Design Flaws Detection in Object-Oriented Software with Analytical Learning Method

Sakorn Mekruksavanich

Abstract—**Design flaws are used as an indicator to identify problematic classes in object-oriented software systems and they directly decrease software quality, such as maintainability. It is unavoidable to use high maintenance cost with design flaw systems. Therefore such design flaws must be identified to avoid their possible negative consequences on development and maintenance of software systems. This paper proposes a new methodology for design flaws detection. Symbolic logic representation and analytical learning technique are used to diagnose design flaw problems in simple way and to extrapolate patterned rules for complex flaws. The methodology is validated by detecting design flaws in an open-source system. The results show expected precision and false positive.**

Index Terms—**design flaws, detection, object-oriented design, code smell, explanation-based learning.**

I. Introduction

In the recent software technology development, most object-oriented software systems are developed under evolutionary process models. Most software that is related to a real-world problem domain, must continuously evolve to cope with the problem domain changes - requirement and environment changes [1]. Object-oriented software design principles and heuristics [2, 3] are proposed to promote good quality software. However, even when maintainers are familiar with those techniques, violation of these design rules may be led to *poor* solutions by deadline pressure, excessive focus on pure functionality, or inexperience programming. Such solutions to recur design and implementation problems hinder software evolution. They, low-level or local problems, are called *design flaws* [4].

A design flaw itself is not an error or problem but a strong indication of poor design of source code structure. Such flaws oblige software systems to directly decrease software quality [5], especially difficult to maintain when the software need modification [6]. In the literatures, many researchers introduce flaws in different ways. Fowler and Beck [6] coin the term "Bad Smell". They present an informal definition of twenty two bad smells that provide a set of characteristics used as indicators for design flaws. Anti-patterns, proposed by Brown *et al.* [7], are a design level literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences. Therefore the identification and removal of design flaws in software are the necessity of recent software development. The identification and removal of design flaws are typically performed during developing, testing, and maintenance phrase of the software development life cycle. Refactoring is the famous technique which deals with these flaws by changing the system's internal design while preserving its observable behavior [6]. This technique is one of particularly important activity in agile methodologies.

The identification of design flaws in the early age depends on manual detection such as code inspection technique [8]. The technique involves carefully examining source codes, design, and documentation of software and checking for potential problems based on past experience. However, being time-consuming, non-repeatable and non-scalable are the disadvantages of this technique. More issues concerning the manual detection of design flaws are identified by Mantyla *et al.* [9]. They show that if an experienced developer increases a certain software system, his ability to perform an objective evaluation of the system as well as his ability to detect design flaws decreases.

To avoid limitations of those issues, heuristic metrics are proposed to identify design flaws in software systems [10, 11]. The detection of software metric uses a pre-defined set of thresholds to interpret the detection results. Although being effective in automated detection, identification depends on proper metrics and thresholds which are used to detect such flaws.

The goal of this paper is to propose the new approach for design flaw detection. Two techniques, symbolic logics representation and analytical learning, are used to efficiently and simply detect design flaws. With symbolic logic, a source code is transformed into the narrow related problem domain and then design flaw problems can be managed in an easy way. Detection rules are then created for complex flaws by using light weight knowledge-based systems combined with well-founded experiential learning principles. The key issue of experiential learning is to extrapolate the experience from the learning setting to real world situations. The proposed detection methodology also can automatically detect design flaws in logic environment to reduce time consumption in detection process.

The structure of the rest of this paper is as follows. Section 2 presents detailed descriptions and representation used to encode the knowledge of the analytical learning. The proposed detection approach is detailed in Section 3. In Section 4, results and discussion show effectiveness of the proposed approach. Related works are presented in Section 5. Conclusion and future works are described in Section 6.

Manuscript received July 5, 2011; revised July 21, 2011.

Sakorn Mekruksavanich is with School of Information and Communication Technology, University of Phayao, Phayao, 56000 THAILAND (e-mail: sakorn.me@up.ac.th).

II. Theoretical Background

Analytical learning is a machine learning algorithm which concerns with the acquisition of knowledge from past experience to enable a problem solver to perform the same or similar tasks more effectively and efficiently in the future [12]. This learning uses prior knowledge and deductive reasoning to augment the information provided by the training examples, so that it is not subject to these same bounds as inductive learning methods. This paper considers an analytical learning called *Explanation-Based Learning* (EBL). In EBL method, prior knowledge is used to analyze, or *explain*, how each observed training example satisfies the target concept. This explanation is then used to distinguish the relevant features of the training example from the irrelevant, so that examples can be generalized based on logical rather than statistical reasoning. In recent years, EBL is successfully applied to learning search control rules for a variety planning and scheduling tasks.

