Softw Syst Model (2005) 4: 55–70 / Digital Object Identifier (DOI) 10.1007/s10270-004-0059-9

An approach for reverse engineering of design patterns

Ilka Philippow, Detlef Streitferdt, Matthias Riebisch, Sebastian Naumann

Technische Universität Ilmenau, Helmholtzplatz 1, 98693 Ilmenau, Germany
E-mail: {ilka.philippow,detlef.streitferdt,matthias.riebisch}@tu-ilmenau.de

Received: 5 December 2002/Accepted: 29 January 2004/Published online: 29 April 2004 –

Abstract. For the maintenance of software systems, developers have to completely understand the existing system. The usage of design patterns leads to benefits for new and young developers by enabling them to reuse the knowledge of their experienced colleagues. Design patterns can support a faster and better understanding of software systems. There are different approaches for supporting pattern recognition in existing systems by tools. They are evaluated by the Information Retrieval criteria precision and recall. An automated search based on structures has a highly positive influence on the manual validation of the results by developers. This validation of graphical structures is the most intuitive technique. In this paper a new approach for automated pattern search based on minimal key structures is presented. It is able to detect all patterns described by the GOF [15]. This approach is based on positive and negative search criteria for structures and is prototypically implemented using Rational Rose and Together.

Keywords: Design patterns – Reverse engineering – Pattern recognition

1 Introduction

During the lifecycle of software systems, maintenance activities are aiming towards the management and integration of new or changed requirements. For this purpose software developers have to understand the existing system completely. Developers have problems to understand software systems, because of missing or poor documentation like specifications or design models. In most cases the original developer cannot be contacted anymore. Developers can be happy, if at least the source code, as the most rudimentary and reliable form of documentation, is available.

Design patterns offer predefined and tested solutions for fundamental design problems. The usage of design patterns leads to benefits for new and young developers by enabling them to reuse the knowledge of their experienced colleagues. Identification of design patterns contained in system as well as determination of source code classes for the identified patterns would lead to an improved understanding of the pattern based part of existing systems. Patterns are not explicitly described in software source code – excluding annotations or references in the documentation. The information about design patterns used in software systems is implicitly hidden and has to be detected manually in most cases.

This paper proposes an approach for the automated detection of patterns in existing source code. The approach is an extended version of existing pattern search algorithms based on minimal key structures. It focuses on the patterns described in [15], since they are a selection of practically relevant and useful patterns for software developers and the de-facto standard. Structural as well as behavioral pattern searching is covered. Extension of the approach to new patterns out of the broad range of available solutions is subject of further research efforts.

The paper is subdivided into the following sections: It starts in Sect. 1.1 with a brief overview of software maintenance and the relevance of source code understandability in connection with design patterns. In Sect. 2 existing approaches and activities for the automated detection of design patterns are discussed. Section 3 contains the new proposed approach, as an integration and enhancement of existing approaches. In addition, a description of the prototypical implementation in Rational Rose is part of this section. The paper concludes with an evaluation of the approach and an outlook onto our further work.

1.1 Maintenance and design patterns

In [14] the results of several studies of different phases of the software life cycle are discussed and summarized. Based on these studies software maintenance can be con-

Table 1. Maintenance activities [14]

Activity	Part of expenses
Understanding of requirement	18%
Understanding of documentation	6%
Understanding of code	23%
Implementation	19%
Test	28%
Adaptation of documentation	6%

sidered as the most expensive part of the software life cycle. The most optimistic study estimated more than 40% effort for software maintenance. Results of the evaluation of maintaining activities are shown in Table 1.

The summary of the first three lines in Table 1 shows that 47% of the maintenance costs are due to the understanding of software. The application and reuse of design patterns will reduce the effort to be put into the understanding of software. This paper is based on design patterns for object-oriented software development out of [15]. Design patterns offer a set of objects and classes for solving a particular design problem in a certain context. A design pattern is described by the following elements:

- Name, brief description of problem and solution.
- Problem description (usage scenarios).
- Solution description (involved classes, objects and their interaction behavior).
- Consequences, advantages and disadvantages.

Patterns can be used for solving recurrent design problems. Design patterns can be categorized as follows: there are generation patterns (generation of objects), structural patterns (composition of classes and objects) and behavioral patterns (interaction of objects). For a detailed description and explanation of design patterns see [15].

How can design patterns help to understand software systems? Each pattern represents an idea for solving a particular design problem. If software developers are familiar with the pattern idea they should be able to understand the function of the involved classes fairly fast. Based on an experiment in [11] it is proven that documented patterns lead to an easier and better understanding of software systems. Within the experiment of [11], more than 70 students were told to modify a given system. As the result of the experiment, students with knowledge about patterns were the fastest in changing well documented code – patterns were documented. Students with knowledge about patterns but without a documentation containing the patterns were a bit slower. Students without knowledge about patterns were the slowest group in the experiment. In addition, architectural changes were carried out faster and better using pattern knowledge.

In the best case, the structure of a software system contains only pattern-based classes, while the program documentation describes which classes belong to each pattern. In practice this idealistic case does not occur and it is difficult to identify patterns in complex software systems. Examining the class structure is not enough for recognizing a special pattern. The class structures of the STRATEGY and STATE pattern are almost identical for instance, but their semantics are different. Usually programming languages like Java or C++ do not support the application and documentation of patterns explicitly. The only possibility is an extensive documentation using comments. Unfortunately this is strongly dependent on the discipline of the developers. Beyond this, the documentation of software developed before the explicit definition of patterns does not contain any hints concerning patterns. The detection of patterns is not trivial and the manual search for patterns would consume high resources. There are existing methods for the automated identification of patterns, but there are too many limitations discussed in the next section. Thus, a new method is needed.

2 State-of-the-art

Existing methods for automated pattern identification are evaluated according to the achieved results of their search algorithms. For a better understanding of the following terms a figure of the process of pattern recognition and the corresponding terms are shown in Fig. 1.

The search for patterns leads to three possible results:

- **True positive** in case a pattern has been recognized and the pattern is really implemented within the software system. This case is desired.
- **False positive** in case a pattern has been recognized and the pattern is not really implemented within the system. This case has to be avoided.
- **False negative** in case an implemented pattern has not been recognized. This case has to be avoided.

Based on the achieved results it is possible to derive metrics for the evaluation of searching tools, as described by the recall and the precision of the corresponding algorithms. Both metrics are used widely for evaluating search results, e.g. in Information Retrieval [8].

- **Recall** is the number of all implemented patterns in a software system divided by the number of recognized

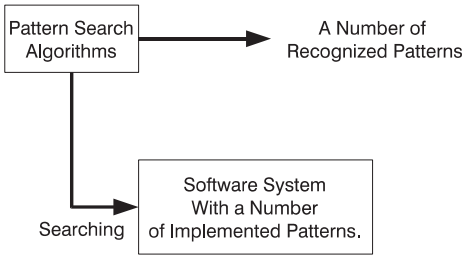


Fig. 1. Basics of pattern searching

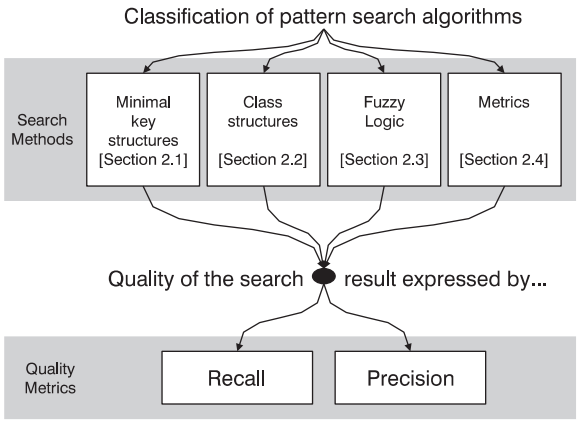


Fig. 2. Evalutation criteria

patterns. A recall of 100% means that at least all implemented patterns were recognized. One might have recognized more, but the implemented patterns are all recognized – case 2, false positive has been avoided.

