Recording Concerns in Source Code Using Annotations

Matúš Sulír, Milan Nosáľ, Jaroslav Porubän

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia

Abstract

A concern can be characterized as a developer's intent behind a piece of code, often not explicitly captured in it. We discuss a technique of recording concerns using source code annotations (concern annotations). Using two studies and two controlled experiments, we seek to answer the following 3 research questions: 1) Do programmers' mental models overlap? 2) How do developers use shared concern annotations when they are available? 3) Does using annotations created by others improve program comprehension and maintenance correctness, time and confidence? The first study shows that developers' mental models, recorded using concern annotations, overlap and thus can be shared. The second study shows that shared concern annotations can be used during program comprehension for the following purposes: hypotheses confirmation, feature location, obtaining new knowledge, finding relationships and maintenance notes. The first controlled experiment with students showed that the presence of annotations significantly reduced program comprehension and maintenance time by 34%. The second controlled experiment was a differentiated replication of the first one, focused on industrial developers. It showed a 33% significant improvement in correctness. We conclude that concern annotations are a viable way to share developers' thoughts.

Keywords: program comprehension, concerns, source code annotations,

Email addresses: `matus.sulir@tuke.sk` (Matúš Sulír), `milan.nosal@gmail.com` (Milan Nosáľ), `jaroslav.poruban@tuke.sk` (Jaroslav Porubän)

1. Introduction

Programmers developing a program 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, some parts of the mental model are not explicitly expressed in the source code. They are either implicitly indicated in complicated implementation details, or lost.

1.1. Motivation

Suppose a developer encounters the Java code excerpt shown in Figure 1 a). While reading it, the following ideas may gradually appear in his or her mind:

- The method `addProduct` adds some product somewhere.

- As it is in the `Catalog` class, it adds the product to the catalog.

- An addition to a catalog changes its state – the catalog is modified.

- There is a condition checking whether a user is logged.

- To successfully perform this operation, a logged user must be a manager; otherwise an exception is thrown.

While this process may seem natural, it has the following shortcomings:

- The fact that the code modifies a catalog is formed only after reading two parts of the code far apart.

- The information that a "manager account" is necessary is formed after reading a rather lengthy piece of code.

- There is one important piece of information not identified at all: The added product actually appears in the catalog only after the administrator approves it. This information is not apparent from this excerpt, and may be the result of cooperation of many remotely related classes.

a) Source code

```
public class Catalog {
  ...
  public void addProduct (Product product) {
    if (!user.isLogged()
        || !user.getRoles().contains(UserRole.MANAGER)) {
      throw new SecurityException("Insufficient privilleges");
    }
    this.productList.add(product);
  }
}
```

b) Mental model

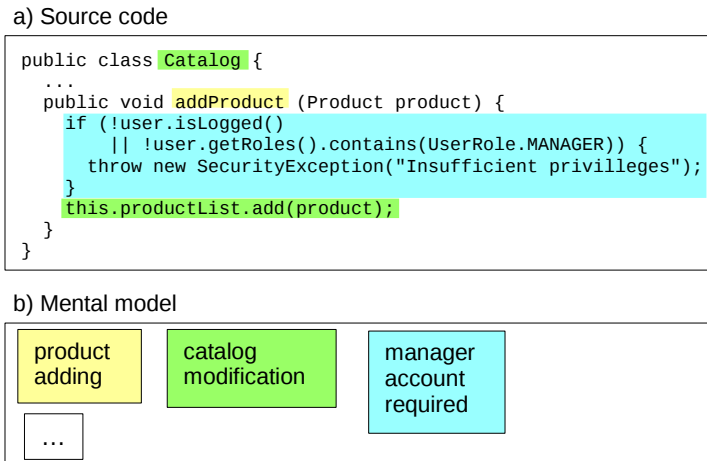| product adding	catalog modification	manager account required

...

Figure 1: An example of unannotated source code and the corresponding mental model

Now suppose the developer originally writing the code, or any other programmer maintaining it, annotated the source code with concern annotations (source code annotations are declarative marks used to decorate source code with metadata [1]), as displayed in Figure 2 a). The developer later reading it would rapidly obtain a mental model containing useful information about the code, without even looking at the method implementation, scrolling through the code to stitch the pieces of information, or even debug it to explain unexpected behavior. Furthermore, it would be possible to find all methods requiring approval in case of refactoring or business policy changes. As annotations are a part of the Java language, no additional tools are required: standard IDE (integrated development environment) features like Find Usages could be used.

1.2. Aim

The purpose of this paper is to investigate the viability of source code annotations as a medium to share parts of developers' mental models. Therefore, we form our **hypothesis** as follows: Annotations created by one group of developers are useful for comprehension and maintenance tasks performed by other developers.

3

a) Source code

```
public class Catalog {
    ...
    @CatalogModification
    @AccountRequired(UserRole.MANAGER)
    @ApprovalRequired(UserRole.ADMIN)
    public void addProduct (Product product) {
        if (!user.isLogged() ...) ...
    }
}
```

b) Mental model

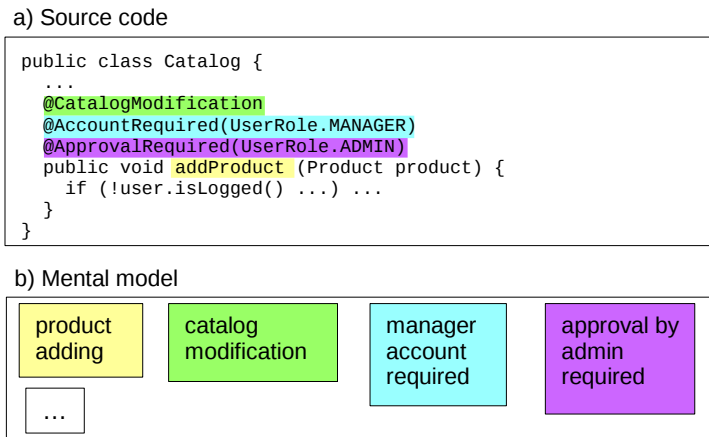| product adding	catalog modification	manager account required	approval by admin required
...			

Figure 2: The annotated source code and the corresponding mental model

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. Then we examine the usability of concern annotations using an observational study and two controlled experiments.

We formulate the research questions as follows:

- **RQ1**: Do programmers' mental models, recorded using concern annotations, overlap?

- **RQ2**: How do developers use shared annotations when they are available?

- **RQ3**: Can 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 method [2].

4

2. Basic Concepts

Before describing the studies, we introduce the notion of concerns, concern annotations, and their relation to a mental model.

2.1. Mental Model

A mental model is "an internal, working representation of the software under consideration" [3]. It is described by its static structures – pieces of knowledge, and by dynamic behavior – the processes forming these pieces. Now we will focus mainly on the static structures.

Since there have been multiple attempts to describe the structure of a mental model, each researcher has its own point of view. Mayrhauser and Vans distinguish these types of structures in their review [3]: *Chunks* are mappings from a label like "sorting" to the program text implementing it, or to a set of other chunks like "comparing" and "swapping". *Plans* are similar, but they may include relationships across various pieces of the code, and the problem domain knowledge. *Hypotheses* are conjectures about parts of the program, which the programmers try to prove or disprove.

Our view of a mental model unifies chunks, plans and hypotheses into a set of concerns. In the programmer's mind, a concern is represented by a short label, as shown in Figure 1 b), and its corresponding source code elements.

2.2. Concerns

A *concern* can be characterized as a developer's intent of a particular piece of code: What should this code accomplish? What am I trying to achieve with it? How would I tersely characterize it? Is there something special about it? Some concerns may be obvious by looking at the code itself (chiefly from the identifiers), but many concerns are hidden. Basically, a concern represents a software system requirement, or a design decision about the code [4].

A concern can be observed in three dimensions [5]: *stakeholders* (from whose perspective is the concern observed – e.g., a programmer); *level* (at what level of construct granularity is the concern expressed, e.g., low-level constructs such as

5

a loop); and *expression* (how is the concern expressed, starting with the mental model and ending with the source code).

Due to these and other reasons, concerns' coverage can overlap. For example, a single method can implement persistence – contributing to the persistence concern; and in the same time it can contain code securing it against unauthorized access – a part of the security concern implementation. Thus, one piece of code can belong to multiple concerns [6], in this case, both persistence and security.

