Proceedings of the Federated Conference on Computer Science and Information Systems pp. 411–420
DOI: 10.15439/2019F161
ISSN 2300-5963 ACSIS, Vol. 18

Supporting Source Code Annotations with Metadata-Aware Development Environment

Ján Juhár
Department of Computers and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
Email: jan.juhar@tuke.sk

***Abstract*—To augment source code with high-level metadata with the intent to facilitate program comprehension, a programmer can use annotations. There are several types of annotations: either those put directly in the code or external ones. Each type comes with a unique workflow and inherent limitations. In this paper, we present a tool providing uniform annotation process, which also adds custom metadata-awareness for an industrial IDE. We also report an experiment in which we sought whether the created annotating support helps programmers to annotate code with comments faster and more consistently. The experiment showed that with the tool the annotating consistency was significantly higher but also that the increase in annotating speed was not statistically significant.**

I. Introduction

THE MAIN hindrance programmers deal with when they need to comprehend source code is known as the *abstraction gap*. This gap exists between the problem domain and the solution domain of a given software system. Many high-level concerns from the problem domain are either lost or scattered as programmers transform them to code. As argued by LaToza *et al.* [1] and Vranić *et al.* [2], programmers often ask questions about the *intent* behind particular source code fragments. In this paper, we present and evaluate an approach for helping to preserve the high-level knowledge within source code annotations.

A. Motivation

Two general approaches for retrieving information otherwise lost or scattered in source code are available:

- *recovery* of pieces of high-level information from the code by means of reverse engineering, and
- *preservation* of a programmer's thoughts and intentions in software artifacts.

Feature location tools from the recovering approach usually produce list of source code elements that are evaluated as relevant to the given feature [3]. Preserving approaches directly assign high-level information to source code elements with annotations [4], [5]. Although in both cases the retrievable data represent *source code metadata* (abbreviated: *metadata*), they are of different nature.

This work was supported by project VEGA No. 1/0762/19: "Interactive pattern-driven language development" and grant of FEEI TUKE no. FEI-2018-55: "Methods of code classification based on knowledge profiles".

Recovering approaches use *intrinsic metadata*, which either define source code elements themselves or can be derived from these elements. In contrast to them, preserving approaches focus on *extrinsic metadata*, which complement the intrinsic ones by adding custom, high-level details explicitly recorded by programmers. On one side, the more accurate preserved knowledge may help to bridge the abstraction gap better than the lower-level recovered one. On the other side, recording programmer's mental model of the code brings in an additional cost: the programmer must spend extra time to record it.

The immediate availability of intrinsic metadata makes them a great choice for code analysing tools in integrated development environments (IDEs) [6]. These can provide *structure-aware* visualizations (e.g., file structure browsers, semantic code highlighting, linting) and actions (e.g., contextual code completion, refactoring). However, intrinsic metadata also restrict these tools to lower-level domains.

It is thus a worthwhile question whether adding the metadata upfront will be too costly compared to any benefits they may bring later. Sulír *et al.* show in [4] how concern metadata in the form of Java annotations can enable rapid construction of reader's mental model of the implementation. Report of Ji *et al.* [7] shows that presence of feature-related metadata within source code comments was beneficial for software product line development. The authors presume that by employing a supporting tool the benefits can be further increased.

To tackle the tool support for such custom metadata in the source code we need to consider both the type of annotations that can be used and granularity of elements where they can be used. Ideally, a programmer would not have to consider all the different ways in which metadata can be bound to the code, but directly *express the intention* to bind metadata to specific source code elements and let a tool to perform the binding. This is the main motivational factor for the work presented in this paper, proposed through the idea of uniform annotation process in a metadata-aware development environment.

B. Goal

The goal of this paper is twofold. First, we present the idea and prototype implementation of the uniform annotation process in an integrated development environment (IDE) extended by metadata-awareness. By IDE metadata-awareness we mean the ability to work with both custom code-bound metadata and

with annotations that bind them as with first-class source code elements. This goal is addressed in Section II.

Second, in sections III and IV we report an experiment we performed to evaluate the effect our prototype tool has on annotating speed and consistency of comment annotations created during code annotating task. An overarching research question for the experiment is *"Does metadata-aware IDE help programmers to annotate code with comment annotations?"*.

II. Annotation Process in a Metadata-Aware Development Environment

IDE tools are adapted to use intrinsic metadata derived from code elements. They can easily bind them to the originating elements and build dynamic code views or projections from them [6], [8]. Extrinsic metadata are available also but mostly limited to data from version control and bug tracking systems. As such, they are bound only to files, or lines of text. We can achieve more specific bindings with source code annotations.

In our work the term *source code annotation* (abbreviated: *annotation*) has a more general meaning than, e.g., *Java* annotation. As per Definition 1, we consider any binding of metadata to source code element as annotation.

Definition 1. *Annotations are in-place or addressing bindings of custom metadata to source code elements.*

A development environment able to utilize the metadata recorded by code authors may provide program comprehension support on a higher level of abstraction, closer to the problem domain of a software system. Our idea of such *metadata-aware development environment* (MADE) comprises of three following aspects:

1) Support for the annotation process, during which a programmer binds metadata to code elements.
2) Preservation of annotations and metadata as code changes.
3) Utilization of the metadata in various IDE tools to facilitate program comprehension.

In the work presented in this paper, we focus on the first aspect: on supporting the *annotation process*, which we define in Definition 2.

Definition 2. *Annotation process, or annotating, is a process in which metadata are being bound to code elements.*

A. Types of Source Code Annotations

When faced with a task to annotate code, a programmer has three following types of annotations to chose from:

- Internal annotations contained within the source code files, further classifiable into two distinct types:
  - *Language-level annotations* (LLAs), which use native programming language constructs for metadata.
  - *Structured comment annotations* (SCAs), which give the "metadata" status to code comments.
- *External annotations* (EAs), which are created with a supporting tool and bound to the source code by addressing the annotated elements.

Each of these annotation types has a different set of inherent limitations, which we discuss in the following.

1) Language-level Annotations: LLAs are formally defined in language's grammar and all standard language tools can work with them. On the other side, they can be used only if the language itself does support them, and only on elements where it supports them. An example of applying custom metadata with LLAs in *Java* language is given in Listing 1.

