

Towards a new code-based software development concept enabling code patterns

Klaus Meffert¹, Ilka Philippow¹

¹ TU Ilmenau
PF 10 05 65
98693 Ilmenau, Germany
Meffert@rz.tu-ilmenau.de
Ilka.Philippow@tu-ilmenau.de

Abstract. Modern software development is driven by many critical forces. Among them are fast deployment requirements, a code base adaptable to new technologies, easy-to-maintain code, and low development costs. These forces are contradicted by the rising complexity of the technological landscape, harder to understand code due to its complexity, missing documentations and quickly hacked in program logic. 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 and whose definition and application will be explained in this paper. 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 that might lead to results requiring manual finishing.

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. Model-driven development and other techniques seen as sophisticated may be established to a certain part but code-related work is the most relevant procedure for many projects, including work with legacy artifacts. By observing and 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. Our technique is especially suited for architectural elements to be interwoven with existing source code. 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 and hopefully easing the usage of (code-related) architectural patterns (compare Buschmann [12]), design patterns (compare Gamma [1]), and refactoring operations (see Fowler [11]). To enable such templates, we introduce explicit semantics to source code, assigning a deeper meaning, or sense, to a piece of code.

Before digging into detail, we make the following assumptions to build our paper on:

- A lot of software projects exist for which source code is the main entity developers work with. We also know that there are other established techniques as well, such as model-driven development, and that the perception of the modus operandi depends on the type of project (embedded systems, commercial vs. academic etc.). However, we assume that the majority of software projects is code-driven.
- We think that it is at least difficult (if not impossible) replacing code-based development with higher-level concepts, such as visual programming, to a satisfying extent in general-domain programming.
- Code patterns are an important means of building maintainable software. Only using them correctly leads to a benefit, applying them differently makes a loss in product quality very likely.
- The number of documented design and other code patterns is way too large to be known by a single person or a small group of developers.
- In our opinion, many design and architectural patterns exist of which each single is too complex to be understood well enough by the majority of software developers. Here, better tool support is needed.
- It seems impossible identifying the sense (i.e. the semantic meaning or intention) of an arbitrary piece of code within a given context by a machine.
- In our eyes, it is not possible to perfectly execute automated transformations for unknown pieces of code. The reason is the missing awareness of the transforming entity about source code, target code and transformations.

The next section describes our perception of current code-based software development practices as a common scenario in software development projects.

1.1 A common software development scenario

A typical scenario for software projects may look like this: After organizational and other management activities are finished, a person or a group sets up the requirements for the development department. Often, the requirements are compiled in an ordinary document, maybe supported by charts, cross links etc. Nevertheless, this important document frequently is informal and not machine understandable in a satisfying way. The developers read the requirements, communicate their understanding of them and questions arising to the creator of the document or to a different, but competent person in this field. After the developers think that they know what the program to be created should do, they begin by extending the existing code base. For that they use an integrated development environment (IDE) available with their project and write plain source code. Often this activity includes searching the right program modules or packages to modify or extend. To finally implement a given requirement, the developers have to dig down into the code and do the adaptations they think fulfill the requirement. Normally, the software development department tries to develop maintainable code that is easy to understand, efficiently adaptable to future requirements and with

as few errors as possible. Many developers rely on code patterns to get maintainable code. This is rational as it is a complex and difficult task to find a suitable micro or macro architecture for a given design problem. However, complex code patterns, typically including architectural and a lot of design patterns, can be used in a critical way. Either the wrong pattern could be chosen for a problem, or a correct pattern could be applied in a wrong way. Additionally, the documentation of the code the pattern is applied to could be incomplete or wrong. While detecting wrong documentation is not in the scope of this paper, the other aspects are. 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. In the following section definitions are introduced that are helpful for understanding the approach presented in the next but one paragraph.

2 Definitions

Throughout the paper, the term annotation is important. Tightly coupled with it is the term semantics that will be defined first. Afterwards, also the important terms transformation and code template are explained.

2.1 Semantics

The meaning of a statement or operator of a programming language (“program statement”) is its assigned static 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, or its sense or intention. Thus, the sense of $x++$ depends on the context it is used with. In this example, 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 of defect pieces? This in turn depends on the context, maybe a wider context.

2.2 Annotation

To express the semantic meaning of statements, declarations or blocks of statements or declarations, annotations are introduced in this paper. An annotation as we see it is a behavior conserving construct that can be put above any valid program construct (in contrast to Java’s JSR 175 [2]). A program construct is a declaration, a statement or a block of statements. An annotation is identified by its name. An annotation can also have parameters. The allowed parameters, including their names and types, are determined by the definition of an annotation. To let an annotation express a sense (intention, deeper meaning, contextual information, statement), a set of predefined senses is made known to a processing tool. In turn, each annotation can then be assigned one single statement that may be as complex as necessary. This way the very complex

problem of program understanding is removed while being unable to process and understand statements generically at the same time.

This paper suggests implementing annotations for Java as specific comments. Other approaches such as JSR 175 propose annotations as first-class program elements in Java. But one major problem with JSR 175 as the main concept for annotations in Java is the limited scope an annotation has (single statements are outside the scope of a JSR 175 annotation).

To distinguish annotations from ordinary comments, a unique prefix is added to them after the comment prefix (for Java the two characters `/*`). This paper suggests the double at-sign `@@`. Meffert [10] describes an implementation for annotations and recommends using a composite annotation that embodies both a machine-interpretable part and a human-readable part. Besides, hash codes can be attached to annotations to recognize changes in annotated code fragments with high probability, and thus be able to ask for revalidating the annotations in question.

