Sharing Developers' Mental Models through Source Code Annotations

Matúš Sulír, Milan Nosáľ
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
Email: matus.sulir@tuke.sk, milan.nosal@gmail.com

***Abstract*—*Context:* Developers possess mental models containing information far beyond what is explicitly captured in the source code. *Objectives:* We investigate the possibility to use source code annotations to capture parts of the developers' mental models and later reuse them by other programmers during program comprehension and maintenance. *Method:* We performed two studies and a controlled experiment. *Results:* Developers' mental models overlap and thus can be shared. Possible use cases of shared annotations are hypotheses confirmation, feature location, obtaining new knowledge, finding relationships and maintenance notes. In the experiment, the presence of annotations reduced program comprehension and maintenance time by 34%. *Conclusion:* Annotations are a viable way to share programmers' thoughts.**

I. Introduction

WHEN programmers develop a new program, they continuously create a mental model, which is a representation of the program in their mind. They try to express the mental model in the programming language constructions. However, not all parts of the mental model are transferred into the source code and some details are lost.

A. Motivation

For example, in small parts of a large information system, we can be forced to deal with a character encoding different than in the rest of the system because of the legacy library limitations.

Later, during the program maintenance phase, developers make a tremendous effort to recover such types of information from the source code [2]. However, they rarely persist their findings [3] and the situation occurs again and again.

We could record the information about the encoding from the aforementioned example in a form of a Java annotation `@Encoding("win-1250", reason="myOldLib 2.4")` above all affected classes or methods. It would be later possible to use the IDE (integrated development environment) searching capabilities on the given annotation to find all methods where this encoding is used if an encoding-related bug is reported or refactoring is planned to replace the legacy library.

This work was supported by VEGA Grant No. 1/0341/13 Principles and methods of automated abstraction of computer languages and software development based on the semantic enrichment caused by communication.
The first two research questions are also elaborated in the second author's PhD thesis [1].

B. Aim

The purpose of this paper is to investigate the viability of annotations as a medium to share parts of developers' mental models.

Hypothesis: Annotations created by one group of developers are useful for comprehension and maintenance tasks performed by other developers.

We formulate the research questions as follows:

- **RQ1**: Do programmers' mental models overlap?
- **RQ2**: How do developers use shared annotations when they are available?
- **RQ3**: Does using annotations created by others improve program comprehension and maintenance correctness, time and confidence?

To answer each of the questions, we used an appropriate empirical research methodology [4].

II. Concern Annotations

Before describing the studies, we briefly introduce the notion of concerns and their kinds.

A. Basic Concepts

A *concern* can be characterized as a developer's intent of a particular piece of code: What should this code accomplish? How would I tersely characterize it? Is there something special about it? Some concerns can be obvious by looking at the code itself (chiefly from names of the identifiers), but many concerns are hidden.

A concern is similar to an aspect in aspect-oriented programming. In contrast to aspects, concerns can overlap – one piece of code can belong to multiple concerns [5]. This complicates the situation: We cannot simply name all classes and methods according to their concerns since one identifier can have only one name. Thus, we can also look at concerns as alternative names for identifiers.

It is possible for a class or method in Java to have more than one annotation. Therefore, source code annotations (attributes in C# terminology) are an ideal candidate to implement concerns.

For each distinct concern, we recommend to create one Java *annotation type*. For example, we can create an annotation type `@Persistence` which tells us the code marked with it fulfills the task of persistent storage of objects.

Subsequently, in our imaginary program, we could mark the methods like `FileDialog.open()`, `Note.load()`, `Note.save()` and a class `FileFormat` with it. We will call them *annotation occurrences*.

At the same time, the first of the mentioned methods could be also annotated with the `@GUI` concern, as it presents a GUI (graphical user interface) dialog to a user.

Compared to traditional source code comments, concern annotations are more formal. We can use standard IDE features like navigating to the declaration, usages searching, refactoring [6] and other on them.

Concern annotations can have parameters to further specify their properties. They may be also commented by natural language comments if needed.

B. Kinds of Concern Annotations

Domain annotations document concepts and features of the application (problem) domain. For example, all source code elements representing the feature of filtering a list of items can be marked with an annotation `@Filtering`. Similarly, all code related to bibliographic citations could be annotated by `@Citing`.

Design annotations document design and implementation decisions like design patterns, e.g., `@Observer`.

Maintenance annotations are intended to replace the traditional TODO and related comments. An example is the `@Unused` annotation for parts of code not used in a project at all.

III. Mental Model Overlapping

Our first goal is to find out whether at least some of the concerns recognized by one programmer can be recognized by other developers. If so, then the mental model of one developer at least partially overlaps with the other persons' mental model. This is a necessary condition for concern annotation sharing to be useful.

A. Method

We asked multiple programmers to independently annotate the source code of an existing program. Then we measured to what extent their annotations overlap.

1) Materials: For this and subsequent studies, we used EasyNotes[1] – a desktop application for bibliographic note-taking. It is a small-scale Java project consisting of around 2500 lines of code located in 33 classes. Except for scarce source code comments, it has no documentation available.

[1]http://github.com/MilanNosal/easy-notes

TABLE I
The number of recognized concerns and annotation occurrences per individual subjects

Subject	Concerns	Occurrences
A	11	70
B	12	56
C	24	108
D	20	79
E	12	56
F	14	89
G	17	140
Total (distinct)	**46**	**464**

2) Participants: This study had 7 participants:

- A — a researcher and lecturer with PhD in Computer Science,
- B — an industrial Java programmer,
- C — a postdoc and Java programmer,
- D — an associate professor with extensive Java experience,
- E, F — first-year PhD students,
- G — the author of EasyNotes.