Listing 1. *Java* annotations as high-level metadata

```
@NoteChange @TagManagement
public void addTag(String tag) { /* ... */ }
```

2) Structured Comment Annotations: SCAs reuse general code comments, which can contain arbitrary text. For that reason we need to define a specific syntax for them that would allow us to parse the metadata. Listing 2 shows an example of such syntax. Such annotations can be used in almost any language, considering that a comment can be put at the desired place in the code. But they require supporting tool that can recognize the metadata and bind them to specific code elements.

Listing 2. High-level metadata in structured comment annotation

```
// [# note change ] [# tag management ]
public void addTag(String tag) { /* ... */ }
```

3) External Annotations: EAs are superimposed over the code by means of an addressing mechanism that locates annotated elements. The mechanism can use simple addresses like element's starting and ending offsets within a file, or more robust descriptors of code elements [9]. Annotations are usually visualized in the code editor (see Fig. 1).

The most significant advantage of EAs is that arbitrary code fragments can be annotated, even inside files the programmer cannot (or does not want to) modify. On the other side, their addresses need to be kept in sync with changes made to the code and they are completely dependent on a supporting tool.

B. Supporting the Annotation Process

Our focus on custom extrinsic metadata allows us to assume that annotating is going to be performed "manually" by programmers. Their goal may be to capture their mental model of the code in a form that can be used by tools and can help future maintainers of the code. In the design of a tool

Fig. 1. Metadata bound to code fragment through external annotations displayed in the editor's gutter

supporting such annotation process, we should strive to remove unnecessary distractions from programmers' primary goal of annotating. For us, it primarily means that regardless of which type of annotation is used the workflow should be the same.

However, annotation of a given type differs from other types in how exactly it is applied to the code and what conditions must be met before it can be applied. The most important differences are the following.

- *Definition*: LLAs are represented by language elements, which may need to be defined before they can be applied (e.g., like *Java*'s annotation types). Similarly, EAs may require definition through a tool [10]. SCAs have no single definition of the source metadata and the programmer needs to maintain them individually in each comment.
- *Application*: Internal annotations must be typed[1] into the code at the appropriate place. EAs are applied through environment's UI and their application may be preceded by code selection.
- *Binding*: LLAs are bound to specific elements in-place according to language's grammar. EAs use addressing bindings, which may be text-level or element-level. And comments, as free-standing statements, have no bindings to the surrounding code elements.

To deal with these differences, we designed an *abstracted annotation process*, which by itself does not require any changes to the annotated code and imitates annotating the code with EAs. In this process, operations required to annotate code fragments[2] should be performed through IDE actions. The actions cover selecting code fragment for annotation and selecting annotation representing required metadata. When these are selected, the annotation should be applied automatically with the configured annotation type.

We implemented a prototype tool called *Connotator*[3], which supports the abstracted annotation process and its configuration per project. The tool is implemented as a plug-in for *JetBrains IntelliJ* platform-based IDEs. The current implementation supports all three annotation types for the *Java* language, and SCAs and EAs for languages *Kotlin* and *Python*. It also provides *source code editor augmentations* [11] related to code annotations. In the following, we describe the annotation process and its realization in *Connotator* in more detail.

1) Defining metadata annotations: All annotations representing custom metadata are in *Connotator* managed through the main tool window (see Fig. 2). Annotations defined there may be applied to the code. The metadata model is currently rather simple: annotations are defined only by their names and optionally they can have a parent annotation. The tool supports annotation name refactoring, which appropriately updates all their existing instances in the code.

2) Selecting annotatable code fragment: Only valid code selections—those matching some AST elements—are meaningful targets for metadata. In general, fine-grained element selection (like statements and expressions) should be possible, but the specific set of annotatable elements should be configurable per-project. These requirements are in *Connotator* met through code fragment selection facility using tree patterns matched against the PSI tree[4] of code elements. A user can specify these patterns as *XPath*-like expressions built from a set of basic element types. For example, the following expression can be used to match *Java* statements without block bodies:

```
codeBlock/statement[not child::blockStatement]
```

To select elements for annotation, the user uses fragment-selecting action (with keyboard shortcut or from menus). The action resolves annotatable elements from current text selection or caret position in the code editor, as can be seen in Fig. 3(a).

3) Applying annotations: Existing metadata annotation can be applied on selected code fragment with dedicated action that allows the user to specify the annotation. Its usage is shown in Fig. 3(b). Once the annotation is selected, *Connotator* finishes code annotation automatically. It selects the annotation type to use from the tool's configuration (it can be specified separately for each type of annotatable elements) or selects one automatically based on their availability and predefined priority (LLA > SCA > EA). Fig. 3(c) shows the result of applying an annotation on class fields when *Java* LLAs are configured for their element type. Note that the tool also generates required *Java* annotation types if they do not exist yet.

The same annotation process is applicable for any annotation type; the only difference is in the final alteration of the code. As an example, Fig. 4 shows a `for` statement annotated with SCA. All the source code editor augmentations, like gutter icon and annotation highlighting, remain the same. An EA would differ only by no visible annotation inserted into the code.

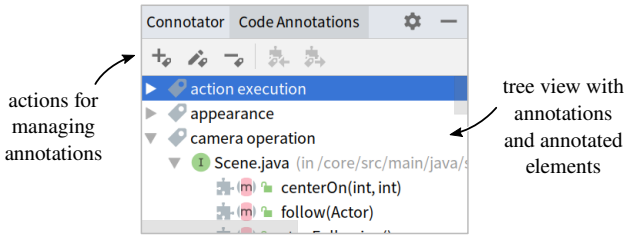

Fig. 2. Annotations tool window.

C. Binding Comments to Code Elements

Going back to the differences among annotation types, the one we did not discuss so far is *binding*. As far as LLAs and EAs are concerned, the binding is defined either by the language's grammar or by specific addressing mechanism used by the tool supporting EAs[5]. On the other hand, SCAs do not have any grammar-based or other bindings to surrounding code

[1]LLAs may take advantage of already existing IDE support like code completion, but some typing is still involved.

[2]We use the term code fragment in a sense of one or more consecutive elements selected for annotation.

[3]*Connotator* is available at https://git.kpi.fei.tuke.sk/jan.juhar/connotator.

