Supporting the Selection of Design Patterns by Formal Definition and Considering Semantics

Klaus Meffert
Technical University Ilmenau
plop@klaus-meffert.de

Abstract

Extensibility and maintainability of software becomes more an issue as the complexity of the software development process rises. Design patterns in the sense of Gamma et al. [3] aid in reducing the problem of architectural decay. However, new publications steadily increase the number of documented patterns. This makes the automated and tool-supported processing of design patterns more important. Here, the definition of pattern templates receives prominent relevance. A pattern template is a semantically enriched definition of a pattern for a specific process, e.g. the selection of the pattern. This paper supports the definition of pattern templates by introducing manually added semantics via annotations to the template and to source code, along with distinguishing several types of elements that correspond between pattern template and source code. Annotations allow for explicitly declaring the sense/intention and the meaning of pattern and program parts that would otherwise not be reasonable by a tool. This leads to the possibility of tool-supported selection of applicable patterns and assists in their application and detection.

Keywords: design patterns, pattern documentation, program transformation, abstracting architecture, annotations, explicit semantics.

1 Introduction

Design patterns help in building maintainable and evolvable software. Typically, a practitioner learns about a pattern by reading a pattern book, a pattern paper, or examining UML diagrams or source code to understand the pattern and capture the activities necessary to select, apply, and detect it. Then, maybe another look at the pattern documentation is necessary in order to really know how to implement the pattern considered as applicable. After selecting a pattern, the next step is to choose among the documented variants. Then, the relevant pattern can be applied by adding the static parts (such as interfaces and abstract base classes) of it without modification, and adapting the dynamic parts to the given context. In the end, dynamic parts are applied by modifying the given source code. Current IDEs assist the developer in applying a pattern by coping with the static parts and generating a frame for dynamic parts, if possible. Often, such a frame is commented with "*fill in your code here*", because the IDE cannot reason about the semantics of a pattern and the given piece of code sufficiently. The missing ability of understanding programs by a machine and thus assisting a developer with a tool with patterns is a result of the information description and complexity of patterns. The author believes that tool-support is vital for developers using patterns efficiently and correctly.

Thus this paper tries to enhance tool-support for working with patterns in projects where source code is the main driver (in contrast to model-driven development, e.g.). It is described how to select patterns for a given source code by setting up a formal pattern definition, which – as well as the source code in question – is enriched with semantic information. The semantic information helps abstracting the code and the architecture to meanings and intentions. A tool is then able to suggest appropriate patterns for a given code to the developer. The paper targets at practitioners that like to (formally) define or use patterns. It tries to envision how development could benefit from bringing in semantic information to source code and pattern documentations.

In the next section, the problem with tool-assisted patterns is described. Section 3 mentions related work. In section 4, a new approach for defining patterns and enhancing tool-support is introduced. Section 5 and 6 show how to make use of the new approach in general. Section 7 introduces a case study. The final section 8 concludes and closes with future tasks.

2 Problem statement

There are several issues to be resolved when working with patterns:

1. Before a developer can select a pattern for a piece of code, he/she must try to understand the meaning of the code. Otherwise it is not feasible to identify possible problems with the code.
2. Besides that, it is vital to have expert knowledge about a set of patterns for working with them, may it be their selection, application or recognition. Without such knowledge, chances

of selecting the wrong pattern or applying it incorrectly are high.

3. It is not enough knowing only very few patterns, because then one is maybe confronted with choosing either no pattern to apply or select one that does not exactly fit to the design problem within a program.
4. The setback gets even worse when applying a pattern wrongly or applying the wrong pattern for a problem; it may do more harm than it does good.
5. Prerequisite for selecting, applying and detecting patterns is a proper definition of them containing syntax as well as semantic information about the pattern parts. This is necessary because the pattern's context and solution have to be identified.
6. An informal pattern description such as the one proposed by Gamma [3] is not sufficient as input for tools as they cannot parse natural language satisfactorily. The author believes that the tool-processable definition of patterns is currently not solved in a satisfying way.

To fully benefit from design patterns, it is vital from the author's point of view overcoming or reducing these problems by supporting the practitioner working with patterns. The main problem that inspired this paper is the desire for effectively supporting tool-based software development using patterns. This problem gets more complex as new patterns are documented from time to time.

A refined problem statement targets at a solution proposed in this paper for answering the question "How can a design pattern be applied automatically onto a given source code?" To answer this question the problem has to be solved how a given source code can be transformed into a desired target code by subsequently applying a number of transformations out of a set of predefined transformations. For this two tasks must be executed. The first one is identifying the transformations suitable for a given source codes in order to apply a specific pattern. The second task connected to the first one is to determine all information necessary to do so.

3 Related work

This section describes existing approaches that try to support the use of patterns during the development process and to resolve the problems discussed previously. Existing tools (e.g. the development environments Borland Together or microTOOL objectiF) already support the definition of immutable pattern parts together with their automated integration into a given piece of code. The tools just mentioned do not support pattern elements depending on the context (i.e. source code). Up to now, numerous approaches are available that pick the description, selection, application or recognition of design patterns as a central theme. Some of them will be discussed as follows.

Ó Cinnéide [6] introduces minipatterns and minitransformations for the definition and tool-supported application of patterns. Minitransformations are refactoring operations resulting in the application of a pattern into a target source code. The purpose of minipatterns is to describe design patterns. A design pattern is then regarded as a composition of minipatterns. A minipattern is realized by applying minitransformations. The operators given by Zimmer [10] are similar to the minitransformations, although Zimmer focuses on frameworks and relates patterns to each other.

