ABSTRACT

Accurate prediction of fault prone modules in software development process enables effective discovery and identification of the defects. Specific models and algorithms referred to identify the fault proneness in a system. Such prediction models serve the large-scale systems, where verification experts need to focus on development process. The ability of software quality models to accurately identify critical components allows for the application of focused verification activities ranging from manual inspection to automated formal analysis methods. Software quality models, thus, help ensure the reliability of the delivered products. The basic hypothesis of software quality prediction is that a module currently under development is fault prone if a module with the similar product or process metrics in an earlier project (or release) developed in the same environment was fault prone. Therefore, the information available early is directly proportional.

1. INTRODUCTION

A Legacy system is subjected to many maintenance activities, which make degradation of the quality and has many symptoms: pollution, embedded knowledge, poor lexicon, coupling, and layered architecture. Consequently, a lot of corporations or organizations choose to re-engineer the legacy system to new ones with high qualities using latest technologies. Object-oriented (OO) program has proved a relatively good solution for designing and developing high-maintainable software systems. Almost all target systems in reengineering projects are OO system. Given the importance of object-oriented development techniques, one specific area concerned with the message content(parameters) and implementation details is required thus we are not interested in collaboration that must occur so that the use case is executed. The DCM takes as input interactions between classes and not between objects. So we sum up the interactions for objects of class appearing in various sequence diagrams to get interaction between classes. DCM considers only concrete classes that interact and we also consider that this concrete class has already inherited the methods of its super class.

2. DYNAMIC CLUSTERING MECHANISM

Typically, a development lifecycle consists of three main phases: analysis, design and implementation. We apply DCM during the analysis phase of a use case driven, object-oriented methodology, where class interactions are derived from use case description. For each use case a sequence diagram is constructed which describes the sequence of class collaboration that must occur so that the use case is executed.

Assumptions

As the DCM is applied during the analysis phase, no implementation details is required thus we are not concerned with the message content(parameters) and the amount of processing work that each class has to execute to fulfill its responsibilities.

The DCM takes as input interactions between classes and not between objects. So we sum up the interactions for objects of class appearing in various sequence diagrams to get interaction between classes.

Inputs

Assigning frequencies to Use Cases
We classify the refined use cases into two categories: Terminal use cases and internal use cases. A use case is considered terminal if it is neither extended nor used by any other use case, others are called internal use case. We define the frequency of a terminal use case as the number of times that this use case occurs within a given interval. These frequencies are provided by the users. The interval is
determined so that all terminal use cases occur at least once through all the use cases that it extends. If the terminal use case extends more than one use case, then the user must give the frequency with which this use case extends each of the other use cases. For determining the frequency of internal nodes we first consider the ‘extends’ relation and then the ‘uses’ relation. The frequency of internal node which is extended by one or more use cases is equal to the sum of the frequency of each use case that extends it. After this, we proceed by assigning the frequency to internal nodes that are used by other use cases. This is done by adding the frequencies of all use cases that use that internal node.

The frequency of the use case is propagated to the collaborations in its respective sequence diagram. As each of these collaborations may occur in more than one use case, the frequency of the collaborations in the collaboration graph (CG) is given by sum of all the frequencies from the sequence diagrams in which this collaboration occurs.

**The Interaction Graph**

The Interaction Graph (IG) is derived from CG which is given as an input to DCM. In the IG, vertices represent classes and edge represents the frequency with which these two classes interact. The interaction frequency is calculated by summing up the frequency of collaborations of each class that is between them.

**The Process**

The DCM takes IG as input and starts by creating one cluster for each class, called as singleton cluster. The DCM also takes Cluster Threshold (CT) as an input. If two classes interacting with each other with a frequency greater than CT, and the belong to different clusters, then we must be combine the two clusters. CT is the minimum interaction frequency between two classes that will promote combining their respective clusters.

**The pseudo-code for DCM:**

For each class
createSingletonCluster(class)
For each interaction do
(*Each loop considers the interaction between class1 and class2*)

If (class1 and class2 belong to different clusters) and
(interaction frequency between class1 and class2) > Threshold
Merge cluster (class1) and cluster (class2)
EndIf

EndFor

Merging two clusters replaces those with a single cluster, which is the union of the two original clusters.

**3. ALGORITHM DEFINITION AND PROPERTIES**

Definition 1: Two classes $C_1$ and $C_2$ directly interact, if either $C_1$ invokes a method from $C_2$ or $C_2$ invokes a method from $C_1$, or both.

Definition 2: A path between two distinct classes $C_1$ and $C_n$ is sequence $<C_1,..,C_n>$ of classes where classes $C_i$ and $C_{i+1}$ directly interact with a frequency $f_j > CT$, for $j = 1, ..., n$.-1.

The three lemmas below discuss our algorithm properties:

**Lemma 1:** If $C_1$ and $C_2$ directly interact with a frequency $f$, such that $f > CT$, then $C_1$ and $C_2$ belong to the same non-singleton cluster CL.