2.3 Transformation

A transformation in the context of this paper is the process of getting to a target code from a given source code by applying defined rules. A rule always produces the same result when applied onto the same source code (with no rule changed in the meantime). 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 by trying to apply the pattern as far as possible. It 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, the transformations and the result. That in turn makes it impossible, in our opinion, to get a machine to transforming arbitrary pieces of code in any context. That is why we suggest a workaround for this problem currently unsolvable in our eyes. Our suggestion leads to an extended support when transforming code that reduces manual work but cannot make it superfluous.

2.4 Code pattern template

A code pattern template contains any information about a pattern necessary for a specific process (selection, application, recognition). This includes static and dynamic parts, and source code as well as semantic information (compare Krahn and Rumpe [8] for a template without explicit semantic information). To use a template effectively it must be connected with a given source code, the context. Connecting the template with code means finding anchors in the context that justify the applicability of the template for the context. Semantic anchors in source code are provided by annotating the code during software development, as this paper proposes.

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 matching process.

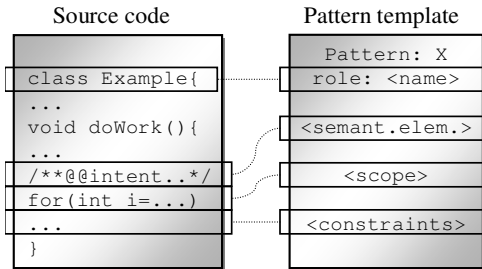


Fig. 1. Matching a pattern template with source code via annotations

As displayed, the template can also contain constraints that can perform extensive validations, like evaluating preconditions. Additional logic included in the template is used for *Abstract Syntax Tree* (AST) analysis (such as finding a variable declaration of a specific type in a given scope) as well as for handling information gained by matching code and template. Routines implemented such logic are called *handlers* here. A handler as well as a constraint checker is implemented in pure Java code. It can utilize a library that provides often required logic. This library is supposed to grow during the creation of new code templates. The reason why handlers and checkers are implemented programmatically and not supplied via a simpler scripting language is the capability and flexibility such logic must have.

The information that led to an explicit code pattern template definition, are stored along with it (see section 4).

3 Towards a new software development concept

The presented approach suggests extending common code-based development scenarios (as described in section 1.1) 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 quite popular concept of annotations. Future tools should support developers in adding annotations to the code they want to enrich semantically. Our vision is that modern IDEs will suggest annotations to the developers which will be added automatically to their code. Besides, an IDE should verify applied annotations and ease their definition (compare Meffert [9]). Adding semantics to code pattern templates can occur in any satisfying way, because templates need not be compilable necessarily (in contrast to Krahn and Rumpe [8] who describe a different approach).

The result of having explicit semantics in source code as well as in code pattern templates allows finding an appropriate template for a given context (i.e. a source code) in order to transform the context 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 [10]). The reason is that it is nowadays simply not possible for a machine reasoning about the sense of an arbitrary piece of code. There are exceptions (such as speaking names), but to put them onto a commonly understood basis for code analysis, these exceptions would have to become standards which sounds contradictory. By introducing annotations, a more practicable mechanism is suggested.

When working with code patterns, tool-support is possible for the processes

- definition of code pattern templates,
- recognition of applied patterns in a given source code,
- selection of an applicable pattern for a given source code, and
- application of a selected pattern onto a given source code.

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. After that, an extended example of creating a code template is given.

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. A tool could assist that process by suggesting annotations for a given code: Either by comparing the code with already known cases or by evaluating speaking names or other coding conventions seen as common. Only presenting valid annotations for a given scope further narrows down the number of annotations to consider.

The introduction of semantic information into non-compileable code templates is different than for source code. As only experienced developers are able to define templates (see the next section), more competence and engagement in proceeding with this process and understanding it is assumed. Code templates also need 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. Each one results from a specific aim the developer has. The alternatives are that the developer

- knows a pattern to apply but wants support in applying it,
- knows a pattern that might apply and wants to get an acknowledgement or a better proposal,

- does not know which pattern to select and wants support in selecting and applying one,
- wants to know which patterns may exist in the code respectively to which extent they are existent,
- knows about a problem or antipattern with the source code and wants to fix it, or
- wants to optimize the source code by introducing a pattern.

Depending on the aim a developer has, the procedure for applying annotations is different. A contextual help system can assist developers in finding annotations finally helping them to solve their design problem. The easiest case is when a pattern is selected already and has to be applied. Adding annotations to source code in this case is quite simple because it is known which annotations are required, namely such that correspond to the ones contained in the appropriate pattern template. During the process of adding known annotations, it might come out that one or more annotations are not applicable to the given code. This is an indicator that the desired pattern might not be valid without refactoring the code. In case the refactoring is successful and the missing annotations can be added, a repository could store those valuable information for detecting similar situations in the future and be able to present a proposal for refactoring for the next similar case. Applying annotations to point out problems or drawbacks in code is similar to ordinarily annotating code. I.e., an annotation expressing “creating this object instance is too slow” can be defined as correspondence to “speedup object creation” (compare the pattern *Cache Proxy* from Gamma [1], among others). Also see Meffert [10] for a description of perspectives when annotating code.

Finding a pattern to apply is more difficult as the source code to be examined has to be annotated appropriately. This means the code must contain annotations corresponding to semantic information defined in templates of applicable pattern. Thus, this case is comparable to proactively annotating code. In general, an annotation has to be added to the code where an analysis tool is not able to present a qualified suggestion (because no reasoning about the sense of a code block is possible). A repository with already annotated code aids the tool in that. Program elements that lie outside the scope of any defined annotation are ignorable. The more input a developer gives the tool (i.e. by answering a question like *What is my goal?*) the more the set of relevant annotations can be narrowed down. By weighting the expressiveness of annotations, developers could be asked to first try to apply more expressive annotations before trying the less informative ones. Another perspective for annotating code is displayed as follows with the creation of a code pattern template.