– **Precision** is the ratio of recognized and really implemented patterns (true positive) divided by the number of recognized patterns (sum of the results true positive and false positive). A precision of 50% means, that half of the recognized patterns are not really implemented in the software system.

Both values have to be taken into consideration for a tool evaluation. A precision value of 100% does not exclude false positive cases.

Several existing approaches for automated pattern search have been evaluated, together with available information about the above-explained metric values. The approaches can be categorized by four different search strategies, explained in the following four sections accordingly.

To sum it up, current pattern search algorithms can be categorized into algorithms searching for minimal key structures, searching for class structures, searching based on fuzzy logic, and searching based on metrics. For each algorithm we can specify values for the quality of the recognition process. Figure 2 depicts the classification and evaluation scheme for pattern search algorithms used in this paper.

2.1 Searching for minimal key structures

A defined key structure is assigned to a particular pattern. Key structures for patterns describe the minimal class and/or object structure that has to be present, so the pattern can be securely identified. The properties of the key structure are used as search criteria. Based on this approach three software systems for automated searching are known. DP++ [2] for C++, KT [4] for Smalltalk and SPOOL [5] realized for C++, applicable for Java and Smalltalk.

DP++ [2] reported that there are search criteria for the following patterns: COMPOSITE, DECORATOR, ADAPTER, FACADE, BRIDGE, FLYWEIGHT, TEMPLATE METHOD and CHAIN OF RESPONSIBILITY. Search strategies are described for the patterns COMPOSITE, DECORATOR and ADAPTER. The applicability for other patterns is not mentioned. The tool consists of three parts: the C++ Code Translation Subsystem for analyzing source code, the Pattern Detection Subsystem for the recognition of generation patterns and the Display Subsystem for the visualization of detected patterns. The search system has been tested with source code containing 30–400 classes. Information about the achieved values of recall and precision is not available.

KT [4] uses defined criteria to find COMPOSITE, DECORATOR, STATE, STRATEGY, COMMAND, TEMPLATE METHOD and CHAIN OF RESPONSIBILITY patterns. The INTERPRETER pattern cannot be recognized. Other patterns are not mentioned. In contrast to DP++, code analysis is not carried out. The recognition relies on the special meta level properties of Smalltalk. Based on the source code, a class diagram and an interaction diagram are created. The recognition is carried out using the diagram information for a query search. As an example, the class diagram serves to recognize COMPOSITE and DECORATOR patterns; the interaction diagram is necessary to recognize the CHAIN OF RESPONSIBILITY pattern. For the representation of results a Rational Rose Petal File has to be generated. The test of the search system was done using software systems with 40–264 classes. The tool was successful in the detection of COMPOSITE, DECORATOR and TEMPLATE METHOD patterns, based on [4]. Information about the achieved values of recall and precision is not available.

SPOOL [5] is capable of searching BRIDGE, FACTORY METHOD and TEMPLATE METHOD patterns. These patterns are important for the understanding of frameworks [5]. Similar to DP++ the search is based on code analysis and recognition of pattern assigned key properties. The SPOOL system consists of several small tools with analyzing and searching functions. Based on the analysis of code a UML model has to be derived and stored in a Design Repository. Using the model information a query search has to be done, similar to KT. The Design Representation Module offers a good graphical presentation of results. KT has been tested using large software systems with 722–3101 classes. Information about the achieved values of recall and precision is not available.

2.2 Searching for class structures

This approach uses the pattern class structures described by [15]. For example, the simplified class structure in Fig. 3 belongs to the COMPOSITE pattern. If a class owns at least two sub classes and if one sub class has a 1 to n aggregation relation to the super class the existence of a COMPOSITE pattern will be reported. Three software

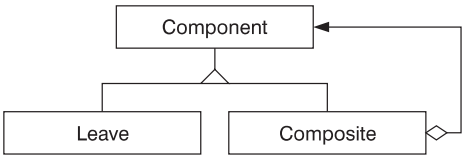


Fig. 3. COMPOSITE pattern

systems for automated search based on complete accordance of classes are known: Pat [7] for C++, IDEA [3] for UML diagrams and the multi step search tool in [1].

Pat [7] is based on the description of the pattern class structure by PROLOG rules. The information in C++ source code has to be transformed into PROLOG facts. A PROLOG query algorithm is used for recognizing patterns. This tool can find ADAPTER, PROXY, BRIDGE, DECORATOR and COMPOSITE patterns. The PAT source code analyzer is examining only the C++ Headers. There are several difficulties, e.g. distinguishing abstract and concrete classes. Furthermore, delegation relations and the visibility of methods cannot be recognized. The search tool has been tested with software systems containing 9–343 classes. The achieved recall values in each of these test cases are 100%. The average precision value is 36.75%.

IDEA [3] is an UML based search approach using class and collaboration diagrams. Similar to Pat, PROLOG rules are used. In addition to PAT, design and application rules are implemented for supporting the pattern user. The search algorithm, depicted in Fig. 4, is able to find the following patterns: TEMPLATE METOD, PROXY, ADAPTER, BRIDGE, COMPOSITE, DECORATOR, FACTORY METHOD, ABSTRACT FACTORY, ITERATOR, OBSERVER and PROTOTYPE. Further information about the test of the IDEA system and achieved metric values could not be found.

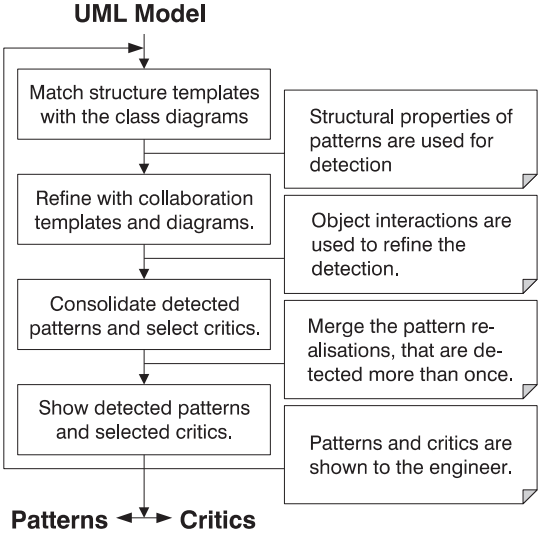


Fig. 4. IDEA search algorithm [3]

In the multi step search process of [1] C++ source code or OMT representation are transformed by tools into an AOL (UML based Abstract Object Language) Code and DESIGN Specification. A Pattern Recognizer compares these AOL representations with pattern assigned AOL representations that are implemented in a so-called Pattern Library. Collaboration diagrams cannot be included in the recognition process. To accelerate the recognition process, constraints are used. The recognition is performed in three steps.

First, the Metrics Constraint Evaluation will extract structural relationships according to defined pattern metrics, which are given by class roles, associations or inheritance relations. Thus, a set of patterns will be identified by its relationships. Within the second step the Structural Constraint Evaluation uses the set of patterns recognized in step one to extract those potential patterns which also conform to all structural relations without delegation. The last step, Delegation Constraint Evaluation, analyzes the difference between delegation and association to enhance the overall pattern recognition process. The patterns ADAPTER, BRIDGE, PROXY, COMPOSITE and DECORATOR can be recognized using this method. Other patterns are not covered. The search tool was tested with different C++ Libraries [1] and resulted in a recall value of 100% and an average precision of 35%.

2.3 Pattern definition and fuzzy logic

Described in [10] this approach considers structural differences between patterns out of [15] and real life software systems. To handle this problem, patterns are considered as a composition of sub patterns with inheritance relations. The proposed idea uses fuzzy logic search algorithms to examine different pattern implementations. The researchers are working on the development and implementation of their idea.

