Illustrating Software Modifiability - Capturing Cohesion and Coupling in a Force-Optimized Graph

Johannes Holvitie, Ville Leppänen
TUCS - Turku Centre for Computer Science &
Department of Information Technology, University of Turku
Turku, Finland
{jjholv,ville.leppanen}@utu.fi

Abstract—**Software visualization aims to provide a more human-readable interface for the various software system aspects and characteristics. As majority of the time spent on modifying software is spent on gaining an understanding of an intangible and virtual system, the area of software visualization is widely researched as a solution to this. The paper in question presents a program visualization approach that focuses on illustrating the two software modifiability characteristics of cohesion and coupling. Unlike other approaches, which provide a visual representation for precalculated values, it uses the underlying cohesion and coupling mechanics to derive the actual layout. This allows the user to perceive the entire structure that has resulted to the cohesion and coupling values present in viewed nodes. There are three distinct steps to our approach. 1) Semantic analysis is used to record the static program structure into a directed and weighted graph. 2) The graph is then laid out using force-optimization to highlight important implementation structures. Finally, 3) sub-graph separation and further visual aids are provided to aid the user in observing cohesion and coupling for specific areas. Discussed benefits for this approach include information production efficiency, the ability to quickly analyze even large software implementations and intuitiveness of the visual delivery method.**

I. Introduction

Increasingly complex hardware has implicitly allowed more complicated software to be ran on it. This, in addition to the popularity of modern document-light, iterative and incremental software development methods, has created a challenging ground for software development tasks that depend on information regarding the software implementation's state. The likes of iteration planning and software maintenance are forced to deal with high levels of inaccuracy as they manage large, self-emergent, and intangible software systems.

Software visualization aims to increase tangibility by way of utilizing graphical illustrations to depict varying structures within the software [1]. The aim is to facilitate both human understanding as well as effective use of the illustrated parts [2]. Program visualization is a sub-method of the former. It works with static and dynamic data to provide views into software implementations' structural and functional properties respectively. Program visualization is fundamentally an information retrieval method.

Software process components that allocate resources either for reparative or function additive actions are interested in how the current implementation is capable of accommodating modifications. Cohesion and coupling are statically assessable measures of modifiability [3]. A number of metrics, e.g. [4]–[8], capture them but they predominantly produce results which require combination and prolonged analysis in order to reach their full potential. Hectic software development environments can find this discouraging.

This paper introduces a novel program visualization approach that captures information regarding software implementation's state through utilizing the cohesion and coupling mechanisms. The main goal of the presented approach is to serve software projects with an efficient, intuitive and easily attainable medium that allows communicating about the state of software and especially its capability to accommodate modifications.

There are three steps to our approach: semantic code analysis, graph layout, and sub-graph separation. In the semantic analysis we traverse the source code of a program and form a set of Abstract Syntax Trees (AST). These ASTs capture program element interaction at the lowest abstraction level. Through filtering the ASTs we produce data on which of the elements call one another and how many times. As cohesion and coupling are indicators of program element interdependency measured through their relations, we transform the filtered call information to a sparse matrix—a basic form for a graph.

The second step, the graph layout, takes the sparse matrix graph and applies a force-directed layout algorithm to it. In the graph, program elements are represented by nodes and directed weighted edges connecting them represent the elements' call directions and frequencies respectively. In force-directed layout, the edges are perceived to be forces which influence the nodes. The layout is complete when a minimum energy state has been found for the graph. As the forces in this graph represent cohesion and coupling meta-information, the produced layout consists from structures that capture and highlight modifiability characteristics.

In the sub-graph separation part the user may query the large system-wide graph with program elements to produce sub-graphs that highlight the cohesion and coupling structures to which these elements belong. The sub-graphs are built by identifying the shortest non-weighted path for each element pair in the query set. The final sub-graph consists from all encountered shortest paths and the nodes immediately adjacent to these. We call these adjacent nodes context nodes as they deliver information on the query's neighborhood. The user can toggle displaying them.

