TOWARDS A NEW CODE-BASED SOFTWARE DEVELOPMENT CONCEPT ENABLING CODE PATTERNS

Klaus Meffert, Ilka Philippow

TU Ilmenau, PF 10 05 65, 98684 Ilmenau, Germany

Meffert@rz.tu-ilmenau.de, Ilka.Philippow@tu-ilmenau.de

Keywords: software development, design patterns, architectural patterns, program understanding, annotations

Abstract: Modern software development is driven by many critical forces. Among them are fast deployment requirements and easy-to-maintain code. These forces are contradicted by the rising complexity of the technological landscape among others. We introduce a concept aiding in lowering these negative aspects for code-based software development. Protagonists of our work are explicit semantics in source code and newly introduced code pattern templates, which enable code transformations. Throughout this paper, the term code pattern includes architectural patterns, design patterns, and refactoring operations. Enabling automated transformations stands for providing means of executing possibly premature transformations.

1 INTRODUCTION

Observing current software development projects leads to the conclusion that for a huge number of these projects working with source code is the main driver. By accompanying a lot of source code-based projects, we noticed the difficulties with state-of-the-art programming techniques. This paper is a contribution to make software development more productive in that segment. To accomplish a raise in software development productivity, we suggest using what is described as code pattern template throughout this paper. Such a template is suited for supporting the usage of architectural patterns, design patterns, and refactoring operations. To enable such templates, we introduce explicit semantics to source code, assigning a deeper meaning, or sense, to a piece of code. As low-level refactoring operations are supported by modern IDEs to a reasonable extent the main focus of this paper lies on design and architectural patterns.

2 DEFINITIONS

In this section definitions are introduced that are helpful for understanding the presented approach.

2.1 Semantics

The meaning of a statement or operator of a programming language ("program statement") is its dedicated function. For example, the Java statement $x{+}{+}$ increases the value of the variable x. The semantics of a statement is its deeper meaning within a context (sense, intention). The context determines the meaning of x and thus the meaning of the statement itself. If x represents a number of pieces, then $x{+}{+}$ increases the number of pieces by one. Is it the number of available or defect pieces? This in turn depends on the wider context.

2.2 Annotation

To express the semantic meaning of program constructs (statements, declarations, blocks) annotations are introduced in this paper. An annotation as we see it is a construct that can be put above any valid program construct (in contrast to Java's JSR 175) without changing the behavior of the program. An annotation can also have parameters, which are determined by the annotation's definition. To let an annotation express contextual information, a set of predefined senses has to be made available.

2.3 Transformation

Here, a transformation defines the process of getting from a given source to a target code by applying defined rules. A rule always produces the same result when applied onto the same source code. At first, transformations are introduced to obtain a code pattern template (see next section). At second, they are helpful to support the developers in applying a code pattern with tool-support. The latter mentioned can only be a try because when automatically transforming code the machine executing the transformations misses what a human would call awareness of the meaning of source code and its transformation. That in turn makes it impossible, in our opinion, to get a machine to transform arbitrary pieces of code correctly. That is why we suggest a workaround for this problem. Our suggestion leads to an extended support for code transformations.

2.4 Code pattern template

A code pattern template contains any information about a pattern necessary for a process like selection, application, or recognition. This includes static and dynamic parts, source code as well as semantics. To use a template effectively it must be connected with a given source code, the context. This connection is proposed to be provided by annotations.

The semantic counterpart in the code template is equivalent to annotations. Also the scope an applied annotation has is relevant. When comparing the name and the scope of an annotation in source code and code pattern template a match can be determined. The next figure illustrates this.
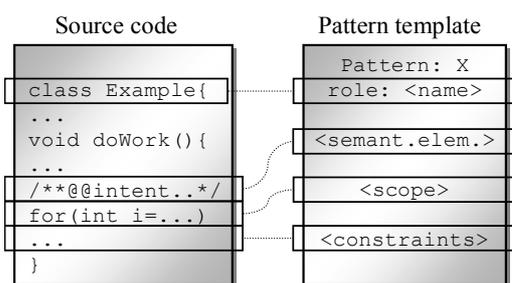

Figure 1: Matching source code with a template via annotations

The source code on the left contains annotations that have a correspondence in the template on the right.

