Iran University of Science and Technology
Deep Learning (Spring 2019)

# A Deep Learning Approach for Finding Bad Smells in Software Systems

**Raana Saheb Nassagh**
Department of Computer Engineering
Iran University of Science
and Technology
name@cs.iust.ac.ir

## Abstract

The creation of high quality software is of great importance in the current state of the enterprise systems and web based applications. High quality software should contain certain features including flexibility, maintainability and a well-designed structure. There are several approaches to write high quality software with such features. Correctly adhering to the object-oriented principles is one such approach to make the code more flexible. Developers usually try to leverage these principles, but many times neglecting them due to the lack of time and the extra costs involved. Therefore, sometimes they create confusing, complex, and problematic structures in code known as code smells. Often, these structures have specific and well-known patterns that can be corrected after their identification with the help of the refactoring techniques. This process can be performed either manually by the developers or automatically. In this project, we will introduce an automated method for identifying a series of code smells in the java programs. The mechanism will be used for performing such automated refactoring by leveraging a deep-learning method. In addition, we will also use a graph-model as the core representation scheme along with the corresponding measures such as betweenness, load, in-degree, out-degree and closeness centrality, as the features of the code smells in the programs. Finally, we will define the proposed method on Junit, Ganttproject and Mockito and validate the results with the IPlasma tool. The summary of the major contributions of the current research are as follows:

1. Finding a comprehensive mapping between complex graph properties and bad smells in the code structure.

2. Identifying bad smells using a deep-learning approach while leveraging the complex graph properties in order to maximize the identification coverage.

3. Introducing an automated model for the identification of code structure.

## 1 Introduction

In recent decades, software programs have played an important role not only in business and scientific domains but also in everyday life. Developing and maintaining high-quality software programs is therefore of crucial importance. A high-quality software is a software which is flexible [1], reusable, reliable and maintainable [2]. To create programs with the stated characteristics, programmers usually can stick to a set of pre-defined rules and principles namely, the object-oriented principle. Most of

the time, object-oriented principles such as single responsibility principle [3], low coupling and high cohesion [4] act as guidelines for the programmers to create better computer programs.

Unfortunately, the lack of time and the associated development costs often leads to technical debt [5] in the projects as well as disregard in the usage of design patterns and object-oriented principles. Therefore, poorly designed structures are introduced in the software's code structure. These poorly structured and complex codes are known as bad smells which are an indicator of deeper problems in the software architecture usually hinting to future flexibility and maintainability problems in the software [6]. Some of the code smells have specific structures and certain properties.

These properties are key for identifying code smells and then re-designing them with the help of the existing refactoring techniques. Refactoring can be performed manually by the developers themselves or automatically using the existing tools and techniques [7]. Although several tools have been implemented to identify and refactor bad smells, none of them satisfies all the requirements of the developers in terms of flexibility and maintainability. For this reason, a lot of research is performed in the previous years in the field of automatic refactoring to improve the existing techniques or introduce new ones. Usually, the aim is for the newly introduced approach to be more capable of identifying a larger set of code smells as well as leveraging more effective refactoring mechanisms in order to more comprehensively correct the poor or badly-designed structures in the code base.

A line of research in creating automated refactoring methods focuses on a combination of using graphical models [8], complex networks parameters [9] and clustering [10] for finding bad smells and measuring object-oriented properties. Calculation based on complex networks parameters creates a simple mapping between the code structure and graph representation. However other graphical methods like clustering methods require extensive computation time as well as complete information about the whole system.

Other methods exist that have applied neural networks [11] and machine learning techniques [12, 13, 14] for refactoring bad smells. Most of these methods focus on reaching automated ways for refactoring code smells without the need for domain experts. The main point of these researches is to find a general method for discovering and refactoring various categories of bad smells. Of course, there are also other approaches that have moved in a similar direction with the same aim of finding and refactoring various bad smells in a generalized way. For example, some of these methods used multi-objective refactoring [15, 16], to find the best combination of software metrics and to improve the code efficiency. Such works are major steps toward a general method for finding and refactoring a wide range of bad smells, but they still require a lot of work to be considered a complete automated refactoring method.

For the current research, we aim to introduce new features which map the system properties. To achieve this, we will use both the advantages of graphical models and complex network parameters which creates the mapping simplicity as well as the automated capabilities that artificial intelligence approaches usually bring to the table especially in the detection stages of the bad smells.

