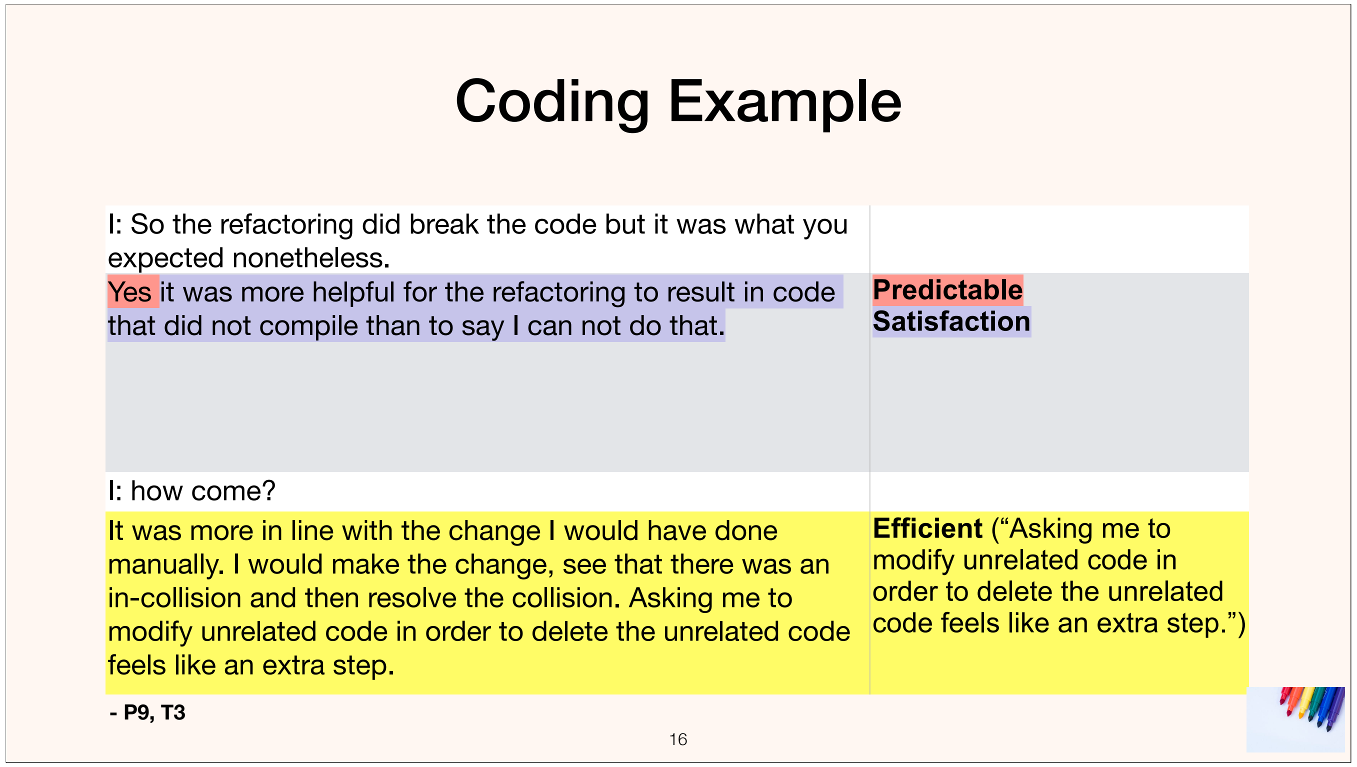


Coding Example

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

He goes on to say that it would have been an extra step to edit what he calls unrelated code - the code that triggered this warning.

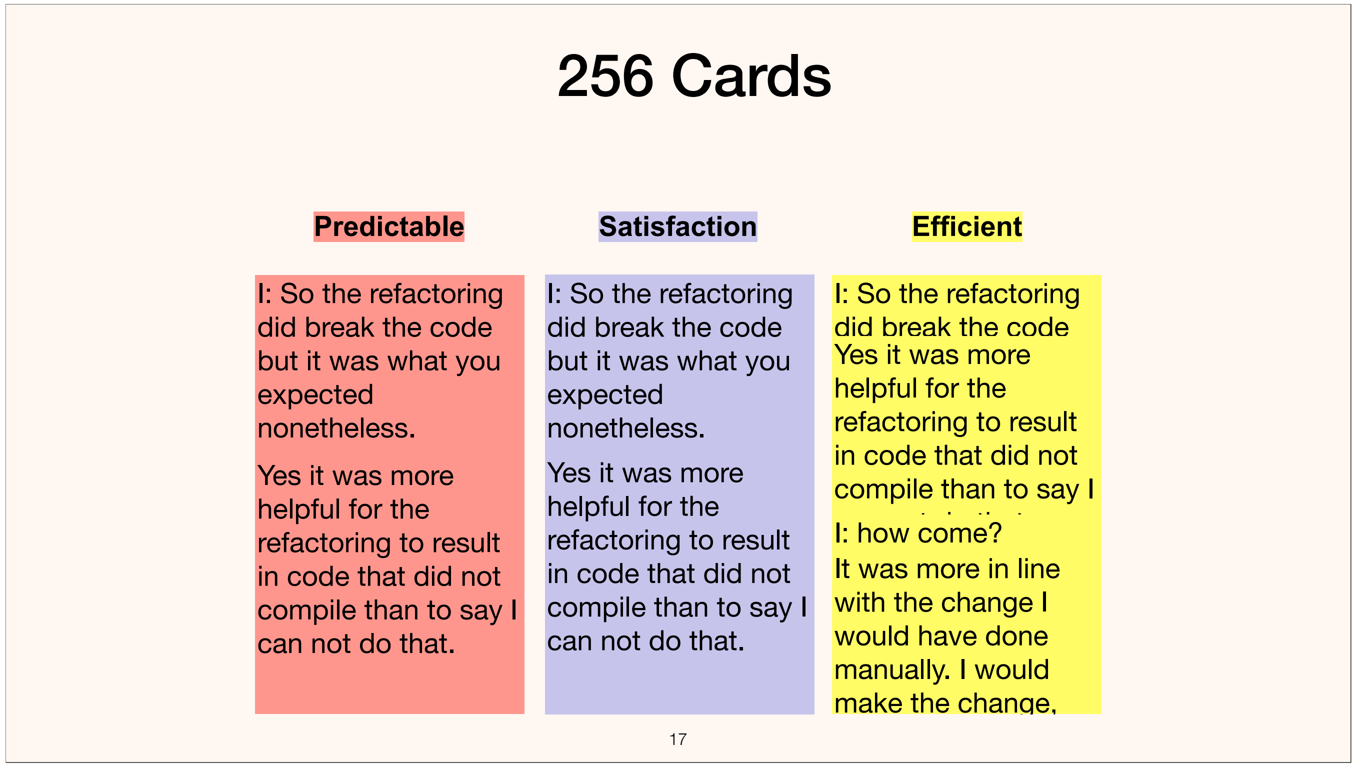


Coding Example

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

So we mark this as efficient.