Symbolic logic is one of the important knowledge representations in EBL. These structures can then be reasoned or inferenced upon by applying inference rules to derive various facts or deduce relationships in a logical manner. First order predicate calculus is widely accepted and employed in symbolic logic. Predicate calculus represents objects and their relationships in logical statements. Logical statements can be represented compound logical statements by logical connectives and quantifier. Predicate calculus can be expressed in the form of equation (1).

$$predicate\ (\arg_1, \arg_2, \arg_3, ..., \arg_n) \quad (1)$$

In the reasoning process, unification mechanism -- matched mechanism -- performs the substitution of the symbol constants. The process of unification is mechanized in programs which perform automated reasoning. PROLOG is used as an automated reasoning system in this paper.

First Order Horn Clauses [13] is a clause (a disjunction of literals) with at most one positive literal. It is an expression of a fragment of first-order logic which is expressed in equation (2).

$$\forall x_1, ..., x_n \bigwedge_{i=1}^{n} L_i(x_i) \rightarrow H(x) \quad (2)$$

Where $H, L_1, L_2, ..., L_n$ are positive literals. H is called the head or consequent of the Horn clause. The conjunction of literals $L_1, L_2 ... L_n$ is called the body or antecedents of the horn clause. The first order horn clause is used to represent complicate clauses such as domain theory and learned hypothesis of EBL learning algorithm in this paper.

III. The Proposed Approach of Flaw Detection

In this section, the proposed approach for design flaw detection is shown in Fig. 1. The approach shows seven steps of the proposed detection approach. The first four steps are in the *Representation phase* which is responsible for representing desired logic programs of design flaw detection. The last three steps are in the *Detection phase* which is responsible for detecting design flaws in the object-oriented software. Step 1 and Step 5 are generic and must be based on a representation set of elements and relations of object-oriented concepts. Step 2 to 4 must be followed when a new flaw is specified. Steps 5 to Step 7 are repeatable and must be applied on source code of software system.

A consideration is placed on explanation-based learning from domain theories that are perfect; that is, domain theories that are *correct* and *complete*. To show how to learn inferenced rules and detect design flaws by such rules, the proposed approach performs the following step-by-step. The proposed methodology is based on *PROLOG-EBG* learning algorithm. A known design flaws, *Data Class*, is chosen to illustrate for more concretely understanding in each step of the methodology. Each step is explained by clear presentation which based on common patterns: input, output, and description of each step.

A. Step 1: Arrangement

Input:

1. Domain theories which are derived from design principles and heuristics [2, 14].
2. Examples of design flaw.
3. Target concepts of design flaws used for learning.

All of information inputs are in the form of predicate calculus and Horn clause (declarative form).

Output: Related information of domain theories and a target concept which are sufficient to generate a detection rule from a specific training example in each learning cycle for generating a logic detection rule.

Description of step 1: The first step deals mainly with the arrangement of domain theories and a target concept to be learned for a training example in each learning cycle for generating a logic detection rule (Step 1 to Step 4). The process begins with a training example of *filterMap Data Class flaw* - shown in JAVA source code - in Fig. 2. In this source code example, the single training object, *FilterMap* class, which is taken from Tomcat's source code (*org.apache.catalina. deploy.FilterMap* class) and is a known positive example of a Data Class, is provided.

Then, domain theories related to a training example of filterMap Data Class flaw are defined. Such information is defined according to principles and heuristics of object-oriented paradigm [14-17]. For the purpose of flaw detection, domain theories are any set of prior beliefs about the object-oriented design and implementation principles, and an inference mechanism is any procedure that suggests new beliefs by combining existing beliefs. Learning concept descriptors are considered for a simplified Data Class. A suitable target concept and domain theories (formed in Horn clause) for this example of Data Class are given in Fig. 3.

