http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 11 (12): 1677-1686, 2012
ISSN 1812-5638 / DOI: 10.3923/itj.2012.1677.1686
© 2012 Asian Network for Scientific Information

Analytical Learning Based on a Meta-programming Approach for the Detection of Object-oriented Design Defects

[1]Sakorn Mekruksavanich, [2]Preecha P. Yupapin and [1]Pornsiri Muenchaisri
[1]Department of Computer Engineering, Chulalongkorn University, Bangkok 10330, Thailand
[2]Faculty of Science, King Mongkut's Institute of Technology Ladkrabang, Bangkok 10520, Thailand

Abstract: This study proposed a new defect-detection approach using a declarative meta-programming technique to analytical learning for object-oriented software. The extrapolating patterns are generated using analytical learning in which certain design defect characteristics can be understood through deductive learning. This study uses declarative meta-programming to represent the specific object-oriented components as logic rules with which design defects can finally be described. Using the two complementary techniques, the object-oriented software is transformed into the narrow related problem domain, in which design defect problems can be managed and simplified. The approach is validated by detecting design defects in certain open-source systems. The results obtained exhibit a superior precision to the conventional method. In application, the proposed strategy can be recognised as a flexible and automated system for detecting software design defects which many object-oriented software systems are able to use.

Key words: Analytical learning, defect detection, antipatterns, code smell, meta-programming

INTRODUCTION

Design defects play an important role in the software development lifecycle. These defects stem from poor internal organisation design in an object-oriented software system and they negatively impact essential quality factors, including maintainability, understandability and ease of evolution (Fowler, 1999; Mens and Tourwe, 2004; Bettenburg *et al*., 2012; Shu *et al*., 2009). Design defects commonly arise between the design and implementation levels of a software system (Changchien *et al*., 2002; Mekruksavanich and Muenchaisri, 2011). These defects may become a concern at the design level, but they concretely manifest themselves in the source code with specific implementations. Design defects are usually known as code smells (Fowler, 1999; Riel, 1996) or antipatterns (Brown, 1998) and many researchers use metaphors to describe design defects. Design defect specifications are often loosely defined because quality assessment is a human-centric process that requires contextual data. There is always a degree of uncertainty about whether any components in a programme are defective.

Detecting design defects can substantially reduce the cost and time of subsequent activities in the software development and maintenance phases (Okeeffe and Ocinneide, 2008; Pressman, 2010; Khomh *et al*., 2012; Parthasarathy and Anbazhagan, 2006). In the last decade, several approaches have been proposed to specify and detect defects. Software inspection (Basili, 1996) with software visualisation (Sanatnama and Brahimi, 2010; Ali, 2011) was proposed to manually detect design defects. However, this approach entails several issues, including being time-expensive, non-repeatable and non-scalable (Biffl, 2003). Mantyla and Lassenius (2009) identified additional issues with manual defect detection. These authors showed that when a certain software system increases, even experienced developers' ability to perform an objective system evaluation of design defects decreases. To avoid some of the drawbacks of a purely manual detection approach, defect detections relying on specification rules were then proposed (Moha *et al*., 2010). The strategies used capture deviations from defined principles which involve quantitative rules with set operators and compare their values against threshold values (Vinayagasundaram and Srivatsa, 2007).

It may be difficult to express these defects as quantitative rules with high precision and low false-positive rates. Effective identification thus depends on using the proper metrics and thresholds to detect such defects (Ferreira *et al*., 2012). Although, many studies have investigated approaches to the detection of design defects, complete information about design defect detection using meta-programming with analytical learning is unavailable. The present paper proposes a new detection method using Explanation-Based Learning

Corresponding Author: Sakorn Mekruksavanich, Faculty of Engineering, Chulalongkorn University, Bangkok 10330, Thailand

(EBL) (DeJong, 2004) based on the declarative meta-programming (DMP) environment (Tourwe and Mens, 2003). This approach automatically eliminates the detection time consumption and avoids using the threshold values on which detection accuracy depends. The obtained results show the proposed method using both techniques to increase the precision of design defect detection. Furthermore, the proposed method can be used in many object-oriented applications, including general purpose software, system software and critical software systems.

THE PROPOSED DETECTION APPROACH

Figure 1 shows the meta-programming architecture for the proposed detection approach. The meta-level constitutes many facts and rules about design defects. Logic programming is used to write DMP meta-programmes because it is the most suitable for meta-programmes that search for design defects in the programme they process. The MB Interface (Meta-Base Interface) module, containing sixty-five specific MESs, analyses and transforms the syntactical and semantic information from the base programme into the meta-programme between the base and meta-level layers. The MES (Meta-Element Specification) representation of a base programme describes how the logic-based representation works. A meta-model of a meta-programme defines MES constituents to represent rules and set operators, the relationships among rules and the properties in a meta-programme.

