CBR AND MICRO-ARCHITECTURE ANTI-PATTERNS BASED SOFTWARE DESIGN IMPROVEMENT

Tie Feng^{1,2}, Jiachen Zhang¹, Hongyuan Wang¹, Xian Wang¹

¹*Institute of Computer Science and Technology, Jilin University, Changchun, 130012, China (fengtie@public.cc.jl.cn);* ²*Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education*

Abstract: This paper presents a Case Based Reasoning (CBR) approach to identifying micro-architecture anti-patterns and replacing them with “good” patterns in order to improve the design of software system. The resulting system design benefits from better flexibility for adapting to future requirement change and expansion. In this approach, both problematic inflexible structures and their corresponding refactoring designs are formally defined and organized in a case base. The identification of anti-patterns is carried out through similarity measurement on class diagrams, sequence diagrams, OO quality metric and semantic constraints. A supporting system CBDIT was developed to aid this approach.

Key words: Case Based Reasoning, Design Pattern, Anti-Pattern, Program Evolution, Refactoring.

1. INTRODUCTION

Once a software product is delivered, the maintenance cycle begins, where continuous changes need to be made to the software product to accommodate further user requirements or adapt to new hardware or software environment. One of the barriers to enabling a smooth software maintenance cycle is the stiffness in the design of micro-architecture of many software systems. Identifying and removing the stiffness of these software design - so called “anti-patterns” will improve the effectiveness and efficiency of software maintenance.

Design pattern is commonly used in object-oriented design methodology to capture objects, collaborations and distribution of responsibility [1]. Anti-pattern is the next step to designing pattern research. Informally, an anti-pattern is a generalization of commonly occurring design solution to a problem that has negative consequence [2]. Most of existing work focuses on describing design pattern, especially anti-patterns, in natural language. We envisage that to be able to represent anti-patterns in formal language and provide a tool to automatically support the identification of anti-patterns and replace them with good quality design patterns will significantly advance the state of the art in this area.

Various attempts have been made in applying AI techniques to engineering practice. One of the most popular AI techniques is CBR that, in the context of software design improvement, can help capture experienced software design knowledge in an accumulated way. The captured software design can be used as big knowledge blocks to identify the problematic micro-architectures.

The following sections are organized as follows. Section 2 introduces the pattern language we use. Section 3 gives an example of anti-pattern and how it can be improved. Section 4 describes the framework of representing case. Section 5 illustrates the procedure of retrieving cases and provides an algorithm for identifying anti-patterns in software design. Section 6 presents an algorithm of applying cases through FACADE-oriented method. Section 7 introduces the architecture of CBDIT, CBR-Based Design Improvement Tools. The related work can be found in Section 8 and finally we reach conclusion.

2. SOFTWARE DESIGN PATTERN LANGUAGE

A software design can span across seven architectural levels, namely, global, enterprise, system, application, macro-component (framework), micro-architecture and object. Micro-architectural level contains a group of cooperative objects that inter-play with each other. The audience of micro-architectural design is normally programmers who implement the software system.

In our system, we adopt the following design language that contains class and relationships between classes. The class has the same meaning as that can be found in OO language. The relationships between classes can be classified into:

- $\text{Inh}(C1, C2)$, where Class $C2$ inherits Class $C1$.
- $\text{Agg}(C1, C2, m)$, where an object of Class $C1$ is aggregated by one or more objects of Class $C2$.

- $Ini(C1, C2)$, where an object of Class C1 initiates an object of Class C2.
- $Del(C1, C2)$, where an object of Class C1 delegates an object of Class C2 to fulfill some task.

Figure 1 shows an example of static design denoted in our design language.

Figure 1. An Example for Static Design

In addition to static design, we also capture the dynamic behavior of software system. Similar to UML, where dynamic behavior is described by sequence diagram and collaboration diagram, we describe the interaction among objects using interaction sequence. An interaction sequence is represented as a list of tuples with three elements: initiating object, response object and message signature. Formally, we have

$IS = (io, ro, msg)^*$, $io \in S_1$, $ro \in S_1$, $msg \in S_2$, where IS is the interaction sequence; S_1 is the set of objects interact with each other at runtime; S_2 is the set of messages between two objects; io is the initiating object and ro is the response object. Figure 2 shows an example of dynamic behavior of system design.

