Design Pattern Recovery in Architectures for Supporting Product Line Development and Application

Ilka Philippow, Detlef Streitferdt, Matthias Riebisch
Technical University of Ilmenau
ilka.philippow[matthias.riebisch, detlef.streitferdt]@.tu-ilmenau.de

Abstract. Product lines can improve the efficiency of software development. Product lines offer reference architecture for the development of similar products. This architecture is developed in an evolutionary process while using existing systems and reusable components. The start of the development of product lines very often is based on the reengineering and generalization of several similar existing applications. Design Patterns can support the understanding of former architectures and the application of product line reference architectures. In this paper a short explanation of the development and application of product lines points out the relevance of patterns. The paper discusses existing pattern search methods and describes an approach suitable for the automated search. This approach enlarges the existing search criterion based methods for pattern recognition for the automated detection of all Gamma Patterns.

1 Introduction

The complexity of software systems has increased extremely during the last decade. Meanwhile, there exist various methods and tools in order to support the development and management of very large and complex software systems. A high degree of software reuse offers possibilities for reducing development efforts and improving software quality. Most reuse approaches are based on object-oriented technology. There are different kinds of reuse, e.g. source code in form of modules, functions, classes or components and other artifacts related to analysis, design, and architectures. The reuse of architecture artifacts is closely connected with the application of patterns [1], [2], [3].

For similar software products the software development based on product lines is connected with expectations for enhancements in reusability, adaptability, flexibility, and control of complexity and performance of software. A software product line is a “group of products” from a specific problem domain [4]. They are based on a system family architecture offering a “common set of core assets” [5]. In the range of a specific problem domain software product lines are derived from predefined architectures these architectures consist of common and variable parts. Variable parts can be changed or adapted to satisfy the special needs of an application.

In section 2 the development process of reusable architectures for product lines is explained in a simplified way to point out the relevance of architecture reengineering. The understanding of architectures and the reuse of architecture artifacts

can be supported by the application of design patterns. In section 3 an approach for detecting design patterns is introduced.

2 The Development and Application of Product Lines

The evolutionary process of the development of product line architecture is described in [6]. The process phases and activities are similar to those of software development in general, but they have to pay more attention to aspects like domain requirements, reuse and configuration ability (Fig. 1).

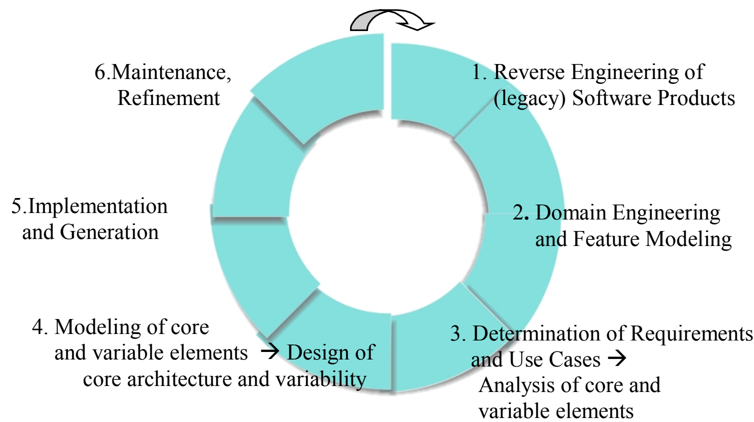

Fig. 1 Iterative Activities of the Development Process for Software Product Lines

The decision to build a product line and the development starting point very often is based on the generalization of several similar applications [7], [8] and the reengineering of legacy software. The phases 1-3 serves for the determination of product line requirements. During the phases 4-6 the product line architecture has to be designed and implemented. New costumer requirements lead to further iterations.