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,

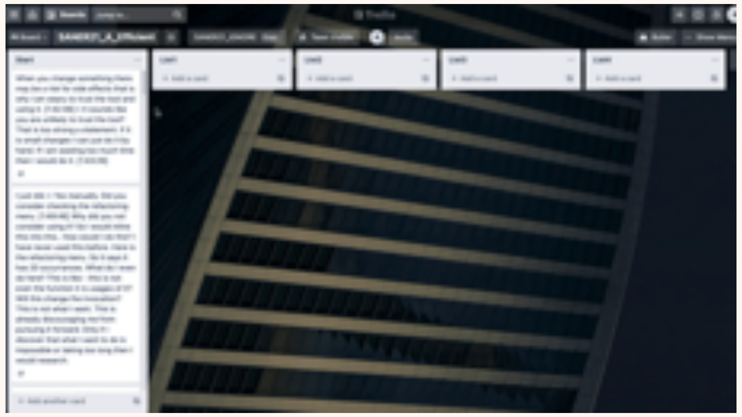
From the resulting data, we do a couple of different things, but I want to focus on the cardsorts that led to usability themes:

we create sets of cards, one set of each label. Here are the three cards we created from the previous example transcript.

Cardsort

Cards: 256

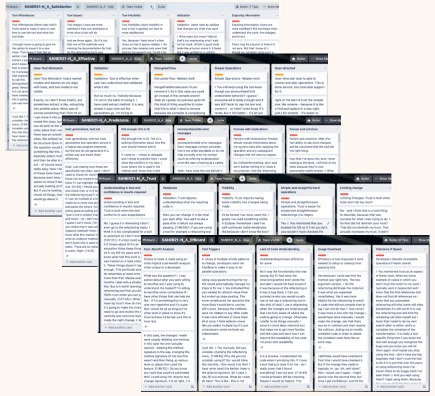
Predictable	79
Effective	53
Satisfaction	51
Efficient	42
Trust	31



(play video)
We do this for all labels, to create 256 cards. For each “set” of cards belonging to one label, the two authors jointly organize these cards into groups based on their perceived similarities. Once groups start emerging, we name and describe them based on the common themes in the cards contained in that group.

Cardsort

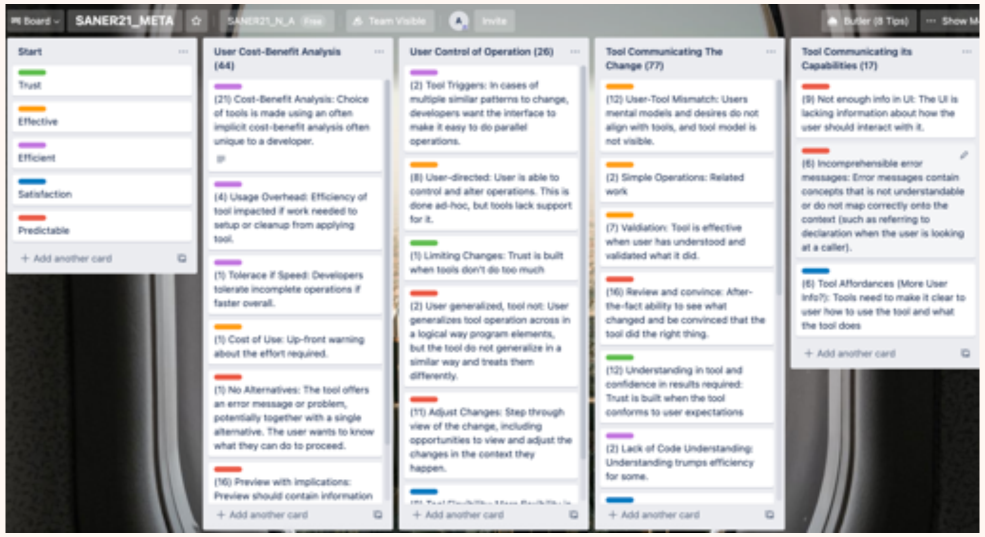
	Cards: 256
Predictable	79
Effective	53
Satisfaction	51
Efficient	42
Trust	31



Themes: 28
7
6
5
5
5

Across all sets, we find 28 themes in total. For each theme we have this little descriptive card, so we use those 28 cards as the starting set for another round of cardsort - kind of meta-cardsort..

Meta-Cardsort



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) Tolerace 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.

🟦

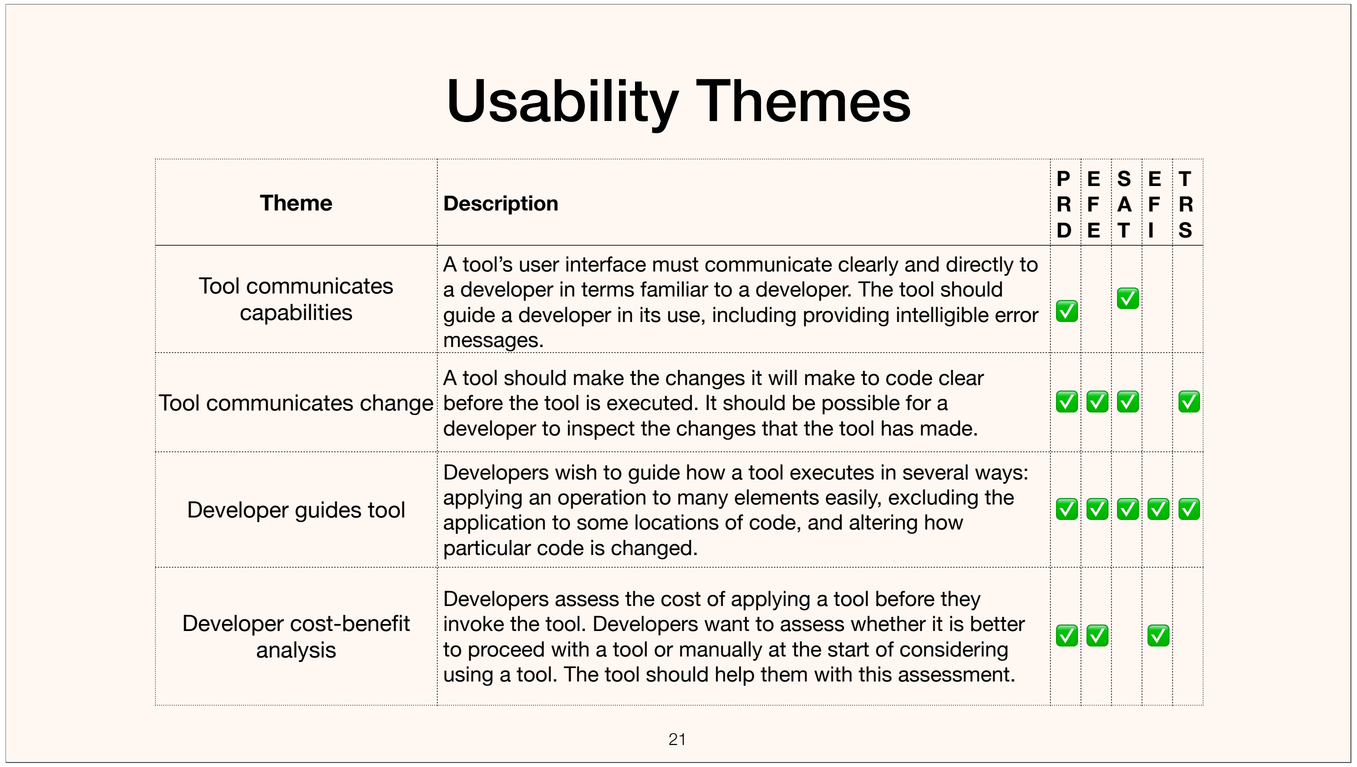
Tool Communicating its Capabilities (17)