Figure 2 provides an overview of the proposed methodology which describes the eight steps and summarises what is necessary to define the proposed detection approach. The first four steps in the representation phase are responsible for generating the desired logic programme rules from the learning mechanism to detect design defects. The four latter steps in the detection phase are responsible for detecting design defects in object-oriented software. All steps are performed in the defined meta-programming architecture environment. Steps 1 and 2 are generic and must be based on a representative set of elements and relations of object-oriented concepts. Steps 5 and 6 are also the same. Steps 3 and 4 must be followed when specifying a new defect. Steps 5-8 are repeatable and must be applied to each source code of the object-oriented software. A set of design defects proposed by Fowler (1999) is chosen to validate the proposed approach, as many studies use these defects. Furthermore, these defects are recent problems that occur in the software development industry. The following details the eight steps of the detection approach.

Step 1: Building block synthesis: All required components in a meta-programming environment, called Building Blocks (BBs), are identified utilising Domain analysis (Neighbors, 1980). This concept analyses related software systems in a specific domain to find their common and variable parts. Domain Analysis considers objects and relations in all systems in an application area. According to the definition of domain analysis activities, this step finds the BBs from the design defect description which follows the object-oriented paradigm. All related BBs are identified, including their basic relations for each input design defect description. These BBs and their

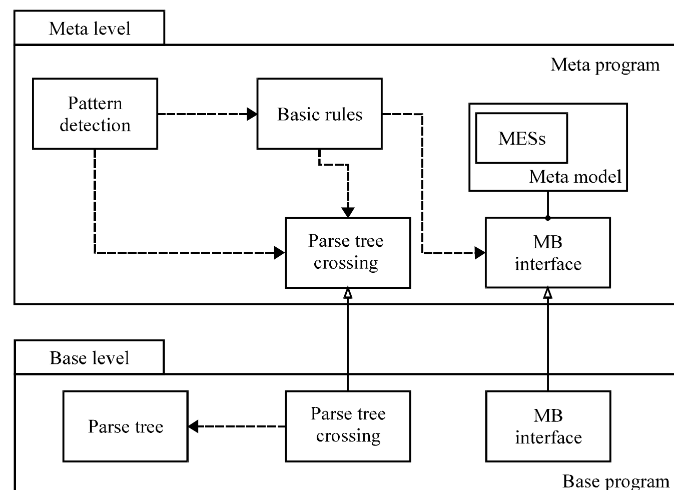

Fig. 1: The meta-programming architecture

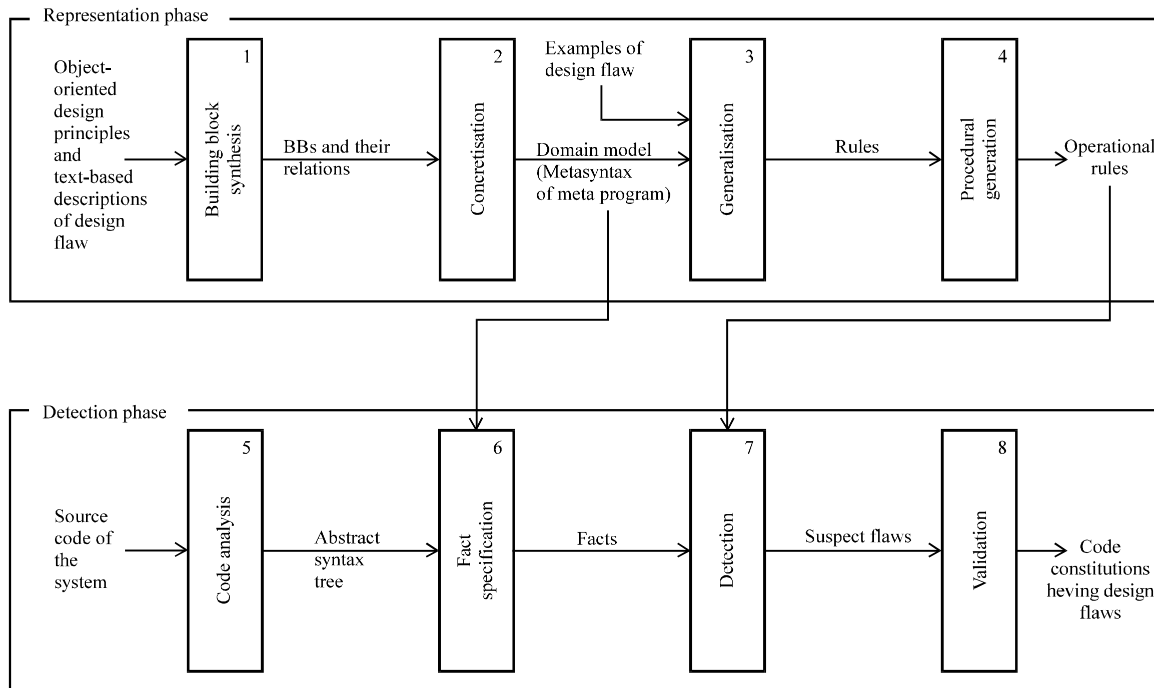

Fig. 2: The methodology for the proposed detection approach

relations refer to specific integrated concepts of object-oriented design and implementation that are used to describe design defects. The process in this step runs iteratively until all design defects of interest are provided. Newly detected BBs are compared to those found in previous iterations to avoid duplication. Eventually, a BB compilation is obtained and it expresses concise and unified components. A simple example of defect analysis to discover BBs (e.g., a text description of the data class defect) is explained in more detail below:

Discovering BBs in a data class

A text description: a class has the data class if the class has data fields and the only operations are the getting and setting operations. The existence of the data class indicates low-quality data abstraction. Classes that passively store data should be avoided and they should contain data and methods to operate on that data.