4 Creating a code pattern template

This section presents an example of how to create a code pattern template. The result is a template for the quite complex design pattern *Composite* (Gamma [1]). It is also described how to retrieve the template definition from an exemplary transformation of

a source code suitable for the application of *Composite*. To sum up, the example shown below exposes the following issues:

- Defining the different variants of *Composite* and finding a suitable one is supported by our approach.
- Defining and executing transformations from source to target code is feasible with the proposed approach.
- The structural pattern *Composite* is more complex as multiple classes build up the pattern.
- Many implementations of *Composite* exist that are seen as correct. Any of them can be handled separately.
- The wrong or partial implementation of *Composite* is possible in a lot of fashions.
- Recognizing wrongly or partially implemented *Composite* is possible with semantic information.

Furthermore, the process of obtaining a code template definition, including annotations, is shown. The process includes the activities:

1. Provide a target code where the selected design pattern already is applied.
2. Choose a suitable source code as base for applying a selected pattern to.
3. Transform code from step one to step two and record the steps 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.

Beginning with activity one, the next paragraphs exercise the process for instance.

4.1 Initial source code

It follows a Java code sample that shows the initial source code for a degenerated *Composite*, beginning with the Client class constructing a composed graphics. To avoid printing the listings a second time, annotations were introduced as follows (normally, annotations are not existent in an initial source code). The code shown is suitable for applying the *Composite* pattern on. Second, it was taken care that the code is not too far away from the target situation shown in section 5.2. Although the latter is not a precondition, it eases explanations and reduces difficulties with transformations (see section 5.3). To simplify the example, very simple annotations without parameters were used (in opposite to what is described in [10]).

```
/*@@COMPOSITE_CLIENT_CLASS*/
public class Client {
    public static java.util.List elements;

    public Client() {
        elements = new java.util.Vector();
    }
}
```

```

        Graphics g = new Graphics();
        Picture p = new Picture();
        /*@@COMPOSITE_ADD_ELEMENT*/
        elements.add(p);
        elements.add(new Text("This is a text"));
        elements.add(new Line(2, 3, 17, 42));
        elements.add(new Line(8, 1, 9, 14));
        g.create();
    }
}

```

In the above code, the annotation *COMPOSITE_ADD_ELEMENT* is just written once. For the other two statements beginning with *elements.add*(the annotation can be skipped in case there is a handler reasoning that these two statements must be analogue to the annotated statement.

Here are the relevant excerpts of the other classes:

```

/*@@COMPOSITE_COMPONENT_CLASS*/
public class Graphics {
    /*@@COMPOSITE_OPERATION*/
    public void create() {
        for (int i = 0; i < Client.elements.size(); i++) {
            Object o = Client.elements.get(i);
            if (o.getClass() == Picture.class) {
                (Picture) o).paint();
            }
            else if (o.getClass() == Text.class) {
                (Text) o).print();
            }
            else if (o.getClass() == Line.class) {
                (Line) o).draw();
            }
        }
    }
}

```

```

/*@@COMPOSITE_COMPOSITE_CLASS*/
public class Picture {
    /*@@COMPOSITE_OPERATION*/
    public void paint() {
        // picture is painted here
    }
}

```

To sum up, the aspects that degenerate the above sample code from *Composite* are:

- The children of class *Graphics* are not held in a list attribute in *Graphics*, but in a static attribute in *Client*.
- Each class has an operating method with different name (*create*, *paint*, *print*, *draw*) instead of a unified name.
- The hierarchy between elements (containing and contained elements) is represented by the order of the elements in the static attribute *elements*, not by holding child elements by their parents.

The classes *Text* and *Line* are analogous to the above class *Picture* and do not contain any logic noticeable in our context. *Text* and *Line* are annotated in analogy to *Picture*, but the annotation name at class level is *COMPOSITE_LEAF_CLASS*. It is remarkable that method *create* in class *Graphics* is satisfactorily determined by adding one annotation *COMPOSITE_OPERATION* above it. The reason is that an assumption is made for any method annotated that way. The idea is that such an annotation equalizes the logic contained in the method's body and reduces it to one "statement", namely "this method is the analogy to the operation-method in Composite's role *X*" (with *X* being a class identified by another annotation, such as *COMPOSITE_COMPOSITE_CLASS*). Conclusions from introducing such homogeneity are that code transformation must rely on known cases, which is not necessarily a problem itself. It is just quite time-consuming identifying similar cases and the transformations into one or several analogous target codes. Also, it might be that an annotation above a method might be too coarse-grained for a method implementing multiple relevant aspects. But then the situation is resolved latest when applying the pattern. There it should prove whether the transformed code still does what it should do. The previously mentioned mechanism of comparing known code blocks will nevertheless be able to pop up an alert to inform developers annotating a multi-purpose method with an unqualified annotation.