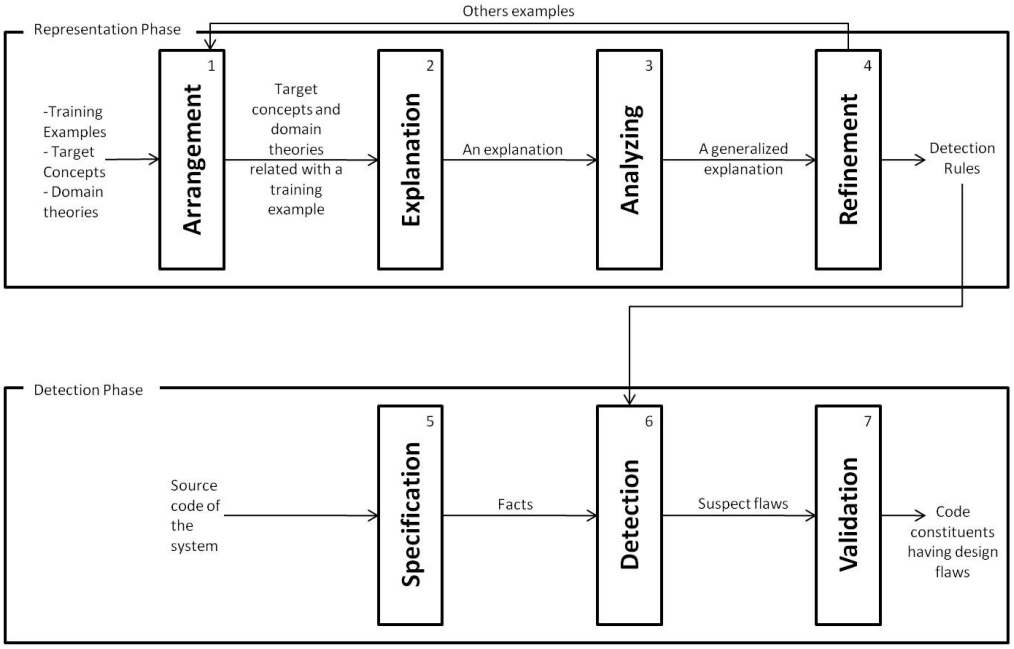

Fig. 1.The proposed approach of design flaw detection

B. Step 2: Explanation

Input: Domain theories and a target concept which relate to a training example in each learning cycle for generating a logic detection rule.

Output: An explanation h in the hypothesis space which h ($h \in H$) is consistent with domain theory B and $B \not\vdash \neg h$.

Description of step 2: Given the information in the first process, the second process is to determine a generalization of the training example which is a sufficient concept definition for the target concept. This process provides its *justification*: constructing an explanation (a proof tree) in terms of the domain theory that proves how the training example satisfies the target concept definition. Then, a set of sufficient conditions under which the explanation structure holds is determined. This determination is accomplished by regressing the target concept through the explanation structure. To see more concretely how the EBL approach works, consider learning the concept of Data Class in Fig. 4.

```
public class FilterMap implements Serializable {
    private String filterName = null;
    public String getFilterName() {
        return (this.filterName);
    }
    public void setFilterName(String filterName)
        this.filterName = filterName;
    }
    private String servletName = null;
    public String getServletName(){
        return (this.servletName);
    }
    public void setServletName(String servletName){
        this.servletName = servletname;
    }
}
```

Fig. 2.Class *FilterMap*

Given class filterMap as a positive example, EBL system attempts to construct an explanation for why class filterMap is indeed a Data Class flaw. From Fig. 4, arrows denote the contribution of each domain theory rule to the explanation. They point from a rule's antecedents to its consequences. For example from a rule in Fig. 3, which is also shown in Fig. 4, rule R8 allows the consequence property accessMethod to be concluded from the antecedents hasMethod, hasAttribute, returnType and parameter. Each pair of literals (nonitalic font) and generalized literals (italic font) between each step (dashed lines) in the explanation show expressions across rules that must match for the explanation to hold. These are enforced by unifying the connected expression.

Target concepts and domain theory:

R1: $\forall x\ \mathrm{class}(x) \wedge \neg \mathrm{not\text{-}dataclass}(x) \Rightarrow \mathrm{dataclass}(x)$

R2: $\forall x\ \forall y\ \mathrm{not\text{-}dataclass}(y) \wedge \neg\ \mathrm{is}(x,y) \Rightarrow \neg \mathrm{not\text{-}dataclass}(x)$

R3: $\forall x\ \forall y\ \mathrm{hasMethod}(x,y) \wedge \mathrm{method\text{-}operation}(y) \Rightarrow \mathrm{not\text{-}dataclass}(x)$

R4: $\forall x\ \neg \mathrm{mutator\text{-}method}(x) \wedge \neg \mathrm{accessor\text{-}method}(x) \Rightarrow \mathrm{method\text{-}operation}(x)$

R5: $\forall x\ \forall y\ \mathrm{accessor\text{-}method}(y) \wedge \neg \mathrm{is}(x,y) \Rightarrow \neg \mathrm{accessor\text{-}method}(x))$

R6: $\forall x\ \forall y\ \mathrm{mutator\text{-}method}(y) \wedge \neg \mathrm{is}(x,y) \Rightarrow \neg \mathrm{mutator\text{-}method}(x))$

R7: $\forall x\ \forall y\ \forall z\ \mathrm{has\text{-}attribute}(z,y) \wedge \mathrm{has\text{-}method}(z,x) \wedge \mathrm{method\text{-}returntype}(x,[VOID,NULL]) \wedge \mathrm{method\text{-}parameter}(x,[\{y,\_\}]) \Rightarrow \mathrm{mutator\text{-}method}(x)$

R8: $\forall x\ \forall y\ \forall z\ \mathrm{has\text{-}attribute}(z,y) \wedge \mathrm{has\text{-}method}(z,x) \wedge \mathrm{method\text{-}returntype}(x,[y,\_]) \wedge \mathrm{method\text{-}parameter}(x,[\{NULL,NULL\}]) \Rightarrow \mathrm{accessor\text{-}method}(x)$

Fig. 3.The *filterMap Data Class* domain theory

C. Step 3: Analyzing

Input: An explanation of a training example.