Data class analysis: this defect exists within a class granularity. All unnecessary details should not be considered. In the data abstraction, all attributes and methods that are logical for objects of a designed class must be part of the class. Both attribute and method elements should thus be considered. BBs and their relations are defined as follows:

- BBs: class, data, operation, getting operation and setting operation
- Relations among BBs: store, contain and operate

All text descriptions of each design defect are then analysed similarly (twenty defects). The detection domain of a set of eight BBs of the object programme is considered to define the meta-programme. The set of all BBs is defined as the relation of the direct product:

$$R_{BBs} = \{p\times c\times o\times a\times m\times s\}$$

where, BBs = {p: ℘PACKAGE, c: ℘CLASS, o: ℘OBJECT, a: ℘ATTRIBUTE, m: ℘METHOD, s: ℘STATEMENT}; R_{BBs} = {OWN, HAS, ISA, EXTEND, IMPLEMENT, INVOCATION, ACCESS FIELD, ASSIGN, EXECUTE} and ℘ represents the power set.

Step 2: Concretisation: This step considers all semantic-level parts related to the defined BBs. This step formalises the domain model of the object programme structures in a meta-programming architecture. The model is used to describe the design defect domain of the object programme. All related specifications of the description of the BBs and their abstract syntaxes which belong to the object-oriented paradigm, are formalised. When

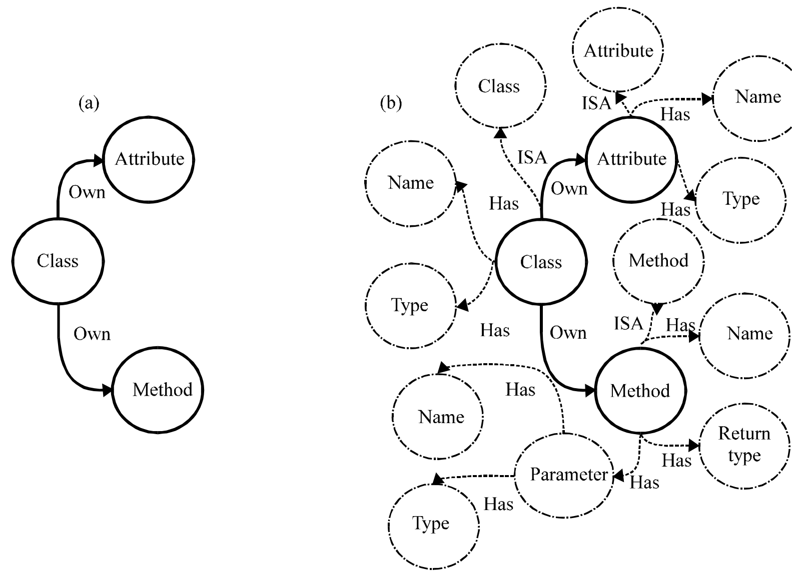

Fig. 3(a-b): Illustration of (a) Considered class and (b) Class concretisation

Target concepts and domain theory:

R1: $\forall x$ class(x) $\wedge$ $\neg$not−dataclass(x) $\Rightarrow$ dataclass(x)

R2: $\forall x$ $\forall y$ hasMethod(x,y) $\wedge$ method−operation(y) $\Rightarrow$ not−dataclass(x)

R3: $\forall x$ $\neg$mutator−method(x) $\wedge$ $\neg$accessor−method(x) $\Rightarrow$ method−operation(x)

R4: $\forall x$ $\forall y$ $\forall z$ has−attribute(z,y) $\wedge$ has−method(z,x) $\wedge$ method−returntype(x,*[VOID,NULL]*) $\wedge$ method−parameter(x,[{y,_}]) $\Rightarrow$ mutator−method(x)

R5: $\forall x$ $\forall y$ $\forall z$ has−attribute(z,y) $\wedge$ has−method(z,x) $\wedge$ method−returntype(x,[y,_]) $\wedge$ method−parameter(x,[{NULL,NULL}]) $\Rightarrow$ accessor−method(x)

Listing 1: Domain theories and a target data class concept

considering CLASS BBs, such a class is formalised as a descriptor of a set of objects with specific properties (according to the BBs {ATTRIBUTE, METHOD} and their relations to {OWN, HAS, ISA}) for all possible structures, behaviours and relationships. Figure 3a illustrates a considered class with a name, attributes and methods. Attributes also have names and types and methods have names, return types and parameters. Figure 3b shows the concretisation of a class.

All BBs and R_{BBs} are semantically constituted to be MESs in the MB Interface of a meta-programming architecture. The notation required for MESs in a domain model is graphically illustrated in the UML class diagram in Fig. 4 which is required to identify design defects for given source code programmes.

