Design Pattern Recovery Based on Source Code Analysis with Fuzzy Logic

Jörg Niere, Jörg P. Wadsack, Lothar Wendehals
Department of Mathematics and Computer Science
University of Paderborn
Warburger Straße 100
D-33098 Paderborn
Germany
[nierej, maroc, lowende]@uni-paderborn.de

ABSTRACT

Program comprehension is one of the most difficult parts in computer science. At the latest, the turn of the year 1999 has shown that program comprehension and tool supported reverse engineering attained more and more attention. Design patterns, especially the famous gamma-patterns [GHJV95], support the forward-engineering process of software development. Since the gamma-patterns are an analysis product of various software systems, they are a result of a reverse engineering process. Our idea is to recover design patterns from source code to support a comprehension process. Therefore, we introduce a flexible definition of patterns consisting sub-patterns. We allow that patterns inherit from other patterns and we use the polymorphism concept to at least reduce the number of pattern definitions. We propose to use inheritance for design variants and fuzzy logic for implementation variants of a pattern. We discuss advantages and disadvantages of the two definition possibilities and present an execution mechanism to recover patterns in source code.

Keywords

design pattern recovery, UML, graph grammars, fuzzy logic, semi automatic process

1 INTRODUCTION

Computer science has survived the largest software update problem since its birth, namely the year two thousand (y2k) problem. Several billion lines of code were investigated in order to ensure the correct functionality of the systems. But even experts were afraid of the turn of the year 1999 because they did not know exactly if their solutions work in all cases. However, the y2k problem was mainly a product of a casual chain of human made decisions over decades of developing and enhancing a system. Decisions made during the development depend on various reasons. For example limited system resources, or a contract with a hard-deadline, or there are currently no developers available. In addition, developers have certain personal programming styles, e.g. using inexpressive variable names or spreading an algorithm into several pieces, which makes the comprehension of the software for other developers difficult.

Especially software engineering principles like documentation, before hand design, and derived tests take a back seat vis-á-vis to be the first on the market or to produce solutions, quickly. Enhancement of those systems gains to the most difficult parts of software development particularly if the core developers left the company. In those cases first, reverse engineering techniques are applied to get an abstract representation and overview of the system. In most cases the reverse engineering is done by hand, because the enhancement comprises of only some modifications, because a complete reverse engineering task is too expensive. This leads eventually to patchworks or hot-fixes and is more an aggravation than an improvement.

To overcome these problems computer science has developed various approaches of reverse engineering processes [Jah99, Wil94] and automated tools to support those processes, e.g. clustering techniques. Typically, the techniques stop at a certain degree of granularity where the reengineer has to perform the more detailed analysis by hand, cf. [CFM93]. Other approaches try to identify the behaviour of certain program parts, but fail for large systems. In general, this results from the high number of syntactical different but semantically equal implementations.

In our previous work we have presented reverse-engineering approaches for the analysis of java card applications [JNW00] and to support round-trip engineering [NWZ01]. Both approaches are very similar and we spotted, that they fits for other purposes too, since both approaches are very technical and domain specific, in this paper we present a generalized technique. We introduce a common notation, and a new target-driven inference mechanism. We will present the technique on the example of recovering design patterns [GHJV95] from Java source code[1].

The rest of the paper is structured as follows: In section 2 we give a short introduction to design patterns, especially their definition for reverse engineering needs using the UML. Section 3 describes our approach to recover design variants

1. Our approach is not limited to Java, other object-oriented languages are also imaginable, e.g. C++ or Eiffel.

of certain design patterns. We use fuzzyness for the recovery of implementation variants, which is introduced in section 4. The execution mechanism of our pattern recognition system and the interaction with the reengineer is described in the following section 5. Subsequently, we discuss related work and close the paper with some conclusions and future work.

2 RECOVERING DESIGN PATTERN

The design patterns presented in [GHJV95] are an analysis result of diverse software systems developed in the laboratories of Big Blue. In the following, we call those patterns gamma-patterns. However, the intend of the gamma-patterns is to provide a collection of ‘good’ design principles and discuss their advantages and disadvantages as well as their relation to other patterns. For each pattern a sample implementation in the C++ language is given.

Although gamma-patterns are a product of a reverse-engineering process, they are applied in forward-engineering tasks. For example to develop new software or for its documentation. To support such forward engineering, programming languages have been extended by new language constructs [CILS93, BFLS99, KS98]. For example [BFLS99] extends Eiffel [Mey88] with a new ‘pattern’ language construct and describes its compilation.

A future trend is to provide so-called pattern-languages, where each language is a collection of patterns like the gamma-patterns for a certain domain or application problem, e.g. concurrency control [CKV96]. Since each language deals with a certain problem in a specific domain, the discussion of advantages and disadvantages is mainly problem driven and particularly based on a special application.

Design patterns are also used to document certain parts of a design. Many techniques have been developed to automatically derive a documentation out of a given design (forward engineering). Other approaches analyse the source code and try to get an automatic generated documentation out of it (reverse Engineering), cf. [SS00]. Those approaches mainly focus on method-bodies and parts of method-bodies in order to document them. In this case design patterns are very helpful, because they comprise information about classes, methods, attributes, and their relation among each other. So, our aim is to recover design patterns and annotate all design parts, which are comprised by the pattern. Consequently, the annotated design can then be used for various purposes, e.g. automatic documentation or redesign.

The Unified Modelling Language (UML) [UML] provides a pattern construct, but it has no defined semantics. It allows only to annotate a current design, mainly class diagrams for documentation and readability reasons. For example, one of the famous gamma-patterns is the composite pattern. Figure 1 shows the structural part of the composite pattern. A gamma-pattern description consists of 12 additional parts, e.g. the pattern’s name, its intention, the structure, the behaviour, etc. The composite pattern in figure 1 is already adapted and contains more information as the original structure, i.e. the dashed class Leaf and its inheritance relation[1]. This means, that for a composite pattern an explicit leaf class is not needed. We took this information out of the collaboration description of the leaf: “If the recipient is a Leaf, then the request is handled directly.”. This does not force tan explicit Leaf class inheriting from Component, but could also be handled by a subclass of Composite. For example, such a design is practised in the Java Swing API [ELW98, HC98a, HC98b].

