Department of Computer and Software Engineering

SOEN 6611-E (Software Measurement)

Winter 2019

Milestone 4

"We certify that this submission is the original work of members of the group and meets the Faculty's Expectations of Originality"

Submitted by (Team 10):

- Hari Narayanan Kannan **(40047827)**
- Harsh Divecha **(40084737)**
- Ramit Basra **(40045043)**
- Dhaval Chandreshkumar Modi **(40083289)**
- Venkata Lakshmi Sai Krishna Chadalavada **(40087977)**

Submitted to:

Dr. Rodrigo Morales Alvarado

Milestone 1

1. Team Formation:

No.	Name	Roles/Responsibilities
1.	Hari Narayanan Kannan	Documentation, Research, Implementation of tools
2.	Venkata Lakshmi Sai Krishna Chadalavada	Documentation, Research
3.	Dhaval Chandreshkumar Modi	Research, Implementation of tools
4.	Ramit Basra	Documentation, Research
5.	Harsh Divecha	Coding, Research, Implementation of tools

2. Type of Study:

We have chosen Type-1 study. In this study we will be studying fault-proneness in relation to C&K. Source for mining repository has been selected as GitHub adhering to all the criteria mentioned in requirements, as it is a very well known and an extensively used revision control system.

3. Related Studies:

- CK metrics will be studied in correlation to faulty classes by detecting bad smells in the code retrieved as binary output (0 for no bad smell and 1 if detected) and finally using Bayesian inference for different releases of the selected open source software. [1]
- Faults prediction techniques will be studied using Logistic regression model and Bayesian inference altogether. Bayesian inference was intended to be evaluated for risk related data. The study suggests that there is a relationship between faulty classes and object-oriented metrics [9].
- Logistic regression model will be used as the base. Moreover, Univariate analysis will be used to identify metrics with significant impact on faults. Statistically significantly metrics will be selected and used for Multivariate model [10].

4. Projects:

Projects selected are all based on Java, as most of the open source software's are available in this language and codes written in java require Object Oriented concepts to be implemented implicitly and it is one of the most extensively used and documented languages known. All the projects selected have more than 300k lines of code which increases the probability of the code being fault prone or consisting of defects in general.

Project Selected:	**Tools and plug-ins [6]:**
• Ant • Mockito • Wildfly	• Understand 5.0.9 by Scitools. • Jupyter notebook. • DesigniteJava.

Revisions Selected for studying the most desirable projects:

Mockito	**Ant**	**Wildfly**
• 2.25.0	• 1.01.0	• 16.0
• 2.21.1	• 1.05.2	• 12.0
• 2.17.6	• 1.10.5	• 8.0

5. Metrics:

5.1 C&K Metrics:

The C&K Metric Suite has been implemented to analyse fault proneness in the projects under consideration [3] [4].
Chidamber and Kemerer (CK) et al. [2] gives the formal definition of the metrics as follows:

(i) **Weighted Methods per Class (WMC):**
This measures the sum of complexity of the methods in a class. The complexity of the class may be calculated by the cyclomatic complexity of the methods. The high value of WMC indicates that the class is more complex as compare to the low values.

(ii) **Depth of Inheritance Tree (DIT):**
DIT metric is used to find the length of the maximum path from the root node to the end node of the tree. DIT represents the complexity and the behavior of a class, and the complexity of design of a class and potential reuse.

(iii) **Number of children (NOC):**
According to Chidamber and Kemerer, the Number of Children (NOC) metric may be defined for the immediate sub class coordinated by the class in the form of class hierarchy. These points are come out as NOC is

used to measure that "How many subclasses are going to inherit the methods of the parent class". The greater the number of children, the greater the potential for reuse. The greater the number of children, the greater the likelihood of improper abstraction of the parent class.

(iv) Coupling between Objects (CBO):
CBO is used to count the number of the class to which the specific class is coupled. The rich coupling decreases the modularity of the class making it less attractive for reusing the class and more high coupled class is more sensitive to change in other part of the design through which the maintenance is so much difficult in the coupling of classes.

(v) Response for class (RFC):
The response set of a class (RFC) is defined as set of methods that can be executed in response and messages received a message by the object of that class. Larger value also complicated the testing and debugging of the object through which, it requires the tester to have more knowledge of the functionality.

(vi) Lack of Cohesion in Methods (LCOM):
This metric is used to count the number of disjoints methods pairs minus the number of similar method pairs used. Since cohesiveness within a class increases encapsulation it is desirable and due to lack of cohesion may imply that the class is split in to more than two or more sub classes. Low cohesion in methods increase the complexity, when it increases the error proneness during the development is so increasing.

<u>5.2 Bad Smells:</u>

Bad smell means a piece of code in program that signifies the design problem in software code structure. Bad smell is a bug not a run time error. These bad smells require refactoring to improve the code quality [8].

We have used DesigniteJava tool to detect bad smells in the projects. The following are the list of bad smells:

Design smells	**Implementation smells**
Abstraction Design smells	Long Method
Encapsulation Design smells	Complex Method
Modularisation Design smells	Long Parameter List
Hierarchy Design smells	Long Identifier
	Long Statement
	Complex Conditional
	Virtual Method Call from Constructor
	Empty Catch Block
	Magic Number
	Duplicate Code
	Missing Default

i. <u>Design Smells [17]:</u>

- **<u>Abstraction Design Smells</u>:**

The definition of abstraction design principle is given as follows:
"The principle of abstraction advocates the simplification of entities through reduction and generalization: reduction is by elimination of unnecessary details and generalization is by identification and specification of common and important characteristics."
The following are the smells detected that violates the principle of abstraction:

Imperative Abstraction, Multifaceted Abstraction, Unutilized Abstraction, **Duplicate Abstraction**.

