Application of Graph Theory to OO Software Engineering

Alexander Chatzigeorgiou Nikolaos Tsantalis George Stephanides
Department of Applied Informatics, University of Macedonia
156 Egnatia str., 54006 Thessaloniki, Greece
Tel: +30 2310891886
achat@uom.gr, nikos@java.uom.gr, steph@uom.gr

ABSTRACT
Graph Theory, which studies the properties of graphs, has been widely accepted as a core subject in the knowledge of computer scientists. So is Object-Oriented (OO) software engineering, which deals with the analysis, design and implementation of systems employing classes as modules. The latter field can greatly benefit from the application of Graph Theory, since the main mode of representation, namely the class diagram, is essentially a directed graph. The study of graph properties can be valuable in many ways for understanding the characteristics of the underlying software systems. Representative examples for the usefulness of graph theory on OO systems based on recent research results are presented in this paper.

Categories and Subject Descriptors
D.2.10 [**Software Engineering**]: Design – *Representation*, D.2.11 [**Software Engineering**]: Software Architectures – *Patterns*, G.2.2 [**Discrete Mathematics**]: Graph Theory – *Graph Algorithms*

General Terms
Algorithms, Design, Theory

Keywords
Graph Theory, "God" classes, clustering, design pattern detection, scale-free

1. INTRODUCTION
A graph is informally defined as a set of objects called vertices connected by links called edges [27]. Graphs are visually represented as dots connected by lines and this convenient form has made them appealing to computer scientists. The amount of technological, social and biological networks that can be represented as graphs is enormous, a fact that made graphs essential part of any Data Structures course in engineering and computer science schools.

Object-oriented systems aim at modeling a given problem as well as its solution as a set of interacting objects. Objects are instances of classes that define their state attributes and a (common) behavior. The static and dynamic aspects of the architecture of an object-oriented system are nowadays being represented by employing one or more diagrams of the Unified Modeling Language (UML) [22]. Among all diagrams, the most common representation is the class diagram depicting the classes, their methods and attributes and most importantly the relationships between them.

Class diagrams can be perfectly mapped to graphs where vertices represent the classes, while edges correspond to a selected type of relationship (e.g. association, generalization, composition, etc). In this paper, it will be shown that by exploiting theorems and algorithms from Graph Theory it is possible to extract important knowledge regarding the represented object-oriented system. The examples that will be presented will include a methodology for identifying "God" classes [3] [21], spectral graph partitioning algorithms for locating clusters of strongly communicating classes, an approach for detecting instances of Design Patterns as well as a proposal for evaluating the "scale-freeness" of an object-oriented system.

Graphs have long been used in several fields of computer science. To mention a few, we will give examples from several phases of the software development lifecycle. During Requirements Specification, Data Flow Diagrams (DFDs) are essentially graphs where vertices represent transformations and edges the data flows. Finite State Machines (FSMs) and Petri Nets have also been successful for capturing the requirements of synchronous and asynchronous systems due to the appealing graphics notation. During design, any sort of Graphical Design Notation (GDN) used for describing relations among modules is essentially a graph. In this broad set of techniques directed edges represent the dependency of one software component on another, which is essential information for the structure of procedural programs [12]. During testing, another example is the control flow of a program associated with the well known McCabe's complexity measure which employs directed graphs for addressing the sequence of executed instructions, the structures that they form and the upper bound of tests for ensuring coverage [7]. Even software process management has benefited from the use of network diagrams for calculating earliest start and latest finish dates (CPM and PERT techniques) [28].

The rest of the paper is organized as follows: In section 2 the representation of an OO system as graph is illustrated. In section 3 the use of Algebraic Graph Theory for identifying "God" classes is discussed. In section 4 the general technique of clustering using spectral graph partitioning is presented. Section 5 introduces the similarity scoring algorithm used for detecting design patterns. In section 6 a novel measure of the scale-freeness of an OO design and its evolution during various releases is discussed.

2. SYSTEM REPRESENTATION
For the following, we will assume that a directed graph $G = (V, E)$ will represent the class diagram of the object-oriented system under study. The set of vertices V corresponds to the

WISER'06, May 20, 2006, Shanghai, China.

classes of the system while the set of all edges E represents a selected kind of relationship between the classes. For example, if associations are to be represented, a directed edge $(p,q) \in E$ indicates an association between classes p and q with a direction from p to q. As a result, to fully represent a class diagram, one has to exploit a number of graphs, one for each kind of relationship.

For example let us consider the simple design of Figure 1. Each piece of information is represented as a separate graph – matrix (Figure 2).