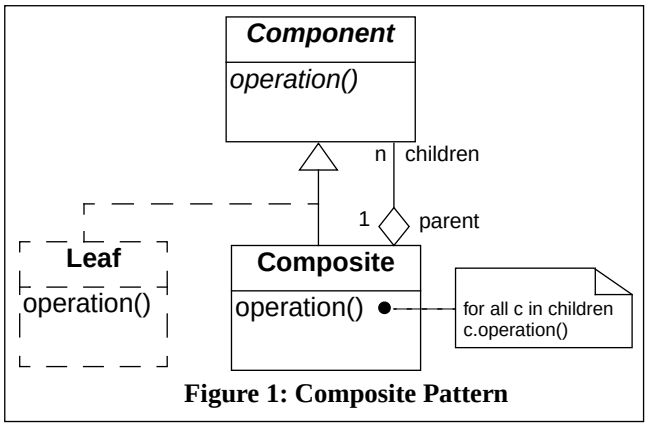

Figure 1: Composite Pattern

Applying design-patterns (gamma-patterns) for reserve-engineering leads to several problems, which have to be solved. First, design patterns have to be formal defined, in order to automate their detection, because the informal described parts of the gamma-patterns are not feasible for tool supported (semi-)automatic recovery. Second, most implementation variants of a pattern have to be covered by the definition. And third, the execution mechanism must be efficient in an appropriate time scale.

In addition to implementation variants of one design pattern, there also exist different design variants of one pattern, e.g. figure 5 shows two design variants of the composite pattern. Considering legacy systems, those variants become especially important, because they contain typically no pure gamma-pattern but use some derivations.

Our Approach

Our recovering technique for design patterns is based on the annotation of an abstract syntax graph (ASG). Therefore we have to parse the source code. Whereas, our abstract syntax graph model is more or less a simplification of the original UML meta model [UML] for performance reasons. In case of Java source code we use the JavaCC (Java Compiler Compiler) [JCC] to generate a parser out of the given grammar, where the back-end of the parser creates directly an ASG.

Basically, the parser generates a rudimentary UML class diagram containing classes with attributes and method declarations as well as inheritances between the classes. Note, so far the class diagram does not contain any associations or other dependencies, cf. [NNWZ00]. Method bodies itself are parsed as activity diagram [KNNZ99a]. For more details about our integrated meta model see [KNNZ00].

1. We took the notation from the graph grammar field, where optional nodes and edges are specify by dashed lines [Zün96].

As mentioned before, our approach bases on the annotation of the syntax graph. Our technique is similar to language and graph parsing techniques, cf. [RS95]. However, the annotated parts (nodes in the syntax graph) are not consumed. Thus, nodes may be members of various annotations (patterns).

Pattern definition

The pattern definition in [GHJV95] consists of more informal parts than formal ones. The most formal one is the structure part of a pattern's definition, because it is typically a class diagram. But, for example, the intention of the pattern is part of its definition, written in prose. And even the collaboration part is mostly informal. But for tool supported reverse-engineering, a formal definition is indispensable. For example, Krämer and Prechelt show in [KP96], that it is not sufficient to analyse the structure only, because this leads to minor positive results only and to many false-positives.

Therefore, in a first step we have specified most of the gamma-patterns using the Fujaba[1] environment [FNTZ98, KNNZ99b]. Fujaba supports our integrated abstract syntax model and allows to generate executable Java code out of a given specification. In case of the gamma patterns, we used class diagrams for the specification of the structure. For the behaviour we used story-diagrams, namely a combination of activity-diagrams and collaboration diagrams, to specify the pattern's behaviour. On the one hand the benefit is a precisely defined pattern (structure as well as behaviour), which could be looked-up in source code, automatically. On the other hand, we lose the pattern's flexible usage because we fixed exactly one implementation.

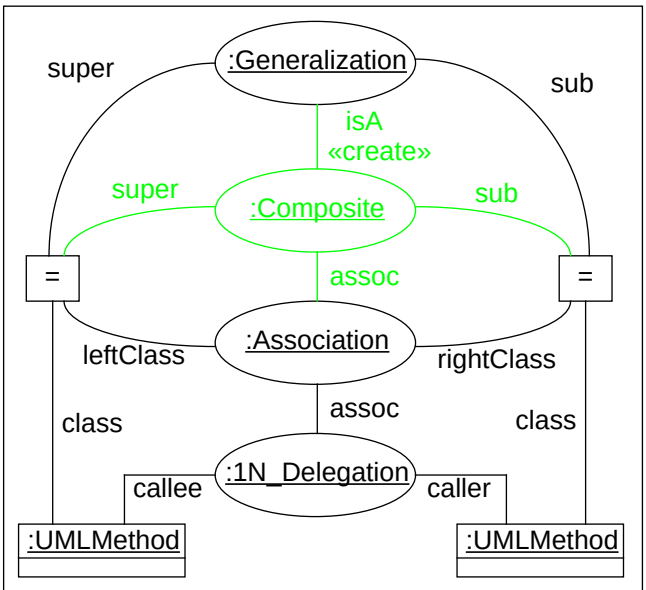

Figure 2: Composite pattern (object structure)

To overcome this inflexibility, we reorganized the patterns and allow to separate common parts of patterns into an own definition. The main difference is, that patterns are constructed out of sub-patterns. Figure 2 shows our definition of the composite pattern taken from [GHJV95]. Since we want to recover patterns, for its definition we use an object diagram at the ASG level. In comparison to figure 1 we left out the Leaf class, because its existence is not necessary, s.o. This definition allows us to concentrate and recover the essential parts of a composite pattern.

The composite definition consists of three kinds of 'objects', i.e. some with oval shape, some original UML objects, and some equal boxes. Oval shaped objects represent other patterns whereas the original rectangle shaped objects represent nodes in the abstract syntax graph. The equal boxes are just an abbreviation of any kind of object, either annotations as well as ASG nodes. Each object must be an instance of a certain class in the domain model, cf. figure 3, and each link between two objects must be an instance of an association in the domain model. The object's type and the link names in the object diagram must be the same as the corresponding class and association names. The (grey) object :Composite marked with «create» indicates, that a new annotation is created, if the black part in the underlying syntax graph is found[2]. This includes the (grey) links from the annotation object to the other objects, too.