Step 3: Generalisation: Two ordered processes of this step involve the learning mechanism to extrapolate specific logic in a meta-programming architecture environment. The initial process defines the target concepts and domain theories to be learned. A generalisation technique then performs the inference learning process to define the combined detection rules. This process provides its justification: constructing an explanation (a proof tree) using the domain theory that proves how the training example satisfies the definition of the target concept. It then determines a set of sufficient conditions under which the explanation structure holds. This determination is accomplished by regressing the target concept against the explanation structure. A simple learning concept descriptor for a data class defect is considered below. Listing 1 defines a target concept and domain theories (formed in a Horn clause) in a meta-programming environment for data class defects.

A positive training example, depicting a set of data class training examples, is provided, namely, the FilterMap

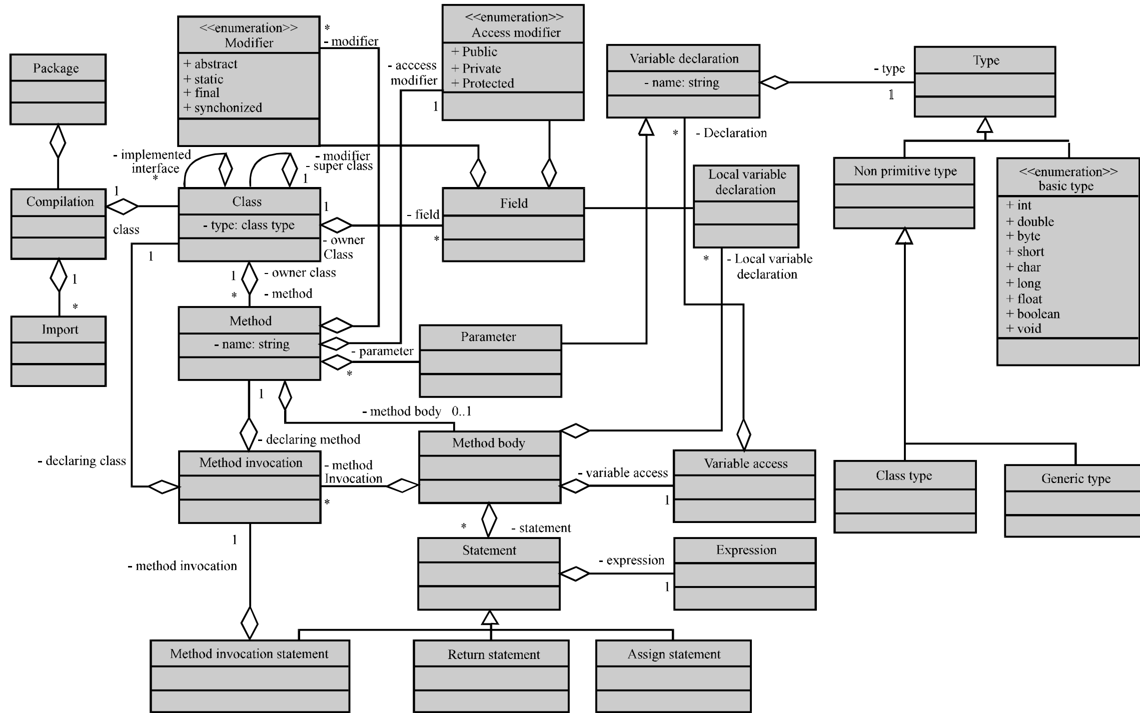

Fig. 4: A unified modelling language model notation of the domain model

```
Public class FilterMap implements serializable {
    private string FilterName = null;
    public string getFilterName ( )  {
        returu (this . FilterName);
    }
pubic void setFiltreName (string FilterName)
        This . FilterName = FilterName;
    }
private string servletName = null;
public string getservlet ( ) {
        returu (this . ServletName);
    }
pubic void setServletName (String servletName) {
        this . ServletName = servletName;
    }
}
```

Fig. 5: Class FilterMap

class which is taken from Tomcat's source code. Figure 5 shows the JAVA source code form for Filter Map. Figure 6 determines and explains the generalisation of FilterMap, domain theories and target concepts. The arrows denote the contribution of each domain theory rule to the explanation, pointing from an antecedent of the rule to its consequence. Rule R5 allows the consequent property accessMethod to be concluded from the antecedents hasMmethod, hasAttribute, returnType and parameter. The underlines indicate the expressions across rules that must match for the explanation to hold and be enforced by unifying the connected expression. EBL computes the most general rule that can be justified by computing the weakest preimage (Waldinger, 1977) of the explanation. Table 1 illustrates the accessMethod example. The negation-as-failure approach is maintained to detect the data class. The final rule for the current example is as follows:

Table 1: Regressing a set of literals given by Frontier using the method of operation rule

REGRESS(Frontier, Rule, Literal, θ_{hi}) where
Frontier = Class(x_1), ¬Is(x_1, x_2), hasMethod(x_2, y_1), methodOperation(y_1)
Rule = methodOperation(z) ← ¬mutatorMethod(z) ^ ¬accessMethod(z)
Literal = methodOperation(y_1)
σ_{hi} = [z/SetFilterName]
head ← methodOperation(z)
body ← ¬mutatorMethod(z) ^ ¬accessMethod(z)
σ_{hl} = [z/y], where σ_{li} = [z/SetFilterName]
Return Class(x_1), ¬Is(x_1,x_2), hasMethod(x_2,x_1), ¬mutatorMethod(y_1), ¬accessMethod(y_1)