Maplesden [7] introduces the DPML (*Design Pattern Markup Language*) for describing patterns using a metamodel. The DPML allows modelling the generalized solutions a pattern offers. It extends UML diagrams to better illustrate patterns with them. The DPML from Balanyi [12] is something different, although under the same acronym. It is an XML-based language introduced for describing patterns with the ability to recognize existent patterns later on. It allows the definition of classes, methods, declarations and dependencies between them. Not considered is semantic information.

The seven metapatterns from Pree [9] serve for illustrating design decisions in frameworks. Metapatterns were introduced for describing patterns in an abstract way. The main field of application are template and hook methods. Template methods represent so called frozen spots. These immutable spots reference hook methods. Hook methods represent hot spots, which are mutable spots inside the domain of a framework. With help of such, a user is led in adapting her system. As mutable spots are critical, metapatterns try to ease their adaptation. Pree [9, p. 170] adverts that metapatterns should only be applied to established frameworks.

FUJABA (*From UML to Java And Back Again*, Niere [14]) 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 (*Abstract Syntax Trees*) to record the (more technical) meaning of declarations and statements. Certain annotations, entitled in Niere [14] as *first-level annotations*, can be determined directly from the source code by analyzing ASTs. Examples for that are setter and getter methods (through speaking names) as well as the annotation *private attribute* used for a field declared with visibility *private*. Niere calls an annotation *second-level* when it is composed of first-level annotations. Such includes for a field the setter and getter method and the field declaration itself. Not considered is the high-level (non-technical) meaning of program elements.

4 Improving the support for patterns

This section presents an approach that aims at resolving the problems with defining and using design patterns as described in section 2. The solution targets at formalizing the pattern description, as this is a prerequisite for understanding the syntactic and semantic essentials of a pattern by tools. A formalized pattern description is called pattern template in this paper. The template could also be called *formal pattern definition*. It contains any information about a pattern necessary for a specific process (selection, application, detection).

To enable the understanding of a pattern besides the syntactic level, semantic elements are introduced to the pattern description to capture statements about intentions and meanings. Semantic information contains the solution parts of the pattern, i.e. the intention of the pattern parts and what their consequences are. In order to find an appropriate pattern for a given source code, a corresponding pattern template has to be identified. To allow this identification, the given source code has to be analyzed regarding its semantics (e.g. *why does method X in the code call method Y?*).

In the proposed solution, source code is enriched via annotations as behaviour-conserving program elements that assign a meaning to program elements. Then, the semantics defined in the pattern templates can be compared with the semantics given in the source code by annotations. If a link can be established for all solution parts of the pattern template with annotations in source code, the pattern seems perfect for the context.

4.1 Key elements and concepts

Throughout the paper, some terms for key elements are used quite often. Furthermore, this approach links together a pattern template with a source code by establishing a connection between them by corresponding elements. These elements are shown in the following table 1.

Pattern template	Correspondence in source code
Role	Class
Syntactic element	Declaration, statement
Slot	Variable part of declaration or statement
Semantic element	Annotation
Scope of semantic elem.	Scope of annotation
Constraint	Possible violation of preconditions

Table 1: Elements connecting pattern and source

Before the terms given above are described, a word to the different elements within pattern template and source code. It can be seen from the above table that within a pattern template and source code different elements exist that correspond. It is not reasonable using the same elements for both template and code. The reason is that with code the elements are already given by the language used (e.g. Java). Only comments can be filled with any content. In a pattern template it makes no sense utilizing ordinary statements (for providing information relevant for the pattern selection process) because isomorphic elements cannot be expressed that way. Isomorphic elements are statements or declarations within source code considered equal in the context of pattern selection. The case is similar with annotations and semantic elements. Annotations are used within source code. Thus, they must represent valid language constructs and must be behaviour-conserving on the other hand. In pattern templates, additional information is needed for such entities (e.g. the scope, described later on). Besides that, it would be an overhead using a syntax constrained by language conventions within a freely realizable pattern template.

The elements mentioned just before and displayed in table 1 are described in the remainder of this section. Before that, the key concept, semantics, is explained.

Semantics

Semantics is the study of sense and meaning. In a program text, syntactic elements (context-free meaning) and their contextual meaning given by program execution (sense) lead to semantics. A part of a program embodies semantic information that ideally equals the (a priori unknown) intention of the software developer. Meanings could possibly be figured out by syntax analysis, e.g. the meaning *object creation*. Syntax analysis often makes it not possible to recognize the reason for the creation of an object instance (that would mean understanding the sense) or under which premises (such as ensuring always having one instance) this happens.

From the author’s point of view, the semantic component of design patterns is currently not considered enough. Approaches considering annotations or comparable artefacts either express syntactic constraints only (such as in *Design By Contract*) or do not support the statement level. The given approach launches annotations with arbitrary scope as well as semantic elements to eliminate the gap. Annotations and semantic elements will be discussed next.

Annotation

An annotation is a property attached to a program element with the purpose of adding a semantic and machine-processable description to the attached element. A program element is a statement or a definition. Finally, an annotation assigns a sense to a program element. Annotations are seen necessary in this paper because the meaning of a piece of code is not reasonable in general by a tool. Even a human-being with no or little knowledge about a programming language may not be capable of understanding a piece of code. Additionally, internal semantics expressed by naming conventions, e.g., are not trustable, as such conventions are arbitrary, can be misused or can be ambiguous.

As an example, the JSR 175 [4] introduced annotations as first-class language constructs to Java and defined a syntax for them. An annotation allows a programmer to conduct statements about her source code and at the same time link these statements with the source code (through the above-mentioned scope/position of the annotation). Such statements could express current design decisions as well as the need for improvement or requirements (such as higher execution speed). For instance, an annotation attached to an object creation could tell "*introduce caching*". Then, a tool could offer the developer means (such as the *Proxy* pattern from Gamma [3]) to realize the object caching, and possibly apply the pattern itself.