[4]PSI (Program Structure Interface) tree is a version of concrete syntax tree backing most structure-aware features of the *IntelliJ* platform [12].

[5]In *Connotator*, we currently use just a very basic offset-based addressing. A more robust solution is out of scope of here presented work.

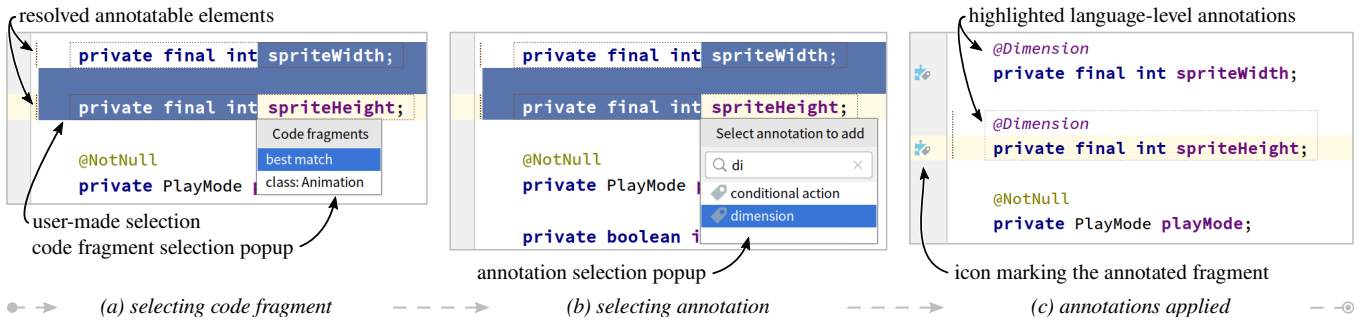

Fig. 3. Annotation process in *Connotator*. **(a)** The user selects code and uses action to resolve annotatable elements. **(b)** The user uses action to apply annotation to the selection and then selects desired annotation. **(c)** The tool applies *Java* LLAs because this type is configured for class fields in the project.

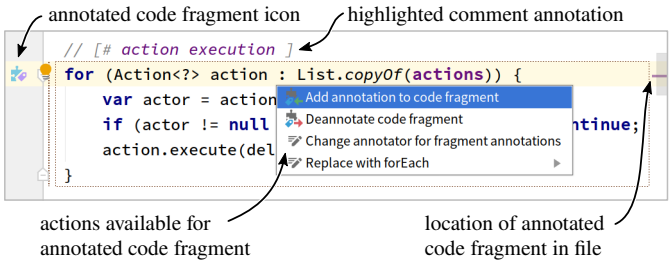

Fig. 4. IDE editor showing annotated block statement with associated actions and source code editor augmentations.

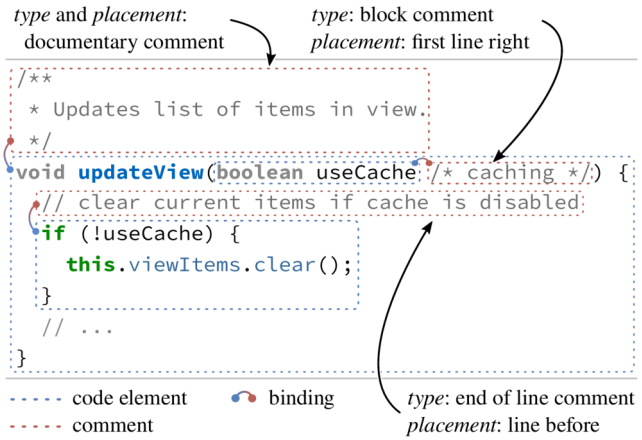

Fig. 5. Comment placements and types with bound code elements

elements; in fact, compilers and interpreters for most languages ignore comments already in the lexing phase.

However, IDEs need to know about every concrete syntax tree token in order to be able to map each character from the file to the corresponding parsed node, and *vice versa*. For this reason they use custom lexers and parsers that preserve comments [12]. Nevertheless, only comment's parent can be determined from the tree (e.g., a comment inside a method), which is not enough to unambiguously assign comments to code elements. Does a comment standing alone on a first line within a method relate to the method (parent) or to a statement below? One possibility for dealing with such ambiguities is to deploy a set of conventions, or rules, to resolve them.

In the design of comment-to-element binding rules for a tool that needs to be able to find comments in the existing code, as well as to generate them when elements are annotated, we need to consider their following two properties:

- *Placement* of comments relative to elements they are bound to. There are many such relative placements that can be supported; see, e.g., the work of Sommerlad *et al.* [13] or our examples in Fig. 5.
- *Type* of comments that should be used. Particularly *end of line* and *block* comments are often supported by languages, sometimes complemented by conventional format of *documentary comment* (as *Java*'s *JavaDoc*).

Examples of possible relative comment-to-element placements and comment types are in Fig. 5. In the following, we describe how these placements and comment types are interpreted by *Connotator*, which supports their configuration for each annotatable element type.

- *End of line* comment is bound to the `if` statement on the next line through the *line before* relative placement.
- *Block comment* is bound to the method parameter `useCache` through the *first line right* relative placement. The *block* type of comment is required in this context because the bound element is followed by more tokens.
- *Documentary comment* is is bound to the method `updateView` through the equally named *documentary comment* placement. This placement expresses the intention to bound the conventional documentary comment to the element it documents.

Because the user can manually insert a comment annotation at invalid position (such that does not bind it to any annotatable element), *Connotator* is able to issue warnings through in-editor highlighting when such comment is found.

III. Experiment: Annotating Code with Comment Annotations

Our main motivation behind the presented tool support for annotation process is in reducing overhead of annotating a source code. To evaluate any effects our prototype tool *Connotator* can have in this regard, we prepared a task in which a portion of a selected application's code base needs to be annotated with comments containing high-level concern

metadata. We specify what kinds of code elements can be annotated and where should the annotating comments be placed relatively to the annotatable elements: together, we will call this set of constraints the *annotation rules*.

In this section we present an experiment in which we compare two groups of students-programmers performing the described task. The first group uses *IntelliJ IDEA* with *Connotator* installed and the second, control group, uses *IntelliJ IDEA* in its default setup. We formulate the following two research questions for the experiment.