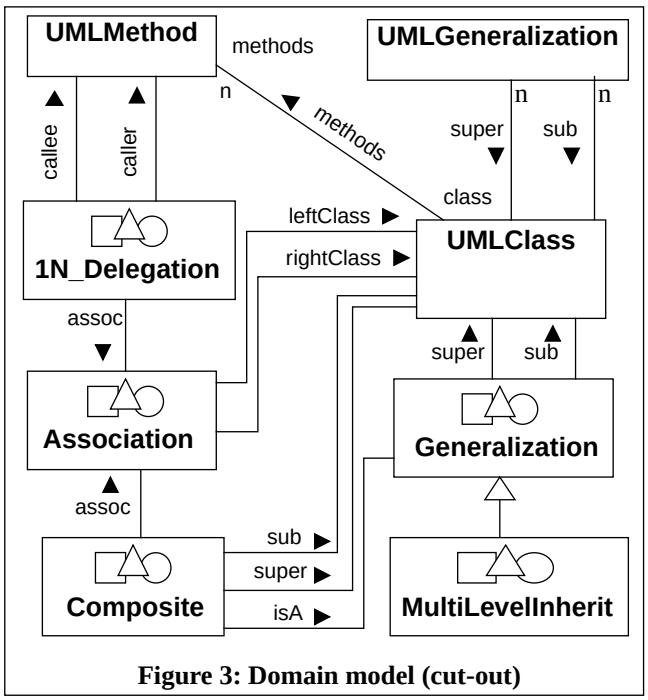

Figure 3: Domain model (cut-out)

The domain model in figure 3, represented as a class diagram, gives an overview of the employed pattern parts and ASG nodes and their relation to each other. Classes representing pattern annotations are marked with a new stereotype «Pattern», which is shown as a specific icon above the class names. Associations are bidirectional, but consist of a defined read direction, which indicates for example that the Composite pattern annotates an Association

1. The Fujaba (From UML to Java And Back Again) environment is developed by the AG Softwaretechnik at the University of Paderborn (www.fujaba.de).

2. This semantics is similar to graph rewrite rules, cf. [Roz97, Zün96]

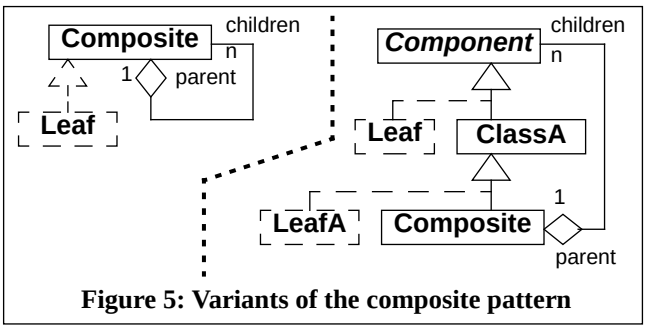

Figure 5: Variants of the composite pattern

and Generalization as well as two UMLClass nodes of the ASG. The default cardinality is exactly one in the read direction and 0..1 in the reverse direction.

In general, common parts in patterns are defined by their own pattern definition. For example, associations and delegations (method calls are delegated to an associated class) exist in many design patterns. Figure 4 shows the

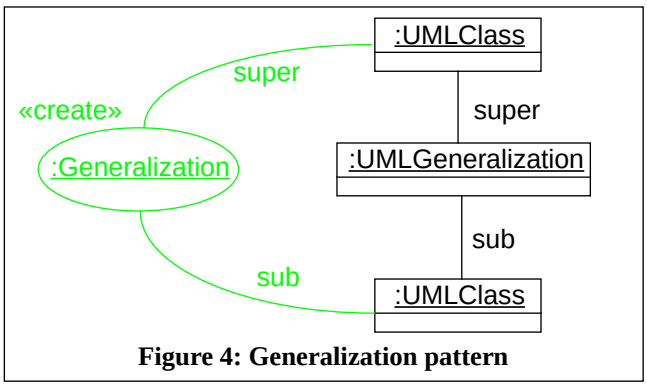

Figure 4: Generalization pattern

generalization pattern definition, which is a straight forward definition, because there already exists an ASG node UMLGeneralization that represents an inheritance relation between two classes.

3 DESIGN VARIANTS

Program comprehension techniques are typically applied for legacy systems in order to get an abstract representation of the software. Design patterns, especially gamma-patterns are one opportunity, but mostly legacy systems do not contain pure original patterns. We analysed some Java programs and libraries, for example the swing library [ELW98], and Fujaba itself [FNTZ98] and found some derivations of the composite pattern, which are shown in figure 5.

On the left hand side, there exists no explicit component class, but a direct aggregation at the composite class itself. This allows each node in the tree to have children. Typically the Leaf class overwrites the access methods for the container or do not access the container in one of its own methods. On the right hand side, the inheritance relation is extended by inserting another class ClassA, which inherits from Component and Composite inherits from ClassA. This design variant allows two different leaf variants, which have common properties.

Such design variants of one pattern result typically in a definition for each variant, in our case two new composite definitions. As an example we allow the definition of patterns consisting of sub-patterns. To facilitate such definitions it is sufficient to define a new annotation called MultiLevelInherit, which annotates the super- and subclass of two connected Generalizations. Figure 6 shows the definition of the multi-level-inheritance pattern.

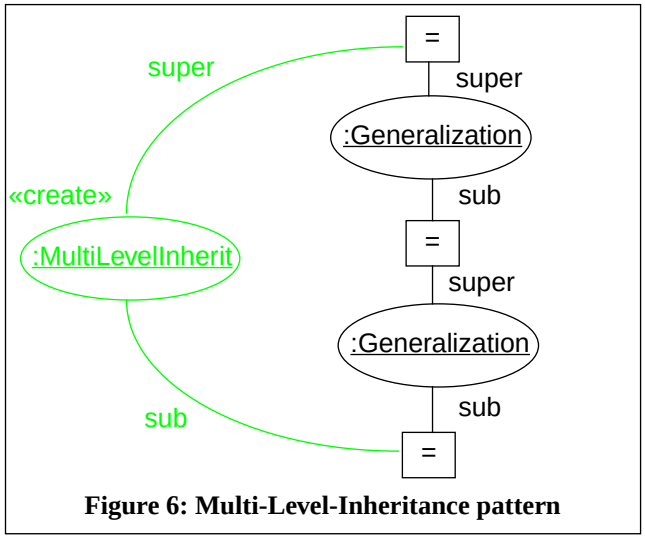

Figure 6: Multi-Level-Inheritance pattern

The multi-level-inheritance pattern is structural, namely the annotation structure, equivalent to a simple generalization pattern. Therefore, in the domain model, cf. figure 3, the multi-level-inheritance pattern is a ‘subclass’ of the generalization pattern. Consequently, the MultiLevelInherit pattern inherits all associations from Generalization pattern and is a Generalization pattern, in the sense of object-oriented concepts.

