The Journal of Systems and Software 83 (2010) 222–234

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Measuring behavioral dependency for improving change-proneness prediction in UML-based design models

Ah-Rim Han [a,*], Sang-Uk Jeon [a], Doo-Hwan Bae [a], Jang-Eui Hong [b]

[a] Division of CS, College of Information Science and Technology, Korea Advanced Institute of Science and Technology (KAIST), Daejeon 305-701, Republic of Korea
[b] Computer Engineering Division, School of Electrical and Computer Engineering, Chungbuk National University, Chungju 361-763, Republic of Korea

ARTICLE INFO

Article history:
Received 2 March 2009
Received in revised form 27 July 2009
Accepted 22 September 2009
Available online 25 September 2009

Keywords:
Change-proneness
UML
Behavioral dependency measure
Object-oriented metrics

ABSTRACT

Several studies have explored the relationship between the metrics of the object-oriented software and the change-proneness of the classes. This knowledge can be used to help decision-making among design alternatives or assess software quality such as maintainability. Despite the increasing use of complex inheritance relationships and polymorphism in object-oriented software, there has been less emphasis on developing metrics that capture the aspect of dynamic behavior. Considering dynamic behavior metrics in conjunction with existing metrics may go a long way toward obtaining more accurate predictions of change-proneness. To address this need, we provide the behavioral dependency measure using structural and behavioral information taken from UML 2.0 design models. Model-based change-proneness prediction helps to make high-quality software by exploiting design models from the earlier phase of the software development process. The behavioral dependency measure has been evaluated on a multi-version medium size open-source project called JFlex. The results obtained show that the proposed measure is a useful indicator and can be complementary to existing object-oriented metrics for improving the accuracy of change-proneness prediction when the system contains high degree of inheritance relationships and polymorphism.

1. Introduction

Software changes either to enhance functionality or to fix bugs (Parnas, 2001). Therefore, changes are unavoidable and the anticipation of such changes is very important in software development and maintenance. Some classes of software may be more prone to changes than others. The likelihood that a change will occur is referred to as *change-proneness*. Predicting change-proneness can be useful for a number of reasons. It enables developers to focus on preventive actions such as peer-reviews, testing, and inspections allowing them to use their resources more efficiently and deliver higher-quality products in a timely manner (Güneş Koru and Liu, 2007). The prediction of change-proneness can also help to choose among possible design alternatives or aid in the assessment of changeability decay (Arisholm and Sjøberg, 2000). Without such an assessment, a greater effort would be required to implement changes (i.e., a lack of maintainability). Other studies have predicted change-proneness as a means of estimating maintenance effort (Li and Henry, 1993).

In current practice, change-proneness prediction is generally performed based on codes during the later stages of software development or maintenance phase, after architectural decisions have been made that cannot easily be reversed. However, if the change-prone classes can be predicted at the earlier stages of software development life cycle, when the design models become available, modifying the current design or choosing design alternatives would be relatively easy and inexpensive. Furthermore, model-based change-proneness prediction gives a high return on investment because decisions made with regard to design models have substantial downstream consequences; predicting change-proneness improves software quality and saves development costs. Model-based change-proneness prediction is also useful for directly visualizing the locations of changes on UML design models. Indeed, this greatly improves the understandability of the software.

Several studies have attempted to predict change-prone classes using established object-oriented software metrics (Arisholm et al., 2004; Arisholm and Sjøberg, 2000; Chaumun et al., 2002; Bieman et al., 2003); complexity, coupling, and cohesion metrics have received considerable interest in this regard. However, in spite of these efforts for developing change-proneness prediction models, a substantial part of change-prone classes is still not explained. Therefore, other important information is needed to build a more accurate and consistent change-proneness prediction model.

Despite the increasing use of complex inheritance relationships and polymorphism in many applications of object-oriented software, there has been less emphasis on the aspect of dynamic

* Corresponding author. Tel.: +82 42 350 5579; fax: +82 42 350 8488.
E-mail addresses: arhan@se.kaist.ac.kr (A.-R. Han), sujeon@se.kaist.ac.kr (S.-U. Jeon), bae@se.kaist.ac.kr (D.-H. Bae), jehong@chungbuk.ac.kr (J.-E. Hong).

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.09.038

behavior when developing design quality metrics. We may obtain a better change-proneness prediction model by accounting for and measuring the behavioral aspects of the software. Thus, we have developed the Behavioral Dependency Measure (BDM). This measure can be derived not only from the structural information, but also from the behavioral information of UML 2.0 design models. Design models in UML 2.0 (OMG, 2007) possess both pieces of information of the software. A class diagram provides the structural information of classes and the relationships between those classes. A Sequence Diagram (SD) and an Interaction Overview Diagram (IOD) are used for capturing the behavioral aspects of the software. An SD depicts the software in terms of a specific sequence of messages between objects. In SDs, *alt*, *opt*, and *loop* combined fragments enable modeling of complex control structures in a manner similar to the modeling in source codes. An IOD, newly introduced in UML 2.0, represents an overview of the control flow of the complete software.

We empirically assess the statistical and practical significance of the BDM for predicting change-prone classes using JFlex (2009). The results of the experiment show that the BDM helps to improve the accuracy of change-prone class prediction over that of metrics such as Chidamber and Kemerer (C&K) metrics (Chidamber et al., 1994), Lorenz and Kidd metrics (Lorenz and Kidd, 1994), and Metrics for Object-Oriented Development (MOOD) metrics (Abreu, 1995; Abreu et al., 1995) when the system contains high degree of inheritance relationships and polymorphism. On the other hand, when inheritance relationships and polymorphism are used less in the system, the BDM made no difference in the prediction of change-prone classes. This indicates that the BDM is more sensitive than other metrics to properties related to dynamic behaviors. Indeed, a consideration of the above mentioned properties may contribute to a more accurate prediction of the change-proneness of a system that contains high degree of inheritance relationships and polymorphism.

The rest of the paper is organized as follows. Section 2 contains a discussion of related studies. In Section 3, we discuss both the behavioral dependencies that might cause changes and the novel features of the proposed BDM. In Section 4, we systematically describe how to calculate the BDM. Section 5 contains a description of the construction of a change-proneness prediction model. In Section 6, we present a case study to evaluate the proposed BDM and discuss the results obtained. Finally, we conclude this study and discuss futures research in Section 7.

2. Related work

Change-proneness prediction is associated with change impact analysis. The former predicts which classes are likely to change in the future (i.e., change over successive versions), whereas the latter predicts which classes may be impacted by a given change. However, *model-based* change-proneness prediction or change impact analysis is rarely discussed, despite the fact that rich algorithms and tools for source code level studies have been developed. Hassan and Holt (2004) proposed several heuristics to predict change propagation. Bohner and Arnold (1993) gave an overview of several formal models of change propagation, introducing a number of tools and techniques, such as slicing and transitive closure, based on code dependencies and algorithms. Li et al. (1996) proposed a set of algorithms that determine which classes are affected when a given change is proposed. Their methodology represents a system as a set of data dependency graphs. Chen and Rajich (2001) developed a tool to identify the location where changes might be propagated. This tool uses the data/control flow dependencies captured in source codes. The advantage to this tool is that accuracy is increased because it also considers conceptual dependencies. However, these dependencies must be manually specified by the user.

Other studies have investigated the relationship between existing metrics (e.g., coupling) and change-proneness or the impact of changes. Briand et al. (1999) empirically investigated whether coupling metrics are related to ripple effects using a commercial object-oriented system. The aim for using coupling metrics is to rank classes according to the probability of containing ripple effects. However, traditional coupling metrics fail to capture complex dependency caused by inheritance relationships and polymorphism. This results in a less accurate prediction of change-proneness prediction of the software that contains high degree of inheritance relationships and polymorphism. Wilkie and Kitchenham (1999) considered whether classes with high CBO (Coupling Between Object) are more likely to be affected by ripple changes. They found that CBO is generally an indicator of change-proneness. Since CBO cannot account for all possible changes, they also suggested that a more comprehensive coupling metric is required for improving predictions of the potential for change ripple effects in each class. Arisholm et al. (2004) investigated the use of dynamic coupling metrics as indicators of change-proneness. Their approach is based on correlating the number of changes, a continuous variable which represents change-proneness, to each class with dynamic coupling metrics and other class-level size and static coupling metrics. However, dynamic coupling requires extensive test suites to exercise the system. Such test suites may not be readily available. There have been several attempts aiming at assessing external quality factors, such as maintainability, flexibility, and changeability of object-oriented designs. In Chaumun et al. (2002), a change impact model was proposed for changeability assessment with the primary goal of investigating the relationship between existing metrics and the impact of change. In Li and Henry (1993), relationship between existing metrics and maintenance effort (i.e., the number of lines changed per class) has been studied.