Fig. 2 illustrates the relations between the development of a software product line and the application development. Every new application is based on the product line reference architecture. Summarizing it can be said, the evolutionary development process of product lines and its application is characterized by the following activities for reuse, refinement and improvement:

- reverse-engineering and understanding existing application architectures,
- comparing new requirements to the former ones,
- creating a new design, including both the new and the former requirements,
- redesigning the architecture and implementing new common and variable parts

- Documenting design decisions, intentions and the new architecture for future refinements.

The process of development and application of product lines can be supported by a pattern based development of the reference architecture. On condition that patterns had been used for the development of former applications these patterns can be found by the reverse-engineering activities. Pattern helps to understand former applications and serves for the determination and description of the product line reference architecture. To integrate pattern into the development and application process of product lines it is necessary not only to apply known standard patterns e.g. [1] but also to recognize and determine domain specific patterns based on the family requirements. During the development process of product lines defined and used domain and standard patterns have to be documented. During the application of product lines these patterns help to understand the product line architecture and applications based on it. To simplify this procedure it is necessary to support the search and detection of patterns automatically.

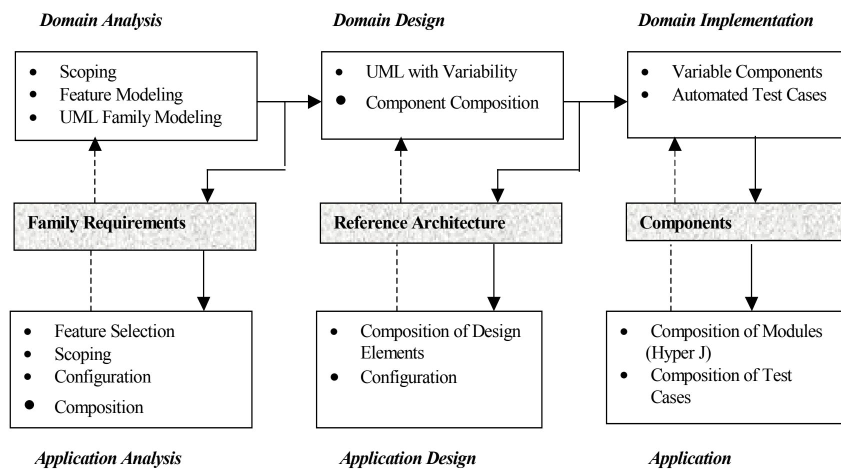

Fig. 2 Relations between Product Line and Application Development

3 An approach for the Automated Detection of Patterns

3.1. State-of-the-Art

There are various methods for automated pattern identification. They are evaluated according to the achieved results of their search algorithms that can lead to three relevant results:

1. *Positive true:* a pattern has been recognized and the pattern is really implemented within the software system (case is desired).
2. *Positive false:* a pattern has been recognized but the pattern is not really implemented within the system. (Case has to be avoided).
3. *Negative true*: a really existing pattern has not been recognized (case has to be avoided).

The achieved results can be used for evaluation of searching tools by two metrics Both metrics are known for evaluating search results, e.g. in Information Retrieval [9].

- *Recall* is the number of all existing patterns in a software system divided by the number of recognized patterns. A recall of 100% means that all existing patterns were found (negative true cases has been avoided).
- *Precision* is the number of really existing and recognized patterns divided by the number of found patterns (sum of the results positive true and positive false). A precision of 50% means, that half of the recognized patterns are not really implemented in the software system.

Both values have to taken into consideration for a tool evaluation. A precision value of 100% does not exclude *negative true* cases. Several existing approaches for automated pattern search have been evaluated, together with available information about the above-explained metric values. There are different approaches for an automated pattern search that can be categorized by four different search algorithms:

Searching for minimal key structures: The properties of a defined key structure that is assigned to a particular pattern are used as search criteria. There are three approaches: DP++ [9] for C++, KT [10] for Smalltalk and SPOOL [11] realized for C++, applicable for Java and Smalltalk.
Searching for class structures: The search is based on the pattern class structures described in [1]. There are three approaches for automated search based on complete accordance of the classes are known: Pat [12] for C++, IDEA [13] for UML diagrams and the multi step search tool in [14].
Searching based on fuzzy logic: This approach considers structural differences between patterns out of [1] and real life software systems. The proposed idea [15] uses fuzzy logic search algorithms to examine different pattern implementations. The researchers are working on the development and implementation of their idea.
Searching based on metrics: In this approach [17] each pattern is characterized by metrics: There are three categories of metrics with examples for each category:

(a) Object oriented Metrics
- weighted methods per class
- depth of inheritance tree
- number of children (subclasses)
- coupling between objects

(b) 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

(c) Procedural Metrics
- pure lines of code
- McCabe's cyclomatic complexity
- lines of comments

These metrics are calculated tool-based for the system in question and for every of the desired patterns.

In [17] a manual search method is described. This method proposes six steps for the finding of 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 very 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.

For a more detailed description of the mentioned approaches see the given references. A comprehensive discussion and comparison of the various approaches is carried out in [18]. The evaluation of existing approaches is based on the information from the papers describing the individual methods. Table 1 summarizes the results of current pattern searching research efforts. Just one of the tool supported approaches has the potential to find all of the 23 patterns described in [1], although with a not satisfying precision value of about 44%. The other approaches are only usable for a subset of the patterns. Search algorithms for minimal key structures are considered to be closest to the human way of thinking. To follow the way of human thinking is very important during the evaluation of search results by software developer for the elimination of e.g. negative true cases. But, in accordance to [10] and [11] it was not possible to find reliable search criteria for all patterns. In the next section an approach is introduced that is focussed on the extension of algorithms for minimal key structure based search described in [9], [10] and [11].

3. 2. Extended Approach for Pattern Search based on minimal Key Structures

For improving the existing search procedures the minimal key structure search basis has been enlarged by the definition of further positive search criteria and, in addition negative search criteria for all patterns in [1]:

- *Positive search criteria* will occur with very high probability during the application of a particular pattern (for commonly used pattern implementations)
- *Negative search criteria* mustn't occur in context of a particular pattern; this leads to the reducing of *positive false* cases

The typical search criteria can be derivated from the common and accepted pattern description in [1]. Examples for search criteria are

- abstract and concrete classes, inheritance relations,
- attributes (visibility, type, name),
- methods (visibility, polymorphisms, return type, name, parameter, abstraction),
- constructors (visibility, name, parameter),
- relations like association, composition, aggregation, delegation, object generation, method calls, variable usage and template usage.

Considering the very high acceptance of the UML (Unified Modeling Language) in practice in this paper we have described patterns by UML diagrams instead of the OMT used form Gamma. For this purpose the UML class diagram has been enlarged for the description of uncertain or forbidden criteria (Fig. 3).

Table 1: Overview of Current Pattern Search

Minimal key structures:
DP++ [9] for C++: tool available - Covers the following patterns: Composite, Decorator, Adapter, Façade, Bridge, Flyweight, Template, Chain of responsibility - Applied to the following system: Drawing Toolkit (44 classes) - Recall: n.a. Precision: n.a.
KT [10] for Smalltalk: tool available - Covers the following patterns: Composite, Decorator, Adapter, Template, Chain of responsibility, Strategy, State, Command - Applied to four different systems (62, 264, 46, 40 classes) - Recall: n.a. Precision: n.a.
Spool [11] for C++: tool available - Covers the following patterns: Template, Factory, Bridge - Applied to two different systems (3103, 1420 classes) and to ET++ (722 classes) - Recall: n.a. Precision: n.a.
Class Structure:
PAT[12] for C++: tool available - Covers the following patterns: Adapter, Bridge, Proxy, Composite, Decorator - Applied to the following systems NME (9 classes), LEDA (150 classes), zApp (240 classes), ACD (343 classes) - Recall: 100% Precision: 37%
IDEA [13] for UML: tool available - Covers the following patterns: Template, Adapter, Bridge, Proxy, Composite, Decorator, Factory, Abstract Factory, Iterator, Observer, Prototype - Applied systems: n.a. - Recall: n.a. Precision: n.a.
Multi level search [14] for C++/OMT: tool available - Covers the following patterns: Adapter, Bridge, Proxy, Composite, Decorator, - Applied for different systems: LEDA, libg++, galib, groff, mec, socket (no further information) - Recall: 100% Precision: 35%
Fuzzy logic based search:
for Java [15]: tool not available - Covers the following patterns: all - Applied systems: n.a. - Recall: n.a. Precision: n.a.
Metric oriented pattern search:
wizzard [16] for C++: tool available - Covers the following patterns: all - Applied systems: three systems without further inform. - Recall: n.a. Precision: 44%
Manual pattern search:
Backdoor [18]: tool not available - Covers the following patterns: all - Applied systems: n.a. - Recall: n.a. Precision: 44%