🟥 (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

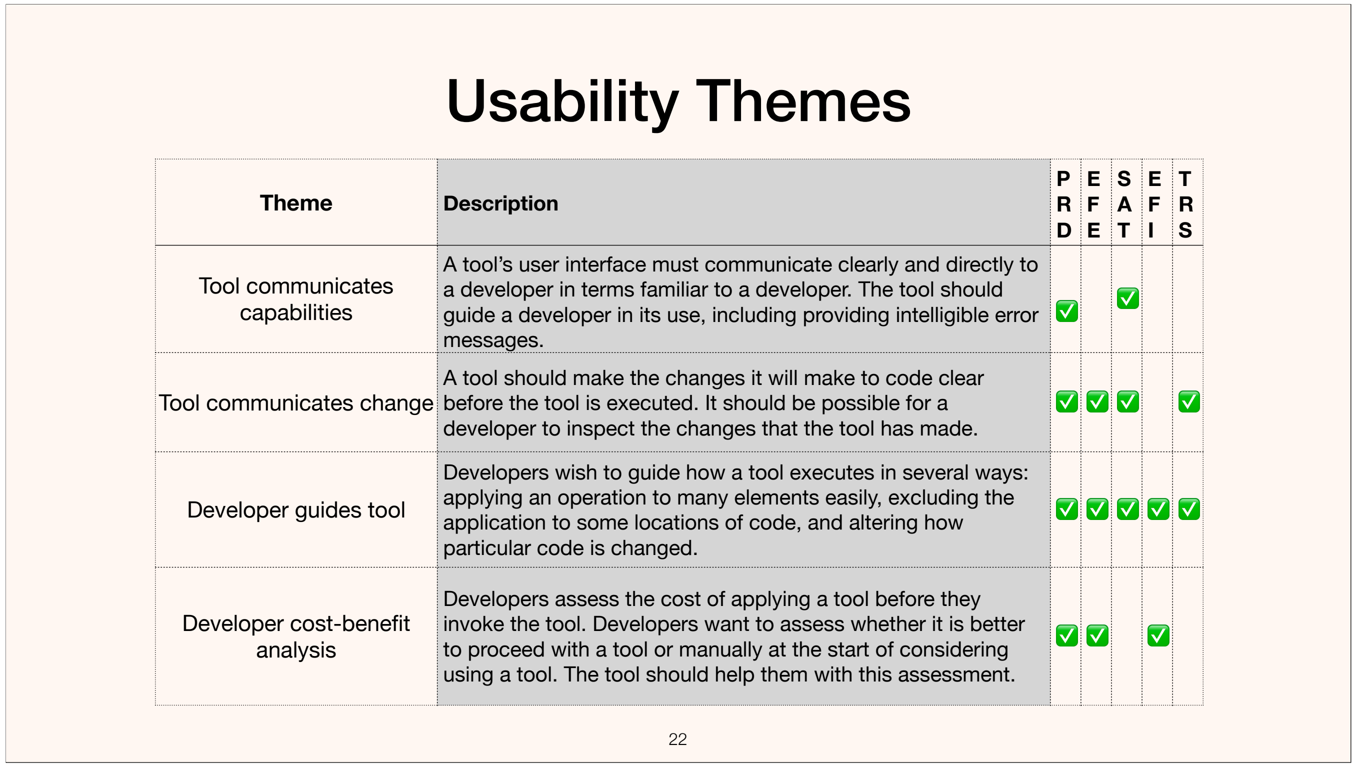
...to identify higher-level themes that emerge across the sets. This “meta-cardsort” results in these four usability themes.



Usability Themes

Theme	Description	PRD	EFE	SAT	EFI	TRS
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.	✅	✅		✅	

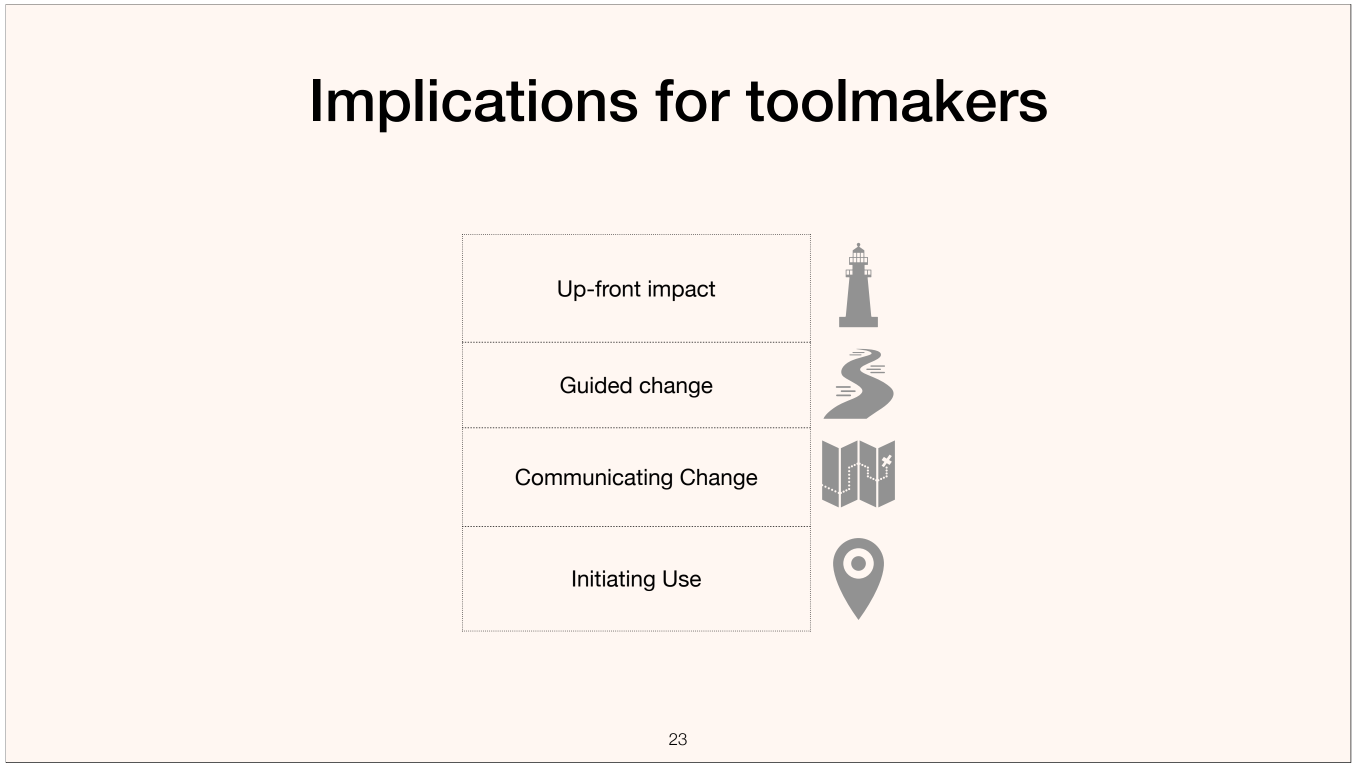
Here you can see the high-level usability themes on the left, and the factors it ties back to on the right. So you can see that for example, predictable is present in all of them, and so you can see for example, that the first one -- capabilities -- come from predictability and satisfaction, while developer guiding tools come from all the factors.