None of the subjects, except for the author, had a previous experience with EasyNotes. The activity was individual and the participants were not allowed to interact during the experiment.

3) Procedure: First, the participants were given the original source code of EasyNotes without any annotations (commit `a299e64`). They had an arbitrary amount of time available to become familiar with the application both from an end-user and programmer perspective.

Next, they were asked to create a custom annotation type for each concern they recognized in the application and to mark classes, member variables and methods with the annotations they just created whenever they thought it was appropriate.

4) Analysis: Finally, we collected the modified projects and analyzed them semi-automatically for an overlap in annotation types and the use of the annotations on specific elements. A manual intervention was necessary because some participants used a slightly different name for the same concern – e.g., `@Tags` and `@Tagging` both represent the "tagging" concern.

B. Results

For each participant, a set of created concerns (annotation types) was constructed. The number of concerns created by individual participants ranged from 11 to 24 and the number of annotation occurrences from 56 to 140 (see Table I).

1) Concern Sharing: We constructed a set of distinct concerns, i.e., an union of all sets of the concerns recognized by the participants. The size of this set, i.e., a number of distinct concerns, is 46 (the Total row in Table I). More than a half of them (26) was shared by at least two participants. We will call them *shared concerns*. A list of all shared concerns is in Table II.

TABLE II
A LIST OF ALL SHARED CONCERNS IN EASYNOTES

	Concern	Shared by *n* participants	Effective agreement
1	Searching	6	61%
2	Note editing	5	60%
3	Note change observing	5	50%
4	Note presenting	5	21%
5	Unused code	4	67%
6	Tagging	4	64%
7	Persistence	4	41%
8	Links	4	39%
9	Note adding	4	38%
10	Data model	4	36%
11	Loading notes	4	35%
12	Saving notes	4	31%
13	GUI	4	26%
14	Note deleting	4	18%
15	UI-model mapping	4	4%
16	Filter implementation	3	18%
17	TODO	3	8%
18	Exceptions	2	100%
19	Utilities	2	50%
20	Model change watching	2	17%
21	Filters management	2	12%
22	Notes manipulation	2	8%
23	Questions about code	2	0%
24	Coding by convention	2	0%
25	BibTeX	2	0%
26	Domain entity	2	0%

2) Occurrence Sharing: Similarly, we constructed a set of all distinct annotation occurrences with a total size of 464 (as noted in Table I). 128 of them were so-called *shared annotation occurrences* which means they were shared by at least two participants.

We define an *effective agreement* as the number of shared annotation occurrences divided by the total number of annotation occurrences. For our EasyNotes study, the overall effective agreement was 27.59%. It is possible to see the values of effective agreement for individual concerns in Table II. We can consider the effective agreement of a specific concern its quality indicator – to what extent multiple developers agree about the mapping of this concern to source code elements.

3) Agreement between Participants: Fig. 1 shows the number of shared concerns between each pair of participants. We can obtain interesting insights from this matrix. The subjects A and D did not create any common annotation type in our study – this could be an indication of a huge difference in the mental models of these two people. In fact, D is a former or current supervisor of all other participants except A. On the other hand, G (the author of EasyNotes) shares the most concerns with all other people. This could mean that the source code author is the best annotator.

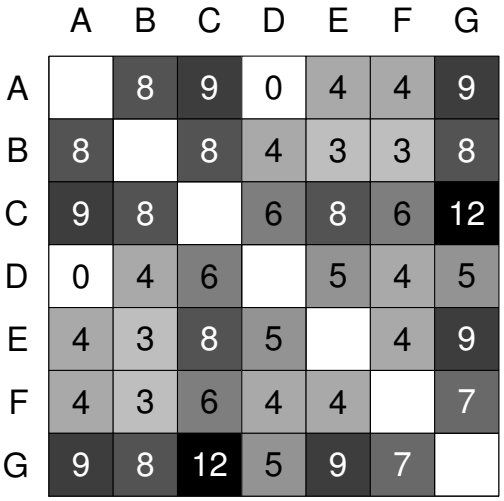

Fig. 1. The number of shared concerns between individual subjects

A similar matrix was constructed for concern occurrences. Qualitatively, it resembled the annotation type matrix.

C. Threats to Validity

1) Internal Validity: While some hypotheses were outlined in this section, being an exploratory study [7], there is a need to properly quantify and statistically confirm or reject them. This suggest interesting future research directions.

2) External Validity: We performed the study only on a small-scale Java project. The participants were all from the same department which could affect their mental models. A more extensive study should be conducted in the future.

D. Conclusion

We studied mental model overlapping, where parts of the mental model were represented by source code annotations. In our study, about

- 57% of all concerns
- and 28% of concern occurrences

were shared by at least two participants. This means there is a potential in recording and subsequent reuse of these data.

IV. CONCERN ANNOTATIONS USE CASES

The goal of the second study is find out how third-party developers (i.e., not the annotation authors) use the annotations in the source code if they are available.

A. Method

We conducted an observational study.

1) Materials: We copied all annotation types shared by at least two subjects (from the first study, see Table II for a list) into the EasyNotes project. For each of these annotation types, we merged all its occurrences (recognized by at least one developer) into the project. The resulting source code was manually edited for consistency by the EasyNotes author. It is published as the commit `f52872b`.

2) Participants: There were three participants:

- K a first-year Computer Science PhD student,
- L a masters degree student with a minor industrial experience,
- M a professional developer with 2 years of industrial experience.

None of them had a previous knowledge of EasyNotes.

3) Procedure: First, all annotations were briefly introduced by the EasyNotes author. The subjects were reminded about common feature location possibilities of the NetBeans IDE.

