Supporting Program Comprehension for Refactoring Operations with Annotations

Klaus MEFFERT
Technical University Ilmenau
meffert@rz.tu-ilmenau.de

Ilka PHILIPPOW
Technical University Ilmenau
ilka.philippow@tu-ilmenau.de

Abstract. Restructuring a program is a concept that aims at increasing the maintainability of a piece of code by changing its structure. The term refactoring is often used synonymously, especially when the observable behavior of a program should not change when transforming the structure of the software to a more sophisticated level, e.g. by using design patterns. Behaviour-conserving program transformations are difficult as the understanding of both the code to transform as well as the transformation is prerequisite for conserving the conduct of a transformed program. In addition, a transformation should only be executed if certain preconditions apply. To capture the semantic and syntactic details about a specific code fragment, it is proposed documenting them by adding machine-processable and at the same time human-readable annotations. These annotations contain explicit information and could be added to source code by tools evaluating the code, as well as by practitioners. With annotations, it may be possible checking preconditions for program transformations to execute, and gain information necessary for these transformations.

Keywords. Program comprehension. Annotations. Preconditions for program transformations. Refactoring operations. Design patterns. AST analysis. Java.

Introduction

Refactoring operations are an important utility in the toolset of a software developer to keep a piece of code maintainable and evolvable. A positive side effect when using standardized operators appropriately is their documentary power. Typical refactoring operations include simple tasks such as *Encapsulate Field* (where a field is made private and dedicated accessors are added, see [1]) and more complex ones such as applying design patterns (such as *Composite*, see [2]).

To ease the developer's life and to avoid routine work, tools are available for automatically performing refactoring operations, including design patterns. Nowadays, most popular development environments (IDEs) support pushbutton refactorings to a certain extent. For design patterns, various workbenches make it possible to apply at least the static parts (such as interfaces and abstract classes) of a pattern to a given context (i.e. a piece of code).

Completely different tools are capable of analyzing source code to gain valuable information that could be used by the above-mentioned programs to strengthen tool-supported refactorings. Examples include the exploitation of common coding conventions (like speaking names), rule-based techniques, or calculation of metrics (such as the cyclomatic complexity of a program module, compare PMD [3];

cyclomatic complexity is the soundness of a program, measured by the number of path through a program module that are linearly independent).

This paper introduces an approach counteracting the problems described in the next section. Currently, this is a work in progress targeting at researchers. The authors hope that in the future also practitioners will benefit from the solution proposed here.

1. Problem statement

As the resource time normally is very limited within a software project, the developers are mainly required by the stakeholders to implement functionality for the software to create. Making the underlying architecture nicer or investigating in behaviour-conserving transformations is an aspect that not any practitioner is allowed to spend part of the development time with, when it goes after the stakeholders paying the project. In the long-term, such investments could help the developer enormously in reducing the time for maintaining and understanding pieces of code. In addition, a well-structured source code is more fun to work with than patchwork.

Even if a programmer gets to using quite complex operators such as design patterns, the result may not be as expected. For a given context, the patterns applicable must be figured out. This may take quite a long time, considering the huge number of to date documented patterns (compare [2] and [4] among others). [20] illustrates some more problems with the complex nature of design patterns, including the need to figure out the design intentions of a piece of code.

Tools analyzing source code and tools performing operations on code often have no common interface to communicate thru. Therefore the information gained during analysis processes of code fragments must be “passed” to executing tools manually, namely by the developer. Transporting such important data to an end-user tool is possible by controlling a graphical user interface with the mouse or the keyboard. It implicitly involves the developer translating a message given by an analyzer program into a concept compatible with and end-user tool. Figure 1 illustrates the processes concerned (the *i*-symbol indicates the flow of information).

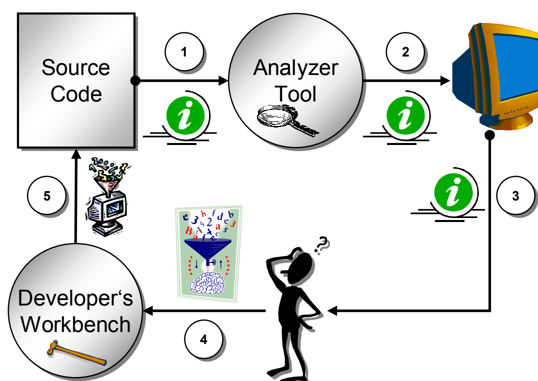

Figure 1: Analyzing and modifying source code

For instance, if an analyzer examining a piece of code (step 1 in figure 1) reports “Class X has Cyclomatic Complexity of 17” (step 2) then the developer must figure out

what that message means (step 3). On the other side, the used IDE may be able to perform a refactoring such as *Consolidate Conditional Expression* (step 5). The developer could possibly be competent enough to know that applying the mentioned operation onto this and that method would help in getting a better cyclomatic complexity (step 4). Anyway, preconditions have to be considered that should be satisfied before applying a refactoring operation. If a precondition was violated, the result may not be as desired and may require rolling back the whole operation.

Besides tools, a practitioner may be in the position to make statements about a code fragment, which may be valuable for executing specific operations onto that code. A simple example of such a statement could be "method too long" (method contains too many statements). A tool may not be able finding the same in any case. Just think of company policies for certain types of methods, e.g. utility functions that should be shorter than original business logic. To continue the example, a developer could mark a method as utility function, which allowed tools to handle it differently.

This paper points at aiding the developer in getting different refactoring tools play together. A second aspect is enabling practitioners to add information to a piece of code that may be relevant for tool-based refactorings. As these may seem two different aspects, the solution proposed will be the same. The proposal has been worked out by asking the question: How could a mechanism look like that is able to support tool-based refactorings?

2. Related Work

This section describes existing approaches that try to support the use of refactoring operations during the development process and to resolve the problems discussed previously.