Output: A logic detection rule from an explanation.

Description of step 3: The explanation constructed by the previous step is generalized with a rule that is the most generally relevant to the target concept. EBL computes the most general rule that can be justified by the explanation, by computing the *weakest preimage* of the explanation. The weakest preimage of the target concept is computed by a general procedure called *regression* [18]. The regression procedure operates on a domain theory represented by an

arbitrary set of Horn clause. It works iteratively backward through the explanation, first computing the weakest preimage of the target concept with respect to the final proof step in the explanation, then computing the weakest of the resulting expressions with respect to the preceding step, and so on. The procedure terminates when it has iterated over all steps in the explanation. The illustrated example of accessMethod rule is shown in Fig. 5. The negation-as-failure approach is used to detect a Data Class flaw. The final rule for the current example is illustrated in Fig. 6.

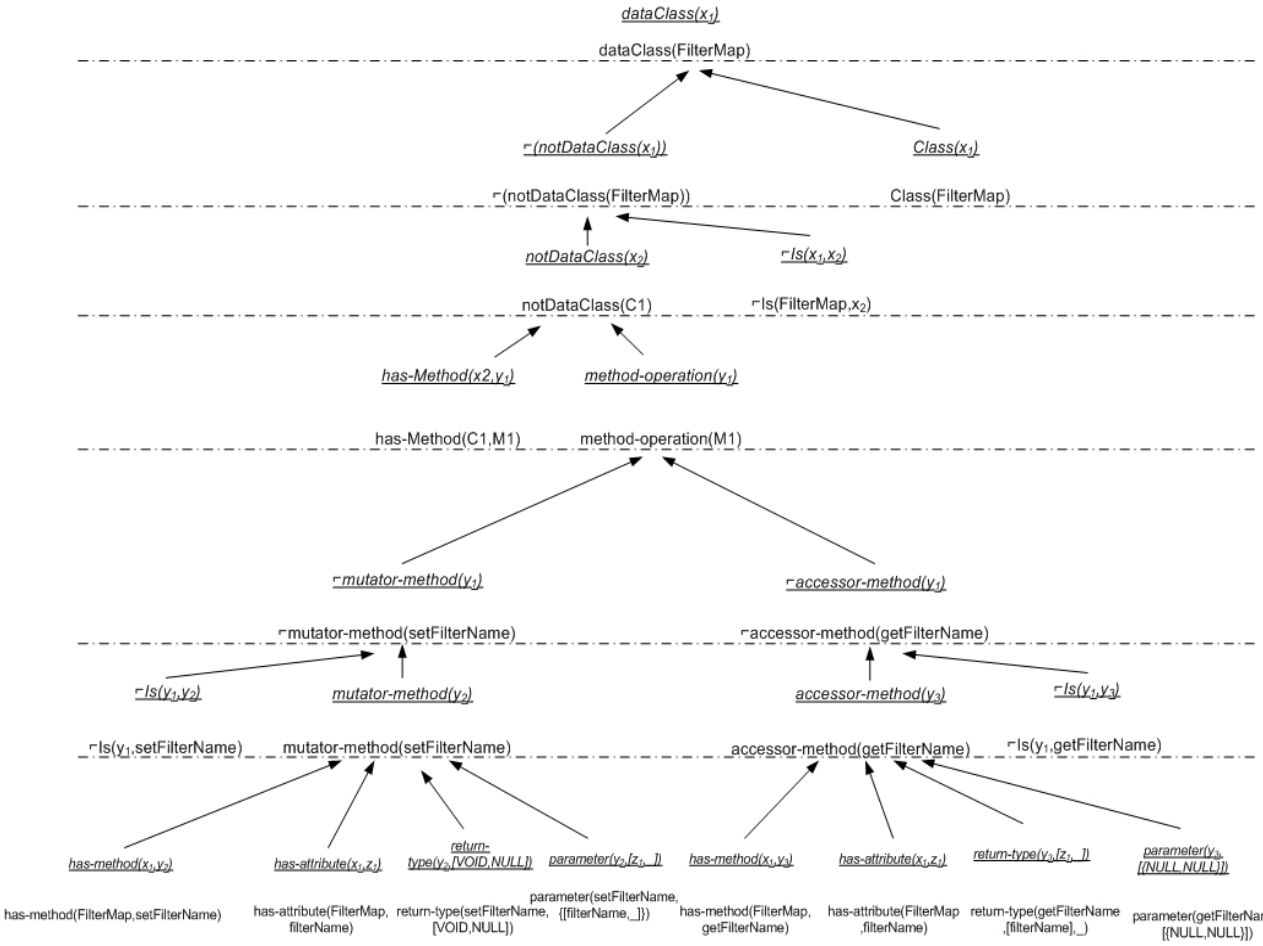

Fig. 4.An explanation for the training example *FilterMap*.

D. Step 4: Refinement

Input: A logic rule from the proof tree.
Output: New formulated rules.
Description of step 4: In this step, logic rules are refined by generalizing rules in accordance with a logic rule from the proof tree. Rules are pruned some literals for making generalization.

At each learning cycle for generating a logic detection rule (Step 1 - 4), the sequential covering algorithm of EBL learning algorithm picks a new positive example that is not yet covered by the current Horn clause rules, explains the new example, and formulates a new rule according to the learning cycle. When more learning examples are provided, the rule of Data Class flaw is refined to be more general that the attribute of mutator method and accessor method cannot be the same attribute.

E. Step 5: Specification

Input: Object-oriented source codes which need to be detected for design flaws.
Output: Logic facts which belong to AST specification.

Description of step 5: In this step, source code is parsed and formed in *Abstract Syntax Tree* (AST). It represents syntactical and semantical information of source code. The representation of source code is a tree of nodes which represents constants or variables (leave node) and operators or statements (inner nodes). These trees are transformed to logic facts. The argument in predicate calculus is represented by leave node and predicate part is represented by inner nodes of AST respectively. The number in each fact is used to represent relations among elements.

