Supporting Design Patterns with Annotations

Klaus Meffert

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

Abstract

Design patterns are an established means for building evolvable and maintainable object-oriented software. However, using them requires the developer's extensive experience. A wrongly selected design pattern may cause more harm than the right pattern would do good. A single developer is not able to totally know all to date documented patterns, or even identify the best pattern for his current design problem. This paper presents an approach aiding the developer in selecting the right pattern for a given context by introducing annotations (expressing meanings) to object-oriented source code. Eventually the approach is based on trying to match the intentions defined for a particular design pattern with those determined for a given source code fragment. As the existence of source code is a prerequisite the approach is suitable for developers directly working with code and not using a modelling tool that updates their code, or for the reengineering phase.

Keywords: semantic assertions, selecting design patterns, annotations, design pattern intentions, design pattern templates, Java.

1 Introduction

Motivation for aiding the developer in selecting (and, later on, applying) patterns is their complexity and their increasing number. It would be a great help for the developer proposing a list of applicable patterns for a given context (i.e. a given piece of compilable source code) along with the possibility to apply a selected pattern with help of a tool, de facto encapsulating implementation details of the pattern in that way. Condition for the selection of a suitable design pattern for a given context is the understanding of the intent of the given context (in combination with the intentions' understanding of

the pattern to be applied). State-of-the-art artificial intelligence algorithms, including the aspects natural language interpretation as well as extracting the purpose of a given source code, are not capable of sufficiently performing this task. One qualification for understanding a piece of code is the ability to extract its higher-level intentions. But source code itself does not clearly express semantics explicitly as significant information needed to understand a system are not part of the software, normally. This is especially true for a developer's reasons leading to a design decision with several alternatives given during the design phase. Additionally, higher-level semantics, such as "This method refreshes another dependent object when the caller's state changes", cannot be reasoned from source code automatically in general (even not by some developers examining an unknown code). The following figure 1 shows the main concept behind the discussed approach.
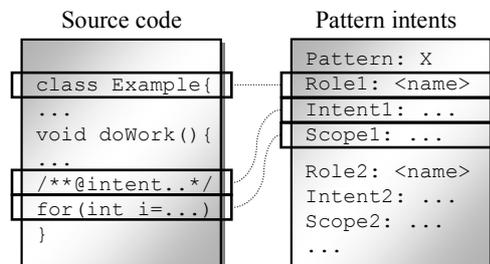

Figure 1: Mapping a pattern's intents

In the figure, the mapping between a source code annotation and corresponding meta-information defined for a pattern is shown, which will be discussed in more detail below. Because of the missing semantics in source code as well as in design pattern definitions, it is proposed adding semantic information to source code via annotations, expressing the intents of the annotated code, as well as defining intents for design patterns in a corresponding way. Source code intents could then be compared with intents defined for a design pattern's parts until a suitable pattern is found. A

suitable pattern would be such for which the intents of its parts are matched by the intents of annotated code in a satisfying way (e.g. at least annotation matches with scope mismatch; configurable by the developer). Adding semantic information to source code could either be done manually for virtually any information, or, to some extent, with help of tools analyzing source code or code flow (such as PMD [7], JLint [9] or JCSC [10], to give a first idea).

The whole development process as it is considered here requires the following five activities in the given order:

1. Defining/Reusing semantic constraints for identifying each design pattern's applicability.
2. Creating source code thru programming.
3. Bringing in annotations to existing source code manually or automatically.
4. Determining design patterns for annotated source code by a tool to be introduced.
5. Letting the developer select a suggested pattern, afterwards applying it manually or tool-supported.

The definition of available semantic constraints (activity 1) must be done for each semantic constraint in a way that connects such constraints to annotations and valid scopes. For each pattern these definitions could then be reused. Activity 1 lies not in the responsibility of the average developer. Such task should have been in the competence of a chief developer or a service provider. Activity 2 is supported by nowadays development tools such as integrated development environments (IDE's). It is independent from activity 1. Annotations from activity 3 would be reflected by semantic constraints from activity 1.

Activities 3 and 4 are described in detail below, besides activity 1. Tools subject to future work would be concerned with the application of design patterns as an integral part of activity 5.