RQ 1. *Do programmers annotate source code with comments more consistently with defined annotation rules if they are guided by a metadata-aware tool that adheres to these rules?*

RQ 2. *Do programmers annotate source code with comments more quickly if the development environment is aware of the comment annotations?*

A. Hypotheses

In the experiment we focus on two aspects of the annotation process: comment annotations *placement consistency* and *annotating speed*. In the following, we formulate related hypotheses. The goal of the experiment is to statistically test those hypotheses with a confidence level of 95% ($\alpha = 5\%$).

1) Comment Annotations Placement Consistency: In *Connotator*, comment annotations placement rules are used to bind SCAs to specific code elements. We hypothesize that *Connotator* will help to *increase* comment annotations *placement consistency* in comparison to manual annotating. We define placement consistency (*PC*) of comment annotations as

$$PC = \frac{N_C}{N_A} \times 100\%$$

where N_C is the number of correctly (according to the annotation rules) placed comment annotations, and N_A is the number of all comment annotations placed during the annotating task. We formulate the following null and alternative hypotheses for RQ 1:

- $H1_{null}$: The **placement consistency** of comment annotations created during code annotation task with *Connotator* **is equal (=)** to the placement consistency of comment annotations created during the same annotation task with the standard *IntelliJ IDEA* setup.
- $H1_{alt}$: The **placement consistency** of comment annotations created during code annotation task with *Connotator* **is higher (>)** than the placement consistency of comment annotations created during the same annotation task with the standard *IntelliJ IDEA* setup.

2) Annotating Speed: Connotator abstracts the annotation process for different annotation types into a set of IDE actions. We hypothesize that these actions, when combined with the annotation rules for annotatable elements, will *increase annotating speed* of programmers performing code annotation task. We define annotating speed (*AS*) as

$$AS = \frac{N_A}{t}$$

where N_A is the number of all comment annotations placed in the code during the annotating task and t is the total time needed to complete the task. We formulate the following null and alternative hypotheses for RQ 2:

- $H2_{null}$: The **annotating speed** during code annotation task with *Connotator* **is equal (=)** to the annotating speed during the same annotation task with the standard *IntelliJ IDEA* setup.
- $H2_{alt}$: The **annotating speed** during code annotation task with *Connotator* **is higher (>)** than the annotating speed during the same annotation task with the standard *IntelliJ IDEA* setup.

B. Setup

1) Participants: Participants of the experiment were 36 bachelor's degree Computer Science students from our department. They were in their fourth semester with programming courses and were familiar at least with languages *C* and *Java*. These students formed two study groups (not equally sized) of the Component Programming course, in which the *Java* language is used. One group of students was chosen as the experimental group where *Connotator* was used: we will call it the *Connotator* group. The other group of students was used as the control group working with standard *IntelliJ IDEA* IDE setup: the *manual* group. The *Connotator* group contained 20 participants and the *manual* group 16.

2) Code for Annotation: As a target for the annotating task we used source code of application for managing notes for bibliographic entries called EasyNotes.[6] It is a small-scale (about 2700 lines of code) project written in *Java*. An advantage of its code base is the presence of high-level concern annotations in a form of *Java* annotations, which were added by its author for the purpose of the study performed by Sulír *et al.* [4]. This provided us a very good starting point for preparing our own annotating task. From 25 available concern annotation types, we selected 10 that covered a large portion of the application's domain logic, its data model and persistence. We left out all the code directly related to the graphical user interface because it contained more complicated code generated by a UI designing application.

However, high-level concerns may be difficult to recognize in an unfamiliar code base. This difficulty of program comprehension represents the main confounding factor for our experiment because it can negatively affect the annotation speed (our dependent variable) and correctness of *contextual* placement of annotations (which we do not evaluate). Our attempt to minimize the influence of this factor was to try and bring the annotating close to a mechanical process, not unlike one performed by a programmer who is already familiar with the code. For this purpose, we renamed several identifiers to names that included some form of the relevant concern name.

3) Format of Comment Annotations: The two groups of participants did not use use exactly the same comment structure

[6]Source code of the application *EasyNotes* is available at https://github.com/MilanNosal/easy-notes.

for annotations. As shown in Fig. 4, *Connotator* uses specific comment annotation format where the annotation name needs to be placed between prefix `[#` and suffix `]` within the text of a comment. These additional symbols are, however, meaningless without the tool and they would pose unnecessary hindrance for participants working without *Connotator*. For this reason, participants in the control group did use only simple prefix `#` before annotation name in comments to clearly designate that the following text is meant to be an annotation.

4) Additional Materials: We prepared two documents for study participants: annotation rules they need to follow during the task and a user guide for *Connotator*. We kept these documents short so they would fit each on one sheet of paper.

The document with annotation rules was designed to guide participants through the task of annotating EasyNotes' code. It described the form of comment annotations, their possible relative placements, and paired each type of annotatable element with required comment placement (for the last see Table I). The document was concluded with a table of actual annotations the participants should use. For each of the 10 annotations it specified words related to high-level concerns that could be found in element identifiers. It also explained the high-level meaning of each annotation. For example, for concern *citing* the table listed *"citing of publications"* as its explanation and *"cite, citation, publication"* as related words in identifiers.

In addition to the annotation rules, we provided participants in the *Connotator* group with a brief user guide of the tool. This guide explained the role of the annotations panel and the available workflows to select, annotate and deannotate code fragments. The guide also presented the *Connotator*'s ways of signalling through code highlighting whether a specific comment annotation is considered to be valid or invalid according to the configuration.

5) Environment: The experiment took place in our department's software laboratory room containing 20 computers with widescreen full HD displays and *IntelliJ IDEA* 2017.3.5 installed on Windows 10 OS. We also set up a screen recording application to record the participants' annotating sessions for extracting the task completion times and for later analysis of their performance. We informed participants that their session was going to be recorded.

C. Procedure

We carried out the experiment in two separate sessions, each for one group of participants and lasting 90 minutes (the duration of a lab lesson). Each session proceeded as follows.

When the participants came for the experiment into the laboratory room, they already had their environment prepared: *IntelliJ IDEA* was running with the EasyNotes project opened and the screen recording was started. The *Connotator* group had the tool configured in accordance with the annotation rules.