Several researchers have proposed probabilistic approaches for evaluating evolution and assessing the probability that each class will change over successive versions (Sharafat and Tahvildari, 2007). Tsantalis et al. (2005) estimated the change-proneness of an object-oriented design by evaluating the probability that each class of the system will be affected when a new functionality is added or when an existing functionality is modified. The output range of the probabilistic measure is from 0 (no changes) to 1 (changes). This is a heavyweight approach with regard to the collection of data in the sense that previous versions of a system have to be analyzed to acquire internal probability values; this could create scalability problems for large systems. In addition, this approach cannot be applied during the early stages of the development process such as at the design level.

3. The behavioral dependency measure for change-proneness prediction

3.1. Behavioral dependency

A change in a class can affect other classes enforcing them to be modified. In order to predict the class affected when a class changes occurs, we need to examine the dependencies of pairs of entities (i.e., classes or objects) in the system. In this paper, we focus on *behavioral* dependency.

Essentially, we assume that the object sending a message has a behavioral dependency on the object receiving the message. This is derived from the insight that modifying the class of the object receiving a message may affect the class of the object sending the message. It is important to note that when an object sends a

message to another object, the class implementing the corresponding method of the message may be different from that of the object receiving the message. This is due to the use of inheritance relationships and polymorphism, which may cause dynamic binding of methods. In this case, the class of the object sending a message must be bound to (i.e., have a behavioral dependency on) the class implementing the actual method of that message. Therefore, we need to consider inheritance relationships and polymorphism according to the behavior of objects in order to correctly identify dependencies between classes and ultimately predict change-proneness accurately. This issue will be explained in detail in Section 3.2. We also assume that a high intensity of behavioral dependency represents high possibility of changes to be occurred. The rationale behind this assumption is that the more external services upon which the class of an object is dependent, the more likely it is that the class will be changed.

To quantify the behavioral dependency, we define two kinds of behavioral dependencies: direct and indirect. Each is defined as follows. Let O denote a set of objects existing in a system.

Definition 1 (*Direct behavioral dependency*). For $op_1, op_2 \in O, op_1$ has a direct behavioral dependency on op_2 if op_1 needs some services of op_2 by sending a synchronous message to op_2 and receiving a reply from op_2. We denote direct behavioral dependency as a relation $\rightarrow$.

Definition 2 (*External service request relation*). For $op_1, op_2, op_3 \in O, op_1 \rightarrow op_2$ and then $op_2 \rightarrow op_3$ because op_1 needs external service which is provided from op_3 via op_2. We denote this as an external service request relation $\curvearrowright$. Therefore, in this case, $(op_1 \rightarrow op_2) \curvearrowright (op_2 \rightarrow op_3)$.

Definition 3 (*Indirect behavioral dependency*). We denote indirect behavioral dependency as a relation $\rightsquigarrow$. For example, for $op_1, op_2, op_3, \ldots, op_n \in O$, if we have the relation $(op_1 \rightarrow op_2) \curvearrowright (op_2 \rightarrow op_3) \curvearrowright \ldots \curvearrowright (op_{n-1} \rightarrow op_n)$, then we can derive the indirect behavioral dependency as $op_1 \rightsquigarrow op_n$ except $n = 2$, which means a direct behavioral dependency $op_1 \rightarrow op_2$.

A synchronous message entails a dependency between two objects since the sender object depends on the receiver object. On the other hand, an asynchronous message does not entail such dependency since the sender object does not wait for a reply but continues to proceed. This means the reply will not affect the sender object's behavior. Therefore, in our approach, we only consider synchronous messages with replies.

Fig. 1 shows two examples of SDs. In SD sd A, object o_1 has a direct behavioral dependency on object o_2 because it sends a synchronous message a to object o_2 and receives a reply from it. On the other hand, object o_1 has an indirect behavioral dependency on object o_3; before object o_1 receives a reply for messagea from object o_2, message b is sent from object o_2 to object o_3. By the same reasoning, object o_1 and object o_4, as well as object o_2 and object o_4, have indirect dependencies. Asynchronous message e from object o_1 to object o_2 does not entail a behavioral dependency since object o_1 (the sender) does not wait for a reply from object o_2. All messages in SD sd B cause direct behavioral dependencies.

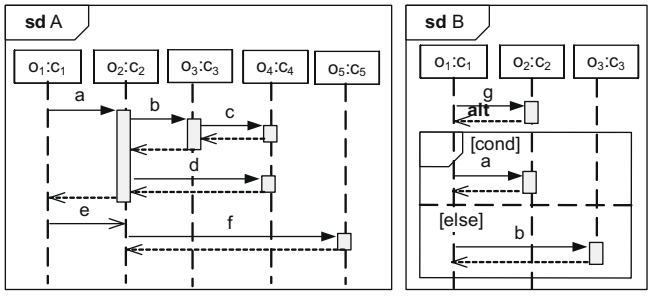

Fig. 1. Examples of Sequence Diagrams (SDs).

It is important to note that an indirect behavioral dependency is not a transitive relation. For example, in Fig. 1, object o_1 and object o_5 do not have a behavioral dependency, even though object o_1 and object o_2 have a behavioral dependency because of message a and object o_2 and object o_5 have a behavioral dependency because of message f. This is because message f is sent from object o_2 to object o_5 after object o_1 receives the reply for message a from object o_2. For this reason, we need to save the information of the message that triggers the current message to precisely identify the indirect behavioral dependency between the two objects. In this way, when object o_i has an indirect dependency on object o_j, we can derive a reachable path (a sequence of exchanged messages between two objects) by traversing stored messages from object o_j to object o_i in a backward direction.

3.2. Features of the behavioral dependency measure

The proposed BDM has a number of features that are different from existing metrics.

First, the most important feature of the BDM that makes it unique is that it considers inheritance relationships and polymorphism. In general, polymorphism indicates method overriding and method overloading. We do not take method overloading into account because it refers to methods that have same name with different numbers or types of parameters in one class; as a result, method overloading does not occur dependency among classes. Therefore, in this paper, polymorphism means method overriding on the classes having inheritance relationships. As the system contains more inheritance relationships and polymorphism, dependency among classes becomes more complex because of dynamic binding of methods. Hence, inheritance relationships and polymorphism as they relate to the behavior of objects need to be considered in order to correctly identify dependency among classes. Indeed, this is critical for the accurate prediction of change-proneness. If we were not to consider inheritance relationships and polymorphism, a class may be mistakenly predicted to be prone to change. The example in Fig. 2 shows the importance of considering inheritance relationships and polymorphism in relation to the behavior of objects when measuring dependency between classes. In the class diagram in Fig. 2a, the Canvas class has an association with the Shape class, which indicates that the Canvas class calls the method draw in the Shape class. In other words, the Canvas class is dependent on the Shape class. This static dependency is the information that we can derive from the class diagram. Most existing coupling metrics are measured based on static dependencies. However, if the message draw is sent to the object that is an instance of the subclass of the Shape class and that subclass overrides the method draw, the dependency is bound between the Canvas class and the subclass, even though the association is specified between the Canvas class and the Shape class. Furthermore, if the message draw is sent to the object that is an instance of the subclass of the Shape class but does not have the method draw, the dependency is bound between the Canvas class and the subclass's one of the parent classes that implement the method draw. The SDs in Fig. 2b illustrate the behavior of the objects that are instances of subclasses (i.e., Triangle class, Circle class, Rectangle class, and Square class) of the Shape class in Fig. 2a. By considering the three foremost SDs, we can determine that the Canvas class is behaviorally dependent on the Triangle class, Circle class, and Rectangle class, all of which override the method draw. By considering the last SD, which tell us that the object of the Square class receives the

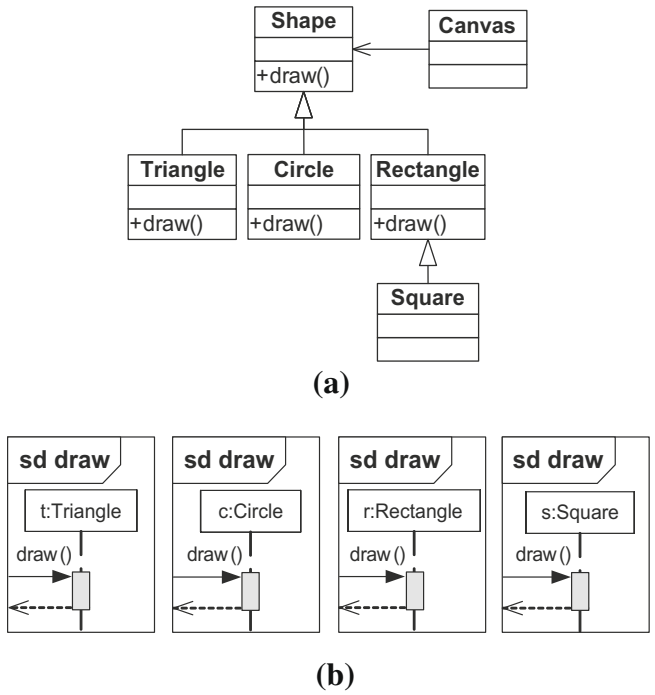

Fig. 2. An example of using inheritance relationships and polymorphism: (a) A class diagram representing classes and their relationships. (b) SDs representing the behaviors of objects that are instantiated from the classes in 2(a).