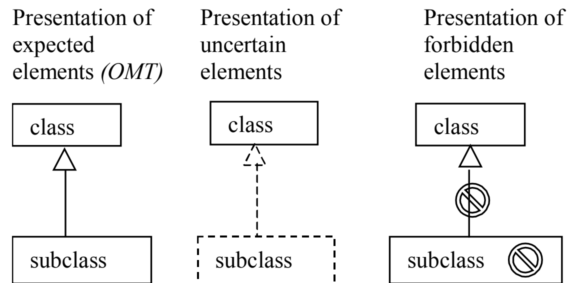

Fig. 3. Description Elements for Search Criteria (Example: Inheritance)

The full description and discussion of the necessary search criteria for all patterns from [1] is given in [18]. The description contains the arguments and reasons for chosen criteria, the extended class diagrams for patterns (graphical presentation), and the description of the criteria based search algorithms.

In this paper the BRIDGE pattern is chosen as an example for the demonstration of the proposed pattern description and search approach. Depending on the particular application using the Bridge pattern the tree structure can be very different in depth and width. In practice (reported in [11]) it is possible that abstractions exist without specialization abstractions and implementations without super classes. The proposed minimal key structure (Fig. 4) consists of one abstraction class and one implementation class as positive search criterion. Usually, an implementation class contains primitive operations. The methods of abstraction classes are defined using these primitive operations [1].

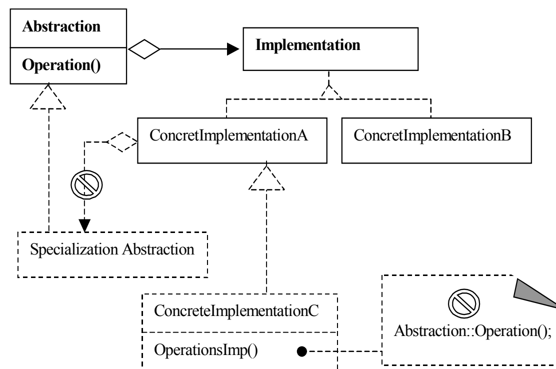

Fig. 4 BRIDGE Pattern –Minimal Key Structure

The following set of search criteria should be enough to find a BRIDGE pattern:
positive criterion:

- there is a relationship from an abstraction class to an implementation class

negative criteria:

- there is no method call from an implementation class to an abstraction class
- there is no relationship from an implementation class to an abstraction class

In accordance to these criteria the graphical description is given in Fig. 4. Using the following algorithm a BRIDGE pattern can be identified:

```
x:     List of classes
y,z:   List of methods
i:     List of abstraction classes in the model
j:     List of implementation classes in the model

FOR_ALL  i
DO
  IF (current i have 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 was found
                   OD
               OD
          OD
   → BRIDGE Pattern was not found
```

In [18] for all patterns from [1] the concrete search algorithms are determined. Table 2 contains the determined search criteria and comments concerning the difference or advantage to the existing approaches mentioned in section 3.1.. To understand the criteria it is necessary to be familiar with the pattern description in [1].