2.1. Refactoring

The idea of [7] is to provide the developer with a set of annotations (in the form of [6]) allowing to relate source code to architectural descriptions. For instance, the annotation *@Component* links a class to a component given by the architectural description. The annotations *@Part*, *@Port*, *@Connects*, *@Connector*, *@AddPart*, and *@RemovePart* are introduced as well. With them, the relations between architectural entities should be expressible and a coupling between architecture and source code could be established. The authors say that this permits to predict code changes for a refactoring and to check consistencies afterwards.

LePUS [14] lets define design patterns in a formal way. It is based on PROLOG and supports the additional activities validation, application, recognition, and discovery of design patterns. [8] provides means for creating pattern templates as well as applying them generatively to frameworks. [15] targets on the description of patterns and only support a subset of them. [17] offers a meta-model suitable for defining patterns and allowing to apply them later on.

[16] tries to select applicable design patterns by comparing two legacy code versions seen as related. [19] allows to search for patterns by previously defined intents. The basis for that is the DRIM (Design Recommendation and Intent Model), capturing the design decisions of existent code fragments.

Jackpot [18] searches Java source code that conforms to generic rules. Each rule contains a Java-statement to be matched and if this match is possible for a code fragment, the rule specifies how to transform the fragment. The transformation is executed by Jackpot and the user is not permitted to contribute to that. The rules to be defined operate on an AST (abstract syntax tree), they are static and can be nested.

2.2. Analyzing Tools

The tool PMD [3] is a rule-based analyzer for Java source code containing many rules. For example, the rule *UseSingleton* suggests introducing the *Singleton* [2] pattern in case only static methods are found for a class. The rule *CyclomaticComplexity* computes the metrics with the same name.

Checkstyle [9] is another analyzer for source-code, aiming more at adhering to coding standards. Also included are rules for checking the design of classes, such as *DesignForExtension* (which inspects if classes are designed for extension by analyzing method signatures).

2.3. Annotations

Besides [7], there are approaches which utilize annotations and which are not related to refactoring. JML [11] and XDoclet [8] introduce annotations based on Javadoc for different reasons, but under the same concept. The concept always is using annotations to let the user manually define syntactic constraints and properties. Examples for such are the multiplicity or the type and value range of an entity. Both methodologies are restricted to relate annotations to declarations, not to statements. They are targeting onto purely low-level technical intentions, not onto higher-level meanings. On the other side other approaches exist that process natural language in a different context (such as [13]), providing much more informal and harder to interpret constructs than annotations.

The Java Specification Request 175 (JSR 175, see [6]) introduces annotations as valid language constructs for Java. The JSR allows to define annotation types as context-free interfaces for concrete annotations to apply onto source code. These interfaces include the definition of valid scopes for the inherited annotations as well as parameters, and default values for parameters, among others. Valid scopes are declarations and other annotations, but not single statements. An *annotation processing tool* (short: *apt*) provides an API for evaluating annotations within code.

2.4. Matching semantics

Compared to the process of matching pattern intentions with source code annotations, question-answer systems like [13] have higher requirements. At first, they rely on natural language queries, not on quite simple statements (such as annotations). Then, those queries have to be matched with facts, instead of with other statements. At last, such systems must find out themselves the semantic meaning of queries and statements.

3. Proposed solution

This section presents a new approach to resolve some problems with refactoring operations and design patterns as described in section 1. The problem with refactoring is the missing tool-support for determining possible operations and then applying a chosen operation based on that information. Besides, a developer is currently not able to add information to source code that can be processed by a range of standard tools (such as IDEs). A proposal for Java is made how to overcome these difficulties. The main idea is to reduce the developer's effort necessary to find appropriate refactoring operations. This is assumed possible by adding a communication layer manifested in source code. The layer's job is to transport information from one entity to another and finally to an IDE executing refactorings. An entity is an analyzer tool or workbench, or a practitioner. The diagram in figure 2 gives an overview of the processes involved.

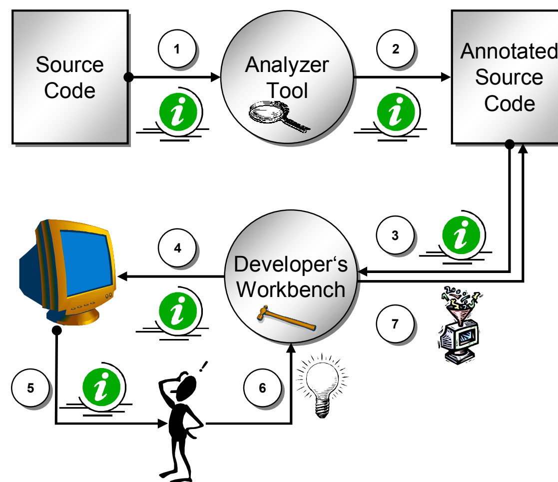

Figure 2: Exploiting annotations for refactoring

The steps displayed in figure 2 are (the *i*-symbol indicates the flow of information):

1. Perform a traditional analysis of source code.
2. Results from the previous step are documented by adding annotations appropriately to the analyzed code.
3. An IDE evaluates the annotated elements and thus exploits the findings from the first step.
4. The information discovered via annotations is displayed to the developer, if possible with sophisticated refactoring suggestions.
5. The developer gets a picture about the current situation, helped by numerous precise analysis reports.
6. After evaluating the presented data and possible suggestions, a decision is made by selecting the operation to be executed in the IDE.
7. To execute the chosen operation, the IDE makes use of the information provided for it via annotations.