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

For unselected examples of positive detection rules, the sequential covering algorithm picks these examples individually, explains the new example and formulates a

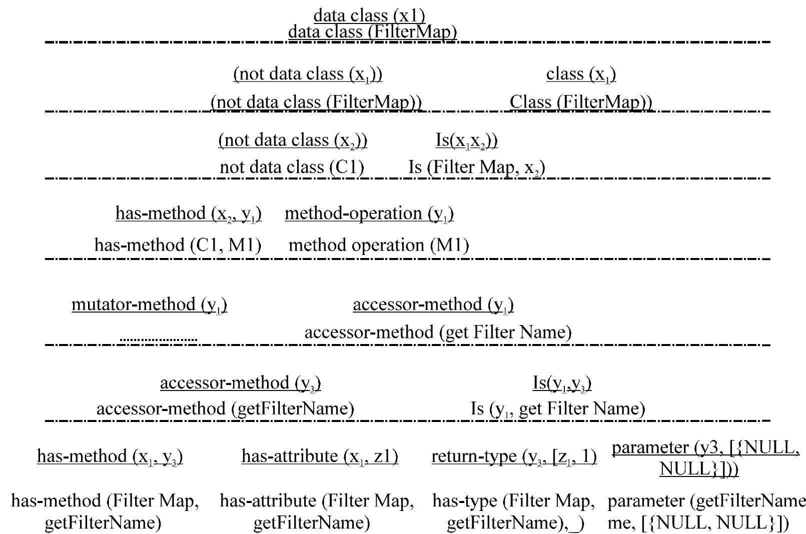

Fig. 6: Explanation of the training example Filter Map

new rule according to the previous processes. The detection rule for data class defects is refined to be more general and is given more learning examples to prevent the mutator and accessor method attributes from being the same.

Step 4: Procedure generation: The explanation trees derived from the EBL learning in the previous step provide several rules for detecting each design defect. This step introduces query transformation which optimises the first-order queries by reordering literals. The first-order queries become more efficient to execute if selective literals are placed first. Listing several requirements to reorder transformations for first-order queries occurs in the following contexts:

- **R1 (Correctness):** The reordering transformation should be correct, i.e., the transformed query should succeed (or fail) for the same examples as the original query
- **R2 (Disagreement):** The reordering transformation should minimise conflicts among literals when conflicts exist. The literal that most clearly reflects such a design defect is chosen in the initial order in query literals.

If sets of facts define all predicates, then the order of the literals does not influence the query result (cf., the switching lemma (Lloyd, 1993)) and the correctness requirement (R1) is met. With the disagreement requirement (R2), literals are inspected manually to classify the conflict levels. Assume two data class rules derived from the learning mechanism: (1) a class with accessor and mutator methods and (2) a class with a public field. Rule (1) is chosen for the first-order query because it can reflect more defects than rule (2).

Step 5: Code analysis: The object-oriented source code (Fig. 7a) is parsed and formed in an Abstract Syntax Tree (AST) which represents syntactical and semantic source code information. The tree nodes represent constants or variables (leaf node) and operators or statements (inner nodes). Figure 7b shows an example of parsing the source code in an AST.

Step 6: Fact specification: AST is transformed to logic facts in this step. This transform agrees with the MESs (Meta-Element Specifications) of the domain model specified in the meta-programming architecture. The argument in the predicate calculus is represented by leaf nodes and the predicate part is represented by inner AST nodes. Figure 7c illustrates an instance of parsing AST to logic facts. The defined numbers in each fact represent relations among elements.

Step 7: Detection: Detection occurs in the meta-programming environment using the pattern-matching mechanism of logic programming between facts and rules. A backward-chaining mechanism in this environment searches to obtain the results. The EBL learning algorithm halts when it finds the first valid proof.

Step 8: Validation: The proposed approach is validated by the results of suspicious-type detections in a complete model of the system and its environment. The validation is inherently a manual task. Thus, this step involves

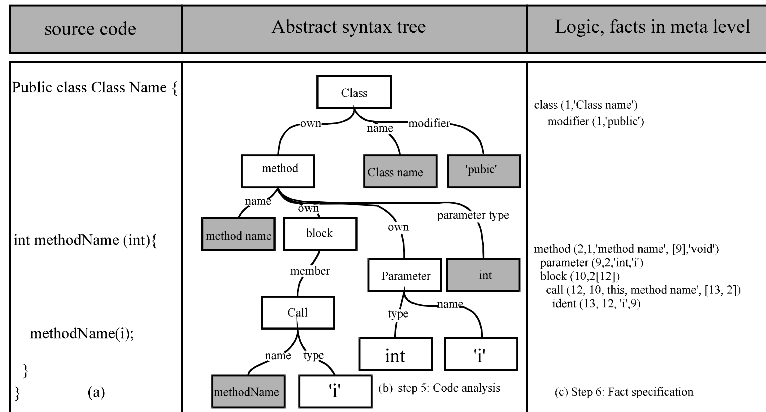

Fig. 7(a-c): Step 5: Code analysis and Step 6: Fact specification

prioritising the detection of certain design defects of different behavioural types. The validation measure uses the precision rate to assess the number of accurately identified defects and the recall rate to assess the number of true misses. Equation 1 and 2 show these rates:

$$\text{Precision} = \frac{\text{True positive}}{\text{True positive} + \text{False positive}} \quad (1)$$

$$\text{Recall} = \frac{\text{True positive}}{\text{True positive} + \text{False negative}} \quad (2)$$