3   TOWARDS A NEW CONCEPT

Our approach suggests extending common code-based development scenarios by a semantic layer. For adding semantics, two perspectives are proposed. The first one is the perspective of code. By adding explicit semantic information to code, analysis tools have much more possibilities of extracting information from the code. The second perspective concerns code pattern templates which are supposed to be enriched by semantic information, too. The vehicle for manifesting explicit semantics in code is the concept of annotations. Adding semantics to code pattern templates can occur in any satisfying way, because templates do not have to be compilable.

The result of having explicit semantics in source code as well as in code pattern templates allows finding an appropriate template for a given source code in order to transform it, according to what the template defines. Matching code and template is easier with help of explicit semantic information than by any ordinary AST-based comparison (see Meffert+, 2006). When working with code patterns, tool-support for the pattern processes definition, recognition, selection, and, application is possible regarding current development environments or papers like (Taibi+, 2003). Each process can profit from explicit semantics that would otherwise not be reasonable by a machine. The next paragraph describes the idea of introducing explicit semantics to code and code template in detail.

3.1 Working with explicit semantics

As said, there are two sides of the medal, for which explicit semantic information is relevant, namely source code and code templates. Code templates are a newly introduced entity (other papers did this in a different way already) not significant for a programmer. However, enriching source code with annotations is significant for developers following our approach. As it is not possible reasoning about the sense of any piece of code and as it is not possible knowing about the design intentions of developers, annotations have to be added to source code manually at least to a certain extent. It may be possible for a machine to introduced annotations for cases similar to known ones.

The introduction of semantic information into code templates is different and more complex than for source code. Code templates have to be described once for a set of similar contexts, whereas

annotating code depends on the individual motivations for doing so. Different motivations for annotation source code are plausible, e.g. given the developer

➢ knows a pattern to apply but wants support in applying it,
➢ does not now which pattern to select and wants support in selecting and applying one, or,
➢ wants to know which patterns may exist in the code respectively to which extent they are existent.

Depending on the aim a developer has, the procedure for applying annotations is different. The easiest case is when a pattern is selected already and has to be applied. In this case adding annotations to source code is quite simple because it is known which annotations are required. Finding a pattern to apply is more difficult as the source code to be examined has to be annotated appropriately. In general, an annotation has to be added to the code where an analysis tool is not able to evaluate the code appropriately to present a qualified suggestion.

4 CODE PATTERN TEMPLATES

The process of obtaining a code template definition, including annotations, contains the activities:
1. Choose a suitable source code as base for applying a selected pattern to.
2. Provide a target code where the selected design pattern is already applied.
3. Transform code from step one to step two and record the actions undertaken. During this process, add annotations for any piece of code that is significant for the pattern.
4. Obtain a code pattern template, including annotation definitions and transformations, from the previous step.
5. Verify and improve the generality of the obtained code pattern template by choosing a different source code for step two and proceed from there.

For the first activity a suitable code can be either produced by providing a degenerated version of a design pattern, or a code on which the pattern could be applied. The target code from activity two contains the applied pattern. In step three, annotations are added to it. For instance, the *Client* class of the *Composite* design pattern could be marked with an annotation such as:

```
/*@@COMPOSITE_CLIENT_CLASS*/
public class Client {...}
```

The transformations undertaken manually during activity three must be backed-up by transformation methods added on demand later on, in case they do not exist already. These methods include routines such as "condense a list of parameters into one dedicated parameter", or "replace a *for-loop* with *Iterator*". At first it should be tried to compose each needed method from already existent lower-level ones to enforce reuse of transformations. In the next paragraph, the procedure for defining code templates is described deeper.

In activity four, the transformations obtained as well as the annotated relation between source and target code is encapsulated under one package.

To validate the overall reusability of the package, a different source code should be chosen for which the examined pattern also is applicable and for which the previously obtained pattern template should work. That source code should contain common parts – especially same annotations – with the initially chosen source code.

In general it should hold that for any annotation added to the source code, a corresponding annotation in the target code must exist. Also, for any difference between source and target code a transformation must be defined.

