Extension for Design Pattern Identification Using Similarity Scoring Algorithm

Jaroslav Jakubík*
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technology
Slovak University of Technology in Bratislava
Ilkovičova 3, 842 16 Bratislava, Slovakia
jakubik@fiit.stuba.sk

Abstract

This paper provides studies in area of design pattern identification in existing software systems. The target was to design and prototype specific extensions of a selected method with better and more precise results. This paper summarizes separate phases of a project analysis of different design pattern representations, analysis of different methods, algorithms for identification of design patterns in software systems, selection of a concrete method for extensions, design, implementation and tests of an extended method. In this paper, the extensions like feature weighting, feature filtering and additional lexicographical analysis are described. Designed and prototyped extensions were experimentally tested on more than 3000 classes of open source systems from different software engineering areas. Based on the test results, the general conclusion for designed extensions are formulated.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics

Keywords

design pattern identification, structural methods, pattern formalization

1. Introduction

Patterns are currently used in almost all software development phases (for example analysis patterns, architecture patterns, design patterns, test patterns). A pattern describes an idea or the best practice used in many situations, in many projects. Design patterns provide solutions to recurring design problems. Design patterns as well as any other patterns are traditionally described in documentation with semiformal description. This description contains context, problem, solution and results.

In some projects, the usage of design patterns has appropriate documentation, but many projects miss this kind of documentation. During service and support phase in software development process, this kind of documentation is very important for system maintenance and system extension. When we are talking about large software (object-oriented) systems, maintenance of system without appropriate documentation is complicated and difficult. With appropriate documentation, a service team can extend the existing system quickly, fix errors and provide good quality service and support. The problem can occur when the usage of design patterns documentation is missing. Therefore, the major task for the service team is to design recover, build higher level of abstraction from source code, understand the design and architecture of the system.

Knowledge about applied design patterns leads to better understanding of design problems solved by software designers, architects when designing software architecture. Understanding of the design problems is a necessary step towards informed changes of the system and of its architecture [7]. The documentation of a design can be also created and updated for an existing software system using reengineering process. Software reengineering can be done manually or by using separate software tools and utilities. There exist many software tools for creating UML models in reengineering process, commercial or not (for example Rational rose, Enterprise architect, Rational XDE, Together). At the moment, these reengineering tools do not provide information about used design patterns (based on source code). For the extraction of design pattern information specific methods and tools must be used.

Methods for pattern identification use wide range of IT technologies for getting results in specified time and quality. Extracting information about using design patterns is based on searching equivalence between concrete design pattern and part of software system. During design pattern identification static-structural representation, dynamic-behavioral representation or both of them can be used. Static-structural analysis uses structural representation of design pattern and software system. In comparison with dynamic analysis, static analysis is relatively quick and simple. Static structure of a software system is

*Recommended by thesis supervisor: Prof. Pavol Návrat Defended at Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava on May 5, 2009.

compared with static structure of a design pattern. Static analysis is executable on source code and structural model too (for example class diagram). Dynamic part of the system which is not a part of the structural model is suppressed in this kind of analysis. Static analysis does not need to run a software system, which can be an advantage in kind of software frameworks and libraries.

Dynamic-behavioral analysis uses behavioral model (behavioral representation) of a design pattern and a software system for pattern identification. Creating behavioral model for a large software system is complicated and serious problem. Mostly, software system is running in special mode, when a tool is saving behavior of a software system and creates internal representation of the saved behavior. Design pattern is described in an appropriate internal structure. Analysis is based on comparison of saved behavior (and its structure) and design pattern behavior. Dynamic analysis is dependent on executed part of the software system, when any part of the system was not executed while creating behavior representation, patterns in this part cannot be identified.

Combination of static and dynamic analysis is sometimes used for more exact identification of used design patterns. Firstly, candidates of patterns are created using static-structural analysis. Afterwards dynamic-behavioral analysis is used for verification of pattern instance candidates. Exactness of combined design pattern identification method is realized by removing pattern instance candidates which do not satisfy dynamic aspects of identified design pattern.

In this paper we propose set of extensions for method for design patterns identification based on static structural analysis realized by similarity scoring algorithm. We introduce mechanisms called:

- feature weighting,
- feature filtering for an effective and precise design pattern identification,
- semantic analysis for extended meta information about used pattern.