4.2 Target code

To demonstrate how a transformation from the source situation above to a defect-free implementation of *Composite* can be executed, the desired target code is shown. It represents a known *Composite* implementation that is a priori context-free. Also some annotations were added, finding them is described later. The corresponding classes could look like this (other alternatives are possible):

```

/*@@COMPOSITE_CLIENT_CLASS*/
public class Client {
    public Client() {
        Graphics g = new Graphics();
        Picture p = new Picture();
        g.add(p);
        g.add(new Text("This is a text"));
        g.add(new Line(2, 3, 17, 42));
        g.add(new Line(8, 1, 9, 14));
        g.operation();
    }
}

public interface IComponent {
    void operation();
    void add(Component component);
}

```

```

/*@@COMPOSITE_COMPONENT_CLASS*/
public class Graphics implements IComponent {
    private List elements = new Vector();
    /*@@COMPOSITE_ADD_ELEMENT*/
    public void add(IComponent c) {
        if (c.getClass().isAssignableFrom(Graphics.class)){
            throw new RuntimeException(
                " Graphics must not be added to itself");
        }
        elements.add(c);
    }
    /*@@COMPOSITE_OPERATION*/
    public void operation() {
        for (int i = 0; i < elements.size(); i++) {
            IComponent c = (IComponent) elements.get(i);
            c.operation();
        }
    }
}

public interface IComposite extends IComponent { }

/*@@COMPOSITE_COMPOSITE_CLASS*/
public class Picture implements IComposite {
    private List elements = new Vector();
    /*@@COMPOSITE_ADD_ELEMENT*/
    public void add(IComponent c) {
        elements.add(c);
    }
    /*@@COMPOSITE_OPERATION*/
    public void operation() {
        paint();//picture itself is background
        for (int i = 0; i < elements.size(); i++) {
            IComponent c = (IComponent) elements.get(i);
            c.operation();
        }
    }

    public void paint() { //paint the picture here }
}

public interface ILeaf extends IComponent { }

/*@@COMPOSITE_LEAF_CLASS*/
public class Text implements ILeaf {
    private String line;
    public Text(String a_text) {
        line = a_text;
    }
    /*@@COMPOSITE_ADD_ELEMENT*/
    public void add(IComponent c) {
        throw new RuntimeException("not supported");
    }
}

```

```

    /*@@COMPOSITE_OPERATION*/
    public void operation() {
        System.out.println(line);
    }
}

```

Class *Line* is not shown as it is analogous to class *Text* (*Line* has just a different implementation of method *operation*). It should be noted that annotation *COMPOSITE_ADD_ELEMENT* moved from class *Client* of the source code from section 4.1 to class *Graphics* in the target code. That is possible because it does not matter if the caller of a method is assigned the annotation or the method itself. The reason to choose class *Graphics* for the annotation is that the client class should not become part of the template in general. The client class is cared by the transformations given in the next section. It is not part of the explicit template definition. Dependencies between the client class and the other roles the pattern in question has (also compare [13]) are given by the existence of source and target code. Thus, the definition of a client class in the template would hold no additional information that could be exploited.

Annotations were added to source and target code wherever an link between them was required that could otherwise not be reasoned about. Thus, an annotation is required for any part in code for which the intention cannot be evaluated via (static) AST analysis satisfactorily. It depends on the capabilities of a tool which evaluations are possible, but an experienced developer should be able telling for which pieces of code a machine cannot uncover its intention (the so-called program understanding).

The annotations in the source code were introduced reversely, i.e. after having the target code. This way, the significant parts of the pattern (manifested in the target code) were known already.

Now we proceed with the transformation of the initial code until the target code is manifested.

5 Defining the transformation from source to target code

This section describes transformations to get from the source code of section 4.1 to the target code from section 4.2. The transformations described below were found intuitively, namely by an experienced developer comparing source with target manually. Tool-support for this comparison is possible and could be accomplished by starting with the source code and doing one transformation after each other to get to (until then not existent) the target code. This is in contrast to the approach presented here, because it is easier to follow in the example when having the source and the target situation presented first and then obtain the transformations. However, both approaches are valid.

The transformation steps concluded can be used to gain valuable information about the transformation rules for revolving a code such as from section 5.1 to a *Composite* version such as the one from section 5.2. The transformations were recognized manually as follows and occur in the given order. Each transformation is described by the aspects

- high-level description: a natural-language like description of the transformation,
- low-level actions: the actions to undertake to execute the transformation
- preconditions: the conditions that must be fulfilled in order to validly execute the transformation, and
- remarks: optional comments on previous descriptions.

The low-level actions are to a certain extent comparable to what Ó Cinnéide [5] calls *minitransformations*. In this paper, each low-level action can either be a call to a service method determining needed information, such as where a specific attribute is used. Or the action is a transformation itself, such as *make attribute X private*. To express the generic character of actions, each variable element in an action is assigned a variable (starting with the capital letter A and ascending in the alphabet).

5.1 Transformations for our example

For the example from the previous section a number of transformations (or actions, to use a more general term) can be determined. For each action, the variable assignment for our *Composite* example is given. The transformations found manually are numerous. They are listed completely and described in detail. We find these (partially low-level) explanations necessary to illustrate the capabilities of and limitations with transformations and the validity of our approach.

5.1.1 Move attribute “element container” from class Client to class Graphics

High-level description:

1. Move attribute *elements* from *Client* to *Graphics*.
2. Ensure visibility *private*.
3. Ensure that attribute’s accessibility level is instance (not static).

Low-level actions:

1. Move declaration for attribute A from class with role B to class with role C ($A = elements, B = Client, C = Composite$) and introduce inline construction of the attribute.
2. Make attribute A in class B private ($A = elements, B = Graphics$).
3. Change accessibility of attribute A in class B to instance level ($A = elements, B = Graphics$).
4. Remove any construction statement of attribute A from class B ($A = elements, B = Client$).

Preconditions:

1. After applying the low-level actions: The attribute *elements* in class *Client* is not used other than to add elements that are processed by class *Graphics*.

Remarks:

- For low-level action 1 there are different ways expressing which class is concerned. The following transformations show possible variations for the description meaning the same in the end (e.g. “Find class with annotation A” instead of “Find class with role B” with A a corresponding annotation

tagging a class as having role *B*). We chose to do so to signalize that even a low-level action can be realized by using another action or combining multiple low-level actions.