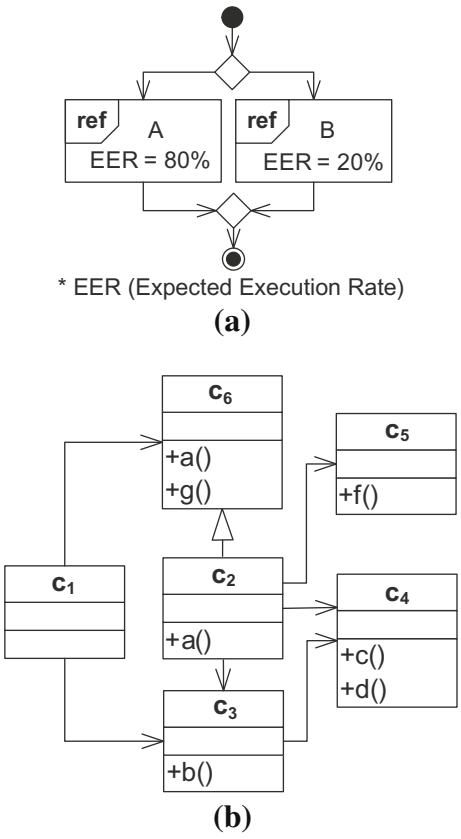

Fig. 3. (a) An example of the Interaction Overview Diagram (IOD). (b) An example of the class diagram that has classes from which the objects, in the SDs in Fig. 1 are instantiated.

message draw, we can also determine that the Canvas class is behaviorally dependent on the Rectangle class, since the method of the message draw is actually implemented in the Rectangle class. As a consequence, no matter where an actual method is implemented, the proposed BDM enables a class of the object sending a message to be bound to the class that implements the actual method of that message; this is the feature that makes the BDM more sensitive to systems with high levels of inheritance relationships and polymorphism.

Second, we consider the extent and direction when measuring the behavioral dependency. No matter how many times a class calls the method of another class, the established coupling metric (e.g., CBO) treats this as one in either direction. This is because the established coupling metric is based on method call dependencies that only capture the static characteristics of couplings. Let us consider two cases; class c_j implements one method called 100 times by class c_i, while class c_k implements two methods that called by class c_i once time for each method. The established static couplings for the former case and the latter case are one and two, respectively. However, class c_i might be more behaviorally dependent on class c_j than it is on class c_k. Therefore, it is important to keep the information relating to the extent and direction of a class's dependence.

Third, we consider the execution rate of the messages based on the control structure and the operational profile. We use two kinds of diagrams, an SD and an IOD, to depict a system's behavior. An SD in UML 2.0 provides combined fragments that allow us to express control structures such as branch and loop. An *alt* combined fragment that corresponds to a branch control structure describes the behavior of two or more mutually-exclusive alternatives. A message in an *alt* combined fragment can be executed depending on the condition. This may affect the behavioral dependency of objects that are related by this message. Without running a program (i.e., dynamic information), it is difficult to determine whether the message will be executed or not. Therefore, the probabilistic execution rate of a message is considered when measuring a behavioral dependency. For example, in the SD sd B of Fig. 1, either message a or message b is executed whether the condition is satisfied or not (i.e., true or false). Therefore, the probabilistic execution rate of each message can be 0.5. An IOD in UML 2.0 illustrates an overview of a flow of control in which each activity node can be an SD. Some scenarios (i.e., SDs) might be executed more frequently than others, as specified in the operational profile (Gittens, 2005). The operational profile provides the expected execution rate of an SD. Therefore, the operational profile also needs to be considered for the better measurement of the behavioral dependency. We suggest specifying the Expected Execution Rate (i.e., the operational profile) of each SD in an IOD. For example, the IOD in Fig. 3a shows that the Expected Execution Rates of SD A and SD B are 80% and 20%, respectively.

4. Behavior dependency measure measurement

In this section, we explain a systematic way of calculating the BDM in UML design models using SDs, a class diagram, and an IOD. An overview of our approach is shown in Fig. 4. The BDM is computed through the following procedures. First, Object Behavioral Dependency Model (OBDM) is constructed for each SD based on all direct and indirect behavioral dependencies between objects by referring to the class diagram and the IOD. After that, we synthesize all OBDMs into the Object System Behavioral Dependency Model (OSBDM) for the entire system. Next, we derive all the reachable paths for each pair of objects in the system from the OSBDM. We then sum the weighted reachable paths for each pair

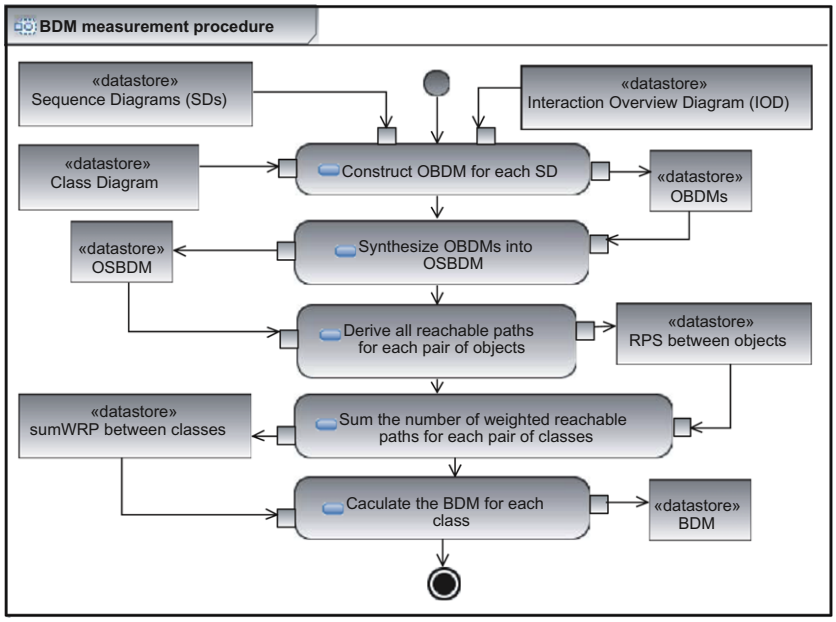

Fig. 4. Overview of our approach of BDM measurement.

of classes (a reachable path is weighted using the distance length between objects and the execution rate of the messages of which the reachable path is composed). Finally, we calculate the BDM for every class in the system. Detailed procedures are described in the following subsections.

4.1. Constructing OBDMs

A dependency model $OBDM_A$ for SD A is a 2-tuple (O, M), where

- O is a set of nodes representing objects in the SD.
- M is a set of edges representing messages that are exchanged between two nodes. Message $m \in M$ represents a synchronous message with a reply, which entails a direct dependency from a sender object to a receiver object. Message m has following six attributes.

– $m_s \in O$ is the sender of the message.
– $m_r \in O$ is the receiver of the message. $m_r \neq m_s$.
– m_n is the name of the message.
– $m_b \in M$ is the instance of a backward navigable message. $m_b \neq m$. "-" means none.
– m_{meL} is the probabilistic message execution rate in an SD. $0 \leqslant m_{meL} \leqslant 1$. The default value is 1.
– m_{meH} is the expected message execution rate in an IOD. $0 \leqslant m_{meH} \leqslant 1$. The default value is 1.

m_s and m_r represent the sender and receiver objects, respectively. Since we do not consider messages from an object to itself, they should not represent the same node. As was pointed out in Section 3.2, when an object sends a message to another object, the class of the object receiving a message may be different from the class implementing the corresponding method. In such a case, the class of the object sending the message may change when the implemented method changes. Therefore, when binding a receiver node, it is important to note whether the method is actually implemented in the class of the receiver object. If not, the receiver node of the message is bound to an object of a parent class that actually implements the method.

m_b represents the message that triggers m and is called a backward navigable message. As was noted in Section 3.1, m_b is essential for identifying indirect behavioral dependencies between objects. We can identify the message that activates the current message by tracing the backward navigable message. When deriving a reachable path from the OSBDM, identification of the message that triggers m prevents infinite loop of traversing.

As was described in Section 3.2, m_{meL} and m_{meH} help to better predict the change-proneness of classes by considering the probabilistic or expected execution rates of the messages. Later, these rates are synthesized according to a reachable path and used to measure behavioral dependency. m_{meL} represents the probabilistic message execution rate in an SD. We consider a branch control structure that might affect the probability of the message execution. Note that a branch control structure is represented as an *alt* combined fragment in UML 2.0. When a message is in an *alt* combined fragment, it is executed only when a condition of the corresponding interaction operand is met. Therefore, m_{meL} is the same as the probability that one of the interaction operands that contain the message is selected. If an *alt* combined fragment is nested, the probability that a message will be executed in the corresponding combined fragment is multiplied to m_{meL} recursively. When a message is not contained in any combined fragments, its m_{meL} is 1. m_{meH} represents the expected message execution rate in an IOD. We specify the Expected Execution Rate (i.e., the operational profile) of each SD in an IOD. A message in an SD is executed only when the corresponding SD is activated. Therefore, m_{meH} is the same as the probability that the control flow of the software reaches the SD to which the message belongs. The m_{meH} values can be obtained by multiplying all the Expected Execution Rates on the way from the initial node to the corresponding SD node in the IOD. If an SD is always activated, the m_{meH} values of all the messages in the SD are 1.