In the above steps 1 and 2 and in figure 2, it is not mentioned that also the developer could add annotations to source code. As will be shown, this is necessary in some cases. The main ideas of the proposed approach are in summary:

- Add syntactic information to source code that otherwise had to be extracted by AST-analysis.
- Add semantic information to source code that cannot be figured out automatically.
- Employ annotations as ambassador of reliable information.
- Use the same syntax for syntactic as well as for semantic annotation.
- Use a form that a human can read like natural language and that can as well be processed by a machine.
- Connect each piece of information with a program element (scope).
- Define valid scopes for each annotation.
- Connect syntactic and semantic information within the source code with such in the pattern documentation.
- Determine the applicability of a pattern by trying to match syntactic and semantic information between source code and pattern documentation.
- Make up an annotation as pragmatic as possible, i.e. by not relying on pseudo natural language expressions, but by combining two parts in one annotation. One part represents the machine-processable information; the other part is the documentation for the developer.
- Syntactic and semantic information useful for selecting a pattern could be beneficial when it comes to applying it.

The next section gives three motivating examples why adding specific information to source code may be helpful. The other ideas mentioned above of the paper will be discussed later on.

3.1. Motivating examples

It follows an example piece of code for which enrichment with semantic and syntactic information would help in refactoring the code. Having a loop over a data structure, a Java code snippet could look like this:

```
java.util.List list;
... // fill "list" with values
for (int i=0; i<list.size(); i++) {
    Object element = list.get(i);
    ... //evaluate "element"
}
```

Figure 3: Classic iteration over a list

For the above code, the pattern *Iterator* [2] would be applicable, because the following preconditions are met respectively assumed to be fulfilled:

1. There is a loop over a list.
2. `list` is a sequentially accessible list type.
3. The index variable `i` runs continuously over the whole list.
4. There is no modification of `i` inside the body of the *for*-loop.

5. There is no modification of a list element inside the loop (as *Iterator* can usually not handle this, except for the removal of the current element of the iterated list).
6. The developer sees and advantage in applying *Iterator* instead of using the conventional *for*-loop.

The first five of these statements are syntactic information reasonable by AST-analysis. The aspects four and five may be more difficult to reason. They are an example for information enrichment that could be contributed by a developer. Aspect six definitively is one strong candidate for semantic information to be added manually.

Given the six above-listed syntactic and semantic information, it would be easy for a tool deciding about the applicability of *Iterator* and in turn, suggest it to the user. Applying the pattern could also rely on these pieces of information. For operations that are more complex additional enrichment of the code with information may be necessary, which is also possible via the universally functional annotations.

If at least one of the above conditions does not apply, then *Iterator* would not be applicable (unless changing the code is thought of). It could be considered displaying the violated and passed constraints to the developer. Then, the developer could decide whether to refactor the source code to conform to the constraints. In other cases it would show that for good reason, one or more constraints are violated in the code. For instance, if the first list element represents an exposed element that could always be skipped then the loop would be starting at index one (and not at zero) intentionally. The number of violated constraints could be an indication for how close the miss for a pattern is and whether it could be seen as a near miss.

Another example shows that without context, the sense of a piece of code cannot be reasoned in general:

```
public int increment(int index) {
  return index + 1;
}
```

Figure 4: A method without context

The limitation for the code in figure 4 is to be only able to recognize the low-level meaning of the method, which is to return the increment of a given integer number. The sense, i.e. higher meaning, of the method still is in question. Assumed the calling code writes as follows:

```
int i = 0;
while (i < list.size()) {
  String s = (String)list.get(i);
  doSomethingWith(s);
  i = foo(i);
}
```

Figure 5: Context of a called method

Then, the higher-level meaning, or sense, as we call it, could be reasoned easily by a human being and maybe by an algorithm relying on a database of rules and sample code blocks (with each of them attached semantic meaning manually). An annotation bound to the code from figure 5 could express the meaning of the code block, namely

to loop over a list from begin to end. This annotation would make any AST-analysis superfluous that targets at ascertaining the sense of the above given code.

A third example shall be given to show how to manifest information in source code that would otherwise not be reasonable or only with great difficulties. In [10] some code smells along with proposed refactorings are listed. One smell regards similar classes with different interfaces. An annotation could tell the system which classes are similar and thus could be refactored according to [10] (e.g., by renaming methods, moving methods or unifying interfaces with the *Adapter* pattern from [2]).

As said, certain information could be coupled with source code via annotations. The next section introduces annotations and explains how to implement them.

3.2. Annotations

The intention of annotations is to add information to a piece of code. More exactly, an annotation allows correlating a piece of information with a specific program element. Thus, for a program element syntactic or semantic information could be added. To correlate annotations with program elements, the former ones are written above the latter ones. Section 3.2.3 explores this in more detail.

Syntactic information results from AST analysis respectively from the understanding a developer has about a piece of code. Semantic information is more difficult to gain. Normally, it can only be reasoned by evaluating the context of the code fragment in question. For example, the purpose of calling a method named *notify* in class *B* from class *A* could be the notification of class *B* about a significant state change in class *A*. A program would normally not be in the position to uncover that, except by relying on a defined list of speaking names or by comparing dedicated code blocks. The main advantage of annotations is that the knowledge persisted by it is assumed to be reliable (section 3.4 describes how to guarantee that trustworthiness). With a set of such reliable statements, reasoning about a refactoring operation is possible without fuzzy assumptions. The question only is if any annotation necessary is present.

In general, an annotation could be added to source code either by an analyzing tool or by a developer. The question is how an annotation should look like to have the following properties:

- Precise and non-ambiguous,
- powerful: any code fragment should be annotatable; any meaning should be expressible (which seems not possible),
- machine-processable, verifiable
- human-readable, and
- easy to maintain.