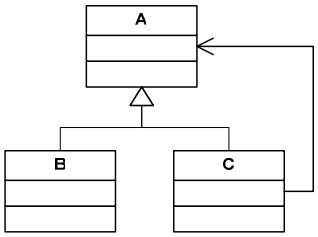


Figure 1: Sample class diagram

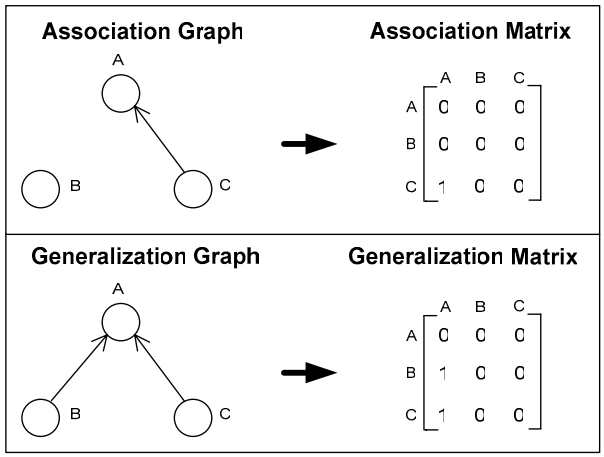


Figure 2: Representation of sample system as graphs/matrices

3. IDENTIFICATION OF "GOD" CLASSES

In this example which has been proposed in [3] the goal is to identify heavily loaded classes of a design, that is classes which play the role of a central manager or controller point or the role of a central data holder. Such classes imply a poorly designed model and have been successfully tagged as "God" classes by Riel [21]. "God" classes violate the principle of uniform distribution of responsibilities and are difficult to implement, test and maintain.

The proposed approach for identifying "God" classes is based on a modification of the well-known *HyperLink Induced Topic Search* (HITS) algorithm, developed by Kleinberg [15], for identifying pages on the World Wide Web that are "authoritative sources" on broad search topics. The key idea behind HITS is that the quality of a page p, referred to as the *authority* of the corresponding document, is not related only to the number of pages pointing to p, called *hubs*, but also to the quality of these hubs. Hubs and authorities exhibit what could be called a mutually reinforcing relationship.

HITS can be applied to OO systems by employing a single graph that represents classes and associations between them. However, each edge (p,q) is annotated with an integer $m_{p,q}$ corresponding to the number of discrete messages sent to the same direction from p to q. (Discrete messages are extracted with static analysis and refer to the number of possible method invocations with different signatures). If a class p sends many messages to classes with large a-values, it should receive a large h-value; if p receives messages from many classes with large h-values it should receive a large a-value. By modifying the approach in [15], this mutually reinforcing relationship motivates the definition of the following two operations:

Operation I: $$a_p = \sum_{q:(q,p)\in E} m_{q,p} h_q$$

Operation O: $$h_p = \sum_{q:(p,q)\in E} m_{p,q} a_q$$

which are graphically depicted in Figure 3.

The above equations can also be written as:

$$a = A^T h, \quad h = Aa$$

where A is the adjacency matrix of the graph G under study, and a, h are the vectors of the authority and hub weights respectively. The above system can be solved either iteratively or by employing the *Power Method* from Linear Algebra and *Perron's Theorem* [3]. Without outlining the details, the authority/hub weights of all classes can be obtained by finding the normalized principal eigenvector of $A^T A$ and AA^T (where the principal eigenvector is the one associated with the largest eigenvalue).

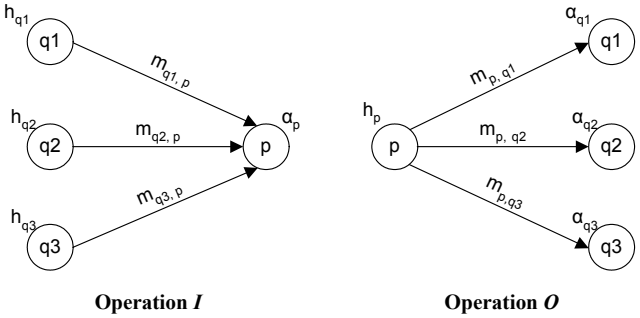


Figure 3: Definition of authorities and hubs

To demonstrate the application of the technique let us consider an example usually found in textbooks, namely a design modeling the operation of a microwave oven, shown in Figure 4. The Oven class is a "Manager"-like object capturing most of the system functionality. The corresponding graph and adjacency matrix are shown in Figure 5.