- **<u>Encapsulation Design Smells</u>:**

The definition of encapsulation design principle is given as follows:

"The principle of encapsulation advocates separation of concerns and information hiding through techniques such as hiding implementation details of abstractions and hiding variations."

The following are the smells detected that violates the principle of encapsulation:

Deficient Encapsulation, **Unexploited Encapsulation.**

- **Modularization Design Smells:**

The definition of modularization design principle is given as follows:

"The principle of modularization advocates the creation of cohesive and loosely coupled abstractions through techniques such as localization and decomposition."

The following are the smells that violates the principle of modularization:

Broken Modularization, **Insufficient Modularization, Hub-like Modularization, Cyclically-dependent Modularization.**

- **Hierarchy Design Smells:**

The definition of modularization design principle is given as follows:

"The principle of hierarchy advocates the creation of a hierarchical organization of abstractions using techniques such as classification, generalization, substitutability, and ordering."
The following are the smells that violates the principle of hierarchy:

Wide Hierarchy, Deep Hierarchy, Multipath Hierarchy, Cyclic Hierarchy, Rebellious Hierarchy, Unfactored Hierarchy, Missing Hierarchy.

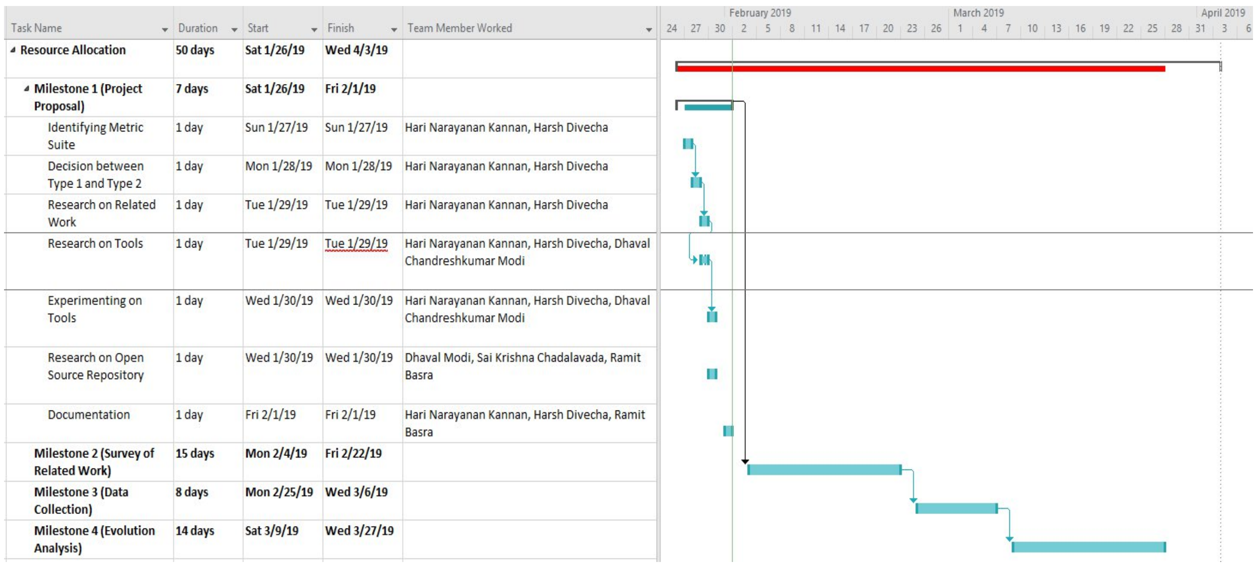
ii. Implementation Smells:

- **Duplicate Code**: Identical or very similar code exists in more than one location.
- **Long Method**: A method, function, or procedure that has grown too large.
- **Long Parameter List**: More than three or four parameters for a method.
- **Magic Number**: A magic number is a numeric value that's encountered in the source but has no obvious meaning. This "anti-pattern" makes it harder to understand the program and refactor the code.
- **Missing Default Case:** A default case is missing in a *case* or *selector* statement.
- **Long Statement:** The code contains long statements (that typically do not fit in a screen).
- **Complex Conditional:** The longer a piece of code is, the harder it's to understand. Things become even more hard to understand when the code is filled with conditions.
- **Virtual Method Call from Constructor**: Constructors of derived types are called after constructors of base types. On the other hand, calls to virtual methods are always executed on the most derived type. This means that if you call a virtual member from the constructor in a base type, each override of this virtual member in a derived type will be executed *before* the

constructor of the derived type is called. If the override in the derived type uses its members, this can lead to confusion and errors.

- **Empty Catch Block:** An empty catch block code smell undermines the purpose of exceptions. When an exception occurs, nothing happens and the program fails for an unknown reason. The application can be in an unknown state which will affect the subsequent processing.
- **Long Identifier**: Using too long identifiers may actually decrease the readability a lot.

