Milestone 2 & 3

1. Related Work

1.1. Summary

a. **Research Goal:** Our goal is to investigate relation between C&K metrics and fault-proneness using bad smells as faulty classes.

b. **Motivation:** As the software evolves, the code and the architecture become more complex, which can inevitably induce bugs in the system. Finding bugs later in the developmental stages can increase the cost of software remodeling. Software developers usually use various metrics for determining the quality of the system. Using these metrics for predicting faults during development as well as every stage can increase the efficiency of the overall development process. [2]

c. **Approach/Methodology:** The response variable used to validate the OO design metrics is binary, i.e., was a fault detected in a class during testing phases? they used logistic regression, a standard technique based on maximum likelihood estimation, to analyze the relationships between metrics and the fault proneness of classes. First univariate logistic regression was used, to evaluate the relationship of each of the metrics in isolation and fault proneness. Then, multivariate logistic regression was performed, to evaluate the predictive capability of those metrics that had been assessed sufficiently significant in the univariate analysis. The global measure of goodness of fit used for such a model is assessed using R^2 method.[2]

d. **Empirical Findings/Experimental Results:** Five out of the six Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during the high and low level design phases of the life-cycle.
   i. DIT : Very Significant
   ii. RFC : Very Significant

iii. NOC : Very Significant
iv. CBO : Significant
v. WMC : Less Significant
vi. LCOM : Insignificant

e. **Conclusions:** C&K metrics are good indicators of quality of the code and can be used for fault proneness in our study.

1.2. Comparison with the study proposed:

a. **Similarities:** The approach we are using for our study is very similar to the one mentioned in paper. We will be using univariate logistic regression for identifying metrics having significant correlation with fault proneness, and multivariate regression for building a predictive model.

b. **Differences:** In the paper, faulty classes are being verified into binary variables during testing of classes, while in our study we will be using DesigniteJava[4] a plug-in tool to detect bad smells then classify them as faulty classes. In addition, after doing some research we concluded that combinations of criteria used for detecting code-smells differ between multiple research papers, also we concluded that different tools give different precision and recall values for detecting code smells, so we will be conducting further study on criteria to be used for our purpose and then custom coding will be done for detection.

c. **Limitations:** The study in paper is performed on systems developed by students, which are taken as study subjects. These systems would consist of bugs and code smells which would have not been present if professionally developed systems were used in the first place. Additionally, faulty classes are being defined during testing manually which might incorporate errors induced by humans.

d. **Improvements:** We propose to study the correlation between the selected metrics individually as well as in combination on fault proneness. Also, we are using specialized software's for the entire process which might improve the results of analysis. Studying multiple releases for different systems ensure the validity of results as each system signify different work setting and different versions signify, the influence of practices on results. Our study will be done on professionally developed systems, which can simulate the use of these metrics in real case scenarios.

2. Data Collection

2.1. Internal Metrics Implementation[5] [6]:

1. CBO (Count Class Coupled): CBO represents the number of classes coupled to a given class. According to this metric "Coupling Between Object Classes" (CBO) for a class is a count of the number of other classes to which it is coupled. Theoretical basis of CBO relates to the notion that an object is coupled to another object if one of them acts on the other, i .e. methods of one use methods or instance variables of another (Chidamber et. al. 1994). As Coupling between Object classes increases, reusability decreases and it becomes harder to modify and test the software system. Chidamber and Kemerer state that their definition of coupling also applies to coupling due to inheritance, but do not make it clear if all ancestors are involuntarily coupled or if the measured class has to explicitly access a field or method in an ancestor class for it to count. In general, the definition of the CBO is ambiguous, and makes its application tough (Mayer et. al. 1999).

2. NOC (Count Class Derived): Number of children (NOC) of a class is the number of immediate sub-classes subordinated to a class in the class hierarchy. Theoretical basis of NOC metric relates to the notion of scope of properties. It is a measure of how many sub-classes are going to inherit the methods of the parent class (Chidamber et. al. 1994).

3. WMC (Sum Cyclomatic): It is the sum of the cyclomatic complexity of each method for all methods of a class.

- If a Class C, has n methods and c_1, c_2 …c_n be the complexity of the methods, then WMC(C)= $c_1 + c_2 + \ldots + c_n$.

The specific complexity metric that is chosen should be normalized so that nominal complexity for a method takes on a value of 1.0. If all method complexities are considered to be unity, then WMC = n, the number of methods. WMC break an elementary rule of measurement theory that a measure should be concerned with a single attribute (Chidamber et. al. 1994).

4. DIT (Max Inheritance Tree): The DIT for class X is the maximum path length from X to the root of the inheritance tree. According to this metric Depth of Inheritance (DIT) of a class is "the maximum length from the node to the root of the tree. The definition of DIT is ambiguous when multiple inheritance and multiple roots are present.

5. LCOM (Percent Lack of Cohesion): The LCOM metric value is calculated by removing the number of method pairs that share other class field from number of method pairs that does not share any field of other class. Consider a Class C1 with n methods M1 , M2 ..., Mn . Let {Ij } = set of instance variables used by method Mi .There are n such sets {I1},{I2}... {In}. Let $P = \{ (I_i, I_j) \mid I_i \cap I_j = \varnothing \}$ and $Q = \{ (I_i, I_j) \mid I_i \cap I_j \neq \varnothing \}$. If all n sets {I1},{I2}... {In}. are ∅ then let P = ∅ [4]. Lack of Cohesion in Methods (LCOM) of a class can be defined as:

$$\text{LCOM} = |P| - |Q|, \text{ if } |P| > |Q|$$

$$\text{LCOM} = 0 \text{ otherwise}$$

The high value of LCOM indicates that the methods in the class are not really related to each other and vice versa. According to above definition of LCOM the high value of LCOM implies low similarity and low cohesion, but a value of LCOM = 0 doesn't implies the reverse (Mayer et. al. 1999).