Figure 2. An Example of Dynamic Behavior of System Design

The interaction sequence is important for identify occurrence of problematic behavior, such as over-frequent interaction with database.

3. AN EXAMPLE OF PATTERN AND ANTI-PATTERN

In this section we give an example of an anti-pattern instance using the denotation of the previous section and, in comparison, an improved pattern. First we have the following definition:

An architecture D is called micro-architecture anti-pattern, if it satisfies the three conditions:

(1)

```
D = < StaticStructure > | < StaticStructure > < DynamicBehavior >
StaticStructure = < SClasses > < SRelationships >
SClasses = Class *
Class = < ClassName > | < Attribute > * < Method > *
SRelationship = Relation *
Relation = < Inh > | < Agg > | < Ini > | < Del >
Inh = "Inh(" < ClassName > , < ClassName > ")"
Agg = "Agg(" < ClassName > , < ClassName > , < Multiple > ")"
Ini = "Ini(" < ClassName > , < ClassName > ")"
Del = "Del(" < ClassName > , < ClassName > ")"
DynamicBehavior = < IS > *
IS = "(" < io > " , " < ro > " , " < msg > ")"
io = < ObjectName >
ro = < ObjectName >
msg = < ReturnType > < MsgName > "(" FormalParameterList ")"
```

(2) D caused, in practice, quality problems in performance and extensibility for at least 3 times. (3) There are more than eight classes and objects in D.

Anti-pattern is an abstraction of a family of anti-pattern instances. Anti-pattern and anti-pattern instance is different in that the names of classes, objects and messages in the former are usually domain-independent (such as abstract product, concrete product, etc.), whereas the those elements in the latter are domain-dependent (such as staff, toolbar, etc.) Also, the number of classes and objects in anti-patterns is much smaller than that in anti-pattern instances because the classes and objects in anti-pattern have comprehensive typicality. Figure 3 shows an example of Anti-pattern and its corresponding XML expression. The result of expected design after improvement can be found in Figure 4.

Figure 3. An Example of Anti-pattern and Its Corresponding XML Expression

Figure 4. Result of Improved Design using Our Approach

4. REPRESENTING CASE

In our design improvement process, we have cases that capture five attributes, namely, Anti-Pattern Description (APD), Semantic Constraints (SC), Quality Metrics Before Improvement (QMBI), Pattern Description (PD) and Quality Metrics After Improvement (QMAI).

4.1 Describing Anti-pattern and Pattern

Both APD and PD contain four sub-attributes: name, effect, static structure and dynamic behavior. Static structure and dynamic behavior play

important role in the similarity measurement of patterns; they can be represented in the forms of UML diagram, structure graphs, XML documents and adjacency list and stored in XML files or SQL databases.

Definition 4.1 A class structure graph is denoted by $G = (N_c, E_r, \beta, L_\beta)$, where N_c is a finite set of nodes indicating classes, $E_r \subseteq N_c \times N_c$ is a finite set of edges indicating relationships, $\beta: E_r \rightarrow L_\beta$ is a edge labeling function, and $L_\beta \subseteq \{Inh, Agg, Del, Ini\}$ contains symbolic labels for edges.

Definition 4.2 An object structure graph is denoted by $G = (N_o, E_m, \beta, L_\beta)$, where N_o is a finite set of nodes indicating objects, $E_m \subseteq N_o \times N_o$ is a finite set of edges indicating messages, $\beta: E_m \rightarrow L_\beta$ is a edge labeling function, and $L_\beta \subseteq PosInt$, where $PosInt = \{x | x \text{ is a integer} \wedge x > 0\}$, contains symbolic labels for edges.

It is obvious that both class structure graph and object structure graph are special examples of directed labeled graph in which symbolic label for node and node labeling function are ignored. The symbolic label for node and node labeling function are deliberately neglected in our approach to guarantee that original design could be equivalent to domain-independent case by renaming its nodes. On the other hand, if detailed information is needed to compare original design with domain-specific case, that information can be easily retrieved from UML diagrams and XML files. Our main focus here is the structure of classes and objects. As an example, the class diagram shown in Figure 1 can be transformed to class structure shown in Figure 5 (a), and the sequence diagram shown in Figure 2 can be transformed to object structure graph, shown in Figure 5 (b).