Fig. 5a shows an example of two OBDMs constructed from SD sd A and SD sd B in Fig. 1. Each node of object o_i, $1 \leqslant i \leqslant 6$, corresponds to an instance of class c_i in Fig. 3b. Each edge of message m is represented as $m_n(m_b, m_{meL}, m_{meH})$. Due to inheritance relationships and polymorphism, which may bring about dynamic binding of methods (Section 3.2), we examine the behavior of objects in SDs

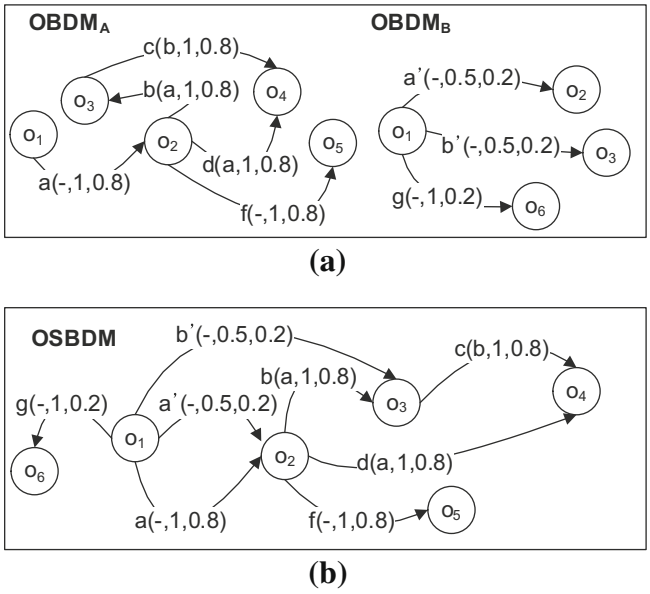

Fig. 5. (a) $OBDM_A$ and $OBDM_B$ correspond to SD sd A and SD sd B in Fig. 1. (b) The obtained OSBDM by synthesizing the two OBDMs in (a).

and the structures of classes in a class diagram in order to correctly identify dependency between classes. In other words, when an object sends a message to another object, we first check whether the method of the message is implemented in the class of the receiving object or in one of its parent classes; we then create an edge corresponding to the message between the node of the object sending the message and the object of which the class actually implemented the method of the message. For instance, when object o_1 sends message a to object o_2, just as in SD sd *A*, we create the edge of message a between node o_1 and node o_2 in $OBDM_A$ since class c_2 overrides method a. On the other hand, when object o_1 sends message g to object o_2, just as in SD sd B, we create the edge of message a between node o_1 and node o_6 in $OBDM_B$ since class c_6 actually implements method g. To distinguish the messages in $OBDM_B$ from those in $OBDM_A$, we rename message a to a′ and b to b′. Since either the message a′ or b′ may be activated depending on the condition of the *alt* combined fragment, both a'_{meL} and b'_{meL} are 0.5. The Expected Execution Rates of SD A and SD B are represented in the IOD in Fig. 3a and are reflected in the execution rates of the messages as 0.8 and 0.2, respectively.

4.2. Synthesizing OBDMs into an OSBDM

To determine the behavioral dependencies between objects in the whole system, we synthesize OBDMs into $OSBDM = (O_s, M_s)$. O_s and M_s denote the set of objects and the union of messages that exist in the system, respectively. The method for constructing the OSBDM will be explained using the example in Fig. 5. Fig. 5b shows the obtained OSBDM by synthesizing the two OBDMs in Fig. 5a. This OSBDM is composed of $O_s = \{o_1, o_2, \ldots, o_6\}$ and $M_s = \{(m \in M \text{ of SD sd A}) \cup (m \in M \text{ of SD sd B})\}$. Note that object o_1 in SD sd A and object o_1 in SD sd B are instantiated from the same class c_1. Therefore only one o_1 remains in the OSBDM. The sending message a from o_1 in SD sd A and another sending message a′ from o_1 in SD sd B are connected with the corresponding target object o_2 in the OSBDM. If a message *m* is triggered by another message in the context of the system by examining the IOD, we set this other message as a backward navigable message of message *m*. There is no such case in this example.

Algorithm 1. RetrieveReachablePathSet($o_1, o_2 : O$)

```
input OUT ← outgoing message set of o1
input IN ← incoming message set of o2
input RP ← ∅ an array for storing reachable path
        /*RP denotes a reachable path*/
input RPS ← ∅ a vector for saving a set of reachable paths
output RPS
for all in ∈ IN do
  for all out ∈ OUT do
    if in == out then
      /*For RPS by the Direct Behavioral Dependency*/
      RPS ← RPS ∪ {in}
    else
      /*For RPS by the Indirect Behavioral Dependency*/
      RP ← RP + {in}
      While in_b! = out&&in_b! = ∅ do
        if in == out then
          RP ← RP + {in_b}
          RPS ← RPS ∪ RP
          RP ← ∅
          break
        else
          RP ← RP + {in_b}
          in ← in_b
```

4.3. Deriving reachable paths

We derive all reachable paths for each pair of objects in the system from the OSBDM. Let $RPS(o_i, o_j) = \{s |$ s is a reachable path between source object o_i and target object $o_j\}$ be a set of all the reachable paths between two objects. To retrieve the $RPS(o_i, o_j)$, we start traversing of the OSBDM from a message incoming to object o_j to a message outgoing from object o_i in reverse. When object o_i has a direct behavioral dependency on object o_j, one of the incoming messages to object o_j and one of the outgoing messages from object o_i are equal. This message is then added into a set of reachable paths. On the other hand, when object o_i has an indirect behavioral dependency on object o_j, we traverse the OSBDM from one of the messages incoming to object o_j iteratively by substituting it with a backward navigable message. In doing this, we finally reach one of the outgoing messages from object o_i. The stored sequence of messages encountered while traversing is the reachable path. The method for retrieving a reachable path set from object o_1 to object o_2 is presented in Algorithm 1. The set of reachable paths for each pair of objects in Fig. 5b is presented in Table 1.

4.4. Summing weighted reachable paths

Prior to calculating the BDM for every class in the system, we sum the weighted reachable paths for each pair of objects using the *RPS* obtained above. In this process, an object is projected onto

Table 1
$RPS(o_i, o_j)$, which is a set of all the reachable paths for each pair of objects (row: o_i, column: o_j) in Fig. 5b.

	o_1	o_2	o_3	o_4	o_5	o_6
o_1	–	{a, a′}	{ab, b′}	{abc,ad}	–	{g}
o_2	–	–	{b}	{bc,d}	{f}	–
o_3	–	–	–	{c}	–	–
o_4	–	–	–	–	–	–
o_5	–	–	–	–	–	–
o_6	–	–	–	–	–	–

Table 2
SumWRP(c_i, c_j), which is the sum of the weighted reachable paths for each pair of classes (row: c_i, column: c_j) and the *BDM*(c_i) value of each class in Fig. 3(b).

	c_1	c_2	c_3	c_4	c_5	c_6	*BDM*(c_i)
c_1	0	0.9	0.5	0.67	0	0.2	2.27
c_2	0	0	0.8	1.2	0.8	0	2.8
c_3	0	0	0	0.8	0	0	0.8
c_4	0	0	0	0	0	0	0
c_5	0	0	0	0	0	0	0
c_6	0	0	0	0	0	0	0

the class from which the object is instantiated. In this manner, the results of the summation of the weighted reachable paths are obtained for each pair of classes.

We formalize the sum of the Weighted Reachable Paths (SumWRP) from class c_i to class c_j as follows:

$$SumWRP(c_i, c_j) = \sum_{\forall s \in RPS(o_i, o_j)} DF(s) \times f_{meL} \times f_{meH}, \quad (1)$$

where o_i and o_j indicate the objects that correspond to projected instances of class c_i and c_j, respectively. We use three factors for weighting reachable path s: distance factor, f_{meL}, and f_{meH}. We define a distance factor by $DF(s) = 1/d$, where d is the distance length (i.e., the number of messages in the corresponding reachable path s). The rationale for using the distance factor is that an indirect behavioral dependency might be weakened by the successive calls. In other words, the farther an object is from the source of changes, the less the object is likely to be changed. Therefore, we need to degrade the impact when the distance of indirect behavioral dependency between two objects is great. We represent the first message in the reachable path s as f. Then, f_{meL}, the probabilistic message execution rate, and f_{meH}, the expected message execution rate, are taken into account as factors for weighting a reachable path.