**Proof:** Through the algorithm above.

**Lemma 2:** Two distinct classes $C_1$ and $C_n$ ($n > 1$) belong to the same non-singleton cluster CL if and only if there exists exactly one path between $C_1$ and $C_n$.

**Proof:** First we prove that if there is at least one path between $C_1$ and $C_n$, then $C_1$ and $C_n$ belong to the same cluster CL. The proof is by induction on the length of the path.

$n=1$: Lemma 1

$n>1$: Induction Hypothesis: For any path of length ($n-1$) between $C_1$ and $C_{n-1}$ implies that $C_{n-1}$ and $C_n$ are in the same cluster CL.

We need to prove that if we have a path of length $n$, $<C_1, ..., C_{n-1}>$, then $C_1$ and $C_{n-1}$ are in the same cluster CL. Since $<C_1, ..., C_{n-1}>$ is of length $n-1$, by Induction Hypothesis, we know that $C_{n-1}$ and $C_n$ are in the same cluster CL. Since $C_{n-1}$ and $C_n$ directly interact with frequency $f_n > CT$, from Lemma 1 we know that $C_{n-1}$ and $C_n$ are in the same cluster. Therefore, $C_1$ and $C_n$ are in the same cluster.

Now, we prove that if $C_1$ and $C_n$ belong to the same cluster CL, then there exists at least one path between $C_1$ and $C_n$. We suggest proving the contra-variant, which is the following:

“If there is no path between $C_1$ and $C_n$, then $C_1$ and $C_n$ belong to different clusters.”

Assume that there is no path between $C_1$ and $C_n$, the condition under which our algorithm groups two classes $C_1$ and $C_n$ is the requirement of an interaction above the threshold between $C_1$ and $C_n$. Since there is no interaction between the threshold between $C_1$ and $C_n$, they must belong to different clusters.

**Lemma 3:** The DCM is not sensitive to the order of input.

**Proof:** Given two classes $C_1$ and $C_2$, after we run the algorithm with a threshold $T$, we can have one of the two situations: $C_1$ and $C_2$ belong to the same cluster CL, $C_1$ belong to CL1 and $C_2$ belong to CL2, where CL1 != CL2.

We need to show that if we run again the DCM with the same threshold $T$, we have the same situation for $C_1$ and $C_2$. Let us start by considering situation 1. After we run the DCM with a given threshold $T$, $C_1$ and $C_2$ are in the same cluster. So, according to Lemma 2, there exists a path connecting $C_1$ and $C_2$ where all interactions in that path have frequency above $T$.
same threshold $T$, $C_1$ and $C_2$ end up in different clusters. But from Lemma 2, $C_1$ and $C_2$ belong to the same cluster. This is a contradiction. Therefore, $C_1$ and $C_2$ must still belong to the same cluster.

Similarly, we prove situation 2 above. No matter how many times we run the DCM with the same threshold $T$, $C_1$ and $C_2$ will always belong to different clusters. According to Lemma 2, there is no path connecting $C_1$ and $C_2$ where all the interaction frequencies are above the threshold $T$, therefore $C_1$ and $C_2$ must belong to different clusters.

Assessing Hot Spots

We run the above algorithm with respect to a given threshold and thus we won’t be able to capture all the different hot spots of the application. That is because, for a given threshold, all the classes that have their maximum interaction frequency below the current threshold will not be clustered. Thus we have to vary the threshold frequency and run our algorithm multiple times. For different values of threshold frequency, we will get different hot spots.

Now to assess the importance of the hot spots with respect to class development at design and implementation phase, we have a metric system.

Metric: For each non-singleton cluster (IN), and then add up the frequencies of the interactions crossing the cluster boundary (OUT).

If $IN > OUT$: The activities inside this cluster are highly used by the application, and therefore should be carefully designed.

If $IN <= OUT$: The activities are not so frequent when compared with the frequency that classes inside the cluster interact with classes outside the cluster.

4. IMPLEMENTATION

Dynamic coupling measures are integrated into existing iterative process. The first few steps are identical to the original process. The improvement happens in the Reengineer Procedures cycle. Comparing with the original process, one more step, Dynamic Coupling Evaluation, is added; and one step, Equivalence Test, is enhanced. Alessandro's iterative reengineering process can be considered static, for all phases are performed under the static analysis of the existing legacy system and never take advantage of the executable properties of legacy system. Once a given component has been reengineered, the process is repeated and next component is reengineered, until the whole legacy system has been reengineered. The iterative model allows the coexistence of the old and new system and it can ensure the system keep working during the reengineering process. Improved process takes full advantage of the runtime information in Equivalence Test and generates dynamic coupling measures in Dynamic Coupling Evaluation phase.

a. Analyze Legacy System: During reengineering process, many maintenance requirements are involved which affect on different sets of components. Requirements should be managed carefully. A requirement should be hold until all relative components are replaced. The first step in the iterative process is identifying and mapping all legacy components to every maintenance requirement.

b. Classify Data: Legacy data records are other critical factor in reengineering process. They are identified and interpreted according to the background. The results include data name, data type and brief description. User can define the data type based on the different business scopes. The purpose of this step is helping software engineers understand the system data.