5.1.2 Introduce method *add* to class *Graphics*

High-level description:

1. Add method *add* with the code shown in section to class *Graphics*.
2. Adapt the logic in the method's body.

Low-level actions:

1. Add method *A* (logic as given in target code, including the annotation) to class *B* ($A = add, B = Graphics$).

Preconditions:

1. Non-critical: Method *add* does not exist already
2. Critical: Method *add* does exist already and has different implementation → User confirmation required or variation known in the repository.

Remarks:

- The method's logic in action one will be replaced by a parameterized version in the code template to obtain (see further down).

5.1.3 Replace access to method *add* from attribute elements with sophisticated method *add* in class *Graphics*

High-level description:

1. Using method *add* from attribute *elements* directly is forbidden.
2. Instead, use method *add* from class *Graphics*.

Low-level actions:

1. Find list variable annotated with *COMPOSITE_ADD_ELEMENT*.
2. Determine any place where method *A* for attribute *B* is called statically ($A = add, B = \text{variable from action one}$).
3. For all found places:
 - a. Determine instance variable of class *A* that is available at the time method *C(D)* of attribute *B* is called ($A = Graphics, B = \text{variable from action one}, C = add, D = \text{signature of method } C$).
 - b. Find class with annotation *A* ($A = COMPOSITE_COMPONENT_CLASS$).
 - c. Replace $A.add(B)$ with $C.add(B)$ ($A = \text{variable from step one}, B = *, C = \text{instance variable for class found in previous step}$).

Preconditions:

1. Action 3a) returns non-empty result for any place found.
2. For action 3a): There exists a class with the given annotation. Such a precondition is omitted for the following transformations.

Remarks:

- We assume that there are no dynamic calls to method *add* from attribute *elements*. Dynamic calls cannot be discovered reliably.
- Action 2a is another way of expressing that a specific class is meant. A precondition for using this description is that class *Graphics* was identified be-

fore as incorporating role *Component* of the *Composite* pattern (compare the first transformation where this already happened).

5.1.4 Adapt method create in class Graphics

High-level description:

1. Rename method *create* in class *Graphics* to *operation*, if necessary.
2. Replace code in renamed method with code from method in target code.

Low-level actions:

1. Find class with annotation *A* ($A = \text{COMPOSITE_COMPONENT_CLASS}$).
2. Find method with annotation *A* in previously found class ($A = \text{COMPOSITE_OPERATION}$).
3. Rename previously found method to *A* ($A = \text{operation}$).
4. Replace code in previously renamed method with logic from method *A* of target code class *B* ($A = \text{method name found in action 2}$, $B = \text{Graphics}$).

Preconditions:

1. Examine whether the method to be adapted corresponds in its behavior with the adapted code (i.e. by evaluating possibly existent annotations or by comparing code blocks from a repository with the source code). Display the recognized behavior to the user. If difference detected: Inform user what the method is supposed to do. If no difference detected: Suggest proceeding with transformation.

Remarks:

- In low-level action four, the method *A* contains code that is context-dependent. The content-dependent parts will later on be parameterized in the code template.
- For reasons of simplifications, here no parameters are passed to the methods called in method *operation*. Considering parameters would be possible by introducing a generic data transfer object (DTO) that is a container object holding any parameter of the operation methods from classes *Picture*, *Text* and *Leaf*. The DTO could then also be introduced to method *operation* and to the callers.

5.1.5 Adapt method paint in class Picture

The following is also valid for class *Text*.

High-level description:

1. Add a new method *operation* that iterates over all children of *Picture* and also executes the original *paint*-method.
2. Find all calls to method *paint* of class *Picture* and replace them with a call to method *operation*.

Low-level actions:

1. Add method *A* to class *B* and adapt so that method *C* is called ($A = \text{operation}$, $B = \text{Picture}$, $C = \text{paint}$).
2. Determine any place where method *A* is called and replace it with call to method *B* ($A = \text{paint}$, $B = \text{operation}$).

Preconditions:

1. Within method operation, any parameter that must be passed to method paint must be known. This is a constraint that cannot be coped with in this paper. It may lead to a transformed code that is not compilable. The main idea behind this is: Better give a developer a nearly perfectly transformed code that needs slight rework than give him no transformed code at all. With annotations it is very probable that the source code to transform conforms at least close to what is expected.

5.1.6 Let class *Graphics* implement interface *IComponent*

The following is valid for the classes *Picture*, *Text* and *Line* in an analogue way (namely with different interfaces).

High-level description:

1. Introduce interface *IComponent* as shown in the target code.
2. Add a directive to class *Graphics* that lets it implement interface *IComponent*.

Low-level actions:

1. Add a class for interface *A* as shown in the target code ($A = IComponent$).
2. Add the statement *implements A* to class *B* ($A = IComponent, B = Graphics$).

The low-level actions described above are a direct start in obtaining reusable actions. With every template definition, the number of actions to be defined from scratch is likely to be reduced.

5.2 Verifying the transformation

Because finding transformations as described above is a nontrivial task, errors may occur during this process. To avoid or uncover errors as good as possible, we suggest following the hints:

- Check that for any annotation added to the source code, a corresponding annotation in the target code must exist.
- For any difference between source and target code a transformation must exist.
- For each transformation the necessary preconditions must exist. It seems as if there is no other technique to obtain these preconditions than by thinking about which are obligatory. See section 5.5 for further considerations on preconditions.
- Test the transformations on a slightly modified piece of code. For example, modify the visibility of attribute, methods, and classes. The modified code needs not be compilable as it is only about verifying transformations in this step.
- Each annotation must either exist in the repository. Or, if not it has to conform to a defined syntax (see [10]).

It is likely to occur that developers defining transformations as described in the previous section forget a transformation or a precondition. To find out such incom-

plete definitions, they have to be proved by being taken under test. A test should ideally be executed before productive use of the template, although the latter may also prove useful in finding malicious definitions.