6. Resource Planning:



Task Name	Duration	Start	Finish	Team Member Worked
Resource Allocation	**50 days**	**Sat 1/26/19**	**Wed 4/3/19**	
Milestone 1 (Project Proposal)	**7 days**	**Sat 1/26/19**	**Fri 2/1/19**	
Identifying Metric Suite	1 day	Sun 1/27/19	Sun 1/27/19	Hari Narayanan Kannan, Harsh Divecha
Decision between Type 1 and Type 2	1 day	Mon 1/28/19	Mon 1/28/19	Hari Narayanan Kannan, Harsh Divecha
Research on Related Work	1 day	Tue 1/29/19	Tue 1/29/19	Hari Narayanan Kannan, Harsh Divecha
Research on Tools	1 day	Tue 1/29/19	Tue 1/29/19	Hari Narayanan Kannan, Harsh Divecha, Dhaval Chandreshkumar Modi
Experimenting on Tools	1 day	Wed 1/30/19	Wed 1/30/19	Hari Narayanan Kannan, Harsh Divecha, Dhaval Chandreshkumar Modi
Research on Open Source Repository	1 day	Wed 1/30/19	Wed 1/30/19	Dhaval Modi, Sai Krishna Chadalavada, Ramit Basra
Documentation	1 day	Fri 2/1/19	Fri 2/1/19	Hari Narayanan Kannan, Harsh Divecha, Ramit Basra
Milestone 2 (Survey of Related Work)	**15 days**	**Mon 2/4/19**	**Fri 2/22/19**	
Milestone 3 (Data Collection)	**8 days**	**Mon 2/25/19**	**Wed 3/6/19**	
Milestone 4 (Evolution Analysis)	**14 days**	**Sat 3/9/19**	**Wed 3/27/19**	

Milestone 2 & 3

1. Related Work

1.1. Summary

Our goal is to investigate relation between C&K metrics and fault-proneness using bad smells for faulty classes. 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. [12]

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.[12]

It was concluded that C&K metrics are good indicators of quality of the code and can be used for fault proneness in our study. Five out of the six Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during the high level 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

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 [15] [16]:

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:

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

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

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. Challenges Faced:

- 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 at least 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.

- Data cleaning and Preparation was a major challenge as the output of Designite java separates the Packages, Methods and Types of classes then detects code smells. This step composes 80% of the work time, the output from Designite java was to be manipulated in a way that it could be merged with the output from Scitool's Understand.

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 [12].
- **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 [12].
- **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 [12].
- **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 [12].
- **RFC:** Classes with larger response sets implement more complex functionalities and are, therefore, more fault-prone [12].
- **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 [12].

2.4. Git Repository:

- https://github.com/HarshDivecha/SOEN661

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

Milestone 4 - Data Analysis

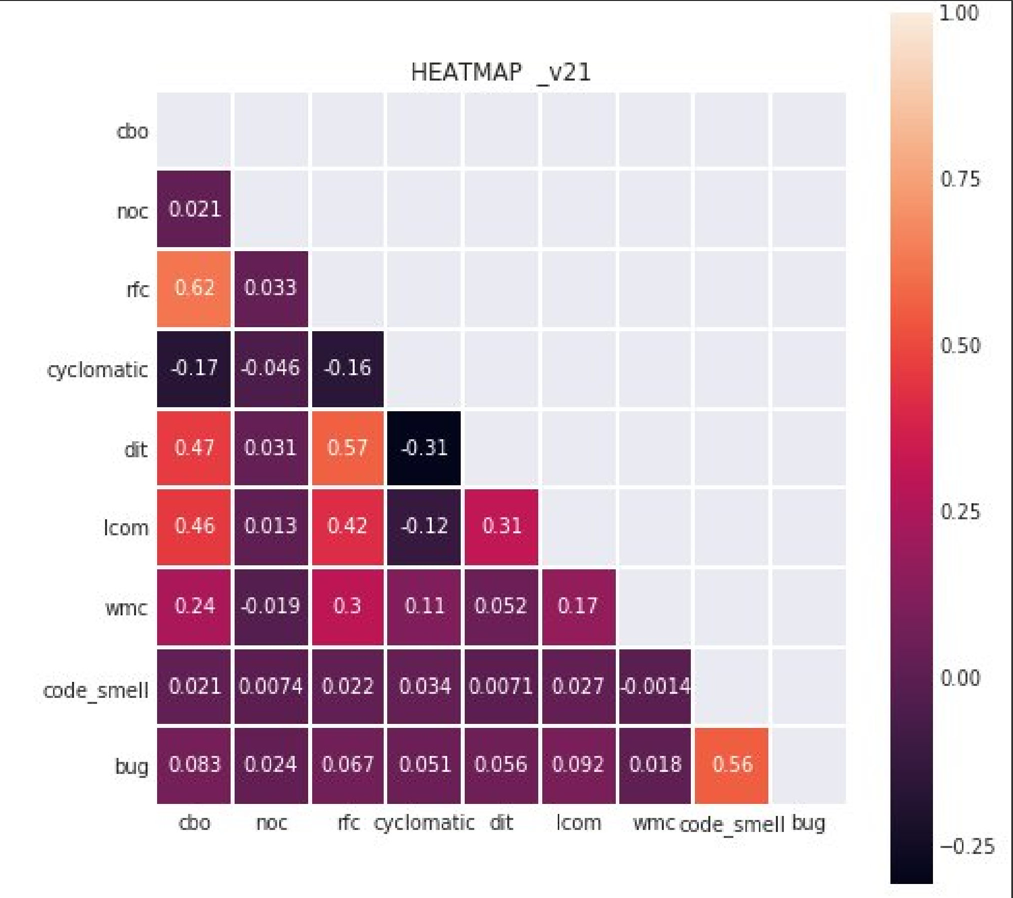
- **Empirical Study Type-1**:

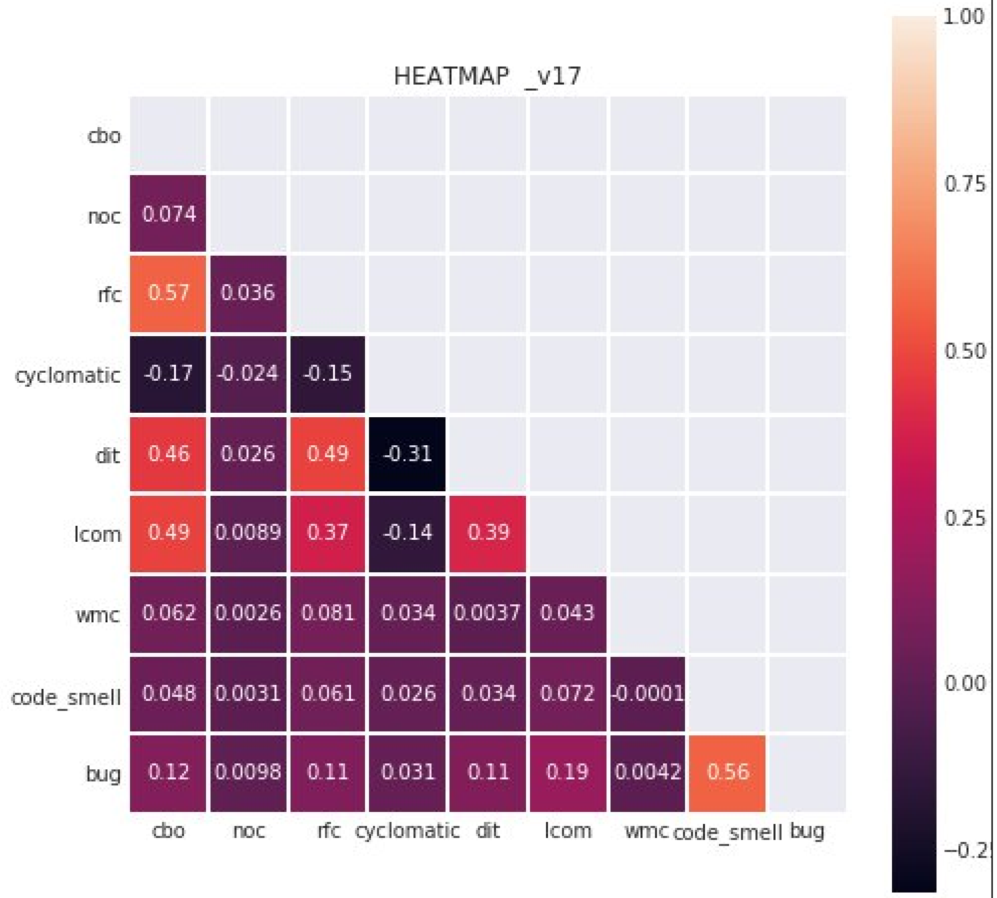
1. <u>Statistical tests</u>:

We have performed regression analysis. Regression analysis is an indispensable tool for analyzing relationships between two or more variables.We consider presence of bug (if code_smell > 0 : 1, else: 0) as dependent variable. CK metrics are considered to be independent variables. The fundamental assumptions of regression analysis is that the relationship between the dependent and independent variables is linear. We are using logistic regression as we have only two outputs, i.e. binary.

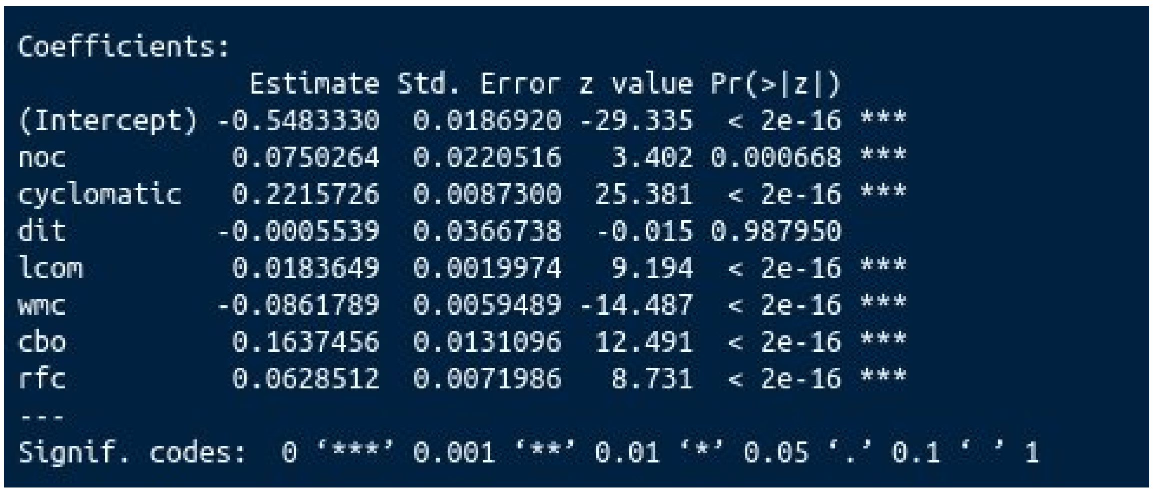
2. <u>Variables with high correlation</u>:

From the open source projects taken under consideration, the following are the high correlations that we observe. They are listed are follows:

<u>Mockito:</u> ***Variables with >45% correlation:***	
CBO vs RFC (62%) CBO vs DIT (47%) CBO vs LCOM (46%) RFC vs DIT (57%)	

<u>Wildfly:</u> ***Variables with >45% correlation:***	
CBO vs RFC (57%) CBO vs DIT (46%) CBO vs LCOM (49%) RFC vs DIT (49%)	

3. <u>Regression with all independent variables in the model</u>:



```
Coefficients:
              Estimate Std. Error z value Pr(>|z|)    
(Intercept) -0.5483330  0.0186920 -29.335  < 2e-16 ***
noc          0.0750264  0.0220516   3.402 0.000668 ***
cyclomatic   0.2215726  0.0087300  25.381  < 2e-16 ***
dit         -0.0005539  0.0366738  -0.015 0.987950    
lcom         0.0183649  0.0019974   9.194  < 2e-16 ***
wmc         -0.0861789  0.0059489 -14.487  < 2e-16 ***
cbo          0.1637456  0.0131096  12.491  < 2e-16 ***
rfc          0.0628512  0.0071986   8.731  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

figure 1.

To pick the independent variables which contribute significantly to the model we refer to the P-value, this value tells us the significance of the variables which show valuable contribution to the model. A well-know and widely accepted significant alpha is 0.5.

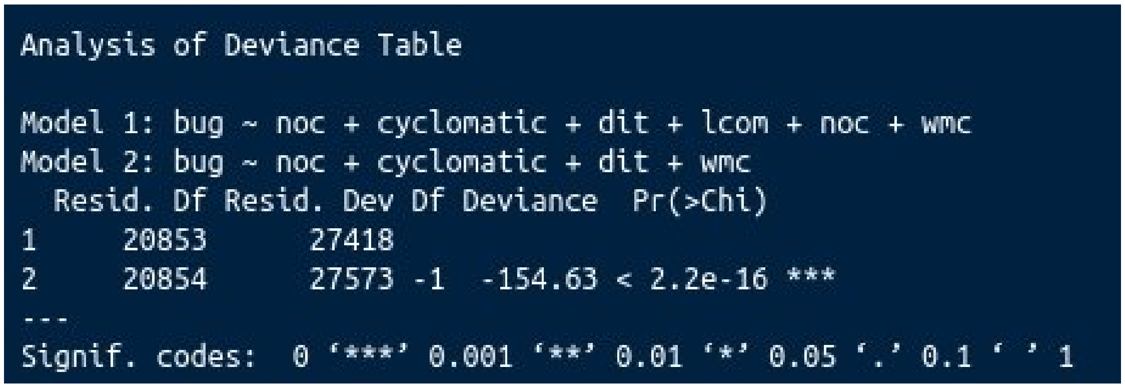
In figure 1 above, significance codes display levels of significance. We accept the variables with *** as they execute a P-value between (0.000, 0.001) which is below alpha = 0.5. We can discard '**dit**' as its significance is **0.98**, which shows that it does not contribute much to our model.

Regression plots have been attached at the end of document after references.

Regression Equation :

-0.54 + 0.07(noc) + 0.22(cyclomatic) + 0.01(lcom) - 0.08(wmc) + 0.16(cbo) + 0.06(rfc).

4. Hypothesis:



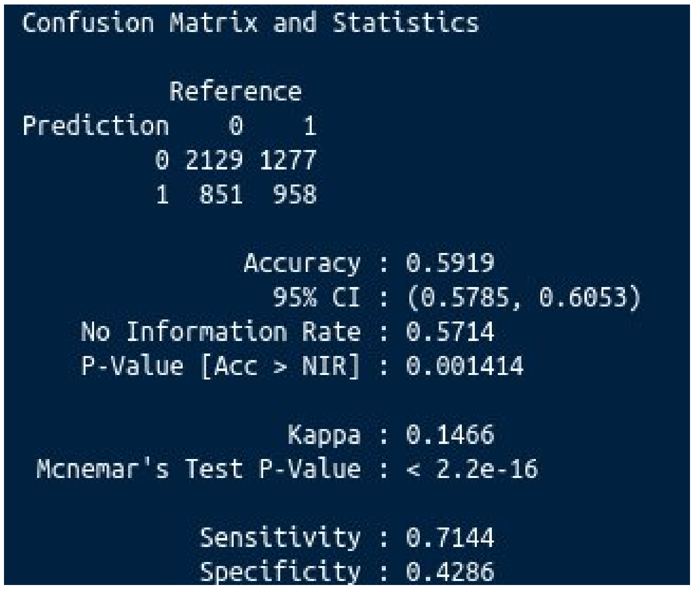
```
Analysis of Deviance Table

Model 1: bug ~ noc + cyclomatic + dit + lcom + noc + wmc
Model 2: bug ~ noc + cyclomatic + dit + wmc
  Resid. Df Resid. Dev Df Deviance  Pr(>Chi)
1     20853      27418
2     20854      27573 -1  -154.63 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

figure 2.

Null hypothesis:
C&K metrics show no correlation with Code Smells/bugs.

Alternative hypothesis:
C&K metrics execute significant correlation with Code Smells/bugs.