We also present a short case study with the comparison of aboriginal and modified method.

2. Static Analysis

This section contains a short description of selected methods for design patterns identification. The detailed description is assigned to three methods with static-structural analysis. The last described method, using similarity scoring algorithm, was selected for next extensions.

At first we briefly describe Browns method [4] for design pattern identification. Browns method identifies four known design patterns from [6] based on principles of reverse engineering. Algorithm uses information about inheritance hierarchy, association and aggregation relationships between classes. Prechelt and Kramer in [10] designed and developed system for identification majority of design patterns in C++ source code. Based on OMT class diagrams which represent design pattern, Prolog rules for identification were created. Used logical approach requires definition of new Prolog rules for each new design pattern. Heuzeroth in [8] first time uses static-structural analysis for getting list of candidates for design pattern. Dynamic analysis was applied on the list of candidates from previous structural analysis. This approach depends on characteristic of a concrete design pattern, for each design pattern algorithm for static analysis needs to be designed and after that defines rules for dynamic analysis. Antoniol in [1] describes a technique for identification of structural design patterns in software system. He defines rules to constrain search space. Pattern instances are identified based on static-structural features. The technique was tested on small and medium size software systems. The main disadvantage was low exactness which was followed by large number of identified false positive instances. Balanyi and Ferenc describe Columbus framework usage in [2]. Columbus framework was used for getting abstract semantic graph and DPML (Design Pattern Markup Language) for description of roles in design pattern. Algorithm compares roles described in DPML with classes in abstract semantic graph. Search space is reduced using structural information. Technique was tested on four medium and large size software systems. The result was that more simple design pattern description is, larger number of false positive instances is identified. Costagliola in [5] describes the usage of a graphical format as intermediate representation in the process of design pattern identification. Design patterns are represented in terms of visual grammar. Identification is provided using technique for parsing visual language and parallel comparison of results with pattern library. A main advantage is the process visualization. On the other side, main disadvantage is absence of tool for source code transformation to visual representation. Vokac in [12] describes dependency between presence of design patterns and number of defects. By using tool for reverse engineering source, code is analyzed and structural metadata are created. Metadata are saved to database. Patterns are selected from database with DML. Structure of selects corresponds to structure of each design patterns. The technique was tested on large commercial software systems. Wyuts in [13] describes so called SOUL environment. Design patterns are described as prolog predicates and programming entities as facts. Prolog interference algorithm was used for unification of predicates and facts, which after that was used for entity identification. An identified entity was in the concrete role of design pattern.

2.1 Bit-Vector Algorithm

Bit-vector algorithm is used for pattern matching in general. Authors in [9] presented an adaptation of bio-informatics bit-vector algorithm to problem of design pattern identification. The identification of a design pattern consists of parallel traversing of a program and a design pattern. During the traversing of a program, the entities, that match entities in the design pattern in structure and in organisation, are recorded. Design patterns identification is a combinatorial problem, requiring all possible combinations of entities to be compared against design pattern entities.

Created iterative bit-vector algorithm identified exact or approximate coincidence between program and design pattern represented by string. Concrete design pattern/ system string representation contains classes (class name) and relations between classes (creation, specialisation, implementation, use, association, aggregation, composition). In general, the length of the string representing a design

pattern is short, less than 20 tokens, while the length of a string representing a program might vary, depending on the size of the system to analyse, usually thousands of tokens.

A string is converted to a set of bit vectors (characteristic vector) (for example if class C occurs in string, 1 occurs in bit vector for C, otherwise 0). Characteristic vectors are used for finding the entities playing a role in design pattern. Iterative bit-vector algorithm iteratively reads triplets of tokens (roles) in the design pattern string representation and associates program entities to the roles by resolving a unification-like problem using the characteristic vectors.