5.3 Obtaining a code template definition

This section demonstrates how a code pattern template can be obtained from providing a source and a target code plus transformations conducting the former into the latter. A template is a definition of a code pattern that can later on be used to be matched with an annotated source code (compare section 4.1). The procedure to get to the template involves the following steps:

1. Consider the target code: For each class (not interface!) in the target code:
 - 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. It may be necessary introducing logical blocks (in Java with `{` and `}`) that group subsequent statements without changing the behaviour of the program.
 - e. For logic that should be kept as the original from the source code is, add a control tag `~INSERT_CODE(Role <role>, Method <annotation the relevant code is annotated with>)`.
 - f. For logic within a block (e.g. within methods) that may be extendable, add a documentation link. A documentation link references a description describing the possibilities to extend the logic. Future work will concentrate on the formalization of this concept.
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. Each interface is assigned visibility *public*.
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 (discusses later):
 - a. Annotation definition does not exist: Create annotation definition.
 - b. Otherwise: Update annotation definition.
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 already, and,
 - c. executing precondition checks.

The *Composite* pattern has the roles *Client*, *Component*, *Composite*, and *Leaf* (compare Gamma [1]). Beginning with the first two steps of the above activity list, we are able adding these roles to the code template by copying the target code from section 4.2 and assigning each class the appropriate role of the *Composite* pattern, except *Client* (see section 4.2):

```

[Interface IComponent]
public interface <role> {
    void operation();
    void add(<role> component);
}

[Role Component]
public class <role> implements <role:IComponent> {
    private List elements = new Vector();
    @COMPOSITE_ADD_ELEMENT
    public void add(<role:IComponent> c) {
        if (c.getClass().isAssignableFrom(<role>.class)){
            throw new RuntimeException(
                "<role> must not be added to itself");
        }
        elements.add(c);
    }
    public void operation() {
        for (int i = 0; i < elements.size(); i++) {
            <role:IComponent> c =
                (<role:IComponent>) elements.get(i);
            c.operation();
        }
    }
}

[Interface IComposite]
public interface <role:IComposite>
    extends <role:IComponent> { }

[Role Composite]
public class <role> implements <role:IComposite> {
    private List elements = new Vector();
    @COMPOSITE_ADD_ELEMENT
    public void add(<role:IComponent> c) {
        elements.add(c); //optionally forbid certain classes
    }
    public void operation() {
        paint(); //picture itself is background
        for (int i = 0; i < elements.size(); i++) {
            <role:IComponent> c =
                (<role:IComponent>) elements.get(i);
            c.operation();
        }
    }
    public void paint() {
        ~INSERT_CODE(Role @@COMPOSITE_COMPOSITE_CLASS,

```

```

        }
    }

    Method @@COMPOSITE_OPERATION

[Interface ILeaf]
public interface <role> extends <role:IComponent> { }

[Role Leaf]
public class <role> implements <role:ILeaf> {
    private String line;
    public Text(String a_text) {
        line = a_text;
    }
    public void add(<role:IComponent> c) {
        throw new RuntimeException("not supported");
    }
    public void operation() {
        ~INSERT_CODE (@@COMPOSITE_LEAF_CLASS,
            Method @@COMPOSITE_OPERATION)
    }
}

```

Some explanations to the above code: The sections marked with Interface are special roles names that have to be consistent throughout the template but can be chosen freely, are realized via slots. A slot is defined by an arbitrary name that is unique within the template containing all the roles. To recognize a slot, its name is surrounded by < and >. The special slot <role> gets assigned the value that follows the section identifier *role* or *interface*. The slot <role:X> references the section marked by *[Role X]* or *[Interface X]* and gets assigned the value of the referenced role. Which concrete name a role and thus a class or interface has depends on what the developer enters when asked about it on the application of the template. The control tag *INSERT_CODE* has two parameters that identify a method in the source code. The code from within this method is to be copied to the target code during transformation. This may lead to premature transformations which we think is unavoidable in some cases.

By looking at the transformations 5.1.2, 5.1.4 and 5.1.5 that contain code to be introduced to the target code, slots can be identified. A slot has to be added for any variable that is context-dependent (see point 1d in the above list and compare method add in the section of role *Composite* above in this paragraph).

A by-product of the process obtaining a code template definition is the definition of annotations. I.e., a concrete (posteriori) definition of annotations introduced a priori during the transformation phase is possible. With the next example the above-mentioned steps are demonstrated:

Step 3 of the activities necessary to create a code template results in annotation definitions. The annotations existent in the source and target code (sections 5.1 and 5.2) are equivalent with respect to their name, but different concerning their scope. The scope of an applied annotation is the statement the annotation is applied above. This concrete scope must be within a generically defined scope that is manifested with the annotation definition. It follows an overview of the annotations applied to the code

from section 5.1 and 5.2, together with the code in source and target code the annotations were applied to.

Table 1. Overview of annotations applied in section 5.1 and 5.2

Annotation name	Code in source	Code in target
COMPOSITE_ADD_ELEMENT	elements.add(p)	public void add(IComponent c)
COMPOSITE_CLIENT_CLASS	public class Client	none
COMPOSITE_COMPONENT_CLASS	public class Graphics	public class Graphics implements IComponent
COMPOSITE_OPERATION	1. public void create() 2. public void paint()	public void operation()
COMPOSITE_COMPOSITE_CLASS	public class Picture	public class Picture implements IComposite
COMPOSITE_LEAF_CLASS	public void print()	1. public class Text implements ILeaf 2. public class Line implements ILeaf

As can be seen, the annotation *COMPOSITE_OPERATION* was applied to two different code statements. Thus, the scope to be defined for the annotation must include both scopes that correspond to the applied code statements. Here, the scope is the same for both cases as just the name of a method is different. The scope matching both methods *public void create()* and *public void paint()* is “public method with empty signature”, or to give it a reusable (arbitrary but unique) symbolic name: *METHOD_PUBLIC_SIGNATURE_0*. Under this symbolic name the scope of other annotations can be expressed concisely. Symbolic names for scopes are stored in a repository that can be accessed by the logic relevant for processing annotations.