Computing precision and recall is evaluated using independent results obtained manually because only humans can assess whether a suspicious class is indeed a defect or a false positive which depends on the system specifications, context and characteristics. The proposed approach aims to achieve high precision. Thus, the specificity rate is also used to validate the approach, as in Eq. 3:

$$\text{Specificity} = \frac{\text{True positive}}{\text{True negative} + \text{False positive}} \quad (3)$$

RESULTS AND DISCUSSION

This section briefly describes the proposed approach for real detection by creating a prototype. Several vital cases on various-sized detection programmes are presented and a few interesting points are discussed in greater detail.

Prototype Implementation: The prototype is implemented using Eclipse v3.6 HERIOS, Prolog Development Tools v0.2.3 and SWI-Prolog v5.8.3 tools. All computation is performed on an Intel platform, i.e., i5 Intel at 2.44 GHz with 4 GB of RAM. In the deductive learning mechanism, 121 training examples (Fowler, 1999; Brown, 1998; Riel, 1996) are used as a training dataset. This mechanism requires a certain number of training examples to achieve a given precision level. The 10-fold cross-validation technique is also used to optimise a model for fewer estimated errors.

The proposed approach uses three open-source systems for validation, namely, CommonCLI v1.0, JUNIT v1.3.6 and GANTTPROJECT v1.10.2. CommonCLI v1.0 is the Apache Commons CLI library which provides an API for parsing command line options to be passed to programmes. This library contains 18 classes and 4,132 lines of code. JUNIT v1.3.6 is an automated testing framework that contains 111 classes and 5,000 lines of code. GANTTPROJECT v1.10.2 is a project management tool used to plan projects with Gantt charts. This tool contains 21, 267 lines of code, 188 classes and 41 interfaces. Three measurement case studies are presented to make this validation robust: precision, recall and specificity.

Case study I: Precision: Figure 8 shows the results of the proposed approach by detecting six design defects (e.g., Lazy Class, Long Parameter List, Data Clump, Switch Statement, Temporary Field and Refuse Bequest) in both open-source software environments (CommonCLI v1.0 and JUNIT v1.3.6). All results (except Long Parameter List which did not exist) showed a 100% precision rate in CommonCLI v1.0. For JUNIT v1.3.6 (except Temporary Field), the proposed approach achieved the maximum rate at 100% and the minimum rate at 62.50%. The experiments became more valuable if they were conducted

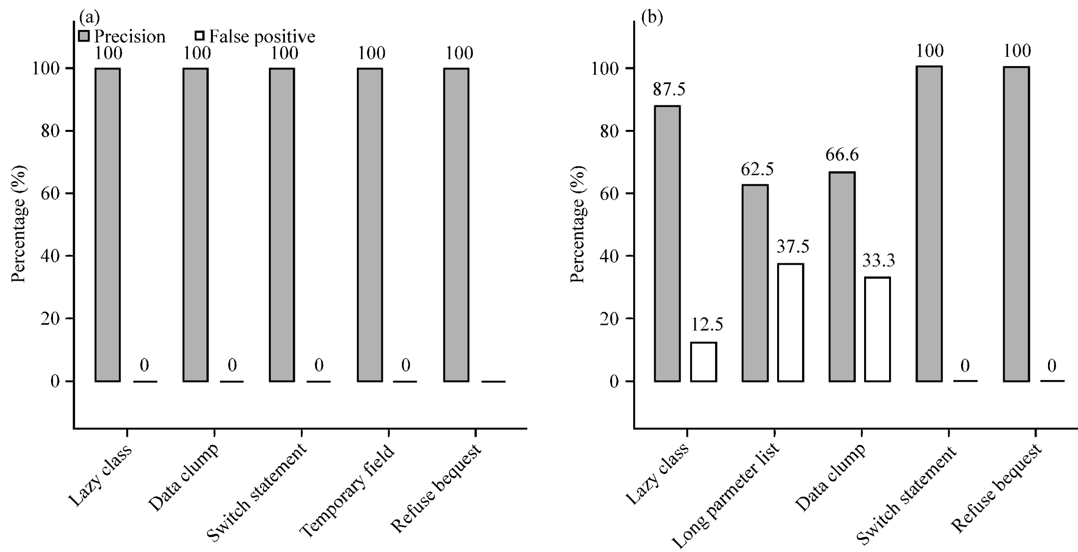

Fig. 8(a-b): The results for precision rate and false-positive rate in (a) CommonCLI v1.0 and (b) JUNIT v1.3.6