In this paper, we focus on identifying bad smells in Java codes. The main approach of the introduced algorithm is to use complex networks properties for finding bad smells. Network properties will be calculated using the extracted graph-model from the code structure. These properties include betweenness, load, closeness, in-degree and out-degree centrality. The rationale behind using these measures is the simple mapping that exists between network properties and well-known bad smells. In other words, a node with special properties like high in-degree centrality in the system can be probably mapped to a class with an attribute such as the level of reusability of a class. This attribute can be mapped to a bad smell such as the shotgun surgery or the lazy class [17, 18].

Different from other methods that doesn't map any features to the software systems, our aim is to find appropriate features and train a model which finds bad smells with the help of this graphical features, independent of the code style or system. The model will extract network features from the source code and create a validate set with the help of the IPlasma-Tool [19].

## 2 Related work/Background

The general idea of object-oriented programming is data abstraction [20], expressing type hierarchy [20], encapsulation [21] and inheritance [21]. Object-oriented principles and design patterns [22, 23] are guidelines that make the code more flexible, more reusable and maintainable. Design patterns

also create the ability to reuse code and improve maintainability problems [24]. Lack of time and development costs often leads to disregards in applying these patterns and principles. Therefore, poorly designed structures, known as bad smells, are introduced in the software code structure. Some of the code smells have specific structures, which can be identified and then re-designed with the help of refactoring techniques.

### 2.1 Introduction of bad smells

Feature envy: This smell refers to a class that uses data or methods of other classes more than its own fields and methods. This smell occurs usually when programmers encapsulate the data of a class as a separate class, without considering to move the respective methods to keep both data and operation in the same place. One of the easiest refactoring approaches to correct this smell is to move each method to its related data class. If a method uses data from more than one class, the method is moved to the class with the most related data. By refactoring this smell one can have an improved code structure having less code duplication and lower coupling [6, 16, 26].

Shotgun surgery: This smell refers to the situation where a single modification makes many other changes to several other code segments. This smell occurs most of the time when programmers ignore the single responsibility principle. In other words, shotgun surgery happens when a large number of external methods are creating a dependency with a class. One of the easiest refactoring approaches to correct this smell is to split the class to smaller classes or move methods and fields to other classes. By refactoring this smell, we can have less code duplication, fewer dependencies, more locality of change and hence better maintenance [6, 16, 25, 26].

Lazy class: A lazy class refers to a class that does not do enough to be maintained. This smell occurs when a class was designed for future use but never got a significant and practical usage. It also occurs when a class becomes too small due to other refactoring or modifications. One of the easiest refactoring approaches is to delete such lazy classes and move the existing code to the most related classes. By refactoring lazy classes, code size can be reduced and hence understandability will be improved [6, 16, 26].

God class: A god class is a huge class which tends to centralize the data or methods of the system. In other words, god classes also known as blobs monopolize the behavior of a system. One of the easiest ways to refactor such classes is to divide them into smaller ones, which have their own well-defined responsibilities. Removing god classes makes the code more flexible, less error-prone and more maintainable [6, 16, 27, 28].

### 2.2 Complex networks properties

The proposed method is based on complex network measures and properties. In this research, we have used different network centrality degrees as the main method for identifying bad smells. Here, we will briefly discuss the considered properties: Betweenness centrality: A way of calculating how much a node influences the flow of information in a graph. This measure can be used to find nodes that act as a bridge between different parts of a graph. Betweenness centrality of a node v is the sum of the fraction of all-pairs shortest paths that pass through v.

Load centrality: The load centrality of a node is the fraction of all the shortest paths that pass through the node. Load centrality is different than betweenness centrality. The load centrality defined by Goh et al. is a betweenness-like measure defined through a hypothetical flow process. [31] [30][32].

Closeness centrality: Closeness centrality of a node u is the reciprocal of the sum of the shortest path distances from u to all n-1 other nodes. Since the sum of distances depends on the number of nodes in the graph, closeness is normalized by the sum of the minimum possible distances n-1 [30] .

In-degree centrality: The in-degree centrality for a node v is the fraction of nodes its incoming edges are connected to [30].

Out-degree centrality: The out-degree centrality for a node v is the fraction of nodes its outgoing edges are connected to [30].