At the present, a prototype program has been developed qualified for processing annotated Java source code along with the convenient definition of the intentions of *Composite*, *Iterator* and *Singleton* [6]. The prototype eventually can detect design patterns applicable for such annotated code by processing pattern templates. This involves the definition of design pattern constraints by means of annotations plus a valid scope per annotation, parsing annotations in source code along with their context (corresponding to a scope), and providing an algorithm for matching pattern constraints with annotations in source code.

2 Annotations

In the context of this paper, an annotation is a well-defined (Javadoc-style) comment in the first place (compare [11] or [5], in contrast to JSR 175 [2]). Secondly, an annotation could virtually contain any piece of information. 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 [5] is one popular tool incorporating this. In the latest Java version, Java 5, annotations are, in contrary to this paper, introduced by JSR 175 as valid language constructs living outside comments as first-class statements. Deciding to express annotations as well-defined comments is more than syntactic sugar. Because with comments it is no matter preserving the behaviour of the annotated code, thus keeping it possible using current tools, compilers and interpreters. Besides, JSR 175 annotations don't work at statement level but only for declarations. It would be a showstopper not allowing to annotate at statement level for the presented approach, because a single statement could potentially carry significant semantics. Moreover, not more than one (atomic) Java 5 annotation can be applied to one declaration.

For the upmarket selection of suitable design patterns, semantic information have to be identified, which in turn is done by using the described annotations. Because a design pattern could potentially reflect several intentions at once, it should be possible defining multiple annotations at once. These annotations could potentially be dependent on or refer to each other, which for instance applies for *Composite* (see section 2.4). Annotations could be used to express different types of a developer's intents, such as purposes, wishes, problems, drawbacks or needs. Annotations added to source code as expressions of a developer's intentions extend the documentation of a system for free, such as unit tests (and to some extent design patterns) do.

2.1 Syntax of annotations

It is most important having a well-defined syntax for annotations making it machine-processable and unambiguous. The syntax should be easy to understand and to use by a developer. Recent works such as JSR 175 [2] and [11] yielded such syntax. In this paper the syntax of an annotation is limited to contain a noun in first place, followed by a colon, then followed by a list of single words separated by blanks. The list of single words allows forming

natural language-like expressions with the restriction that those words should only be significant, pointed and not be fillers (except known ones). Following this paradigm, quite short "sentences" with up to about six words should result. To distinguish an annotation from an ordinary comment, each annotation begins with the at-symbol @. Some examples of valid annotations (comment omitted):

- `@intent:   flexible   object creation`
- `@problem:  inflexible  object creation`
- `@drawback: creation too slow`
- `@need:     speedup   on   object creation`

The first two examples indicate a factory pattern; the last two would possibly lead to *Flyweight* [6]. The above proposed syntax could be seen as equivalent to the form presented in JSR 175 and [11] to the degree that it allows expressing virtually any information. Besides the stronger formalization of JSR 175, the difference might be the better human-readability of the syntax proposed above. To allow programmatic evaluation of word-terms following the described syntax and paradigm, the dictionary should be quite reduced and the possible combinations between words should be well defined (e.g. after attribute "flexible", only one of five nouns or compound nouns, such as "object_creation" or "object creation", should be allowed).

2.2 Valid scope of an annotation

The scope of an annotation is the source code element the annotation is placed above within a piece of code. An annotation could virtually be placed ahead of any program construct, 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. A source code element is a declaration or a statement. To group such elements, types of them are introduced as follows. A type of source code element is a characteristic implementation distinguishable from other implementations. For instance, in Java a method which has visibility other than *private*, no return type (namely the pseudo type *void*) and no parameters could be regarded as a characteristic

type of a source code element. Therefore the methods

```
public void doSth() throws … {…}
```

and

```
protected void doWork() {…}
```

represent the same type of method under the above determination. The definition of such source code element types is arbitrary and must to be chosen carefully, ideally by utilizing the experience of a practiced developer. Currently a huge number of groups of operators, statements, and declarations has been distinguished by the author, providing many permutations. Arbitrarily grouping definable program constructs could be supported with a technique called *Programming by Example* (see [12]).