2.3. Concern Annotations

Since a method can belong to multiple concerns, expressing concerns using only identifiers (naming conventions) is not sufficient: naming conventions do not scale well. See Listing 1 for an example using the Persistence keyword in the identifier to express that the method implements persistence, and Secured-Method with OnlyAdmin to specify that it contributes to the security concern by allowing access only to an administrator.

Listing 1: Expressing concerns using naming conventions

```
public addNewUserPersistenceSecuredMethodOnlyAdmin(User user) { ... }
```

It is possible for a class or method in Java to have more than one annotation, scaling well when used for expressing concerns. Therefore, source code annotations (attributes in C# terminology) are a better alternative to identifiers for expressing concerns directly in the source code. Listing 2 shows the example from Listing 1, rewritten using concern annotations. For the persistence concern, we used the `@Persistence` annotation; for the security concern, we used the `@SecuredMethod` annotation. To specify which roles are allowed to access the method, we used the `allowed` parameter of the `@SecuredMethod` annotation.

Listing 2: Concern annotations

```
@Persistence
@SecuredMethod(allowed={User.ADMIN})
```

```
public addNewUser(User user) { ... }
```

For each distinct concern, we recommend creating 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 object storage. Subsequently, we can mark the methods such as `addNewUser()` or `deleteUser()`, and the class `Database` with it. We will call them *annotation occurrences*.

2.4. Concern Annotations Properties

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, and other on them. However, we want to note that concern annotations are not meant to substitute traditional comments, but rather to complement them. Furthermore, annotation type definitions may be also commented by natural language comments if needed.

Still, we expect annotations to have some advantages over comments when considered from the viewpoint of readability. For example, let us consider the thread safety concern and `javax.swing.JPanel`[1] in listing 3. Thanks to its brevity, the `@NotThreadSafe` annotation is easier to spot than its natural language counterpart in the JavaDoc comment (even though we omitted the rest of the comment). This is because the annotation compresses two natural language sentences into a single identifier – `NotThreadSafe`. Of course, the comment provides the same information, however, due to its brevity, the annotation is easier to spot and comprehend. For a more detailed discussion on annotations and comments, we refer the reader to our previous work [7].

Listing 3: The JPanel JavaDoc comment and the annotation

```
/**
 * ... comments ...
 * <strong>Warning:</strong> Swing is not thread safe. For more
 * information see <a
```

---

[1] `http://docs.oracle.com/javase/8/docs/api/javax/swing/JPanel.html`

```
 * href="package-summary.html#threading">Swing's Threading
 * Policy</a>.
 * ... comments ...
 */
@NotThreadSafe
public class JPanel extends JComponent implements Accessible
{
```

Often a concern annotation is just a simple label without a structure – like
the mentioned `@NotThreadSafe` annotation. However, in some cases, param-
eters may be convenient. To avoid natural language usage, enumeration type
parameters are preferred to strings. See the `@SecuredMethod` annotation in
Listing 2 for an example. As we will see in section 3.1.3, developers adhere to
this rule.

2.5. Kinds of Concern Annotations

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

Design annotations document design and implementation decisions like de-
sign 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 the project at all.

2.6. Limitations of Concern Annotations

There are some limitations of annotations as a technique for expressing con-
cerns.

The first one is their granularity. Annotations annotate identifiable program
elements – thus, they can be used to express concerns at the covering constructs
level (methods, classes), partially low level constructs (variables, parameters),

8

and partially conceptual constructs (subsystems represented by packages that can be annotated) as well. However, they cannot be used to associate every line of code with its concern.

Another limitation is the flat structure of annotations. Each concern is expressed only by an annotation name, and optionally also by its parameters. Although annotations support nesting (an annotation within an annotation), they do not support cycles (general tree structures). Right now, annotation type inheritance is not supported, which limits the options to structure concern annotations. Nevertheless, annotations can denote an "is-a" relationship between concerns, like "Observer is-a DesignPattern", if desired. This is possible thanks to meta-annotations[2]: We could annotate the annotation type definition of `Observer` with a meta-annotation `@DesignPattern`.

3. Mental Model Overlapping

In the first study, we examine whether programmers' mental models recorded using concern annotations overlap.

Note that by "developers" (or its synonym, "programmers"), we mean both the original creator of the program, and the persons reading it or trying to understand some parts in order to maintain it. Nowadays, software development is a highly collaborative process and the distinction between a standard programmer and a maintenance programmer is practically nonexistent.

3.1. Method

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

---

[2]Meta-annotations are annotations that annotate annotation types. There are several built-in meta-annotations in Java, such as `@Target`, which is used to annotate annotation types to indicate the kinds of program elements to which the annotation type is applicable. If for any annotation type its `@Target` annotation includes the `ANNOTATION_TYPE` element kind, it means that the given annotation type is applicable to other element types – meaning it is a custom meta-annotation type.

extent their concern annotations overlap.

3.1.1. Materials

For this and subsequent studies, we used EasyNotes[3] – 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.

3.1.2. Participants

This study had 7 participants:

A a researcher and lecturer with a PhD in Computer Science,

B an industrial Java programmer,

C a postdoctoral researcher and Java programmer,

D an associate professor with extensive Java experience,

E, F first-year PhD students,

G the author of EasyNotes.

The author knew the program well, since he was its sole author. The other subjects had performed a small maintenance task on it in the past.

The activity was individual and the participants were not allowed to interact during the experiment.

3.1.3. 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.

---

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

Next, they were asked to create a custom annotation type for each concern they recognized in the application and to mark the classes, member variables and methods with the annotations they just created whenever they considered it appropriate. Participants were not given any special instructions on how to recognize the concerns, nor how to design corresponding annotation types. We decided to give them this freedom so that we could get the best insight into how they would represent their mental model using concern annotations. Furthermore, the participants were asked to comment each annotation type to explain the concern it represents in natural language.

Participants were recommended to use either marker annotations (without parameters, only the name of the annotation is significant), or to use parameters that are not strings (strings would allow using natural language inside of parameters). This recommendation aimed to keep annotations as structured as possible – to avoid parsing of arbitrary strings. In the end, most of the annotation types were just marker annotations. The exceptions included several enumeration parameters (with between 2 and 4 enumeration constants), boolean parameters, and several string parameters. All usages of string parameters were justified; all of them were used in maintenance annotations that required an arbitrary text to be associated with them. For instance, a TODO concern annotation with a task that needed to be done, a Question About Code concern annotation with a particular question, etc. Examples of such parameters can be found in the commit `f52872b`.

The participants were not limited in the time they needed for comprehending and annotating the code, and the time was not measured. However, all the subjects managed to finish the job approximately between 1 and 1.75 hours.

3.1.4. Analysis

Finally, we collected the modified projects and analyzed them semi-automatically, looking for an overlap in annotation types and the use of the annotations on specific elements.

First, we identified concerns expressed by concern annotations. This was

done by the second author of this paper, who manually inspected all concern annotations in each version of the EasyNotes project annotated during the experiment. The process consisted of a manual inspection of annotation types and their natural language comments. For each annotation type, we performed a further inspection of its annotations in the code to check whether the code annotated with concern annotations was consistent with our understanding of the concern. In ambiguous cases, we followed up by having a discussion with the participant – the author of the concern in question. This was also the reason why we decided not to use automatic matching based on ontologies. This thorough process was necessary because participants did not use a shared dictionary to ensure the same terminology, and so there were multiple cases where the same concern was expressed using different terms (see section 3.2.3). We considered using a dictionary a threat to validity, because it would have influenced the participants and the terms in the dictionary would have suggested them the possible concerns.

The second step of the analysis was the semi-automatic identification of concern occurrences overlapping (formally defined in section 3.2.2). For each annotated project, we created an XML snapshot which contained a list of all annotation types with the names of program elements annotated by them. These snapshots were created using the Monitor feature of the SSCE NetBeans plugin[4] which inspects source code using standard Java APIs (application programming interfaces). The document was processed by a short Java program which created a report that contained a list of all concerns, for each concern a list of all distinct program elements associated with the given concern, and for each program element a list of participants who annotated the given program element. Finally, this report was analyzed using XPath to obtain the results presented in the following sections.

---

[4]`http://github.com/MilanNosal/sieve-source-code-editor`

Table 1: 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**

3.2. Results

The number of concerns (annotation types) created by individual participants ranged from 11 to 24, and the number of annotation occurrences from 56 to 140 – see Table 1.