Usability Themes

Theme	Description	PRD	EFE	SAT	EFI	TRS
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.	✅	✅		✅	

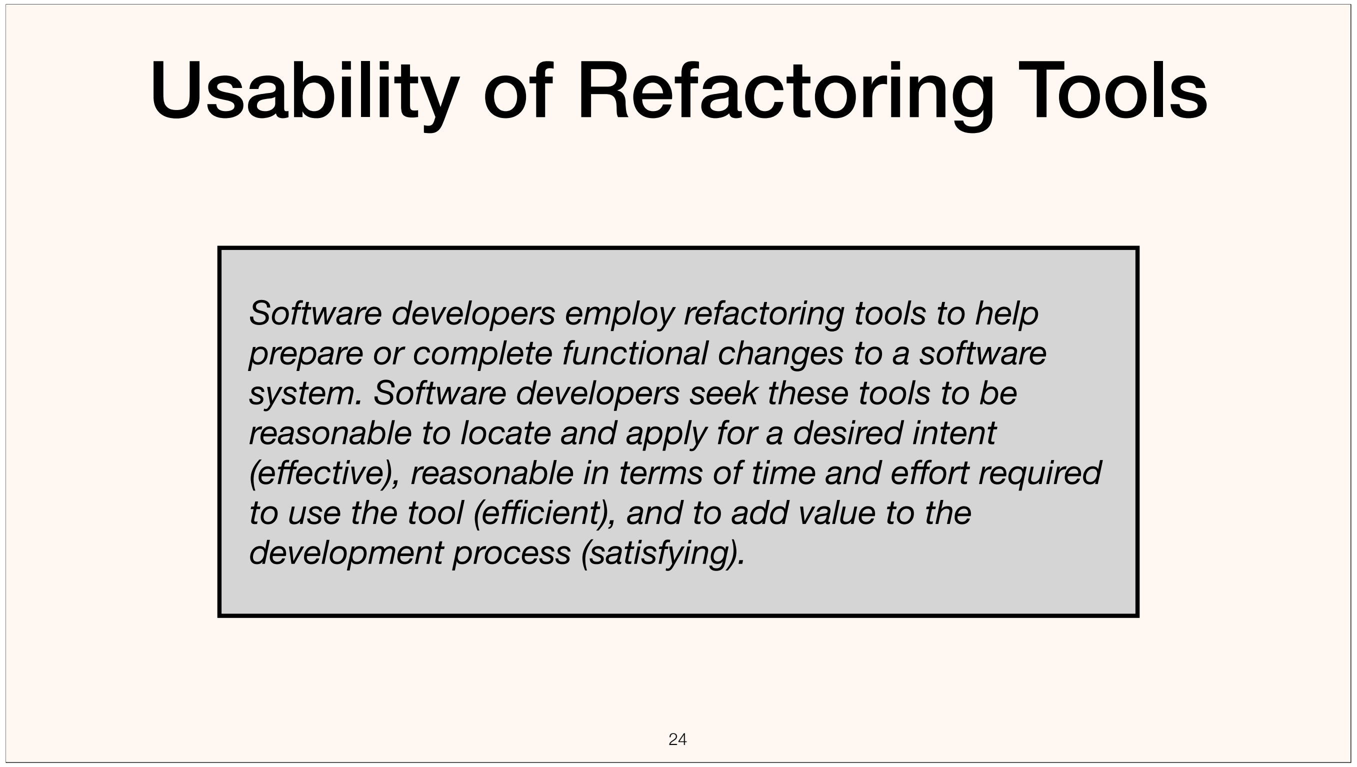
In order to develop our descriptions, we referred back to the transcripts that they arose from and to the screen recordings to figure out exactly what happens when participants encountered these themes. So this is how we expand our understanding of these themes,



Implications for toolmakers

Up-front impact
Guided change
Communicating Change
Initiating Use

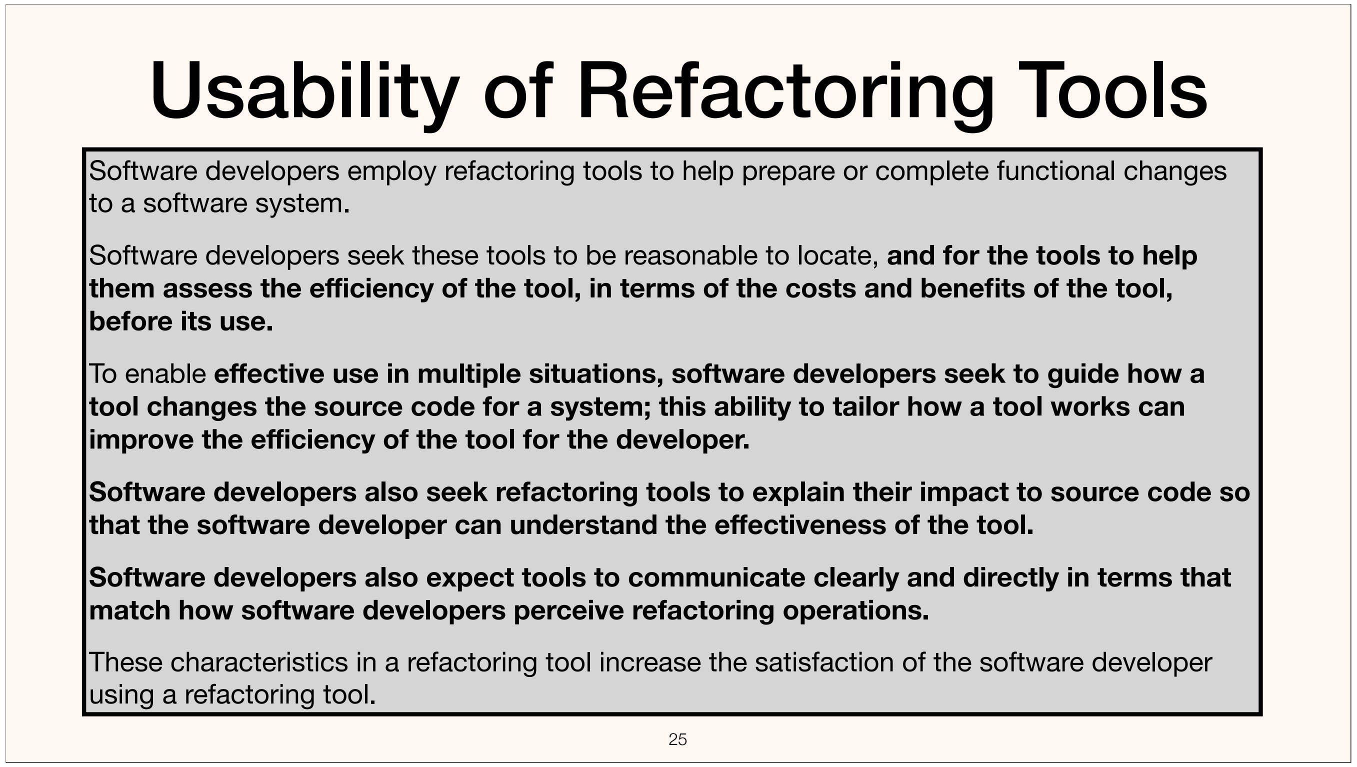
and we use this understanding both to identify ways in which the tools can be improved - this is described quite well in the paper so I will skip it for time -



Usability of Refactoring Tools

> *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).*

...and we use it to refine our definition that we created, to kind of flesh it out..