A difficult aspect seems to be to make an annotation human-readable and at the same time machine-processable. Furthermore, a machine should know exactly what is expressed by an annotation. It should not be necessary to figure out with a certain fuzziness what an annotation actually means. Refactoring operations are a narrowed domain. To make statements about a given set of operations requires collecting the statements necessary to describe each known operation. As each refactoring operation (including design patterns) has a limited number of purposes (maybe only one) and a limited set of constraints conditioning whether the operation is applicable, the required set of annotations to handle them can also be limited. This is in contrast to natural

language that is domain-independent and is used to describe completely different aspects from numerous domains.

3.2.1. Composed annotations

A possible way to meet the specification of both human-readability and machine-processability is to compose an annotation of two parts: One part is intended for the machine evaluating the annotation and one for the developer reading it in the source code as informative comment. The following figure depicts the idea schematically:

Figure 6: Machine- and human-readable part of the same tag

The left side of the above tag contains the machine-readable information, and the right side the part for the human being. A real world example is a price tag that is scanned by service staff as well as read by customers.

As a practitioner is not required to understand the part for the machine, this part can be constructed conveniently. To get distinguishable elements simplest, we propose using unique identifiers (IDs). Each ID represents a single annotation. The machine-processable part of an annotation is abbreviated with MID in the following. The essential MID's constituents are:

1. ID, and
2. parameters (for each parameter: name, type).

An ID could be represented by a number, beginning at one and incrementing with each new ID needed. In turn, the human-readable part of the annotation is abbreviated with HID in the following. It consists of:

1. ID of the MID the HID belongs to,
2. natural language description including parameters, and
3. language of the description.

The parameters included in the description of the HID have the same name as in the MID. They are marked by a special prefix (we chose the at-sign). The language parameter of the HID allows supporting descriptions in multiple languages for the same MID. An example of a (MID, HID)-definition is:

```
MID:  1234(("var", String), ("class", String))
HID:  MID=1234, "The local variable @var is not used in
      class @class", English
```

Figure 7: Definition of an annotation

Please notice that both parameters ("var" and "class") are redundant to some extent in the above case as they could be figured out by evaluating the annotated element and the general scope (e.g. the class the program element is located in). As this requires additional effort and could be problematic having more than one program element in a single line, the information is included. The types used in figure 7 (here: *String*) and the

languages (here: *English*) are constants that have to be defined once. The connection between MID and HID is realized by the reference from the HID to the MID with the MID's ID as well as by using the same parameter names in MID as well as in HID.

The advantage of the above-described composition of annotations over annotations representing a single statement that is required to be both human-readable and machine-processable is discussed in the next section. After that, the usage of annotations in source code is shown, relying to the definition shown in figure 7.

3.2.2. Advantage of composed annotations over dual annotations

A composed annotation contains a MID and a HID. A dual annotation contains only one statement, e.g. following JSR 175 [6], [11] or [20]. A dual annotation is called dual because it has a twofold aim. The first aim is to provide a human-readable expression. The second aim is to make this expression machine-processable.

The advantage of composed annotations is that one can concentrate on defining annotations that transport a certain statement, such as "a variable v is not used within a class" or "object creation is too slow for object o". There is no need finding an expression that makes that statement <u>and</u> obeys to a natural language-like style processable by a developer as well as by a computer. Dual annotations always require keeping these three aspects in mind, namely

1. to make a statement
2. that is human-readable, and
3. machine-processable at the same time.

With composed annotations these three aspects can be worked out independently as they do not depend on each other because of the introduction of MID and HID that are independent regarding their concept.

One possible advantage of dual annotations is similarity between two of them. To express that a variable *v* is not used within a class *c*, the term `notUsed(localVariable(v))` could be added to the class *c* as a dual annotation (compare to figure 7). Then, a parameter *p* not used within a method *m* could be expressed similarly by `notUsed(methodParameter(p))`, added above method *m*. The apparent advantage of dual annotations in this case over composed annotations is the self-explaining similarity between the two `notUsed(...)` expressions (because of the same outer function and only differing inner functions). Similarity between composed annotations must be defined explicitly, e.g. by building a resemblance table containing pairs of similar expressions. The advantage of such explicit definition is the opportunity to add a description and additional information to each similarity relation. To accomplish that for dual annotations, almost the same lookup table would be necessary.

3.2.3. Annotations in source code

Each annotation considers a single program element or a block of them. Annotations could be realized as language constructs (see [6]) or as comments (compare [8], [11] and [20]). As [6] lacks some capabilities necessary here (such as annotating arbitrary program elements, which [7] also criticizes), we chose to add them as comments. To distinguish annotations from ordinary comments, a special prefix is added to each comment introducing an annotation. Javadoc-Annotations fit well into the concept of

the Java platform. They have been broadly used in earlier Java versions as a custom-made mechanism. XDoclet [8] is one popular tool incorporating this. Deciding to express annotations as well-defined comments is more than syntactic sugar. For with comments it is no matter preserving the behaviour of the annotated code, thus keeping it possible using current tools, compilers, and interpreters.

To distinguish an annotation from an ordinary comment, each annotation begins with the at-symbol `@`. This work proposes distinguishing between syntactic and semantic annotations. The former annotation starts with *@syntactic*, the latter one with *@semantic*. Syntactic annotations contain statements that could be reasoned by AST-analysis, semantic ones conclude about an intention of an AST-branch. The Java platform provides Javadoc-comments that can spawn over multiple lines. A Javadoc-comment begins with the characters `/*` and ends with `*/`. An example of an annotated local variable in Javadoc-style with the mentioned prefix *@syntactic*:

```
/*
 * @syntactic
 * MID: 1234("surname", "x")
 * HID: Local variable surname is not used in class x
 */
private String surname;
```

Figure 8: Applied syntactic annotation

This example shows the concrete usage of the annotation defined in figure 7. Compared to that definition, the applied annotation does not contain language information. The reason is that the locale is determined when a project is opened in the IDE. Otherwise it would be necessary to include any HID-description defined for a certain language.