2.3 Annotations in practice

The piece of code (e.g. some few classes) to be considered for annotating is determined by the concern a developer has. For instance, if the creation of an object is too slow (and the developer wants to change that, e.g. by lazy instantiation), the piece of code would be the construction statement. Only such code needed be considered that lies in the context of the problem. Identifying the context could be helped by with a contextual help system. Nevertheless it would be possible annotating as a matter of prevention, for which a tool guiding the developer could be provided.

It follows an example for annotated source code. Having a loop over a data structure, the annotated source code could look like this (annotation printed boldly):

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

Figure 2: Annotated *for*-loop

The wish expressed by the above annotation could obviously be satisfied by an *Iterator*. Instead of a wish, the expression of an intention, a problem or a drawback could lead to the same conclusion! It is important to notice that the given annotation should only lead to an *Iterator* when certain constraints (preconditions) apply:

- The annotation's scope is a *for*-loop (or a comparable construct).
- `list` has to be a sequentially accessible list type.
- The index variable `i` must iterate over the whole list.
- No modification of `i` inside the *for*-loop is allowed.
- No modification of a list element inside the loop is allowed (as *Iterator* can usually not handle this, except for the removal of the current element of the iterated list).

If at least one of the above (generally significant) conditions does not apply, then *Iterator* would not be applicable. These technical conditions could be checked programmatically by inspecting an abstract syntax tree (AST). It could be considered displaying the violated and passed constraints to the developer. Then, the developer could decide whether to refactor his 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 source 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.

Flyweight as a more complex pattern unites many (also semantic) constraints making it outstandingly difficult to use for the developer. One example for a semantic constraint of *Flyweight* is the precondition "object to be converted to *Flyweight* needs to potentially have many memory-consuming instances". *Flyweight*'s semantic constraints could be represented by annotations, which – as comments – could easily be inspected, complementing technical constraints. Then, the annotation of a source code as well as the definition of constraints for the *Flyweight* design pattern make it possible to conclude the applicability of the pattern for an annotated code fragment.

2.4 Design pattern intentions

Design pattern intentions are defined in a separate file, not in or as source code. Semantic constraints (intentions) of design patterns must be represented in a way allowing it to match them with above given annotations of source code (also compare figure 1). The first step that prepares the ground is using the same syntax for both design pattern intentions and source code annotations (only skipping the comment elements for the intentions).

Furthermore, the intentions of a design pattern could be described by providing a set of tuples. Each tuple contains an intention (corresponding to one or many annotations) as well as a valid scope. A valid scope is as described before (e.g. for *Iterator* it would be a certain type of *for*-loop). To recognize a valid scope one could, technically, provide a class name denoting a handler. Such a handler class would be able identifying valid scopes of an annotation by examining a given AST for conformance to an expected structure. All handler classes must have the same interface in common. A handler is suggested here over declaring scope in a non-programmatic fashion because a scope could be formed indiscriminately. The concept P*rogramming by Example* (see [12]) supports the definition for such handlers by supplying samples of valid scopes.

It might be possible that a certain scope for a particular annotation would influence the possible set of scopes (i.e. eventually the responsible handler) allowed for another annotation. One idea resolving this is providing multiple mutual exclusive sets of tuples for a single pattern, with only one set being active at a time.

Defining *Iterator*'s intents plus scopes
For the *Iterator* pattern the definition expressing the pattern's intent and scope, could be containing the following information:

```
Role: Iterator
Intention: abstract list traversal
Scope_Handler: FOR_STATEMENT_TYPE1
```

Figure 3: Defining *Iterator*'s intent + scope

The role is described below for the multi-role pattern *Composite*, for *Iterator* with only one role its name is only relevant when applying a pattern to connect the *Iterator* role with the *Client* role. The potentially complex scope is identified by the scope handler class `FOR_STATEMENT_TYPE1`. In the case of *Iterator*, we would expect a *for*-statement, beginning at zero, incrementing by one and looping over the whole length of a structure recognized as a list etc. (as described above). For Java, the classes embodying a list could be declared in a repository as most container implementations rely on a very small set of classes (such as `java.util.List`). A sophisticated algorithm can furthermore recognize offsprings of those classes representing a list most probably, either.