Each participants was given the same task: To add a "note rating" (one to five stars) feature to EasyNotes. The fulfillment of this task required a modification of multiple application layers – from the model to the user interface.

We used a think-aloud method [7], i.e., the participants were kindly requested to comment their thoughts when comprehending the code.

B. Results

We will now look at typical use cases of concern annotations during our study.

1) Confirming Hypotheses: The most common use of concern annotations was to confirm hypotheses about the code. For example, the participant K used the `@NotesSaving` annotation to confirm that a particular piece of stream-writing code actually saves notes.

2) Feature Location: In contrast to traditional comments, it is possible to use the Find Usages feature on an annotation type to find all concern occurrences. Our participants were finding the occurrences of the "filtering", "note adding", "note saving" concerns and others. This was considered helpful especially to find if they did not forget to implement the necessary methods for a particular aspect of the note rating feature.

3) Non-Obvious Concerns: The developers also used annotations to obtain new knowledge about the source code. For instance, a UI (user interface) code contained a method used both when adding a new note and when editing an existing one. However, just a note editing concern was obvious from a brief source code inspection. Only thanks to the concern annotation `@NoteAdding`, the participant M noticed the code is used for note adding, too.

4) Elements Relationship: The subjects noticed that if two or more elements are marked with the same annotation type, there is an implicit relationship between them. For instance, when using the MVC (Model-View-Controller) design pattern, the code in the model marked with a specific annotation is linked with the UI code with the same annotation.

5) Maintenance Notes: The annotation `@Unused` marks methods not used in the rest of the code. This helped the participants to skip them when scanning the code and thus save time.

C. Conclusion

1) Advantages: The participants stated that compared to traditional natural language comments, annotations are much shorter and thus easier to spot. They are also better structured and usually less ambiguous. The ability to find all usages of a particular concern through standard features present in contemporary IDEs was also appreciated.

2) Disadvantages: The participant with an industrial experience (M) remarked there is a possible scaling problem. Even in a small project like EasyNotes, 26 shared concerns were identified. In large projects, where this number is expected to grow, some sort of concern categorization would definitely be needed. As Java annotations do not support inheritance, marking them with meta-annotations or sorting them to packages are possible solutions.

V. The Effect of Annotations on Program Maintenance

We performed a controlled experiment to study the effect of the annotated source code on program comprehension and maintenance.

The guidelines to perform software engineering experiments on human subjects [8] were used. To present our findings, the experiment reporting guidelines [9] were followed. We customized them to the specific needs of this experiment.

A. Hypothesis

Similar to Barišić et al. [10], we were interested in correctness, time and confidence.

We hypothesize that the presence of concern annotations in the source code improves program comprehension and maintenance correctness, time and confidence. Thus we formulate the null and alternative hypotheses:

$H1_{null}$: The correctness of the results of program comprehension and maintenance tasks on an annotated project = the correctness on the same project without concern annotations.

$H1_{alt}$: The correctness of the results of program comprehension and maintenance tasks on an annotated project > the correctness on the same project without concern annotations.

$H2_{null}$: Time to complete program comprehension and maintenance tasks on an annotated project = time to complete them on the same project without concern annotations.

$H2_{alt}$: Time to complete program comprehension and maintenance tasks on an annotated project < time to complete them the same project without concern annotations.

$H3_{null}$: Participants' confidence of their answers to program comprehension questions on an annotated project = their confidence on the same project without concern annotations.

$H3_{alt}$: Participants' confidence of their answers to program comprehension questions on an annotated project > their confidence on the same project without concern annotations.

We will statistically test the hypotheses with a confidence interval of 95% ($\alpha = 5\%$).

B. Variables

Now we will define independent variables, i.e., the factors we control, and dependent variables – the outcomes we measure.

1) Independent Variables: There is only one independent variable – the presence of concern annotations in the project. It has a nominal scale which means there is a finite number of possible values without any meaningful ordering [4]. The levels (possible values) of this variable are: yes ("annotated") and no ("unannotated").

2) Dependent Variables: The correctness was measured as a number of correct answers (or correctly performed tasks) divided by the total number of tasks (5). The tasks are not weighted, each of them is worth one point. The assessment is subjective – by a researcher.

The second dependent variable is the time to finish the tasks. Its scale is of a ratio type since the ratio between two values is meaningful [4]. Although it was technically measured with millisecond precision, we will use the unit "minutes" rounded to two decimal places in subsequent analysis. We are interested mainly in the total time, i.e., a sum of times for all tasks.

Instead of measuring just time alone, it is possible to define efficiency as a number of correct tasks and questions divided by time. On one hand, efficiency depends on correctness, which already is a dependent variable. On the other hand, efficiency can deal with participants who fill the answers randomly to finish quickly [11]. We decided to use efficiency only as an auxiliary metric to make sure that time differences are still significant even if the correctness is considered.

For each comprehension question, we also asked a subject how confident (s)he was on a 3-point Likert scale: from Not at all (1) to Absolutely (3). Since we asked a subject about the confidence equally for each task, we consider it meaningful to calculate the mean confidence, which is the third dependent variable.

C. Experiment Design

1) Materials: Again, the EasyNotes project was used. This time, we prepared two different versions:

- with shared concern annotations (as in the second study)
- and without annotations.

As the project was only scarcely commented, we deleted all traditional source code comments from both versions to remove a potential confounding factor. Only comments for the annotation types themselves were left intact, as we regard them as their integral part.

During this experiment, we used the NetBeans IDE.

2) Participants: We used 18 first-year master's degree Computer Science students as participants. Carver et al. [12] recommend to integrate software engineering experiments performed on students with teaching goals. We decided to execute the experiment as a part of the Modeling and Generation of Software Architectures course, which contained Java annotations in its curricula.