We explain how to calculate SumWRP using Table 1. To calculate *SumWRP*(c_2, c_4), for example, we first obtain reachable paths, {bc,d}, between object o_2 and object o_4. We then identify weighting factors for each reachable path. For reachable path bc, the distance factor is 1/2, because the number of messages in this reachable path is 2. The first message of this reachable path is b. Therefore, b_{meL}, 1, and b_{meH}, 0.8, are applied for weighting the reachable path. For reachable path d, the distance factor is 1; d_{meL} and d_{meH} are 1 and 0.8, respectively. Finally, we sum the weighted reachable paths and obtain *SumWRP*(c_2, c_4) as follows:

$$SumWRP(c_2, c_4) = (1/2 \times 1 \times 0.8) + (1 \times 1 \times 0.8) = 1.2.$$

4.5. Calculating the behavioral dependency measure

Finally, the BDM for every class c_i in the system is obtained as follows. Let $C = \{c_i | 1 \leqslant i \leqslant n\}$ be all the classes existing in the system.

$$BDM(c_i) = \sum_{\forall c_j \in C, i \neq j} SumWRP(c_i, c_j). \quad (2)$$

Table 2 summarizes the sum of the weighted reachable paths obtained from the OSBDM in Fig. 5b and the BDM of each class in Fig. 3b. The BDM is used to predict change-proneness; the higher the class's BDM, the larger the likelihood the class will be changed.

5. Change-proneness modeling

In this section, we describe the method for building a change-proneness prediction model. In our study, the change-proneness is used for predicting change-prone classes in the successive versions.

5.1. Model construction method

To build the change-proneness prediction model, there are a large number of modeling techniques from which to choose, including standard statistical techniques (e.g., logistic regression) and data mining techniques (e.g., decision trees Han and Kamber, 2006). Multiple linear regression provides a regression analysis of variance for a dependent variable explained by one or more factor variables. Hence, we choose a stepwise multiple regression (Edwards, 1976) to build the change-proneness model in this study. While constructing the regression, we remove outliers that are clearly over-influential on the regression results. Two kinds of techniques can be used for outlier analysis: Standard errors of the predicted values (S.E. of mean predictions) and the Mahalanobis distance (Mahalanobis, 1936). The former is an estimate of the standard deviation of the average value for dependent variable for cases that have the same values with the independent variables. The latter is a measure of how much a case's values on the independent variables differ from the average of all cases; case means a data instance for constructing a prediction model. Hence, we identify and remove the instances that have extremely large S.E. of mean predictions and large Mahalanobis distance values.

5.2. Model variables

We first collected several data types from the object-oriented software.

The independent variables include the C&K metrics, Lorenz and Kidd metrics, MOOD metrics, and the BDM. We collect the C&K metrics and Lorenz and Kidd metrics using (Together, 2006). These are the most widely used metrics for evaluating object-oriented software. The set of metrics used in the case study are listed in the Appendix. To calculate the BDM, which is measured on UML models, we have developed a tool built on the EMF (Eclipse Modeling Framework). It imports the UML 2.0 models in the format of XMI generated from (Rational Software Architect, 2008), an Eclipse-based UML 2.0 modeling tool made by the Rational Division of IBM.

Following a common analysis procedure (Arisholm and Briand, 2006), we first perform a Principal Component Analysis (PCA) to identify the dimensions actually present in the data relating to the independent variables. We do not make use of a PCA to select a subset of independent variables since, as discussed in Briand and Wust (2002), experience has shown that this usually leads to suboptimal prediction models even though regression coefficients are easier to interpret. The resulting principal components can be described in terms of categories such as size, complexity, cohesion, coupling, inheritance and polymorphism (see the Appendix).

The dependent variable of the model is the change-proneness. To compute the change-proneness, the change data, which are obtained using a class-level source code *diff*, are collected for each application class. Based on this change data, the total amount of changes (i.e., source lines of code added and deleted) within consequent releases are measured.

6. Case study

This section presents the results of a case study, the objective of which is to validate the usefulness of the BDM presented above. The first subsection explains the details of the system. In the next subsection, the goal of the case study and the validation method are described. In the third section, results are presented and interpreted. The last subsection ends with a discussion.

6.1. The subject of the case study

In order to investigate whether the BDM is statistically related to change-proneness, we need a target system that has well documented UML models with a class diagram, SDs an IOD, and subsequent releases for extracting change-related information. For our experiment, we reverse-engineered the UML design models from the existing system, JFlex, using a reverse-engineering tool with manual supports. JFlex is a lexical analyzer generator for Java, which is written in Java. JFlex takes a specially formatted specification file containing the details of a lexical analyzer as input and creates a Java source file for the corresponding lexical analyzer. A number of reasons led us to select JFlex for the case study:

- It has evolved through 14 generations (at the time that we conducted this case study) and recorded the history of changes.
- The full source code of each version is available because it is an open-source project.
- It contains a relatively large number of classes.
- It is mature. The release dates are February 20, 2001 for the initial version (version 1.3) and January 31, 2009 for the latest version (version 1.4.3).
- It was written in Java. Our BDM is applied in object-oriented software that uses inheritance relationships, so polymorphism and dynamic binding may occur.

Among the 14 releases of JFlex, version 1.3.5 is not considered in the case study because the changes made between this version and version 1.3.4 are negligible. Table 3 represents the number of ground facts regarding the 13 successive versions of JFlex. The initial version of JFlex 1.3 consists of 44 Java classes and 1394 reachable paths, while the latest version of JFlex 1.4.3 consists of 62 Java classes and 3319 reachable paths. The total number of reachable paths can be less than the total number of invoked messages in each version of the system in the following cases: (1) invoked messages for which call methods from the library are not considered (the scope of the measurement is limited to the application classes of JFlex) and (2) invoked messages for which call methods within the same class are not considered, since these internal messages do not cause behavioral dependency.

We collected several types of data (i.e., existing object-oriented software metrics, the BDM, and change data) for each class from nine versions of JFlex based on reverse-engineered models. It should be noted that we collected metrics that are available on design models. In other words, we did not gather metrics that are obtainable only from source codes, such as source lines of code (SLOC), number of fields (NOF), and number of parameters (NOP). To select classes with a long history of changes, we included the classes from the initial version that remain in the latest version (i.e., 42 classes for each version of JFlex). For each version on which the BDM and other metrics are measured, the change data was measured by counting the total number of changes in the next four subsequent versions. This change data is used as change-proneness. We take 9 of the 13 releases into account because the change data is not available in the last four versions; we finally obtained 378 instances of classes.

We easily reverse-engineered the class diagram from JFlex source code. On the other hand, reverse-engineering SDs is difficult and sometimes even impossible (Briand et al., 2003), because an SD represents the partial behavior of the overall system; SDs can exist in various forms according to the various users' view on the system. Thus, in this case study, we construct the SD for each reachable path that consists of consecutive invoked messages, while extracting the structural information from source codes and reflecting it in the SD as *alt*, *opt*, or *loop* combined fragments. It should be noted that the SD in UML 2.0 uses the combined fragments to represent one or more sequences (traces) rather than specifying all the possible scenarios (Rountev et al., 2004). Hence, we do not need to execute the system and monitor its execution to retrieve meaningful information and reverse-engineer SDs from source codes. Fig. 6 shows an example of the reverse-engineered SD obtained from JFlex version 1.3. This SD corresponds to the reachable path from the object of the LexParse class to the object of the Out class with four messages: CUP$LexParse$do_action, make-

Table 3
The number of ground facts regarding 13 subsequent versions of JFlex (versions 1.3–1.4.3).