The process of design pattern identification described in [9] is composed from the following specific tasks:

- software models and design pattern models are converted into digraphs (A model created on the basis of a structure of design pattern or software system is a graph where classes are vertices and relationships between classes are oriented edges. If there exists more than one identical relationship between two same vertices, only one edge in graph is kept.)
- digraphs are converted to Eulerian digraphs (Digraph, in general, is not Eulerian digraph if it does not contain Eulerian circle. Graph conversion consists of adding dummy edges between vertices with unequal in-degree and out-degree.)
- Eulerian digraphs are converted to strings (Eulerian digraph contains an minimal Eulerian circuit, which is a cycle traversing each edge exactly once. Minimal Eulerian cycle is transformed to string representation, which characterizes software system/design pattern.)
- modified iterative bit-vector algorithm on the string representations is applied to identify exact and approximate occurrences of the design pattern.

Although the entire process is optimized for quick design pattern identification, the representative strings of large software systems are so long that the identification of the design pattern can be time consuming.

2.2 Design Pattern Fingerprint

In a not-so-distant past, individuals could be identified by external attributes only, such as height, weight, color of hair, of eyes, of skin. Thus, it was difficult to identify with certainty an individual uniquely, almost impossible without eyewitnesses. This situation changed when Sir Edward Henry devised and introduced in 1896 his classification system to identify criminals in Bengal using their fingerprints.

In [7] authors define a method for identifying design patterns based on design motives external attributes, called design pattern fingerprint. External attributes contain specific characteristics, for example size, filiation, cohesion, coupling.

Two or more classes may have identical values for a given set of external attributes, but two or more classes may play the same role in different uses of the design pattern. Authors formulate rules, a kind of fingerprints for design patterns roles using external attributes of classes. Based on concrete values of external attributes search space will be reduced for example in case of:

- observer – classes in role Observer in Observer design pattern are characterized by low coupling,
- singleton – classes in role Singleton in Singleton design pattern are characterized by low coupling and are located upper in inheritance tree.

Based on fingerprints called mechanism authors define method for identifying design patterns in the following steps:

- repository creation (repository of design patterns instantions created in manual way)
- metric extraction (metrics mining on design patterns instantions)
- rule learning (rules created based on extracted metrics)
- rule validation (process for removing unsuccessful rules)
- interpretation (identification new instances of patterns based on defined rules)

Metrics are counted only once for each software system. Rules for design patterns can be repeatedly reused. On the other side, method initialization is difficult and time consuming. Method initialization requires large set of pattern instances for rules mining. Method is more convenient for reducing search space in more complex method used for identifying design patterns in large software systems.

2.3 Similarity Scoring Algorithm

The work that we present in this paper is built on the ideas of [11] where the author presents design pattern detection method based on similarity scoring algorithm. Bondel in [2] defines an iterative algorithm for calculating the similarity between vertices of two different graphs by similarity matrix.

In the context of design pattern detection, the similarity scoring algorithm is used for calculating similarity score between a concrete design pattern and analyzed system. Let GA(system) and GB(pattern) be two directed graphs with NA and NB vertices. The similarity matrix Z is defined as an NB×NA matrix whose entry SIJ expresses how similar vertex J (in GA) is to vertex I (in GB) and is called similarity score between two vertices (I and J). Similarity matrix Z is computed in iterative way:

1. Z0 = 1,
2. iterate an even number of times and stop upon convergence,
3. Z is similarity matrix, where A, B are adjacency matrices of graphs GA and GB.

In [11] authors define a set of matrices for describing specific (pattern and software system) features (for example associations, generalizations, abstract classes). For each feature, a concrete matrix is created for pattern and for software system, too (for example association matrix, generalization matrix, abstract classes matrix). This process leads to a number of similarity matrices of size NB×NA (one for each described feature). To obtain overall picture for the similarity between the pattern and the system, similarity information is exploited from all matrices.

In the process of creating final similarity matrix, different features are equivalent. To preserve the validity of the results, any similarity score must be bounded within the range $\langle 0, 1 \rangle$. Higher similarity score means higher possibility of design pattern instance. Therefore, individual matrices are initially summed and the resulting matrix is normalized by dividing the elements of column i (corresponding to similarity scores between all system classes and pattern role i) by the number of matrices (ki) in which the given role is involved.

Using this algorithm for whole large system will be time consuming (long time process). The process of design pattern identification includes:

- creating one matrix N × N for one feature, where N is number of classes in system, in pattern,
- comparing matrixes for each feature between pattern and system,
- computing final similarity matrix.