Figure 5. Simplified Class Structure Diagram and Object Structure Diagram

Definition 4.3 Let $G_i = (N_i, E_i, \beta_i, L_{\beta_i})$, $G_s = (N_s, E_s, \beta_s, L_{\beta_s})$ be two class structure graphs or two object structure graph, the map from G_i to G_s is denoted by $\eta: G_i \rightarrow G_s$ which satisfies:

- $\forall n_i \in N_i$, n_i maps to one and only one $n_s \in N_s$, denoted by $\eta(n_i) = n_s$;

- $\forall e_i = (n_{i1}, n_{i2}) \in E_i$, if $\eta(n_{i1}) = n_{s1}$, $\eta(n_{i2}) = n_{s2}$ and $e_s = (n_{s1}, n_{s2}) \in E_s$, then e_i maps to e_s , denoted as $\eta(e_i) = e_s$.

From above definition, it is obvious that if there is a map from G_i to G_s , $\eta: G_i \rightarrow G_s$, the number of nodes in G_i is equal to or less than that of G_s . In the context of anti-pattern identification, if we can find a map between an anti-pattern and a trunk of graph in the original design, the anti-pattern is thereby identified and the case attached to the anti-pattern can be retrieved and applied to the original design, for instance, the anti-pattern is replaced with a good design pattern.

It is also noted that different relationships in a class diagram contribute differently to describing the static structure of a case. Thereby, every edge in the class structure graph in APD is assigned with a weight to denote that contribution. The weight can be adjusted by domain experts. Similarly, every edge in the object graph in APD is also assigned with a weight value.

Let NEC be the number of edges in class structure graph representing class diagram in APD, the default weight of each edge is assigned as $0.3 \cdot (1/NEC)$.

Let NEO be the number of edges in object structure graph representing sequence diagram in PD, the default weight of each edge is assigned as $0.2 \cdot (1/NEO)$.

4.2 QMBI and QMAI

QMBI and QMAI each contain eight sub-attributes: Weighted Methods per Class (WMC), Number of Children (NOC), Depth of Inheritance Tree (DIT), Response for a Class (RFC), Coupling Between Objects (CBO), Lack of Cohesion in Methods (LCOM), Number of Abstract Class (NOAC) and Number of Relation with Abstract Class (NRWAC). The definition of the first six sub-attributes can be found in [3]. In addition, we believe that the last two sub-attributes, in the micro-architecture context, implies the rate of application of polymorphism mechanism in OO design – an indicator of extensibility of an OO system, and thereby also need to be included. As an example, the NOAC and NRWAC for the designs in Figure 3 and Figure 4 are listed in Table 1.

Table 1. Changes in NOAC and NRWAC before and after Design Improvement

	Design in Figure 3	Design in Figure 4
NOAC	0	1
NRWAC	0	4

4.3 Semantic Constraints

Structural information alone is not sufficient to decide whether change can be made to the original design. Take the following case for example, although there exists a structural mapping between the design on the left-hand side in Figure 6 with that in Figure 3, the two are not referred to the same design pattern because of the mismatch of the semantic information here. As a result, we need to incorporate semantic information into the matchmaking process.

Figure 6. An Example for Structurally-Matched while Semantically-Not-Match Design

Semantic constraints are represented by weighted logic predicates that describe design knowledge. For example, the semantic constraints in the design improvement case shown in Figure 3 and Figure 4 are given as below:

- $\text{MeaningSimilar}(\text{Class}_{i1}.\text{Function}_{i1}, \text{Class}_{i2}.\text{Function}_{i2}, \dots, \text{Class}_{im}.\text{Function}_{im})$ [$\omega 1$]
- $\text{Invoke}(\text{Class}_z.\text{Function}_y, \text{Class}_{j1}.\text{Function}_{j1}, \dots, \text{Class}_{jm}.\text{Function}_{jm})$ [$\omega 2$]
- $\text{Class}_{i1}.\text{Function}_{i1} = \text{Class}_{j1}.\text{Function}_{j1}, \dots, \text{Class}_{im}.\text{Function}_{im} = \text{Class}_{jm}.\text{Function}_{jm}$ [$\omega 3$]