	1.3	1.3.1	1.3.2	1.3.3	1.3.4	1.4pre1	1.4pre3	1.4pre4	1.4pre5	1.4	1.4.1	1. 4.2	1.4.3
Package	3	3	3	4	4	4	4	5	5	5	5	5	5
Class	44	44	44	48	48	47	47	61	59	59	59	62	62
Interface	4	4	4	4	4	4	4	3	3	3	3	4	4
Invoked Message	1768	1828	1828	2042	2048	2071	2135	2426	2418	2401	2376	2651	2651
Reachable Path	1394	1442	1442	2335	2339	2362	2282	3244	3244	3249	3242	3317	3319

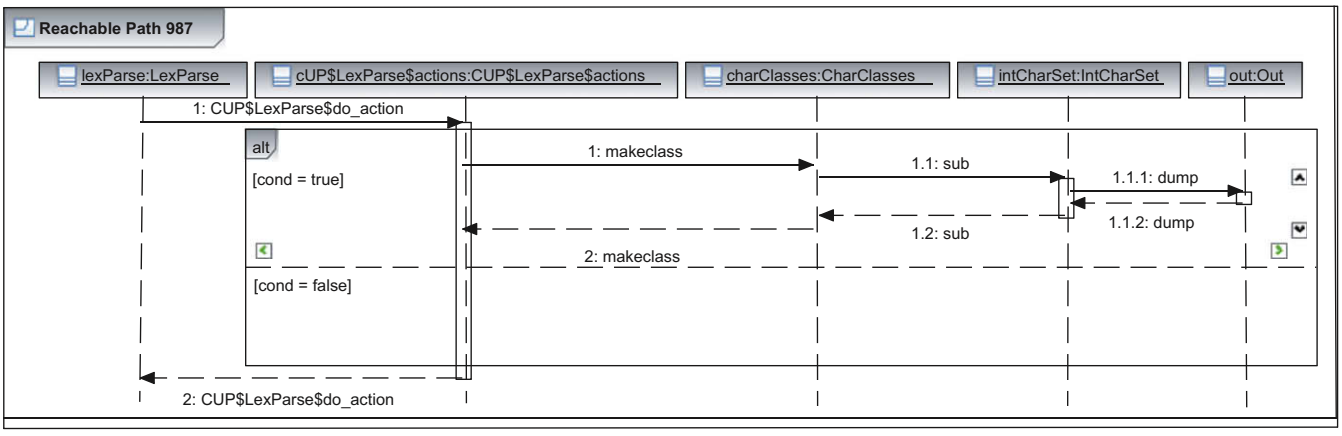

Fig. 6. An SD that was reverse-engineered from source codes of JFlex version 1.3.

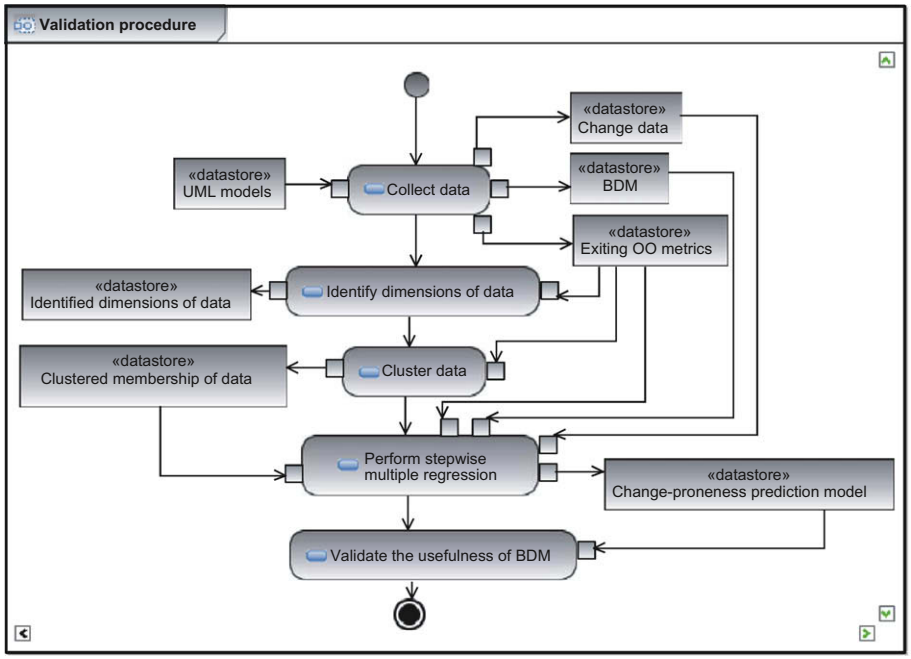

Fig. 7. A validation procedure followed during in this case study.

Class, sub, and dump. By analyzing source code from JFlex 1.3, for example, we extracted 1394 reachable paths and, at the same time, constructed 1394 SDs with 1768 messages. The IOD cannot be reversed from source codes; it is specified only from the early stages of software development to help developers get an overview of the system. Thus, in this experiment, the Expected Execution Rate in the IOD was not considered when calculating the BDM.

6.2. Goal and validation methodology

The goal of this case study is to confirm that the BDM is a significant additional explanatory variable over and above that which has already been accounted for by other existing metrics when the system contains complex inheritance relationships and polymorphism. It should be noted that the BDM considers dynamic features (see Section 3.2). As a result, we expect the BDM to more accurately predict behavioral dependency when the system is involved in a complex dynamic binding occurrence environment. Indeed, accurate prediction of behavioral dependency helps to construct a better change-proneness prediction model. In order to achieve this goal, the validation procedure depicted in Fig. 7 was performed.

To investigate whether the effect of the BDM is different according to intensity of use of inheritance relationships and polymorphism, we divide the data set into two groups and independently build change-proneness prediction models for each group. We cluster the data into the following groups:

- (Group 1) contains comparatively more complex inheritance relationships and polymorphism.
- (Group 2) contains comparatively less complex inheritance relationships and polymorphism.

To classify the data set into these two groups, a clustering technique was applied. Clustering is also called data segmentation and is used to partition large data sets into groups according to similarities. Clustering may serve as a preprocessing step for classification, which would then operate on the detected clusters and the selected attributes or features (Han and Kamber, 2006). We used K-means clustering (Cios et al., 1998) with four inheritance- and polymorphism-related metrics: PF (Polymorphism Factor), MIF (Method Inheritance Factor), NOC (Number of Children), and DIT (Depth of Inheritance). Table 4 represents the analysis-of-variance (ANOVA) table that includes the results for each clustering variable. At the $\alpha = 0.05$ significance level, MIF and PF are significant explanatory variables to divide the groups since Sig. (p-value) $\approx 0.000 \leqslant 0.05 = \alpha$. The descriptive statistics with respect to the four attributes are provided for Groups 1 and 2 in Tables 5 and 6, respectively. The classification results show that the classes in Group 1 have higher PF and NOC values than those in Group 2. Therefore, Group 1 can be characterized as having more complex inheritance relationships and polymorphism than Group 2. As there is not much difference in MIF values and no difference in DIT values between the two groups, we did not consider these two attributes for identifying each group's characteristics. In analyzing the cluster membership of the classes in JFlex across the nine versions, we noticed that the classes that developed earlier tend to belong to Group 1, while the classes that developed later tend to belong to Group 2. In other words, the JFlex system has evolved in that it now has less complex inheritance relationships and polymorphism.

For Groups 1 and 2, we construct two change-proneness prediction models: one between the change and existing metrics and the other between the change and the BDM in addition to existing metrics. Consequently, we analyzed four change-proneness prediction models in total. To validate the assertion that the BDM helps

Table 4
The analysis-of-variance (ANOVA) table that includes the results for each clustering variable.

	Cluster		Error		F	Sig.
	Mean square	df	Mean square	df		
MIF	320.333	1	.439	376	729.972	.000
PF	7500.000	1	.383	376	19583.333	.000
DI T	.000	1	2.168	376	.000	1.000
NOC	.593	1	.673	376	.881	.349

Table 5
Descriptive statistics with respect to the four attributes for Group 1 (294 instances).

	N	Range	Minimum	Maximum	Sum	Mean		Std.	Variance	Skewness	
	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Std. error	Statistic	Statistic	Statistic	Std. error
PF	294	2	36	38	11088	37.71	.041	.701	.491	−2.052	.142
MIF	294	2	70	72	20958	71.29	.041	.701	.491	−.462	.142
NOC	294	5	0	5	84	.29	.051	.882	.778	4.033	.142
DIT	294	5	0	5	210	.71	.086	1.471	2.164	1.864	.142

Table 6
Descriptive statistics with respect to the four attributes for Group 2 (84 instances).

	N	Range	Minimum	Maximum	Sum	Mean		Std.	Variance	Skewness	
	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Std. error	Statistic	Statistic	Statistic	Std. error
PF	84	0	26	27	2268	27.00	.000	.000	.000	.000	.263
MIF	84	1	73	74	6174	73.50	.055	.503	.253	.000	.263
NOC	84	2	0	2	16	.19	.060	.548	.301	2.782	.263
DIT	84	5	0	5	60	.71	.161	1.477	2.182	1.888	.263

to explain additional variations in change-proneness, we compare the goodness-of-the-fit of those two models. As a result, it is clear that the BDM contributes to obtain a better model fit. The results of the change-proneness prediction models are presented and discussed in detail in the next subsection.

6.3. Results

6.3.1. Results of the change-proneness prediction models

We evaluated the performance of the prediction models according to the goodness-of-the-fit (R-square) and a sequence of the selection of independent variables. The sequence of the selection is important because the independent variable that has the largest positive or negative correlation with the dependent variable is selected at each step in a stepwise selection.