## 3 Proposed method

This study contains two main parts:

1. Creating the dataset with the help of network features and IPlasma

2. Create and train a suitable network

### 3.1 Dataset

In this project the train datasets will be the classes in JUnit (240 classes), Ganttproject (648 classes) and Mockito (1225 classes). All of them are known datasets in the bad smell field which are used a lot in different researches. All of the source codes are available in Github. For converting the Classes to input and output data of our network we need to go through these steps:

1. Converting classes of each code to a graph. For this step we need to convert the Code with the Visual Paradigm Tool to a UML-Diagram. After that we parse the xmi-format of the diagram and convert it to a graph by finding the relations and dependencies of the classes to each other.

2. After the graph of a code is constructed, we need to calculate the graph measures of each class. For this step we can use the networkx-library of python, which calculates the network measurements (betweenness, closeness, load, indegree and outdegree centrality).

3. The graph measurements must be normalized between 0 and 1 for having a normalized input layer.

4. At last we need to have the labels of the classes. In the first model, we just need to know if a class has bad smells or not. At the second model we need to know which specified bad smell each class can have. For both of the models we used the IPlasma-Tool to label the classes.

At last out dataset had 10 features. 8 of the features were coupling features and 2 of them cohesion features.

For the labels we had to options and used both f them in different models:

- Binary Classification: Has this class a bad smell or not - Multi-Label: Which bad smell has this class

Figure 1 shows an overview of the features and labels:

### 3.2 The deep structure

Following steps were done:

- First the simple dataset in Figure 2 was used. In this part the goal was to test the features. In this step the features were normalized and preprocessed.

  To avoid overtraining dropout and activity regulizer were used.

  Also the correlation of the features were calculated to see if the features were collected right. For the test also polynomial features were added. This simple model was just tested to see how the features were trained. In this step each source code was cross validated because of the small dataset of each source code.

- In the second step we used the model in Figure 3. Here we have multiply inputs. One f them is for coupling metrics (indegree, outdegree, ..) the other for cohesion (LCOM 1 and LCOM 2) one.

- In the last step we used the model in Figure 4. Here we have also multiply outputs. One f them is for the recognition of bad smells (true or false) and the kind of bad smells (god class, feature envy,..)

## 4 Results

Here we see the two different models with two different scenarios:

- We tested the first model with each source code as training set and another source code as testing set. Here we got good results that are described in Figure 5. This results show that

| Node Properties (Coupling) | Class Property (Cohesion) | Bad Smells |
|---|---|---|
| Degree Centrality | LCOM1 | God Class |
| Indegree Centrality | LCOM2 | Lazy Class |
| Outdegree Centrality | | Feature Envy |
| Closeness Centrality | | Shotgun Surgery |
| Load Centrality | | |
| Betweenness Centrality | | |
| Entropy | | |
| Critical | | |