Usability of Refactoring Tools

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.**

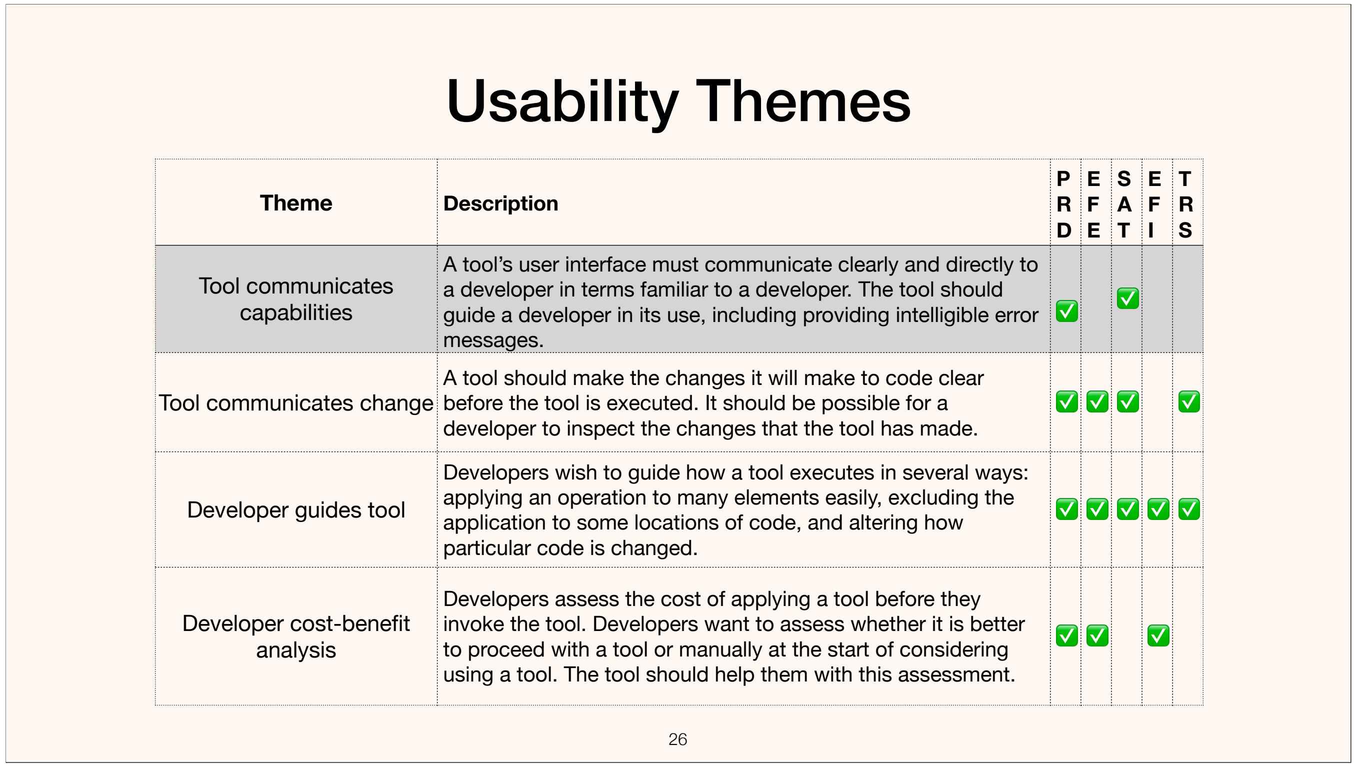
To enable **effective use in multiple situations, software developers seek to guide how a tool changes the source code for a system; this ability to tailor how a tool works can improve the efficiency of the tool for the developer.**

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.

..to be more concrete about the needs that developers had that fell under usability..
Again, this is in the paper. So I want to go back and talk about each of these themes ..



Usability Themes

Theme	Description	PRD	EFE	SAT	EFI	TRS
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.	✅	✅		✅	

...because it was through tying them back to the transcripts and the screen recordings we were able to create both the implications and the refinements to our theory, so they encompass much of that spirit.

Starting from the top, we noticed that developers need help understanding the tool's capabilities throughout many of their different interactions with it.

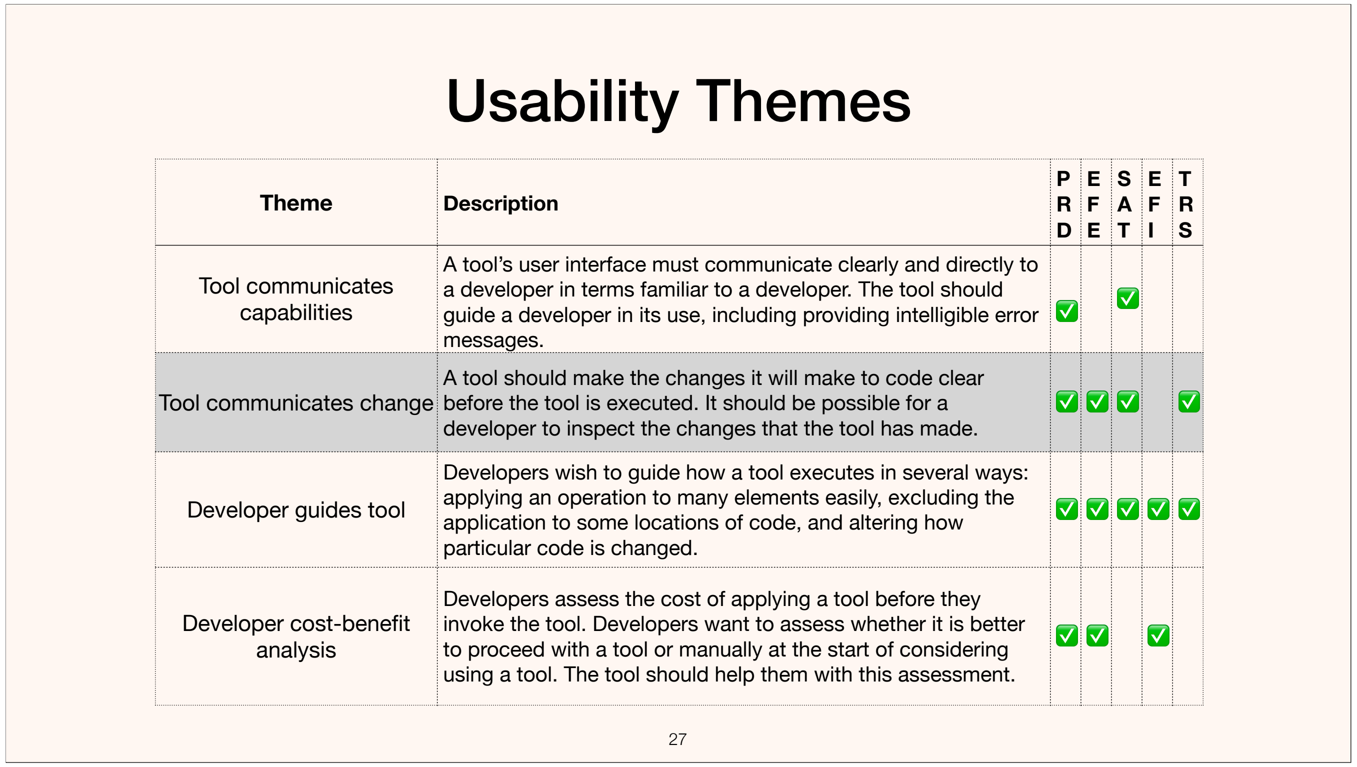
This includes the tool's affordances, such as when can a user expect to be able to invoke it, how do they invoke it, and is there anything they must do first, such as locating the right type of code element or making a certain selection.

