THE UTILITY OF GRAPH THEORETIC SOFTWARE METRICS: A CASE STUDY

Aleksander B. Demko, Nicolino J. Pizzi
Institute for Biodiagnostics, National Research Council
435 Ellice Avenue, Winnipeg MB, R3B 1Y6
pizzi@nrc.ca

ABSTRACT

The adoption of and adherence to object-oriented design and programming principles have allowed the software industry to create applications of ever-increasing complexity. A concomitant need arises for strategies to identify, manage, and, wherever possible, reduce this software complexity. One such strategy is the systematic collection, interpretation, and analysis of software metrics, mappings from software objects or constructs to sets of numerical features that quantify relevant software attributes.

We describe a novel approach that employs various graph theoretic algorithms to analyze the higher level, application-wide class relationship graphs that emerge from object-oriented software. In addition to the software's overall inheritance tree characteristics, these algorithms will use metrics that reflect information on the import and export coupling of class-attribute and class-method relationships. Further, we incorporate information relating to the response sets for each object in the software, that is, the number of methods that can be executed in response to messages being received by objects.

Keywords: *Software Engineering, Graph Theory, Software Metrics.*

1. INTRODUCTION

The software industry's adoption of object-oriented design and programming principles has permitted the creation of applications of increasing complexity and size. Managers, programmers and software architects appreciate tools that may aid in the identification, minimization or correction of this complexity. Completely automated tools - ones that glean all relevant features from the source code itself, without programmer input - could be considered "holy grails." That is, they never require additional information from the programmer - information that if not updated, becomes out of sync from the code base. In this regard, one area of research uses *software metrics* to identify and categorize potentially complex areas [3] of applications. Armed with this information, managers hope to assign programmers to these areas to remove, or at least reduce, the unnecessarily complex parts of their applications.

Many existing software metrics focus on complexity at the object level. Metrics such as *lines of code*, *number of methods* and *comments to code* reflect the internal complexity of an object. Metrics such as *number of children* [1], *number of parents*, *number of siblings*, and *response set for an object* begin to reflect how objects relate to the software system as a whole.

We extend this notion by looking only at inter-object relationships. Two graph theoretic flooding algorithm will be presented, compared, and examined. The algorithms aid software developers by automatically locating key sub-systems within software projects. This subsystem breakdown provides a logical view of the application as a whole.

Finally, all metrics presented here are gathered and computed directly from source code, without any additional developer input. As a result, the metrics are perfectly synchronized with a changing code base. The metrics may be effortlessly applied to an already existing project. The computed observations have a multitude of uses, including: providing managers/lead programmers with overall design input and providing new developers with additional information about the code base.

2. METHODOLOGY

The metric acquisition and computation modules were written in C++ as modules in Scopira [2]. As a module algorithm development framework, metrics may be added or modified by simply writing additional Scopira modules while still reusing the existing infrastructure.

2.1. Data Acquisition

Raw C++ source code is processed through GCC-XML, a “virtual” output target for the GNU C++ compiler. The GNU C++ front end tokenizes and parses the source files as usual, but instead of writing out an executable, the GCC-XML back end takes the parsed information and stores it in a matching XML file. By utilizing a modern C++ compiler, we insure that all information is captured.

We process all the XML files produced by GCC-XML to build the *application graph*. In this graph, a node is a concrete class (i.e. templates are not included, but their instantiations are).

Between the nodes, we have three (six if direction is taken into account) types of edges: *parent of/child of*, *contains a (as a member object)/is contained by* and *knows about/is known by*.

Finally, we filter out all classes that do not belong to our application, by pruning the nodes in the graph that do not belong to the application’s name space(s). This weeds out system objects, which are referenced by the application, but are of no interest to our analysis.

2.2. Finding Key Classes

The first stage of analysis involves *key* classes. A key class is defined as an interface (a class that specifies only an interface, no body), a base class (a class that is the foundation of many other classes) or any class in between.

A straightforward, yet effective criteria was used to find key classes; if a class had three or more immediate decendant classes, it was deemed key.

For each key class, a supplementary metric *interface-rate* was computed. Interface-rate is the number of immediate descendant classes divided over the total (limited to a distance of three) descendant classes. Higher values indicate a greater likelihood that this key class is an interface, and not an implementation-providing base class. This metric attempts to capture the essence that interface classes tend to have *wide*, but shallow hierarchies, compared to base classes that often have *fuller* hierarchies.