The effective scope of an annotation is determined either by its specification that only allows for one scope; or by its position, which is preceding the referenced language construct. An annotation must reference the relevant aspect(s) of a language construct non-ambiguously, e.g. for an assignment consisting of a left and a right side (in the following code block the left side of the last statement is *MyCls o* and the right side is *new MyCls()*, both separated by an equals-sign). An annotation with these characteristics could be realized as a well-defined comment. Given a well-defined comment, it is possible to distinguish it from ordinary comments. Meffert [11] describes how annotations based on comments could look like. An example for such an annotation expressing the requirement that a speedup is required for the object creation:

```
public void myMethod() {
  ...
  @required: speedup for
             object creation
  MyCls o = new MyCls();
}
```

The speedup could be realized by caching, lazy instantiation, or object reuse, to name a few.

Currently, about 50 annotations have been identified by the author, distinguishing the 23 GoF patterns. They were found by first determining annotations for each single pattern and then adapting each annotation, if possible, to describe another of these patterns.

Annotations should be understandable by human-beings as well as by computers. Thus, it should be considered splitting an annotation in two parts: One part reflects the human-readable element, the second part a unique qualifier which is known to the tool evaluating annotations. This concept of composed annotations will have to be elaborated in the future. It is not part of this paper.

Semantic elements in pattern templates

Semantic elements are used in pattern templates. They are to a pattern template what an annotation is to a source code. They are capable of expressing in the same manner as annotations, but can have an arbitrary syntax (because they don't reside in code but in the freely designable pattern template).

For each semantic element, one or more annotation definitions exist. This allows linking together a pattern and a context given by a piece of code needing improvement to solve a design problem. The scope of an annotation is dependent on the corresponding semantic element, which defines the valid scopes for its dependent annotations. An example for multiple annotations per semantic element (given in the first line):

```
Reuse (cache) object instances
➢ @problem: object creation too slow
➢ @required: speedup for object
  creation
➢ @required: reuse object instances
```

A semantic element allocates an explicit meaning to exactly one syntactic element. Semantic elements represent a linguistic expression approximated to common speech, which contains a statement about the sense mentioned above in this section. They serve for specifying a sense of a pattern part. This is necessary, because for many cases it is possible only that way and with arguable effort to determine information about the sense of a pattern's pieces. Such statements contain semantic information for pattern parts, like their intention or their context (compare the example in section 6.1). With semantic elements it is, for instance, possible to specify for a method defined in a pattern template that this method has the intention to be called whenever another object's state is changed (as with *Observer*). The generally possible scopes result from the set of distinguishable types of language constructs. For Java, packages, classes, methods, constructors, field declarations, statements, and blocks can be distinguished. A semantic element always refers to the language construct it precedes. Matches between semantic elements and annotations can be verified by a prefabricated logic that also considers their scopes. In section 5, it is shown how semantic elements could be identified for a pattern from annotations.

Constraint

Often a pattern is only applicable under certain conditions (syntactic and semantic preconditions). E.g., *Singleton* is only applicable (without modification) to classes exclusively having constructors with empty parameter signature. *Iterator* is only applicable for loops running sequentially and completely over a set of elements and for which the loop variable is not modified within the loop. Preconditions of that kind could have any complexity. Either, they relate to a pattern as a whole or to a pattern role or to a part of a role. Ultimately, constraints ascertain whether a certain annotation exists in a piece of code or whether the AST equals such an annotation. The latter case can only be covered to a limited amount in general, as the intention of an AST is not reasonable in any case.

Preconditions for a given pattern (variant) can be seen as static. Because of their possible complexity, it is suggested to manifest constraints by program logic as methods with defined signature (similar to Ó Cinnéide [6]) instead of using script languages or expressions with a specific grammar.

Slot

Slots (compare Minsky's work in Haugeland [8]) are contextual parts in syntactic elements of a pattern that will be adapted to a given context during the application of the pattern. The source code the pattern should be applied to represents the context. Slots are used in the process of the pattern definition to make it adaptable to specific contexts. In the *getInstance()* method of *Singleton*, e.g., the return type (thus the logic within the method) that depends on the object type to be created could be expressed by a slot allowing its contextual replacement after the type is known by choosing the class for which to apply *Singleton*. After a value has been assigned to a slot, there is no difference between that slot and a constant.

Slots include arbitrarily configurable entities (user parameters) of static and dynamic parts of patterns, such as names of classes, methods or fields. Additionally included are references to objects of the context within statements and declarations that should be added by a pattern and that should reference those objects (which could be marked by annotations).

Pattern template

A pattern template represents a definition of a specific pattern. This definition contains information necessary to select or recognize the pattern for a given piece of code or apply it to the code. It is possible to consider one or two, but not all three of the pattern processes, namely selection, application and recognition. Therefore, a template contains syntactic as well as semantic information, both connected with each other.

Static parts of a pattern, such as interfaces, are defined by providing their implementation along with semantic information as an anchor to identify their meaning. As names within static parts may be interchangeable, slots are used in order to adapt them. The dynamic parts of a pattern are partly modelled with help of slots. As the application of dynamic parts is a very complex issue, their definition can become very complex, too. Thus, a second means of defining dynamic parts is pure program logic that could rely on a framework designed to help implementing such logic.

Role

A pattern is composed of multiple pieces (or parts). Le Guennec [13] and Smith [15] define coarse-grained pieces as roles. E.g., the pattern *Observer* consists of the roles *Subject*, *Observer*, *ConcreteSubject*, *ConcreteObserver* and *Client*.

A class in the source code corresponds to one or more roles in the pattern definition. The partitioning of a pattern into roles is oriented on the definitions in Gamma [3]. A full description of all roles of a pattern is indispensable for the application and the recognition of it. For the selection of a pattern, several roles (such as interfaces) can be skipped (compare the example in section 6.1).