First, the experimenter—the author of this paper—introduced the concept of annotating source code with high-level concerns. Then he presented the EasyNotes application, explaining its purpose and demoing its functionality.

In the next step, the experimenter handed over printouts of prepared materials, each labeled with a unique participant number. Then he walked the participants through its individual sections: form of comment annotations, possible comment placements, the comment annotation placements rules and the annotations themselves. For the *Connotator* group, the experimenter then covered the *Connotator* user guide complemented by presentation of its usage.

Next, the participants were asked to proceed with the task. They were also asked to minimize the IDE window and notify the experimenter when they finish. When each participant finished their task, he or she was asked to fill out a prepared questionnaire. At the end, the experimenter collected annotated projects and videos of annotating sessions.

TABLE I
ANNOTATION RULES FOR THE ANNOTATING TASK

Annotatable code fragment type	Comment annotation placement
Class Method Class field	Documentation comment
Simple statement Method parameter	First line right
Block statement	Line before

D. Evaluation

The first phase of evaluation consisted of analyses of captured videos, in which we needed to determine the actual duration of each participant's annotating task. Next, we proceeded with analysing annotated projects. The projects were analysed by counting created comment annotations. In every project annotated by an experiment participant we counted:

- Total number of annotations in comments.
- Number of invalid annotations, which were further categorized as comment annotation with:
  - *invalid placement*: annotations occurrences in comments that, given their placement in the code, did not annotate any element according to the annotation rules,
  - *invalid syntax*: comment annotations that did not use required syntax,
  - *invalid context*: comment annotations that annotated elements not related to the concerns expressed by these annotations.

E. Results

Our data samples were unpaired, as each participant was either in *Connotator* or in *manual* group. There was one independent variable—availability of the *Connotator* tool—with nominal values "available" and "unavailable". Our dependent variables were comment annotations placement consistency and annotating speed, none of which looked normally distributed. We used the *Mann-Whitney U test* as a statistical test for our hypotheses. We report the results in the following.

1) Comment Annotations Placement Consistency: The placement consistency was higher in the *Connotator* group. The mean value in this group was 99.59%, compared with the mean of 83.55% in the *manual* group. Only 3 comment annotations were placed incorrectly for the *Connotator* group (participants typed them manually and ignored the tool's warnings). On the other hand, *manual* group had 2 extreme outliers, who reached *PC* of only 46.43% and even 0.0%, respectively. If we exclude these two participants, the mean for the *manual* group rises to 92.17%. Statistical results are summarized as box plots in Fig. 6.

With or without the two extreme cases in the *manual* group, the computed p value is well below 0.001 (6.42×10^{-7} and 1.74×10^{-6} respectively), which is also below our significance level (0.05). Thus, we reject $\mathbf{H1_{null}}$ and accept $\mathbf{H1_{alt}}$. The conclusion is that comment annotations placement consistency was higher in the *Connotator* group and the result is statistically significant.

2) Annotating Speed: The annotating speed was also higher in the *Connotator* group, with the mean of 2.07 annotations per minute. The *manual* group reached the mean of 1.48 annotations per minute. See Fig. 7(a) for the box plot.

The computed p value for annotating speed is 0.11, which is above our significance level (0.05). Thus, we fail to reject $\mathbf{H2_{null}}$. The conclusion is that while annotating speed was higher in the *Connotator* group, the result is **not** statistically significant.

It is, however, interesting to note that when we consider just the duration of the annotating sessions (box plot shown in Fig. 7(b)), the difference between groups is more prominent: participants from the *Connotator* group finished their task in 42.64 minutes in average, while for *manual* group the average is 52.76 minutes. The computed p value for session durations is < 0.001. We discuss the possible reasons for this discrepancy between annotating speed and annotating session duration (among other observations) in Section IV.

F. Threats to Validity

In the following, we discuss threats to the validity of the experiment and relevant control actions taken.

a) Internal validity.: Assignment of participants into groups was not strictly random: we used existing groups of students, in which randomness is not guaranteed. The alternative was to randomly assign half of each study group to the experimental group and the other half to the control group. However, in such arrangement, the experimentator would need to present the annotating tool in front of participants assigned to the control group (they would be in the same room), which could also have an effect on the result.

Pilot-testing was limited to one participant who performed the annotating task with *Connotator*. At that time, 13 high-level annotations were selected. As we considered the time needed to reasonably complete the task too long, we decided to lower the count of annotations to 10.

b) External validity.: Participants of our experiment were students, not professional programmers. According to the findings of Salman *et al.* [14], it might not have great effects on the results, because the tested approach—the *Connotator* tool—is new for both students and professionals. Nevertheless, we would need code authors or maintainers (who would knew the project in detail) to eliminate the effect of program comprehension on the result. We attempted to eliminate this effect by making the task more mechanical, as described in Section III-B2.

Within the broader approach of using extrinsic metadata for program comprehension, we tested only its part—the annotation process. Without further integration of bound metadata into the IDE, their presence in annotations is not well utilized. However, the annotation process is the most time-consuming part of the approach, and we strive for a better supporting tool.

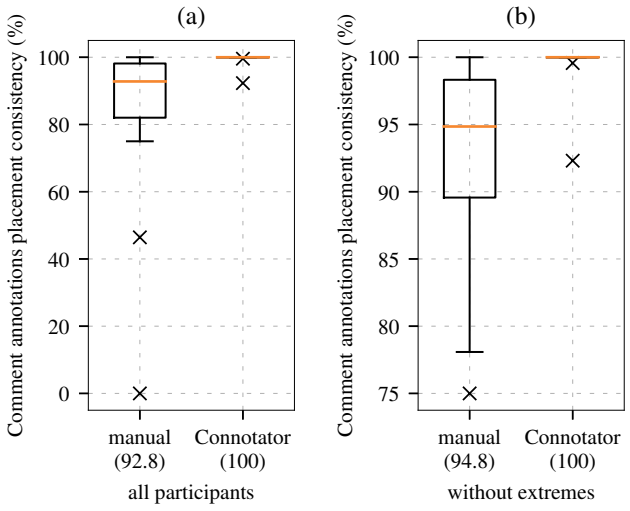

Fig. 6. The results of statistical evaluation of differences between groups of participants regarding their comment annotations placement consistency: **(a)** with all participants, **(b)** with the two outliers (0% and 46.4% PC) from the *manual* group removed. Median values are included below group names.