We **reject** the null hypothesis, based on the mentioned observations.



```
Confusion Matrix and Statistics

          Reference
Prediction    0    1
         0 2129 1277
         1  851  958

               Accuracy : 0.5919
                 95% CI : (0.5785, 0.6053)
    No Information Rate : 0.5714
    P-Value [Acc > NIR] : 0.001414

                  Kappa : 0.1466
 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 0.7144
            Specificity : 0.4286
```

Analysis on initial prediction model (To be improved in final report):

- In regression plots we see that not all independent variables show linear correation with presence of bugs. Including these variables have decreased the model accuracy as they are performing a linear but discrete correaltion.
- The basic model without hyper-parameter tuning has a 60% prediction accuracy which will be improved further by eliminating the metrics not executing a linear relationship with presence of bug.
- Specificity of 42% and sensitivity of 71%, shows that the model is not overfitted. It is important to take sensitivity and specificity into consideration because a model with 90% accuracy but specificity and sensitivity being very low would show over or under fitting of the model.
- In *figure 2,* after eliminating insignificant variables, we see that the model 2 is significant compared to model 1.

5. References:

[1] Amit Sharma, Sanjay Kumar Dubey, “Comparison of Software Quality Metrics for Object- Oriented System”, IJCSMS International Journal of Computer Science & Management Studies, Special Issue of Vol. 12, June 2012 ISSN (Online): 2231 –5268.

[2] C. Shyam, Kemerer, F. Chris, "A Metrics Suite for Object- Oriented Design" M.I.T. Sloan School of Management, pp. 53-315, 1993.

[3] Istehad Chowdhury, Mohammad Zulkernine, “Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities”, Journal of Systems Architecture 57 (2011) 294–313.

[4] Ramanath Subramanyam and M.S. Krishnan, “Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects”, IEEE Transactions on Software Engineering, Vol. 29, No. 4, April 2003.

[5] E Fioravanti, P. Nesi, “A Study on Fault-Proneness Detection of Object-Oriented Systems”, Proceedings Fifth European Conference on Software Maintenance and Reengineering, 07 August 2002, 6905819.

[6] Rüdiger Lincke, Jonas Lundberg and Welf Löwe, “Comparing Software Metrics Tools”, Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008.2.

[7] Radu Marinescu, “Measurement and Quality in Object-Oriented Design,” PhD thesis, Politechnica University of Timisoara, Romania, 2002.

[8] Sukhdeep Kaur1 & Dr. Raman Maini2, Analysis of Various Software Metrics Used to Detect Bad Smells, 2016

[9] Heena Kapila & Satwinder Singh, Analysis of CK Metrics to predict Software Fault-Proneness using Bayesian Inference, International Journal of Computer Applications (0975 – 8887)

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

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

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

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

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

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

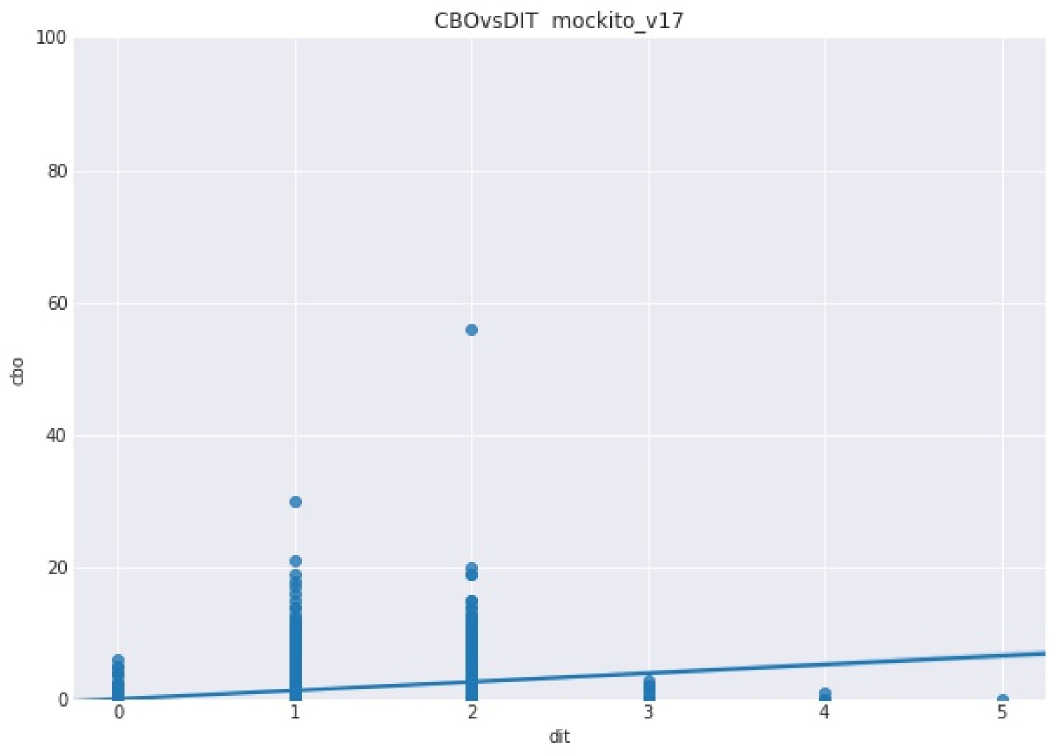
[16] "Analysis of Various Software Metrics Used to Detect Bad Smells", Sukhdeep Kaur, Dr. Raman Maini, The International Journal of Engineering and Science (IJES), Volume 5, Issue 6, Pages PP -14-20 2016 ISSN (e): 2319 – 1813 ISSN (p): 2319 – 1805.