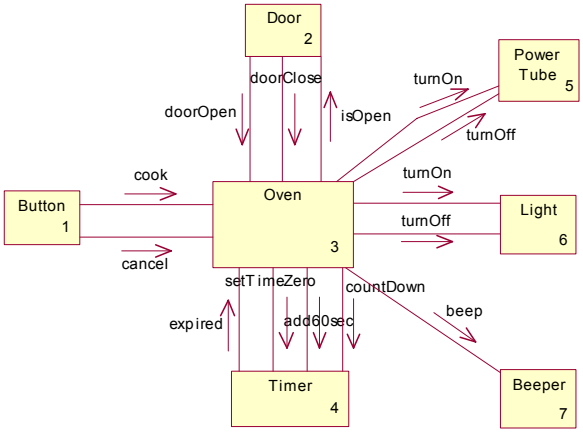


Figure 4: Microwave oven design with "God" class

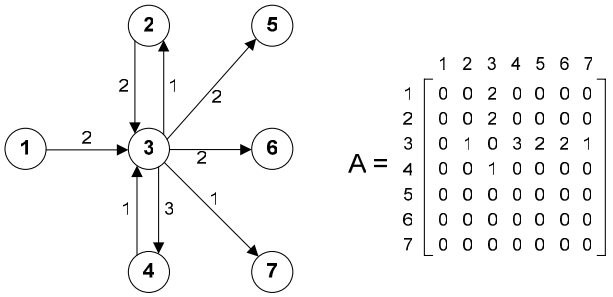


Figure 5: Corresponding graph and adjacency matrix for Figure 4

For this system, the authority and hub weights (calculated as the principal eigenvectors of matrices $A^T A$ and AA^T) are given by the vectors:

$$a_n{}^T = [0 \quad 0.229 \quad 0 \quad 0.688 \quad 0.459 \quad 0.459 \quad 0.229]$$

$$h_n{}^T = [0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0]$$

The central Oven class has an authority weight of zero, since classes sending messages to it, do not receive messages by any other class than the central. On the other hand, it has a hub weight of one, indicating clearly that this class initiates any activity in the system.

Consequently, the authority and hub weights for the classes of an OO system provide a natural way for quantifying their importance in the design, according to the responsibilities that they hold.

4. CLUSTERING

In general, software clustering aims at partitioning a software system into subsets of modules/components, so that the modules in each cluster share some common trait. A common criterion for partitioning, dating from the earliest days of Structured Analysis and Design, is to come up with clusters that exhibit high cohesion and low coupling. Spectral graph partitioning techniques first appeared in the early seventies in the research work of Donath and Hoffman [4] and Fiedler [8], [9]. They explored the properties of the algebraic representations of graphs (Adjacency Matrix, Laplacian Matrix) and introduced the idea of using eigenvectors to partition graphs.

In the OO systems domain, clustering can be viewed as the process of partitioning the system into sets of strongly communicating classes or hierarchies of classes. Such dense communities of classes exhibiting intense interaction in terms of method invocations, might imply relevance of functionality or even possible reusable components. Clustering can also be helpful in reducing the search space of algorithms that seek to identify patterns or specific structures within an OO system.

Given the $n \times n$ adjacency matrix A of an undirected graph G representing the class diagram of an object-oriented system containing n classes, the *degree* matrix of G is the $n \times n$ matrix $D = [d_{ij}]$ defined as:

$$d_{ij} = \begin{cases} \sum_{k=1}^{n} a_{ik} & ,if\ i = j \\ 0 & ,if\ i \neq j \end{cases}$$

The Laplacian matrix of G is the $n \times n$ symmetric matrix defined as $L = D - A$. It should be noted that the smallest eigenvalue of the Laplacian matrix is zero, with an associated eigenvector whose all entries are equal to one.

The properties of the eigenvector x_2 associated with the second smallest eigenvalue λ_2 have been explored by Fiedler [8], [9]. The eigenvector x_2 and its associated eigenvalue λ_2 are therefore known as the *Fiedler vector* and *Fiedler value*, respectively. The *Fiedler* value is related to a vast amount of valuable information concerning a graph, including connectivity, diameter, mean distance etc [17]. Here we will review the use of the Fiedler vector for providing an optimum partition for the graph *G*. By viewing the bisection of a graph as a discrete optimization problem [14] it has been proved that by clustering the vertices of a graph G in two sub-graphs according to the positive and negatives entries of the Fiedler vector, corresponds to a partition which minimizes the total weight of the edge cut between the two sub-graphs. This is the foundation of spectral methods.

Practically, by extracting the Laplacian matrix for the graph representing an OO system and its Fiedler vector, it is possible to obtain two sets of classes, which are well separated from each other (exchange the least number of messages) but are dense in the sense that they are strongly communicating. Since bipartitioning can be performed for undirected graphs, the adjacency matrix of the initial graph is constructed by summing the number of messages exchanged between a class pair in both directions and setting the sum as weight for the edge linking the two classes. Partitioning can be performed iteratively (by bipartitioning each resulting graph); However, an appropriate criterion should be used for stopping the partitioning process, otherwise we would come up with clusters containing a single class. Although many different criteria can be used, the one that has been experimentally found to provide accurate results is to stop the partitioning if a resulting graph is less cohesive than its parent graph. In other words, if the number of external edges (the number of edges connecting classes inside the cluster to classes residing in other clusters) exceeds the number of internal edges (the number of edges between classes inside the cluster).

To demonstrate this clustering technique let us consider a hypothetical object-oriented system consisting of a BusinessLogic part, a GUI part and a Database part, shown in Figure 6.