3.2.1. Concern Sharing

We constructed a set of distinct concerns C, i.e., a union of all sets of the concerns recognized by the participants. The size of this set, i.e., the number of distinct concerns, is 46 (the Total row in Table 1).

More than a half of the concerns (26) was shared by at least two participants. We will call them *shared concerns*. A list of all shared concerns is in Table 2.

3.2.2. Occurrence Sharing

We define an *annotation occurrence* as a relation R_{AO} between a set of all concerns C and program elements P. If and only if a concern annotation for the concern $c \in C$ annotates the program element $p \in P$, then $(c, p) \in R_{AO}$. R_{AO} relates the programmer's concerns to source code elements, where this relation is expressed using concern annotations.

We define a magnitude $m(c, p)$ of the relation $(c, p) \in R_{AO}$ as a number of different developers who associated the concern c with the given program

Table 2: A list of all shared concerns

	Concern	Shared by n participants	$EA(c)$	$wEA(c)$	$d_L(K_C)$	$stdev(K_C)$
1	Searching	6	61%	84%	2.22	3.58
2	Note editing	5	60%	77%	4.48	3.24
3	Note change observing	5	50%	67%	9.36	5.2
4	Note presenting	5	21%	42%	6.67	4.14
5	Unused code	4	67%	85%	0.0	0.0
6	Tagging	4	64%	81%	4.63	3.97
7	Persistence	4	41%	60%	11.2	6.18
8	Links	4	39%	58%	0.5	0.5
9	Note adding	4	38%	63%	6.13	3.89
10	Data model	4	36%	63%	3.0	1.8
11	Loading notes	4	35%	62%	7.75	4.76
12	Saving notes	4	31%	54%	7.39	4.23
13	GUI	4	26%	41%	2.0	2.12
14	Note deleting	4	18%	41%	6.25	3.83
15	UI-model mapping	4	4%	7%	8.63	5.24
16	Filter implementation	3	18%	36%	0.0	0.0
17	TODO	3	8%	15%	3.56	3.98
18	Exceptions	2	100%	100%	7.0	7.0
19	Utilities	2	50%	67%	0.0	0.0
20	Model change watching	2	17%	29%	8.0	8.0
21	Filters management	2	12%	21%	4.0	4.0
22	Notes manipulation	2	8%	15%	4.44	4.97
23	Questions about code	2	0%	0%	5.56	3.95
24	Coding by convention	2	0%	0%	3.5	3.5
25	BibTeX	2	0%	0%	2.0	2.0
26	Domain entity	2	0%	0%	4.0	4.0

element p. The magnitude $m(c, p)$ is an integer bounded by the interval $[1, n]$, where n is the number of developers who recognized the concern. This means at least one developer has to associate c with p for (c, p) to be included in R_{AO}.

A relation of *shared annotation occurrences* R_{SAO} is a subset of R_{AO} such that the ordered pair $(c, p) \in R_{AO}$ belongs also to R_{SAO} if and only if $m(c, p) \geq 2$, i.e., there are at least two different developers who associated the concern c with the given program element p.

We constructed a set of all distinct annotation occurrences with a total size of 464 (as noted in Table 1). 128 of them were shared annotation occurrences – they were shared by at least two participants.

We define *effective agreement EA* as the number of shared annotation occurrences divided by the total number of annotation occurrences. Now the computation of effective agreement $EA(c)$ for a particular concern c can be described by the following formula:

$$EA(c) = \frac{\sum_{p \in P} [(c, p) \in R_{SAO}]}{\sum_{p \in P} [(c, p) \in R_{AO}]} \cdot 100\% \tag{1}$$

The formula uses the Iverson bracket notation [8]: $[P]$ is defined to be 1 if P is true, and 0 if it is false.

Overall effective agreement for the whole project can be computed using the following formula:

$$EA = \frac{\sum_{c \in C} \sum_{p \in P} [(c, p) \in R_{SAO}]}{\sum_{c \in C} \sum_{p \in P} [(c, p) \in R_{AO}]} \cdot 100\% \tag{2}$$

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 2. We can consider the effective agreement of a specific concern its quality indicator – to what extent multiple developers agree on the mapping of this concern to source code elements.

In this extended version of [9], we introduce *weighted effective agreement* $wEA(c)$ that puts more weight on annotation occurrences with a higher mag-

nitude. We propose using the following formula to compute $wEA(c)$ for a particular concern:

$$wEA(c) = \frac{\sum_{p \in P} \left([(c,p) \in R_{SAO}] \cdot m(c,p)\right)}{\sum_{p \in P} \left([(c,p) \in R_{AO}] \cdot m(c,p)\right)} \cdot 100\% \tag{3}$$

We expect the weighted effective agreement to be a better quality indicator for concerns recorded in the code. It prefers annotation occurrences with a higher magnitude, which are more likely to be reused by other developers besides the authors recording the concerns. In Table 2 there are wEA values computed for shared concerns.

In Table 2, the concerns are ordered by n, then by EA. Note that if we ordered them by wEA instead of EA, some concerns would change their positions. For example, concern no. 11 (Loading notes) would be moved above concern no. 7 (Persistence). This is due to the fact that annotation occurrences of concern no. 11 have a higher average magnitude, signifying they were agreed upon by more developers than in the case of concern no. 7. The wEA should be therefore superior to EA.

The wEA (or EA) metric can be used to evaluate the overlapping of concerns in source code which are identified by multiple developers. We expect that multiple developers will record concerns in code – author(s) of the source code, but also its maintainers who went through the process of its comprehension. In such a case, the wEA metric and a threshold value can be used to determine which concerns have the potential for comprehension improvement, and which are likely to be useless for other developers. However, there is a need to experimentally determine the value for this threshold. This belongs to the set of future directions of this research.

3.2.3. Lexical Overlapping

We also calculated an average Levenshtein edit distance between the set of keywords that were used for each particular shared concern. A Levenshtein distance [10] of two strings is a number of edit operations (add a character,

16

substitute a character, delete a character) needed to change one of them to the other. Two strings are identical if their Levenshtein distance is 0.

The average Levenshtein distance quantifies the closeness of terms used to express the same concern. Considering K_C to be the set of all keywords for a given concern, we computed its average Levenshtein distance $d_L(K_C)$ and its standard deviation $stdev(K_C)$ using the average Hamming distance formula from [11] adapted for the Levenshtein distance. The Levenshtein distance was computed using the algorithm presented in [12].

The computed values are presented in Table 2. They indicate that even though the concerns were shared, the programmers identified them by various words. Although in some cases (e.g., 'Unused code', or 'Filter implementation') the programmers used the same lexical terms, the overall average Levenshtein distance was 4.7 edits, with the highest values up to 11.2 for the Persistence concern.

The Persistence concern is the boldest example of the nomenclature problem in concern identification. It has the highest Levenshtein distance – it was recognized by four developers who used the following keywords: 'Persistence', 'NoteIO', 'FileManipulation', and finally the developer G who used two terms for the concern to provide finer granularity – 'NotesPersistenceFormat' and 'WorkingWithFiles'. Although in principle the developers think about the program using the same concerns, they do not always think in the same terms. This variability indicates that a concern vocabulary could be beneficial for the purposes of recording concerns in team projects. These results indicate a specific case of the vocabulary problem in human-computer communication [13].

3.2.4. Agreement between Participants

Fig. 3 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.

17

	A	B	C	D	E	F	G
A		8	9	0	4	4	9
B	8		8	4	3	3	8
C	9	8		6	8	6	12
D	0	4	6		5	4	5
E	4	3	8	5		4	9
F	4	3	6	4	4		7
G	9	8	12	5	9	7	

Figure 3: Numbers of shared concerns between individual subjects

On the other hand, G (the author of EasyNotes) shares the most concerns with everyone else. This could mean that the source code author is the best annotator.

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

3.2.5. Concern Kinds from the Viewpoint of Shared Concerns

In section 2.5, we specified three kinds of concerns (or concern annotations) – domain, design, and maintenance. A categorization of the concerns recognized by the participants is in Table 3. Many of the concerns do not have the characteristics of only a single kind. For example, the "Note change observing" concern speaks in terms of the problem domain (the term 'Note'), but is mainly used to record the observer pattern. We always selected a concern kind which we considered dominant for the concern.