REGRESS(*Frontier, Rule, Literal*,θ_{hi}) where
Frontier = $Class(x_1)$, $\neg Is(x_1,x_2)$, $hasMethod(x_2,y_1)$,$methodOperation(y_1)$
Rule = $methodOperation(z) \leftarrow \neg mutatorMethod\ (z) \wedge \neg accessorMethod(z)$
Literal = $methodOpearation(y_1)$
σ_{hi} = [z/SetFilterName]
head ← $methodOperation(z)$
body ← $\neg mutatorMethod\ (z) \wedge \neg accessorMethod(z)$
$\sigma_{hl} \leftarrow [z/y]$,where σ_{1i} = [y/SetFilterName]
Return $Class(x_1)$, $\neg Is(x_1,x_2)$, $hasMethod(x_2,y_1)$,$\neg mutatorMethod\ (y_1)$, $\neg accessorMethod(y_1)$

Fig. 5.An example for regressing a set of literals given by *Frontier* through *methodOperation* Rule.

F. Step 6: Detection

Input: Logic facts and formulated logic rules.

Output: Results of the detection in each flaw.

Description of step 6: The detection of design flaw takes place in this step. The source code used for design flaw detection is transformed to first order logic facts. The detection performs in declarative programming environment by using pattern matching mechanism between facts and rules. Backward chaining mechanism in this environment performs searching to get the results. EBL learning algorithm halts once it finds the first valid proof.

$$\begin{aligned}&\text{dataCLass}(x_1) \leftarrow \text{Class}(x_1) \wedge \neg\ \text{Is}(x_1, x_2) \wedge \\ &\text{hasMethod}(x_2, y_1) \wedge \neg\ \text{Is}(y_1, y_2) \wedge \text{hasMethod}(x_1, y_2) \wedge \\ &\text{hasAttribute}(x_1, z_1) \wedge \text{returnType}(y_2, [\text{VOID}, \text{NULL}]) \wedge \\ &\text{parameter}(y_2, [z_1, \_]) \wedge \neg\ \text{Is}(y_1, y_3) \wedge \\ &\text{hasMethod}(x_1, y_3) \wedge \text{hasAttribute}(x_1, z_1) \wedge \\ &\text{returnType}(y_3, [z_1, \_]) \wedge \text{parameter}(y_3, [\{\text{NULL}, \text{NULL}\}])\end{aligned}$$

Fig. 6.he final rule of filterMap Data Class

G. Step 7: Validation

Input: Results of the detection.

Output: Precision and other rates of the detection by the proposed approach.

Description of step 7: Results of the proposed detection approach are validated by analyzing the suspicious types in the context of the complete model of the system and its environment. The validation is inherently a manual task. Therefore, the detection of a few design flaws in different behavioral types is chosen to apply.