Syntactic elements in pattern templates

Syntactic elements represent valid program elements and reflect concrete implementation details. They are not specific to design patterns. Syntactic elements can be used to express the implementation of a pattern's parts. Multiple sequential syntactic elements could be grouped logically to allow for macro definitions. Implementation variants (expressed by syntactic elements and slots) generically defined within pattern templates are used for the definition of scopes on which the selection and the application of patterns is bound to. All variants according to one generic variant are isomorphic (i.e. considered equal in a certain context). E.g., for the applicability of *Observer* it does not make any difference whether a method named *notify* declares an exception or not. The generic definition *object creation* is another example. It would contain all language constructs that create and return an object instance.

These language constructs could be defined with help of syntactic elements and annotations. In Niere [14] such generic definition of artefacts is called implementation variant, in Smith [15] it is named isotope. The definition of such artefacts has to happen incrementally and evolutionary because of the endless combinations possible. A new type will be defined, when it is needed (thus incrementally). The approach should be evolutionary because after adding a type it could be necessary to condense or modify types. Section 5 shows how to identify syntactic elements of a pattern.

4.2 Vision and process overview

This paper reflects an approach that is still under research. Thus, its vision is sketched here to show the roadmap for the potential the approach may have.

The vision of this paper is to set tools into the position of selecting design and architectural patterns (e.g. from Buschmann [5]) automatically by evaluating pattern templates containing annotations describing their solution parts. Pattern templates may be defined by senior developers, or a pattern community. Then, an application developer is able to choose a pattern and apply it onto her source code project using a tool.

To target this vision, the idea is to inject semantic information into pattern templates as well as into source code. This eases all following processes as that semantic information could otherwise not be obtained easily or not at all (compare Montero [16]). This work focuses on the definition and selection of patterns, the application, and detection of them cannot be examined in detail due to their complexity.

As design patterns are handier than architectural patterns and the former ones are easier to follow and to illustrate, this paper skips architectural patterns intentionally and concentrates in design patterns. However, the main usage of the approach may be in the field of architecture, because there supporting the making of good decisions has more an impact on a system's quality than micro-architectural issues have.

The assumed development process for using the approach refers to source code as main entity. Therefore, the approach is applicable for coding during the forward engineering as well as for restructuring code during reengineering activities. Working directly with source code is a common practice in commercial software development both for approaches not being model-driven and for such relying on a modelling tool with subsequent editing of the source code.

In the next figure, an overview of the core processes involved with the described approach is given.

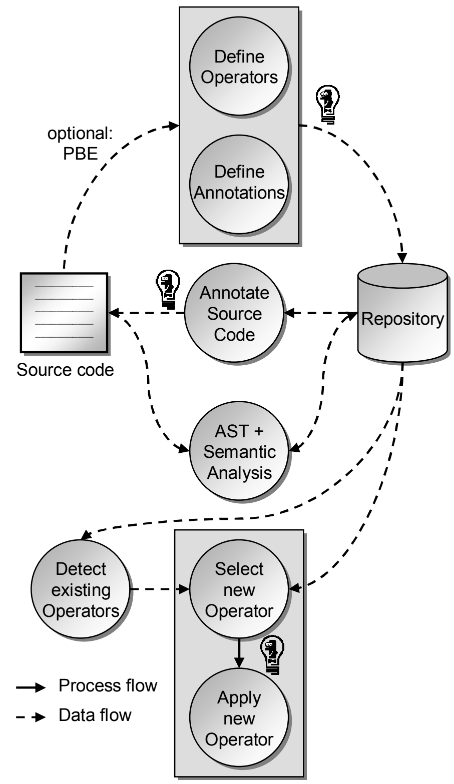

Figure 1: Core processes of the approach

At the beginning is the definition of pattern templates, or more generally, of operators (an operator transforms a piece of code; it could be a design pattern, an architectural pattern, a refactoring operation, or something similar). The operator definition can happen from scratch. Or it could happen by transforming a piece of code manually by subsequently applying a reasonable pattern part by part and gaining operators from that transformation (named *PBE – Programming By Example* after Lieberman [2] – in the above figure). Annotations have to be defined as well, in any case by hand. The approach to do that is analogous to the one for defining operators. The definitions are stored in a repository to have access to them later on. Annotating source code can then rely on that repository. The annotating process adds formal descriptions of design intentions, drawbacks, and developer's wishes to the code. After that, operators (such as design patterns, and even architectural patterns) can be selected for the annotated code. To avoid selecting already existing patterns, the code to apply a pattern on should be scanned for such patterns.

Before going into detail of the presented approach, the next section gives an overview of important terms used throughout this paper.

4.3 Connecting pattern template and source code

In order to select a pattern for a given piece of code, the pattern template and the code have to be matched. A matching can occur on the syntactic and on the semantic level. The syntactic level includes statements and declarations. The semantic part contains the concepts *annotation* and *semantic element* described in the previous section. It expresses a contextual meaning (sense, intention). Following Alexander [1], a pattern is "(…) a three-part rule, which expresses a relation between a certain context, a problem, and a solution." In this work, possibilities will be shown for describing the context (in source code and in pattern templates), the problem (in source code) and the solution (in the form of a pattern template). The context is the link between pattern template and source code for which a pattern is applicable, present in both entities.