It should be pointed out that the intention in figure 3 should be seen as a master, or normalized intention, because potentially many annotations could semantically correspond to it.

Defining *Composite*'s intents plus scopes

Composite as another example is more complex than *Iterator*, alone by the number of roles it contains. The role information is necessary to distinguish different parts of a context to be mapped with a pattern. The role names are arbitrary and come into play not until the application of a pattern. The role's intentions could be described as follows:

```
Role: Component
Intention1: element_of_list
Scope_Handler1: listField
Intention2: perform_operation
Scope_Handler2: METHOD_NONPRIVATE

Role: Composite
Intention1: element_of_list
Scope_Handler1: LISTFIELD
Intention2: perform_operation
Scope_Handler2: METHOD_NONPRIVATE
Intention3:  add_child
Scope_Handler3: LIST_ADDER

Role: Leaf
Intention: perform_operation
Scope_Handler: METHOD_NONPRIVATE
```

Figure 4: Defining *Composite*'s intents + scopes

As can be imagined, complex conditions could be thought of that are built of combinations of intentions connected by Boolean operators. Representing such operators is not a problem to be solved by such defined intentions but by appropriate terms representing the connections between them and by glue logic enabling this. In the example from figure 4 there is a scope handler, LIST_ADDER. This handler is used to check whether a given method contains logic to add an instance to the list specified by the intention element_of_list of role *Composite*. To allow the handler to know about the list annotated that way, the following procedure could be undertaken: Provide access to all data available from the pattern definition and from previous actions undertaken for the pattern (like resulting from execution of other handlers). With that, the LIST_ADDER could easily ask the LISTFIELD handler to return the matching object representing an element list. That way, dependencies between intentions, and therefore annotations, could be handled in general.

Regarding the intentions from figure 4, the *Leaf* role is partly contained in the roles *Composite* and *Component*. To distinguish between *Leaf* and the other two, one would need to check whether another

role also applied and whether the latter one contained additional annotations. If this check was positive, *Leaf* wouldn't apply, but instead the more comprehensive role. For the *Composite* pattern there is no obvious need defining negative constraints, meaning: If a certain negative condition holds true within an otherwise potential role class of a source code, then that class is disqualified (compare [19]). For *Leaf*, such a negative condition would be the appearance of intention add_child. But because *Leaf*'s definition is fully contained within the *Composite* role definition, the definition of negative conditions isn't necessary for *Leaf*.

Details such as the multiplicity allowed for a role to have within a pattern are not given in figure 4. Representing them within a pattern definition is not related to annotations, but just to introduce ordinary parameters within the definition. This holds true as well for marking roles as optional within a pattern; in the above example this would be the case for the *Leaf* role within the *Composite* pattern.

3 Tool-supported selection of patterns

The process of selecting suitable patterns for a given target source code includes evaluating annotations of both the given source code and all design pattern intentions defined. If a positive match over all pattern annotations exists, the pattern whose annotations were matched could be seen as highly apt for being presented to the developer as recommendation.

The problem of matching a design patterns intent with a source code annotation could be reduced to the problem of deducting that a near natural language expression corresponds (and not necessary equals) to another one. For instance, the problem formulated by "need mechanism for speedup" corresponds to the solution "provide mechanism for speedup". Comparing the intention from figure 3 to the annotation shown in figure 2, the correspondence seems clear: Both only differ in the first word before the double colon. Not in any case might the situation be that trivial. In the easiest case this matching could be accomplished by defining all source code annotations seen as corresponding to a given expression describing a pattern's intent first one. To complement this repository-driven approach, a thesaurus-like service could be used, for single words as well as for phrases within a sentence. The chatbot A.L.I.C.E. [4], suitable for identifying complex synonymic phrases, could support such service.