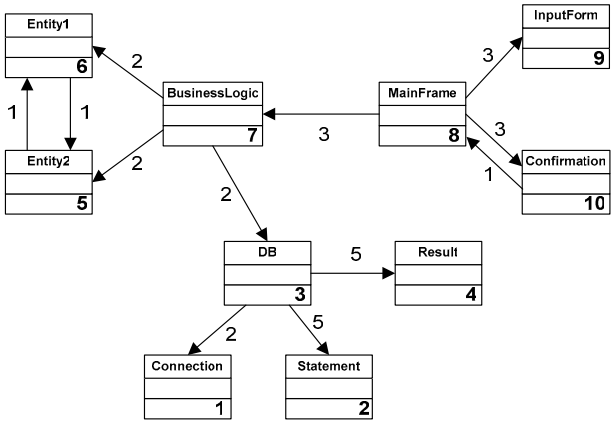


Figure 6: Object-oriented design with three discrete parts

The corresponding undirected graph results if classes are mapped to vertices and edges between them are weighted according to the total number of messages exchanged in both directions. The graph is shown in Figure 7.

The adjacency, degree and Laplacian matrices for this graph are shown below:

$$A=\begin{bmatrix}0&0&2&0&0&0&0&0&0&0\\0&0&5&0&0&0&0&0&0&0\\2&5&0&5&0&0&2&0&0&0\\0&0&5&0&0&0&0&0&0&0\\0&0&0&0&0&2&2&0&0&0\\0&0&0&0&2&0&2&0&0&0\\0&0&2&0&2&2&0&3&0&0\\0&0&0&0&0&0&3&0&3&4\\0&0&0&0&0&0&0&3&0&0\\0&0&0&0&0&0&0&4&0&0\end{bmatrix},\quad D=\begin{bmatrix}2&0&0&0&0&0&0&0&0&0\\0&5&0&0&0&0&0&0&0&0\\0&0&14&0&0&0&0&0&0&0\\0&0&0&5&0&0&0&0&0&0\\0&0&0&0&4&0&0&0&0&0\\0&0&0&0&0&4&0&0&0&0\\0&0&0&0&0&0&9&0&0&0\\0&0&0&0&0&0&0&10&0&0\\0&0&0&0&0&0&0&0&3&0\\0&0&0&0&0&0&0&0&0&4\end{bmatrix}$$

$$L=D-A$$

The Fiedler vector of the L matrix (calculated as the eigenvector corresponding to the second largest eigenvalue) is given by:

$$x_2^T=[-0.446,-0.359,-0.317,-0.359,0.152,0.152,0.108,0.313,0.388,0.366]$$

clearly differentiating the cluster of classes 1-4 (those related to the database) from the rest of the system. If spectral graph partitioning is performed in the same manner to the larger of the remaining clusters (classes 5-10), the Fiedler vector of the corresponding 6×6 Laplacian will differentiate the cluster of classes 5-7, as shown graphically in Figure 8.

Spectral graph partitioning techniques ensure optimum clustering; However, it should be considered that the calculation of eigenvectors is an extremely computationally intensive task which might lead to prohibitive times for large systems.

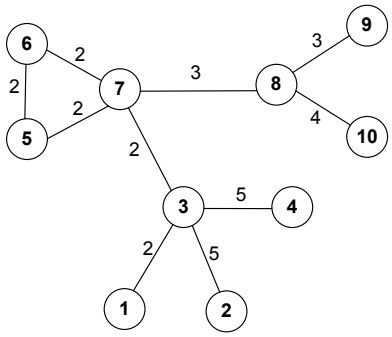


Figure 7: Corresponding undirected graph

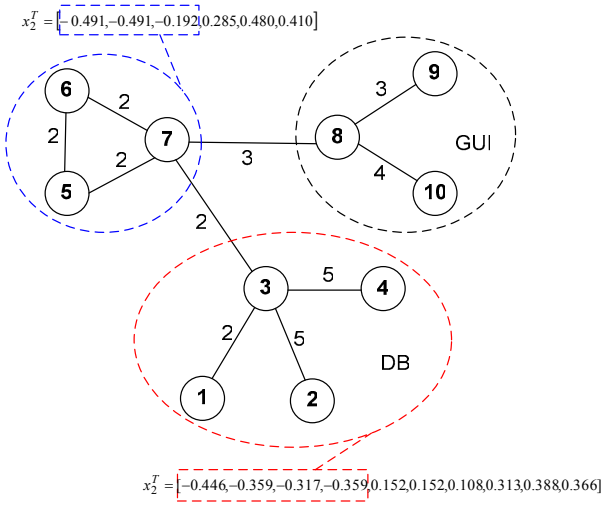


Figure 8: Clustering based on spectral graph partitioning

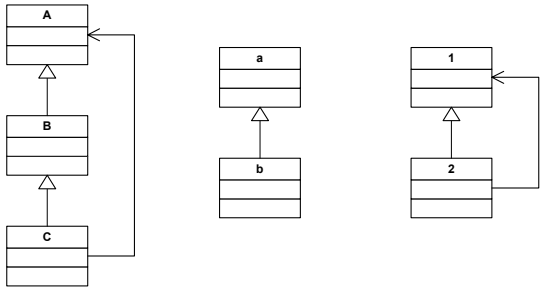
5. DESIGN PATTERN DETECTION

Design patterns are generally defined as descriptions of communicating classes that form a common solution to a common design problem. Since the publication of the most well-known catalogue of patterns [11], they have widely and rapidly attracted the interest of the software engineering community. Their proponents argue that their use leads to the construction of well-structured, maintainable and reusable software systems.