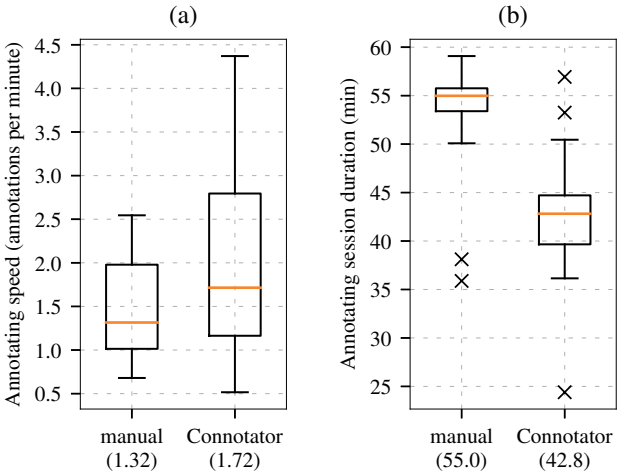

Fig. 7. The results of statistical evaluation of differences between groups of participants regarding **(a)** their annotating speed and **(b)** duration of their annotation sessions. Median values are included below group names.

c) Conclusion validity.: The most prominent threats to conclusion validity are the small number of subject participating in the experiment (36) and the confounding factor of needed comprehension of annotated source code.

IV. Discussion of the Experiment

In this section, we present observations regarding the data we obtained by analysing source codes annotated in the experiment.

A. Differences in Number of Created Annotations

Based on our version of annotated EasyNotes' source code, which itself was based on annotations from its author and refined to finer granularities allowed by SCAs, we consider 70 to 90 comment annotations for an optimal result of annotating. Participants annotated code on the basis of the annotations table that we provided them. Ultimately, the specific code elements—and their count thereof—that participants chose to annotate depended on their understanding of both the annotations and of the code. In processed projects, we saw that the numbers of annotations created during the task varied significantly.

From the box plot in Fig. 8, we can see that the *manual* group performed better with regard to the number of annotations. The median (72.5 annotations) is closer to our optimal count than the median of *Connotator* group (61.5 annotations) and there are no very low (<30) nor very high (>200) values.

The most frequently and most inconsistently used was annotation *domain entity*. Some participants took this annotation too broadly and annotated a majority of variables named `note` or `notes`. Interestingly, the extreme cases of such very general understanding of this annotation (almost 100 occurrences in the project) were present only in the *Connotator* group.

Described differences in number of created annotations, especially the tendency towards lower count of annotations in the *Connotator* group, may be behind the discrepancy between annotating speed and session duration distributions (see Fig. 7). Also, 3 participants in the *Connotator* group annotated code in less than 40% of files they were asked to annotate, in comparison to only one such participant in the *manual* group.

We conclude that in order to more reliably assess the effect of our tool on annotating speed, the task should more precisely define both the elements to be annotated and the determining factor of when the task can be considered as completed. This may reduce the variability in annotations counts or at least allow us to exclude clearly incomplete tasks.

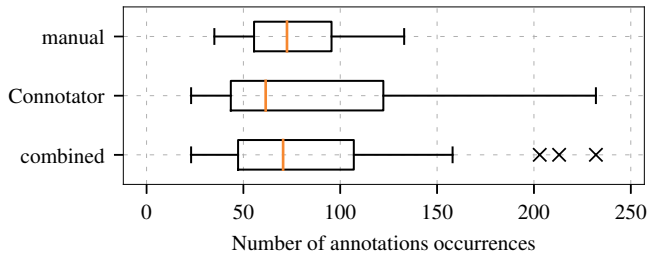

Fig. 8. Distributions of number of created comment annotations. Shows distributions for both groups and for all participants combined.

B. A Closer Look at Placement Consistency

Comment annotations placement consistency (*PC*) showed to be significantly higher in the *Connotator* group (see Section III-E1). This result may be not surprising as this group used tool that prevented *misplaced*[7] comment annotations. Our interest was, however, to find out how many misplaced annotations there would be in the *manual* group and whether the difference would be significant.

Fig. 9 shows absolute numbers of misplaced annotations in relation to all annotations created by individual participants. Only one participant in the *manual* group managed to make no placement mistakes, but he made totally only 35 annotations (the lowest number in the group).

C. Invalid Syntax or Context of Created Comment Annotations

There were 0 comments with invalid syntax in the *Connotator* group. *Manual* group made syntactic mistakes in 1.6% of comments in average, mainly by omitting the `#` prefix. Three participants in the *Connotator* group misspelled name of one annotation, which resulted in having them misspelled at every occurrence in the code. However, due to *Connotator*'s annotation renaming feature, such issue can easily be fixed. On the other hand, *manual* group had 2.6% of comments in average with misspelled annotation names. Without a supporting tool, such errors lead to inconsistencies that hinder the usage of such *comment tags* with common search tools [15].

As the tested annotation process does not influence program comprehension, we expected to found no difference in the numbers of contextually invalid annotations. This expectation was confirmed as these was only small and statistically insignificant difference: median values for percentages of contextually invalid annotations were 13.6% and 11.7% for *Connotator* and *manual* groups, respectively.

D. Observations from the Questionnaire

In the questionnaire we asked participants questions about how they would rate their understanding of annotations

[7]Misplaced comment annotation is an annotation within a comment that is not bound to any code element because its placement or comment type does not match annotation rules.

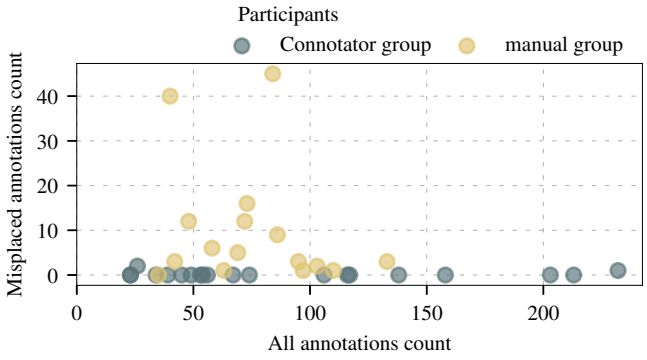

Fig. 9. Number of misplaced annotations of each participant.

meaning and of EasyNotes source code on a 5-point scale. Generally, participants expressed that they understood meanings of annotations and that they were able to comfortably navigate source code of EasyNotes. Differences between groups were minor, with marginally higher (i.e., better understanding) means in the *manual* group.