As shown in Fig. 4, IDEA uses UML diagrams as input. This graphical representation of the code structure is used to find the patterns described with UML as well. Critics are formulated as requirements, hints and constraints for a given pattern. These critics have to be met by subparts of the input UML diagram to be identified as pattern. Reference patterns, to be recognized in the UML diagram, are described with PROLOG. IDEA claims to be able to find all patterns of [15] without an available implementation or any values given for the quality of the algorithm.

2.4 Metrics

The basic idea of metric-based approaches is to characterize each pattern by metrics summarized in this section. Described in [6] , there are three categories of metrics with examples for each category:

- Object-oriented Metrics
  - weighted methods per class
  - depth of inheritance tree
  - number of children (subclasses)
  - coupling between objects
- Structural Metrics
  - Fan-in, number of modules sending information to the observed module
  - Fan-out, number of modules receiving information of the observed module
  - information flow, structural complexity
- Procedural Metrics
  - pure lines of code
  - McCabe's cyclomatic complexity
  - lines of comments

These metrics are calculated with a tool for the system in question, for every of the desired patterns. A signature will be assigned to each pattern, which basically is a list of values for the metrics described above. For each class with all subclasses of the evaluated system the metrics are calculated as well. Potential patterns can be recognized by comparing the list of metrics for each class of the system with the pattern metrics.

For the metric approach a pattern repository and a pattern wizard are available. The tools have been tested and the tests resulted in a low average precision value of 43.55%. Based on [6] the approach is applicable for all kinds of patterns in [15].

2.5 Manual search for patterns

A manual search method is described in [16]. It is a simple six step description of how to find patterns.

1. Read and try to understand the specification documents.
2. Setup a brief class model with the class declarations in the code.
3. Refine the class model based on the implementation.
4. Try to find patterns in the model using inheritance and associations between the classes of the system.
5. Analyze the potential pattern of step 4.
6. Try to consult the original programmers and developers for a better understanding of the system.

Within a student test this approach has proven to be intuitive. The structural strategy is embedded in the steps of the manual method, due to its close relation to the human way of thinking and searching for patterns.

Section 2 has shown the difficulties of current pattern searching research efforts. Just one of the approaches has the potential to find all of the 23 patterns described in [15], although with a low precision value. The other approaches are only usable for a subset of the patterns. The most promising algorithm searches for minimal key structures, as described in [2, 4], and [5], what is also closest to the human way of thinking.

3 Our approach: Automated pattern searching

The automated pattern search is not a trivial problem. As summarized in Table 2, most of the approaches only detect a subset of the patterns described in [15]. For some patterns no search criteria can be recognized according to [5] and [4]. The approach based on metrics seams to find all patterns, but with a precision value of 43.55% the approach is not satisfactory. The known approaches based on the complete accordance of structure will run into problems for pattern structures differing from descriptions out of [15].

To solve the problem of automated pattern search it is necessary to meet the following requirements:

- Identification criteria for all patterns in [15] are needed.
- Search algorithms for all patterns need to be defined,
- to achieve high metric values (ideal case 100%) for precision/recall and thus, to find all patterns described by Gamma [15], false negative and false positive cases have to be avoided.

3.1 Positive and negative criteria

The objective of this approach is to contribute to the realization of the requirements described above.

We start with the assumption that developers can validate potential patterns a lot easier when they understand the searching process. Based on experience the human search process is closely oriented to pattern properties, what itself is similar to the automated search approach based on minimal key structures.

In this paper the proposed approach is based on the results of [2, 4] and [5]. For improving these existing search procedures the minimal key structure search basis has been extended by:

- Positive search criteria.
  Determination of search criteria for all patterns in [15] that will occur with high probability in the implementation of the particular patterns; this leads to the inclusion of most commonly used pattern implementations into the search process.
- Negative search criteria.
  Determining of search criteria that are not allowed in context of a particular pattern, what leads to the reduction of false positive cases.

Typical search criteria that can be derived from pattern descriptions are: abstract and concrete classes, inheritance, attributes (visibility, type, name); methods (visibility, polymorphism, return type, name, parameter, abstraction), constructors (visibility, name, parameter), association-, composition/aggregation-, delegation-, and friend-relations, object generation, method calls, usage of variables and templates. For the description of search criteria the Unified Modeling Language (UML) is used similar to the description of patterns.

Table 2. Overview of current pattern search methods

Name	Tool available?	Method	Covers these Patterns	Applied to these systems	Recall	Precision
DP++ [2] (C++)	yes	Min. key struct.	Composite, Decorator, Adapter, Facade, Bridge, Flyweight, Template, Chain of responsibility	Drawing Toolkit (44 classes)	n.a.	n.a.
KT [4] (Smalltalk)	yes	Min. key struct.	Composite, Decorator, Adapter, Template, Strategy, Chain of responsibility, State, Command	4 Systems (62, 264, 46, 40 classes)	n.a.	n.a.
SPOOL [5] (C++)	yes struct.	Min. key	Template, Factory, Bridge.	2 Systems (3103 and 1420 classes), ET++ (722 classes)	n.a.	n.a.
Pat [7] (C++)	yes	Class structure	Adapter, Bridge, Composite, Decorator, Proxy	NME (9 classes), LEDA (150cl.), zApp (240 cl.), ACD (343 classes)	100%	37%
IDEA [3] (UML)	yes	Class structure	Template, Proxy, Adapter, Bridge, Composite, Decorator,	n.a. Factory, Abstract Factory, Iterator, Observer, Prototype	n.a.	n.a.
Multilevel search [1] (C++ / OMT)	yes	Class structure	Adapter, Bridge, Proxy, Composite, Decorator	LEDA, libg++, galib, groff, mec, socket, no further information	100%	35%
Fuzzy logic [10] (Java)	no	Polym. pattern def. / Fuzzy Logic	all	n.a.	n.a.	n.a.
Pattern Wizard [6] (C++)	yes	Metrics	all	3 Systems without any further information	n.a.	44%
BACK-DOOR [16]	no	Manual	all	n.a.	n.a.	n.a.

For the approach described here it is not necessary to distinguish composition and aggregation. Within the description of criteria and within search algorithms, delegation relations can be substituted for method calls and association relations can be substituted for the usage of variables, method calls, method parameters and aggregation. The new approach extends the UML by the description of uncertain and forbidden criteria, depicted in Fig. 5.

Based on the description with positive and negative search criteria it is possible to determine search algorithms for all patterns given in [15]. In detail, arguments for criteria, their graphical description, and search algorithms are explained in [9]. The explanation consists of

- description and reasons for the criteria of a particular pattern,
- graphical presentation of the criteria,
- description of the pattern search algorithm,
- estimated search difficulty degree and search algorithm run time.

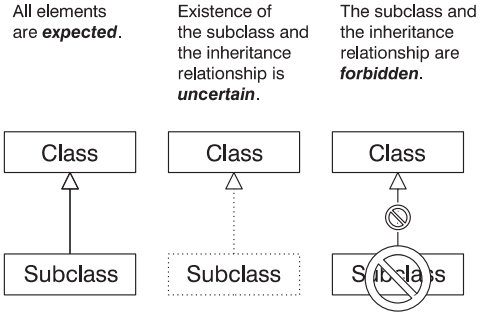


Fig. 5. Description for search criteria – example: inheritance relationship

As an example the new approach is explained using the BRIDGE pattern. The BRIDGE pattern can be used in different applications to de-couple an abstraction from its implementation so that the two can vary, see [15]. It is obvious that abstraction tree structures and implementations can be different in depth and width. In [5] it is reported that there are abstractions without specialization abstractions and implementations without common super classes. Here, a minimal key structure (positive criterion) containing one abstraction class and one implementation class, serves as search criterion.

Implementation classes usually consist of primitive operations. Methods of abstraction classes are defined using these primitive operations.