In order to decide which pattern could be applicable for a certain context (i.e. a source code fragment), the intents defined for the pattern and the annotations brought in to the context has to be matched. Not only need contents (in the sense of concrete comment texts here) of annotations be regarded during the matching process. Additionally, the scope of the annotations compared should match. As for each pattern the scope for all intents is given, the necessary data is present and could be used by an algorithm executing the matching process. In short, the algorithm could consist of the following steps, given an annotated source code and a repository of design pattern intentions:

```
For each pattern definition in the
repository:
  For each intention defined:
     Try to match with annotation
     from source code, considering
     scope given for intention:
        Positive     match:     mark
        intention for pattern and in
        code as matched.
        Negative     match:     mark
        mismatch.
  If all pattern intentions marked
  as match:
     Do the same check for negative
     intentions (if any):
        If at least one match: exit.
        Else: mark pattern as being
        apt for appliance.
```

Instead of exiting in case of a negative match of an intention or a positive match of a negative intention in the above algorithm and therefore excluding a pattern's applicability categorically, one could proceed with the matching process and reduce the quality of the applicability determined (which would be displayed to the user). Near-misses could be recognized by allowing either the annotation or the scope to mismatch or by allowing some annotations to miss completely while others don't.

4 Automated application of annotations

For several language constructs it may be obvious (in the sense of automatically recognizable) which improvements would be applicable. Figure 2 displays such a case as 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 AST's, 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.

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 most cases (until there are no ambiguities), since otherwise the developer should think about a new way naming his methods. For class names the same applies as classes potentially fulfill a role within a pattern. As reasoning based on single terms could not be trusted fully, the reasoning tool would always display its results as suggestions to the user. He/She 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.

A tool selecting patterns for the application onto target code should not see any difference between the situation raised by manual annotation and the situation where a mechanism as described above in this section reasoned positively about facts. This could be realized by forcing a reasoning tool to not evaluate proprietary source code (except for scope determination) but let it look for annotations. Adding this layer of indirection to the process, it would be possible extending the capabilities of the pattern matching system without changing or refactoring the technological basis.

5 Applying selected patterns

The process of defining the intentions of a particular pattern is independent from defining pattern templates (see [8]) that also contain information necessary for applying pattern fragments to a given context, although the other way round there is a dependency. A pattern fragment is a part of a pattern that is to be applied to a specific context, independently of other fragments of the pattern. Because those fragments are bound to a certain context, the concept of annotations described above could be used as strong indicator for applying those fragments.

Applying a selected pattern by a tool requires a pattern template definition to contain

implementation details, such as described in [8]. Although mixing in annotations to pattern templates is possible without restrictions, this subject is omitted in this paper because of its complexity. To allow for the application of a design pattern, the contexts to which the pattern fragments are to be woven in, have to be known. This is the case for the approach described: Annotations, complemented by valid scopes, provide the means for finding the suitable places in a given source code to apply pattern fragments on.

6 Related work

6.1 Annotations

JML [11] and XDoclet [5] 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 technical 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.

6.2 Defining and applying design patterns

LePUS [14] permits defining 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. Other approaches targeting on the description of patterns are [15] and [1]. Any of them does only support a subset of design patterns, or is restricted to frameworks (as [1]). [17] offers a meta-model suitable for defining patterns, which allows to apply them later on.

6.3 Selecting design patterns

[16] tries to select applicable design patterns by comparing two legacy code versions seen as related. [20] 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.

6.4 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 must be matched with facts, in lieu of with other statements. At last question-answer systems have to find out themselves the semantic meaning of queries and statements.

6.5 Reverse engineering of design patterns

As semantics is not an explicit part of source code, current approaches must cope with either analyzing technical characteristics of code (and maybe intrinsic semantics) or looking at the design model of a program. [19] searches patterns in legacy code by considering obligatory elements as well as forbidden and unimportant ones. Other approaches consider metrics or use UML diagrams as input.