The course was attended by students focused not only on software engineering, but also on other computer science subfields. Inclusion criteria were set to select mainly students with a prospective future career as professional programmers. Additionally, as EasyNotes is a Java project, a sufficient Java language knowledge was required.

The experiment was performed during three lessons in the same week – the first time with four students, then 9 and finally with the remaining 5 students. Each session lasted approximately 1.5 hours. The study was executed in a separate room, so the participants were not disturbed.

3) Design: When assigning the subjects to groups, we applied a completely randomized design [4]. This means that each group received only one treatment – either an annotated or an unannotated program – and the assignment was random. Each participant drew a piece of paper with a number on it. Subjects with an odd number were assigned to the "annotated" group, participants with an even number to the "unannotated" one. Our design was thus balanced, with n=9 per group.

4) Instruments: To both guide the subjects and collect the data, we designed an interactive web form[2]. All fields in the form were mandatory, so the participants could not skip any task.

We asked the subjects to install a NetBeans plugin, SSCE[3]. Although it is not its primary feature, it provides an option to record programming sessions – time elapsed, a list of open files and NetBeans windows gaining the focus. Just before the start of each task, a subject clicked the button to start a new session, named after the task (e.g., "Filter"). Immediately after the task was finished, (s)he clicked the button again to end the session. The collected data were written to an XML file which the participants uploaded to the web form at the end of the experiment.

D. Procedure

1) Training: At the beginning of the experiment, the users were given 5 minutes to familiarize themselves with EasyNotes from an end-user perspective, without looking at the source code. This provided them an overview of the application domain and helped them to better understand their subsequent tasks. Then, the session monitoring plugin was briefly introduced.

A short presentation about concern annotations usage in the NetBeans IDE followed. A researcher presented how to show a list of all available concerns, how to use the Find Usages feature on an annotation type and how to navigate from the annotation occurrence to an annotation type.

Just before each of the two maintenance tasks, the researcher presented the participants a running application with the required feature already implemented. This had two positive effects:

- It significantly lowered the task ambiguity. While a natural-language description was available in the web form during the tasks, seeing the finished application gave the participants greater confidence, so almost nobody asked unnecessary questions.
- We consider this a replacement for unit tests. As GUI code is notoriously difficult to test [13], we decided not implement the test code. At the same time, the

[2]http://www.jotformeu.com/sulir/sharing-annotations
[3]http://github.com/MilanNosal/sieve-source-code-editor

researcher's discretion about the task correctness was not indispensable during the experiment. Students just knew they finished the task when their application did the same thing as we had showed them.

In addition, the participants later uploaded their modified version of the source code to the web form. This way, potential disputes could be resolved without time stress.

2) Tasks: The experiment comprised of:

- one additive maintenance task (we will name it *Filter*),
- three program comprehension questions (*Test*),
- one corrective maintenance task (*Cite*),

in that order.

The tasks were formulated as follows:

Filter In a running EasyNotes application, load the sample notes and look at the searching feature (the down-right part of the window). Try how searching in the note text works (the option "Search in": "text"). Your task will be to add a filter to the EasyNotes application, which will search in the notes title (the option "title").

Cite In the EasyNotes application, there is a field "Cite as:" (the down-right window part). Currently, it displays information in the form: *somePubID* where somePubID is the value of the "PubID:" field. Your task is to modify the program so the "Cite as:" field will display information in the form: *\cite{somePubID}*.

Both tasks were simple, although the *Filter* task was slightly more complex than the latter. It required the creation of a new class with approximately 15 lines of code, whereas the *Cite* task could be accomplished by modifying just one source code line.

The questions asked in the *Test* about program comprehension were:

Q1 What does the `runDirMenuItemActionPerformed` in the class `easynotes.swingui.EasyNotesFrame` do?

Q2 How is the class `easynotes.model.abstract-Model.UTFStringComparator` used in the EasyNotes project?

Q3 What method/s (and in which class) do perform note deleting?

3) Demographic Data Collection: At the end of the experiment, the form contained three demographic questions about the subjects' abilities:

- a general programming experience,
- their experience with Java, annotations and NetBeans
- and their English level.

Each question had possible answers on a 5-point Likert scale: from Beginner to Expert. We did not perform any specialized tests to measure programming experience, as a subjective opinion is good enough [14].

4) Debriefing: We also included a question asking to what extent did the subjects use annotations when comprehending the code. Possible answers ranged from Never to Always on a 5-point Likert scale. Finally, the form also contained a free-form question where the participants could describe how the annotations helped them in their own words.

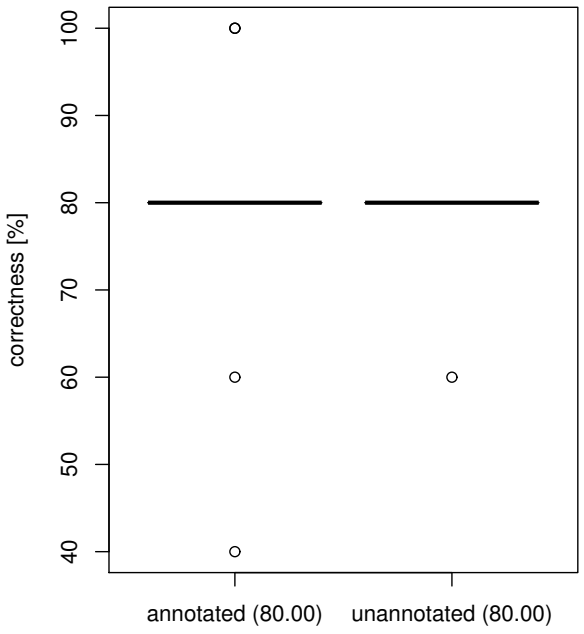

Fig. 2. The ratio of correct answers for each group.