According to Parnas [19] software engineering deals with the development of *multi-version* projects. That is, most large-scale software projects result in evolving products. The evolving nature combined with the large number of components often leads to an architecture that is complicated and quite messy. Design patterns are known to impose structure to the system due to the abstractions being used. Consequently, the identification of implemented design patterns could be useful for the comprehension of an existing design and provides the ground for further improvements [13], [26]. This is evident in the ongoing effort of most CASE tools that attempt to embed design pattern detection techniques in the reengineering process.

Considering the representation of an OO system as a set of graphs, a pattern detection methodology can be formed based on graph matching algorithms. In such a methodology both the system under study as well as the design pattern to be detected are described in terms of graphs. However, conventional graph matching algorithms fail to detect patterns that differ from the standard representation usually found in textbooks. That is because, actual pattern instances, are often implemented as modified versions. Let us consider for example that the system under study in which patterns are sought, has two segments represented by the corresponding class diagrams of Figure 9. The pattern to be detected is also graphically depicted. This pattern is known as the *RedirectInFamily* elemental design pattern [24] which forms a part of the well known Decorator and Composite patterns. Obviously, the class diagram of segment 1 is a modified version of the design pattern, containing an additional inheritance level. On the other hand, segment 2 does not form a pattern since it only consists of a simple hierarchy of classes.

However, exact graph matching algorithms will not detect any match while inexact matching algorithms will erroneously detect that segment 2 is closer to the pattern.



System segment 1 **System segment 2** **Elem. Des. Pattern**

Figure 9: Class diagrams of two system segments and a pattern

Exploiting recent research on graph similarity, an algorithm proposed by Blondel et al. [2] which is based on Kleinberg's link analysis [15] can be used for the detection of patterns. The algorithm generalizes the concepts of authority and hub and iteratively calculates the similarity between the vertices of two different graphs. Let G_A and G_B be two directed graphs with,

respectively, n_A and n_B vertices. The **similarity matrix** $\boldsymbol{S}$ is defined as a $n_B \times n_A$ matrix whose real entry s_{ij} expresses how similar vertex j (in G_A) is to vertex i (in G_B) and is called the **similarity score** between the two vertices. The algorithm used for calculating the similarity matrix S is shown below:

1. Set $Z_0 = 1$
2. Iterate an even number of times

$$Z_{k+1} = \frac{BZ_kA^T + B^TZ_kA}{\left\| BZ_kA^T + B^TZ_kA \right\|_1}$$

and stop upon convergence

3. Output $\boldsymbol{S}$ is the last value of Z_k

where:

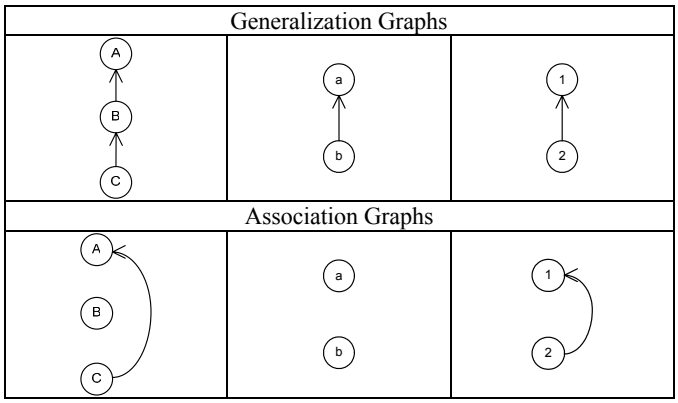
A, B are the adjacency matrices of graphs G_A and G_B, respectively,

Z_0 is a $n_B \times n_A$ matrix filled with ones

$\|.\|_1$ is the 1-norm of a matrix

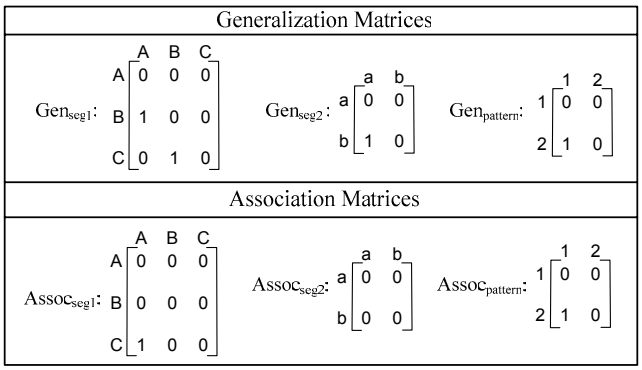
For the application to pattern detection, the algorithm takes as input the graph that represents the system under study as well as the graph describing the pattern of interest. Next, it calculates similarity scores between their vertices. The major advantage of this approach is the ability to detect not only patterns in their basic form but also modified versions of them.