The facts that satisfies the above predicates are:

$\text{MeaningSimilar}(\text{Student.Register}, \text{Teacher.Enroll}, \text{DashMinCheckin})$ [$\alpha 1$]

$\text{Invoke}(\text{StaffManagement.NewTerm}, \text{Student.Register}, \text{Teacher.Enroll}, \text{DashMinCheckin})$ [$\alpha 2$]

$\text{Student.Register} = \text{Student.Register}, \text{Teacher.Enroll} = \text{Teacher.Enroll}, \text{DashMinCheckin} = \text{DashMinCheckin}$ [$\alpha 3$]

Let NSC is the number of semantic constraints, the default weight of each

predicate is assigned as $0.3 \cdot (1/\text{NSC})$. For example, default weight of MeaningSimilar, Invoke and equations are both $0.3 \cdot (1/3) = 0.1$.

Synonym library can be consulted to judge the value of predicates, such as MeaningSimilar. Synonym library are indexed by domain name and part of speech. In our system, 8 domains are considered: Manufacturing, Telecommunications, Health care, Insurance, Financial services, Professional services, Travel and E-Commerce Models. Our survey suggests that complete data model resource has been explored, however very few work on domain application model resource has been explored [4]. With the application of the system, more domain catalogs and synonyms can be added by the domain experts.

5. RETRIEVING CASES

Case retrieval consists of 4 parts: class structure graph match, object structure graph match, quality metric data match and semantic constraints match. In our approach, the maximal values of degree of similarities (DS_c , DS_o , DS_q and DS_s) of above 4 parts are guaranteed as 0.3, 0.2, 0.3 and 0.2 according to our calculation method of degree of similarity. That is $\text{Max}(\text{DS}) = \text{Max}(\text{DS}_c + \text{DS}_o + \text{DS}_q + \text{DS}_s) = 0.3 + 0.2 + 0.2 + 0.3 = 1$.

5.1 Similarity Measure of Structure Graphs

After designs are transformed into directed labeled graphs (class structure graph and object structure graph), algorithms for matching class diagrams and for matching sequence diagrams are similar. Matching problem is equivalent to a sub-graph isomorphism detection problem. Although sub-graph isomorphism detection problem has been shown to be NP-complete [5], graphs transformed from diagrams of APD are very small ones usually only with less than 8 nodes. Furthermore, there exist techniques, such as tabu search [6], which use domain specific knowledge for graph-structured case retrieval, and decomposition approach [7].

Taking class structure graph as example (the algorithm for object structure graph is similar), to measure degree of similarity of two graphs, we first represent class structure graph as adjacency list. An adjacency list is a representation for a directed graph that has n vertexes and an array of n lists of vertexes. List i contains vertex j if there is an edge from vertex i to vertex j . Adjacency list records information of node index, some marks indicate node counterpart, edge label, weight of edge, whether a node or an edge has

been successfully checked, and so on. The algorithm for matching between two adjacency lists is listed in Figure 7.

Input: two adjacency lists, one represents class structure graph of APD (adjAPD), the other one represents original design (adjOD); a threshold of degree of similarity (TDS).

Output: adjacency lists, each represents a sub-graph of class structure graph of original design which have a higher DS than TDS.

Figure 7. Algorithm for Matching between Two Adjacency Lists

5.2 Similarity Measure of Quality Metric Data

$DS_i = 0.2 \times \left[\sum_{i=1}^8 \left(1 - \frac{|VI_i - VC_i|}{VC_i} \right) \times \frac{1}{8} \right]$, where VI_i and VC_i 's meaning are listed in Table 2. It is obvious that DS_q obtains its maximum 0.2 when All $VI_i=VC_i$ ($i=1, \dots, 8$).