Approach defined in [11] defines whole procedure for design pattern detection and identification including search space constrains. Defined approach contains set of predefined assumptions and specific steps for constraining design pattern search space by subsystem separation. In general, each subsystem contains only one pattern. Designed approach contains following four steps:

- reverse engineering for system under study (Each systems characteristic is represented as matrix n×n, where n is number of classes.)
- detection of inheritance hierarchies (The creation of inheritance trees is based on all kinds of generalization relationships. Multiple inheritance cannot be modelled with simple tree, so if class C has multiple parents A, B, it occurs in multiple inheritance trees with A and with B. This step is included for constraining search space, patterns are divided based on number of inheritance hierarchy.)
- construction of subsystem matrices (The creation of subsystems is based on inheritance trees and design pattern characteristics. If a design pattern contains one inheritance hierarchy (for example Composite, Decorator), one inheritance tree equals to subsystem. If a design pattern contains more inheritance hierarchies, the subsystems are formed by combining all system hierarchies.)
- application of similarity algorithm between subsystem matrices and the pattern matrices (Normalized similarity scores between each pattern role and each subsystem class are calculated.)
- extraction of patterns in each subsystem (Usually one instance of design patterns is present in each subsystem.)

3. Extended Method

3.1 Final Score Counting with Weighting Extension

Main drawback of the approach proposed by Tsansalis in [11] was a fact, that all of the structural features (like for example presence of inheritance, presence of an abstract class or interface) were treated equally. This resulted in the following facts:

1. In case, that the instance of the design pattern has some modifications, some of the structural features can not be identified. This leads to the lower similarity score and as a result the instance could be overlooked.
2. Lowering the threshold score, which represent the decision point between identifying a design pattern and denying of a structure which is not a design pattern, could lead us to incorrect identification of structures, that are similar to the structure of a design pattern but are not design pattern instances.

The solution to these two problems is the application of weighting of the structural parts of the pattern, so the more important parts, in relation to the essence of the design pattern, play more important role in the process of calculating of the final score (so has higher weight). The idea of weighting is based on a consideration, that every design pattern has some structural features that are essential for the pattern and some other structural features, which are in most cases domain specific [3]. These other structural features are also a part of the pattern structure, but their modification or absence has lower impact on the essence of the pattern (so has lower weight). For example, when we take the Composite pattern as described in [6], we can observe structural features:

- the presence of an abstract class,
- the presence of inheritance between Component class and Composite class,
- the presence of collection method invocations in the Composite class, which represents the presence of aggregation between Composite and Component classes,
- invocation of abstract methods inherited from abstract class.

Because we look at the Composite pattern as at a structure, that should provide a way of designing composite, tree-like structure, we can assume that the presence of collection methods invocation in the Composite class is important to the essence of the pattern, because this feature forms the composite structure.

Based on these considerations we can divide the structural features of design patterns into two major groups:

1. Features, which form the essence of the design pattern. These features are basic for the design pattern and therefore they should be domain invariable, so they should appear unmodified in every instance of the pattern – base features. In the process of similarity score calculation features from this group should be counted with higher weight.

2. Features, which help top up the structure of the design pattern, as it is described by the catalogue (see [6]). Into this group we can include structural features with a higher rate of domain specific modifications and therefore these features can absent in the pattern structure or can be modified in some way – additional features. As absence or modification of these features does not affect the essence of the pattern, we should count them with a lower weight, lowering so their importance for the final similarity score.

In case of previous example, where Composite design pattern is described with four features, similarity score is counted in consideration of allocation of weights for individual features (Figure 1).

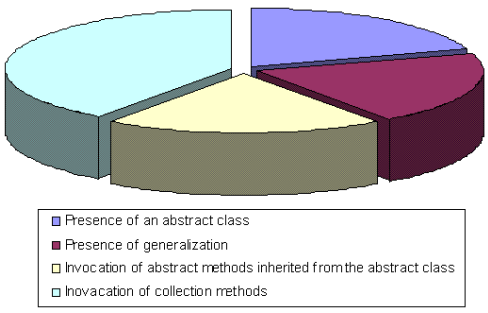

Figure 1: Extensions of original method.