Consider for example the graphs describing the relationships of the class diagram shown in Figure 9. The system segments and the pattern are fully specified by a generalization and an association graph as shown in Figure 10. The corresponding adjacency matrices are shown in Figure 11.



System segment 1 **System segment 2** **Elem. Des. Pattern**

Figure 10: Corresponding graphs for Figure 9



System segment 1 **System segment 2** **Elem. Des. Pattern**

Figure 11: Corresponding adjacency matrices for Figure 9

The similarity matrices between the corresponding graphs of segment 2 and the pattern are:

$$Gen_{pattern,\,seg2} = Similarity(Gen_{pattern}, Gen_{seg2}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$Assoc_{pattern,\,seg2} = Similarity(Assoc_{pattern}, Assoc_{seg2}) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

The sum of the two matrices is:

$$Sum_{pattern,\,seg2} = Gen_{pattern,\,seg2} + Assoc_{pattern,\,seg2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

To preserve the validity of the results any similarity score must be bounded within the range [0, 1]. Therefore, the resulting matrix is normalized by dividing the elements of column i (corresponding to similarity scores between all system classes and pattern role i) by the number of matrices (k_i) in which the given role is involved. This is equivalent to applying an affine transformation in which the resulting matrix is multiplied by a square $n_A \times n_A$ diagonal matrix, where element (i, i) is equal to $1/k_i$.

Consequently, the normalized scores that will eventually highlight similar nodes are calculated as:

$$NormScores_{pattern,\,seg2} = Sum_{pattern,\,seg2} \cdot \begin{bmatrix} 1/k_1 & 0 \\ 0 & 1/k_2 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix} = \begin{matrix} \\ a \\ b \end{matrix} \begin{matrix} 1 & 2 \\ \end{matrix} \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

where k_1 and k_2 correspond to the number of matrices in which pattern roles 1 and 2 are involved, respectively. (In this case both roles are involved in the association and the generalization matrix).

On the other hand, the similarity matrices between the corresponding graphs of segment 1 and the pattern are:

$$Gen_{pattern,\,seg1} = Similarity(Gen_{pattern}, Gen_{seg1}) = \begin{bmatrix} 0.5 & 0 \\ 0.5 & 0.5 \\ 0 & 0.5 \end{bmatrix}$$

$$Assoc_{pattern,\,seg1} = Similarity(Assoc_{pattern}, Assoc_{seg1}) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$NormScores_{pattern,\,seg1} =$$

$$\left(Gen_{pattern,\,seg1} + Assoc_{pattern,\,seg1}\right) \begin{bmatrix} 1/k_1 & 0 \\ 0 & 1/k_2 \end{bmatrix} = \begin{matrix} \\ A \\ B \\ C \end{matrix} \begin{matrix} 1 & 2 \\ \end{matrix} \begin{bmatrix} 0.75 & 0 \\ 0.25 & 0.25 \\ 0 & 0.75 \end{bmatrix}$$

The two larger entries in the last matrix indicate the strong similarity between classes (A, 1) and (C, 2) of the corresponding UML diagrams for system segment 1 and the pattern, shown in Figure 9. In contrast to the results from the inexact matching algorithm which indicates that the pattern is much closer to the structure of segment 2, the similarity algorithm correctly identifies the pattern being implemented in the structure of segment 1. The $NormScores_{pattern,seg2}$ similarity matrix also indicates similarity between classes (a, 1) and (b, 2), which is reasonable since the generalization matrices of segment 2 and the pattern in Figure 9 are the same, but the strength of similarity is lower due to the difference of their association matrices.

Consequently, similarity scoring can be used for identifying design pattern instances in an OO design, even if the implementation of the pattern is modified or refactored [10].

6. SCALE-FREENESS OF OO SYSTEMS

Recently, the investigation of *scale-free* graphs has become one of the most popular topics in a wide range of scientific domains involving networks. Examples of such graphs range from the topology of the Internet to biological and even social networks. Initiated by the work of Barabasi et al. [1] and Faloutsos et. al [6] an extensive study of whether several network properties (such as the in and out degree of nodes) follow power-laws has been performed. Particularly the Internet seems to display quite a number of power-law distributions such as the number of visits to a site, the number of pages within a site and the number of links to a page. At a next level, research has focused on the mechanisms that generate scale-free networks, such as preferential attachment and random growth. As a logical consequence, it has also been investigated whether scale-free phenomena exist in purely technological networks such as class collaboration graphs [18] and object graphs [20], in OO systems.

According to most studies, including those referring to OO systems, a scale free phenomenon shows up statistically in the form of power law. Mathematically, a distribution $P(k)$ obeys a *power law* or *scaling relationship* if $P(k) \sim k^{-\gamma}$, where k is the scaling index. Power laws are usually graphically detected, since the relationship of $P(k)$ versus k, plotted on a log-log scale appears as a line of slope $-k$. For example, in the diagram of Figure 12 we have plotted the cumulative frequency of associations between classes for three open-source software systems, namely JUnit, JHotDraw and JRefactory. (The term association is used in a general sense to indicate a relationship between two classes that exchange at least one message). It appears that this measure follows a power-law and according to most researchers this single evidence would imply that the network, regarding the association degree is scale-free.