In Table 4, there is a summary of sharing across concern kinds. As we can see, 15 concerns are domain, 25 design and 6 maintenance. This corresponds to 33%, 54%, and 13% of 46 concerns in total. This indicates that developers tend to record their perceptions of design more than of the problem domain and maintenance. We hypothesize that this is because source code identifiers usually already express the domain concern of the program element: e.g., a GUI (graphical user interface) method `showNote(Note note)`.

Table 3: Kinds of recognized concerns

Concern	Kind	Concern	Kind
Searching	Domain	Coding by convention	Design
Note editing	Domain	BibTeX	Domain
Note change observing	Design	Domain entity	Design
Note presenting	Domain	Debug logs	Maintenance
Unused code	Maintenance	Currently selected note	Domain
Tagging	Domain	Swing table model	Design
Persistence	Design	Closing notes	Domain
Links	Domain	Exit code	Design
Note adding	Domain	Cancel changes	Design
Data model	Design	Alternative naming	Design
Loading notes	Domain	Optimization	Design
Saving notes	Domain	Design pattern	Design
GUI	Design	Factory pattern	Design
Note deleting	Domain	Abstract keyword	Maintenance
UI-model mapping	Design	Input for searching	Domain
Filter implementation	Design	Domain method	Design
TODO	Maintenance	Aggregator	Design
Exceptions	Design	Handler pattern	Design
Utilities	Design	Template method	Design
Model change watching	Design	Links UI	Design
Filters management	Design	Comments	Maintenance
Notes manipulation	Domain	Notes sorting	Domain
Questions about code	Maintenance	Dynamic content panels	Design

Table 4: Sharing of concerns across concern kinds

Concern kind	Shared	Not shared	Total
Domain	11 (73%)	4 (27%)	15
Design	12 (48%)	13 (52%)	25
Maintenance	3 (50%)	3 (50%)	6

Table 5: Distribution of concern kinds according to developers

Developer	Domain	Design	Maintenance
A	7 (64%)	2 (18%)	2 (18%)
B	7 (58%)	4 (33%)	1 (8%)
C	14 (58%)	7 (29%)	3 (13%)
D	5 (25%)	11 (55%)	4 (20%)
E	6 (50%)	6 (50%)	0 (0%)
F	1 (7%)	11 (79%)	2 (14%)
G	8 (47%)	8 (47%)	1 (6%)

When we consider a ratio of shared concerns to unshared concerns of a particular kind (Table 4), we can see that the most shared concern kind are problem domain concerns (73% of domain concerns were shared). This might indicate that developers are more likely to correctly recognize and identify problem domain concerns in the code. When it comes to code design, it seems that developers tend to look at the code from different perspectives.

3.2.6. Concern Kinds from the Viewpoint of Developers

We have calculated a number of particular concern kinds for each developer in particular. The distribution is presented in Table 5.

Multiple participants slightly prefer domain concerns. However, the concerns recognized by developers D (an associate professor) and F (a PhD student) are mostly oriented towards design and maintenance. We have not stated any hypothesis about the distribution of concern kinds recognized by a developer, but examining the possible causes of a particular distribution in the case of a particular developer seems like an interesting future research direction.

3.3. Threats to Validity

While some hypotheses were outlined in this section, because this is an exploratory study [14], there is a need to properly quantify and statistically confirm or reject them. This suggests an interesting direction in future research.

In this study, only classes, member variables and methods were annotated. This means that a set of possible program elements (section 3.2.2) was limited to elements of these types.

The participants were unable to mark only a part of a method (a set of statements). We therefore studied overlapping with method granularity. With statement granularity, the results could be different. For instance, if two participants annotated disjoint sets of statements inside one method, the overlap for this particular method would be 0% instead of 100%.

The majority of participants had only minor experience with EasyNotes. However, they were given a chance to become familiar with the source code – both before the study and during it.

We performed the study only on a small-scale Java project. A more extensive study should be conducted in the future.

All participants were previous or current Computer Science students. Furthermore, they were all from the same department. This could make their mental models similar to each other.

3.4. 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 potential in recording and subsequent reuse of these data.

Analysis of the average Levenshtein distance between the terms used to identify concerns also showed that a concern dictionary should be used to support comprehension.

An interesting observation is that although in general more design concerns were identified, the most effective sharing was achieved on domain concerns.

4. Concern Annotations Use Cases

The goal of the second study is to find out how third-party developers, i.e., not the annotation authors, use the concern annotations in the source code if

they are available.

4.1. Method

We conducted an observational study.

4.1.1. Materials

We copied all annotation types shared by at least two subjects (from the first study, see Table 2 for a list) into the EasyNotes project. Thus, all concerns that were recognized by at least two participants were included. Then, for each of these annotation types, we merged all its occurrences (recognized by at least one developer) into the project. The resulting annotations were manually edited for consistency by the EasyNotes author. The resulting source code is published as the commit `f52872b`.

4.1.2. Participants

There were three participants:

K a first-year Computer Science PhD student,

L a masters degree student with minor industrial experience,

M a professional developer with 2 years of industrial experience.

None of them had any previous knowledge of EasyNotes.

4.1.3. Procedure

First, all annotation types (concerns) were briefly introduced by the EasyNotes author. The subjects were reminded about common feature location possibilities of the NetBeans IDE.

Each participant 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 [14], i.e., the participants were kindly requested to comment their thoughts when comprehending the code.

4.2. Results

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

4.2.1. Confirming Hypotheses

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

4.2.2. 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 searching for occurrences of the "filtering", "note adding", "note saving" concerns and others. This was considered helpful especially to find whether they did not forget to implement necessary methods for a particular aspect of the note rating feature.

4.2.3. Non-Obvious Concerns

The developers also used annotations to obtain new knowledge about the source code. For instance, the 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`, participant M noticed that the code is used for note adding too.

4.2.4. 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.

4.2.5. Maintenance Notes

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

4.3. Conclusion

Concern annotations have several advantages. The participants stated that compared to traditional natural language comments, annotations are much shorter and thus easier to spot. Therefore, they confirmed our expectations discussed in section 2.4. They are also better structured and usually less ambiguous[5]. The ability to find all usages of a particular concern through standard features present in contemporary IDEs was also appreciated.

Regarding disadvantages, the participant with 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.

5. 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 [15] were used. To present our findings, the experiment reporting guidelines [16] were followed. We customized them to the specific needs of this experiment.

---

[5]This claim refers to concern annotations in the EasyNotes project, commit `f52872b`. In general, an annotation with a single string parameter containing the whole JavaDoc comment could also be considered a concern annotation, but would not be any more structured than the conventional comment. To generalize this advantage, we need to design guidelines or rules on how to define concern annotations (e.g., encourage using enumeration parameters, etc.). We leave this task for our future work.

5.1. Hypothesis

Similarly to Barišić et al. [17], 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$_{null}$: The time to complete program comprehension and maintenance tasks on an annotated project = the time to complete them on the same project without concern annotations.

H2$_{alt}$: The time to complete program comprehension and maintenance tasks on an annotated project < the 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 level of 95% ($\alpha = 5\%$).

5.2. Variables

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

5.2.1. Independent Variables

There is only one independent variable – the presence of concern annotations in the project. The possible values are: yes ("annotated") and no ("unannotated").

5.2.2. Dependent Variables

The first dependent variable, 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. We are interested mainly in the total time, i.e., a sum of times for all tasks.

Instead of measuring only the time, 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 quickly finish [18]. 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 he/she was on a 3-point Likert scale[6]: 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.

---

[6]A Likert scale [19] is a technique to measure agreement or disagreement with statements. For example, a 3-point Likert scale about frequency looks like: "always", "sometimes", "never". Strictly speaking, a true Likert scale must encompass a set of question-scale pairs. However, we will use a common terminology which calls each such scale "Likert".

5.3. Experiment Design

5.3.1. Materials

Again, the EasyNotes project was used. This time, we prepared two different versions: with shared concern annotations (as in section 4), 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. We do not compare concern annotations to traditional comments, since as we noted in section 2, concern annotations are not meant to substitute comments.

During this experiment, we used the NetBeans IDE.

5.3.2. Participants

We used 18 first-year, master's degree Computer Science students as participants. Carver et al. [20] 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.

5.3.3. Design

When assigning the subjects to groups, we applied a completely randomized design [2]. 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 numbered piece of paper. 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.

5.3.4. Instruments

To both guide the subjects and collect the data, we designed an interactive web form[7]. All fields in the form were mandatory, so the participants could not skip any task.