The application of the weighting based on the separation of the features into two groups of importance has two expected results:

- Lower rate of mistaken identifications of parts of the analyzed system, which look like a design pattern instances, but in fact are not real instances.
- At a higher threshold score we should be able correctly identify more design pattern instances, because the methodology proposed by this paper should be less sensitive to the modification of parts of the design pattern, by which we expect a higher rate of domain specific modifications, or even their absence.

3.2 Instance filtering extension

Lowering threshold score means lowering method sensibility in identification modified instances of design pattern. When threshold score is too low, almost any structure can be identified as design pattern. On the other side when threshold score is too high some design pattern occurrences can be overlooked.

Weighting extension described in the previous example moves final score up if instance contains base features. When threshold score is too low and class structure contains some of base features, a class structure can be identified as a design pattern instance. For minimizing this kind of false positive instances we extend an original method with instance filtering extension. Method extensions and their relations are conceptually depicted on Figure 2.

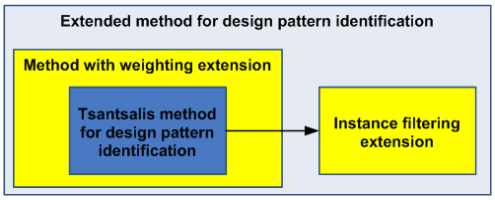

Figure 2: Extensions of original method.

Instance filtering extension is based on dividing features. Base features need to be used in every instance of pattern and additional features are domain specific, so in special cases they can be modified or even not present. Instance filtering extension checks possible candidates (from static analysis with weighting) for presence/not presence of base features. When a candidate contains every base feature connected with an appropriate design pattern, the candidate is checked as a design pattern instance. On the other side when a candidate does not contain only one of base features, candidate is not a design pattern instance.

The application of instance filtering extension based on the separation of the features into two groups of importance has two expected results:

- Lower rate of mistaken identifications of parts of the analyzed system, which look like a design pattern instances, but in fact are not real instances especially in case of low threshold score.
- At a higher threshold score results will not be modified, or will be modified only in small differences, because high final score can be obtained only with presence of all base features. When any of base features is missing, the final score is lower.

3.3 Lexicographical Analysis

Design patterns are not only structures, classes, relations but design patterns add specific terminology to design process. A main part of design pattern terminology is connected with design pattern roles. Based on Gamma recommendation in [6] classes need to be named using specific domain name and role name. Is this recommendation used in real software products, in real design process?

For confirmation of usage of this recommendation we design third extension of method for pattern identification. This extension is based on simple lexicographical analysis of pattern terminology and concrete instance terminology. We used existing instance representation in the method and based on the instance model we are looking for role names in design pattern instance terminology.

For example Composite design pattern was identified in JHotDraw 6 framework. Class CompositeFigure plays role of Composite in the selected instance of Composite pattern. Using simple lexicographical analysis which finds role name in class name we can say that Gammas recommendation was adhered.

Lexicographical analysis was designed to identify a difference between concrete class name and role name. The difference is expressed by using difference coefficient which is counted by the following similarity algorithm.

Similarity algorithm is based on words lexicographical comparison. The difference coefficient is:

- set to 0 if element (class, method, attribute) name equals to role name; for example class name Iterator is in role Iterator, difference coefficient is 0,
- set to length difference between element name and role name, if role name exists in element name; for example TestDecorator is in role Decorator, difference coefficient is
  length(TestDecorator) − lenght(Decorator) = 4,
- set to lexicographical difference between element name and role name plus offset constant, if role name does not exist in element name; for example Picture is in role Composite, difference coefficient is offset-constant + (Picture − Composite) = 100 + 13 = 113.

Application of lexicographical analysis in this case means only basic or first step of analysis. In the future complicated rules, based on for example usage of natural language, can be used.