2.3. Area of Effect

For each key class, we will use a graph theoretic flood fill algorithm to determine its *area of effect*. The area of effect is the (estimated) set of all classes that are dependent upon the key class. This dependence is stronger for classes that were found to be “closer” to the key class during the flooding.

Two flooding algorithms were developed and compared: Descendant Flooding (DF) and Variable Flooding (VF).

2.3.1. Descendant Flooding

DF only uses the descendant edges of the application graph. This reflects the natural tendency for developers to organize sub-systems by base classes. The algorithm is as follows:

For each key class K and some node N, initialize $v(K, N) = 0$. This is the current dependence value for node N to key class K.

Execute $flood(K, 10)$, where $flood(N, f)$, for node N and flood value f is defined (recursively) as: “If $f = 0$, stop. If $v(K, N) > f$, stop. Otherwise, let $v(K, N) = f$ and for each child C of N, execute $flood(C, f - 1)$”.

A node N belongs to key class K’s area of effect if, and only if, $v(K, N) > 0$.

In summary, this flooding technique simply finds all the classes that are 10 or less edges away from the key. The v matrix represents the relative proximity to each key class.

The initial value of 10 was chosen – partially arbitrarily – to represent the weak effect (and often loose coupling) base classes have over classes that are 10 or more nodes apart.

2.3.2. Variable Flooding

VF utilizes all the edge types in the application graph. By including the *contains-a* and *knows-about* relationships, this algorithm captures the dependencies that form from the coupling and use of objects by other objects. The algorithm is similar to DF:

For each key class K and some node N, initialize $v(K, N) = 0$. This is the current dependence value for node N to key class K.

Now, execute $flood(K, 10)$, where $flood(N, f)$, for node N and flood value f is defined (recursively) as: “If $f = 0$, stop. If $v(K, N) > f$, stop. Otherwise, let $v(K, N) = f$. For each connection type t, and each neighbor C of N (and type t), execute $flood(C, f - cost(t))$”. Where $cost(t)$ is defined below.

The cost function $cost(t)$ for a connection type t is the cost (dependency-wise) to flood that particular connection type. This represents the relative ease (low cost) or resistance (high cost) over which dependency constraints flow over the various connection edges. The function is defined as:

- $cost(descendants) = 1$. This represents the direct dependence descendant classes have to their parent classes.
- $cost(is - contained - by) = 2$. Container classes are dependent on their member classes, but not quite as dependent as they are to their parent classes.
- $cost(known - by) = 3$. Users are highly dependent on classes they utilize. However, this relationship is weaker than containment and inheritance.
- $cost(parents) = 6$. Descendant classes have very limited (if any) direct influence over parent classes. Any influence they do have would purely be application domain specific (for example, if a parent needs to be adjusted to accommodate additional descendant types).
- $cost(contains - a) = 6$. Container classes have limited influence over the items they contain.
- $cost(knows - about) = 6$. Users have little (direct) influence over the classes they use.

A node N belongs to key class K's area of effect if, and only if $v(K, N) > 0$.

2.4. Grouping

Regardless of the flooding algorithms chosen, a matrix v is produced, such that $v(K, N)$ is the relative influence key class K has over class N.

Because N may be influenced ($v(K, N) > 0$) for many K, we will say N is part of group (class set) K_g, if and only if, K has the highest influence of all other keys classes for that particular N.

Classes that have zero influence ratings for all key class are discarded.

Each class set K_g is said to be a sub-system or group or related classes.

2.5. Ordered Groups

We then sort all class clusters K_g by the size of their set. The size is a simple measure that effectively relates the visibly and effect that particular key class K has over its members.

Visualization is performing by showing the developer this sorted list of clusters and providing and interface to drill down and inspect individual clusters.

It is with these clusters of classes that the developer or manager may verify and inspect clusters of classes. Anomalies and unintended clusters or groupings are quickly noticed.

2.6. Results

To test out the accuracy of the sub-system detection/key class grouping, these algorithms were applied to the Scopira source code base and then examined by its lead developer for accuracy and usefulness.

Scopira contains about 95,000 lines of C++ source code. 49,000 lines encompass the Scopira core: scheduling engine, fundamental data types and graphical front end. About 46,000 lines of source code make up dataclassification and related algorithm, utility and visualization modules.

2.7. Key Classes

The key class search found 33 key classes from a total 762 classes. All 33 key classes were confirmed to be base or interface classes. The search did fail to find some interface and base classes. By definition, these missed key classes all had less than three immediate descendants – too small of a tree to be counted as a full sub-system leader.