E. Results

The measured values and their summary is presented in Table III. To analyze the results, we used the R scripting language, auxiliary Ruby scripts and spreadsheets.

Each specific hypothesis considers one independent variable on a nominal scale with two levels (annotated, unannotated) and one dependent variable (either correctness, time or confidence). For each dependent variable, we displayed the values on a histogram and a normal Q-Q plot. None of the variables looked normally distributed, so we used the Mann-Whitney U test as a statistical test for our hypotheses.

1) Correctness: The median of correctness for both the "annotated" and "unannotated" group was 80%. Except for a few outliers, all subjects answered exactly 4 out of 5 questions correctly. See Fig. 2 for a plot.

The computed p-value (roughly speaking, the probability that we obtained the data by chance) is 0.3898, which is more than 0.05 (our significance level). This means we accept $\mathbf{H1_{null}}$. We did not prove that the presence of annotations has a positive effect on program comprehension and maintenance correctness.

As we can see in Table III (column Correctness), the most difficult question was Q2. Only two participants answered it correctly – both from the "annotated" group. The class of interest was not used in EasyNotes at all. This fact was noticeable by looking at the `@Unused` annotation.

TABLE III
THE EXPERIMENT RESULTS FOR INDIVIDUAL SUBJECTS

The "annotated" group																	
ID	**Correctness [true/false]**						**Time [min]**				**Efficiency [tasks/min]**	**Confidence [1-3]**				**Files**	**Annotations useful? [1-5]**
	Filter	**Cite**	**Q1**	**Q2**	**Q3**	**Total**	**Filter**	**Cite**	**Test**	**Total**		**Q1**	**Q2**	**Q3**	**Mean**		
1	1	1	1	0	1	80%	12.03	3.00	13.96	28.99	0.14	1	2	3	2.00	19	1
3	1	1	0	0	0	40%	5.23	2.81	11.13	19.17	0.10	3	2	3	2.67	10	3
5	1	1	1	1	1	100%	17.43	3.93	6.71	28.07	0.18	3	3	3	3.00	18	4
7	1	1	1	1	1	100%	7.79	1.43	11.85	21.07	0.24	3	3	3	3.00	10	4
9	1	1	1	0	1	80%	6.72	5.86	3.87	16.45	0.24	3	3	2	2.67	20	2
11	1	1	1	0	1	80%	8.41	4.58	4.32	17.31	0.23	3	3	3	3.00	13	4
13	1	1	0	0	1	60%	20.97	3.48	8.80	33.25	0.09	3	2	3	2.67	11	3
15	1	1	1	0	1	80%	4.64	1.91	5.22	11.77	0.34	2	2	3	2.33	11	2
17	1	1	1	0	1	80%	25.08	6.38	6.99	38.45	0.10	2	2	3	2.33	16	4
Median	**1**	**1**	**1**	**0**	**1**	**80%**	**8.41**	**3.48**	**6.99**	**21.07**	**0.18**	**3**	**2**	**3**	**2.67**	**13**	**3**
Std.dev.	**-**	**-**	**-**	**-**	**-**	**18.56%**	**7.41**	**1.67**	**3.57**	**8.80**	**0.08**	**-**	**-**	**-**	**0.35**	**4.06**	**-**
The "unannotated" group																	
ID	**Correctness [true/false]**						**Time [min]**				**Efficiency [tasks/min]**	**Confidence [1-3]**				**Files**	**Annotations useful? [1-5]**
	Filter	**Cite**	**Q1**	**Q2**	**Q3**	**Total**	**Filter**	**Cite**	**Test**	**Total**		**Q1**	**Q2**	**Q3**	**Mean**		
2	1	1	1	0	1	80%	2.84	4.72	10.66	18.22	0.22	3	2	3	2.67	9	NA
4	1	1	1	0	1	80%	8.76	23.45	8.10	40.31	0.10	3	2	3	2.67	19	NA
6	1	1	1	0	1	80%	18.24	5.23	5.62	29.09	0.14	3	2	1	2.00	17	NA
8	1	1	1	0	1	80%	6.47	5.59	11.23	23.29	0.17	3	2	3	2.67	8	NA
10	1	1	1	0	1	80%	4.82	9.64	17.50	31.96	0.13	3	2	3	2.67	8	NA
12	1	1	1	0	1	80%	11.11	2.09	11.30	24.50	0.16	2	2	2	2.00	13	NA
14	1	1	0	0	1	60%	30.73	7.19	5.50	43.42	0.07	2	2	3	2.33	17	NA
16	1	1	1	0	1	80%	12.56	18.39	16.07	47.02	0.09	3	2	3	2.67	14	NA
18	1	1	1	0	1	80%	25.54	9.59	12.94	48.07	0.08	3	2	3	2.67	16	NA
Median	**1**	**1**	**1**	**0**	**1**	**80%**	**11.11**	**7.19**	**11.23**	**31.96**	**0.13**	**3**	**2**	**3**	**2.67**	**14**	**NA**
Std.dev.	**-**	**-**	**-**	**-**	**-**	**6.67%**	**9.56**	**6.98**	**4.18**	**11.06**	**0.05**	**-**	**-**	**-**	**0.30**	**4.22**	**-**

2) Time: The differences in the total time for all tasks between two groups are graphically depicted in the box plot in Fig. 3. The median time changed from 31.96 minutes for the "unannotated" group to 21.07 minutes for the "annotated" one, which is a decrease by 34.07%.