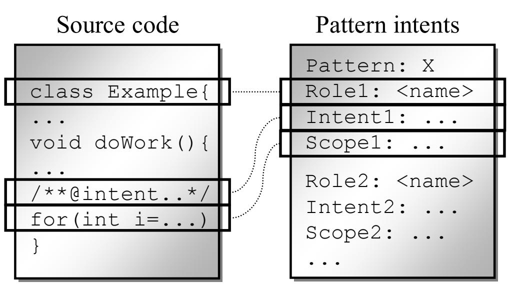

Figure 2: Connecting code with pattern template

Figure 2 illustrates the link between source code and pattern template with help of annotations, semantic and syntactic elements. Just the syntactic elements are existent already in source code and can be examined via AST analysis.

The next section describes the elements the approach introduces, and the following section discusses the possibilities of linking code and pattern template.

5 Determining elements for a pattern

This section describes how a senior developer could obtain the elements needed to describe a pattern template. As this paper focuses on the selection of a pattern, the approach described here will lead to a template containing information for the pattern selection mainly. Obtaining information for the application of a pattern will only be sketched.

The main idea is transforming a non-optimal source code for which a pattern is applicable, by applying that pattern gradually. During this process, syntactic transformations are undertaken as well as semantic information is added. Transformations are comparable to the minitransformations from Ó Cinnéide [6] and can be regarded as part of a refactoring process. During this transformation, the information gained can be used to determine the annotations (and thus semantic elements) and the scope of annotations (via isomorphic elements) needed to describe the pattern.

A by-product is an idea about the transformations necessary to apply the pattern onto an unknown source code. The procedure of doing a transformation and recording (in mind or by a program) the steps undertaken is similar to what Lieberman [2] calls *Programming By Demonstration*. Wherever a transformation of source code is necessary, an annotation should be considered to be added as an anchor for a later algorithm to select patterns with. The procedure for determining a pattern's elements contains the following steps:

1. Identify the transformations necessary to apply the pattern.
2. For each transformation identified, add an annotation describing the characteristics of the transformed piece of code.
3. For each annotated class, method or statement, also regard referenced elements of the same class. If they are of significance for the pattern, annotate them.
4. For each annotation introduced determine a semantic element corresponding to it (either by defining a new one or by reusing one from a repository).

The transformations mentioned lead to syntactic elements constituting implementation details important for the application of a pattern. A simple example for finding annotations and corresponding scopes is the *Singleton* pattern. For any other creational pattern this holds true and only becomes more complicated when it comes to constraints and transformations. The procedure described above has been tried with most of the creational patterns from GoF (including the more complex *Flyweight*), with *Composite*, *Template Method*, *Strategy*, *Iterator*, *Observer*, *Proxy*, and *Adapter*.

To give an example: A source code suitable for applying *Singleton* on is such where a statement exists creating an instance of an object. Two cases could be distinguished: Firstly, the obvious way of constructing an object by calling the constructor of the object's class. Secondly, the indirect way where a method is called doing this. The former case leads to an annotation's scope such as "*all calls to constructors*". Meaning, all constructor calls would be regarded as isomorphic. The latter case would lead to a scope such as "*any call to a method*" with the additional constraint that the method is annotated accordingly. The mentioned annotation will have to express that the intention of the method is the creation of an object. With help of this annotation it is no problem even recognizing methods that use unobvious means for object creation, such as by reflection in Java. The text the annotation contains is gained by thinking about the intention to be fulfilled by introducing the *Singleton*. For that pattern, the annotation could write "*ensure always one instance present*".

Processing the transformation as above for *Observer* leads to similar results. As an observable calls a method of any eligible observing class, the case is similar to *Singleton*. The annotation found would be different of course (here maybe "*notify on state- change*" as annotation for the method to be called). *Iterator* is more complex regarding the possible scope of an annotation. As the scope is determined by identifying isomorphic elements, any sequence of statements must be seen as isomorphic that loops over a list sequentially, covers all list elements and does not modify an element during the loop. To give an example, the transformation for *Iterator* could start from a source code such as

```
10 List l = …
20 ... // any code
30 for(int i=0; i<l.size(); i++) {
40 Object obj = l.get(i);
50 ... // process "obj"
60 }
```

An annotation could be introduced in line 25:

```
25 /**@simplify: loop */
```

Here, the annotation is an ordinary comment beginning with an at-sign and a keyword to distinguish it from other comments. The result of the transformation could be:

```
A List l = …
B ... // any code
C Iterator it = l.iterator();
D while (it.hasNext()) {
E Object obj = it.next();
F ... // process "obj"
G }
```

As can be seen line 30 has been replaced with lines C and D, and line 40 with E. Thus, the scope for an annotation would result from line 30, a constraint would consider lines 30 to 60 (checking that the whole list is looped over and no modification to the list is made). After the code transformation a semantic element must be considered reflecting the annotation given with line 25. In this case the expressed sense of the semantic element could write as the annotation, or it could write "*abstract a loop*", e.g. the type *Iterator* (line C) is newly introduced to the example code. It could be described in the form of an interface in the pattern template. See section 6.1, role *IComponent* for a concrete definition in an analogous way. In case the class representing variable *l* (line 10) does not implement the method named *iterator* with return type *Iterator*, this also has to be introduced via transformation. The implementation aspect of this part is not important for the selection of the pattern (but for its application and detection), thus it will not be considered here further.

To be able to put the found syntactic and semantic elements of a pattern in the appropriate place in the pattern template, the pattern's roles must be declared. This can happen by looking at an applied pattern (or the UML diagram of it) and then defining the roles of the pattern, such as Gamma [3] did. For *Iterator* the roles would be *Aggregate*, *ConcreteAggregate*, *Iterator* and *ConcreteIterator*.