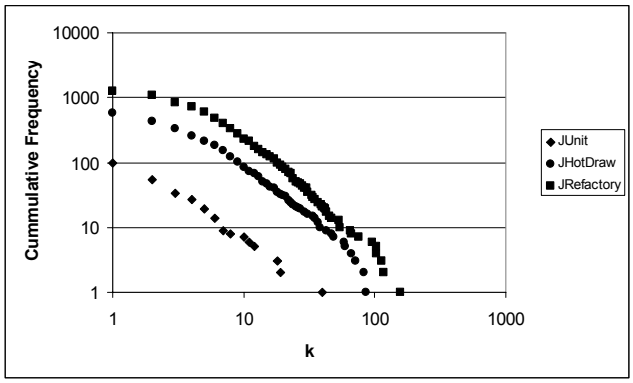


Figure 12: Power law for association degree distributions in three open-source OO systems.

Concerning the most widely acknowledged theory for scale-free networks, most studies agree in that scale-free networks:

- have power law degree distributions
- can be generated by certain random processes
- have highly connected "hubs" (nodes with large degree)

However, in a recent study [16] it has been pointed out that simply the presence of a power law degree distribution (which is the criterion used by most previous studies to claim that a network is scale-free) is not sufficient to imply the other properties. More importantly, a structural metric has been proposed whose value can be used to evaluate the ***scale-freeness*** of a network. For a graph g having fixed degree sequence $D = \{d_1, d_2, \ldots, d_n\}$ the metric is defined as:

$$s(g) = \sum_{(i,j)\in\varepsilon} d_i d_j$$

where (i, j) denotes a graph edge belonging to the edge set ε. The metric value is maximized when high-degree nodes ("hubs") are connected to other high-degree nodes.

Among all (undirected and connected) graphs having the same degree sequence, there is a graph s_{max} that maximizes the value of the metric $s(g)$ and a graph s_{min} that minimizes it. Thus, in order to obtain a normalized value for the scale-freeness of a given graph g one has to find s_{max} and s_{min} and calculate:

$$s = \frac{s(g) - s_{\min}}{s_{\max} - s_{\min}}$$

It is important to mention, that using this definition, most graphs satisfying a given degree sequence are scale-free since they have large s values. According to [16] extreme diversity is exhibited by graphs having small s values which are called scale-rich and are rare.

Given such a metric, it is not only possible to validate whether a given object-oriented system (represent as graph) is scale-free, but also to evaluate the evolution of the system as successive generations of the software are released. We have extracted the scale-freeness of the three open source systems (The calculation of the s_{max} graph can be performed according to a heuristic provided in [16]. For the calculation of the s_{min} graph we have developed a novel heuristic that is based on the idea of connecting the highest degree nodes to the lowest degree nodes and at each point it is checked whether the "remaining" degree sequence is realizable according to the Erdös and Gallai equation [5]). The results are shown in Figure 13.

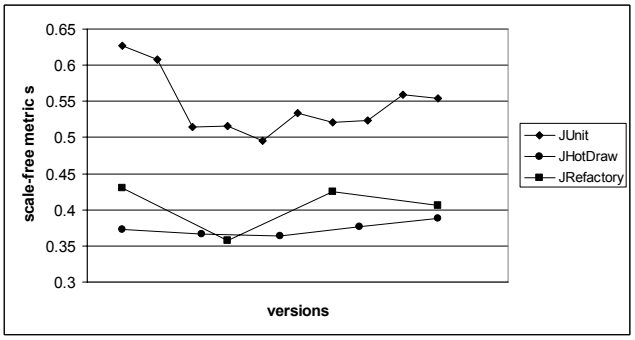


Figure 13: Evolution of scale-freeness for multiple versions of three open-source systems

As it can be observed, in contrast to previous studies, the extent by which an object-oriented system is considered scale-free, changes as the system evolves. For the particular systems, even if the log-log plot of the cumulative degree frequency indicated that all three systems are scale-free, their scale-freeness according to the s metric is quite different. It remains an open question whether any conclusions can be drawn for OO systems having multiple versions regarding the tendency to become more or less scale-free by the time. Even more useful if such an allegation is verified, would be to obtain insight to the processes that reduce/increase the scale-

freeness of an OO system and in what ways this information is related to the architecture of the system. Future research could investigate the relationship of scale-freeness to other structural metrics such as cohesion and coupling, use of polymorphism, inheritance, etc.

7. CONCLUSIONS

The study of graphs and their properties is a classical subject in most computer science departments around the world. Graph Theory can be further exploited by object-oriented software engineering, taking advantage of recent research results in various fields. In this paper four different applications of Graph Theory have been demonstrated, concerning: the identification of "God" classes, clustering, detection of design patterns and scale-freeness of OO systems. Since the architecture of an OO system can be naturally represented as one or more graphs, it is believed that research on the application of Graph Theory on such systems will be fruitful and that CS studies should strengthen the role of graphs in their curriculum.