– there is no relationship from an implementation class to an abstraction class (negative criterion),
– there is no method call from an implementation class to an abstraction class (negative criterion)
– and there is a relationship (positive criterion) from an abstraction class to an implementation class.

Figure 6 shows the graphical representation of the properties of the BRIDGE pattern using the extended UML notation.

To find the BRIDGE pattern in a given model, Fig. 7 shows the corresponding search algorithm.

The approach described in this paper provides the extended UML description for all of the 23 patterns out of [15] together with a search algorithm for each of the patterns derived from this description. In Appendix A the extended UML notation and the positive/negative search criteria are listed. To understand the criteria it is necessary to be familiar with the pattern description.

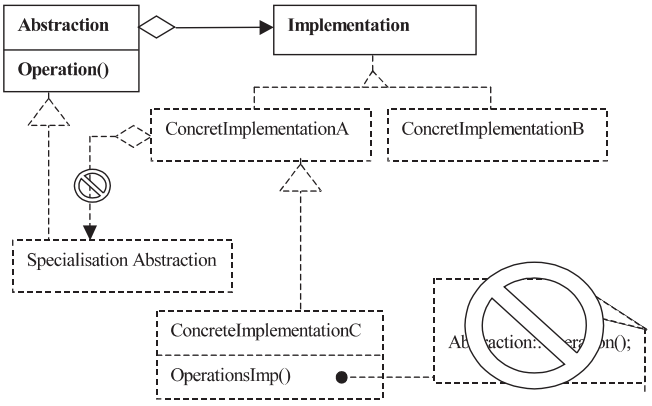


Fig. 6. Key structure of the bridge pattern

```
i:List of Abstraction classes in the model
j:List of Implementation classes in the model
x:List of classes
y,z:List of methods

FOR_ALLi
DO
 IF (current i has a reference to any j) THEN
 DO
   STORE current i with all subclasses in x;
   STORE all methods of every x in y;
   STORE all methods of every j in z;

   IF (no methods in z are also in y) THEN
   DO
            IF (none of j
                  has a reference to any x)
THEN
            DO
                 Bridge pattern recognized!
            OD
   OD
 OD
OD
Bridge pattern not recognized!
```

Fig. 7. Pseudo-code for the bridge pattern search algorithm

Table 3. Assessment of the new approach – creational patterns

Pattern	Complexity	Comments
Abstract Factory	O(n)	Identifiable without doubt, compared to [6], where Abstract Factory was recognized without being present in the analyzed software system
Builder	O(n)	Also described in [6]
Factory Method	O(n)	Comparable to [5]. The pattern is identifiable without doubt by our algorithm
Prototype	O(n)	More detailed recognition characteristics, based on our positive and negative search criteria
Singleton	O(n)	Identifiable without doubt, compared to [6], where several Singletons were identified but not present in the analyzed system

In Tables 3, 4 and 5 pattern search algorithms of the approach described in this paper are listed with their search complexity and a comment clarifying their difference or advantage compared to the existing algorithms.

The assessment of our approach is based on the student projects of Table 6 described in the next section. In the second column of Tables 3, 4 and 5 the worst case time complexity of our approach is stated by assessing the upper bounds of time consumption of our algorithms.

Table 4. Assessment of the new approach – structural patterns

Pattern	Complexity	Comments
Adapter	O(n)	None
Bridge	O(n3)	Our algorithm has additional negative search criteria, none of the evaluated algorithms can offer
Composite	O(n)	Our algorithm has additional negative search criteria, none of the evaluated algorithms can offer
Decorator	O(n2)	More detailed recognition characteristics, based on our positive and negative search criteria
Facade	O(n3)	Our algorithm has additional negative search criteria, none of the evaluated algorithms can offer
Flyweight	O(n2)	None
Proxy	O(n)	Our algorithm has additional negative and positive search criteria, none of the evaluated algorithms can offer

Table 5. Assessment of the new approach – behavioral patterns

Pattern	Complexity	Comments
Command	O(n2)	Our algorithm has additional negative search criteria, none of the evaluated algorithms can offer
Interpreter	O(n2)	Also described in [6]
Iterator	O(n)	None
Mediator	O(n2)	Also described in [6]
Memento	O(n)	Also described in [6]
Observer	O(n2)	Our algorithm has additional uncertain search criteria, none of the evaluated algorithms can offer
State	O(n2)	Our algorithm has additional negative search criteria, none of the evaluated algorithms can offer
Strategy	O(n2)	Our algorithm has additional negative search criteria, none of the evaluated algorithms can offer
Template Method	O(n)	Comparable to [5] and [6], identifiable without doubt. The pattern is identifiable without doubt by our algorithm
Visitor	O(n)	Also described in [6]
Chain of responsibility	O(n2)	None

Table 6. Student projects

Project	LOC	Classes	Patterns
Web-Based Course Enrolment System	1800	23	Singleton, Factory, Facade
Graphical Editor	200	14	Singleton, Strategy
Pattern Search	6000	147	All 23 patterns

3.2 Prototype implementation

The procedure of pattern detection, depicted in Fig. 8, consists of three basic parts:

- source code based extraction of the UML diagrams,
- pattern search and
- pattern representation.

The prototypical implementation has been realized for C++ source code using the Rational Rose and Together case tool. The Rational Rose C++ Analyzer enables UML diagram extraction out of C++ source code. Using the Rose Extensibility Interface, shown in Fig. 9 and described in [12] it is possible to access single UML model elements by either Rational Rose Automation or Rational Rose Script. For the prototype, pattern search algorithms are formulated using the Rational Rose Script Language.

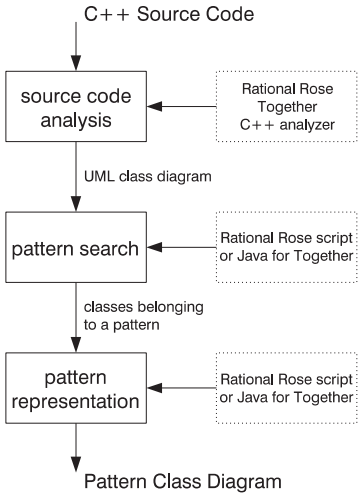


Fig. 8. Implemented pattern search procedure

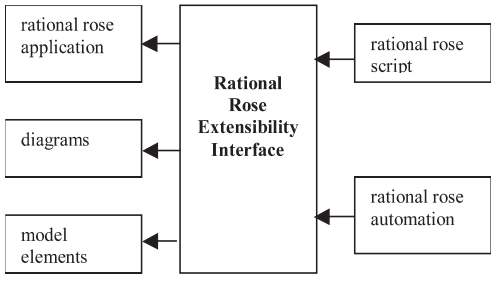


Fig. 9. Rose extensibility components

The static aspects of the source code are represented as a tree of classes or, in other words, a class diagram. Using the access possibilities of of Rational Rose or Together, we can move through the tree, request sub trees or lists of elements defined by constraints. In addition it is possible to search for elements or relationships defined in the UML model.

Compared to the Rational Rose Extensibility Interface, add-ins for Together are fairly easy to implement, since Together as well as the add-ins are Java coded. The reverse engineering capabilities of Together are better than those of Rational Rose. As a result of the prototypical implementation, the C++ Rational Rose Analyzer has turned out to be insufficient due to the fact that some relevant properties (aggregation-, delegation-, friend-relation, object generation, usage of variables, method call and template) are missing in Roses automatic reengineering capabilities. Thus, only patterns not dependent on the aforementioned properties have been prototypically implemented and tested. Model elements of recognized pattern classes will be marked in the model using colors, depicted by the pattern representation module in Fig. 8.

Our approach provides search algorithms for all of the 23 patterns of [15]. Algorithms for the SINGLETON, INTERPRETER and COMPOSITE pattern have been implemented using Rational Rose Script. SINGLETON and INTERPRETER were successfully tested with a precision value of 100% and a recall of 100%. Both algorithms were tested using code examples of student projects, as shown in Table 6.