For method *iterator* in role *Iterator* a semantic element will be provided with a sense as given above (e.g. "*abstract a loop*"). The role *ConcreteIterator* contains implementation details not important for the pattern selection, the same holds true for *ConcreteAggregate*. Role *Aggregate* would also contain a method *iterator* with return type *Iterator.* A semantic element provided for that method indicates the possibility of obtaining an *Iterator* object, thus allowing to conclude that this method (with dependent logic) does not have to be added in a transformation undertaken when applying a pattern). For *Composite* as another example the step 3 of the algorithm described above in this paragraph leads to annotating the field in the parent class holding the child elements (e.g. in a list) to be processed sequentially. In case that for another source code there would not be such a field holding child elements, but – as a solution suboptimal with respect to good coding practice – maybe some variables (for each child one), then a pattern selection algorithm would possibly find AST characteristics plus all but one annotation necessary to identify *Composite* clearly. As a result, the selection algorithm could report a *near miss* forcing the developer to think about the current implementation.

6 Usage of the introduced elements

Section 4 introduced elements to define patterns and to enrich source code with semantic information. The previous section showed how to obtain these elements for a given pattern. In this section the usage of those elements is demonstrated.

Table 2 presents an overview of the processes for design patterns supported by the introduced elements. A plus sign indicates the eligibility of an element to support a process, a minus sign means the opposite, and an *o* characterizes a partial eligibility. The next paragraph demonstrates the usage of these elements for defining a pattern template. The dependent processes will be discussed after that. At this, the paper's focus is on the selection of patterns. The next section begins with the prerequisite of pattern selection, namely the pattern definition.

Process → Element ↓	Definition	Selection	Application	Detection	Understanding
Syntactic element	+	+	+	+	o
Annotation	-	+	+	+	+
Scope of annotation	+	+	-	+	o
Semantic element	+	+	+	+	+
Constraint	+	+	+	+	o

Table 2: Patterns – processes and proposed elements

6.1 Definition of a pattern template

This paragraph exhibits the definition of a pattern template with help of the discussed elements. The pattern template discussed here is intended to serve as input for tools, and it does not get into contact with a developer anymore, after it has been defined once.

The form of a pattern template depends on the field of application. Three forms are to be considered, namely for the processes

a) selection of an applicable pattern,
b) application of a selected pattern, and
c) recognition of an already applied pattern.

It should be pointed out that for each variant of a pattern, a different template must be defined. In this context, variants are considered different if their semantics are different or their syntactic elements cannot be expressed by isomorphs.

Exemplary the pattern template for one possible variation of *Composite* is given, divided into the pattern's roles. The example shows the form a) and, in outlines, the form b) for static parts. The roles *IComposite* and *ILeaf* as well as *Composite* and *Leaf* will be skipped because of their similarity to *IComponent* resp. *Component*. It follows role *IComponent*:

```
application [
 source [
  public interface <IComponent> {
   void operation();
   void add(<IComponent> component);
   <IComponent> getChild(int i);
  }
 ]
]
```

Figure 3: Role *IComponent* of pattern *Composite*

Role *IComponent* is only relevant for the application of the pattern. The following role, *Component*, also contains elements regarding the selection of the pattern:

```
// Definition of list holding children
selection [
 semantics [
  sense: hold child elements
  scope: any_collection
  constraint: contains_classes(
    {<IComponent>,<ILeaf>})
 ]
]
// Method addChild
selection [
 semantics [
  sense: add child elements
  scope: nonprivate_method_with_param
  constraint: added_element_valid_for
              (name_of_field(@1))
 ]
]
// Method operation
selection [
 semantics [
 sense: execute operation on children
 scope: any_nonprivate_method
 constraint: call_to_classes(
             {<IComponent>;<ILeaf>})
 ]
]
```

Figure 4: Role *Component* of pattern *Composite*

The pattern template consists of the blocks

- *selection* (for the selection of applicable patterns),
- *application* (for the application of a pattern, only outlined here), and the sub block
- *semantics* (for the specification of semantic information).

For the selection of patterns, especially the semantic elements given by the *selection* blocks are helpful. For the application of patterns, implementation details (syntactic elements) as well as transformations will be needed additionally. Those additions will be defined in *application* blocks. For the recognition of patterns, the *semantics* blocks inside *selection* blocks are obsolete. Instead of them, *semantics* blocks inside *application* blocks would be used. This is because only with their help the intention of a piece of code to be applied can be identified correctly. Names to be filled in dependent on the context are marked by slots. An identifier surrounded with angle brackets, like <*IComponent*>, labels a slot. Semantic elements as an important part of a pattern template are specified by a sense, a scope and a constraint. These triplets are relevant for all forms of pattern templates (above mentioned by a) to c)). The sense represents a natural language-like expression being a "normalized" expression for one to n annotations. Constraints and scopes are given by a method name, because a script-like declaration of them seems not realizable offhand due to their complexity, lack of readability and deficient support by IDEs. A constraint has access to any information acquired before. This includes information from blocks processed before and information for the AST element belonging to the scope. In figure 4 the value of the first parameter of the method belonging to the scope defined by "@1". The constraint following that one receives as parameters a list of roles determined by slot values ("*{<IComponent>;<ILeaf>}*").

Dependencies between single roles that should be known for the selection of patterns are expressed in the pattern template by semantic elements and constraints connected with them. For instance, it is sufficient for a method call either to annotate the caller or to annotate the callee, because a syntactic link between them already exists.

The *application* block could contain a source block including implementation details of an exemplary solution for parts of a pattern. The information given by the selection and application blocks could be reused for the pattern recognition.

6.2 Selection of a pattern

It could be decided about the applicability (and thus selection) of a pattern for a given, annotated source code by comparing the source code with elements defined in the template of the pattern (Meffert [11] illustrates this in more detail). Source code annotations together with associated syntactic elements as well as non-annotated syntactic elements (accessible to an AST analysis) serve as indicators. The indicators matched between pattern template and source code could be displayed to the developer selecting from a list of that way proposed pattern to support her decision.