Rest of the paper is structured as follows. Section II reviews background literature introducing cohesion and coupling, program visualization and applicable graph processing algorithms as well as our previous work. Section III introduces the approach by defining used graph components, formation of the graph and its layout, and querying for sub-graphs. Section IV demonstrates the approach by applying it for a large open-source software product. Received graphs are analyzed, discussed and evaluated in Sections V, VI and VII respectively. Finally, Section VIII concludes this paper and makes some sights into future.

II. Related Work and Background

In this paper we introduce a program visualization approach that applies force-directed layouting to a graph containing cohesion and coupling details for a software system. The first subsection here defines cohesion and coupling to provide a reference point for the approach's graph formation (see Section III-A). The second subsection derives requirements for the visualization in discussing related work on program visualization. The third subsection describes the chosen graph processing algorithms and other considered options. Finally, the last subsection presents our previous work to which the approach herein is an extension.

A. Cohesion and Coupling

The concepts of software cohesion and coupling were introduced by Stevens et al. in [3]. Coupling is defined as the measure of module interdependence. A high degree of coupling for a module indicates that its functionality is heavily dependent on the existence of other modules. A modification in these can be expected to cause a modification in the dependent. Reducing coupling minimizes the propagation of modifications. If modules are built from elements, then we need to minimize the relations between elements not in the same module. Cohesion captures this in measuring the interdependence of elements within the same module. High cohesion for a module indicates that elements forming it are together well capable of implementing its functional requirements—without outside assistance.

The rather abstract definition of cohesion and coupling has lead to the emergence of several capturing metrics for them. Many of them are not exclusive as they capture cohesion and coupling with slightly different characteristics. For capturing cohesion, the most well known metrics are the six versions of LCOM (Lack of Cohesion in Methods). Versions zero through three, presented in [4], [5], [6] and [7] respectively, capture lack of cohesion as the volume of those member functions that do not share a common variable. In LCOM4 member classification is ignored and cohesion is measured as the number of inter-function calls and the variables they share [7]. Finally, LCOM5 measures cohesion as the number of functions assessing variables [5].

Two notable coupling metrics are components to the Instability metric defined by Martin in [8]. Instability measures change resistance for a software component by calculating its Efferent (E_c) and Afferent Coupling (A_c)—the component's independence and responsibility respectively. Afferent coupling captures the number of outside components that depend upon modules within the component. Efferent coupling does the opposite in measuring the number of foreign modules required by the target component.

B. Program Visualization

Caserta and Zendra argue in their survey of static program visualization approaches [9] that *graphs* have the most suitable visualization characteristics for architecture-level illustrations of source code snapshots. They add that for large systems graph occlusion and edge congestion are possible challenges. From the surveyd approaches the *CodeCity* [10] is the only one utilizing cohesion or coupling for layout placement. While this is seen to be intuitive and effective, drastic changes caused by snapshot updates are source of confusion.

Langelier et al. [11] discuss a visualization approach for software development and maintenance. While also acknowledging the role of cohesion and coupling in the process, they conclude by claiming that for target analysis a hybrid approach of efficient automated visualization and human interaction for context limiting is a good compromise. In [12] Koschke lists problems relating to maintenance and re-engineering related software visualization. Most notable ones from this list are large graph size, possibility of filtering, software evolution driven visualization updates and taking node and edge semantics into account for automatic layouts.

C. Graph Processing

Finding global minima for a force-directed graph is an extremely difficult process with no proven solutions [13]. There however exist methods with the ability to produce "good" results. These approaches generally use a multi-algorithm or -level approach. An example of a multi-algorithm approach is presented in [14]. While, the multi-algorithm ones produce exemplar results, they suffer from high time and space complexities. Yifan Hu's multilevel algorithm [13] is an example of the latter. Using graph coarsening, initial layouting and then refinement, the algorithm produces quality results with acceptable complexity.

For our layout, we use the continuous force-directed layout algorithm ForceAtlas2 [15]. It has adapted its energy model from the energy models proposed by Fruchterman and Reingold [16] as well as Noack [17]. The complexity is decreased from $O(n^2)$ to $O(n * ln(n))$ by applying Barnes Hut [18] approach for force approximation. According to tests conducted by the algorithm's author, ForceAtlas2 fares well against the notable multilevel layout algorithm by Yifan Hu [13]. Yifan Hu's multi-level algorithm would have been used in this work, if not for its inability to consider edge weights.