The Web-Based Course Enrollment System is in use by our faculty to give the students the opportunity to sign in for courses, check their schedule or change their schedule. The project was developed as part of a lecture for graduate computer science students. The students were asked to develop the system by using modern software engineering methods, with patterns being one of them. In addition, a documentation including the patterns used, was written. The Graphical Editor was the result of a single student project, to present strengths of pattern programming. The last project, Pattern Search, was developed solely to be used with the pattern search algorithms. All 23 patterns were implemented using Visual C++ without any add-ins for pattern creation. All patterns were implemented just using the description of [15].

The COMPOSITE pattern could not be tested exhaustively, due to the limitations of the used CASE tool. For a full implementation further effort will be put into the integration of the Gen++ tool described in [5]. The above stated values for precision and recall are based on a search of patterns in small student projects. Thus, they are just marginally comparable with other values elaborated with other source code. In essence, a reference project, containing all patterns of [15] in different flavors with a full documentation is needed to produce comparable values with all approaches.

3.3 Conclusion

In this paper a new approach for the automated pattern search in software systems is proposed. The pattern search is oriented according to the human way of searching. The approach is based on similar approaches that are described in Sect. 3.1. These approaches have been extended by positive and negative search criteria, leading to new search algorithms for the standard patterns described in [15]. Using this approach, false positive results can be avoided and the precision value can be improved, based on the results of the student projects. Future research efforts will be spent on the evaluation of the method with bigger software systems.

The experience with the prototypical implementation of the approach confirmed the need for further human interaction, already stated by existing approaches. Thus, human-oriented search procedures are suitable. In addition, source code should contain pattern-oriented documentation. Programming languages have to be extended for pattern-oriented design and implementation. For this purpose the suitability of OpenJava [13] should be evaluated. The pattern search evaluation requires highly qualified developers familiar with pattern structures and pattern application.

The automated pattern search strongly depends on the quality of source code analysis tools. At least they should be able to extract the search criteria out of the source code, as listed in Appendix A.1.

Our new approach is still subject an ongoing evaluation process with another CASE tool, due to missing reengineering capabilities of Rational Rose. Currently we are in contact with a tool vendor for the integration of the algorithms of our approach into their CASE tool. In addition, a reference source code example is going to be established to get comparable results with all algorithms.

Appendix : Search structures for patterns

Here, the extended UML notation and the positive/negative search criteria for all patterns are listed. To understand the criteria it is necessary to be familiar with the pattern description.

A.1 Creational patterns – ABSTRACT FACTORY

- Search for a concrete factory.
- A concrete factory has at least two methods contained in the definition of the factory method pattern.

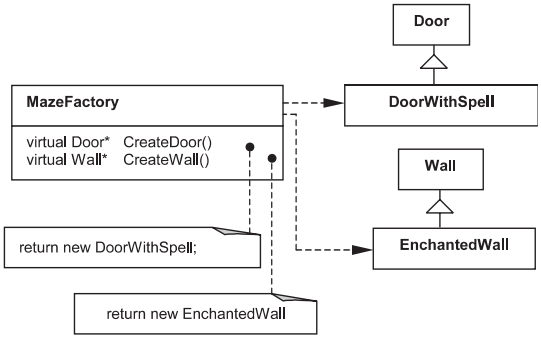


Fig. 10. ABSTRACT FACTORY search criteria

A.2 Creational patterns – BUILDER

- Search for a concrete builder.
- The concrete builder has a method returning the complete product.

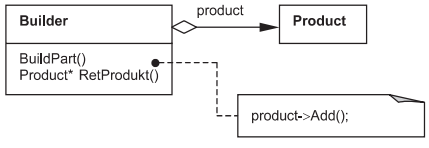


Fig. 11. BUILDER search criteria

– Builder has an aggregation relation to the product.
– The concrete builder has at least one construction method referring to the reference variable of the product.

A.3 Creational patterns – FACTORY METHOD

– Search for a concrete generator containing a factory method.
– The factory method is virtual.
– The factory method generates an object of another class (concrete product); the return type is a class (abstract product) differing from the generated object.
– As return type (abstract product), the super class of the created object is used.

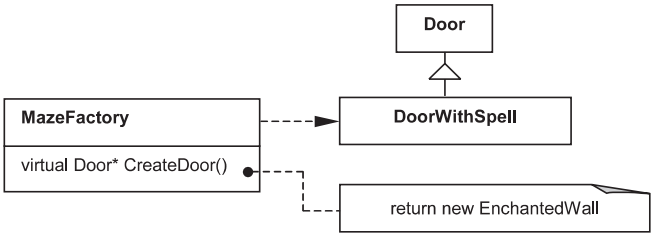


Fig. 12. FACTORY METHOD search criteria

A.4 Creational patterns – PROTOTYPE

– Search for a clone operation of a concrete prototype.
– The clone operation generates an object of the own class using its copy constructor.
– A copy constructor has to be available.
– The return type of the clone operation is the own class or a super class.
– The clone operation is virtual.

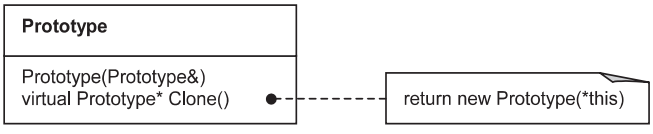


Fig. 13. PROTOTYPE search criteria

A.5 Creational patterns – SINGLETON

– Search for a singleton class.
– The class does not have a public constructor, it has only a private or protected constructor.
– The class has a static exemplar operation; the return type is the own class or a super class.
– There is a declaration of a static variable of the own class or a super class type.

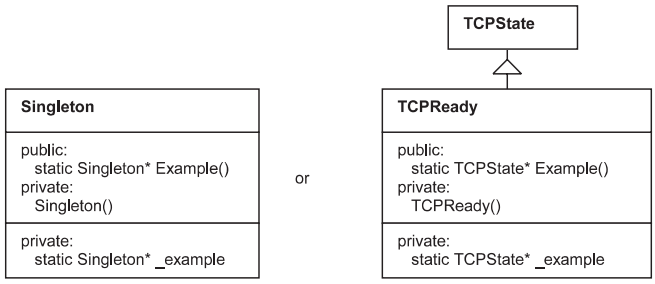


Fig. 14. SINGLETON search criteria

A.6 Structural patterns – (Class) ADAPTER

– Search for an adapter.
– The adapter inherits from two classes: from the first class public (destination) and from second class private (adapted class).
– Adapter overwrites at least one operation of the destination class; this operation calls an operation of the adapted class that is polymorphic and declared virtual.

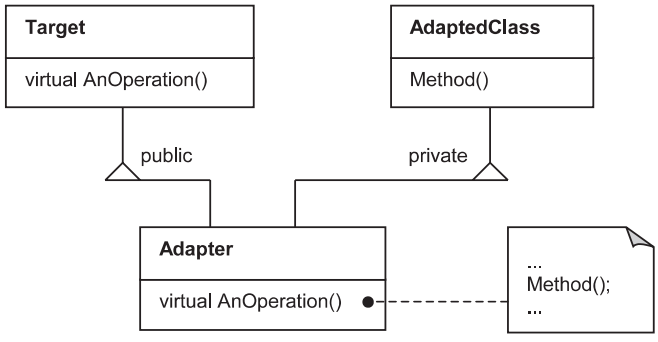


Fig. 15. (Class) ADAPTER search criteria

A.7 Structural patterns – (Object) ADAPTER

– Search for adapter class.
– Adapter is a subclass of another class (destination).
– Adapter has a reference to the adapted class.
– Adapter overwrites at least one method from destination (virtual declaration); this method calls a method from the adapted class.