For example to use move method, is it possible to select many methods and invoke move (which several users did), or must first invoke move, then select methods from a dialogue (move static method works this way), or do I need to invoke it on the method declaration itself (that is how move instance method woks). We did not see any way that the tool helped with this and participants concluded, after failing a few tries, that the tool was not capable of working in this case.

Furthermore, if the tool identifies a precondition violation, it will often provide some error or warning. These have to be intelligible for developers, but they also need to communicate what the tool's capabilities are in that situation. We saw that even when developers understand the error message as it related to the code, they did not understand what it meant to them as users of the tool: is the tool saying that it cannot proceed, or that it can but there is a risk of behavior change?

We saw some developers immediately canceled and said that the tool was not capable of continuing, whilst other developers did apply refactorings despite problems -- like P9 who had compiler errors introduced -- and were quite satisfied with it.

Finally, when a warning or error is accompanied with a proposed "alternative actions", say, when trying to move an instance method without callers, the tool might suggest “do you want to make this instance method static and move it”, but there was no information about whether these alternatives were equivalent to the original refactoring the developer invoked. We saw developers who misunderstood this and thought the tool was capable of doing the refactoring in this different way, and introduced behavior changes in their code.



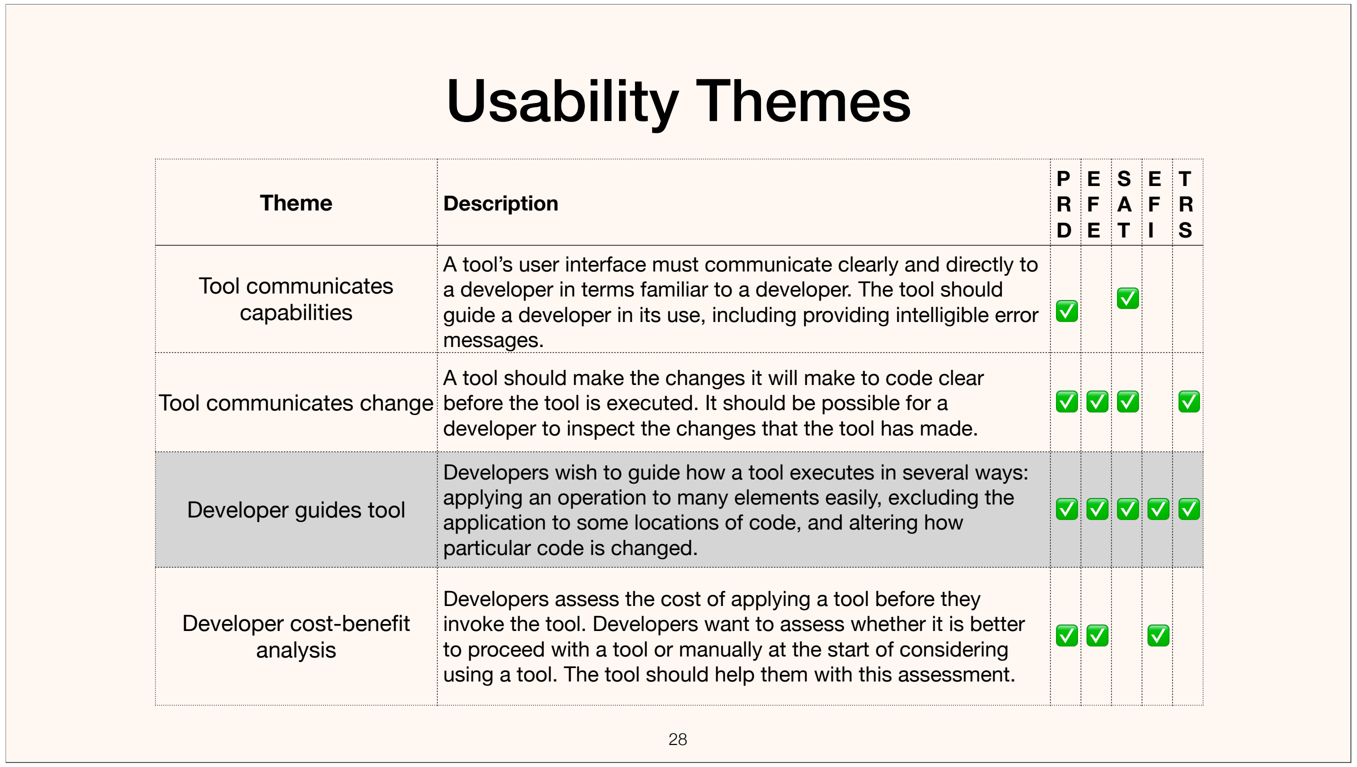
Usability Themes

Theme	Description	PRD	EFE	SAT	EFI	TRS
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.	✅	✅		✅	

Secondly, the tool has to communicate the change. This sounds like a preview window, but this relates to both before and after the applications. Often we heard developers exclaim, what happened? Did anything change? And we found that a surprising number of our participants resorted to either git diff or rapidly flipping between undo-redo to visually animate what had changed in their code. In a few cases, participants did not use any such techniques to locate changes, and actually failed to recognize changes that the tool had applied which later made problems for them.

This was in spite of interacting with the preview window when they applied the tool. So we saw a need for better communication of the change, and a need to be able to look at the change at a later point.

This is especially relevant when the tool makes “invisible” changes that are distributed across files, and the developer would not move on to work in the other file until later, then despite inspecting the preview view at the time he applied it, he wasn’t so to say, mentally ready to interpret what that code change meant to his workflow, or what the implications were. It was only once they moved on to this file, that they realised they, perhaps regretted the application.