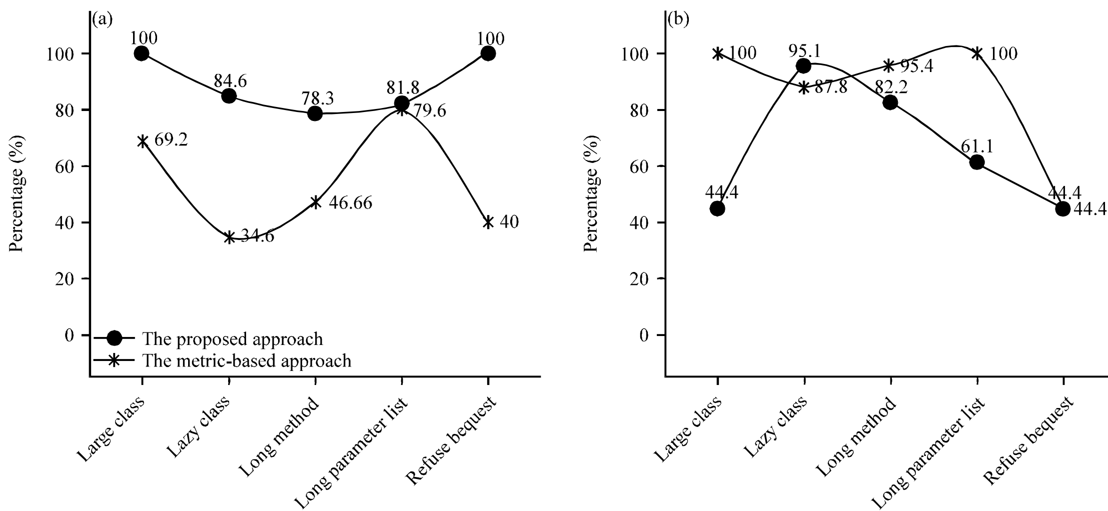

Fig. 9(a-b): The rate of (a) Precision and (b) Recall for the proposed and metric-based approaches on GANTTPROJECT v1.10.2

several times by comparing the precision with DECOR (Moha *et al*., 2010), the recent metric-based approach; Fig. 9(a) shows five results from GANTTPROJECT v1.10.2. These results provide the maximum, minimum and average rates of 100, 78.37 and 88.95%, respectively.

The overall results indicate that the gained precision rates are absolutely superior to the conventional detection approaches. Due to the fine-grained elements in the meta-programming architecture using deductive learning with real examples, the derived detection rules of the proposed approach contain remarkable logic features that are suitable for defect characteristics. Thus, high precision can be expected when detecting defects, with an average precision of 88.95%. However, the subjective defect level inevitably affects detection levels. The logic rule for detection may suffer from the sharpness of deduction, in which logic rules from proof trees have high specialisation levels. Accordingly, defining the most general rules that still enable the detection of defects in such a context is a difficult task. The proposed approach can handle this major problem by applying the tree pruning technique and the minimum precision is still over 78%. An instance of optimum precision is obtained from GANTTPROJECT when detecting Large Class, as the proposed approach can use many feature logic rules to describe this defect entirely. Some Long Method

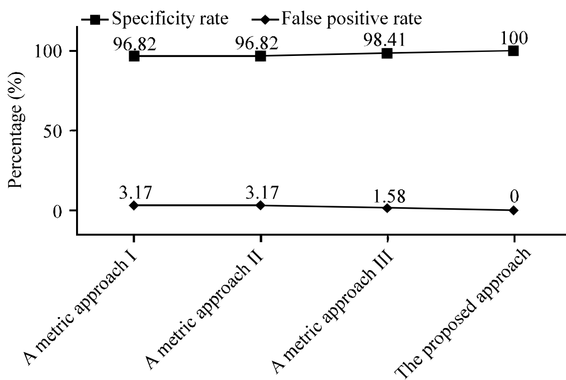

Fig. 10: Specificity and its false-positive rate in data class detection compared to other detection techniques in JUNIT v1.3.6

defects also become subjective cases in detected defects. Decision-making to designate defects depends on the software reviewers' investigations and this case does not cover the proposed detection rules.

Case study II: Recall: For detecting GANTTPROJECT in Fig. 9(b), the maximum and minimum recall rates were 95.12% and 44.44%, respectively. The proposed approach technique is based on MESs of BBs and R_{BBs} in the meta-programming architecture and it considers only skeletal MESs that are composed to describe such defects. The meta-programming environment can then provide expected recall values, provided the subject of detection remains the object-oriented paradigm. In this case, 39 suspicious defects are found for the Lazy Class and an expected recall rate is acquired from this defect in detecting GANTTPROJECT. However, the derived minimum recall, Refused Bequest, differs. This defect illustrates the inverse problem: it is difficult for software engineers to identify all of its occurrences, as they must appreciate whether a class uses appropriate public and protected methods/fields for any of its superclasses. Moreover, software engineers always consider bequests provided by library classes, but the proposed approach is applied only on the chosen software system without considering libraries.