Before the selection of an applicable pattern, there are different statements possible in a source code – dependent on the perspective of the developer – such as:

- Intention: "*increase speed of execution*".
- Wish/Requirement: "*higher execution speed needed*".
- Drawback/Problem: "*execution too slow*" or "*application does not run efficiently enough*".
- Pattern to be applied (directly given): "*Iterator*".

To manifest such statements within source code, annotations are offered. Annotations could be added by the developer during programming or afterwards to an existing piece of code. The developer does not have to insert annotations when the meaning or need for some piece of code is obvious (compare *Iterator*). Annotations for intentions not obvious must be added manually as long as they are relevant for a pattern (e.g., the wish for higher execution speed). An annotation could contain semantic information reasonable from a piece of code, as well as such that are not reasonable or are likely to be reasoned only by an experienced developer. Likewise, annotations allow for expressing design decisions that otherwise (then often incompletely) could possibly be concluded by runtime analysis (e.g. dynamic call of methods). This paper suggests providing annotations even for semantic statements that are identifiable by design time or runtime analysis. Thus, annotations would be the uniform basis for further source code analyses.

In the pattern selection process, annotations will be compared with semantic elements, associated syntactic elements with scopes defined in the pattern template; syntactic elements alone will be verified by constraints. The applicability of a pattern will be determined by the degree of correspondence between pattern template elements and found (or not found) elements in the source. The developer receives a report with possible patterns, sorted by the mentioned degree. A distinct weighting of single elements, that has to be found evolutionary, influences the degree of correspondence. This degree would be decreased, if an annotation corresponds with a semantic element, but its associated syntactic element does not obey the scope, to give an example. Furthermore, it is possible that for some semantic elements no corresponding annotations could be found. A weighting of the misses supports the output of a report of pattern suggestions sorted by hit rate.

6.3 Application of a pattern

The information gained during the selection of a pattern recognized as applicable could be reused for the process of applying that pattern. Particularly annotations deliver necessary information for the application of patterns; they serve for concluding about design decisions as well as for linking them with (Java) statements. The existing correspondence between elements of a pattern template and annotated source code makes it possible to apply the syntactic elements and transformations defined in the template exactly onto the source code (compare figure 2). New code such as interfaces simply is added by creating new classes. New code within classes is added by finding the right context with code annotations and semantic elements from the pattern template and then adding the code there.

Transformations should consider isomorphic AST elements. The precursor mentioned in Ó Cinnéide [6] as a precondition for the feasibility of a minitransformation is defined by the constraints and semantic elements mentioned here. To ensure a correct transformation, Ó Cinnéide asks for specifying postconditions. It should be added, that newly created unit tests are convenient to ensure the correctness of a transformation to a high degree.

6.4 Detection of existing patterns

For the detection of existing patterns within a piece of code, especially the semantic elements are of relevance. It is essential for any semantic element of a pattern to find a correspondence in the source code. Eventually this causes the existence of annotations in the code. To make such possible it is proposed to let AST scanners add their results as annotations to the analyzed source code. Now, the recognition of an existing pattern is comparable with the selection of a pattern.

With information about design decisions or intentions for a code fragment missing, it may become difficult detecting patterns with sufficient determination.

6.5 Understanding of a pattern

Understanding a pattern clearly is very important to remove the chance of selecting a wrong pattern or applying a pattern wrongly. Understanding the sense and the meaning of a pattern relies on several aspects. At first, the understanding of the pattern template (sort of scaffolding of the pattern) is important. An UML diagram displaying the general structure of a pattern could help to some extent. The understanding can be enhanced by presenting the developer the semantic information given in the pattern template. An UML diagram could be enriched with such semantic information easily as each piece of semantic information concerns one class or the relation between two classes.

Secondly, an applied pattern intensifies the understanding about it, as it represents a use case. Finally an annotated source code along with selected (because applicable) patterns for it, demonstrates the circumstances under which a pattern could be applied appropriately.

7 Case study

This section demonstrates the application of the presented approach by showing a practical example with *Composite*. In this example, the pattern template from the previous section is referred to in order to show how to relate it with a piece of code by adding annotations to the latter. The existence of necessary annotations is assumed, as their introduction would require quite lengthy descriptions.

Several possible initial situations are imaginable that justify the application of *Composite*. Some of them are quite close to the target structure as proposed by the pattern. Others are far away from it and difficult to be transformed to *Composite*'s structure. As an example, the entities *Graphics*, *Picture*, *Line* and *Text* have been used. Each one is represented by a single class. *Graphics* is a top-level element potentially containing the other three mentioned. *Picture* is an element that could contain another *Picture* as well as *Line* and *Text*. The latter two are atomic. The operation known from *Composite* is painting these entities resulting in a graphical element.

Three examples have been examined. In the first one, the four entities have no super class or interface. This is different from *Composite*. The composition of *Graphics* and *Picture* is similar to the pattern as an *add*-method is used. The operation is related moreover because it iterates over all sub components, although during that loop a case distinction should be made concerning the absence of a *Component* interface (as known from the *Composite* pattern). The four entities in the second example have a common base class allowing to realize the operation similar to *Composite*. As a difference, the composition of the graphical element is done via a sequential list. The list is defined in a client class. Each element in the list points onto its parent. After sorting the list so that elements having fewer parents more are at the beginning of the list than elements having more parents, the operation (namely painting the picture) could be executed. Each next element of the list is painted, followed by its children. Then these painted elements are removed from the list and the next element, if any, is processed. This algorithm is not optimal and chosen intentionally. In the third example the pattern is selected proactively, meaning there are empty classes for the four graphical elements but no further implementations. Now an annotation could be attached to the top of the class *Graphics*. The annotation could express "*contains children*" to get an indicator leading to *Composite*. A definition mapping a sense defined in section 6.1 allows identifying possible patterns.