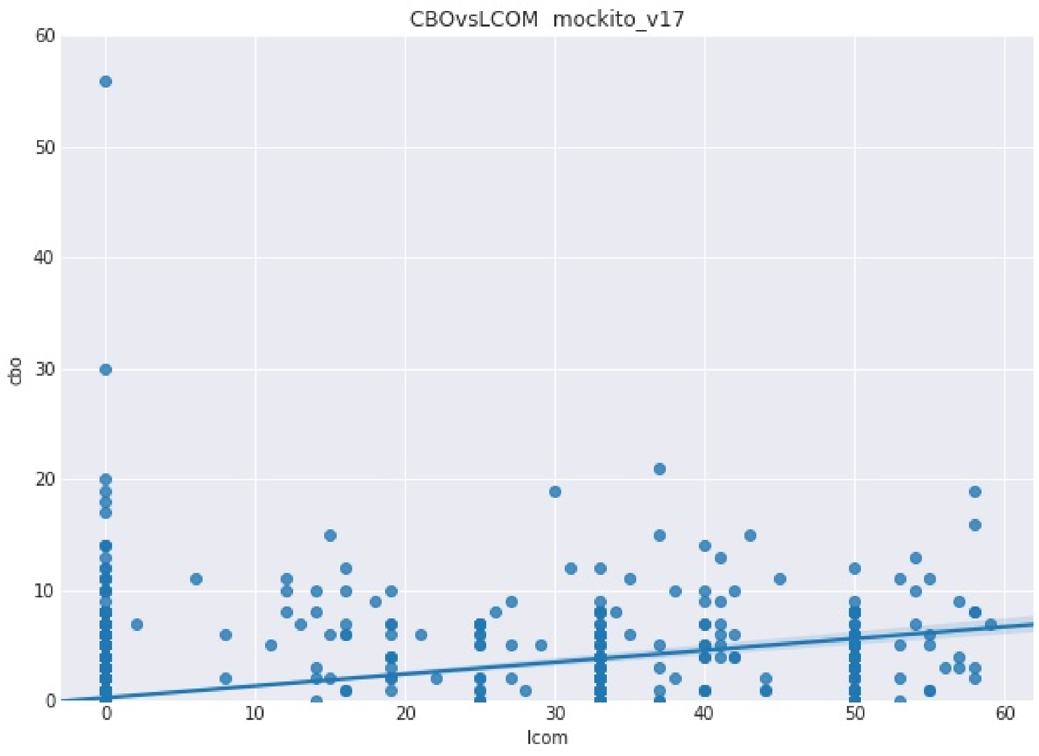
[17] "Refactoring for Software Design Smells: Managing Technical Debt"-

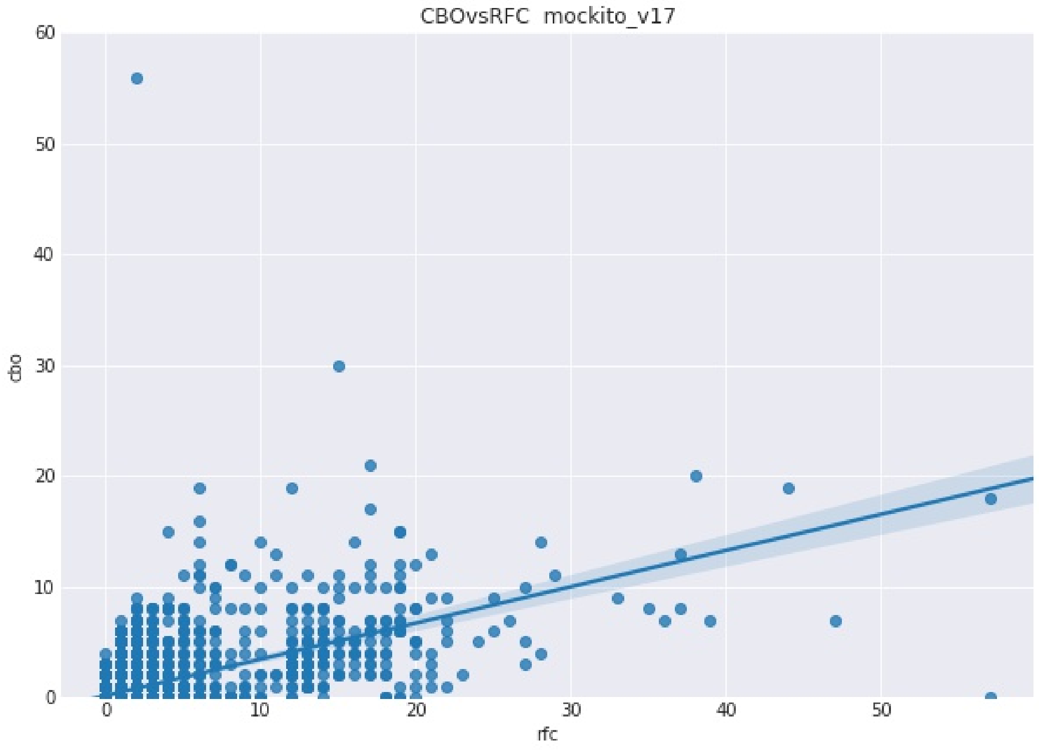
Ganesh Samarthyam, Girish Suryanarayana, and Tushar Sharma.