Case study III: Specificity: Because the proposed approach results in the high precision rates, it also considers specificity in defect detection to indicate true positive and true negative defects. Scrupulous detection rules from both techniques, i.e., DMP and EBL, further promote this advantage. When detecting 63 true negatives with JUNIT v1.3.6 in Fig. 10, the results show a 100% precision rate for detecting true-negative defects with the proposed approach. Compared to three metric-based approaches, the proposed approach presents a higher specificity rate.

CONCLUSIONS

This study proposes an interesting approach to detecting object-oriented design defects. Using analytical learning and declarative meta-programming techniques, the design defects can be formulated and simply and logically described. The results presented in this study show that the proposed approach evaluated by both techniques can detect defects with a good average precision rate of over 88.95%; an expected recall rate is also acquired with an average recall rate of 76.55%. Moreover, the experiments are conducted and compared with general detections and the results provide superior rates to conventional approaches. This simple method can be applied to other design defect paradigms by manipulating the meta-programming architecture which potentially provides a quality guideline for software engineers in software construction. Explanation-based learning based on meta-programming can be used in further investigations, including software refactoring, software clone detection and software testing.

REFERENCES

Ali, J., 2011. Object visualization support for learning data structures. Inform. Technol. J., 10: 485-498.

Basili, V.R., 1996. The Empirical Investigation of Perspective-Based Reading. University of Maryland, UK., Pages: 76.

Bettenburg, N., W. Shang, W. Ibrahim, B. Adams, Y. Zou and A.E. Hassan, 2012. An empirical study on inconsistent changes to code clones at the release level. Sci. Comp. Program., 77: 760-776.

Biffl, S., 2003. Evaluating defect estimation models with major defects. J. Syst. Softw., 65: 13-29.

Brown, W.J., 1998. Anti-Patterns: Refactoring Software, Architectures and Projects in Crisis. Wiley, USA., ISBN-13: 9780471197133, Pages: 309.

Changchien, S.W., J.J. Shen and T.Y. Lin, 2002. A preliminary correctness evaluation model of object-oriented software based on UML. J. Applied Sci., 2: 356-365.

DeJong, G., 2004. Explanation-Based Learning. In: Computer Science Handbook, Tucker, A. (Ed.). Chapman and Hall/CRC Press, Boca Raton, FL., USA., pp: 68.1-68.18.

Ferreira, K.A.M., A.S.M. Bigonha, R.S. Bigonha, L.F.O. Mendes and H.C. Almeida, 2012. Identifying thresholds for object-oriented software metrics. J. Syst. Soft., 85: 244-257.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, New York, USA., ISBN-13: 9780201485677, Pages: 431.

Khomh, F., M.D. Penta, Y.G. Gueheneuc and G. Antoniol, 2012. An exploratory study of the impact of antipatterns on class change and fault-proneness. Empirical Softw. Eng., 17: 243-275.

Lloyd, J.W., 1993. Foundations of Logic Programming. Springer-Verlag, New York.

Mantyla, M.M. and C. Lassenius, 2009. What types of defects are really discovered in code reviews? IEEE Trans. Softw. Eng., 35: 430-448.

Mekruksavanich, S. and P. Muenchaisri, 2011. A meta-programming and machine learning for detecting object-oriented software design flaws. Ph.D. Thesis, Chulalongkorn University, Thailand.

Mens, T. and T. Tourwe, 2004. A survey of software refactoring. IEEE Trans. Softw. Eng., 30: 126-139.

Moha, N., Y. Gueheneuc, L. Duchien and A.F.L. Meur, 2010. Decor: A method for the specication and detection of code and design smells. IEEE Trans. Softw. Eng., 36: 20-36.

Neighbors, J.M., 1980. Software construction using components. Ph.D. Thesis, University of California, USA.

Okeeffe, M. and M. Ocinneide, 2008. Search-based refactoring for software maintenance. J. Syst. Soft., 81: 502-516.

Parthasarathy, S. and N. Anbazhagan, 2006. Analyzing the software quality metrics for object oriented technology. Inform. Technol. J., 5: 1053-1057.

Pressman, R.S., 2010. Software Engineering: A Practitioner's Approach. McGraw-Hill Education, India.

Riel, A.J., 1996. Object-Oriented Design Heuristics. Addison-Wesley Longman Publishing Co., Inc., Boston, MA., USA., ISBN-13: 9780201633856, Pages: 379.

Sanatnama, H. and F. Brahimi, 2010. Graph drawing algorithms: Using in software tools. J. Applied Sci., 10: 1894-1901.

Shu, Y., H. Liu, Z. Wu and X. Yang, 2009. Modeling of software fault detection and correction processes based on the correction lag. Inform. Technol. J., 8: 735-742.

Tourwe, T. and T. Mens, 2003. Identifying refactoring opportunities using logic meta programming. Proceedings of the 7th European Conference on Software Maintenance and Reengineering, March 26-28, 2003, Benevento, Italy, pp: 91-101.

Vinayagasundaram, B. and S.K. Srivatsa, 2007. Software quality in artificial intelligence system. Inform. Technol. J., 6: 835-842.

Waldinger, R., 1977. Achieving Several Goals Simultaneously. In: Machine Intelligence, Elcock, E.W. and D. Michie (Eds.). Ellis Horwood, Ltd., Chichester, UK., pp: 91-136.