Figure 1: Extracted Features and labels

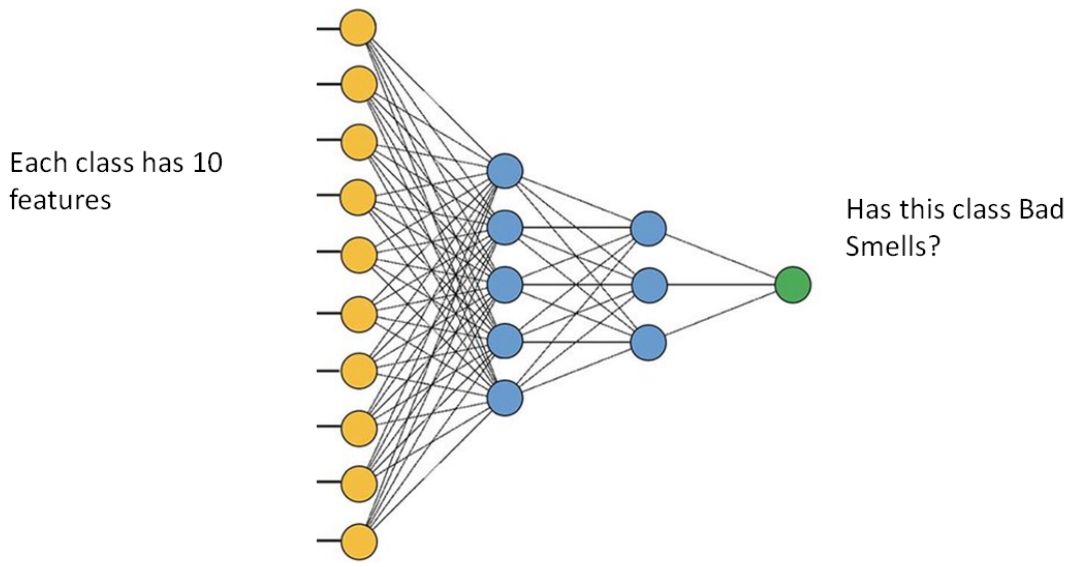


Figure 2: The first simple model

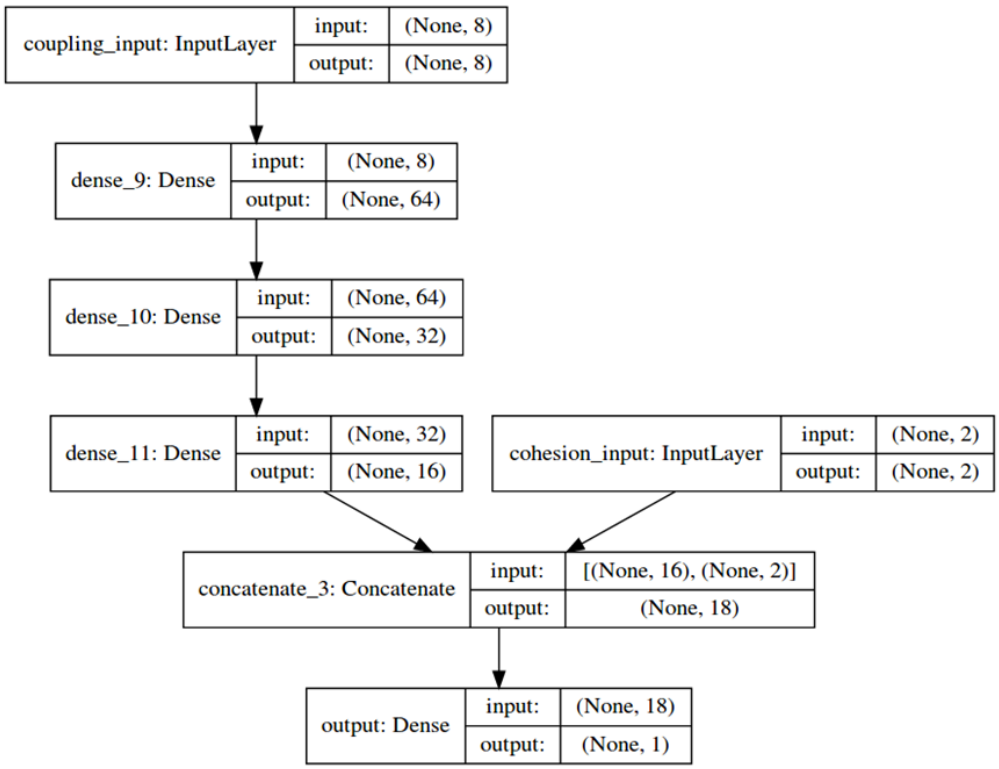


Figure 3: multiply inputs

are features are independent from the code style and source code which is a really important point.

- We tested the second model with 1500 classes from all of the source code as training set and 613 of them for testing set. Here we got good results in the output accuracy (is it a bad smell or not) and an acceptable accuracy in the bad smell output (which kind of bad smell it is). The results are shown in figure 6.

## 5 Discussion

Here are some benefits of out model:

1. We extracted features which we show are suitable features for the software codes
2. We trained a model which is independent of code style and source code
3. We created a flexible model. Here we just used 8 graphical features. In the future anyone can add more features.
4. We showed that the graph structure is a suitable mapping of the source code.

Here are some ideas for the futures:

1. Can the picture of subgraph also be mapped to the classes? If yes, we cold use a conv model to find bad smells.
2. Is the model also independent of the code language?

## References

[1] Alexander, J.A. & Mozer, M.C. (1995) Template-based algorithms for connectionist rule extraction. In G. Tesauro, D.S. Touretzky and T.K. Leen (eds.), *Advances in Neural Information Processing Systems 7*, pp. 609–616. Cambridge, MA: MIT Press.