Defining an annotation means creating an initial definition which can then be improved in iterations. A new iteration is possible if the annotation already defined is applied to another program element. After application, a review of the applied annotation’s scope is possible followed by a comparison of the annotation definition regarding its conformance to the new application.

After a code template definition is created, different source codes can be evaluated with it. When comparing other source codes with the target code of the template, for every applied annotation three cases can be distinguished:

1. Annotation both exists in source and target code, possibly with different names.
2. Annotation only exists in source code.
3. Annotation only exists in target code.

Case one means that source and target code match conceptually. Annotations of case two are not necessary in the target code because they can be reasoned by in a different way (e.g. by matching with a different annotation, or by exploiting AST-information with help of a handler routine) or because in the target code there is miss-

ing a corresponding (i.e. semantically equivalent) code block. The third case includes source code that does not contain (semantically equivalent) parts of the target code. Case one produces a direct match between source and target code whereas for cases two and three making a match depends on whether a missing annotation is just missing because it was forgotten or it is not adequate applying it (here user input is required). Making a match between source and target code means that the transformations connected with the code template are likely to be possible. Verifying the preconditions stored with the template allows to distinguish the applicability of the pattern expressed by the template better. In case no precondition is violated the applicability is given to a high degree. A violated precondition lowers the applicability.

5.4 Utilizing a code template for varying source code

It is likely that a piece of code is different from a source transformed initially to obtain a template (as in section 5.3). This paragraph illustrates with an example how the template can be exploited under different contexts. Further, it is shown how new variations of the template can be achieved.

Assume we have a class *Client* that is different to that shown in section 4.1 and that looks like:

```
/*@@COMPOSITE_CLIENT_CLASS*/
public class Client {
    public Client() {
        Graphics g = new Graphics();
        /*@@COMPOSITE_ADD_ELEMENT*/
        g.addPicture(new Picture());
        /*@@COMPOSITE_ADD_ELEMENT*/
        g.addText(new Text("This is a text"));
        /*@@COMPOSITE_ADD_ELEMENT*/
        g.addLine(new Line(2, 3, 17, 42));
        g.addLine(new Line(8, 1, 9, 14));
        g.create();
    }
}
```

In this instance class *Graphics* is different to the initial example in that it provides methods for adding the child elements *Picture*, *Text*, and *Line*. The above code already contains annotations to avoid printing the code a second time (normally, the annotations are to be added after the code is considered!). As can be seen, each *addXXX*-method must be annotated the same way to match *Composite's operation*-method.

Taking the transformations from section 5.1, we can recycle some steps, skip others, modify some and introduce new actions. Transformation 5.1.1 is obsolete. The transformations 5.1.2 and 5.1.3 must be modified. About the other transformations, the example above does not allow making any assumptions, for sake of simplicity we assume them to be equivalent. To modify the two mentioned transformations, it could be thought of the *addXXX*-methods in class *Graphics* of three methods with different signature but equivalent logic (e.g. *elements.add(inputparam)*). The only arguable

reason having such a code is that classes *Picture*, *Text*, and *Line* do not share a common interface. This means we would obtain an additional transformation that introduces an interface to each of these three classes (compare the refactoring activity *Introduce Interface* described by Fowler [11]). Transformations 5.1.2 and 5.1.3 would then include the replacement of all *addXXX*-methods by one *add*-Method in class *Graphics* and for the callers of the methods.

Concluding, the template defined previously could remain unchanged for most parts, just some transformations would have to be adapted, deleted or introduced. For these, previously defined actions could be recycled. Besides, in the definition of annotation *COMPOSITE_ADD_ELEMENT* the scope would have to be extended so that not only calls to a method for a class derived from a collection (such as *java.util.Vector*) are included but also calls to methods of other classes (such as *Graphics*).

5.5 Exploiting the code pattern template

Recapitulating, the information we have after defining a code template is: the initial source code, the annotated source code, the transformed and annotated target code, the transformations leading from source to target, a code template and annotation definitions. Source and target code as well as the code template contain information about the role relations and the relations of annotated pieces of code. This valuable data can be used to determine if preconditions met before selecting or applying a pattern, or to detect applied patterns within a given source code. By comparing applied annotation within a source code with code pattern templates, partly implemented patterns can be recognized, or, to see it from a different perspective, missing annotations can be detected (in case the pattern in question is applied fully).

Having explicit semantics in source code can be an enormous advantage for implementing preconditions because not only can syntactic constraints be validated but also semantic ones.

The information of a number of source codes that can be transformed into the same target code allows to reason about the similarity or equality (in the sense of comparable behaviour) of different source codes. This in turn can be exploited when trying to detect if changes in annotated code are significant or not (e.g. $x++$ is equal to $x+1$, to give a very basic example). Recognizing synonymous code blocks statistically comes into reach when having large amounts of data accessible (compare the machine translation work by Wang [15]). By gathering information about which pieces of code could be seen as isomorphic (i.e. analogous in their behaviour) it is possible comparing ASTs not for exactly matching with specific elements but for matching with isomorphic elements. This could be seen as a search for synonyms rather than for exact compliance.

6 Related work

Relevant approaches cope with the issues refactoring or code transformation, code templates, annotations in general, and program understanding.