4.1 Getting a code template definition

The procedure to get to the template involves the following steps:
1. Consider the target code. For each class:
   a. Create a new role section. The name of the section reflects the role of the class within the pattern. If the class plays no assigned role in the pattern, choose a unique role name.
   b. Copy the whole target code for the class into the role section.
   c. Determine the context-dependent parts of the copied code. Here, reflecting on the found transformations benefits.
   d. Introduce a placeholder (called slot) for each context-dependent part. For context-dependent sequences of statements introduce an annotation above any sequence that has a distinct meaning.
   e. For logic that should be kept as the original from the source code is, add a control tag with parameters.

f. For logic within a block (e.g. within methods) that may be extendable, add a documentation link. A documentation link references a description with possibilities to extend the logic.

2. Add each interface of the target code as is to the template by putting each code block in a section identified by the keyword *interface* plus the name of the role the interface plays.

3. For any annotation added to the source code (and thus also to the target code) a definition must be created or updated. There are two cases:
   a. Appropriate Annotation definition not existent: Create it.
   b. Otherwise: Update annotation definition if changes are necessary and possible.

4. Implement handler routines for
   a. extracting information from relevant annotations and providing relating these information to other annotations,
   b. actions for which handlers do not exist, and,
   c. executing precondition checks.

Step 3 of the above activities results in annotation definitions. The annotations existent in the source and target code are equivalent with respect to their name, but maybe different concerning their scope. The scope of an applied annotation is the statement the annotation is applied above.

5 RELATED WORK

Relevant approaches cope with the issues code transformation, code templates, annotations, and program understanding. With Java 5 annotations have been introduced as first-class language constructs (JSR 175). These annotations are restricted in that they cannot be applied to arbitrary scopes (i.e. single statements are out of the scope). (Jackpot) browses Java source code for conformance with generic rules and executes transformations on ASTs automatically. Each rule contains a Java-statement to be matched and specifies how to transform the fragment. (Krahn+, 2006) introduce compilable code templates to execute automated refactorings for Java code via code generation. To include directives into the template, comments are used. The approach does not consider semantic information. FUJABA (Niere+, 2006) aims at extending UML for specifying method bodies and generating code from UML diagrams up to the level of statements. Additionally, the generation of UML diagrams from source code is supported.

6 CONCLUSIONS AND FUTURE WORK

This paper suggested introducing explicit semantics to source code as a means for identifying the contextual sense of program elements. Using annotations as a vehicle for transporting semantic information, an adaptation of current coding practices is proposed. In turn, the definition of code pattern templates allows obtaining key information about a pattern, including its structure, preconditions, transformations and annotation definitions.

A code pattern template contains a lot of useful information and allows recognizing synonymous pieces of codes. When defining code templates, variants of a pattern and variations of source code have to be considered separately to a certain extent. Reuse is possible but reflecting on the validity of existing templates is important. There should be a consolidation feature which puts common parts of two similar templates into one master template and adds the different parts by referencing them.

Currently, we are examining complex patterns in order to demonstrate the practicability of the described approach. Especially the capability to reason about missing information/annotations is focused.

7 REFERENCES

Gamma, E., Helm, R., Johnson R., Vlissides, J. (1995): Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

JSR 175: A Metadata Facility for the Java Programming Language. http://www.jcp.org/en/jsr/detail?id=175

Jackpot. http://jackpot.netbeans.org/index.html

Niere, J., Schäfer, W., Wadsack, J. P., Wendehals, L., Welsh, J. (2000): Towards Pattern-Based Design Recovery. Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, 241-251

Krahn, H., Rumpe, B. (2006): Techniques For Lightweight Generator Refactoring. In: Lämmel, R., Saraiva, J., Visser, J.: Proceedings of Summer School on Generative and Transformational Techniques in Software Engineering (LNCS 4143), Springer.

Meffert, K., Philippow, I. (2006): Supporting Program Comprehension for Refactoring Operations with Annotations. In: Fujita, H., Mejri, M. (eds.): New Trends in Software Methodologies, Tools and Techniques - Proceedings of the fifth SoMeT_06, Vol. 147, 48-67.

Taibi, T.; Chek Ling Ngo, D. (2003): Formal Specification of Design Patterns – A Balanced Approach. In: Journal of Object Technology, vol. 2, no. 4, July-August 2003, S. 127-140.