During the detection process, the polymorphism is used to gain more flexibility in the process. For example, running the detection algorithm on the right variant of the composite pattern in figure 5, results in two (four including the leafs) Generalization annotations. One between Component and ClassA and the second between ClassA and Composite. Without the multi-level-inheritance pattern no composite pattern would be found. But, using it creates an annotation MultiLevelInherit between Component and Composite. Because a MultiLevelInherit annotation is a Generalization annotation it may serve as a match for the Generalization annotation required in figure 2 and a composite annotation is created for the right example in figure 5. Note, a MultiLevelInherit annotation may serve as a match for a Generalization in figure 6, too. Thereby, the recursive use of the multi-level-inheritance pattern allows to detect composite patterns with more than one class in the inheritance hierarchy from Component to Composite.

To recover also the left variant of the composite pattern, we could either add another sub-pattern of Generalization in the domain model or we could define a new sub-pattern of the Composite pattern without inheritance in it. Both solutions are feasible but the second one is better regarding the number of annotations created, because otherwise each class in the abstract syntax graph will have a generalization annotation, automatically.

4 IMPLEMENTATION VARIANTS

In addition to design variants, i.e. after handling the multitude of design pattern variants, we have to deal with implementation variants. For some parts of a design pattern there exist several ways to implement them. For simple syntactical variants, e.g. i=i+1 vs. i++, we refer to parser techniques [ASU86, JCC] for details.

Like mentioned before, analysing only the structure is not sufficient, cf. [KP96]. Method bodies have to be analysed, too, in order to raise the quality of the results. There, we discovered a multitude of implementation variants, one example are head- vs. foot-controlled loops. The mass of different implementation ways is not only limited to methods. Associations can also be coded in several variants. In fact, the recovery of method bodies and associations induces the same problems, i.e. the handling of the syntactical variants for the same semantics. Here we focus on associations only, due to the lack of space.

First, we have different kinds in multiplicity of binary associations, namely OneToOne, OneToMany, ManyToOne, and ManyToMany. Additionally consider qualified, sorted and n-ary associations. We presume, that associations are bidirectional implemented, otherwise we call them references.

Second, an association is composed of two references, which we have to distinguish in their cardinality (0..1, 1, n, 1..n)[1]. To manage this mass of annotation definitions, we have developed a domain model analogous to figure 3. For details see [NWZ01], where a similar approach to support round-trip engineering is presented. This hierarchy concept controls annotation explosion, but does not solve it entirely. We still annotate information that may not be requested and this leads to scalability problems.

Therefore we introduced fuzzy logic [Zad78] in our recovering process. In many cases, we only need to recover the information that an association exits, cf. the composite pattern figure 2. The intentional omitting of information in order to control annotation explosion, leads to less reliability.

For example, figure 7 shows implementation variants for OneToOne associations. Here we annotate the source code directly, because the ASG is very large and gives no new insights. In variant 1, just a public reference to the associated class is supplied, whereas variant 2 implements a private reference and provides read/write operations. To recover the multiplicity of the association, the method bodies have to be analysed. This is done in the ReadOp and WriteOp with the same principle as presented here, cf. [NWZ01].

Variant 3 differs from variant 2 by implementing only one write operation (removing the element is done by setClassF(null)). All-over the common part of these three variants is an attribute pointing to another class. The common parts to infer an association is that two classes point to each other. Thus the reengineer decided that such a pair of references is sufficient to indicate an association in the class

1. The same holds for associations implemented as a separate class.

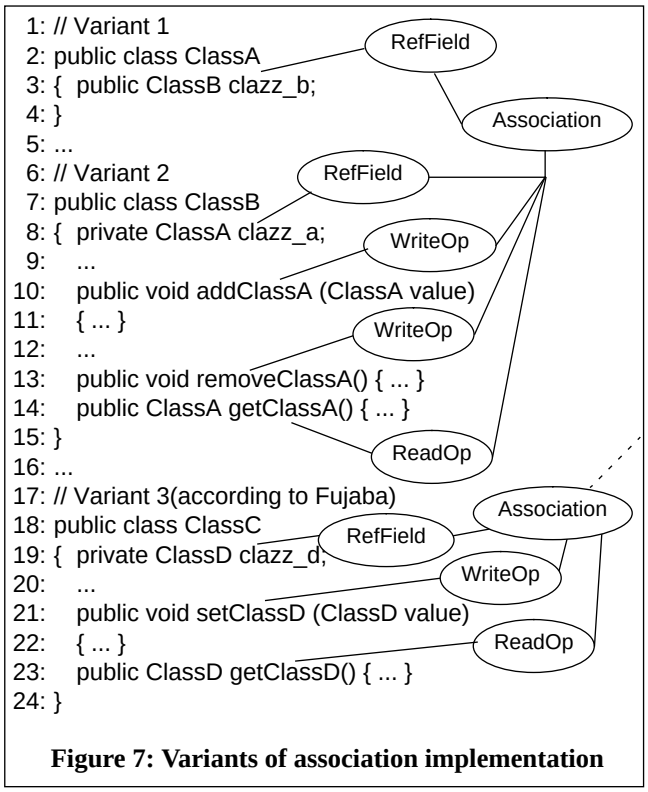

Figure 7: Variants of association implementation

diagram with a certain degree of confidence. To handle more than one association between two classes, again, we have to analyse the method bodies for the explicit assignment.

Figure 8 shows the rule that describes how to find an association. We assume, that an association between two classes consists of one attribute in each class referencing the other class. So, in the ASG there should be a class c1 with an attribute a1 referencing class c2 and vice versa. There must also be annotation objects ann2 and ann5, that represent these references. To confirm the supposed association between classes c1 and c2, optional nodes - indicated by dashed lines - are added. Object ann1 annotates, that there is a method r1 in class c1, which reads attribute a1 and returns its value. This is ensured by the ReadOp pattern, which analyses the method's body, appropriately. Method w1 assigns a value to attribute a1. For class c2 it is the same with methods r2 and w2. Finally, an annotation object ann7 is created to represent an association between classes c1 and c2 with confidence 0.6.

In general, each rule defines a so called fuzzy belief and a threshold value between 0.0 and 1.0. They are specified in the top left corner of the implication after the implications name. In figure 8 the implication i1 has a minimum fuzzy belief of 0.6 and a threshold value of 0.5. A fuzzy belief indicates the certainty, that a given sub-graph represents the searched pattern.