2.8. Groups

This section will compare some of the more notable key class groups as produced by the DF and VF algorithms. groups

- *kernel_i* (DF: 1st with 142, VF: 1st with 152), base interface for all modules. All algorithm modules implement this interface causing this class to be highly rated by both flooding types. Similar group sizes, identical rankings.
- *object* (DF: 2nd with 65, VF: 2nd with 116) A general base class for all reference countable (via auto pointers) and serializable objects. A fundamental class. The VF group was notably almost twice the size of the DF group – a reflection of the pervasiveness of this base class across many edge types.
- *proponentwidget* (DF: 3rd with 22, VF: 5th with 25) Comparable group sizes, but slightly different positioning in the sorted report list.
- *oflow_i* (DF: 5th with 12, VF: 3rd with 27) Another case of VF awarding this pervasive interface a group size of almost twice that of DF.

- *full_image_matrix_base* (DF: 30th with 4, VF: 7th with 12) VF gave this key class a much higher group size, and thus, a much better ranking.

In summary, for very large key class groups, DF and VF give comparable rankings. and the expected primary key classes surfaced to the top of the ordered list for both algorithms. VF often will give base (non-interface) classes larger effect groupings than DF. Closer examination from a sampling of such classes showed that VF also includes – as expected – not only the descendant classes, but several auxiliary and utility classes near the key class. Finally, for all key class types, VFs group sizes were at least as large, and often larger than DF's. Again, this is expected because VF utilizes all the edge types.

2.9. Orphans

DF orphaned 354 (46%) classes (out of 762) while VF orphaned 210 (28%). Orphaned classes are those classes that did not have any (zero) dependence on any key class. Subsequent review of the orphaned classes illustrate that small (parent-less) utility classes were orphaned, as well as sub-systems with head classes that were not defined as key classes (i.e., the top parent class had less than the required three immediate descendant classes). These cases were less numerous with VF that utilizes all the edge types – and not just the parent/descendant edges – to propagate dependence information.

However, the bulk of the orphaned classes came from Scopira's heavy use of Generic Programming (via C++'s template mechanism). These classes were the results of template instantiations, and usually had no parent or descendant classes were relatively small and contained many in-line methods. This grouping of classes include auto pointers (memory management), trait structures (type information), thread lockers (concurrency control) and fundamental mathematical classes.

2.10. Utility Interfaces

One possible concern of the grouping algorithm was the lack of distinction between key interface/base classes and utility interface classes. Utility interface classes tend to have descendants that span multiple sub-systems as they tend to describe sub-system neutral activities.

Scopira is much more template heavy than interface heavy, and as a result, only has two such utility interfaces. Both utility interfaces (each with empty class groups) were at the end of the sorted group list. For applications and libraries that are more interface heavy (and as a result, have more utility interfaces) this phenomena may be a concern.

One possible method of detecting utility interfaces could exploit the pattern that most utility classes are not the sole parent classes for many of their descendants. Therefore, for a key class K, we count how many of its immediate descendant classes have more than one parent. The greater this count, the greater likelihood that class K is a utility interface.

3. CONCLUSION

Overall, we found that the flooding based algorithm correctly found all the notable key classes and properly grouped their member classes. On our test source code base, the reported sub-systems correctly matched those that were intended in the application design.

Furthermore, by examining two different algorithms, we found that the flooding algorithm that did not limit itself to traditional descendant/parent relationships gave more accurate grouping reports.

Finally, these metrics are gathered and computed directly from the application's source code, without developer input. This permits their application to existing projects and gives developers instantaneous feedback.

4. ACKNOWLEDGMENTS

NSERC is gratefully acknowledged for its financial support of this work.

5. REFERENCES

[1] S.R. Chidamber and C.F. Kemerer. A metrics suide for object-oriented design. *IEEE Trans. Software Engineering*, 6:476–493, June 1996.

[2] A.B. Demko, N.J. Pizzi, and R.L. Somorjai. Scopira - a system for the analysis of biomedical data. In *Canadian Conference on Electrical and Computer Engineering*, pages 1093–1098, 2002.

[3] Pizzi N.J., Demko A.B., and Vivanco R. Discrimination of software quality in a biomedical data analysis system. In *Proc Joint 9th IFSA World Congress and 20th NAFIPS Intl Conf, Vancouver, Canada*, pages 1702–1707, July 2001.