With Java 5 annotations have been introduced as first-class language constructs [2]. They are compilable and can be evaluated either during compile-time or run-time. These annotations are restricted in that they cannot be applied to arbitrary scopes (i.e. single statements are out of the scope). Besides, their form is very brief and not composed as described in Meffert [10]. The Java Specification Request 305 [3] aims at introducing annotations for software defect detection. This work is currently still in progress, but the group's discussions show that the expected result will lead to annotations allowing quite restricted precondition checks (such as a *NotNull* constraint for a method's input parameter).

Jackpot [4] searches Java source code that conforms to generic rules and if, executes transformations automatically. 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, they are static and can be nested.

Krahn and Rumpe [8] 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 (From UML to Java And Back Again) [6] 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. FUJABA adopts annotations for ASTs to record the (more technical) meaning of declarations and statements. Not considered is the high-level (non-technical) meaning of program elements.

Pinot [14] uses static program analysis to capture program intent in order to recognize patterns. Pinot detects design patterns by relying on AST-information and on definitions of obligatory pattern elements. Similar to this paper's approach, Pinot relies on concrete definitions. The difference is that Pinot does not use explicit semantic information and seems to solely rely on significant AST-elements found.

7 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. Developers are required to learn annotating the code they are working with. In turn, the definition of code pattern templates allows obtaining key information about a pattern, including its structure, preconditions, transformations, annotation definitions, and valid source codes to apply the defined pattern to. Defining actions

used for transforming source to target code with parameters makes it easy recycling them.

Creating a code pattern template generates a lot of useful information. Done multiple times a large number of code blocks known by their semantic meaning. This allows to search for and recognize synonymous pieces of codes. Exploiting such information makes complex program evaluation and transformation much more easier than it is today (see [15] for a comparable suggestion in machine translation). The barrier to let a machine “understand” source code can be lowered significantly by having annotated code in the back. Trying to get a machine understanding programs by just examining their ASTs is impossible in our opinion.

To guide developers, they should be assisted in annotating source code. This can be done by a context-sensitive annotation function included in an IDE (e.g. by providing a plugin for Eclipse). By that developers can only choose valid annotations for a certain piece of code, and they get an offer which pieces of code to consider for annotation. To support developers in finding applicable annotations each annotation has to be enriched with informative texts, such as motivating questions that would lead to the application of the annotation if answered positively. Annotating legacy systems is assumed to allow more comfort as specific naming conventions or coding styles are likely to be found throughout an existing code base, at least for the code of each single developer.

Code templates can only be achieved by experienced developers. 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 additional complex patterns, such as *Flyweight* [1] and *MVC* (Fowler [11]), in order to demonstrate the practicability of the described approach. In order to obtain transformations that translate a source into a target code, a tool acting in the manner of a macro-recorder could assist enormously. The implementation of such a tool is optional, but important enough to be considered in the future. Tightly coupled with this issue is the more capable support of control tags like *INSERT_CODE*. Here, code to be transferred from source to target must possibly be adapted generically to avoid premature transformations, although by splitting a code block into chunks, literally, by introducing an annotation for every single distinct code block processing transformations gets easier.

Every annotation definition contains a scope that only permits the annotation to be applied above certain program elements. However, an applied annotation that commits to that scope may not be semantically correct. It would be helpful if a tool remarked about such cases, e.g. by evaluating known cases.

As noticed in section 1.1, no implicit link exists between requirement documents and source code. Here, future work is required, may it be by adding explicit semantic information to requirements (compare Krahn [7]), or by providing them in a machine-processable way.

A vision of this work is a process that requires the developers to add explicit semantic information to important parts of their code. This may be an IDE that will

report errors when certain pieces of code are not annotated. Code of that category is such that is either not known when compared with other projects stored in a repository. Or it may be code that has an ambiguous meaning, because at least two different annotations for the same piece of code are recorded in the repository. In the latter case, a proposal could be presented, leading to a more transparent type of code considering program understanding.

8 References

1. Gamma, E., Helm, R., Johnson R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
2. Java Specification Request 175: A Metadata Facility for the Java™ Programming Language. <http://www.jcp.org/en/jsr/detail?id=175>
3. Java Specification Request 305: Annotations for Software Defect Detection. <http://www.jcp.org/en/jsr/detail?id=305>
4. Jackpot. <http://jackpot.netbeans.org/index.html>
5. Ó Cinnéide, M., Nixon, P.: A Methodology for the Automated Introduction of Design Patterns. Proceedings of the International Conference on Software Maintenance, Oxford (1999)
6. 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 (2000), 241-251
7. Krahn, H., Rumpe, B.: Towards Enabling Architectural Refactorings through Source Code Annotations. In: Mayr, H.C., Breu, R. (eds.): Proceedings der Modellierung 2006, Innsbruck (2006)
8. Krahn, H., Rumpe, B.: 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 (2006)
9. Meffert, K.: Supporting Design Patterns with Annotations. 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS '06), Potsdam (2006) 437-445
10. Meffert, K., Philippow, I.: 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 (2006) 48-67
11. Fowler, M: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
12. Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.: A System of Patterns. Pattern-Oriented Software Architecture. John Wiley and Sons (1996)
13. Hedin, G.: Language Support for Design Patterns Using Attribute Extension. In: Bosch and Mitchell (Eds.): Object-Oriented Technology. ECOOP'97 Workshop Reader (LNCS 1357), Springer (1997) 137-140
14. Shi, N., Olsson, R.A.: Reverse Engineering of Design Patterns from Java Source Code. In: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06) - Volume 00 (2006) 123-134
15. Wang, Y.: Grammar Inference and Statistical Machine Translation. Carnegie Melon University (1998)