Table 2. Meaning of VI_i and VC_i

	1	2	3	4	5	6	7	8
VI (Value of Initial design)	WMC	NOC	DIT	RFC	CBO	LCOM	NOAC	NRWAC
VC(Value of Case)	WMC	NOC	DIT	RFC	CBO	LCOM	NOAC	NRWAC

5.3 Similarity Measure of Semantic Constraints

$DS_s = \sum_{i=1}^n M_i$, where n is the total number of semantic constraints; $M_i = \omega_i$ if the i th semantic constraint is satisfied; $M_i = 0$ otherwise. It is obvious that the maximum value of $DS_s = \sum_{i=1}^n \omega_i = 0.3$.

6. FACADE-ORIENTED CASE APPLICATION

```

i = 2 ;
While (Scase ≠ ∅) Do
    D1 = D1 ; Get one Case ∈ Scase ; Let Scase = Scase - Case ;
    Let S1 = { class | class ∈ D1 ∧ class ∉ Case.APD }
    Let S2 = { class | class ∈ Case.APD }
    Remove all classes ∈ S1 from D1 ;
    Add a class FACADE to D1 ; Add all classes ∈ Case.PD to D1 ;
    Remove all relationships ∈ Case.APD from D1 ;
    Add all relationships ∈ Case.PD ;
    While (S1 ≠ ∅) Do
        Get one C1 ∈ S1 ; Let S1 = S1 - C1 ;
        While (S2 ≠ ∅) Do
            Get one C2 ∈ S2 ; Let S2 = S2 - C2 ;
            If (∃ R ∈ { Inh, Ini, Del } in D1 ∧ R(C1, C2)) Then
                Remove R(C1, C2) from D1 ; Add R(C1, FACADE) to D1 ; EndIf ;
            If (∃ R ∈ { Inh, Ini, Del } in D1 ∧ R(C2, C1)) Then
                Remove R(C2, C1) from D1 ; Add R(FACADE, C1) to D1 ; EndIf ;
            If (∃ Agg(C1, C2, k) in D1) Then
                Remove Agg(C1, C2, k) from D1 ; Add Agg(C1, FACADE, k) to D1 ; EndIf ;
            If (∃ Agg(C2, C1, k) in D1) Then
                Remove Agg(C2, C1, k) from D1 ; Add Agg(FACADE, C1, k) to D1 ; EndIf ;
        EndWhile ;
    EndWhile ;
EndWhile ;

```

Figure 8. Algorithm of FACADE-Oriented Case Application

Case application is the procedure to replace the problematic micro-architecture. The algorithm for FACADE-oriented case application is listed in Figure 8.

Input: a set of selected cases which will be applied into original design, original design D_1 .

Output: a sequence of improved designs, each one of which is obtained by applying one case.

7. CBDIT: A CBR-BASED DESIGN IMPROVING TOOL

Figure 9. Architecture of CBDIT

The architecture of CBDIT, which is shown in Figure 8, includes four sub-systems.

- UML Editor sub-system is designed to visually draw class diagram and sequence diagram.
- Design Extraction sub-system is designed to extract design information from legacy code with incomplete design documentation.
- DB and KB Management sub-system is designed to create, delete, modify and search data and knowledge stored in databases and knowledge bases.
- CBR Reasoning Engine is the heart of CBDIT whose working mechanism has been explained in previous sections.

8. RELATED WORK

Paulo Gomes [8] presented a method of using CBR for automating software design patterns and implemented it into a system called REBUILDER. Differences can be identified between Gomes's method and ours:

- The cases in REBUILDER are used to capture similar design pattern across applications, whereas our cases are specifically designed to describe anti-patterns.
- REBUILDER uses WordNet to help retrieve design cases. In Wordnet, single terms are interconnected through simple relations. However, a single word is insufficient in describing and matching case. In our approach, we use context information, such as combined graph and semantic information, to help identify cases.

Alexandre L. Correa [9] described an approach that uses heuristics, design patterns and anti-patterns to reusing object oriented design expertise and a tool called OOPDTool. In their method, OO design constructions are identified by meta-model predicates and rule-based reasoning.

Richard C. Gronback [10] introduced a mechanism of using audits, metrics and refactoring in Boland Together ControlCenter to remodel software. This method is driven by the automated measurement of quality factors, such as CBO (Couplings Between Objects), RFC (Response For Classes), and audits, such as NOIS (Negation Operator on 'if'). Unfortunately, only conversation based manual refactoring interface is provided.