4. Case Study

For testing purpose we use following four open source frameworks from different domains (graphics, web services, database access, web applications):

- JHotDraw 6.0 (`http://www.jhotdraw.org/`)
  JHotDraw is a Java GUI framework for technical and structured graphics. It has been developed as a "design exercise" but is already quite powerful. Its design relies heavily on some well-known design patterns. JHotDraw's original authors have been Erich Gamma and Thomas Eggenschwiler. [`http://www.jhotdraw.org/`]
- Apache Axis 1.4 (`http://ws.apache.org/axis/`)
  Axis is java framework for supporting web services development. Axis framework was developed in multiple versions for example for implementation languages C++, Java. Axis implements JAX-RPC API, standard for web services implementation.
- Hibernate 3 (`http://www.hibernate.org/`)
  Hibernate is a powerful, high performance object/relational persistence and query service. Hibernate is intended for developing persistent classes following object-oriented idiom – including association, inheritance, polymorphism, composition, and collections, express queries in its own portable SQL extension (HQL), as well as in native SQL, or with an object-oriented Criteria and Example API. [`http://www.hibernate.org`]
- Apache MyFaces 1.2.3 (`http://myfaces.apache.org/`)
  MyFaces is java framework for supporting web applications development using JSF technology. MyFaces is implementation of JSF – J2EE specification defined by JSR127 and JSR252.

Open source framework was used for availability of source code, so automated design pattern identification can be controlled by manual analysis. Four testing frameworks were implemented in Java and has together 3147 classes for testing purpose.

4.1 Original Method vs. Extended Method

Tests were realized with 2 values of threshold score:

- 1.0 – which means 100
- 0.75 – which means 75

Results of original and extended methods were controlled by manual source code analysis. We introduce TP (as true positive) and FN (as false negative) numbers of identified design pattern instances.

In comparison of original and extended method with threshold score 1.0 results were very similar. For design patterns Prototype, Composite, Decorator and Visitor only one difference was indentified. In case of Composite design pattern one instance was identified with the extended method in Hibernate 3 framework in comparison with no instance indentified with the original method.

Previous results mean that the extended method identifies all instances which were identified by the original method. Final similarity score of instances can be more than 1.0 (threshold score) only in special cases (e.g. Composite in Hibernate 3). The structure of identified instance needs to be equal to design pattern internal structure. Number of instances identified by the extended method is at least equal to number of instances identified by the original method. In table Tab. 1 results of original and extended methods for all four tested software systems with threshold score 0.75 are introduced.

In comparison to the original and extended method with threshold score 0.75 results were different in many cases. In this case main features move final score up over 0.75 threshold score which means that the extended method using weighting can moves final score over threshold score – more correctly identified instances.

In case of Decorator pattern more instances were identified in Axis 1.4 framework, Hibernate 3 framework and JHotDraw 6.0 framework. The difference between number of instances identified by the original and extended method was from 8 (in case of Axis 1.4) to 13 (in case of Hibernate framework) instances. In case of Composite and Prototype design patterns, difference between number of instances identified by the original and extended method was identified in case of JHotDraw 6.0 framework.

4.2 Result Analysis

Based on previous case study we can formulate the following results for weighting extension:

- Weighting extension of Tsantsalis method improves results in case of identification modified design patterns where some of features are suppressed. In this case feature weighting extension moves final similarity score of a design pattern instance up. This results in more identified instances of the concrete design pattern.

Table 1: Original and extended method results (with threshold score 0.75)

	Axis 1.4				Hibernate 3			
	TP		FN		TP		FN	
Pattern	Original	Extended	Original	Extended	Original	Extended	Original	Extended
Prototype	0	0	0	0	0	0	0	0
Composite	1	1	0	0	1	1	0	0
Decorator	**0**	**8**	0	1	**1**	**14**	0	1
Visitor	0	0	0	0	19	19	0	0
	JHotDraw 6.0				MyFaces 1.2.3			
	TP		FN		TP		FN	
Pattern	Original	Extended	Original	Extended	Original	Extended	Original	Extended
Prototype	**6**	**10**	-	-	0	0	0	0
Composite	**0**	**1**	0	0	0	0	0	0
Decorator	**1**	**11**	0	0	0	0	0	0
Visitor	1	1	0	0	0	0	0	0

- Weighting extension in its natural form cannot modify results in case of setting threshold score to 1.0. In this case instance coincidence with design pattern is 100 percent, so modifying weights of specific pattern features cannot affect to final results. Extended method in this case identifies each pattern instance identified by original method.
- Weighting extension of the original method moves final score down in case of structures which are not instances of the concrete pattern. An absence of base feature moves final score down twice in comparison with an absence of additional feature. When the structure does not include any of base features final score is moved down markedly. The final score in this case is lower than threshold score, which can lead to lower number of identified instances.
- Weighting extension in its natural form cannot modify results in case of setting all features weight to 1.0. Each feature has an impact on the final result with equal weight (1.0) in process of counting final score of instance.