3.2.4. Scope of annotations

As the previous section showed, the scope of an annotation can be figured out by just considering the element directly following the annotation in question. An annotation could virtually be placed ahead of any program element, including declarations (package, class, method, and field) and statements (single, block). By this definition, each applied annotation has exactly one concrete scope, whereas an annotation definition could potentially contain several valid scopes. The following figure 9 illustrates the wide range of scopes possible for an annotation.

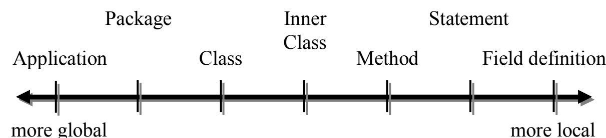

Figure 9: Locality of program elements

For any of these program elements, syntactic and semantic information could be added by annotations. In the case of applications and packages it might be necessary providing additional files as class files originally used with Java may not be sufficient.

In figure 8 the scope is the declaration of the private variable *surname* with type *String*. It is easy to see that an annotation concerning a local variable does not make

sense when applied above a method, a class or a statement creating an object. Thus, the possible scopes must be defined for each annotation. For example, the annotation with MID 1234 makes sense for what is given in figure 8. It also makes sense for a static declaration, e.g.: `private static String surname`. This static declaration is different from the nonstatic one regarding its AST representation. Therefore it is vital defining the annotation's valid scopes in a sophisticated way.

As [20] proposes, isomorphs could be used for defining annotations' scopes generically. An isomorph is a program element that is seen equivalent to another one. For instance, a method throwing an exception could be isomorphic to one throwing no exception when considered for the applicability of an annotation. In this case, both method signatures are seen equivalent with respect to their AST. One task to go for each annotation is to identify possible scopes exemplary. After that, for each example discovered, isomorphic representations could be found. That way, the valid scopes for each annotation can be determined. Arbitrarily grouping definable program constructs could be supported by a technique called *Programming by Example* (see [12]). Currently a huge number of groups of operator, statement, and declaration types has been distinguished by us, resulting in many permutations. These types include:

- **Interface types**: Marker vs. normal interface
- **Method types**: Constructor vs. instance vs. static method; Visibilities private, protected., package, public; Return type void vs. other; Number of parameters
- **Class types**: Abstract vs. concrete; Position in hierarchy; Visibilities; Function of class: data container vs. utilities vs. persistence
- **Attribute types**: Dimension; Function of attribute; primitive vs. other
- **Statement types**: Assignment: yes/no; Method call vs. construction vs. calculation vs. key word etc.; Block vs. single statement
- **Operator types**: Arithmetic vs. Boolean comparison vs. Boolean condition vs. logical vs. bit-shift

For each of these groups, isomorphs can be created separately. After an isomorph has been defined, it can be assigned a unique name (sort of ID) so that it can be referenced from an annotation's definition. Each unique isomorph name is connected with a routine that determines for a given program element whether it is a valid scope or not. By allowing to group scopes with Boolean functions, the isomorphs defined do not need to cover any permutation atomically. It is enough if there is a Boolean combination from two or more isomorphs that express the needed scope appropriately.

Annotating a block of program elements is not always possible by the above-mentioned mechanism, namely by determining the annotated element by the annotation's position. To allow specifying a number of program elements for an annotation, another annotation is introduced. It is called end-marker and always has the same form. The end-marker is not a composed annotation because it is so simple that human-readability is not a question. The principle is that the end-marker references the corresponding annotation by a concrete ID. The concrete ID of an annotation is a running number that is given for each applied annotation and that is not equal to the ID from the MID-part. The end-marker is defined as given in the following example showing the annotation of an arbitrary block of subsequent program elements:

```
/*
 * @syntactic
 * ID=4711
```

```
 * MID:...
 * HID:...
 */
int x = getAmount();
doSomethingWith(x);
print(x);
/* @end-marker ID=4711*/
```

Figure 10: End-marker annotation

The concrete ID used here is part of any applied annotation. Other properties of annotations are added in the next section.

3.3. Types of annotations

Until now, syntactic and semantic annotations have been distinguished. Syntactic annotations may contain anything that can be determined by AST-analysis. For Java, there is a limited number of valid AST elements and properties per element. Thus, the data embodied by AST elements is of general interest to be expressed by annotations. For syntactic annotations, several aspects could be distinguished:

1. Diagnosis,
2. suggestion, and
3. requirement.

A syntactic annotation is marked by the initial keyword *@syntactic* followed by a subsection as given above. To express a diagnosis, the annotation would start with *@syntactic diagnosis*. An example for a diagnosis is the identification of an isomorph for a given program element. A suggestion could include a refactoring operation, and a requirement could contain the refactoring operation to be applied.

Semantic information could result from different perspectives to look on a code fragment. This variety could be concerned with the aspects

1. intention,
2. requirement,
3. problem respectively identified antipattern or code smell,
4. conclusions,
5. diagnoses, and
6. suggestions.

An intention assigns a meaning to a code fragment. A requirement states what should be done, this could include applying a design pattern or refactoring operation. A problem reveals what may be wrong with the code. Conclusions are asserted as correct statements (e.g. by AST-analysis or as an opinion of a component developer). A diagnosis is a conclusion that has been evaluated in context it was made for. From a diagnosis a suggestions can follow. Semantic annotations are marked by the initial keyword *@semantic*, followed by a subsection as given in the previous enumeration. To express an intention, the annotation would start with *@semantic intention*.

3.4. Data included with annotations

In the previous section, the examples already mentioned some of the important data included with annotations. This section summarizes about that and adds further data that is part of an applied annotation. At the end of the section, a short summary of the data included with annotation definitions is given.