Precision is used to show the number of true identified flaws, and *false positive* for the number of false positive missed by the detection. Two such rates are shown in the equation (3) and (4).

$$Precision = \frac{true\ \ positive}{true\ \ positive + false\ \ positive} \quad (3)$$

$$False\ \ positive = \frac{false\ \ positive}{true\ \ positive + false\ \ positive} \quad (4)$$

IV. Experiment and Discussion

This section starts with brief description of the model in the real detection by creating the prototype. Then, some case studies on various-sized programs are presented. Some interesting points from the proposed approach are also discussed.

A. Prototype

The aim is to validate the detection approach by specifying and detecting many varied types of flaws and computing the precision and some rates of the generated prototype on *CommonCLI v1.0* and *JUNIT v1.3.6*, two open source systems:

CommonCLI v1.0 is the Apache Commons CLI library. It provides an API for parsing command line options passed to programs. It contains 18 classes and 4132 lines of code. JUNIT v1.3.6 is a testing framework which performs automated testing. It contains 111 classes and 5000 lines of code.

The prototype model is implemented for design flaw detection. *Eclipse v3.6 HERIOS*, *Prolog Development Tools v0.2.3* and *SWI-Prolog v5.8.3* are used for implementing the prototype. All computations are performed on an *Intel* platform - i5 Intel at 2.44 GHz with 4 GB of RAM.

B. Results and Discussions

Fig. 7 and Fig. 8 report numbers of true flaws and true detection of six flaws, respectively for the CommonCLI and JUNIT systems. Fig. 9 shows precision rates and false positive rates of six flaws for such two software systems. Fig. 10 shows Data Class flaw detection of the proposed approach while comparing results with other approaches.

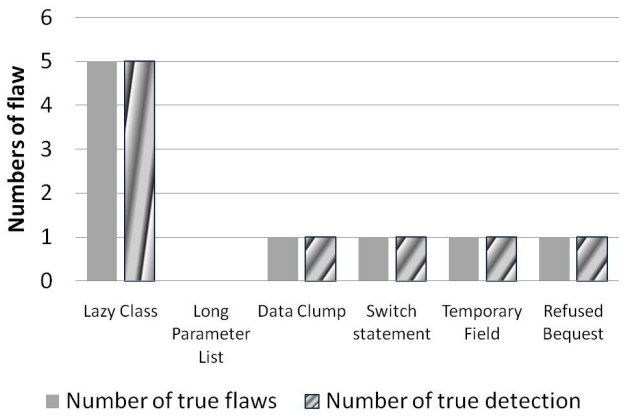

Fig.7.The result of true flaws and true detection with six flaws in CommonCLI v1.0

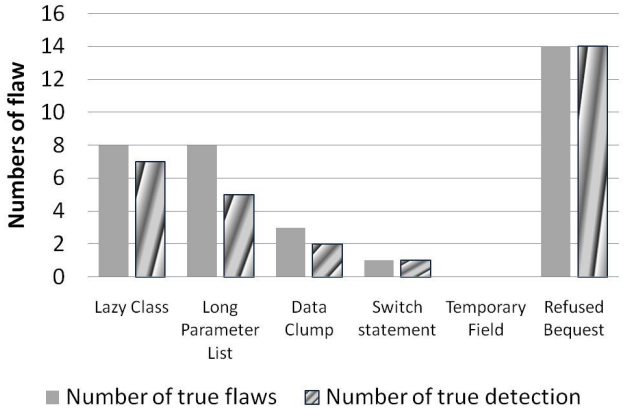

Fig. 8. The result of true flaws and true detection with six flaws in JUNIT v1.3.6

After analyzing the result data, two important points can be made for discussion:

Result data from the proposed approach: The validation shows that the proposed approach of design flaw detection with expected precision and good false positive.

In the details of Fig. 7 to Fig. 9, the proposed detection is performed on six types of design flaw. With *Lazy Class* and *Data Clump* are 100% of precision rate except *Long Parameter List* which does not exist in CommonCLI. As well as the last three flaws -- *Switch Statement, Temporary Field* and *Refuse Bequest* -- which are 100% precision rate. In the details of flaw detection of JUNIT, precision rates in flaw detection are 87.50%, 62.50% and 66.66% of *Lazy Class, Long Parameter List* and *Data Clump* respectively. It is 100% of precision rate of all flaws for the last three flaws. The results from the detection have, mainly, an average

precision rate in detection of 100% with the flaw detection in CommonCLI and 85% with the flaw detection in JUNIT.