Following results were formulated for the instance filtering extension:

- Instance filtering has lower impact on results in case of high threshold score (for example 0.75, 1.0). Instance filtering is based on presence of base features, instances with final score higher than high threshold score needs to include base features, otherwise instances (or structures) cannot have final score higher than threshold score.
- Instance filtering extension has higher impact on results in case of using low threshold score (0.1, 0.3). In this case structures which have final score higher than threshold score can miss base features, instance filtering removes this kind of structures form design pattern instance list.

The following results were formulated for the instance filtering extension:

- Instance filtering has lower impact to results in case of high threshold score (for example 0.75, 1.0). Instance filtering is based on presence of base features, instances with final score higher than high threshold score needs to include base features, otherwise instances (or structures) cannot have final score higher than threshold score.

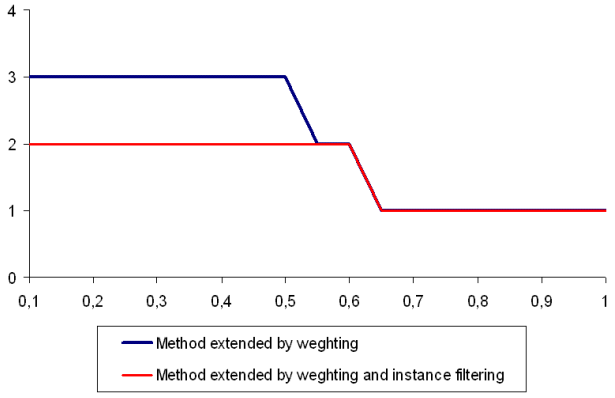

Figure 3: Instance number with/without instance filtering extension.

- Instance filtering extension has higher impact to results in case of using low threshold score (0.1, 0.3). In this case structures which have final score higher than threshold score can miss base features, instance filtering removes this kind of structures form design pattern instance list (Figure 3).

And the following results were formulated for lexicographical analysis of naming of pattern instances:

- Recommendation of Gamma for used naming convention is almost not currently used. In case of 3147 analyzed classes, 399 pattern instances were identified. Rule for class naming convention was used only in 5 percent of all identified pattern instances (Figure 4).

5. Conclusions

The proposed method is a partial solution of the problem of design pattern identification in a fully automated way. We have designed the weighting into the process of similarity score computation, instance filtering based on presence (or absence) of specified structural features of

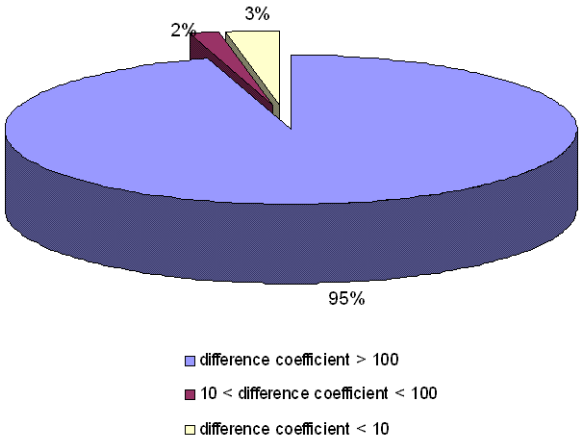

Figure 4: Difference coefficient for patterns class names.

design pattern and additional analysis with using lexicographical distance between a design pattern terminology and an instance terminology for additional information about a design pattern usage.

Weighting gave us a possibility to consider the partial results in relation with the essence of the identified pattern and later to affect the impact of the results on the final result of the identification. The modifications, we have made, have brought an improvement in two aspects:

- the rate of mistaken identifications of structures that look like a design pattern is lower than the rate of the original similarity scoring methodology,
- at a higher threshold score we are able to identify more instances of design patterns, which means that the proposed methodology is less sensitive to modifications of design patterns and thus is better at identifying of modified design patterns.