As stated, after converting the implementation structure into a graph, we may utilize link structure analysis in order to calculate global importance ranking for its nodes. We don't consider query specific ranking algorithms here as similar information is already produced by our approach through the query based sub-graphs. One of the most used and studied non-query ranking algorithm is the PageRank by Page and Brin [19] and there exist several adaptations of it for the software context. Many of these have found applying PageRank to being especially useful in change impact analysis for indicating globally and locally important elements. Calculating PageRank values for nodes in our graphs allows us to carry similar information to the queried sub-graphs—in addition to the visual information.

D. Previous Work

In our previous work we have introduced a mechanism for capturing notions about technical debt and displaying them at both the software implementation level as well as the project management level [20]. The tool that we have implemented for this mechanism is an Eclipse development environment plug-in called DebtFlag. The plug-in allows the use of different propagation models in order to support technical debt accumulation for implementations where the accumulation process differs due to e.g. the used implementation technique. We have also conducted a case study on a refactorization project in order to explore the various capabilities of technical debt accumulation and the role of dependency propagation in realizing them [21]. In this, we noted that the number of incoming dependencies for a module correlates with the number of modifications it invokes thus indicating that dependency propagation is a driver in technical debt accumulation, technical debt diminished due to dependency propagation and that the role of a system component explains, to a certain extent, the size and distribution of technical debt. In addition to this, we have also studied cohesion metrics in [22].

III. Approach

In this section we give an abstract description for our program visualization approach. We start by defining the graph components that present program elements and their relations. This is followed by a description for the semantic analysis procedure used to capture relations from the target program and to populate the graph. The complete graph is then laid out using force-optimization to visualize the information carried in its directed and weighted edges. Finally, queries can be made to the complete graph in order to separate sub-graphs to examine areas of interest.

A. Graph Components

Section II-A introduced cohesion and coupling as indicators of program complexity that base on information regarding the interdependence of individual program elements. In the first step of our approach this information is captured in a graph. The graph nodes represent the program elements and edges represent their interactions. Regarding the graph's input into force-optimization, actual information is carried as node and edge weights. As per the definitions of cohesion and coupling, this information constitutes only the degree of dependence between each program element pair. This leads to the following definitions for the graph components.

A node is a representation of a program element at an abstraction level in which explicit dependencies are formed. The abstraction level is dependent on the source implementation's paradigm and technique. For example, the class member level is considered for the object-oriented paradigm. In object-oriented implementations dependencies are formed against interfaces, interfaces are defined as classes, and classes consist from methods and variables. Hence, the program elements captured for an object-oriented implementation are the interface-forming methods and variables (applied in Section IV). Further, we note that the definitions of cohesion and coupling do not differentiate between types of software elements. From the perspective of force-optimized layout this means that all the nodes capturing the program elements should exert a uniform force. Hence, the nodes are left with equal weights and they only carry the program element's name as a unique identifier.

Edges capture relationships for all nodes in the graph. Cohesion and coupling are proportional to the number of dependencies between program elements. Interpreting this as a force-optimization problem means that for all program element pairs, the edge weights between nodes representing them is equal to the number of references between the elements. As both elements in a pair may invoke the other, the edges are directional to capture the two-way connection. When input to force-optimization, the total force between a node pair is the sum of directional weights for edges that directly connect them.

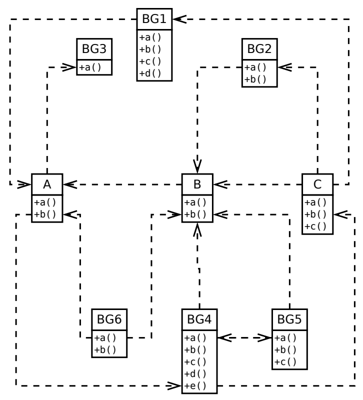
As a special case, a program element may refer to itself. Here the modeled edge has the same node in both ends. From the perspective of cohesion and coupling this type of reference has no value as it does not affect how the element connects to the surrounding system. Our approach takes this into account by default. Since the node weights are uniform and the edge capturing the self-reference is a loop, there is no observable force outside the node. Hence it can not be taken into account by force-optimization. This does however become an issue when applying link structure analysis. For example, the PageRank algorithm distributes node rank according to outbound edges. This values the rank of a self-referencing node a bit higher. As a remedy, loops with length one can be ignored when calculating these rankings.