For Fig. 10, the proposed approach allows locating a Data Class flaw by the highest precision with 98.41% -- equal to metrics approach in technique III and more than two generic metrics which are 96.82% and 96.82% in approach I and II respectively. These results show that the proposed methodology can detect design flaws more conveniently than other approaches in case of automatic detection, and it is effective in case that the results of precision are not inferior to other approaches. The prototype still has a false positive with the minimum rate with 1.58% when compared with other approaches. This result also supports that with rather high precision, the proposed approach still has the satisfied low false positive. It is convincing that with characteristics of rules from deduction learning (rules derived from deduction learning from exact flaws), rate of false positive is guaranteed in a low level or the level can be bounded.

Threats to the validity: The validity of the results depends directly on the flaw specifications of rules. Experiments are performed on more representative examples of flaw to lessen the threat to the internal validity of the validation. However, 100% precision rate of the proposed methodology is difficult to obtain with complex design flaws because such flaws typically are subjective (although reasonable). The threat to external validity is related to exclusive use of open-source system, which may limit the generalization of results to other systems. The subjective nature of interpreting and specifying flaws and related domain theory is a threat to construct validity of this validation.

V. Related Works

Surveyed research works are directly related to definition of design flaws, the detection techniques and tools. Design flaws in the context of object-oriented programming are first introduced by Webster's book [19]. The book contributes to conceptual, political, coding and quality-assurance problems.

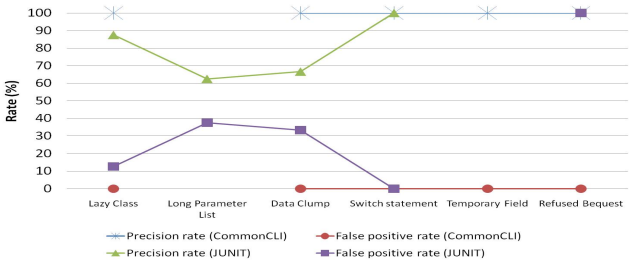

Fig. 9. Precision rates and false positive rates of six flaws in CommonCLI v1.0 and JUNIT v1.3.6

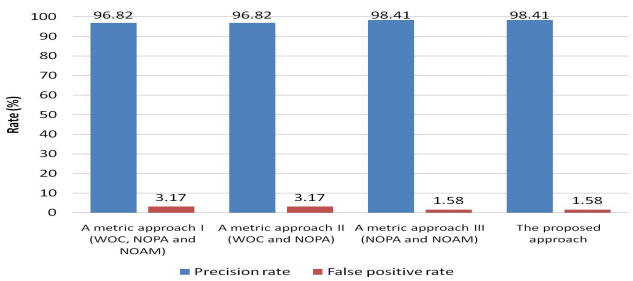

Fig. 10. Precision and false positive of Data Class detection with other detection techniques in JUNIT.

Riel [2] defined some groups of heuristics characterizing good object-oriented programming to assess a system quality manually for improving its design and implementation. Fowler [6] proposed 22 low-level *code smells*. These smells are in design problems in source code, and they should be removed by *Refactoring*. These smells describe in the informal way and need interpretation with a manual detection process. The classification of design flaws is proposed in many works [20, 21]. These taxonomies are introduced to provide better understanding of the smells and to recognize the relationships between smells. By these classifications, researchers can improve the precision of design flaws detection. Brown *et al.* [7] define 40 antipatterns which are described in-depth in terms of code smells-like description. They are the basis of all approaches to specify and to detect code smell semi automatically and automatically.

Several works propose approaches to specify and detect code smell and antipatterns. The detection in early age is in manual approach, based on software inspection techniques on text-based descriptions [22-24]. Software inspection involves carefully examining the code, the design, and the documentation of software and checking them for aspects that are known to be potentially problematic based on past experience. It is generally accepted that the cost of repairing a flaw is much lower when that flaw is found early in the development cycle. However, manual inspection is time-consuming process and strongly depends on developer's programming expertise. For lessening this problem, software visualization [25] is used to support flaw detection. This strategy allows reducing the search space, time- and resource-consumption and compensates for human intervention.

One famous and important technique to detect design flaws for improving software quality is using software metric. Many pieces of literature suggest the assistance of metrics in the detection of design flaws. Simon *et al.* [26] use object-oriented metrics to identify Bad Smells and also propose covered refactoring to improve the design of the code. Marinescu [10] presents a metric-based approach to detect design flaws with *Detection Strategies* mechanism, formulating metrics-based rules that capture deviations from good design principles and heuristics and combining metrics with set of operators. Improving accuracy detection of metric-based techniques in design flaw detection are supported by the *Tuning Machine* method [27], based on a genetic algorithm, which tries to find automatically the proper threshold values. However, design flaws cannot be directly measured by software metrics. Consequently metric-based techniques translate a flaw into measurable code properties which are thought to be related to the flaw. This technique is insufficient to precisely identify design flaws [28].