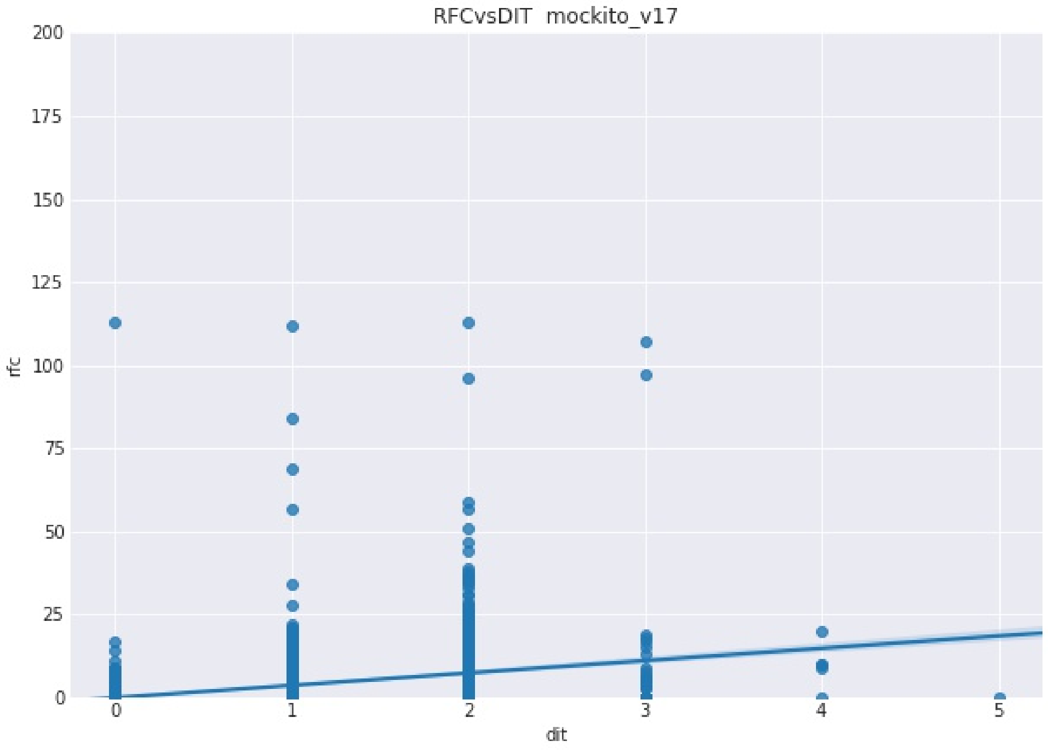
Git Repository:

https://github.com/HarshDivecha/SOEN661

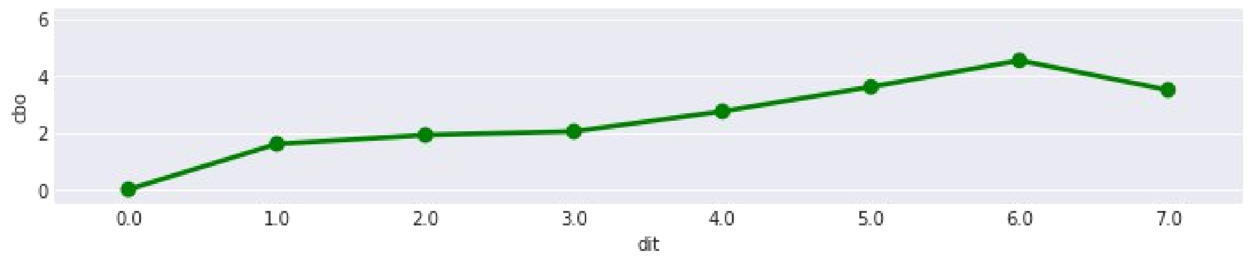

CBOvsDIT mockito_v17
cbo
dit
0
20
40
60
80
100
0
1
2
3
4
5


CBOvsLCOM mockito_v17
cbo
lcom
0
10
20
30
40
50
60
0
10
20
30
40
50
60

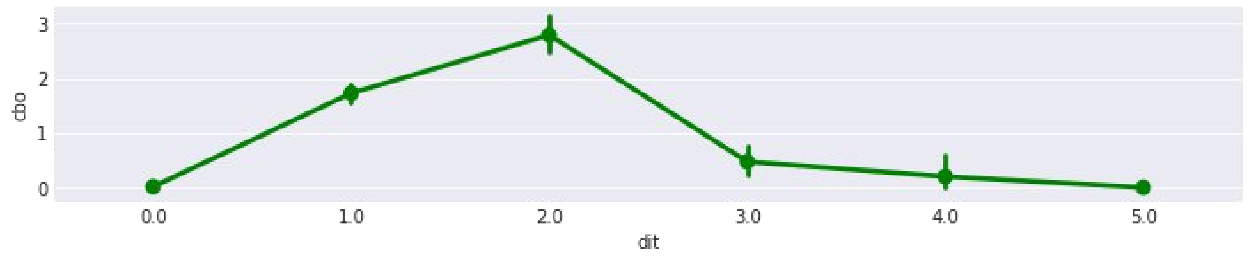

CBOvsRFC mockito_v17
cbo
rfc
0
10
20
30
40
50
60
0
10
20
30
40
50


RFCvsDIT mockito_v17
rfc
dit
0
25
50
75
100
125
150
175
200
0
1
2
3
4
5

We Observe different CBO vs DIT behaviour in Mockito and Wildfly:



Wildfly



Mockito