The SSCE NetBeans plugin was used to collect data like session times to an XML file. The participants uploaded this file to the web form at the end of the experiment.

5.4. Procedure

The experiment was performed during three lessons in the same week – the first time with four students, then with 9 and finally with the remaining 5 students. Each session lasted approximately 1.5 hours.

5.4.1. 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. 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 an annotation occurrence to the 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 conveniently supplemented natural language task descriptions.

5.4.2. Tasks

The experiment comprised of:

---

[7]http://www.jotformeu.com/sulir/sharing-annotations

- 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 search 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.abstractModel.UTFStringComparator` used in the EasyNotes project?

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

Although the tasks were not equal in the expected completion time, the comprehension part was comparably difficult, as none of them required extensive understanding of the code.

5.4.3. 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 scale. Finally, the form also contained a free-form question where the participants could describe how the annotations helped them in their own words.

5.5. Results

The measured values and their summary is presented in Table 6.

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.

5.5.1. Correctness

The median correctness for the "annotated" was better than for the "unannotated" group: 80% vs. 60%. See Fig. 4a for a plot[8].

The computed p-value (roughly speaking, the probability that we obtained the data by chance) is 0.0938, which is more than 0.05 (our significance level). This means we accept **H1$_{\text{null}}$** – the result is insignificant. 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 6 (column Correctness), the most difficult question was Q2. Only two participants answered it correctly – both from the "anno-

---

[8]The "correctness" is slightly different than in the original paper [9], as we did not spot less obvious errors in the subjects' code that were identified during the revision of the work for this paper. Because the results for correctness remain insignificant, this does not threaten the validity of the original article.

Table 6: The experiment results for individual subjects

ID	Correctness [true/false]						Time [min]				Efficiency	Confidence [1-3]				Annotations
	The "annotated" group															
	Filter	Cite	Q1	Q2	Q3	Total	Filter	Cite	Test	Total	[task/min]	Q1	Q2	Q3	Mean	useful? [1-5]
1	1	1	1	0	1	80%	12.03	3.00	13.96	28.99	0.14	1	2	3	2.00	1
3	0	1	0	0	0	20%	5.23	2.81	11.13	19.17	0.05	3	2	3	2.67	3
5	1	1	1	1	1	100%	17.43	3.93	6.71	28.07	0.18	3	3	3	3.00	4
7	0	1	1	1	1	80%	7.79	1.43	11.85	21.07	0.19	3	3	3	3.00	4
9	1	1	1	0	1	80%	6.72	5.86	3.87	16.45	0.24	3	3	2	2.67	2
11	1	1	1	0	1	80%	8.41	4.58	4.32	17.31	0.23	3	3	3	3.00	4
13	1	1	0	0	1	60%	20.97	3.48	8.80	33.25	0.09	3	2	3	2.67	3
15	1	1	1	0	1	80%	4.64	1.91	5.22	11.77	0.34	2	2	3	2.33	2
17	1	1	1	0	1	80%	25.08	6.38	6.99	38.45	0.10	2	2	3	2.33	4
Median	**1**	**1**	**1**	**0**	**1**	**80%**	**8.41**	**3.48**	**6.99**	**21.07**	**0.18**	**3**	**2**	**3**	**2.67**	**3**
Std.dev.	**-**	**-**	**-**	**-**	**-**	**22.36%**	**7.41**	**1.67**	**3.57**	**8.80**	**0.09**	**-**	**-**	**-**	**0.35**	**-**
	The "unannotated" group															
	Filter	Cite	Q1	Q2	Q3	Total	Filter	Cite	Test	Total	[task/min]	Q1	Q2	Q3	Mean	useful? [1-5]
2	1	1	1	0	1	80%	2.84	4.72	10.66	18.22	0.22	3	2	3	2.67	NA
4	0	0	1	0	1	40%	8.76	23.45	8.10	40.31	0.05	3	2	3	2.67	NA
6	1	1	1	0	0	60%	18.24	5.23	5.62	29.09	0.10	3	2	1	2.00	NA
8	1	1	1	0	1	80%	6.47	5.59	11.23	23.29	0.17	3	2	3	2.67	NA
10	1	0	1	0	1	60%	4.82	9.64	17.50	31.96	0.09	3	2	3	2.67	NA
12	1	0	1	0	1	60%	11.11	2.09	11.30	24.50	0.12	2	2	2	2.00	NA
14	0	1	0	0	1	40%	30.73	7.19	5.50	43.42	0.05	2	2	3	2.33	NA
16	1	1	1	0	1	80%	12.56	18.39	16.07	47.02	0.09	3	2	3	2.67	NA
18	1	1	1	0	1	80%	25.54	9.59	12.94	48.07	0.08	3	2	3	2.67	NA
Median	**1**	**1**	**1**	**0**	**1**	**60%**	**11.11**	**7.19**	**11.23**	**31.96**	**0.09**	**3**	**2**	**3**	**2.67**	**NA**
Std.dev.	**-**	**-**	**-**	**-**	**-**	**16.67%**	**9.56**	**6.98**	**4.18**	**11.06**	**0.06**	**-**	**-**	**-**	**0.30**	**-**

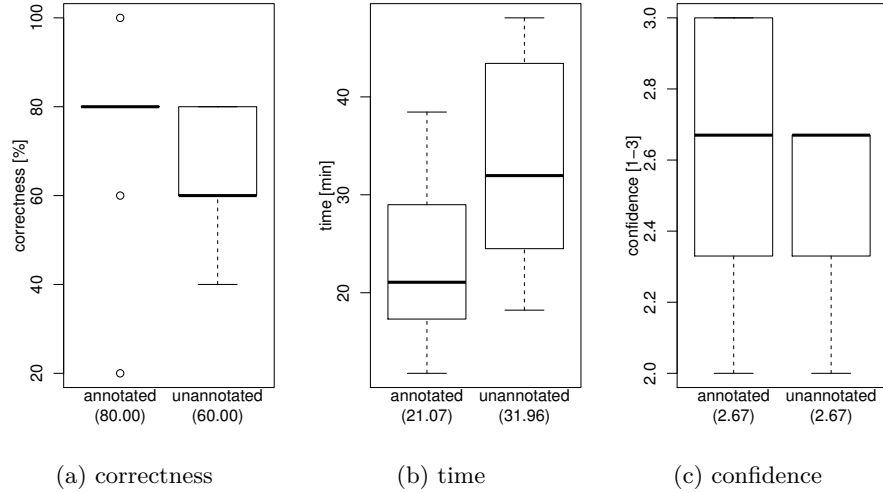| | (a) correctness | (b) time | (c) confidence |

Figure 4: The results of the controlled experiment.

tated" group. The class of interest was not used in EasyNotes at all. This fact was noticeable by looking at the `@Unused` annotation.

5.5.2. Time

The differences in the total time for all tasks between the two groups are graphically depicted in the box plot in Fig. 4b. 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, which is less than 0.05. The difference is statistically significant, therefore we reject **H2$_{\text{null}}$** and accept **H2$_{\text{alt}}$**. The presence of concern annotations improves the program comprehension and maintenance time.

It is possible to see from Table 6 (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 per-

formed tasks (and answers) divided by total time, raised by 89.76% (p=0.0313). This means the time improvement is significant even if we take correctness into account.

5.5.3. Confidence

The median of mean confidence is the same for both groups (2.67), as obvious from Table 6, column Confidence / Mean and Fig.4c. The p-value is 0.1710 ($>$0.05) and therefore we accept **H1$_{\textbf{null}}$**. The effect of concern annotations on confidence was not demonstrated.

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

5.5.4. Other Findings

As seen from Table 6, column "Annotations useful?", concern annotations were perceived relatively helpful by the participants (median 3 on a 5-point scale).

Answers to a free-form question asking how specifically the annotations were useful included: faster orientation in a new project, faster searching (mainly through Find Usages on annotations), less scrolling. The participants also stated: "they helped me to understand some methods", "annotations could be perfect, but I would have to get used to them".

5.6. Threats to Validity

To analyze threats to validity of this experiment, we used also [21] as guidelines.

5.6.1. Construct Validity

Like Kosar et al. [18], 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 – it 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 selected 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 find a statistically significant difference in confidence.

5.6.2. Internal Validity

We divided the subjects into groups randomly. Another possibility was a quasi-experiment (nonrandom assignment), i.e., to divide the subjects evenly according to a co-factor such as their programming experience. However, random assignments tend to have larger effect sizes than quasi-experiments [22].