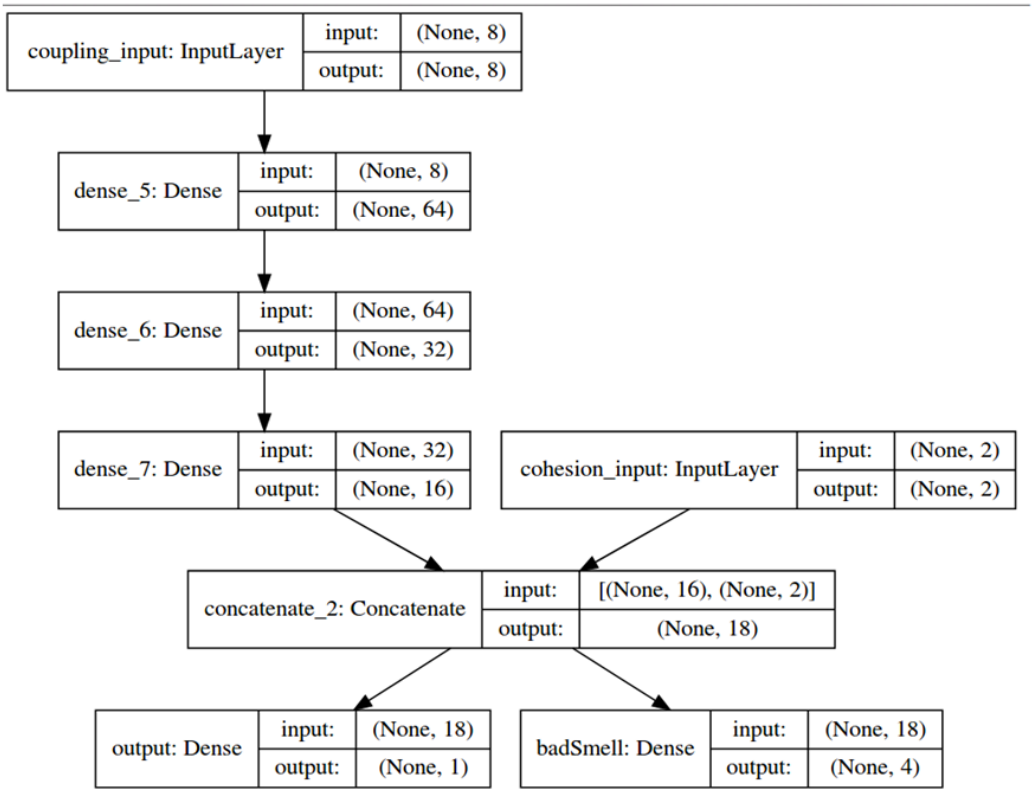


Figure 4: multiply outputs

| Train Set | Test Set | Accuracy (Recognizing Bad Smell) |
|---|---|---|
| Junit | Ganttproject | 0.786 |
| Junit | Mockito | 0.786 |
| Ganttproject | Junit | 0.834 |
| Ganttproject | Mockito | 0.786 |
| Mockito | Junit | 0.767 |
| Mockito | Ganttproject | 0.794 |
| ALL [1500] | ALL[613] | 0.827 |

Figure 5: Evaluation of model 1

| loss | output_loss (Is Bad) | badSmell_loss (Which Bad Smell) | output_acc (Is Bad) | badSmell_acc (Which BS) |
|---|---|---|---|---|
| 0.21 | 0.09 | 0.10 | 0.88 | 0.70 |

Figure 6: Evaluation of model 2


[1] H. Subramaniam, H. Zulzalil, Software quality assessment using flexibility: A systematic literature review, International Review on Computers and Software, vol.7, pp.2095-2099, 2012.

[2] A. Yamashita, "How good are code smells for evaluating software maintainability? Results from a comparative case study," in Proceedings of the 2013 IEEE International Conference on Software Maintenance, 2013, Netherland, Eindhoven, pp.566-571.

[3] D. Wampler, "Aspect-oriented design principles: Lessons from object-oriented design," in Proceedings of the 6th International Conference on Aspect-Oriented Software Development, March 2007, Vancouver, British Columbia, Canada.

[4] E. Johann, G. Kappel, and M. Schrefl, Coupling and cohesion in object-oriented systems, Technical Report, University of Klagenfurt, 1994.

[5] R. N. Charette, Why software fails [software failure], IEEE spectrum, vol.42, no.9, pp.42-49, 2005.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the design of existing code, MA, USA: Addison Wesley, 1999.

[7] S. Negara, et al. "A comparative study of manual and automated refactoring," in Proceedings of the European Conference on Object-Oriented Programming, Springer, Berlin, Heidelberg, 2013.

[8] B. Bafandeh Mayvan, A. Rasoolzadegan, Design pattern detection based on the graph theory, Knowledge-Based Systems, vol.120, no.3, pp.211-225, 2017.