For the first two examples, the annotation of the four graphical elements denoting their roles in *Composite* is equal. It should be noted that marking a class as a specific role already requires an understanding of the pattern. So, a tool should guide the developer in selecting such an annotation from a limited set of annotations. The set is narrowed down as other annotations, already existent in source code, are considered. E.g., for the first example, the *add*-method could be annotated with "*function add child*".

For the second example, the sequential list could be annotated by "*contains parent-child relationship*". These annotations would correspond to the sense elements contained in the role definitions in section 6.1. Then the system could search through the pattern templates for any pattern that allows for adding child elements and present the differences to the developer. After excluding certain patterns, the system presents annotations that are defined in the remaining pattern templates and in turn are missing in the source code. By that, the developer is guided to annotating further program elements. At some point, the developer may decide to apply a specific pattern. This can be supported by the information gained via annotations in source code.

8 Conclusion and future work

The definition of pattern templates is a prerequisite for all further processes connected with design patterns. In this paper elements were introduced suitable for describing patterns in a machine processable way. It could be considered to let senior developers (from the company, the pattern community, or a tool vendor) define pattern templates, and not the application developer. It has been taken care that a link between pattern template and source code could be established with help of corresponding elements. To express semantics not superficially recognizable in pattern template or source code, semantic elements as well as annotations were introduced. Annotations add documentation to the annotated source code. With annotations it is possible determining the meaning of an annotated program element non-ambiguously. For instance it is possible marking the roles a class plays with annotations. This may especially be helpful in case the class in involved in more than one pattern.

In the author's opinion, certain elements of a pattern are not definable easily in a declarative way, including conditions and transformations. They should be expressed by specialized methods superiorly. Either way, it is an additional effort bringing in explicit semantic information (no matter which construct is chosen). This effort is necessary whenever the sense of a piece of code is not reasonable automatically or only with considerable difficulties. A benefit from explicit semantics is additional documentation of the code.

A prototype exists that parses a pattern template, recognizes according annotations with generic scopes, and handles isomorphic constraints to identify some applicable patterns (*Iterator*, *Composite*, *Observer*, some creationals). Here, further investigations are necessary, especially the usage of annotations in legacy projects to verify the general practicability of the approach. Another future issue is the definition of a set of annotations being as compact and as powerful as possible, e.g. for describing all 23 patterns from Gamma [3]. The currently proposed form for a pattern template has the potential to be optimized further to make it easier to maintain and to give it a look more appealing to the person editing it.

The author believes that considering explicit semantic information simplifies approaches for the selection, application, and recognition of patterns significantly, because the lacking formality of the Alexandrian form (compare Alexander [1], Gamma [3], and Buschmann [5]) makes interpretation by tools very difficult.

A consequence with formalizing patterns is that their flexibility may be reduced. For example, existing variants of a pattern have to be defined separately, as if each variant was a pattern of its own. As it may not be possible considering any valid variant of a pattern, the pattern's solution space may be reduced.

Acknowledgment

My appreciation goes to Susan Kurian for reviewing the paper very helpfully before submitting the first version, and to Tim Wellhausen for his constructive on-site feedback at the conference. Both led me to a much improved version of the paper.

References

[1] Alexander, C.: The Timeless Way of Building. Oxford University Press, 1979.
[2] Lieberman, Henry: Your Wish Is My Command: Programming By Example. Morgan Kaufman, 2001.
[3] Gamma, E.; Helm, R.; Johnson R.; Vlissides, J.: Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, 1995.
[4] Java Community Process 175 (Annotations for Java): http://www.jcp.org/en/jsr/detail?id=175.
[5] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stahl, M.: Pattern Oriented Software Architecture. Wiley, 1996.
[6] Ó Cinnéide, M.; Nixon, P.: A Methodology for the Automated Introduction of Design Patterns. Proceedings of the International Conference on Software Maintenance, Oxford, Sept. 1999.
[7] Maplesden, D.; Hosking, J.G.; Grundy, J.C.: Design Pattern Modelling and Instantiation using DPML. In Proceedings of Tools Pacific 2002, Sydney, 18-21 February, 2002, CRPIT Press.
[8] Haugeland, J. (ed.): Mind Design II – Philosophy, Psychology, Artificial Intelligence. MIT Press, Cambridge, London, 2nd edition (1997).
[9] Pree, W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.
[10] Zimmer, W.: Frameworks und Entwurfsmuster. Shaker Verlag, 1997.
[11] Meffert, K.: Supporting Design Patterns with Annotations. *ecbs*, pp. 437-445, 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06), 2006.

[12] Balanyi, Z.; Ferenc, R.: Mining Design From C++ Source Code. Proceedings of the International Conference on Software Maintenance, 2003.

[13] Le Guennec, A.; Sunyé, G.; Jézéquel, J.-M.: Precise Modeling of Design Patterns. In Proceedings of UML 2000, pp. 482-496, 2000.

[14] Niere, J.; Schäfer, W.; Wadsack, J. P.; Wendehals, L.; Welsh, J.: Towards Pattern-Based Design Recovery. Proceedings of the 22nd International Conference on Software Engineering 2000, Limerick, Ireland, pp. 241-251, ACM Press, June 2000.

[15] Smith, J.M.; Stotts, D.: Flexible Automated Design Pattern Extraction from Source Code. Proceedings of the 2003 IEEE International Conference on Automated Software Engineering, Montreal QC, Canada, October 2003, pp. 215-224.

[16] Montero, S.; Díaz, P.; Ignacio, A.: A Semantic Representation for Domain-Specific Patterns, Lecture Notes in Computer Science, Volume 3511, Jul 2005, Pages 129 - 140, Springer, 2005.