The instance filtering extension has an impact on results in special cases, when threshold score is lower and instances, which are missing some basic structural features, have higher final score than the specified threshold score. The instance candidate is in the final list of instances only in case that it contains specified basic structural features.

The lexicographical analysis extension adds only additional information about design pattern usage. It uses lexicographical distance between design pattern terminology and pattern instance terminology. The lexicographical analysis points to wrong usage of the design pattern terminology.

Proposed extensions (especially weighting and instance filtering) improve identification of true positive design pattern instances.

References

[1] G. Antoniol, G. Casazza, M. D. Penta, and R. Fiutem. Object oriented design patterns recovery. *J. Systems and Software*, 59(2):181–196, 2006.

[2] Z. Balanyi and R. Ferenc. Mining design patterns from c++ source code. In *Proceedings of IntŠl Conf. Software Maintenance*, pages 305–314, 2003.

[3] F. Bergenti and A. Poggi. Improving uml designs using automatic design pattern detection. In *Proceedings of 12th IntŠl Conf. Software Eng. and Knowledge Eng.*, pages 336–343, 2000.

[4] K. Brown. Design reverse-engineering and automated design pattern detection in smalltalk. *Technical Report TR-96-07*, 1996.

[5] G. Costagliola, A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi. Design pattern recovery by visual language parsing. In *Proceedings Ninth European Conference Software Maintainance and Reeng.*, pages 102–111, 2005.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison Wesley Profesional, Massachusetts, 2004.

[7] Y. G. Guéhénuc, H. Sahraoui, and F. Zaidi. Fingerprinting design patterns. In *Proceedings of 11th Working Conference of Reverse Engineering*, pages 172–184, November 2004.

[8] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe. Automatic design pattern detection. In *Proceedings of 11th IEEE IntŠl Workshop Program Comprehension*, page 94, 2003.

[9] O. Kaczor, Y. G. Gueheneuc, and S. Hamel. Efficient identification of design patterns with bit-vector algorithm. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, pages 175–184, Marec 2006.

[10] L. Prechelt and C. Kramer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *J. Universal Computer Science*, 4(12):866–882, December 1988.

[11] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, November 2006.

[12] M. Vokač. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Transactions on Software Engineering*, 30(12):904–947, December 2004.

[13] R. Wuyts. Declarative reasoning about the structure of object oriented systems. In *Proceedings of the 26th conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124, August 1998.

Selected Papers by the Author

J. Jakubík and P. Návrat. Reuse of Patternt's Source Code. In *Proceedings of the Seventh Joint Conference on Knowledge-Based Software Engineering, JCKBSE 2006,* pages 143–146, Tallin, Estonia, IOS Press, 2006.

J. Jakubík and M. Cichý. Design patterns identification using similarity scoring algorithm with weighting score extension. In *Proceedings of the Eighth Joint Conferencene on Knowledge-Based Software Engineering, JCKBSE 2008*, pages 465–473, Athens, Greece, IOS Press, 2008.

J. Jakubík. Modeling systems using design patterns. In *Proceedings of IIT.SRC 2005 student research conference*, pages 151–158, STU, Bratislava, 2005.

J. Jakubík. Isolating General Parts of the Composite Design Pattern. In *Objekty 2005, 10th Conference.* V. Snášel (Ed.), Ostrava, Czech Republic, 2005. pages 74–84. (in Slovak)

J. Jakubík. Comparison of CASE tools based on design patterns source code support. In *Proceedings of IIT.SRC 2006 student research conference*, pages 197–203 STU, Bratislava, 2006.

J. Jakubík. Possibilities of design patterns reuse. In *Proceedings of Software development 2006, 32nd Conference with International Participation,* Ostrava, 2006, pages 59–66. (in Slovak).

J. Jakubík. Using Design Patterns in Software Projects. In *Objekty 2007, 12th Conference*, VŠB, Ostrava, 2007. (in Slovak)

J. Jakubík, M. Cichý. Design patterns identification using similarity scoring algorithm with weighting score extension. In *Proceedings of IIT.SRC 2008 student research conference*, STU, Bratislava, 2008.