A prototypical implementation has been carried out by using the Rational Rose CASE tool. The application of the Rational Rose C++ Analyzer enables to extract UML diagrams out of source code. The Rose Extensibility Interface [19] is used to access single UML model elements. The search algorithms are implemented by using the Rational Rose Script language The prototypical implementation has shown that the Rational Rose C++ Analyzer is insufficient due to missing reengineering abilities for finding object generation, delegation, aggregation- and friend-relation, usage of variables, methods and templates. Therefore, a successful implementation of search algorithms was only possible for patterns that are not depending on these not identifiable search properties. The search algorithms for COMPOSITE, SINGLETON and INTERPRETER had been implemented. For SINGLETON and INTERPRETER the achieved values for precision and recall are 100%. These values are based on the application in small student projects. To achieve really comparable values to other approaches a unified reference architec-

ture would be necessary. Due to the limitations of the used case tool the further effort will be put on the integration of a better case tool.

4 Conclusion

The proposed pattern search is oriented to the human way of searching and requires software developers that are familiar with pattern structures and able to evaluate search results. The search approach is based on similar approaches for minimal key structures. These approaches have been extended by additional positive and negative search criteria, leading to new and better search algorithms. This approach improves the precision value by avoiding positive false results.

The success of search algorithms strongly depends on the quality of the source code analyzing tool that should be able to extract all the necessary search criteria.

The introduced approach for pattern search is still the subject of ongoing research and implementation effort. Currently we are in contact with a tool provider for the integration of our algorithms into their case tool. To get comparable results with other search approaches we are working on the possibility to establish a reference source code example.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
2. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse : Architecture, Process and Organization for Business Success. Addison Wesley (1997)
3. Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley (1996)
4. Kotler, P., Bliemel, F.: Marketing-Management: Analyse, Planung, Umsetzung und Steuerung. Schäffer-Poeschel, 9th edition (in german) (1999)
5. Clement, P., Northrop, L.: A framework for software product line practice, version 2.7., (1999)
6. Philippow, I.; Riebisch M.: Systematic Definition of Reuseable Architectures. Proceedings 8.th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems pp. 128-136 (2001)
7. Koskimes K., Moessenboeck H.: Designing a Framework by Stepwise Generalization. 5th European Software Engineering Conference Barcelona, Lecture Notes in Computer Science 989, Springer, (1995).
8. Riebisch, M.; Franczyk, B.: Evolutionary Development of Frameworks – from Projects to System Families IDPT 1999, Kusadasi, Turkey, June 27th – July 2nd, 1999. In: Tanik, M.M., Ertas, A.: [eds.]: IDPT 1999. Society for Design and Process Science, pp. 13. ISSN 1090-9389 (2000)
9. Bansiya, J.: Automatic Design-Pattern Identification. Dr. Dobb´s Journal, (1998). Available online at http://www.ddj