[9] A. Gu, X. Zhou1, Z. Li1, Q. Li1 and L. Li, Measuring object-oriented class cohesion based on complex networks, Arabian Journal for Science and Engineering, vol.42, no.8, pp.3551-3561, 2017.

[10] H. Masoud, S. Jalili, A clustering-based model for class responsibility assignment problem in object-oriented analysis, The Journal of Systems and Software, vol.93, pp.110-131, 2014.

[11] K. Jaspreet; S. Satwinder, Neural network based refactoring area identification in software system with object-oriented metrics, Indian Journal of Science and Technology, vol.9, no.10, 2016.

[12] S. Kebir, I. Borne, D. Meslati, A genetic algorithm-based approach for automated refactoring of component-based software, Information and Software Technology, vol.88, no.3, pp.17-36, 2017.

[13] A. Boukhdhir, M. Kessentini, S. Bechikh, J. Dea, and L. Ben-Said. "On the use of machine learning and search-based software engineering for ill-defined fitness function: A case study on software refactoring," in Proceedings of the International Symposium on Search Based Software Engineering, pp. 31-45. Springer, Cham, 2014.

[14] F. Arcelli, M. Zanoni, A. Marino, and M.V. Mäntylä. "Code smell detection: Towards a machine learning-based approach,", in Proceedings of the 2013 IEEE International Conference on Software Maintenance, September 2013, Eindhoven, The Netherlands, pp. 396-399.

[15] U. Mansoor, M. Kessentini, M. Wimmer and K. Deb, Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm, Software Quality Journal, vol.25, no.2, pp.473-501, 2017.

[16] A. Ouni, M. Kessentini, M. Ó Cinnéide and H. Sahraoui, MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells, Journal of Software: Evolution and Process, vol.29, 2017.

[17] C.Y. Chong, S.P. Lee, Automatic Clustering constraints derivation from object-oriented software using the weighted complex network with graph theory analysis, The Journal of Systems Software, vol.133, 2017.

[18] S. Jenkins and S. Kirk, Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution, Information Sciences. vol.177, no.12, pp.2587-2601, 2007.

[19] C. Marinescu, R. Marinescu, P.F. Mihancea D. Ratiu, and R. Wettel, “iPlasma: An integrated platform for quality assessment of object-oriented design,” in Proceedings of the ICSM Conference, 2005, Budapest, Hungary pp.77-80.

[20] B. Stroustrup, What is object-oriented programming? IEEE Software, vol.5, no.3, pp.10-20, 1988.

[21] A. Snyder, “Encapsulation and inheritance in object-oriented programming languages”, in Proceedings on Object-oriented Programming Systems, Languages and Applications Conference, 1986, Portland, USA, pp.38-45.

[22] J. Dooley, Object-oriented design principles, Software Development and Professional Practice, Apress, 2011, pp. 115–136.

[23] T. Sharma, G. Samarthyam, and G. Suryanarayana, “Applying design principles in practice”, in Proceedings of the 8th India Software Conference, February 2015, Bangalore, India, pp.200-102.

[24] M. Huaxin and J. Shuai, “Design patterns in software development,” in Proceedings of the 2011 IEEE 2nd International Conference on Software Engineering and Service Science, July 2011, Beijing, China.

[25] S. Gurpreet and V. Chopra, A study of bad smells in code, International Journal for Science and Technologies with Latest Trends, vol.7, no.1, pp.16-20, 2013.

[26] “Refactoring”, URL: https://refactoring.guru/refactoring, Access Date: 7 February 2019.

[27] R. Wirfs-Brock, A. McKean, Object Design: Roles, responsibilities, and collaborations, Addison-Wesley Professional, 2003.

[28] A. J. Riel, Object-oriented design heuristics, Addison-Wesley Reading, Vol. 338, 1996.

[29] U. Brandes: On variants of shortest-path betweenness centrality and their generic computation, Social Networks, vol.30, no.2, pp.136-145, 2008.

[30] “Overview of NetworkX”, URL:https://networkx.github.io/documentation/, Access Date: 20 February 2019.

[31] K. Goh, B. Kahng and D. Kim, Universal behavior of load distribution in scale-free networks, Physical Review Letters, vol.87, no.27, pp.1–4, 2001.

[32] K.I. Goh, B. Kahng, and D. Kim, Universal behavior of load distribution in scale-free networks, Physical Review Letters, vol.87, no.27, 2001.