We first present the results of the two regression models obtained from the data set of Group 1. The results of the stepwise regression using C&K metrics, Lorenz and Kidd metrics, and MOOD metrics as candidate covariates are presented in Table 7. The prediction model included five variables. The model explains around 57% of the variance of the data set and shows an adjusted R^2 of 0.54. The sequence of variables entering into the model is COC, PProtM, NOC, WMC, and MNOL. The result after including the BDM in addition to existing metrics to make a prediction model, we obtain the result as shown in Table 8. Around 64% of the variance in the data set is explained and an adjusted R^2 of 0.63 is obtained. In this model, COC, BDM, PProtM, NOC, MNOL, WMC, and NORM variables were included in the order of the sequence as listed. Note that the BDM is the second variable to be included with significant-level of p-value < 0.00001. Even when accounting for the difference in the number of covariates, the coefficient of determination (R^2) is increased by 9% (from 0.54 to 0.63) when using the BDM. Therefore, this experiment shows that the BDM helps to obtain a better change-proneness prediction model. In other words, even though the existing metrics still do most of the lifting, the BDM captures additional dimensions that enable the construction of a more accurate change-proneness prediction model.

From the data set of Group 2, we also constructed two regression models, one using only existing metrics for the baseline of the comparison and the other using the BDM in addition to existing metrics, to investigate whether the effect of the BDM performs differently according to the intensity of inheritance relationships and polymorphism. In this group, the BDM was not included when constructing the regression model. Hence, the results of the two models are the same. Table 9 shows the results of the prediction model obtained from Group 2. This model explains the change variance of around 75% and shows an adjusted R^2 of 0.74. The goodness-of-the-fit of the prediction model in Group 2 is considerably higher than that of the prediction models in Group 1 because smaller data instances were used to create the model.

6.3.2. Interpretation of results

In the experiment, we divided the data set into two groups, one with comparatively more complex and the other with comparatively less complex inheritance relationships and polymorphism, in order to investigate the effects of the BDM according to the intensity of inheritance relationships and polymorphism. Intensity of use of inheritance relationships and polymorphism can be explained through the attributes (i.e., inheritance- and polymorphism-related metrics) used for K-means clustering groups; PF and NOC were used for characterizing each group, as mentioned in Section 6.2. PF equals the number of actual method overrides

Table 7
Prediction model using existing metrics in Group 1.

Selected variables	Unstandardized coefficients		Standardized coefficients	t	Sig.
	B	Std. error	Beta		
(Constant)	.050	.054		.915	.361
COC	.050	.011	.215	4.695	.000
PProtM	−.010	.004	−.122	−2.809	.005
NOC	.199	.042	.204	4.715	.000
WMC	−.019	.004	−.399	−4.444	.000
MNOL	.114	.030	.249	3.748	.000

Table 8
Prediction model using existing metrics and *BDM* in Group 1.

Selected variables	Unstandardized coefficients		Standardized coefficients	t	Sig.
	B	Std. error	Beta		
(Constant)	.048	.053		.894	.372
COC	.050	.011	.214	4.632	.000
BDM	.019	.008	.108	2.431	.000
PProtM	−.012	.004	−.144	−3.263	.001
NOC	.202	.042	.206	4.811	.000
MNOL	.120	.029	.264	4.114	.000
WMC	−.017	.004	−.361	−3.931	.000
NORM	.008	.003	.184	2.465	.014

Table 9
Group 2 prediction model (the result is same whether the BDM is used or not because the BDM is not included in the model).

Selected variables	Unstandardized coefficients		Standardized coefficients	t	Sig.
	B	Std. error	Beta		
(Constant)	−1.211	.397		−3.052	.003
NOO	−.013	.003	−.307	−4.280	.000
CL	.021	.004	.571	5.393	.000
NOIS	.816	.107	1.252	7.615	.000
RFC	−.139	.024	−.798	−5.832	.000

divided by the maximum number of possible method overrides and is calculated as a fraction. The PF value increases as the system uses method overriding; if the system overrides everything, the PF is 100%. If subclasses seldom override their parent's methods, PF will be low. NOC equals the number of immediate subclasses derived from a base class and measures the breadth of a class hierarchy. Conversely, DIT measures the depth. Therefore, it can be concluded that Group 1 contains relatively complex inheritance relationships and polymorphism than Group 2 since the former has higher PF and NOC values than the latter.

It was determined that the BDM is a significant indicator as it helped to improve the accuracy of change-proneness prediction in Group 1 only. This result was anticipated because in Group 1, the system redefines the parent's methods more often (i.e., high PF) and inherits parent classes more often (i.e., high NOC) than the system, in Group 2. In short, Group 1 contains high degree of inheritance relationships and polymorphism, which may be the reason for the high probability that dynamic binding will occur. In Group 2, the BDM could not be selected as a variable to explain the variances in change-proneness. This indicates that the BDM is no more useful than existing metrics in systems that contain low degree of inheritance relationships or polymorphism. By analyzing the results of the experiment, we reached the conclusion that the BDM can help accurately predict changes when the system contains high degree of inheritance relationships and polymorphism. The reason for improving the accuracy of change-proneness prediction is the BDM's feature enabling a class of the object sending a message to be bound to the class that actually implements the method of the message, as mentioned in Section 3.2. To put the point another way, the BDM is a specific measure for a consideration of the dynamic behavior of the system.

6.4. Discussion

In the case study, we showed that the BDM is the significant indicator for predicting change-proneness when the system contains high degree of inheritance relationships and polymorphism. However, it is not possible to determine the exact thresholds of the system's high degree of inheritance relationships and polymorphism because these thresholds are relative and empirical. However, we do not need specific guidelines that tell us when to use the BDM in change-proneness prediction. The BDM is an additional variable that may be used in conjunction with existing metrics for explaining variance in change-proneness for systems where dynamic binding is likely to occur. When constructing a change-proneness prediction model, the BDM is not selected if it cannot capture any features over and above those captured by existing metrics. In other words, the BDM is selected as a significant variable only if it helps to improve the accuracy of change-proneness prediction in addition to existing metrics.

In the case study, we used reversed UML models that were obtained from source codes, even though model-based change-proneness prediction was the goal of the study. This is because, in practice, most legacy systems which have been developed and maintained for a long period of time do not have well documented design models, especially for SDs and IODs. It is worth reiterating that an IOD is specified from the early stages of software development and cannot be reversed from source codes. If an IOD specified with the Expected Execution Rate in each SD is available, a more accurate BDM may be obtained. In the future, we plan to use the UML models which will soon be available from the Repository for Model-Driven Development (REMODD) project (France et al., 2007) in order to explore the usefulness of the BDM for model-based change-proneness prediction.

The fitness of the models for model-based change-proneness prediction is rather low compared to models for code-based change-proneness prediction. This is because the information extracted from UML models is not as sufficient for change-proneness prediction as the information from source codes. If other metrics derivable from only source codes were considered when building the change-proneness prediction model, the R^2 values would be higher. For example, SLOC, which indicates the size of the class, is known as a significant indicator to affect change-proneness (Arisholm et al., 2004). Of course, the goal of this study is to determine whether the BDM helps to obtain a better model fit. Therefore, we need to see that it provides an improved predictive model when compared to models considering only existing metrics that are available on UML models. This provides the benefit of early change-proneness prediction at the moment a design model becomes available, without the necessity of implementing source codes.

The results from our earlier study (Han et al., 2008) on JFreeChart JFreeChart (2005) also showed that the BDM is a strong indicator and complementary to C&K metrics for explaining the variance of changes. In this paper, we performed the new experiment to compare the effect of the BDM with varying degrees of inheritance relationships and polymorphism. We used another system, because complex dynamic bindings may not occur in the system examined in the previous study, JFreeChart, since it is the graphic library for generating various types of charts. In other words, in JFreeChart, dependencies occurred by method calling would be simple.

In our previous paper, we only used C&K metrics as the existing metrics. However, in this case study, we used more metrics to confirm that the BDM is effective with regard to change-proneness prediction.

7. Conclusion and future work

In this paper, we proposed the BDM which was obtained from UML design models of object-oriented software for improving accuracy over existing metrics when predicting change-prone classes. We first provided the definitions of behavioral dependencies and suggested a systematic approach for calculating the BDM based on the defined behavioral dependencies. We then performed a case study to evaluate that the BDM is a useful and complementary indicator for change-proneness prediction when the system contains complex inheritance relationships and polymorphism. The results of the case study show that the BDM is the specific measure for considering the dynamic behavior of the system.

Model-based change-proneness prediction using the BDM has several advantages, even though the goodness-of-the-fit of the model-based change-proneness prediction may be lower than that of code-based change-proneness prediction. In the early stages of development, model-based change-proneness prediction can provide a way to modify the current design or allow users to choose design alternatives in a relatively easy and inexpensive manner; using the BDM is certainly more cost-effective than reworking

the implemented system. Model-based change-proneness prediction can also help users better understand the software by visualizing the locations of changes on UML design models.

Our future work will include the followings: (1) extending the BDM to take into account other dependency attributes such as time; (2) investigating other applications of the BDM, such as fault-proneness prediction or object allocation in a distributed system; (3) visualizing change-prone classes on a modeling tool such as Rational Software Architect; and (4) confirming the BDM's usefulness by applying it to a system that has maintained UML documents and change logs of released versions.