8. REFERENCES

[1] Barabasi, A.L., Albert, R., Jeong, H., and Bianconi, G. Power-law distribution of the World Wide Web. *Science*, 287, (2000), 2115b.

[2] Blondel, V. D., Gajardo, A., Heymans, M., Senellart, P. and Van Dooren, P. A Measure of Similarity between Graph Vertices: Applications to Synonym Extraction and Web Searching. *SIAM Review, 46,* 4 (2004), 647-666.

[3] Chatzigeorgiou, A., Xanthos, S., and Stephanides, G. Evaluating Object-Oriented Designs with Link Analysis. In *Proceedings of the 26th International Conference on Software Engineering* (*ICSE'2004*), Edinburgh, Scotland, May 23-28, 2004

[4] Donath, W. E., and Hoffman, A. J. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development, 17,* (Sep. 1973), 420-425.

[5] Erdös, P., and Gallai, T. Graphs with prescribed degrees of vertices. *Mat. Lapok* (Hungarian), 11, (1960), 264-274.

[6] Faloutsos, M., Faloutsos, P., and Faloutsos, C. On power-law relationships of the internet topology. *Computer Communication Review*, 29, (1999), 251-262

[7] Fenton, N. E., and Pfleeger, S. L. *Software Metrics: A Rigorous & Practical Approach.* International Thompson Publishing, Boston, MA, 1997.

[8] Fiedler, M. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal, 23*(*98*)*,* (1973), 298-305.

[9] Fiedler, M. A property of eigenvectors of non-negative symmetric matrices and its applications to graph theory. *Czechoslovak Mathematical Journal, 25*(*100*)*,* (1975), 619-633.

[10] Fowler, M. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, Boston, MA, 1999.

[11] Gamma, E. Helm, R. Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Boston, MA, 1995.

[12] Ghezzi, C., Jazayeri, M., and Mandrioli, D. *Fundamentals of Software Engineering*. 2nd edn., Prentice Hall, Upper Saddle River, NJ, 2003.

[13] Guéhéneuc, Y. G., Sahraoui, H., and Zaidi, F. Fingerprinting Design Patterns. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE '04),* Delft, Netherlands, November, 2004.

[14] Holzrichter, M., and Oliveira, S. A Graph Based Method for Generating the Fiedler Vector of Irregular Problems. *Lecture Notes in Computer Science*, *1586*, (1999), 978-985.

[15] Kleinberg, J. M. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM, 46,* 5 (Sep. 1999), 604-632.

[16] Li, L., Anderson, D., Tanaka, R., Doyle, J. C., and Willinger, W. *Towards a Theory of Scale-Free Graphs: Definition, Properties and Implications*. Technical Report CIT-CDS-04-006, California Institute of Technology, Pasadena, CA, October 2005.

[17] Mohar, B. Some applications of Laplace eigenvalues of graphs. Graph Symmetry: Algebraic Methods and Applications, *NATO ASI Series C*, 497, (1997), 227-275.

[18] Myers, C. R. Software systems as complex networks: Structure, function and evolvability of software collaboration graphs. *Physical Review E*, 68, 046116, (2003).

[19] Parnas, D.L. Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs, Programming Methodology. In *Proceedings of the 4th Informatik Symposium*, September 1974, 225-235.

[20] Potanin, A., Noble, J., Frean, M., and Biddle, R., Scale-Free Geometry in OO Programs. *Communications of the ACM*, 48, 5 (May 2005), 99-103.

[21] Riel, A. J. *Object-Oriented Design Heuristics*. Addison-Wesley, Boston, MA, 1996.

[22] Rumbaugh, J., Jacobson, I., and Booch, G. *Unified Modeling Language Reference Manual*. 2nd edn., Addison-Wesley, Boston, MA, 2004.

[23] Shokoufandeh, A., Mancoridis, S. and Maycock, M. Applying Spectral Methods to Software Clustering. In *Proceedings of the Working Conference on Reverse Engineering* (WCRE'2002), Virginia, USA, October, 2002.

[24] Smith, J. M. *An Elemental Design Pattern Catalog.* Technical Report TR-02-040, Department of Computer Science, University of North Carolina, 2002.

[25] Smith, J. M., and Stotts, D. SPQR: Flexible Automated Design Pattern Extraction from Source Code. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE 2003),* Montreal, Canada, October, 2003.

[26] Vokač, M. An efficient tool for recovering Design Patterns from C++ Code. *Journal of Object Technology*, 2, 2 (July/August 2005).

[27] West, D. B. *Introduction to Graph Theory*. 2nd Edn., Prentice Hall, Upper Saddle River, NJ, 2000.

[28] Wild, R. *Productions and Operations Management*. 5th edn., Cassell Educational Ltd, London, UK, 1995.