Next, we asked question regarding the annotation process, in which the participants answered as follows.

- 45% of participants from the *Connotator* group stated that annotating was simple and fast, compared with only 12.5% in the *manual* group.
- 43.8% of participants from the *manual* group considered annotating as laborious and 62.5% stated they copied existing comment annotations to create new ones.
- 90% of participants from the *Connotator* group considered the annotating tool as helpful.

We also asked how often they needed to check the document with annotation rules (on a 10-point scale from *"in 1 out of 10 cases"* to *"in 10 out of 10 cases"*). The responses are plotted in Fig. 10 and show that participants in the *Connotator* group reported less frequent usage of annotation rules (median of *3/10*) than in the *manual* group (with median of *5.5/10*). These responses indicate that participants working with our tool were less occupied by the details of the annotation process.

V. Related Work

In this section we look at other approaches and tools that share similarities with ours presented in this paper.

Mattis *et al.* present an approach named *Concept-Aware Programming Environment* [16]. They are interested in making programming environments aware of *concepts* that are present in the code through identifier names, with the goal to help programmers to build their mental model of the code. Another use-case is in detecting *architectural drift*: change in meanings and distributions of words used in names of identifiers during program evolution. Their method for finding concepts in the code is automated, but allows programmer's intervention and correction. For sharing of corrected concepts in a distributed workspace, they suggest to embed them in comments, which would result in SCAs conceptually similar to our ones. Integration of their approach into development environments consists of tools for concept exploration, custom class diagrams, and concept-augmented IDE editor, debugger and VCS tools.

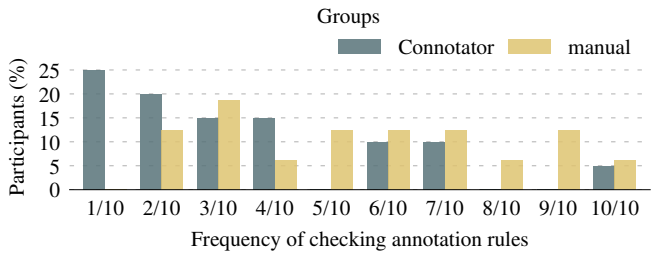

Fig. 10. The frequency of checking annotation rules by participants in n out of 10 cases of inserting an annotation into the code.

We presented a uniform annotation process that works for all annotation types and, considering their limitations, allows to choose the appropriate type per use case. Cazzola *et al.* extended LLAs for languages *C#* [17] and *Java* [18] to finer granularities through customized compilers, by which they extended the applicability of this single annotation type to code blocks and expressions. In comparison with our approach, the availability of annotations in the compiler and for processing through reflection at runtime is an advantage. However, a special tool (in this case a custom compiler) is still needed, and each such solution is restricted to a single language.

Current implementation of *Connotator* has only a simple metadata model, where annotations can represent high-level concerns through their names. A practical extension of this model would be to link external resources to the metadata. Similar thing was done by Baltes *et al.* who designed *SketchLink* tool [5] for linking sketches documenting high-level design to code elements through SCAs. Their design included a service for uploading and managing images of sketches from mobile devices and web browsers. IDE plug-in linked these images through unique identifier included in source code comments.

Our approach to select structurally valid annotatable code fragments uses AST patterns. Kästner *et al.* [19] and Behringer *et al.* [20] use AST rules to ensure structurally valid separation of features in feature-oriented development of software product lines. Kästner *et al.* call the annotation process *coloring* and Behringer *et al.* extend it with *snippet* code organization system for managing feature variability.

Cséri *et al.* [21] present their approach to assign source code comments to specific elements of the AST. They were interested in comment-to-element assignment for software maintenance tool, but had to work with legacy codebases, inside of which the assignment needed to happen. Their solution, similar to ours, consists of project-specific rules for defining relative comment placements, but the rules are more complex, supporting, e.g., assignment of a single comment node to multiple code elements. On the other hand, they do not differentiate types of comments, because they only process existing comments and do not need to generate new ones.

Rule-based comment assignment is also used in tool *TagSEA* by Storey *et al.* [22]. It uses a simple rule: comments are bound to the closest enclosing *Java* element. Such rule is sufficient if metadata granularity does not go below methods.

In contrast to our per-project configurable comment-to-element assignment, Sommerlad *et al.* [13] used fully-automatic assignment, distinguishing *leading*, *trailing* and *freestanding* comments. Their goal was to retain all comments and their positions while refactoring the code.

We used annotations for high-level metadata from the problem domain. Sulír and Porubän in their approach [23] used annotations to preserve low-levle runtime information.

VI. Conclusion and Future Work

In this paper, we presented our work towards allowing programmers to more easily preserve their high-level knowledge of source code they create by annotating it.

We gave an overview of the concept of the metadata-aware development environment and focused on its first building block: the support for the annotation process. For annotating code with three different types of annotations, and making the annotating workflow uniform, we designed an abstraction of the process. First, a programmer chooses source code elements to annotate and annotation representing metadata that should be bind to these elements. Then, a specific annotation is applied automatically by a supporting tool. We also described our prototype of such a tool in the form of a plugin for *IntelliJ* platform-based IDEs, called *Connotator*.

Finally, we reported the experiment in which we evaluated *Connotator* regarding its effect on the annotation process. We confirmed the hypothesis that the tool could increase placement consistency of comment annotations with annotation rules. The group of participants using the tool achieved higher consistency and reported less distraction by the details of the annotating than the group without the tool. The hypothesis that the tool increases annotating speed was not confirmed, although the group using the tool tended to finish the task sooner.

The natural next progress is to explore in detail the remaining two aspects of MADE to better utilize the preserved metadata and facilitate program comprehension. An interesting direction may also be in merging intrinsic metadata already available in IDEs with the preserved, extrinsic ones, and providing a querying facility for the resulting model. The queries could be used to customize views of code provided by an IDE, similarly to the concept of *scriptable* IDE presented by Asenov *et al.* [24].