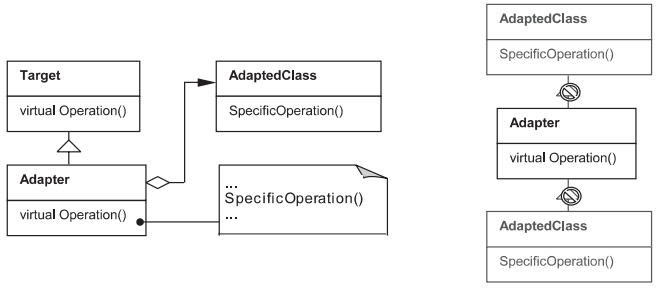


Fig. 16. (Object) ADAPTER search criteria

– Adapter is not a super or sub class from the adapted class.

A.8 Structural patterns – BRIDGE

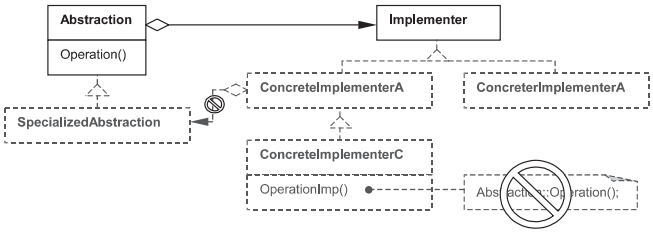


Fig. 17. BRIDGE search criteria

was described above.

A.9 Structural patterns – DECORATOR

– Search for decorator class.
– There is a 1-to-1 aggregation to a super class.
– Decorator has at least one sub class (concrete decorator).
– Concrete decorator has a method that calls decorator::operation(); in this method a local operation is called.
– The method decorator::operation() calls a method of the component class with the same name.

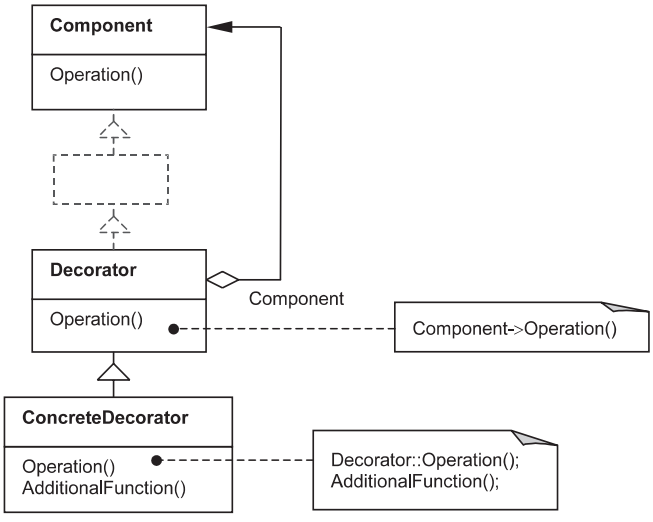


Fig. 18. DECORATOR search criteria

A.10 Structural patterns – FACADE

– Search for facade.
– A set A of classes has references to facade.
– Facade has references to a sub system (set B).
– The sub system classes don't know the facade.
– The sub system classes don't know the classes of set A.

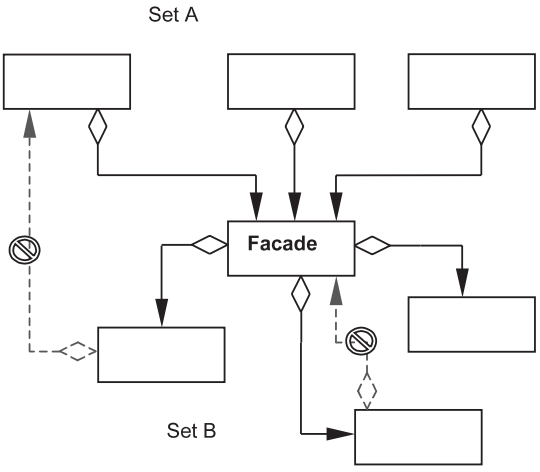


Fig. 19. FACADE search criteria

A.11 Structural patterns – FLYWEIGHT

– Search for three classes: flyweight factory, flyweight and concrete flight weight.
– The factory uses methods returning exactly what they are generating.
– The factory has a 1 to n reference to the flyweight class.
– All operations of the flyweight class always receive a particular parameter (extrinsic state). Methods can also receive additional parameters.
– The concrete flyweight is a subclass of flyweight; it is generated by the factory.

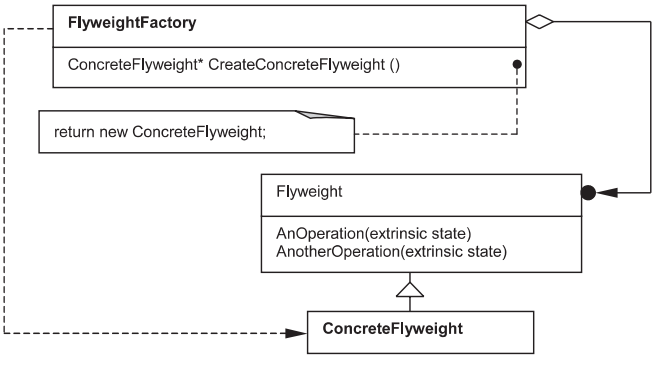


Fig. 20. FLYWEIGHT search criteria

A.12 Structural patterns – COMPOSITE

– Search for a composite class.
– A composite class has a 1 to n aggregation relation to one of its super classes (component).
– Existing sub classes won't add functionality to the composite class, which means, they don't call methods of the composite class followed by an own method.

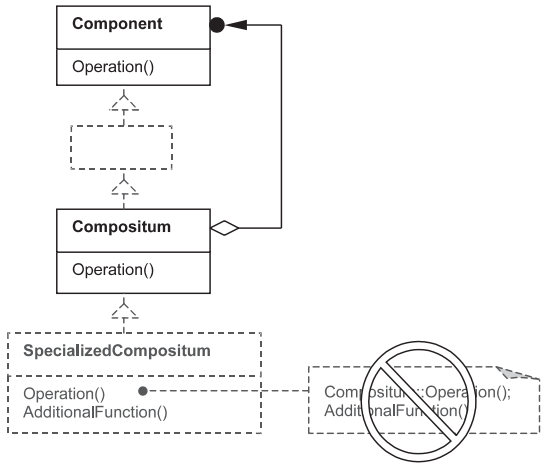


Fig. 21. COMPOSITE search criteria

A.13 Structural patterns – PROXY

- Search for proxy.
- Proxy is a sub class.
- Proxy has a reference to a class of a real subject or only a subject.
- All public methods of proxy are existent in the class that is referenced by proxy.
- In each of these proxy methods there is a call of the method with the same name in the referenced class.

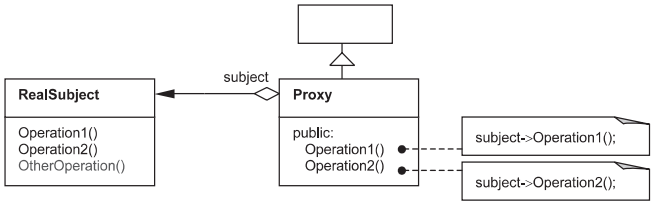


Fig. 22. PROXY search criteria

A.14 Behavioral patterns – COMMAND

- Search for a structure out of several classes: the caller, the abstract command, the concrete command, the client and the receiver.
- The command class is abstract.
- The caller has a 1-to-1-aggregation relation to command.
- Concrete command is a sub class of command.
- Concrete command has a reference to his receiver. The receiver will be passed as parameter of the constructor of the concrete command.
- There is a client instantiating the concrete command.