c. Redesign Database: In previous phase, software engineers have classified and understood the data records in legacy system very well. In this phase, software engineer can redesign the database. The guidelines are keeping primary business data which is necessary for executing business functionality, removing residual control data which are only related to legacy system implementation and enhancing or restructuring data records which are essential for implementing new system design.
d. Restore legacy components and Migrate Data: Different from previous steps, these two steps are normally executed concurrently in every iterative cycle. Before every iterative cycle, the access of all data records should be redirected in every related legacy components. These activities are named as Restore legacy components. The purpose is only making the new system compatible with the original one when the data are migrated. Meanwhile, the legacy data records are migrated from old format to redesigned database.

e. Equivalence Test: In original iterative process, engineers ensure that the functionalities and operation of the reengineered system is the same as the one before by executing all test cases to. In additional, collecting dynamic coupling data which are used to perform following Dynamic Coupling Evaluations is enhanced.

f. Reengineer Procedure: In this phase, software engineers analyze the degraded procedure and introduce suitable remedies to improve the functionality and quality requirements. Normally, they perform three activities: managing the data access and their relations to present domain knowledge; updating user interface and making it friendlier to enhance users' experiences; executing maintenance requirements to fulfill the functionality requirements. an additional task, collecting dynamic executing information, is added to enhance the original iterative process. At the end of this phase, the preconditions for dynamic coupling are ready.

g. Dynamic Coupling Evaluations: This is the emphases of this paper. In reengineering projects, external quality is one of the significant standards. Software coupling measures are involved in our process to make the reengineered system reach high external quality. Evaluations on system or component level are performed by aggregating class level coupling to help software engineers improve external quality reengineer procedure phase.

h. Empty Residual DB: Our iterative process allows residual DB work with redesigned DB together. The residual DB will be removed until they are not access by reengineered system.

i. Reconstruct Documents: This phase should be executed for every phase in the process, because it is necessary to keep the documents up to data and can describe the system's implementation.

5. PRONENESS IN OBJECT ORIENTED SOFTWARE SYSTEM

Proneness throws multiple meaning, considering the proneness as related with an object oriented software system, it definite will not correspond to “face down” or “lying” or any other physical form, but it is rather used in the following sense: Susceptible, vulnerable, disposed (to), likely, tending. So in a software system, Proneness is associated with the characteristics of the system. Proneness gives the inclination or tendency or likeliness of the software system to have a certain characteristic or behavior.

For example, when we say that a given module (class in an Object Oriented scenario) is “error prone”, we actually mean that the class under consideration is more likely to contain an error with respect to other classes present. The error proneness of a given module is proportional to its size. Generally a logarithmic relation. There exist two main types of proneness in an object oriented system: Fault Proneness and Change Proneness. Fault proneness of an Object Oriented system is the tendency of a component whether a class or a cluster of classes to generate faults. Requirement of accurate prediction:-

Accurate prediction of fault prone modules in software development process enables effective discovery and identification of the defects. Specific models and algorithms referred to identify the fault proneness in a system. Such prediction models serve the large-scale systems, where verification experts need to focus on development process.

The ability of software quality models to accurately identify critical components allows for the application of focused verification activities ranging from manual inspection to automated formal analysis methods. Software quality models, thus, help ensure the reliability of the delivered products. The basic hypothesis of software quality prediction is that a module currently underdevelopment is fault prone if a module with the similar product or process metrics in an earlier project (or release) developed in the same environment was fault prone. Therefore, the information available early is directly proportional.

Fault Proneness as an external Quality Attribute:- Fault Proneness is an external quality attribute which can be used to predict the fault proneness - prone modules. Fault proneness is defined as a probability of detecting a fault in the class. We focus on attribute, which can be measured easily and objectively, and which is least affected by variability. The example of the same could be fault proneness and defect density. Defect Density is a software quality attribute, which gives the reliability measure of the software product. Defect Density is given as defects found per KLOC (k lines of code). The result can be high-risk prone modules as per the Pareto principle that greater defects lie in the 20% of the modules and considered.

6. CONCLUSION

A typical software fault prediction process includes two steps, as shown in Figure, First, a fault prediction...
model is built using previous software metrics and fault data belonging to each software module (class or method level). After this training phase, fault labels of program modules can be estimated using this model the selection of metrics type is dependent on the programming paradigm used in the project and research targets. Research has revealed that 60% of projects used method-level metrics. However, there are cases when previous fault data are not available. For example, a software company might start to work on a new project domain or might expect fault predictors newly in their development cycle. In addition, current software version’s fault data might not be collected and therefore, there might not exist in any previous fault data for the next release of the software. In these cases, supervised learning approaches cannot be developed because of the absence of class labels. There are a few software fault prediction studies which do not use prior fault data for modeling. Zhong et al. used K-means and Neural-Gas algorithms to cluster modules and an expert explored several statistical data within each cluster to label each cluster as fault-prone or not fault-prone. Furthermore, the selection of the cluster number, K, is done heuristically when k-means clustering method is chosen and this process can affect the model’s performance drastically.

7. References
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.