Acknowledgements

This research was supported by the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2009-(C1090-0902-0032)) and this work was partially supported by Defense Acquisition Program Administration and Agency for Defense Development under the contract.

Appendix A. Object-oriented metrics definition

Metrics	Category	Description	Definition
CBO	Coupling	Coupling between objects	Represents the number of other classes to which a class is coupled to
CL	Cohesion	Class locality	Is computed as the relative number of dependencies that a class has in its own package
COC	Coupling	Clients of class	Is the number of classes that use the interface of the measured class
DIT	Inheritance	Depth of inheritance	Is the length of the inheritance chain from the root of the inheritance tree to the measured class
LCOM	Cohesion	Lack of cohesion in methods	Is the number of pairs of methods in the class using no attribute in common minus the number of pairs methods that do
MIF	Inheritance	Method inheritance factor	Is the sum of inherited methods over total methods available in classes
MNOL	Maximum	Maximum number of levels	Counts the maximum depth of *if*, *for* and *while* branches in the bodies of methods
MSOO	Maximum	Maximum size of operation	Counts the maximum size of operations for a class
NOAM	Polymorphism	Number of added methods	Counts the number of operations added by a class
NOC	Inheritance	Number of children	Counts the number of classes directly or indirectly derived from the measured class
NOIS	Size	Number of import statements	Counts the number of imported packages/classes
NOO	Size	Number of operations	Counts the number of operations
NOOM	Polymorphism	Number of overridden methods	Counts the number of inherited operations, which a class overrides
NORM	Complexity	Number of remote methods	Processes all methods and constructors and counts the number of various remote methods called
PF	Polymorphism	Polymorphism factor	Is the number of actual method overrides divided by the maximum number of possible method overrides
PPrivM	Ratio	Percentage of private members	Counts the percentage of private members in a class
PProtM	Ratio	Percentage of protected members	Counts the percentage of protected members in a class
PPubM	Ratio	Percentage of public members	Counts the percentage of public members in a class
RFC	Coupling	Response for a class	Counts the number of methods in the response set for a class, which includes the number of methods in the class and the number of remote methods invoked by the methods in the class
WMC	Complexity	Weighted methods per class	Is the sum of the complexity of all methods for a class

References

Abreu, F., 1995. The MOOD metric set. In: Proceedings of the ECOOP'95 Workshop on Metrics.

Abreu, F., Goulao, M., Esteves, R., 1995. Toward the design quality evaluation of object-oriented software systems. In: Proceedings of the Fifth International Conference Software Quality, pp. 44–57.

Arisholm, E., Briand, L.C., 2006. Predicting fault-prone components in a java legacy system. In: ISESE '06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering. ACM, New York, NY, USA, pp. 8–17.

Arisholm, E., Sjøberg, D., 2000. Towards a framework for empirical assessment of changeability decay. The Journal of Systems and Software 53 (1), 3–14.

Arisholm, E., Briand, L., Foyen, A., 2004. Dynamic coupling measurement for object-oriented software. IEEE Transaction on Software Engineering 30, 491–506.

Bieman, J., Andrews, A., Yang, H., 2003. Understanding change-proneness in OO software through visualization. In: Proceedings of the 11th IEEE International Workshop on Program Comprehension, pp. 44–53.

Bohner, S., Arnold, R., 1993. Impact analysis – towards a framework for comparison. In: Proceedings of the International Conference on Software Maintenance, pp. 292–301.

Briand, L.C., Wust, J., 2002. Empirical studies of quality models in object-oriented systems. In: Advances in Computers. Academic Press, pp. 97–166.

Briand, L., Wurst, J., Lounis, H., 1999. Using coupling measurement for impact analysis in object-oriented systems. In: Proceedings of the International Conference on Software Maintenance, pp. 475–482.

Briand, L., Labiche, Y., Miao, Y., 2003. Towards the reverse engineering of UML sequence diagrams. In: Proceedings of the 10th Working Conference on Reverse Engineering. pp. 57–66.

Chaumun, M., Kabaili, H., Keller, R., Lustman, F., 2002. A Change Impact Model for Changeability Assessment in Object-Oriented Software Systems, vol. 45. Elsevier. pp. 155–174.

Chen, K., Rajich, V., 2001. RIPPLES: tool for change in legacy software. In: Proceedings of the IEEE International Conference on Software Maintenance. pp. 230–239.

Chidamber, S., Kemerer, C., MIT, C., 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20 (6), 476–493.
Cios, K., Pedrycz, W., Swiniarsk, R., 1998. Data mining methods for knowledge discovery. IEEE Transactions on Neural Networks 9 (6), 1533–1534.
Edwards, A., 1976. An Introduction to Linear Regression and Correlation. W.H. Freeman, San Francisco, CA.
France, R., Bieman, J., Cheng, B., 2007. Repository for model driven development (ReMoDD). Lecture Notes in Computer Science 4364, 311.
Gittens, M., 2005. The Extended Operational Profile Model for Usage-based Software Testing. Library and Archives Canada Bibliothèque et Archives Canada.
Güneş Koru, A., Liu, H., 2007. Identifying and characterizing change-prone classes in two large-scale open-source products. The Journal of Systems and Software 80 (1), 63–73.
Han, J., Kamber, M., 2006. Data Mining: Concepts and Techniques. Morgan Kaufmann Series in Data Management Systems, second ed., San Francisco, CA.
Han, A., Jeon, S., Bae, D., Hong, J., 2008. Behavioral dependency measurement for change-proneness prediction in UML 2.0 design models. In: Proceedings of the 32nd Annual IEEE International Computer Software and Applications, pp. 76–83.
Hassan, A., Holt, R., 2004. Predicting change propagation in software systems. In: Proceedings of the 20th IEEE International Conference on Software Maintenance. pp. 284–293.
JFlex, 2009. Gerwin Klein. <http://jflex.de/>.
JFreeChart, 2005. <http://www.jfree.org/jfreechart>.
Li, W., Henry, S., 1993. Object-oriented metrics that predict maintainability. The Journal of Systems and Software 23 (2), 111–122.
Li, L., Offutt, A., LLC, A., 1996. Algorithmic analysis of the impact of changes to object-oriented software. In: Proceedings of the International Conference on Software Maintenance, pp. 171–184.
Lorenz, M., Kidd, J., 1994. Object-Oriented Software Metrics: A Practical Guide. Prentice Hall, Englewood Cliffs, NJ.
Mahalanobis, P., 1936. On the generalized distance in statistics. In: Proc. Nat. Inst. Sci. India, vol. 2, pp. 49–55.
OMG, 2007. UML 2.1.2 Superstructure Specification. Object Management Group, (formal/2007-11-02) Edition. <http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF>.
Parnas, D.L., 2001. Some Software Engineering Principles. Addison-Wesley, Longman Publishing Co., Inc., Boston, MA.
Rational Software Architect Standard Edition, 2008. IBM Rational. <http://www-01.ibm.com/software/awdtools/swarchitect/standard/>.
Rountev, A., Volgin, O., Reddoch, M., 2004. Control flow analysis for reverse engineering of sequence diagrams.
Sharafat, A., Tahvildari, L., 2007. A probabilistic approach to predict changes in object-oriented software systems. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, pp. 27–38.
Together 2006 Release 2 for Eclipse, 2006. Borland, <http://www.borland.com/us/products/together/index.html>.
Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., 2005. Predicting the probability of change in object-oriented systems. IEEE Transaction on Software Engineering 31, 601–614.
Wilkie, F., Kitchenham, B., 1999. Coupling measures and change ripples in c++ application software. In: Proceedings of the EASE.

Ah-Rim Han is a Ph.D. candidate in the Computer Science division in the College of Information Science & Technology of KAIST. She received BS (2004) from the Sogang University and MS (2007) from the Korea Advanced Institute of Science and Technology (KAIST) in South Korea. Her research focuses on object-oriented software analysis/design with UML, and software quality assessment based on measurement and empirical software engineering.

Sang-Uk Jeon is a Ph.D. student in the Division of Computer Science at KAIST, Korea. He received the B.S. and M.S. degrees in computer science from KAIST, in 2001 and 2003, respectively. His research interests are design pattern, refactoring, and embedded software development with UML.

Doo-Hwan Bae is a professor of computer science at KAIST. He received his Ph.D. at the Department of Computer Science in the University of Florida. He currently leads the Software Process Improvement Center, funded by Ministry of Knowledge Economy, Korea. His research interests include software process improvement, quantitative project management, software measurement, object-oriented software development, component-based software development, aspect-oriented programming, and embedded software design.

Jang-Eui Hong is an associate professor of Computer Engineering at the school of Electrical and Computer Engineering, Chungbuk National University, Cheongju, Korea. He received his Ph.D in computer science from KAIST, Korea, in 2001. He served as a research member at ADD(Agency for Defense Development) from 2001 to 2004. His research interests include software quality, aspect-oriented programming, embedded software modeling, low-energy software model, and software process improvement.