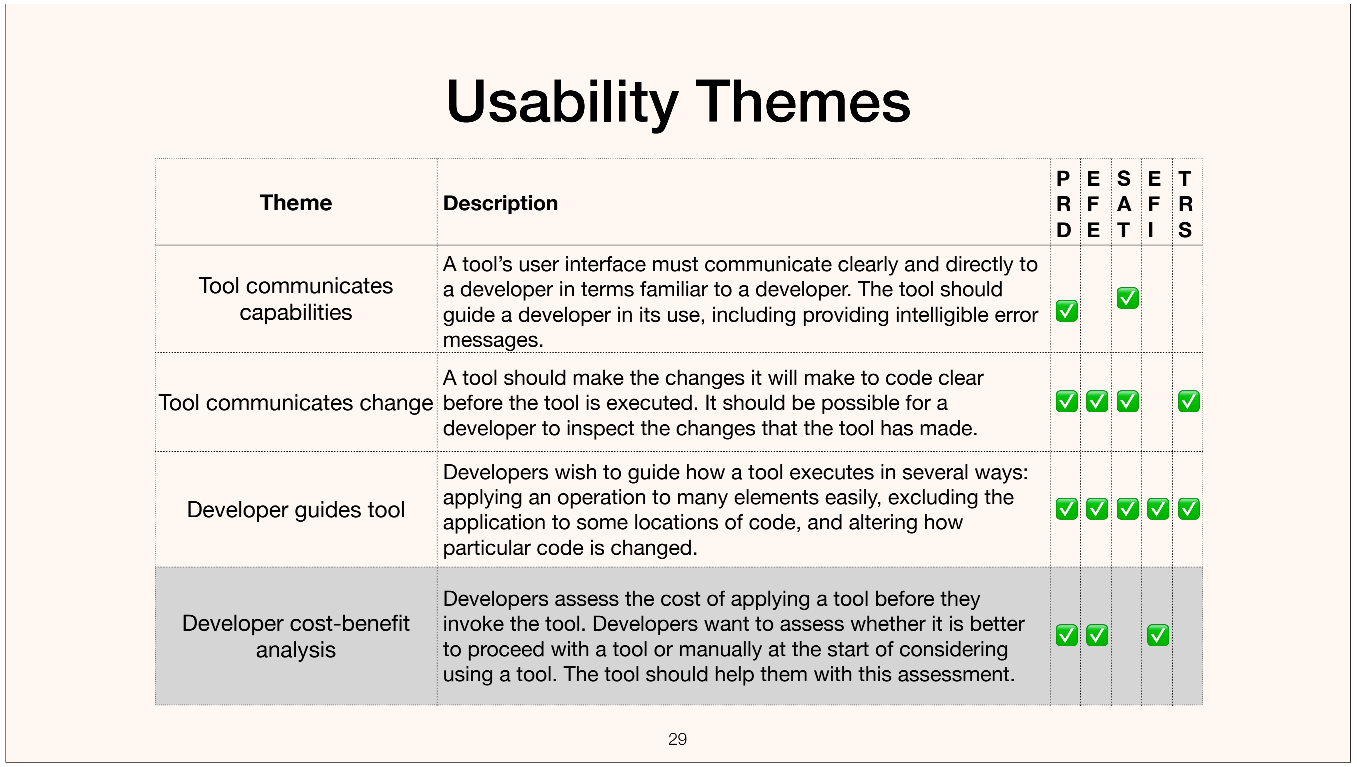
Usability Themes

Theme	Description	PRD	EFE	SAT	EFI	TRS
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.	✅	✅		✅	

Thirdly, the participants wanted not only to be aware of the change, but in some cases to guide it.

For example, they wanted to limit the impact in certain areas, such as test files to avoid changing tests and functional code at the same time. And they did not necessarily want to introduce compile errors where they were excluding: a use case we saw deterred many of the most experienced users from applying tools, was the need to separate functional code and test code. They wanted to control the impacts of the tool, and it gave them only the options to also change callers in test code, or to break callers in test code. It was important for them to make the changes stepwise, in one file at a time, and this led them to avoid tools despite initially trying them.

Participants also wanted to step through the locations that were changed and make additional changes or readability improvements in them. This emphasis on improving the code was justified by their newly acquired ownership of the code: when they changed it, it was now “theirs”, it had to undergo code review, and it is a good time to make sure it was of high quality.



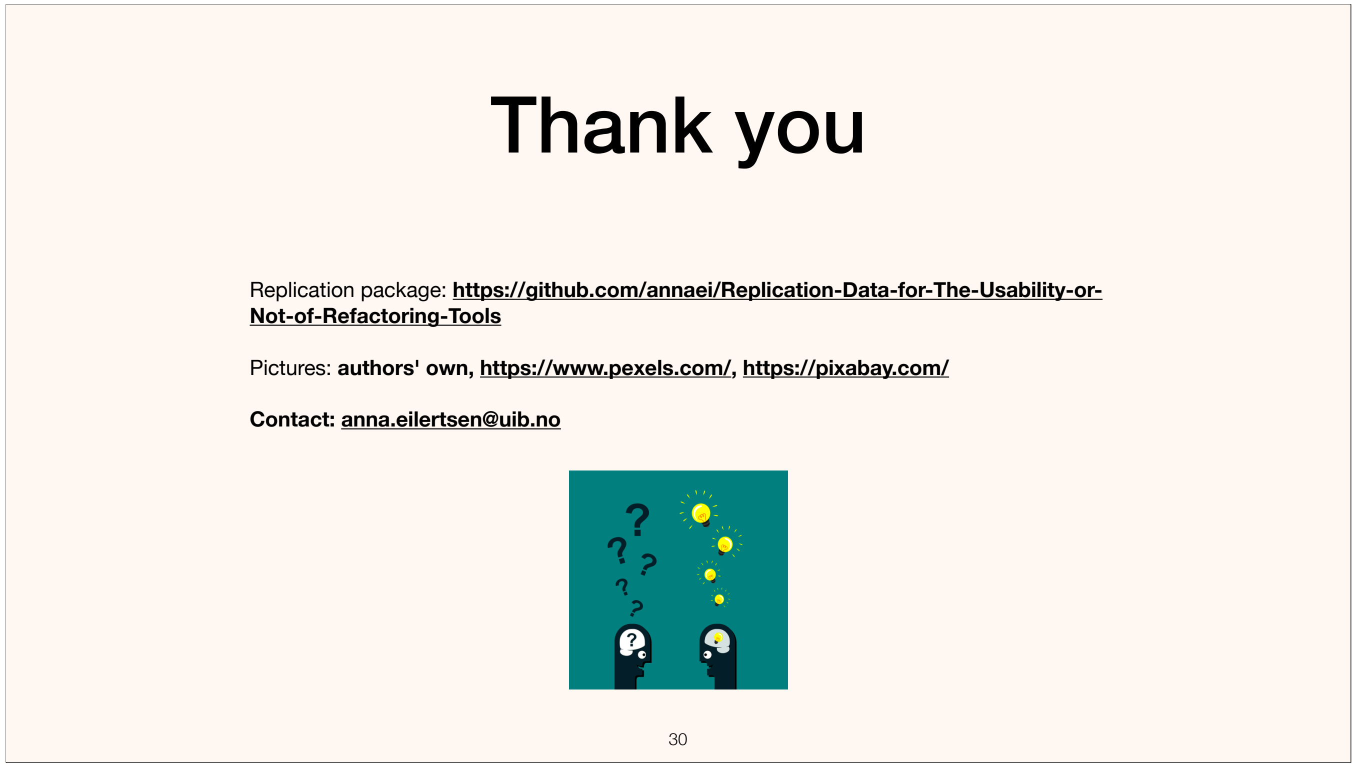
Usability Themes

Theme	Description	PRD	EFE	SAT	EFI	TRS
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.	✅	✅		✅	

Finally, we learn that developers constantly assess the cost and benefit of using a tool, and especially for more complex situations this assessment becomes important. They evaluate how much work is the tool going to save them, how much extra work is it going to be to use a tool, and how much work would it be to simply make the change manually.