7 Conclusion and future work

The presented approach lays the foundation for selecting suitable patterns for annotated source code. For it, the paper introduced a simple grammar for annotations and showed how intents for patterns could be described. It furthermore showed how matching source code annotations with design pattern intentions could be done. As an annotation could virtually contain any information, there is no limitation in using annotations to express problems, intents, wishes or drawbacks, considering that a matching between two differently expressed meanings is possible. It follows that the approach is appropriate for any design pattern that could be described non-ambiguously. The applicability of the approach is dependent on the existence of source code. This implies either starting with programming or doing the coding after an initial modelling phase. Also, the approach could be appropriate for the reengineering phase. Nevertheless it is important to support annotations by development tools. It is essential asserting the correctness (syntax, spelling, scope) of manually entered annotations as well as allowing the developer to let the system suggest annotations with help of a contextual dialog or code completion for annotations. For the reverse engineering of patterns, annotations backed up by

definitions of design pattern intentions could complement classical pattern detection approaches.

A drawback with annotations may be the need adapting the development process. Applying annotations costs more time. Before using them for selecting patterns, the only advantage would be a better documented code. An open question is how to best guide the developer letting him provide annotations within source code necessary to determine suitable patterns. This could be done by a tool examining characteristic places in the code, like object creation, method invocation or usage of speaking names. The definition of pattern templates is a difficult and iterative process, anyway, no matter which approach is chosen. It has to be examined which standardized modules to provide for allowing to define a template by selecting existent parts, at best.

An important task currently in progress is the definition of a common set of annotations along with a dictionary of words allowing to constructing them as well as synonymic annotations. In this process, typically, for a defined pattern intention there will be defined a few matching source code annotations. Although the general principle of defining a common set of annotations is equal to the JSR 250 [3], the subject is different, because JSR 250 copes with general-purpose annotations not related to design patterns at all. As before mentioned the tool-supported application of annotations to source code is an important task to go for. The automated (or even the tool-supported) application of design patterns is tightly coupled with the tool-supported selection of them. The information available from the latter process (definition of intentions of pattern parts, identifying of contexts for pattern parts etc.) would be highly useful for applying a selected pattern. A problem with the automated application of a pattern is the user-friendly definition of the context-dependent parts (compare ANGIE [18]) as well as the definition of the refactoring operations necessary to apply such parts. However, a number of cases could be distinguished. They could be oriented on the possible characteristics of AST's (reflecting types of source code elements, as discussed above).

Definitions of pattern intents could possibly come into play for e-learning systems to allow for presenting a pattern's meanings in concise and at the same time precise form to the learner.

8 References

[1] Pree, Wolfgang: Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.

[2] Java Specification Request 175: A Metadata Facility for the Java^{TM} Programming Language. http://www.jcp.org/en/jsr/detail?id=175.

[3] Java Specification Request 250: Common Annotations for the Java^{TM} Platform. http://www.jcp.org/en/jsr/detail?id=250.

[4] A.L.I.C.E. Chatbot. http://www.alicebot.org/.

[5] XDoclet. http://xdoclet.sourceforge.net/.

[6] Gamma, E.; Helm, R.; Johnson R.; Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[7] PMD. http://pmd.sourceforge.net.

[8] MacDonald, S.; Bromling, S.; Szafron, D.; Schaeffer, J.; Anvik, J.; Tan, K.: From Patterns to Frameworks to Parallel Programs. Parallel Computing, v.28 n.12, pp.1663-1683, Dec. 2002.

[9] JLint. http://jlint.sourceforge.net/.

[10] JCSC. http://jcsc.sourceforge.net/.

[11] Leavens, Gary T.; Cheon, Yoonsik: Design by Contract with JML. http://www.jmlspecs.org, 2003.

[12] Lieberman, Henry: 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, Toufik; Chek Ling Ngo, David: 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, Jeon: An Approach to Automatically Identifying Design Structure for Applying Design Pattern. Korea, 2003.

[17] Albin-Amiot, Hervé; Guéhéneuc, Yann-Gaël: Meta-modeling Design Patterns: Application to Pattern Detection and Code Synthesis. Workshop on Automating Object-Oriented Software Development Methods. ECOOP, 2001.

[18] ANGIE Generation Now! http://www.d-s-t-g.com/neu/pages/pagesger/et/common/techn_angie_frmset.htm.

[19] Streitferdt, Detlef; Heller, Christian; Philippow, Ilka: Searching Design Patterns in Source Code. Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC); Edinburgh; July 2005, pp. 33-34.

[20] 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.