The p-value of time is 0.0252, that is less than 0.05. The difference is statistically significant, therefore we reject $\mathbf{H2_{null}}$ and accept $\mathbf{H2_{alt}}$. The presence of concern annotations improves the program comprehension and maintenance time.

It is possible to see from Table III (column Time) that the median time for each individual task was better for the "annotated" group. The most prominent difference was for the task *Cite*. This can be due to the fact that the project contained the concern annotation `@Citing` which helped the participants find the relevant code quickly.

The median of efficiency, which we defined as the number of correctly performed tasks (and answers) divided by total time, raised by 42.32% (p=0.0385). This means the time improvement is significant even if we take correctness into account.

3) Confidence: The median of mean confidence is the same for both groups (2.67), as obvious from Table III, column Confidence / Mean and Fig.4. The p-value is 0.1710 (>

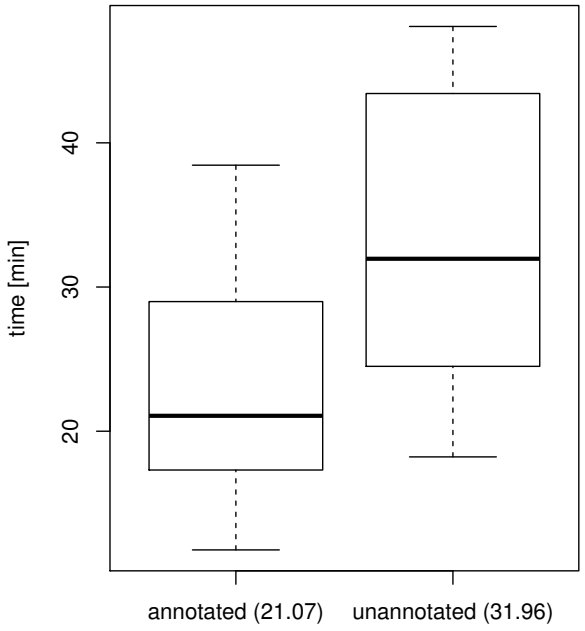

Fig. 3. Time to complete comprehension and maintenance tasks on an annotated vs. unannotated project

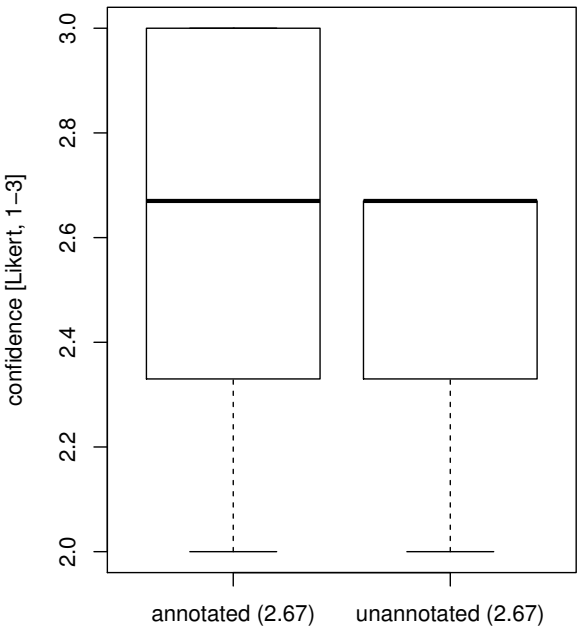

Fig. 4. The mean confidence for the "annotated" and "unannotated" group

0.05) and therefore we accept $\mathbf{H1_{null}}$. The effect of concern annotations on confidence was not demonstrated.

Looking at individual questions, Q2 was clearly perceived the most difficult. This corresponds with our previous finding about the correctness. An interesting fact is that no participant in the "unannotated" group was confident enough to select the level 3 (Absolutely), while in the "annotated" group, there were 4 such subjects and two of them really answered correctly.

4) Other Findings: Although not included in our hypotheses, we also measured how many unique Java source files did the subjects open in the IDE during the whole experiment. We excluded the annotation type files in these numbers. The results are in Table III, column Files. There was only a marginal and statistically insignificant improvement in medians (from 14 to 13).

As seen from Table III, column "Annotations useful?", concern annotations were perceived relatively helpful by the participants (median 3 from 5-point Likert scale).

Answers to a free-form question asking how specifically were the annotations useful included:

- faster orientation in a new project,
- faster searching (mainly through Find Usages on annotations),
- less scrolling,
- "they helped me to understand some methods",
- "annotations could be perfect, but I would have to get used to them".

F. Threats to Validity

To analyze threats to validity of this experiment, we used [15] as guidelines.

1) Construct Validity: Similar to Kosar et al. [11], we compensated the students with points for the participation in the experiment, which increased their enthusiasm. Unlike them, we did not reward the participants with bonus points for good results because our experiment spanned several days with the same tasks and this would motivate the students to discuss the task details with classmates, which could negatively affect the construct validity (interaction between subjects). Furthermore, the participants were explicitly told not to share experiment details with anyone. Therefore, we do not consider executing the experiment in three separate sessions an important validity threat.

To measure confidence, we used only a 3-point Likert scale. This decision was not optimal. Because subjects rarely select the value of 1 (which can be interpreted as guessing an answer), there were only two values left. This could be one of the reasons we did not found a statistically significant difference in confidence.

2) Internal Validity: There could be a selection bias in the experiment because we selected the participants using subjective criteria. As we already mentioned, we concentrated on subjects with a potential future career as developers.

We divided the subjects into groups randomly. Another possibility was a quasi-experiment (nonrandom assignment), i.e., to divide the subjects evenly according to the most important co-factor affecting the results, like their programming experience. However, random assignments tend to have larger effect sizes than quasi-experiments [16].