An interesting insight here, is that the tool is always competing with the manual process, and not only with that, but with the developers perception and sense of safety in the progression of their manual process.

We saw that if developers were in doubt about the value of using the tool, or feared that they would lose time by using, they opted to make the change manually because they knew that they could, and they could estimate how fast they would be.



Thank you

Replication package: **https://github.com/annaei/Replication-Data-for-The-Usability-or-Not-of-Refactoring-Tools**

Pictures: **authors' own, https://www.pexels.com/, https://pixabay.com/**

Contact: anna.eilertsen@uib.no

I defer to the paper for the rest: in summary, we present a refined theory for refactoring tool usability which we think can help toolmakers create more useful tools that may or may not be traditional refactoring tools.

Thank you for attending my talk, I'll take questions now.

Implications for toolmakers

Up-front impact
Guided change
Communicating Change
Initiating Use

Upfront Impact

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

Upfront Impact

It is all right, it is useful I think when you are changing a lot of places. In some cases, it was not so useful because it was as much work to run that as to go and change those places.

- P8

Upfront Impact

Maybe I should have done this before deleting the parameter cause now it's going to be hard to look at the tests and know what they were doing before. Oops.

I: Did you expect the refactoring you used to do this?

Yes. I just didn't think far enough ahead to think if this is what I wanted.

- P9, T3

Yeah so I guess my previous goto for this kind of thing would be to know that this is what I need to remove because the compiler is complaining but because this kind of tool automatically did a lot more for me it kind of disrupted my flow.

- P4, T3

Guided Change

> I would expect the tool to guide me through the code so I can double check the code and see how I can improve it.
>
> If the tool is able to pinpoint me to all the location it is touching so I can backtrack to these places later that's fine. I don't want a button that just says it's done and then I don't know where unless I do a git diff or something like that.

- P6

Guided Change

...If I can introduce a default parameter to all call sites. Maybe I have to go through and see what the real value is at each call site but the tool has identified the call sites for me and has injected a marker for you need to do some work here.

...if I do a transformation and there's obvious incomplete stub-bits around then I see that the tool is behaving predictably - I think that is key - and I see where human attention is required.

- P9, T3

Communicating Change

> sometimes moving or extracting - you can't always see what happens, it's just here and you can't see what changed. I guess I can use git, but sometimes it is better to just plow through and see that this thing is broken.
>
> **- P8, T1**

Communicating Change

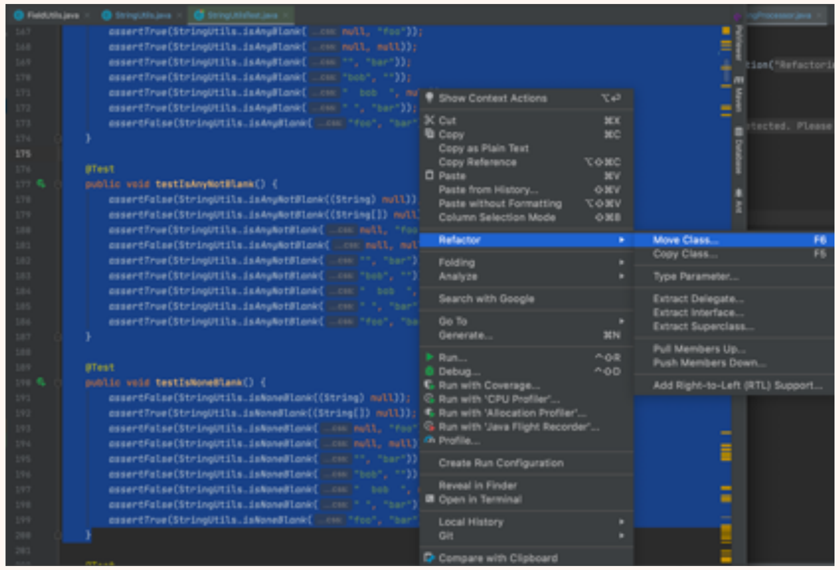
> I don't want to rely on a tool that will not help me become familiar with the source code. I'm quite certain that if I had to change this file again I would do it much faster because I started to see these dependencies. For example that thing casting a generic object to a field object, and if a tool does not give me that level of detail, when I try to use it to change the code it is not reliable.
>
> **- P6**

Communicating Change

Sometimes inspecting changes after the fact can be hard. There is going to be code that is unrelated to your refactoring, since normally refactoring tasks is done as part of other tasks.

- P19

Initiating Use



```
        assertTrue(StringUtils.isAnyBlank( ...css: null, "foo"));
        assertTrue(StringUtils.isAnyBlank( ...css: null, null));
        assertTrue(StringUtils.isAnyBlank( ...css: "", "bar"));
        assertTrue(StringUtils.isAnyBlank( ...css: "bob", ""));
        assertTrue(StringUtils.isAnyBlank( ...css: "  bob  ", nu
        assertTrue(StringUtils.isAnyBlank( ...css: " ", "bar"));
        assertFalse(StringUtils.isAnyBlank( ...css: "foo", "bar"
    }

    @Test
    public void testIsAnyNotBlank() {
        assertFalse(StringUtils.isAnyNotBlank((String) null))
        assertFalse(StringUtils.isAnyNotBlank((String[]) null
        assertTrue(StringUtils.isAnyNotBlank( ...css: null, "foo
        assertFalse(StringUtils.isAnyNotBlank( ...css: null, nul
        assertTrue(StringUtils.isAnyNotBlank( ...css: "", "bar")
        assertTrue(StringUtils.isAnyNotBlank( ...css: "bob", "")
        assertTrue(StringUtils.isAnyNotBlank( ...css: "  bob  ",
        assertTrue(StringUtils.isAnyNotBlank( ...css: " ", "bar"
        assertTrue(StringUtils.isAnyNotBlank( ...css: "foo", "ba
    }

    @Test
    public void testIsNoneBlank() {
        assertFalse(StringUtils.isNoneBlank((String) null));
        assertTrue(StringUtils.isNoneBlank((String[]) null));
        assertFalse(StringUtils.isNoneBlank( ...css: null, "foo"
        assertFalse(StringUtils.isNoneBlank( ...css: null, null)
        assertFalse(StringUtils.isNoneBlank( ...css: "", "bar"))
        assertFalse(StringUtils.isNoneBlank( ...css: "bob", ""))
        assertFalse(StringUtils.isNoneBlank( ...css: "  bob  ", 
        assertFalse(StringUtils.isNoneBlank( ...css: " ", "bar")
        assertTrue(StringUtils.isNoneBlank( ...css: "foo", "bar"
    }

    @Test
```

```
tion("Refactori
tected. Please
```

Implications for toolmakers

Up-front impact

Guided change

Communicating Change

Initiating Use