6. Cyclomatic Complexity: Cyclomatic complexity metric used for measuring the program complexity through decision-making structure such as if-else, do-while, foreach, goto, continue, switch case etc. expressions in the source code. Cyclomatic complexity only counts the independent paths by a method or methods in a program. Complexity value of program is calculated using following formula:

V (G) =E -N +P, where
V (G) Cyclomatic complexity
E No. of edges of decision graph
N No. of nodes of decision graph
P No. of connected paths

If there is no control flow statement like IF statement then complexity of program will be 1. It means there is single path for execution. If there is single IF statement then it provides two paths: one for TRUE condition and other for FLASE condition, so complexity will be 2 for it with single condition.

7. RFC (Count Decl Method All): According to this metric Response For a Class (RFC) can be defined as | RS |, where RS is the response set for the class. The response set for the class can be expressed as:

$$RS = \{M\} \cup \text{all } i \, \{ R\, i \}$$

where {Ri} = set of methods called by method i and {M} = set of all methods in the class. The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. (Henderson et. al. 1991) But here the point to be noted is that because of practical considerations, Chidamber and Kemerer recommended only one level of nesting during the collection of data for calculating RFC. This gives incomplete and ambiguous approach as in real programming practice there exists “Deeply nested call-backs” that are not considered here. If CK intend the metric to be a measure of the methods in a class plus the methods called then the definition should be redefined to reflect this (Mayer et. al. 1999).

2.2. <u>Challenges Faced:</u>

- For this study Scitools Understand software is used for metrics collection, it provides the data of metrics not only on classes, but also on files and directory, resulting to an incorrect analysis, so extra cleaning had to be done on the data.

- Software systems selected for study are well known, widely used and large systems, retrieving data on personal laptops having lower ram memory led to multiple crashes and was time consuming to some extent. Arrangements had to be done for a laptop with RAM size of atleast 12 GB for metrics collection.
- Initially we were using only CK metrics and with single criteria for each type of code-smell category, as our knowledge was limited. After some discussions and extra study, we concluded that there should be multiple criteria for each type of code-smell detection. We finally settled in for the use of an external tool, but we propose to study further and develop our own set of criteria for code-smell classification in coming milestones.

2.3. Correctness of Metrics :

C&K metrics are widely used and have been validated multiple times, so we mention the hypothesis for the construct of these metrics instead, with regards to our proposed study.

- **WMC:** A class with significantly more member functions than its peers is more complex and, by consequence, tends to be more fault-prone [2].
- **DIT:** A class located deeper in a class inheritance lattice is supposed to be more fault-prone because the class inherits a large number of definitions from its ancestors. In addition, deep hierarchies often imply problems of conceptual integrity, i.e., it becomes unclear which class to specialize from in order to include a subclass in the inheritance hierarchy [2].
- **NOC:** Classes with large number of children (i.e., subclasses) are difficult to modify and usually require more testing because the class potentially affects all of its children. Furthermore, a class with numerous children may have to provide services in a larger number of contexts and must be more flexible. We expect this to introduce more complexity into the class design and, therefore, we expect classes with large number of children to be more fault-prone[2].

- **CBO:** Highly coupled classes are more fault-prone than weakly coupled classes because they depend more heavily on methods and objects defined in other classes [2].
- **RFC:** Classes with larger response sets implement more complex functionalities and are, therefore, more fault-prone [2].
- **LCOM:** Classes with low cohesion among its methods suggests an inappropriate design (i.e., the encapsulation of unrelated program objects and member functions that should not be together) which is likely to be more fault-prone [2].

2.4. Git Repository :

- https://github.com/HarshDivecha/SOEN6611

3. External Metrics Collection:

3.1. Tools used:

- Scitools Understand: For collecting C&K metrics.
- DesigniteJava: For Code-Smells
- Jupyter Notebook: For data analysis.

3.2. Tools Explored:

- BugZilla
- PMD
- JDeodrant
- InFusion

4. References

[1] Software Measurement PPTs by Dr. Rodrigo Morales Alvarado.

[2] A Validation of Object-Oriented Design Metrics as Quality Indicators, Victor R. Basili, 1996

[3] Evaluation of code smells and detection tools: https://doi.org/10.1186/s40411-017-0041-1

[4] DesigniteJava: https://github.com/tushartushar/DesigniteJava

[5] “A Critical Suggestive Evaluation of CK Metric”, Pacific Asia Conference on Information Systems, PACIS 2005, Bangkok, Thailand, July 7-10, 2005.

[6] “Analysis of Various Software Metrics Used To Detect Bad Smells”, Sukhdeep Kaur, Dr. Raman Maini, The International Journal Of Engineering And Science (IJES), Volume 5, Issue 6, Pages PP -14-20 2016 ISSN (e): 2319 – 1813 ISSN (p): 2319 – 1805.