To reproduce which entity (tool, developer) added an annotation, the name of the annotator should be included with an applied annotation. For the tool PMD [3] this may be the string "tool-pmd", for a developer named Henry, it may be "developer-henry". This data may especially prove useful in case of diagnoses and proposals.

An additional free-text field allows the annotator to add an unevaluated comment to the annotation. E.g., Henry may add why he annotated or PMD may add the name of the rule that led to the annotation.

At first, an annotation, as proposed in this paper, is an ordinary comment. It depends on the tool integrated in the development process how annotations are interpreted. Initially, an annotation is put above a certain program element. If that element changes later on significantly, the annotation would probably fit not any longer. This is because the annotation attaches an intention to the annotated element. Thus, such mismatch should be identifiable. This could be accomplished by computing a hashcode value for the annotated element and adding that value to the annotation. For different scopes (determined by the annotated element), different elements would be hashed. For an annotation stating "all methods of the class are static" it is necessary hashing all method signatures of the class (and maybe of its super classes). For an annotation referring to a method, the elements to be hashed would be the method signature and body. Assumed an IDE supports annotations and the source code would only be edited with such an integrated editor, the editor would be able to notice itself that an annotated element has been changed. This would be possible at least in most cases, for indirect dependencies the case would be different. In summary, each applied annotation is proposed to contain:

- Initial marker: *@syntactic <subsection>*, *@semantic <subsection>* or *@end-marker*,
- ID of applied annotation,
- hashcode for applied annotation (including annotated elements),
- ID of annotator,
- optional comment from annotator,
- MID: machine-processable part of annotation, and
- HID: human-readable part of annotation.

All fields but the last two are called header fields. They contain metadata about the annotation. For annotation definition, any information contained has been mentioned before. This data includes:

- Type of annotation (i.e. syntactic or semantic),
- subsection of the type (i.e. intention, requirement etc.),
- MID,
- HID, and
- list of valid scopes.

The list of valid scopes is built up by the unique names of scopes defined, and delimited by Boolean operators (*and*, *or*, *not*) and brackets. Such a list could be:

```
ANY_NONPRIVATE_METHOD
or (ANY_PRIVATE_METHOD and ANY_NONSTATIC_METHOD)
```

In this example, the annotation could be added to any method that is not private or private and nonstatic.

3.5. Defining annotations

Before annotations can be added to source code, they have to be defined. Several alternatives are possible, including the definition of annotations

a) from scratch,
b) by transforming a sample code, and
c) by retrieving definitions from a public repository or a service provider.

The first alternative is suitable in case it is known in advance which annotations are needed in the context of possible design patterns. This may apply for design intentions a pattern expresses, such as “Ensure there is exactly one instance of an object at a time” (as with *Singleton*). Then, the scope for these intentions has to be easy to uncover, too. This is because an annotation definition contains a semantic statement as well as valid scopes. For many of the refactoring operations Fowler [1] mentions the design intentions and the corresponding valid scopes are easy to identify.

Alternative b) is the most complex and defined way of retrieving annotation definitions for a given pattern. Basically, a suboptimal piece of sample code serves as input. Subsequent manual transformations apply the pattern, each transformation giving the chance of finding new annotations for the pattern. After each transformation step, the developer figures out whether adding one or more annotations to program elements is necessary. An annotation is regarded necessary in case it adds information to the code that is not included with the code’s AST and that is needed for reasoning about the applicability of a pattern. This especially concerns design intentions, which can be expressed via annotations.

The third alternative basically is equivalent to the former ones except that the source the annotation definitions are retrieved is external.

3.6. What should be annotated?

This question cannot be answered in general, because anything could be annotated. The result would be a source code that contains myriads of annotations. For a more specific requirement, the answer is different. If an experienced developer intends to apply a design pattern, he/she will hope to get as much tool-support as possible. A tool can only select and apply a pattern if the information available is sufficient. To determine the information necessary to get a pattern applied, the pattern definition must be considered (see [20]). In general, a diagnosing annotation should only describe characteristics that have been there for the program element before it has been annotated. The aspects named in section 3.3 give an idea of what to annotate.

If a *Singleton* should be applied, then a tool could do this automatically, if it is known for which class the *Singleton* should be applied. This can be given by an

annotation that also states that it is required to have one but no more instances of a specific class. In the case of *Singleton,* annotations that could be relevant for other refactoring operations are not important.

The issue becomes more difficult when a practitioner tries to improve the quality of a piece of code. Then, annotations are used proactively. It is suggested to let a tool guide the practitioner through the given code in order to annotate it. Analyzing tools considering performance issues, best practices and other aspects examinable by a program could assist in finding potential places for annotations (also see the next subsection). A more experienced developer knowing a set of annotations is assumed able to find such places himself/herself.

We hope that future IDEs support annotations as suggested in this work. Then, it could be possible to hide annotations from source code, e.g., if the code is currently modified. If the developer were only interested in suggestions, the IDE would only show applicable annotations.

3.7. Automatically determining annotations

For several language constructs, it may be obvious (in the sense of automatically recognizable) which improvements would be applicable. An AST-analyzer could easily recognize *for*-loops or even *while*-loops suitable for applying the *Iterator* pattern. But this trivial case does not necessarily apply for numerous other patterns because they might not base upon fixed ASTs, and they have not the intention of just refactoring a piece of code to a more sophisticated (maybe more self-describing) structure, as *Iterator* does. On the other hand, specifying and evaluating metrics would make it easy to annotate suggestions for refactoring operations such as *Extract Method* ([1]).

A tool could furthermore search for speaking names in order to apply annotations onto source code. By keeping a dictionary of common method names, a tool could reason that a method with name `refresh` is targeting at refreshing the object after a change in the system has happened. This will not always be true but should be in many cases (until there are no ambiguities) if usual naming conventions are followed. As reasoning based on single terms could not be trusted fully, the reasoning tool would always display its results as suggestions to the user who could then decide whether to accept or to decline each offered suggestion. Dependent on the user's action, the tool would record a correct or wrong suggestion. After a certain number of suggestions for one term, it could be determined whether the term should be removed from the repository due to too many incorrect suggestions.