Full pilot testing with third-party participants was not performed, we 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 of the 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 scale) of English knowledge, as stated by the subjects on the form.

5.6.3. External Validity

We invited only students to our experiment, no professional developers. We can consider 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.

5.6.4. Reliability

The concern annotations training (tutorial) was presented manually by a researcher. However, there were two independent researchers who 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).

5.6.5. 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.

5.7. 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 the 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.

6. Replication of the Controlled Experiment

While replication is a crucial scientific method, less than 18% of controlled experiments in software engineering are replications [23]. In this extended version of [9], we decided to repeat the experiment with slightly modified conditions, i.e., to perform a differentiated replication [2].

6.1. Method

The hypotheses, the independent variable, dependent variables (except confidence), comprehension questions and maintenance tasks were exactly the same as in the original experiment. The recruitment, demographic characteristics of the participants, training and execution details were different. In the following sections, we report the modified aspects.

6.1.1. Dependent Variables

As a 3-point Likert scale for confidence was shown to be too narrow in the original experiment, we decided to change it to a 5-point scale. This allows the subjects to more precisely distinguish between the confidence levels.

6.1.2. Recruitment

This time, we concentrated on developers with industrial experience. Besides industrial experience, the main requirement was to have at least some Java knowledge (so that the comprehension would not be affected by an unfamiliar programming language).

We contacted the developers whom we knew directly by e-mail, or in person. Additionally, we distributed an announcement using a social network and an Internet forum. The participation was voluntary and no compensation was provided.

6.1.3. Participants

20 subjects decided to participate in the experiment. However, one participant completed only the first task, so we excluded him completely. During the analysis of time results for 19 subjects, we found an outlier. After contacting the person, we found out that he was disrupted during the experiment. We decided to exclude this subject, too.

Next, we present the results for 18 subjects who participated in the whole experiment. We divided them into two groups – "annotated" and "unannotated" – based on their IDs. Each group contained 9 participants.

6.1.4. Demographic Characteristics

The average age of the subjects was 26.8 years. All participants except for one were men. Their full-time programming experience ranged from 0 (mostly students with part-time jobs) to 15 years, with a mean of 2.7 years.

We also included an optional question about the participant's company in the form. Overall, the subjects were working in at least 6 companies, so the selection is not biased towards any domain or work practices.

6.1.5. Procedure

The participants were sent a link to the web form[9], which contained all necessary information. They could execute the experiment in their natural settings – at home. Therefore, the training example was not presented by a researcher. Instead, the web form contained a detailed tutorial. A participant was guided to try the IDE features related to the concern annotations himself, according to the examples presented in the tutorial.

After the completion of the experiment, every subject uploaded the zipped project and the XML file with time data to the web form. We then checked the correctness of each implementation.
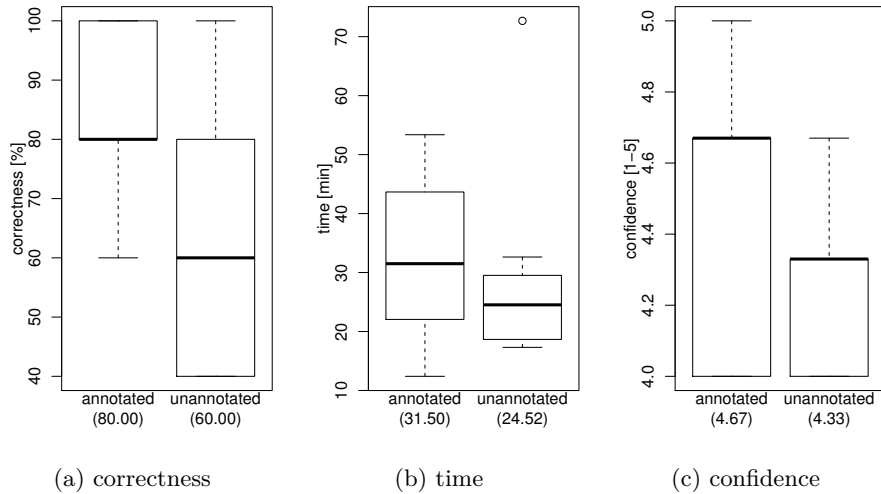
---

[9]http://www.jotformeu.com/sulir/sharing-annotations

Figure 5: The results of the experiment replication.

6.2. Results

The results of the experiment replication are in Table 7.

6.2.1. Correctness

The correctness for the "annotated" group was better: the median was 80%, compared to the "unannotated" one, which was 60% (see Fig. 5a). The p-value is 0.0198, which is less than 0.05. Therefore, the result is statistically significant. In this replication, we reject **H1$_{\mathbf{null}}$** and accept **H1$_{\mathbf{alt}}$**.

6.2.2. Time

The time for the "annotated" group was now about 28% higher – see Fig. 5b. However, the result is statistically insignificant, since p=0.1487. We therefore cannot claim any difference and accept **H2$_{\mathbf{null}}$** in the replication.

If we look at the results in terms of efficiency, i.e., the number of correct tasks per minute, the annotated group performed slightly better (Table 7, column Efficiency). Nevertheless, the p-value is 0.3652, so the difference is insignificant.

Table 7: The detailed results of the experiment replication.

	The "annotated" group															
ID	**Correctness [true/false]**						**Time [min]**				**Efficiency**	**Confidence [1-5]**				**Annotations**
	Filter	**Cite**	**Q1**	**Q2**	**Q3**	**Total**	**Filter**	**Cite**	**Test**	**Total**	**[task/min]**	**Q1**	**Q2**	**Q3**	**Mean**	**useful? [1-5]**
19	1	1	1	0	1	80%	11.94	5.48	35.95	53.37	0.07	5	4	4	4.33	2
21	1	1	1	1	1	100%	19.87	14.49	9.30	43.66	0.11	4	5	5	4.67	3
23	1	1	1	0	1	80%	5.97	23.20	6.69	35.86	0.11	5	4	5	4.67	4
27	1	1	1	1	1	100%	10.67	5.01	28.90	44.58	0.11	4	3	5	4.00	3
29	1	1	1	0	1	80%	19.80	3.22	7.12	30.14	0.13	4	4	4	4.00	3
31	1	1	0	0	1	60%	12.17	3.80	6.08	22.05	0.14	4	5	5	4.67	3
51	0	1	1	0	1	60%	9.12	3.59	6.29	19.00	0.16	4	5	5	4.67	4
53	1	1	1	1	1	100%	9.07	13.43	9.00	31.50	0.16	5	5	5	5.00	3
55	1	1	1	0	1	80%	2.83	3.82	5.75	12.40	0.32	4	4	4	4.00	1
Median	**1**	**1**	**1**	**0**	**1**	**80%**	**10.67**	**5.01**	**7.12**	**31.50**	**0.13**	**4**	**4**	**5**	**4.67**	**3**
Std.dev.	**-**	**-**	**-**	**-**	**-**	**15.63%**	**5.67**	**7.01**	**11.34**	**13.32**	**0.07**	**-**	**-**	**-**	**0.37**	**-**
	The "unannotated" group															
ID	**Correctness [true/false]**						**Time [min]**				**Efficiency**	**Confidence [1-5]**				**Annotations**
	Filter	**Cite**	**Q1**	**Q2**	**Q3**	**Total**	**Filter**	**Cite**	**Test**	**Total**	**[task/min]**	**Q1**	**Q2**	**Q3**	**Mean**	**useful? [1-5]**
20	1	1	0	0	0	40%	7.88	15.09	49.71	72.68	0.03	5	3	4	4.00	1
22	1	1	1	0	1	80%	4.71	3.19	11.93	19.83	0.20	5	2	5	4.00	1
24	1	1	1	1	1	100%	6.78	6.12	16.62	29.52	0.17	4	3	5	4.00	1
26	1	0	0	0	1	40%	4.68	5.61	14.23	24.52	0.08	4	4	5	4.33	3
30	1	0	1	0	0	40%	2.53	9.78	5.00	17.31	0.12	5	4	4	4.33	1
32	1	1	1	0	1	80%	9.27	2.29	17.87	29.43	0.14	4	5	5	4.67	1
50	1	1	0	0	0	40%	5.37	1.71	11.58	18.66	0.11	5	4	3	4.00	1
52	1	1	1	0	0	60%	7.02	8.22	17.39	32.63	0.09	5	4	4	4.33	1
54	1	1	0	0	1	60%	4.21	8.79	4.85	17.85	0.17	4	5	5	4.67	1
Median	**1**	**1**	**1**	**0**	**1**	**60%**	**5.37**	**6.12**	**14.23**	**24.52**	**0.12**	**5**	**4**	**5**	**4.33**	**1**
Std.dev.	**-**	**-**	**-**	**-**	**-**	**22.36%**	**2.08**	**4.25**	**13.34**	**17.30**	**0.05**	**-**	**-**	**-**	**0.28**	**-**

6.2.3. Confidence

The confidence in answering program comprehension questions was almost 8% better for the "annotated" group (see Fig. 5c). The p-value is 0.1417, so the result is insignificant and we accept $\mathbf{H3_{null}}$.