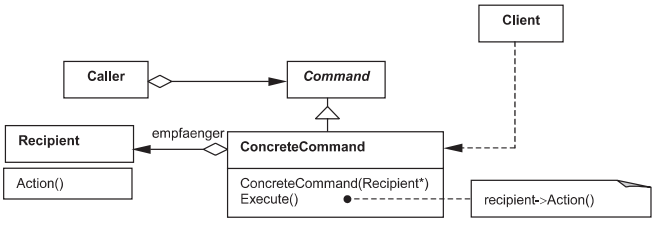


Fig. 23. COMMAND search criteria

A.15 Behavioral patterns – OBSERVER

- Search for subject, observer and concrete observer.
- Subject has a 1-to-n reference to observer.
- Subject has two methods receiving observer as parameter.
- Concrete observers are subclasses of observer.
- A concrete observer has a reference to subject or a sub class of subject (concrete subject).

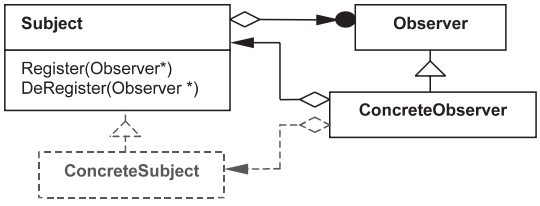


Fig. 24. OBSERVER search criteria

A.16 Behavioral patterns – VISITOR

- Search beginning with the visitor class.
- A visitor has operations receiving the elements of other classes as parameter.
- Each of these element classes has a method for receiving the visitor class as parameter.
- Within this method of the element class, a call of the corresponding method of the visitor class is done; parameter is the element itself.

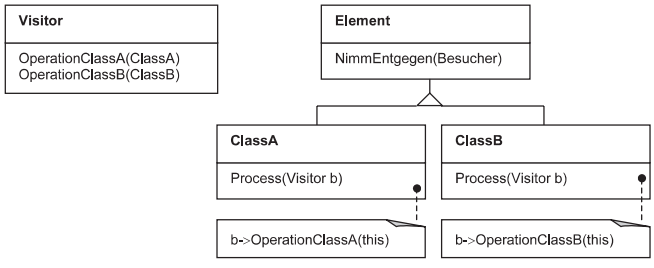


Fig. 25. VISITOR search criteria

A.17 Behavioral patterns – INTERPRETER

- Search for a tree.
- The root class is abstract.

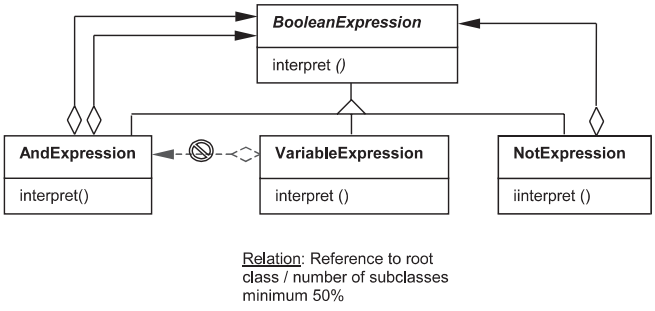


Fig. 26. INTERPRETER search criteria

- Each sub class implements one of the methods always as a new method.
- The ratio of simple aggregation relations to the root class divided by the number of subclasses is at least 50%.
- Sub classes don't reference each other directly.

A.18 Behavioral patterns – ITERATOR

- Search for two templates: list and iterator.
- Iterator has a reference to the list.
- List generates the iterator within a method.

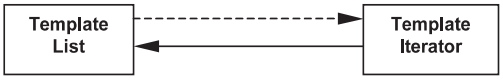


Fig. 27. ITERATOR search criteria

A.19 Behavioral patterns – MEMENTO

- Search for memento.
- Memento doesn't have a public constructor.
- Memento is generated by an originator class.
- Memento has a method for setting its state and a method for returning its state.
- The originator is allowed to access the private interface of memento (friend class declaration).
- The generator doesn't have a reference to memento.
- A container is handling a reference to memento without generating it.

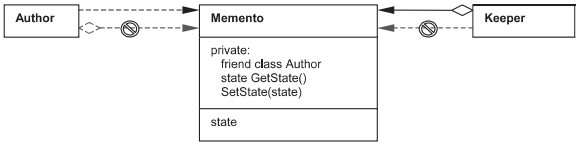


Fig. 28. MEMENTO search criteria

A.20 Behavioral patterns – TEMPLATE METOD

- Search for template method.
- The template method is not polymorph.

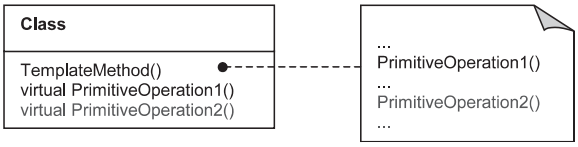


Fig. 29. TEMPLATE METOD search criteria

- The template method calls at least one local polymorph method.

A.21 Behavioral patterns – STRATEGY

- Search for a tree of strategy classes.
- The root class (abstract strategy) is abstract.
- All sub classes (concrete strategies) have the same public interface as their abstract super class.
- No concrete strategy holds a reference to the abstract strategy.
- No concrete strategy holds reference to another concrete strategy.
- The compositor holds a reference to the abstract strategy but not to concrete strategies.

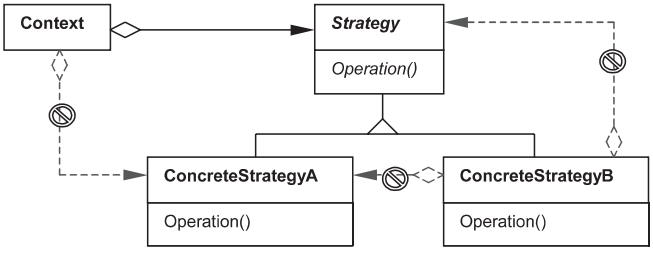


Fig. 30. STRATEGY search criteria

A.22 Behavioral patterns – MEDIATOR

- Search for a concrete mediator.
- A concrete mediator has references to its concrete colleagues.
- Concrete colleagues don't have references between each other.
- If there is an abstract colleague, the object handles a reference to the abstract mediator or directly to

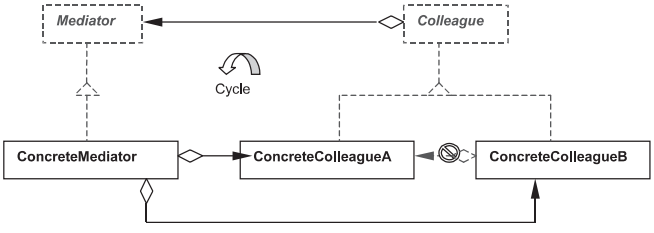


Fig. 31. MEDIATOR search criteria

the concrete mediator, in case there is no abstract mediator.
– If there is no abstract colleague, each concrete colleague is handling a reference to the abstract mediator or direct to the concrete mediator, in case there is no abstract mediator.