Although our approach supports multiple types of annotations, we used only one type in the presented experiment. We will focus on assessing the annotation process using a mixture of annotation types in future evaluations.

REFERENCES

[1] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Evaluation and Usability of Programming Languages and Tools on - PLATEAU '10*. ACM Press, oct 2010. doi: 10.1145/1937117.1937125 pp. 1–6.

[2] V. Vranić, J. Porubän, M. Bystrický, T. Frťala, I. Polášek, M. Nosáľ, and J. Lang, "Challenges in Preserving Intent Comprehensibility in Software," *Acta Polytechnica Hungarica*, vol. 12, no. 7, pp. 57–75, 2015. doi: 10.12700/APH.12.7.2015.7.4

[3] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, jan 2013. doi: 10.1002/smr.567

[4] M. Sulír, M. Nosáľ, and J. Porubän, "Recording concerns in source code using annotations," *Computer Languages, Systems and Structures*, vol. 46, pp. 44–65, nov 2016. doi: 10.1016/j.cl.2016.07.003

[5] S. Baltes, P. Schmitz, and S. Diehl, "Linking sketches and diagrams to source code artifacts," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. New York, New York, USA: ACM Press, 2014. doi: 10.1145/2635868.2661672 pp. 743–746.

[6] M. Nosáľ, J. Porubän, and M. Nosáľ, "Concern-oriented source code projections," in *Proceedings of the 2013 Federated Conference on Computer Science and Information Systems*, Kraków, 2013, pp. 1541–1544.

[7] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki, "Maintaining feature traceability with embedded annotations," in *Proceedings of the 19th International Conference on Software Product Line - SPLC '15*. New York, New York, USA: ACM Press, 2015. doi: 10.1145/2791060.2791107 pp. 61–70.

[8] J. Juhár and L. Vokorokos, "Exploring code projections as a tool for concern management," *Acta Electrotechnica et Informatica*, vol. 16, no. 3, pp. 26–31, 2016. doi: 10.15546/aeei-2016-0020

[9] K. Rástočný and M. Bieliková, "Metadata Anchoring for Source Code: Robust Location Descriptor Definition, Building and Interpreting," in *24th International Conference on Database and Expert Systems Applications*. Prague: Springer, Berlin, Heidelberg, 2013. doi: 10.1007/978-3-642-40173-2_30 pp. 372–379.

[10] M. P. Robillard and F. Weigand-Warr, "ConcernMapper: simple view-based separation of scattered concerns," in *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange - eclipse '05*. New York, New York, USA: ACM Press, oct 2005. doi: 10.1145/1117696.1117710 pp. 65–69.

[11] M. Sulír, M. Bačíková, S. Chodarev, and J. Porubän, "Visual augmentation of source code editors: A systematic mapping study," *Journal of Visual Languages & Computing*, vol. 49, pp. 46–59, dec 2018. doi: 10.1016/J.JVLC.2018.10.001

[12] D. Jemerov, "Implementing refactorings in IntelliJ IDEA," in *Proceedings of the 2nd Workshop on Refactoring Tools - WRT '08*. ACM Press, oct 2008. doi: 10.1145/1636642.1636655 pp. 1–2.

[13] P. Sommerlad, G. Zgraggen, T. Corbat, and L. Felber, "Retaining comments when refactoring code," in *Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA Companion '08*. New York, New York, USA: ACM Press, 2008. doi: 10.1145/1449814.1449817 p. 653.

[14] I. Salman, A. T. Misirli, and N. Juristo, "Are Students Representatives of Professionals in Software Engineering Experiments?" in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, may 2015. doi: 10.1109/ICSE.2015.82 pp. 666–676.

[15] A. T. T. Ying, J. L. Wright, S. Abrams, A. T. T. Ying, J. L. Wright, and S. Abrams, "An exploration of how comments are used for marking related code fragments," in *Proceedings of the 2005 workshop on Modeling and analysis of concerns in software - MACS '05*, vol. 30, no. 4. New York, New York, USA: ACM Press, 2005. doi: 10.1145/1083125.1083141 pp. 1–4.

[16] T. Mattis, P. Rein, S. Ramson, J. Lincke, and R. Hirschfeld, "Towards concept-aware programming environments for guiding software modularity," *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience*, pp. 36–45, 2017. doi: 10.1145/3167110

[17] W. Cazzola, A. Cisternino, and D. Colombo, "Freely annotating C#," *Journal of Object Technology*, vol. 4, no. 10, pp. 31–48, 2005. doi: 10.5381/jot.2005.4.10.a2

[18] W. Cazzola and E. Vacchi, "@Java: Bringing a richer annotation model to Java," *Computer Languages, Systems and Structures*, vol. 40, no. 1, pp. 2–18, 2014. doi: 10.1016/j.cl.2014.02.002

[19] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*. New York, New York, USA: ACM Press, 2008. doi: 10.1145/1368088.1368131 p. 311.

[20] B. Behringer, L. Kirsch, and S. Rothkugel, "Separating features using colored snippet graphs," in *Proceedings of the 6th International Workshop on Feature-Oriented Software Development - FOSD '14*. New York: ACM Press, 2014. doi: 10.1145/2660190.2660192 pp. 9–16.

[21] T. Cséri, Z. Szügyi, and Z. Porkoláb, "Rule-based assignment of comments to AST nodes in C++ programs," in *Proceedings of the Fifth Balkan Conference in Informatics - BCI '12*. New York, New York, USA: ACM Press, 2012. doi: 10.1145/2371316.2371381 pp. 291–294.

[22] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby, "Shared waypoints and social tagging to support collaboration in software development," in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work - CSCW '06*. New York, New York, USA: ACM Press, 2006. doi: 10.1145/1180875.1180906 pp. 195–198.

[23] M. Sulír and J. Porubän, "Exposing Runtime Information through Source Code Annotations," *Acta Electrotechnica et Informatica*, vol. 17, no. 1, pp. 3–9, 2017. doi: 10.15546/aeei-2017-0001

[24] D. Asenov, P. Müller, and L. Vogel, "The IDE as a scriptable information system," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. New York, New York, USA: ACM Press, 2016. doi: 10.1145/2970276.2970329 pp. 444–449.