6.2.4. Other Findings

Now we report some of the free-form answers of the subjects:

- "It simplified the orientation in code."

- "It took me a while to understand the meaning of the given annotations."

- "Annotations helped me to relatively quickly identify interesting code parts, especially the annotations close to the problem domain (e.g., note deleting)."

- "If the architecture was more separated, annotations like Filter and UI would be very useful."

- "Annotations served as a correctness assurance to me."

6.3. Threats to Validity

6.3.1. Internal Validity

Since the design of the experiment was almost the same as in the original experiment, no new internal validity threats were introduced.

6.3.2. External Validity

In the original experiment, only students participated in the experiment. In this replication, all participants had industrial experience. We therefore raised the external validity considerably. Furthermore, the participants were not from one company, so they represented a variety of domains.

6.3.3. Reliability

Instead of relying on a researcher to train the subjects, we provided them a tutorial in a textual form. It was exactly the same for all participants, which has a positive impact on reliability.

Table 8: A comparison of the original experiment and the replication. The "change" means the difference between the annotated and unannotated group. Significant results are highlighted in bold.

	Correctness		Time		Confidence	
	change	p-value	change	p-value	change	p-value
Original	+33.3%	0.0938	**−34.1%**	**0.0252**	0%	0.1710
Replication	**+33.3%**	**0.0198**	+28.5%	0.1487	+7.9%	0.1417

6.4. Conclusion

Table 8 summarizes the results of both the original experiment and the replication. If we look at statistically significant results, we come to the following conclusions:

- In the original experiment, the time was improved by about a third.

- In the replication, the correctness was improved by more than a third.

The correctness was improved also in the first experiment, although the difference was statistically insignificant. The only negative change is the increase of time in the replication. However, it is not statistically significant and it is likely the difference was only by chance. Another possible reason is that users without annotations often submitted incorrect results quickly, while the "annotated" group checked their submissions more carefully.

Regarding confidence, no important difference was shown in any of the experiments. We widened the scale from a 3-point in the first experiment to a 5-point in the replication. Although the difference is small and insignificant, the confidence results incline positively to the "annotated" group.

7. Related Work

There are multiple challenges in source code concern (intent) preservation. For a more broader overview of them we refer the reader to our survey presented in [5]. In this section, we will focus on works related to concern annotations with regard to program comprehension improvement.

7.1. Mental Model Overlapping

Revelle et al. [24] also studied concern overlapping. However, their study included only two concern sets, compared to our seven. Their results are positive, too. This further confirms our hypothesis that it is possible to share mental models.

7.2. Annotations

Annotations are a program-level declarative marking technique [1]. They are usually used as an implementation technique for pure embedding [25], since they can be considered an alternative to embedded domain-specific language (DSL) implementation techniques [26, 27, 28]. Today, another 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 [29], to declare references between elements in a domain-specific language [30], or for memory management [31]. This way, annotations can indirectly modify the annotated program semantics. In contrast, concern annotations utilize annotations just as clues for a programmer which are processed by an IDE when needed. If annotations used by some tool record information about the problem domain of the application (e.g., as is in case of feature model-driven generation [32]), they can be beneficial for program comprehension as well.

In Java, annotations can be only applied to packages, classes, member variables, methods and parameters. @Java [33], an extension to the standard Java language, brings annotations below the method level. It allows the marking of 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.

7.2.1. Domain Concerns

Domain annotations (sometimes called semantic annotations [34]) are used mainly in aspect oriented programming to deal with the fragile pointcut problem. Kiczales et al. [35] recommend using them to express semantic properties of the source code that can later be used to bind some code to annotated program elements by the weaving process. Meffert [36] uses semantic annotations to record semantic concerns of the code. He presents a recommendation tool that analyzes source code structure and its concerns to propose design pattern application.

Ji et al. [37] recommend marking the presence of individual features in source code via specially formatted comments. Using a simulation study, they showed that the cost of adding and maintaining such marks is negligible. This suggests that using our approach, which captures also non-feature concerns and uses Java annotations, could be comparably beneficial.

7.2.2. Design Annotations

Design annotations are used to explicitly express design decisions concerning annotated program elements. Hedin [38] uses attribute extensions (structured comments corresponding with annotations) for design pattern formalization. Each design pattern defines roles for program elements that implement it. Hedin suggests using these attribute extensions to record design pattern instances in the code. Sabo et al. [39] use design annotations to preserve the design patterns in source code during the development and evolution of the system. They present a tool that can check whether a design pattern was not violated by evolution. Kajsa et al. [40] build their work on the work of Sabo et al., but add tool support for design pattern artifacts generation and also for design pattern evolution.

7.2.3. Maintenance Notes

Developers often write "TODO comments" like `// TODO: fix this` to mark parts of source code which need their attention [41]. 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.

TagSEA [42] uses a special source code comment format to tag source code elements with maintenance notes and navigation hints. However, their approach relies on a plugin to search and filter the tags.

7.3. Concerns Recorded in Identifiers

Identifiers identify program elements using problem domain terms (or possibly in design terms; e.g., index, Observer, etc.). Deissenboeck et al. [43] notes that 33% of all tokens in the source code are identifiers, and they create up to 70% of the source code characters (measurements of the contemporary version of the Eclipse project). Although identifiers can be used to encode all concerns, annotations provide much better scalability [44].

7.4. Concern Annotations Utilization

Our motivation for using annotations for recording concerns is based on the fact that annotations are first-class citizens of the language and there are standard tools for their parsing [1]. This provides a great potential for concern annotations reuse, and for implementing tooling for their utilization. For example, in [6], we used concern annotations as one of the ways to perform source code projections. Projections allow looking at the 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.

Niu et al. [45] propose the application of HFC (hierarchical faceted categories) on source code. They enable categorizing source code fragments into hierarchical categories on different facets. Their approach requires specialized tools whereas we use source code annotations which have standard IDE support.

8. Conclusion and Future Work

In this paper, we presented an idea of using Java source code annotations to capture and share parts of programmers' mental model, namely the concerns (intents), hence the name, *concern annotations*. Each concern (e.g., "searching", "editing", "GUI code", "unused code") is implemented as an annotation type. Subsequently, all classes, methods and fields relevant to that concern are marked with the given annotation. We defined the metrics *effective agreement* and *weighted effective agreement* to measure the concern overlap of multiple developers. Two case studies, one experiment and its replication were conducted to assess the practical implications of this approach.

A precondition to the usefulness of sharing concern annotations among multiple developers is that their mental models at least partially overlap. In the first study, we showed that this precondition holds. More than a half of the concerns created by one of seven 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.

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, and discover hidden relationships between elements. Concern annotations can also be used as a replacement of traditional TODO comments.

The first controlled experiment, which used only student subjects, 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.

A differentiated replication of the experiment was performed, now focused on professional programmers. A statistically significant improvement of question and task correctness was reached for the "annotated" group. While the time

was slightly worse, the difference was statistically insignificant.

In our studies, the source code was commented scarcely or not at all. An interesting future comparison would consider an annotated program versus a program without annotations, but in both cases with high-quality traditional source code comments (this set-up would better reflect the real world scenario).

Studying the difference of mental models between individual developers, like the causality between the developer's experience and his preferred annotation types, is another interesting future research direction.

Acknowledgment

References

References

[1] M. Nosáľ, M. Sulír, J. Juhár, Source code annotations as formal languages, in: 2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015, 2015, pp. 953–964. `doi:10.15439/2015F173`.