We did not perform a full pilot testing with third-party participants, only tested the comprehension questions on one of the researchers. We rejected 3 out of the 6 prepared questions because we considered them too difficult and ambiguous. Despite this, all tasks were either completed by almost all participants (Filter, Q1, Q3, Cite) or by almost none (Q2) during the experiment. This negatively affected the results for the "correctness" variable.

During the actual experiment, we used a native language (Slovak) version of the web form to eliminate the effect of English knowledge on the results. While the source code was in English, this did not present a validity threat since all participants had at least a medium level (3 on a 5-point Likert scale) of English knowledge.

3) External Validity: We invited only students to our experiment, no professional developers. We can take this fact positively – concern annotation consumers are expected to be mainly junior developers, whereas potential annotation creators are mostly senior developers. Furthermore, some students may already work in companies during their study.

EasyNotes is a small-scale Java project – around 3 KLOC (thousands of lines of code), including annotations. The effect of concern annotations on larger programs should be investigated.

4) Reliability: The concern annotations training (tutorial) was presented manually by a researcher. However, there were two independent researchers which took turns.

The experiment is replicable, as we published the data collection form (`http://www.jotformeu.com/sulir/sharing-annotations`) which contains both the guidelines and links to the materials (two versions of the EasyNotes project).

5) Conclusion Validity: A small number of subjects (n=9 per group) is the most important conclusion validity threat. If we used a paired design (to assign both treatments to every subject), we could easily reach n=18. However, the participants would quickly become familiar with EasyNotes and the second set of tasks would be affected by their knowledge.

G. Conclusion

We successfully confirmed the hypothesis that concern annotations have a positive effect on program comprehension and maintenance time. The group which had concern annotations available in their project reached a time more than 34% shorter than the group without them ($p < 0.05$).

On the other hand, we did not discover a significant change in correctness and confidence. The difference was neither negative (which could mean the annotations are confusing because of their "visual noise") nor positive and it was probably a result of the discussed validity threats.

VI. Related Work

A. Concerns

Reinikainen et al. [17] present concern-base queries to reason about the program. However, their approach is based on UML (Unified Modeling Language) models, while we use the source code of a system.

Niu et al. [18] propose the application of HFC (hierarchical faceted categories) on source code. Their approach requires specialized tools whereas we use source code annotations which have a standard IDE support.

B. Source Code Projections

In [5], we used concern annotations as one of the ways to perform source code projections. Projections allow to look at one source code from multiple different perspectives. For example, an IDE plugin can filter all code marked with a specific annotation type and display it in an editable window – even if it is spread across multiple files.

In this work, we take a more pragmatic approach. We investigate the possibility to use concern annotations in contemporary IDEs, using the features already available, like Find Usages. Above all, we perform three empirical studies to assess the viability and find possible use cases of concern annotations in everyday developer's life.

C. Annotations

Today, one of the most common applications of annotations is configuration. A programmer marks selected source code elements with appropriate annotations. Later, they are processed either by an annotation processor (during compilation) or by reflection (at runtime). For example, annotations can be used to define concrete syntax in parser generators [19] and to declare references between elements in a domain-specific language [20]. This way, annotations can indirectly modify the annotated program semantics. In contrast, our approach utilizes annotations just as clues for a programmer which are processed by an IDE when needed.

Sabo and Porubän [21] use source code annotations to preserve design patterns. Therefore, their approach does not include recording and sharing domain and maintenance annotations.

In Java, annotations can be only applied to packages, classes, member variables, methods and parameters. @Java [22], an extension to the standard Java language, brings annotations below the method level. It allows to mark individual source code statements like assignments, method calls, conditions and loops with annotations. This could be useful to capture e.g., algorithmic design decisions with concern annotations.

D. Mental Model Overlapping

Revelle et al. [23] also studied concern overlapping. However, their study included only two concern sets (compared to our 7). Their results are positive, too. This further confirms our hypothesis that it is possible to share mental models.

E. Code Bookmarks

Source code bookmarks are one of the standard features present in today IDEs. They allow a developer to mark specific source code lines with notes and later list the bookmarks and jump to each of them.

Collective code bookmarks [24] provide a way to share parts of developers' mental models. The plugin allows to share code bookmarks between developers in a team. However, this approach is IDE-dependent and the bookmarks are not saved to the original source code files which complicates standard procedures like versioning and merging.

F. Maintenance Notes

Developers often write "TODO comments" like `// TODO: fix this` to mark parts of source code which need their attention [25]. IDEs can then try to parse and display these notes in a task list window.

Our approach is more formal, as annotations are a part of the standard Java grammar and can be parsed unambiguously. Furthermore, it is possible to distinguish between multiple maintenance note types through individual annotation types.

VII. Conclusion and Future Work

In this paper, we presented an idea to use Java source code annotations to capture parts of developers' mental model, namely their concerns (intents), thus the name "concern annotations". Each concern (e.g., "searching", "editing", "GUI code", "unused code") is implemented as an annotation type. Subsequently, all classes and methods relevant to that concern are marked with the given annotation. Two studies and one experiment were conducted to assess the practical implications of this approach.

It is possible to share these annotations because the developers' mental models overlap: More than a half of the concerns created by one of 7 developers in our study was recognized by at least two of them. More than 1/4 of concern occurrences (locations in source code where a particular annotation is used) were shared by at least two participants.

An interesting future research question is to what extent the source code itself, when the same program is created separately by multiple people, overlaps.

In the second study, we discovered that concern annotations are particularly useful to confirm hypotheses about the code, locate the features, find out non-obvious concerns which a method fulfills, discover hidden relationships between elements. Concern annotations can be also used as a replacement of traditional TODO comments.