In this example the fuzzy belief is evaluated as follows. If only the necessary parts of the association are found, the implication i1 has a certainty of 0.6. For each optional annotation ann1, ann3, ann4 and ann6, the fuzzy belief of the optional node is added to the overall fuzzy belief. So, if a

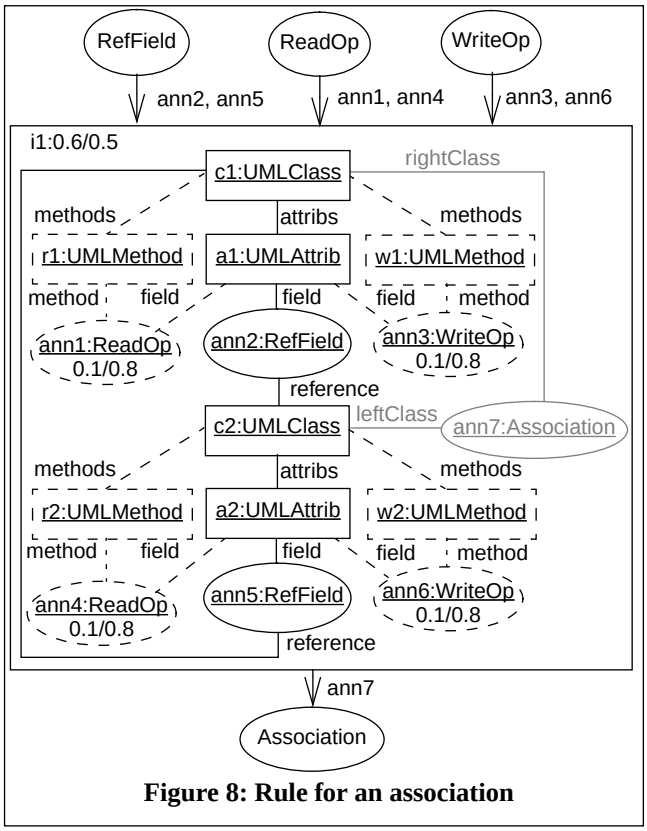

Figure 8: Rule for an association

match for all optional nodes can be found, implication i1 will return a certainty of 1.0. This value is saved by the created annotation object ann7. The threshold value of implication i1 prevents the use of annotations (e.g. ann2 and ann5) with fuzzy beliefs lower than 0.5. Optional nodes have their own threshold values, cf. ann1, ann3, ann4 and ann6 in Figure 8.

Applying fuzzy logic lets us define a look up rule for a pattern which is not a leaf in the look up hierarchy. Children in the hierarchy may recover more detailed information. For example, we employ a similar fuzzy rule to recognize a more detailed association, e.g. a OneToMany association. Of course, an identified OneToMany association with a certain fuzzy belief leads to an association annotation with the same fuzzy belief.

In addition, we provide the definition of contra indications, which can improve the recognition process. For example, the Java programming language does not provide user defined datatypes. User defined datatypes must be implemented at a certain class, which is immutable. Immutable means that all values of an instance of the class cannot be changed after its creation, e.g. complex numbers. The immutable property can be defined as a pattern like associations. Such a class may be annotated by an immutable pattern during the recognition process. In case of UML, classes having a reference to an immutable class are handled as a datatype displayed as an attribute and not as an arrow to the class. This fact can be underlined by defining a contra indication from the immutable pattern to the association pattern. In general, contra indications can be used where language constructs are ambiguously used to implement different design items. Fixing one solution, e.g. a datatype, early in the analysis, can provide better results later on.

In summary, we overcome the flow of information with results from the multitude of variants by a hierarchy concept of sub-pattern and annotations. Combined with optional parts in the rules we appraise the fuzzy values for the certainty of our propositions.

5 INFERENCE MECHANISM

Concerning the search of patterns, Jahnke and Walenstein state in [JW00] that “reverse engineering is an imperfect process driven by imperfect knowledge”. They propose, that reverse engineering tools should be used as a media for reengineering where the reengineer gains high support for all performed tasks and could engage in the analysis wherever needed.

Once a pattern is defined, it has to be recovered in legacy code. To do so, we need an inference engine. Jahnke developed Generic Fuzzy Reasoning Nets (GFRN) for reverse engineering of relational database applications [Jah99, JSZ97]. These nets are described by reverse engineering rules consisting of predicates and implications. An inference engine expands these GFRN on the legacy database code into Fuzzy Petri Nets (FPN) and evaluates them. To serve the purpose of recovering design patterns and clichés, we will adapt this inference engine. Originally, GFRN implications are formulated by relational algebra. We will replace the relational algebra in implications by graph rewrite rules and introduce analysis machines for the inference mechanism, that interpret these rules appropriately.

As mentioned before, the legacy code is parsed into an abstract syntax graph similar to the UML meta model. The ASG is analysed with graph rewrite rules stemming from [FNTZ98]. They look-up and annotate sub-graphs of the ASG. To prevent solving a sub-graph parsing problem [Meh84], we introduced annotation objects to the ASG. These annotation objects serve both, as starting points for searching sub-graphs and as objects that hold information about the found pattern, such as fuzzy beliefs and participating objects.

A fuzzy rule (pattern) for searching a composite design pattern is shown in figure 9. The implication infers with a certainty of 1.0, if a match for this graph rewrite rule is found. The predicate Generalization in the upper left is painted by a thick oval. Such predicates serve as starting points of the inference process. The generalization pattern lies on the lowest level in the hierarchy, which we can infer directly from the ASG. Consequently, we take it as a starting point in this example. In general, each pattern, which consists only of abstract syntax graph nodes can serve as starting point. If there are many possible starting patterns, those are preferred, which are part of patterns on a high level of the hierarchy. Also, the user can define predicates as starting points, which are preferred during the execution. In contrast to starting patterns, associations or delegations are clichés with a lot of variants. A generalization is usually denoted by a key word in the source code. It can be directly parsed into an ASG.