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

[3] A. von Mayrhauser, A. M. Vans, Program comprehension during software maintenance and evolution, Computer 28 (8) (1995) 44–55. `doi:10.1109/2.402076`.

[4] J. Greenfield, K. Short, S. Cook, S. Kent, Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley, 2004.

[5] V. Vranić, J. Porubän, M. Bystrický, T. Frťala, I. Polášek, M. Nosáľ, J. Lang, Challenges in preserving intent comprehensibility in software, Acta Polytechnica Hungarica 12 (7) (2015) 57–75.

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

[7] M. Nosáľ, J. Porubän, Reusable software documentation with phrase annotations, Central European Journal of Computer Science 4 (4) (2014) 242–258. `doi:10.2478/s13537-014-0208-3`.

[8] K. E. Iverson, A programming language, in: Proceedings of the May 1-3, 1962, Spring Joint Computer Conference, AIEE-IRE '62 (Spring), ACM, New York, NY, USA, 1962, pp. 345–351. `doi:10.1145/1460833.1460872`.

[9] M. Sulír, M. Nosáľ, Sharing developers' mental models through source code annotations, in: 2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015, 2015, pp. 997–1006. `doi:10.15439/2015F301`.

[10] V. I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, in: Soviet physics doklady, Vol. 10, 1966, pp. 707–710.

[11] X. Shutao, F. Fu, On the average Hamming distance for binary codes, Discrete Applied Mathematics 89 (1–3) (1998) 269–276. `doi:10.1016/S0166-218X(98)00081-X`.

[12] S. Hjelmqvist, Fast, memory efficient Levenshtein algorithm (Mar. 2012). URL `http://www.codeproject.com/Articles/13525/Fast-memory-efficient-Levenshtein-algorithm`

[13] G. W. Furnas, T. K. Landauer, L. M. Gomez, S. T. Dumais, The vocabulary problem in human-system communication, Commun. ACM 30 (11) (1987) 964–971. `doi:10.1145/32206.32212`.

[14] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2) (2009) 131–164. `doi:10.1007/s10664-008-9102-8`.

[15] A. J. Ko, T. LaToza, M. M. Burnett, A practical guide to controlled experiments of software engineering tools with human participants, Empirical Software Engineering 20 (1) (2015) 110–141. `doi:10.1007/s10664-013-9279-3`.

[16] A. Jedlitschka, D. Pfahl, Reporting guidelines for controlled experiments in software engineering, in: 2005 International Symposium on Empirical Software Engineering, 2005, pp. 95–104. `doi:10.1109/ISESE.2005.1541818`.

[17] A. Barišić, V. Amaral, M. Goulão, 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, PLATEAU '11, ACM, New York, NY, USA, 2011, pp. 65–72. `doi:10.1145/2089155.2089170`.

[18] T. Kosar, M. Mernik, J. C. Carver, Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments, Empirical Software Engineering 17 (3) (2012) 276–304. `doi:10.1007/s10664-011-9172-x`.

[19] R. Likert, A technique for the measurement of attitudes, Archives of Psychology 22 (140) (1932) 1–55.

[20] J. C. Carver, L. Jaccheri, S. Morasca, F. Shull, A checklist for integrating student empirical studies with research and teaching goals, Empirical Software Engineering 15 (1) (2010) 35–59. `doi:10.1007/s10664-009-9109-9`.

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

[22] V. B. Kampenes, T. Dybå, J. E. Hannay, D. I. K. Sjøberg, A systematic review of quasi-experiments in software engineering, Information and Software Technology 51 (1) (2009) 71–82, special Section - Most Cited Articles in 2002 and Regular Research Papers. `doi:10.1016/j.infsof.2008.04.006`.

[23] D. I. K. Sjoberg, J. E. Hannay, O. Hansen, V. By Kampenes, A. Karahasanovic, N.-K. Liborg, A. C. Rekdal, A survey of controlled experiments in software engineering, IEEE Transactions on Software Engineering 31 (9) (2005) 733–753. `doi:10.1109/TSE.2005.97`.

[24] M. Revelle, T. Broadbent, D. Coppit, Understanding concerns in software: insights gained from two case studies, in: Proceedings of the 13th International Workshop on Program Comprehension, IWPC 2005, 2005, pp. 23–32. `doi:10.1109/WPC.2005.43`.

[25] S. Erdweg, P. G. Giarrusso, T. Rendel, Language Composition Untangled, in: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, LDTA '12, ACM, New York, NY, USA, 2012, pp. 7:1–7:8. `doi:10.1145/2427048.2427055`.

[26] L. V. Reis, V. O. Di Iorio, R. S. Bigonha, An on-the-fly grammar modification mechanism for composing and defining extensible languages, Computer Languages, Systems & Structures 42 (C) (2015) 46–59. `doi:10.1016/j.cl.2015.01.002`.

[27] E. Vacchi, W. Cazzola, Neverlang: A framework for feature-oriented language development, Computer Languages, Systems & Structures 43 (2015) 1–40. `doi:10.1016/j.cl.2015.02.001`.

[28] J. Kurš, M. Lungu, R. Iyadurai, O. Nierstrasz, Bounded seas, Computer Languages, Systems & Structures 44 (2015) 114–140. `doi:10.1016/j.cl.2015.08.002`.

[29] J. Porubän, M. Forgáč, M. Sabo, M. Běhálek, Annotation based parser generator, Computer Science and Information Systems 7 (2) (2010) 291–307. `doi:10.2298/CSIS1002291P`.

[30] D. Lakatoš, J. Porubän, M. Bačíková, Declarative specification of references in DSLs, in: 2013 Federated Conference on Computer Science and Information Systems, FedCSIS 2013, 2013, pp. 1527–1534.

[31] C. Stancu, C. Wimmer, S. Brunthaler, P. Larsen, M. Franz, Safe and efficient hybrid memory management for Java, ACM SIGPLAN Notices 50 (11) (2015) 81–92. `doi:10.1145/2887746.2754185`.

[32] R. Táborský, V. Vranić, Feature model driven generation of software artifacts, in: 2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015, 2015, pp. 1007–1018. `doi:10.15439/2015F364`.

[33] W. Cazzola, E. Vacchi, @Java: Bringing a richer annotation model to Java, Computer Languages, Systems & Structures 40 (1) (2014) 2–18, special issue on the Programming Languages track at the 28th ACM Symposium on Applied Computing. `doi:10.1016/j.cl.2014.02.002`.

[34] J. Cachopo, Separation of concerns through semantic annotations, in: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '02, ACM, New York, NY, USA, 2002, pp. 2–3. `doi:10.1145/985072.985074`.

[35] G. Kiczales, M. Mezini, Separation of concerns with procedures, annotations, advice and pointcuts, in: Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 195–213. `doi:10.1007/11531142_9`.

[36] K. Meffert, Supporting design patterns with annotations, in: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, ECBS 2006, 2006, pp. 437–445. `doi:10.1109/ECBS.2006.67`.

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

[38] G. Hedin, Language support for design patterns using attribute extension, in: Object-Oriented Technology, Vol. 1357 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1998, pp. 137–140. `doi:10.1007/3-540-69687-3_29`.

[39] M. Sabo, J. Porubän, Preserving design patterns using source code annotations, Journal of Computer Science and Control Systems 2 (1) (2009) 53–56.

[40] P. Kajsa, P. Návrat, Design pattern support based on the source code annotations and feature models, in: SOFSEM 2012: Theory and Practice of Computer Science, Vol. 7147 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 467–478. `doi:10.1007/978-3-642-27660-6_38`.

[41] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, J. Singer, TODO or to bug, in: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, 2008, pp. 251–260. `doi:10.1145/1368088.1368123`.

[42] M.-A. Storey, L.-T. Cheng, I. Bull, 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, ACM, New York, NY, USA, 2006, pp. 195–198. `doi:10.1145/1180875.1180906`.

[43] F. Deissenboeck, M. Pizka, Concise and consistent naming, Software Quality Control 14 (3) (2006) 261–282. `doi:10.1007/s11219-006-9219-1`.

[44] V. Cepa, Attribute enabled software development: illustrated with mobile software applications, VDM Verlag, Saarbrücken, Germany, 2007.

[45] N. Niu, A. Mahmoud, X. Yang, Faceted navigation for software exploration, in: 2011 IEEE 19th International Conference on Program Comprehension, ICPC 2011, 2011, pp. 193–196. `doi:10.1109/ICPC.2011.18`.