A.23 Behavioral patterns – STATE

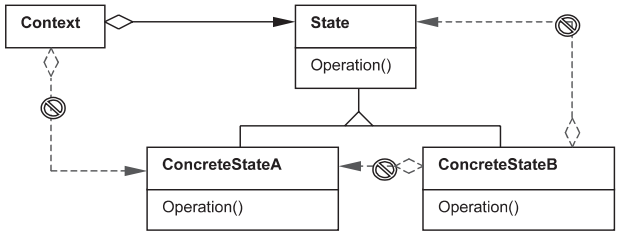


Fig. 32. STATE search criteria

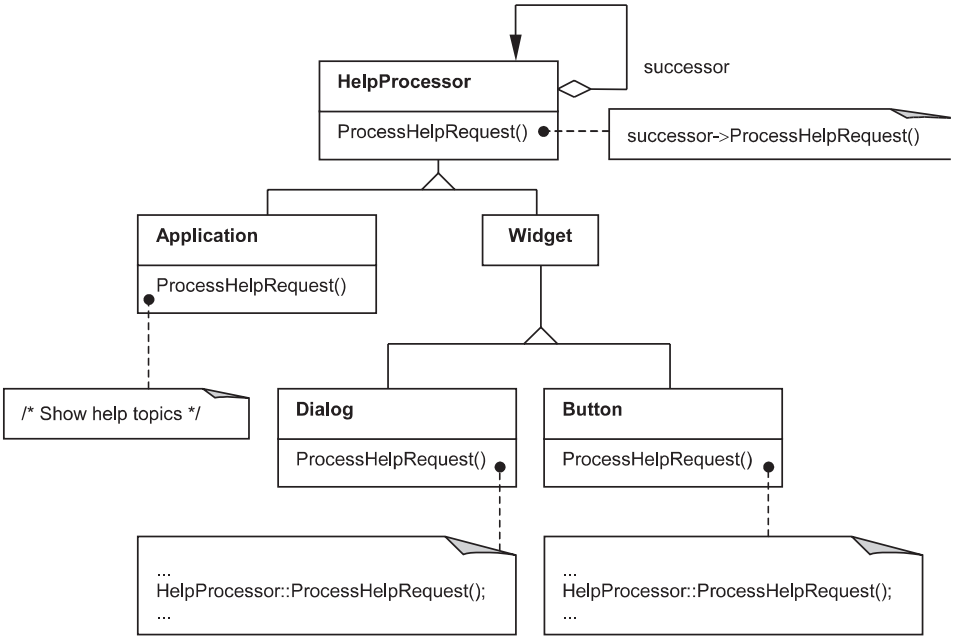
– Search for a tree of state classes.
– The root class (state) is not abstract.
– All sub classes (concrete states) have the same public interface as their super class.
– No concrete state holds a reference to the root class.
– No concrete state holds a reference to another concrete state.
– In context there is a reference to the root class but not to the concrete states.

A.24 Behavioral patterns – CHAIN OF RESPONSIBILITY

– Search for a tree.
– The root class is a HelpHandler.
– HelpHandler implements the HandleHelp method.
– The HandleHelp method won't be overwritten in all sub classes, but it is overwritten mostly inside the tree structure.
– Classes can be divided in two categories, based on the HandleHelp method. For this categorization only classes overwriting the HandleHelp method will be considered.
– The first category contains all classes forwarding the own HandleHelp method to HandleHelp methods of other classes.
– The second category contains all classes without forwarding mechanism within the HandleHelp method.
– At least 75% of the classes have to belong to the first category.

References

1. Antoniol G, Fiutem R, Cristoforetti L (1998) Design pattern recovery in object-oriented software. In: 6th International Workshop on Program Comprehension, June, pp 153–160
2. Bansiya J (1998) Automatic Design-Pattern Identification. Dr. Dobb's Journal. Available online at: `http://www.ddj.com`
3. Bergenti F, Poggi A (2000) Improving UML design using automatic design pattern detection. In: Proc. 12th. International



- *ProcessHelpRequest()* is the most overwritten operation of the root class in the tree.
- At minimum 75% of the classes overwriting *ProcessHelpRequest()* will realize this by forwarding the request.

Fig. 33. CHAIN OF RESPONSIBILITY search criteria

Conference on Software Engineering and Knowledge Engineering (SEKE 2000), pp 336–343
4. Brown K (1996) Design reverse-engineering and automated design pattern detection in Smalltalk. Master's thesis. Department of Computer Engineering, North Carolina State University. Available online at: http://www.ncsu.edu/
5. Keller RK, Schauer R, Robitaille S, Page P (1999) Pattern-based reverse engineering of design components. In: Proc. Of the 21st International Conference On Software Engineering. IEEE Computer Society Press, pp 226–235
6. Kim H, Boldyreff C (2000) A method to recover design patterns using software product metrics. In: Proc. of the 6th International Conference on Software Reuse (ICSR6). Available online at: http://www.soi.city.ac.uk/∼hkim69/publications/icsr6.pdf
7. Kraemer C, Prechelt L (1996) Design recovery by auto-mated search for structural design patterns in object-oriented software. In: Proc. of the Working Conference on Reverse Engineering, pp. 208–215
8. Salton G, McGill M (1983) Introduction to Modern Information Retrieval. McGraw-Hill, New York
9. Naumann S (2001) Reverse Engineering of Design Patterns. Diploma Thesis. Technical University of Ilmenau (in German)
10. Niere J, Wadsack JP, Wendehals L (2001) Design pattern recovery based on source code analysis with fuzzy logic. Technical Report tr-ri-01-222, University of Paderborn. Available online
11. Prechelt L, Unger B, Philippsen M (1997) Documenting Design Patterns in Code Eases Program Maintenance. In: Proc. Of the ICSE Workshop on Process Modeling and Empirical Studies of Software Evolution, pp 72–76
12. Rational (2000) Using the Rose Extensibility Interface Rational Rose 2001 Rational Rose Software Corporation
13. Tatsubori M, Chiba S (1998) Programming Support of Design Patterns with compile-time Reflection. OOPSLA 98 Workshop Reflective Programming in C++ and Java, Vancouver, Canada
14. West R (1993) Reverse Engineering – An Overview, HMSO, London
15. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns – Elements of Reusable Object-Oriented Software, Addison Wesley
16. Shull F, Melo WL, Basili VR (1996) An inductive method for discovering design patterns from object-oriented software systems. Technical Report UMIACS-TR-96-10, University of Maryland

Ilka Philippow is a full Professor for Process Informatics at the Ilmenau Technical University since 1992.

In the eighties she was working in the field of software development for technical and embedded systems. She received the PhD in Computer Science in 1981 and finished her habilitation in 1989 at the Technical University of Ilmenau.

During the last decade she has achieved experience in reqirements engineering and modeling for complex software systems and tool supported software reuse. She was leading involved in the development of the modeling tool OTW that enhances the UML ability with the Object-Process-Model (OPM). OPM can be used for simulation of software processes as well as for business processes. Her research activities are oriented on the evolutionary development of software and software product lines, UML model based pattern recognition and UML model based test case generation. The latest activities are focussed on feature based supported architecture recovery and restructuring.

She is working very close with industrial partners like Siemens AG in Munich and Daimler-Crysler Research Center in Ulm.

Her previous position include: Dean of the Faculty for Informatics and Automation (1995–1998) and Vice President of the Ilmenau Technical Unicersity (1998–2000).

Detlef Streitferdt is a PhD student at the Technical University of Ilmenau, Germany. He received his degree in Computer Science from the University of Stuttgart, Germany. As part of his diploma studies he spent a year studying Computer Science at the University of Waterloo in Canada.

His current research interests include requirements engineering for software product lines and pattern recognition in reverse engineered UML models. He was program co-chair of the "Modeling Variability for Object-Oriented Product Lines" – workshop at the 17th European Conference on Object-Oriented Programming (ECOOP).

Matthias Riebisch is an Assistant Professor at the Technical University of Ilmenau. He holds a degree in Automation Engineering (Dresden Technical University). Matthias Riebisch received his PhD from the Technical University of Ilmenau, Germany.

His current research interests include Software architectures, object-oriented modelling, software reusability, evolvability of software, software development processes, architecture and modelling of information systems, maturity of software development processes, methods and organizations, software quality management, best practice management and information security.

Current reserach projects include methods for the development of software product lines ("Alexandria"), UML-based test case generation in cooperation with the University of Central Florida and sponsored by the DFG (Deutsche Forschungsgemeinschaft).

He was workshop chair of the "Modeling Variability for Object-Oriented Product Lines" – workshop at the 17th European Conference on Object-Oriented Programming (ECOOP).

Sebastian Naumann studied Computer Sciences at the Technical University of Ilmenau, Germany between 1996 and 2001. Since 2002 he has been working at the Institut f. Automation und Kommunikation e.V. Magdeburg (ifak) as a scientific employee in the working group Intelligent Transport System.