4. Annotations for Refactoring Operations and Design Patterns

In analogy to JSR 250 [5], where common annotations for Java are proposed, this section offers common annotations for refactoring operations, including design patterns. The aim is enabling tools and practitioners to add syntactic and semantic information to a piece of code and evaluate it in order to apply refactoring operations. The next section suggests annotations common to ordinary refactoring operations and more complex design patterns. The sections following the next one concentrate on refactoring operations respectively on design patterns.

4.1. Common annotations

Syntactic annotations are common to refactoring operations and design patterns. They contain information about an AST-analysis. Some examples of such annotations are given in the following. They have been elaborated by examining the output of the tool PMD [3], by looking at refactoring operations listed in [1], the design patterns from [2] and other papers related to refactoring operations. The list of suggestions for syntactic aspects contains a short description of each annotation (some contain alternatives), equalling a brief HID-description:

- Diagnosis: Variable/Method parameter/Method *x* is not used (compare [3]).
- Diagnosis: Class *x* contains only static methods (compare [3]).
- Diagnosis: Next statement creates instance of class *x* (compare [16]).
- Diagnosis: Following method/code block is isomorphic to <*description or unique name of isomorph*> (compare [16]).
- Diagnosis: Method signature contains too many parameters.

Semantic annotations containing suggestions are common among refactoring operations and design patterns to some extent. For the former, it is sufficient in many cases to suggest an operation to be applied onto a single program element (like a method or a class as a whole). A design pattern often requires multiple operations to be executed in order to apply the pattern. For an *Iterator* it would be enough (in case the role *ConcreteIterator* already exists) to annotate a *for*-loop, e.g.:

```
/*
 * @semantic suggestion
 * ... header fields and MID skipped
 * HID: introduce Iterator for variable "list"
 */
java.util.List list = buildList();
for(int i=0;i<list.size();i++) {...}
```

Figure 11: Annotation embodying a semantic suggestion

The list of proposals for common semantic annotations includes:

- Diagnosis: The relation between this class *x* and class *y* is an association with multiplicity [1..*] from *x* to *y* and [1..1] from *y* to *x*.
- Suggestion: The operation <*operation name*> is suggested to be applied.
- Diagnosis: Class *x* cannot be modified (annotation must be added somewhere else than in source code of class *x*).
- Diagnosis: The annotated element represents the entity <*unique entity name*> within the context <*unique context name*>.

4.2. Refactoring Operations

This section proposes some annotations specific to quite simple refactoring operations:

- Diagnosis: Name of variable/class/method *x* is too long (compare [3]).
- Diagnosis: Method is too long (see [1]).
- Suggestion: Move method/field to superclass (see [1]).
- Method *x* from class *y* is not used in this class *z* (see [1]).

Further annotations could easily be identified by examining the descriptions of refactoring operations given in [1]. For each operation a suggestion with the same descriptional text as in [1] could be defined, and for each reason given the same could be done as a diagnosis.

4.3. Design Patterns

This section proposes some semantic annotations specific to more complex design patterns. For each suggestion, the design pattern concerned is given as well as the type of annotation (requirement, diagnosis etc.). The suggestions are oriented on the pattern descriptions in [2].

- *Strategy*, *Template Method*: Requirement: remove conditionals in method *x*.
- *Strategy*, *Template Method*: Requirement: enable new algorithms without sub classing in method *x*.
- *Observer*, *Mediator*: Requirement: decouple collaborators in classes *x* and *y*.
- *Iterator*: Suggestion: facilitate list traversal for list variable *x*.
- *Iterator*: Diagnosis: Elements of list variable *x* are not modified within following block.
- *Façade*: Requirement: Simplify usage of calling methods *x* and *y* in class *z*.
- *Façade*: Requirement: Provide macro function for calling methods *x* and *y* in class *z*.
- *Singleton*: Requirement: Class *x* should exactly have one instance.
- *Adapter*: Requirement: adapt interface of class *x* to interface of class *y*.
- *Flyweight*, *Proxy*: Diagnosis: Instance creation of class *x* consumes too much memory.
- *Proxy*: Requirement: Add caching/logging/access control to class x without modifying it.
- *Decorator*: Requirement: add functionality from class *x* to class *y* without subclassing *x*.
- *Interpreter*: Diagnosis: Interpreted grammar is simple.
- *Interpreter*: Diagnosis: Performance of interpreting grammar is uncritical.
- *Interpreter*: Requirement: Provide a concept for building a grammar interpreter.

From the last example given here for *Interpreter*, it can be seen that an annotation sometimes relies on other annotations. In the case of *Interpreter*, it has to be figured out which class represents a literal, a repeatable expression and so on. It is easy adding an annotation definition and a concrete annotation to a class expressing that the class represents a literal within a certain context (here, the context is the grammar to be interpreted). See the last example in section 4.1.

5. Selecting refactoring operations

With annotations, candidate spots (see [16]) can be identified to be able to select appropriate refactoring operations. This is possible when annotations applied to source code could be correlated with pattern definitions. A candidate spot is an indicator for a

refactoring operation. Preconditions supplement candidate spots when it comes to the decision for or against an operation to be applied. Annotations are also capable of expression fulfilled or violated preconditions. This makes sense as long as either a tool or a developer is capable of deciding about such preconditions, which is possible in general because otherwise it would not be feasible defining the preconditions for refactoring operations, and especially design patterns. A precondition may be a semantic constraint, such as that strict object identity is not required (compare *Flyweight* [2]). A precondition may also be a syntactic constraint. An example for the latter is the existence of a reference from class X to class Y.