10. Brown, K.: Design reverse-engineering and automated design pattern detection in SmallTalk. Master´s thesis, Department of Computer Engineering, North Carolina State University, (1996). Available online at http://www.ncsu.edu/
11. Keller, R. K., Schauer, R. Robitaille, S., Page, P.: Pattern-based reverse engineering of design components. In Proc. Of the 21st International Conference On Software Engineering, pages 226-235, IEEE Computer Society Press (1999).
12. Kraemer, Ch., Prechelt, L.::Design recovery by automated search for structural design patterns in object-oriented software", In Proc. of the Working Conference on Reverse Engineering, pp. 208-215 (1996)
13. Bergenti, F., Poggi, A.:. Improving UML design using automatic design pattern detection. In Proc. 12th. International Conference on Software Engineering and Knowledge Engineering (SEKE 2000), pp. 336-343 (2000).
14. Antoniol, G., Fiutem, R.; Cristoforetti, L.: Design pattern recovery in object-oriented software. In 6th International Workshop on Program Comprehension, pp. 153-160, June (1998).
15. Niere, J., Wadsack, J. P., Wendehals, L.: Design pattern recovery based on source code analysis with fuzzy logic", Technical Report tr-ri-01-222, University of Paderborn, (2001). Available online.
16. Kim, H., Boldyreff, C.: A method to recover design patterns using software product metrics. In Proc. of the 6th International Conference on Software Reuse (ICSR6), (2000). on line at: http://www.soi.city.ac.uk/~hkim69/publications/icsr6.pdf
17. Forrest Shull, Walcélio L., Melo, Victor, Basili, R.: An inductive method for discovering design patterns from object-oriented software systems", Technical Report UMIACS-TR-96-10, University of Maryland (1996)
18. Naumann, S.: "Reverse Engineering of Design Patterns", Diploma Thesis, Technical University of Ilmenau (2001). (in German)
19. Rational: Using the Rose Extensibility Interface Rational Rose 2001. Rational Rose Software Corporation (2000)

Table 2: Search Criteria and Assessment of the extended approach

Search criteria	Comments
ABSTRACT FACTORY	
- Search for a concrete factory. - A concrete factory has at least two methods contained in the definition of the factory method pattern	Identifiable without doubt, compared to [7]
BUILDER	
- Search for a concrete builder. - The concrete builder has a method returning the complete product. - 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.	Also described in [7]
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.	Comparable to [6], identifiable without doubt
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.	More detailed recognition characteristics.
SINGLETON	
- Search for a singleton class. The class doesn't 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.	Identifiable without doubt, compared to [7]
(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.	Result comparable with [6], [7]
(Object) ADAPTER	
- Search for adapter class.	None.

<table>
<tr><td>- 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.
- Adapter is not a super or sub class from the adapted class.</td><td></td></tr>
<tr><th>Search criteria</th><th>Comments</th></tr>
<tr><th colspan="2">BRIDGE</th></tr>
<tr><td>- was described above</td><td>Additional negative search criteria</td></tr>
<tr><th colspan="2">COMPOSITE</th></tr>
<tr><td>- 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.</td><td>Additional negative search criteria</td></tr>
<tr><th colspan="2">DECORATOR</th></tr>
<tr><td>- 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.</td><td>More detailed recognition characteristics.</td></tr>
<tr><th colspan="2">FACADE</th></tr>
<tr><td>- Search for Façade.
- A set A of classes has a reference to façade.
- Façade has a reference to a sub system (set B).
- The sub system classes don`t know the façade.<br>- The sub system classes don`t know classes of set A.</td><td>Additional negative search criteria</td></tr>
<tr><th colspan="2">FLYWEIGHT</th></tr>
<tr><td>- 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.</td><td>None.</td></tr>
<tr><th colspan="2">PROXY</th></tr>
<tr><td>- 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.</td><td>More flexible search.</td></tr>
</table>

- In each of these proxy methods there is a call of the method with the same name in the referenced class.	
ITERATOR	
- Search for two templates: list and iterator. - Iterator has a reference to the list. - List generates the iterator within a method.	None.

Search criteria	Comments
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 most within 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.	None.
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.	Additional negative search criteria
INTERPRETER	
- Search for a tree. The root class is abstract. - 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	Also described in [7]
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 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.	Also described in [7]
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.	Also described in [7]

Search criteria	Comments
- 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.	
Search criteria	**Comments**
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).	Also described in [7]
STATE	
- 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.	Additional negative search criteria
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.	Additional negative search criteria
TEMPLATE METHOD	
- Search for template method. - The template method is not polymorph. - The template method calls at least one local polymorph method	
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. - At least 75% of the classes have to belong to the first category.	Also described in [7]