Since generalization along several levels is allowed by a

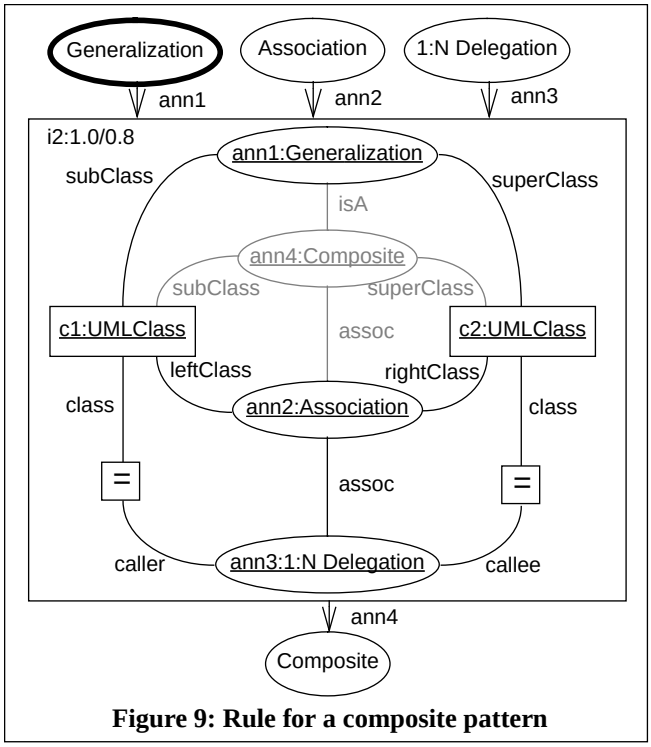

Figure 9: Rule for a composite pattern

composite pattern, the structural inheritance introduced in section 3 is provided by our enhanced GFRN. The annotation ann1 in figure 9 means both Generalization and MultiLevelInherit, cf. figure 4 and 6.

The original GFRN used in an earlier approach [JNW00] have to be changed to support these new features. First of all the relational algebra has to be replaced by graph rewrite rules as mentioned above. Then the inference machine has to be adjusted. The old strategy creates basic facts out of all axioms - predicates with no antecedent implication. This mechanism is no longer feasible. Axioms like ReadOp or RefField would produce too much basis facts when we reengineer legacy code of several thousand lines of code, cf. [Wil94].

We have developed an algorithm that differs from the original inference mechanism of GFRN. Our new approach for the recognition of design patterns and clichés can be compared to a seed, that first grows up to the light and then builds some roots to fortify its stability. Predicates, that are preconditions to an implication, can be triggered with additional parameters by this implication. So we are able to mark only a few axioms as basis facts. Axioms like RefField and ReadOp can be triggered to reduce investigations to the objects of interest. We will explain the algorithm by the composite pattern example below.

The recognition is kicked off by an UMLGeneralization object g in the ASG. It triggers the rule for the basis fact Generalization and an annotation object for the two participating classes c1 and c2 is created. Note, a Generalization pattern lies on the lowest level of the hierarchy. Since the rule in figure 9 declares the Generalization as a precondition for a Composite, the Composite is triggered. Now, the direction changes from

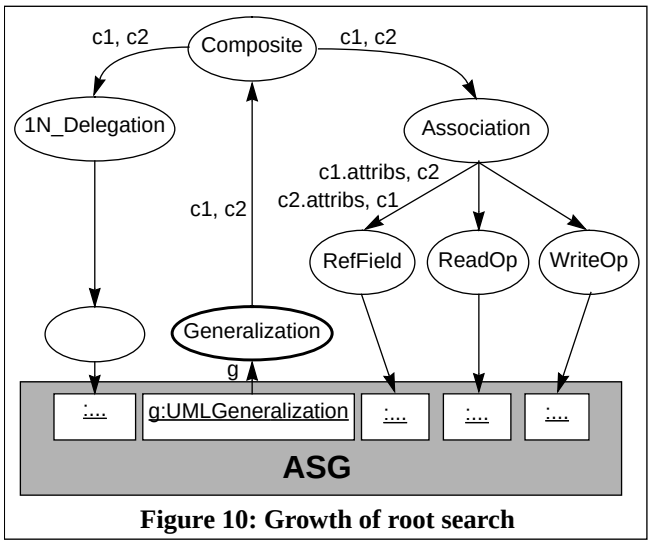

Figure 10: Growth of root search

bottom-up to top-down, in order to manifest the result. As specified in figure 9, to infer a composite we also need an Association and 1N_Delegation, too. These last ones may not be detected, yet. This results from the existence of an association or delegation annotation missing to fulfil the rule. Supposing both are missing, the recognition of an association is started by transferring the UMLClass objects c1 and c2 to the Association pattern rule. Now, a reference field in each class needs to be found. So the RefField rule has to be evaluated twice with different parameters, class c2 and all attributes of type c1 and vice versa. The same has to be done for all optional annotations ReadOp and WriteOp. To infer a composite pattern, each antecedent implication has to reach a threshold value of 0.8 for its fuzzy belief. If no optional annotation in the implication i1 for Association is found, the fuzzy belief of the association annotation is only 0.6. At least two optional ReadOp or WriteOp annotations must be found. Otherwise the composite pattern can not be inferred.

Figure 11 presents the informal algorithm for the evaluation

```
 1: evaluate (Set trigger, Boolean up)
 2:     calculate partial match for story pattern with triggers
 3:     for all antecedent implications antImp do
 4:       if not antImp.isEvaluated () then
 5:         create set of triggers for antImp
 6:         antImp.evaluate (setOfTriggers, false)
 7:       endif
 8:     calculate rest of match for story pattern
 9:     if match.isComplete () then
10:       calculate fuzzy belief
11:       calculate minimum min of fuzzy beliefs of
            antecedent implications
12:       create resulting annotation object ann
13:       if up and min>threshold then
14:         for all consequent implications conImp do
15:           conImp.evaluate (ann, true)
16:       endif
17:     endif
18: end
```

Figure 11: Algorithm for evaluating implications

of implications. A set of triggers and a direction for the evaluation is given to the algorithm. First of all in line 2 a partial match as complete as possible for the graph rewrite rule of the implication is calculated. If the implication in figure 9 is evaluated for example, the trigger would be an annotation object ann1 of type Generalization. The first match would include the two classes c1 and c2. Now, all invalidate antecedent implications have to be verified (line 3 to 7). Evaluation for the implication of Association (figure 8) is called with triggers c1 and c2 in line 6. After the recursive evaluation of all antecedent implications, line 8 calculates a complete match for the graph rewrite rule. If no match can be found, the algorithm terminates. Otherwise the fuzzy belief of this implication and the minimum of all fuzzy beliefs of antecedent implications is calculated (lines 10 to12). If the minimum is greater then the threshold, all consequent implications will be evaluated in line 15.