In [20], the idea of corresponding elements between source code and pattern definitions is explained in more detail. To obtain a pattern definition, syntactic and well as semantic constraints must be determined. The textual description from [2], which is not machine-processable, can be expressed via annotations in a machine-processable form. The main concept allowing this transfer quite easily is the annotation composed from two parts, the MID and the HID.

6. Conclusion and future work

Working with refactoring operations together with design patterns, includes their appropriate selection and application. For that, a given piece of code must be analyzed sophisticatedly. Current tools do not support the context-aware application or even the selection of refactoring operations. Existing tools analyzing source code are capable of finding information that would help in finding suitable operations to advance the quality of the code. Other tools that would on the other hand be capable of executing the indicated operation do not understand the result. This is often because reporting is only done in an informal way tailored exclusively to a human being. This work presented a way of manifesting syntactic as well as semantic information within source code. With that, tools could exchange information necessary to select and execute code transformation operations. Developers could also add their statements about the code they evaluated.

The paramount concept to realize annotations that are both machine-processable and human-readable is the composed annotation. By employing one part of the annotation for machine-processability and another one, which is coupled with the former one by referencing to it, for the human-readability, general problems with annotation definitions are avoided. Syntactic and semantic aspects have been distinguished, and for each of these aspects, subsections have been introduced to particularize the statements possible with annotations. Besides, they extend the documentation of a system “for free”. A focus has been set on the timeliness of annotations in case of code changes. By adding a hashcode to an annotation, relevant code changes could be figured out.

A prototype tool exists that can parse pattern templates, discover the proposed annotations from source code and build ASTs from the code, including semantic information. Besides that, a set of annotations has been identified for the 23 patterns from [2] with help of the technique *Programming By Example* [12].

Future work will have to concentrate on adapting current tools, such as PMD [3] and AST-analyzers, to support annotations. A standard set of annotations will have to be established to make different tools understand each other and allow the developer to add standardized annotations to the code. It may also be helpful supporting variable

HID-descriptions for different parameter values. This would lead to descriptions that are more fluent. Besides that, a methodology supported by a tool will have to be setup that leads the developer in adding annotations to source code and in generating definitions of refactoring operations, including preconditions and transformations.

The similarity between different MIDs is important for figuring out near misses of applicable operators. Thus, a similarity matrix should be set up for standardized annotations later on. An algorithm such as given in [20] could then determine near misses possibly.

The information available from source code annotations and from annotated design pattern documentations could be used for the detection application of patterns. Certain patterns such as *Façade* [2] cannot be detected sufficiently without explicit information offered by annotations, as *Façade* may be difficult to distinguish from ordinary code.

References

[1] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.
[2] Gamma, E.; Helm, R.; Johnson R.; Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
[3] PMD. http://pmd.sourceforge.net.
[4] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.: A System of Patterns. Pattern-Oriented Software Architecture. John Wiley and Sons. 1996.
[5] Java Specification Request 250: Common Annotations for the JavaTM Platform. http://www.jcp.org/en/jsr/detail?id=250.
[6] Java Specification Request 175: A Metadata Facility for the Java™ Programming Language. http://www.jcp.org/en/jsr/detail?id=175.
[7] Krahn, H.; Rumpe, B.: Towards Enabling Architectural Refactorings through Source Code Annotations. In: Proceedings der Modellierung 2006. 22.-24. März 2006, Innsbruck. GI-Edition - Lecture Notes in Informatics, LNI P-82, ISBN 3-88579-176-5, 2006.
[8] XDoclet. http://xdoclet.sourceforge.net.
[9] Checkstyle. http://checkstyle.sourceforge.net.
[10] Industriallogic: Smells to Refactorings Cheat Sheet. http://www.industriallogic.com/papers/.
[11] Leavens, G. T.; Cheon, Y.: Design by Contract with JML. http://www.jmlspecs.org, 2003.
[12] Lieberman, H.: Your Wish Is My Command: Programming By Example. Morgan Kaufman, 2001.
[13] Nyberg, E.; Mitamura, T; Callan, J.; Carbonell, J; Frederking, R.; Collins-Thompson, K.; Hiyakumoto, L.; Huang, Y.; Huttenhower, C.; Judy, S.; Ko, J.; Kupsc, A.; Lita, L. V.; Pedro, V.; Svoboda, D.; Van Durme, B.: The JAVELIN Question-Answering System at TREC 2003: A Multi-Strategy Approach with Dynamic Planning, Proceedings of TREC 12, Nov. 2003.
[14] Eden, A. H.: LePUS: A Visual Formalism for Object-Oriented Architectures. The 6th World Conference on Integrated Design and Process Technology. Pasadena, CA, June 26-30, 2002.
[15] Taibi, T.; Chek Ling Ngo, D.: Formal Specification of Design Patterns – A Balanced Approach. In: Journal of Object Technology, vol. 2, no. 4, July-August 2003, pp. 127-140.
[16] Sang-Uk, J.: An Approach to Automatically Identifying Design Structure for Applying Design Pattern. Korea, 2003.
[17] Niere, J.; Schäfer, W.; Wadsack, J. P.; Wendehals, L.; Welsh, J.: Towards Pattern-Based Design Recovery. Proceedings of the 22nd International Conference on Software Engineering 2000, Limerick, Ireland, pp. 241-251, ACM Press, June 2000.
[18] Jackpot. http://jackpot.netbeans.org/index.html.
[19] Pena-Mora, F.; Vadhavkar, S.: Augmenting Design Patterns with Design Rationale. Artificial Intelligence for Engineering Design, Analysis, and Manufacturing, Vol. 11, pp. 93-108, 1997.
[20] Meffert, K.: Supporting Design Patterns with Annotations. *ecbs*, pp. 437-445, 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06), 2006.