VI. Conclusion and Future Works

In this paper, the design flaw identification methodology by using analytical learning called *Explanation-Based Learning* (EBL) technique has been presented. The explanation interprets examples of the interesting design flaw, bringing into focus alternative coherent sets of features

which might be important for correct classification. With this proposed detection, rules generated from EBL can detect flaws more conveniently than other approaches in case of automatic detection and ignorance of proper threshold consideration and be effectively in case of the results of precision is good rate, and it is not inferior to other approaches.

The validation in term of precision and false positive sets some future positions. A future plan is to perform such a comparison of this work with previous approaches in different design flaws. Another future plan is to look for more domain theory information which can support inference mechanism of EBL to increase detection accuracy rate. There will also be the implementation of a tool and the study the meta model of design flaw logic representation in term of further specification of detection model.

REFERENCES

[1] F. M. Bravo, et al., "A Logic Meta-Programming Framework for Supporting the Refactoring Process," Master's Thesis, Vrije Universiteit, Brussel, 2003.

[2] A. J. Riel, Object-oriented design heuristics: Addison-Wesley Pub. Co., 1996.

[3] E. Gamma, et al., Design Patterns: Elements of Reusable Object-Oriented Software: Addison-Wesley Professional, 1994.

[4] M. Radu, "Measurement and Quality in Object-Oriented Design," 2005, pp. 701-704.

[5] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," J. Syst. Softw., vol. 80, pp. 1120-1128, 2007.

[6] M. Fowler and K. Beck, Refactoring: improving the design of existing code: Addison-Wesley, 1999.

[7] W. J. Brown, et al., AntiPatterns: refactoring software, architectures, and projects in crisis: Wiley, 1998.

[8] T. Gilb, et al., Software inspection: Addison-Wesley, 1993.

[9] M. V. Mantyla, et al., "Bad Smells " Humans as Code Critics," presented at the Proceedings of the 20th IEEE International Conference on Software Maintenance, 2004.

[10] R. Marinescu, "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws," presented at the Proceedings of the 20th IEEE International Conference on Software Maintenance, 2004.

[11] M. Lanza, et al., Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems: Springer, 2010.

[12] T. M. Mitchell, Machine Learning: McGraw-Hill, 1997.

[13] A. Horn, "On Sentences Which are True of Direct Unions of Algebras," Journal of Symbolic Logic, vol. 16, pp. 14-21, 1951.

[14] P. Coad and E. Yourdon, Object-oriented analysis: Yourdon Press, 1991.

[15] J. Lakos, Large-scale C++ software design: Addison-Wesley Pub. Co., 1996.

[16] R. E. Johnson and B. Foote, "Designing Reusable Classes," Journal of Object-Oriented Programming, vol. 1, pp. 22-35, 1988.

[17] S. L. Pfleeger and J. M. Atlee, Software engineering: theory and practice: Prentice Hall, 2009.

[18] R. Waldinger, et al., Achieving several goals simultaneously vol. 8: Wiley, 1977.

[19] B. Webster, Pitfalls of Object Oriented Development: M \& T Books, 1995.

[20] M. Mantyla, et al., "A Taxonomy and an Initial Empirical Study of Bad Smells in Code," presented at the Proceedings of the International Conference on Software Maintenance, 2003.

[21] M. Zhang, "Improving the Precision of Fowler's Definitions of Bad Smells," 2008, pp. 161-166.

[22] G. Travassos, et al., "Detecting defects in object-oriented designs: using reading techniques to increase software quality," SIGPLAN Not., vol. 34, pp. 47-56, 1999.

[23] M. E. Fagan, "Advances in software inspections," in Software pioneers, ed: Springer-Verlag New York, Inc., 2002, pp. 609-630.

[24] M. Fagan, "Design and code inspections to reduce errors in program development," in Software pioneers, ed: Springer-Verlag New York, Inc., 2002, pp. 575-607.

[25] G. Langelier, et al., "Visualization-based analysis of quality for large-scale software systems," presented at the Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, Long Beach, CA, USA, 2005.

[26] F. Simon, "Metrics Based Refactoring," 2001, pp. 30-30.

[27] P. F. Mihancea, "Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems," 2005, pp. 92-101.

[28] N. Moha, et al., "Automatic Generation of Detection Algorithms for Design Defects," presented at the Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, 2006.

Sakorn Mekruksavanich received the B.Eng. degree in Computer Engineering from Chiangmai University in 1999. He received M.S in Computer Science from King Mongkut's Institute of Technology Ladkrabang in 2004. Currently, he is a Ph.D. candidate of Chulalongkorn University, Thailand. His research interests are software engineering and software quality improvement.