The purpose of this detection algorithm is the principle to get predictions about design pattern occurrences fast but best founded. In general, this procedure creates mainly the same results as the original inference mechanism of the GFRN. The difference is that we highly embrace the reengineer to make defined decisions, early in the execution. For example, during the recognition, the reengineer has the possibility to intervene results. Annotations can be rejected or fuzzy beliefs can be changed by the user. The inference engine takes these changes directly into account. For example, if the inference machine creates an annotation for an association with a fuzzy belief of 0.6 and the user rejects this annotation, the inference machine re-evaluates all consequences. If the engineer raises the fuzzy belief to 1.0, the inference machine has to re-evaluate all consequences, too, because of the low fuzzy belief, the annotation could have been rejected by a higher threshold elsewhere. However, this also works the other way around. The inference machine can ‘ask’ the user in certain situations to make a decision, because otherwise the produced results are not reliable enough. Another opportunity may be that the machine directs the reengineer to those annotations, which have been defined as important or interesting by the user during the definition of the pattern.

Since providing a complete solution for all purposes (patterns) in legacy systems is not possible, adaption opportunities are very important. Our idea is to provide a basic library of design pattern implementation variants a reengineer can start with. After some analysis, the reengineer can adapt the library for his special purposes.

6 RELATED WORK

In [HN90] Harandi and Ning present program analysis based on an Event Base and a Plan Base. Therefore, they construct rudimentary events from source code. Plans are used to define the correlation between one or more (incoming) events and they fire a new event which corresponds to the plan’s intention. Annotations visualize the event flow and plan definition. The annotation technique allows to use the same event more than one time. This is similar to context-sensitive graph parsing presented in [RS95]. In [PP94] Paul and Prakash introduce a matching algorithm for syntactic patterns based on a non-deterministic finite automaton. The non-determinism is used to provide dummy variables for special pattern symbols representing syntactical information like variables or function calls. Both [HN90] and [PP94] need one definition for one implementation variant, which lets the approaches fail for at least legacy systems with unknown code-styles.

An approach to recognize clichés, which are commonly used computational structures, is presented in [Wil94], within the GRASPR system. Legacy code to be examined is represented as flow graphs by GRASPR, clichés are encoded as an attributed graph grammar. The recognition of clichés is formulated as a sub-graph parsing problem. Solving the sub-graph parsing problem would find matches of clichés, but this has proven to be NP-complete and lets the approach fail for systems with more than several 1000 lines of code.

There exist also various approaches to recover design patterns out of legacy code. Krämer and Prechelt present in [KP96] an analysis approach to extract structural information. Therefore they analyse only the structural parts of a program, i.e. the header files in C++. Skipping the dynamic parts lets their approach fail, because many patterns are structurally equivalent, but behaviourally different. In addition, the detection of associations, which are structural information, can typically not be recovered without analysing the dynamic parts, c.f [NWZ01].

Analysing structure as well as behaviour based on patterns is presented by Keller et al. in [KSRP99]. They use a common abstract syntax graph model for UML, namely the CDIF format to represent the source code as well as the patterns. Matching the pattern’s syntax graph on the program’s syntax graph is done by scripts, which have to be implemented manually. The approach uses sub-patterns, which allows a flexible adaptation for certain systems, but each pattern must be manually adapted and there exist no constraint analysis of the patterns.

In contrast to the manual implementation of pattern matching algorithms, Radermacher uses in [Rad99] the graph rewrite system Progress [Zün96] to match patterns on the program. The syntax graph model is a simplification of the UML meta model. Radermacher also shows how to replace bad implementations of patterns by good ones [JZ97].

Concerning the execution, Quilici has stated that a pure bottom-up approach is not feasible for large systems. Especially the [Wil94] approach fails due to the very high number of ‘base’ results, which are not used in further analysis. A combined bottom-up and top-down approach will produce better results [Qui94], because the analysis is done more goal driven.

Jahnke uses Generic Fuzzy Reasoning Nets (GFRN) for reverse engineering relational database applications [Jah99, JSZ97]. These nets are described by reverse engineering rules consisting of predicates and implications. Predicates are divided into data-driven, which are unrevisable analysis results, and goal-driven predicates. The goal driven concept allows to suspend a time intensive analysis or to invoke them on demand. The GFRN were applied on a commercial

relational database system in a project with a large german drug distributor.

7 CONCLUSION AND FUTURE WORK

In this paper we have presented an approach to recover design patterns from Java source code. Design patterns, especially the gamma-patterns, are mostly informal, which is not feasible for reverse-engineering facilities. In order to ensure a flexible adaption, we construct our patterns out of sub-patterns. Therefore we use graph rewrite rules and fuzzy logic. We also introduced the difference between design variants and implementation variants of patterns. For design variants we provide an inheritance concept for patterns and in case of implementation variants, we use fuzzy logic to reduce the number of pattern definitions.

To overcome the problem of long execution times of the pattern look-up process, we presented a highly user involved inference mechanism. This mechanism is goal driven and allows the reengineer to react on intermediate analysis results in order to benefit of his knowledge.

The resulting annotation structure on top of the abstract syntax graph may be used for different purposes. Documenting the design and the source code may be one reason, but also further work may be done, e.g. rewriting bad implementations by good implementations. Another opportunity may be to recover structural information as we presented in [NWZ01]. It is also possible to use our technique to review company specific code styles, which is of main interest in developing and testing phases.

Currently, we are working on tool support for our approach. Therefore, we want to enhance our Fujaba environment, which already comprises of a recovering module. Unfortunately, this module is hard-coded and inflexible. Our first attempt was to use graph rewrite rules to look-up a pattern, whereas method bodies were analysed using regular expressions. The integration of the new inference mechanism is partly done, but not finished, yet.

Future work is to provide a complete and user friendly environment for the recovery of patterns. This includes an easy way to define and adapt patterns and a human based interaction graphical user interface for the execution process. The next step will be to perform case studies and apply the tool in other domains As case studies, we want to take the round-trip approach [NWZ01] and the analysis of java card application in [JNW00]. Therefore we aim to provide a database and the opportunity to exchange and adapt patterns.

In parallel, we want to improve the inference process by providing rules in order to control the process itself. We assume that such an improvement provides faster and more reliable results.

ACKNOWLEDGEMENTS

We thank Ulrich A. Nickel and Albert Zündorf for hard discussions, many contributions to this paper, and proof reading. And many thanks to Jürgen Maniera and Friedhelm Wegener, who have provided us with the newest technical equipment to produce this paper. Thank you.