In the controlled experiment, we showed there is a statistically significant improvement of development time when performing program comprehension and maintenance tasks on a small-scale Java project. The group which had an annotated version of the same program available, consumed 1/3 less time than the group which did not have concern annotations present in the source code.

In our studies, the source code was commented scarcely or not at all. An interesting future comparison would consider an annotated program vs. a program without annotations, but with high-quality traditional source code comments instead.

Currently, the source code is annotated manually by a programmer, which is a time-consuming task. An interesting future work would be a method of (semi-)automatic annotation.

REFERENCES

[1] M. Nosáľ, "Leveraging program comprehension with concern-oriented projections," PhD thesis, Technical University of Košice, Apr. 2015.

[2] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 492–501. http://dx.doi.org/10.1145/1134285.1134355

[3] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 31:1–31:37, Sep. 2014. http://dx.doi.org/10.1145/2622669

[4] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[5] J. Porubän and M. Nosáľ, "Leveraging program comprehension with concern-oriented source code projections," in *3rd Symposium on Languages, Applications and Technologies*, ser. OpenAccess Series in Informatics (OASIcs), M. J. V. Pereira, J. P. Leal, and A. Simões, Eds., vol. 38. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 35–50. http://dx.doi.org/10.4230/OASIcs.SLATE.2014.35

[6] J. Kollár, I. Halupka, S. Chodarev, and E. Pietriková, "pLERO: Language for grammar refactoring patterns," in *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, Sept 2013, pp. 1503–1510.

[7] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009. http://dx.doi.org/10.1007/s10664-008-9102-8

[8] A. Ko, T. LaToza, and M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, 2015. http://dx.doi.org/10.1007/s10664-013-9279-3

[9] A. Jedlitschka and D. Pfahl, "Reporting guidelines for controlled experiments in software engineering," in *Empirical Software Engineering, 2005. 2005 International Symposium on*, Nov 2005, pp. 95–104. http://dx.doi.org/10.1109/ISESE.2005.1541818

[10] A. Barišić, V. Amaral, M. Goulão, and B. Barroca, "Quality in use of domain-specific languages: A case study," in *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU '11. New York, NY, USA: ACM, 2011, pp. 65–72. http://dx.doi.org/10.1145/2089155.2089170

[11] T. Kosar, M. Mernik, and J. C. Carver, "Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments," *Empirical Software Engineering*, vol. 17, no. 3, pp. 276–304, 2012. http://dx.doi.org/10.1007/s10664-011-9172-x

[12] J. Carver, L. Jaccheri, S. Morasca, and F. Shull, "A checklist for integrating student empirical studies with research and teaching goals," *Empirical Software Engineering*, vol. 15, no. 1, pp. 35–59, 2010. http://dx.doi.org/10.1007/s10664-009-9109-9

[13] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: reverse engineering of graphical user interfaces for testing," in *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, Nov 2003, pp. 260–269. http://dx.doi.org/10.1109/WCRE.2003.1287256

[14] J. Feigenspan, C. Kastner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring programming experience," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, June 2012, pp. 73–82. http://dx.doi.org/10.1109/ICPC.2012.6240511

[15] A. A. Neto and T. Conte, "Threats to validity and their control actions – results of a systematic literature review," Universidade Federal do Amazonas, Technical Report TR-USES-2014-0002, Mar. 2014.

[16] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg, "A systematic review of quasi-experiments in software engineering," *Information and Software Technology*, vol. 51, no. 1, pp. 71 – 82, 2009, special Section - Most Cited Articles in 2002 and Regular Research Papers. http://dx.doi.org/10.1016/j.infsof.2008.04.006

[17] T. Reinikainen, I. Hammouda, J. Laiho, K. Koskimies, and T. Systa, "Software comprehension through concern-based queries," in *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, June 2007, pp. 265–270. http://dx.doi.org/10.1109/ICPC.2007.36

[18] N. Niu, A. Mahmoud, and X. Yang, "Faceted navigation for software exploration," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, June 2011, pp. 193–196. http://dx.doi.org/10.1109/ICPC.2011.18

[19] J. Porubän, M. Forgáč, M. Sabo, and M. Běhálek, "Annotation based parser generator," *Computer Science and Information Systems*, vol. 7, no. 2, pp. 291–307, Apr. 2010. http://dx.doi.org/10.2298/CSIS1002291P

[20] D. Lakatoš, J. Porubän, and M. Bačíková, "Declarative specification of references in DSLs," in *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, Sept 2013, pp. 1527–1534.

[21] M. Sabo and J. Porubän, "Preserving design patterns using source code annotations," *Journal of Computer Science and Control Systems*, vol. 2, no. 1, pp. 53–56, 2009.

[22] W. Cazzola and E. Vacchi, "@Java: Bringing a richer annotation model to Java," *Computer Languages, Systems & Structures*, vol. 40, no. 1, pp. 2–18, 2014, special issue on the Programming Languages track at the 28th ACM Symposium on Applied Computing. http://dx.doi.org/10.1016/j.cl.2014.02.002

[23] M. Revelle, T. Broadbent, and D. Coppit, "Understanding concerns in software: insights gained from two case studies," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, May 2005, pp. 23–32. http://dx.doi.org/10.1109/WPC.2005.43

[24] A. Guzzi, L. Hattori, M. Lanza, M. Pinzger, and A. van Deursen, "Collective code bookmarks for program comprehension," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, June 2011, pp. 101–110. http://dx.doi.org/10.1109/ICPC.2011.19

[25] M. Storey, J. Ryall, R. Bull, D. Myers, and J. Singer, "TODO or to bug," in *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, May 2008, pp. 251–260. http://dx.doi.org/10.1145/1368088.1368123