Jirapun Daengdej [11] attempts to use CBR for design reuse. However, their method doesn't address design pattern or anti-pattern issue. Instead, they search similar E-R diagram in design library given some domain specific key words.

There is also some other research work on applying design patterns to legacy code analysis [12][13][14]. However, their focus on design patterns is to improve the understandability of legacy system.

9. CONCLUSION

This paper describes an approach to improving software design for increased flexibility, expansibility, and maintainability of object-oriented system. Based on CBR technique, this approach is capable of identifying micro-architecture anti-patterns in original design and substituting it with high quality designs.

The future work includes efficiently addressing the variation of micro-architecture anti-patterns, extending knowledge base to cover more application domains and design knowledge, and developing powerful reengineering tool to help extract more precise design information.

REFERENCES

- [1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements Of Object-Oriented Software Architecture*, Addison-Wesley, Reading, MA, 1995
- [2] William J. Brown, Raphael C. Malveau, Hays W. “Skip” McCormick III, Thomas J. Mowbray, *Antipatterns: Refactoring Software, Architectures, And Projects In Crisis*, Wiley Computer Publishing, 1998
- [3] Chidamber, S.R., and C.F. Kemerer, “A Metrics Suite for Object-Oriented Design”, *IEEE Transactions on Software Engineering*, 20(6), pp.476-493, 1994.
- [4] Len Silverston, *The Data Model Resource Book Revised Edition Volume2: a Library of Universal Data Models by Industry Types*, John Wiley&Sons, Inc. New York, 2001
- [5] Garey, M.R., and Johnson, D.S., *Computers and Intractability a Guide to the Theory of NP-Completeness*. W.H.Freeman, 1977.
- [6] Sanja Petrovic, Graham Kendall, and Yong Yang, “A Tabu Search Approach for Graph-Structured Case Retrieval”, In *Proc. STAIRS 02*, volume 78 of *Frontiers in Artificial Intelligence and Applications*, pp. 55-64. IOS Press, 2002.
- [7] Messmer, B.T., and Bunke,H., “Efficient Subgraph Isomorphism Detection: A Decomposition Approach”, *IEEE transactions on knowledge and data engineering*, 12-2, pp.307-323, 2000.
- [8] Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José Ferreira, Carlos Bento, “Using CBR for Automation of Software Design Patterns”, in *Proc. of the European Conference on Case-Based Reasoning (ECCBR’02)*, Aberdeen, Scotland, UK, September, 2002, LNAI 2416, pp. 534-548, Springer-Verlag Berlin Heidelberg, 2002
- [9] Alexandre L. Correa, Cláudis M.L. Werner, Gerson Zaverucha, “Object Oriented Design Expertise Reuse: an Approach Based on Heuristics, Design Patterns and Anti-Patterns”, URL: <http://citeseer.nj.nec.com/439618.html>
- [10] Richard C. Gronback, “Software Remodeling: Improving Design and Implementation Quality”, A Borland White Paper, January, 2003.
http://www.togethersoft.com/events/webinars/archive/software_remodeling_whitepaper.pdf
- [11] Jirapun Daengdej, Peerapol Moemeng, and Supawat Charoenvikrom, “Toward the Use of Case-Based Reasoning for Design Reuse”, 2002.
<http://citeseer.nj.nec.com/534321.html>
- [12] Niere, W.Schafer, J.Wadsack, L.Wendehals, and J.Welsh. “Towards Pattern-based Design Recovery”, In *Proc. of the 24th International Conference on Software Engineering*, Orlando, Florida, USA, pp. 338-348, May, 2002.
- [13] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, Welf Lowe, “Automatic Design Pattern Detection”, In *Proc. of the 11th IEEE International Workshop on Program Comprehension (IWPC’03)*, Portland, Oregon, May 10-11, pp.94-104, 2003, USA.
- [14] Herve Albin-Amiot, Pierre Cointe, Yann-Gael Gueheneuc and Narendra Jussien. “Instantiating and detecting design patterns: Putting Bits and Pieces Together”, In *Proc.*

of the **16th** IEEE International Conference on Automated Software Engineering (ASE'01), San Diego, California, pp.166-173, November 26-29, 2001.