REFERENCES

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullmann. *Compilers – Principles, Techniques and Tools*. Addison Wesley, 1986.

[BFLS99] S. Bünnig, P. Forbig, R. Lämmel, and N. Seemann. *A Programming Language for Design Patterns*. Technical Report 258-8/1999, University of Hagen, Hagen, Germany, September 1999.

[CFM93] A. Cimitile, A.R. Fasolino, and P. Marascea. Reuse Reengineering and Validation via Concept Assignment. In *Proc. of the 3rd International Conference on Software Maintenance*, pages 216–225. IEEE Computer Society Press, September 1993.

[CILS93] D.D. Cowan, R. Ierusalimschy, C.J.P. Lucena, and T.M. Stepien. Abstract data views. *Structured Programming*, 14(1):1–13, January 1993.

[CKV96] J.O. Coplien, N.L. Kerth, and J.M. Vlissides. *Pattern Languages of Program Design (Volume 2)*. Addison Wesley, 1996.

[ELW98] R. Eckstein, M. Loy, and D. Wood, editors. *Java Swing*. O'Reilly, 1998.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In G. Engels and G.Rozenberg, editors, *Proc. of the 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*. Springer Verlag, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[HC98a] C.A. Horstmann and G. Cornell. *Core Java 2, Volume 1: Fundamentals*. Java Series. Prentice Hall, 1st edition, 1998.

[HC98b] C.A. Horstmann and G. Cornell. *Core Java 2, Volume 2*. Java Series. Prentice Hall, 1st edition, 1998.

[HN90] M. T. Hanrandi and J. Q. Ning. Knowledge Based Program Analysis. In *Journal IEEE Software, volume 7, number 1*, pages 74–81, January 1990.

[Jah99] J.H. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Paderborn, Germany, September 1999.

[JCC] SUN Microsystems. *JavaCC, the SUN Java Compiler Compiler. Online at http://www.suntest.com/JavaCC.*

[JNW00] J.H. Jahnke, J. Niere, and J.P. Wadsack. Automated Quality Analysis of Component Software for Embedded Systems. In *Proc. of the 8th Int. Workshop on Program Comprehension (IWPC), Limerick, Irland*. IEEE Computer Society Press, 2000.

[JSZ97] J.H. Jahnke, W. Schäfer, and A. Zündorf. Generic Fuzzy Reasoning Nets as a basis for reverse engineering relational database applications. In *Proc. of European Software Engineering Conference (ESEC/FSE)*, number 1302 in LNCS. Springer Verlag, September 1997.

[JW00] J.H. Jahnke and A. Walenstein. Reverse Engineering Tools as Media for Imperfect Knowledge. In *Proc. of the 7th Working Conference on Reverse Engineering (WCRE), Brisbane, Australia*. IEEE Computer Society Press, 2000.

[JZ97] J.H. Jahnke and A. Zündorf. Rewriting poor Design Patterns by Good Design Patterns. In Serge Demeyer and Harald Gall, editors, *Proc. of the ESEC/FSE Workshop on Object-Oriented Re-engineering*. Technical University of Vienna, Information Systems Institute, Distributed Systems Group, September 1997. Technical Report TUV-1841-97-10.

[KNNZ99a] T. Klein, U. Nickel, J. Niere, and A. Zündorf. *From UML to Java And Back Again*. Technical Report to appear, University of Paderborn, Paderborn, Germany, September 1999.

[KNNZ99b] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. *Using UML as a visual programming language*. Technical Report tr-ri-99-205, University of Paderborn, Paderborn, Germany, August 1999.

[KNNZ00] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proc. of the 22th Int. Conf. on Software Engineering (ICSE), Limerick, Irland*. ACM Press, 2000.

[KP96] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proc. of the 3rd Working Conference on Reverse Engineering (WCRE), Monterey, CA*, pages 208–215. IEEE Computer Society Press, November 1996.

[KS98] R.K. Keller and R. Schauer. Design Components: Towards software composition at design level. In *Proc. of the 21th Int. Conf. on Software Engineering, Los Angeles, USA*, pages 302–310. IEEE Computer Society Press, April 1998.

[KSRP99] R.K. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-Based Reverse-Engineering of Design Components. In *Proc. of the 21th Int. Conf. on Software Engineering, Los Angeles, USA*, pages 226–235. IEEE Computer Society Press, May 1999.

[Meh84] K. Mehlhorn. *Graph Algorithms and NP-Completeness*. Springer Verlag, 1st edition, 1984.

[Mey88] B. Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 1988.

[NNWZ00] U. Nickel, J. Niere, J. Wadsack, and A. Zündorf. Roundtrip Engineering with FUJABA. In *Proc of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany*. Technical Report University of Karlsruhe, 2000.

[NWZ01] J. Niere, J.P. Wadsack, and A. Zündorf. Recovering UML Diagrams from Java Code using Patterns. In *Proc. of 2nd Workshop on Soft Computing Applied to Software Engineering, Enschede, The Netherlands*, Lecture Notes in Computer Science (LNCS). Springer Verlag, 2001.

[PP94] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, June 1994.

[Qui94] A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, May 1994.

[Rad99] A. Radermacher. Support for Design Patterns through Graph Transformation Tools. In *Proc. of Int. Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade, The Netherlands*, LNCS. Springer Verlag, 1999.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, Singapore, 1997.

[RS95] J. Rekers and A. Schürr. A Graph Grammar Approach to Graphical Parsing. In *Proc. of the IEEE Symposium on Visual Languages, Darmstadt, Germany*. IEEE Computer Society Press, 1995.

[SS00] P. Selonen and T. Systä. Scenario-Based Syntesis of Annotated Class Diagrams in UML. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Minneapolis, Minnesota USA*. IEEE Computer Society Press, October 2000.

[UML] Rational Software Corporation. *UML documentation version 1.3 (1999). Online at http://www.rational.com.*

[Wil94] L.M. Wills. Using Attributed Flow Graph Parsing to Recognize Programs. In *Int. Workshop on Graph Grammars and Their Application to Computer Science*, Williamsburg, Virginia, November 1994.

[Zad78] L.A. Zadeh. Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems*, 1978.

[Zün96] A. Zündorf. Graph Pattern Matching in PROGRES. In *Proc. of the 5th International Workshop on Graph-Grammars and their Application to Computer Science*, LNCS 1073. Springer Verlag, 1996.