B. Software Implementation Graph and Layout

Forming the software implementation graph considers limiting off the implementation area, setting up a semantic program code analyzer and laying out the complete graph. These matters are overgone in the following. To facilitate efficient application we have provided a solution to automate these steps after initial user input (see Section IV).

A software implementation graph is a static call graph where the directed and weighted edges record call directions and frequencies for the uniform nodes that represent program elements (as defined in Section III-A). Before forming the graph, we must dictate which parts of the system will be considered. Usually this is a trivial matter of drawing the line between modifiable and unmodifiable components. The division has a drastic effect on the graph composition as unmodifiable components usually consist of static libraries towards which most relationships are introduced. Leaving the unmanageable assets out focuses the visualization but provides a less realistic overall picture. Abstract description of the delimitation allows automatic classification of encountered components.

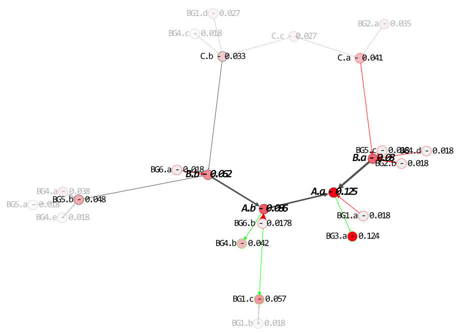
After the limitation we are left with a number of source code files encompassing the target implementation. Large size calls for automated approaches to determine relationships. A number of semantic analyzers exist for different programming languages to overcome this. The expected analysis output is an Abstract Syntax Tree (AST). The AST captures program element names and their types. This information can be used



(a) UML class diagram

$$
\begin{array}{c|cccccccc}
 & A.a & A.b & B.a & B.b & C.a & C.b & C.c & BGn \\
\hline
A.a & & & & & & & & \dots \\
A.b & 3 & & & & & & & \dots \\
B.a & 4 & & & & & & & \dots \\
B.b & & 5 & & & & & & \dots \\
C.a & & & 1 & & & & & \dots \\
C.b & & & & 1 & 2 & & 1 & \dots \\
C.c & & & & & 1 & & & \dots \\
BGn & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots
\end{array}
$$

(b) Relations captured in a sparse matrix



(c) Complete graph laid out with ForceAtlas2 and displaying a context query `B.a, B.b`

Fig. 1: Example program

to determine relations between them. Consulting the language's semantic specification [23] allows to distinguish the abstraction level discussed in Section III-A and to derive a library of possible element types. This library can then be used to transform the AST to a sparse matrix where column and row headers represent valid program elements and cells capture the number of directed relations between them.

The sparse matrix (see Figure 1b) is a complete graph lacking visual representation. To layout the graph, we use the ForceAtlas2 layout algorithm (see Section II-C). No user input is required for this step if the chosen default values for running the layout algorithm are accepted. In our approach we utilize the Gephi visualization engine and toolkit [24]. It allows us to convert the sparse matrix in to an in-memory graph and to automate its layout.

Finally, we may calculate a global importance ranking for the nodes in the graph in order to highlight interesting areas prior to context specific querying. PageRank is well supported by several static analysis tools as well as graph visualization tools. Figure 1c presents the final layout for the program in Figure 1a with corresponding PageRank values indicated for its nodes.

C. Context Querying for Sub-Graph Extraction

The graph formed in Section III-B can be used to make general observations but observing cohesion and coupling for a specific program element subset, a context, requires that it is separated from the system graph. Queries for the discussed element-level are supported as a default, since the graph nodes contain the element identifiers. Support for higher level queries requires that the approach is provided with semantic knowledge that can be utilized to convert the query back to the element level. After receiving the context query the separation is visualized by identifying relationships between the query's elements.

Relationships within a context are recorded as shortest paths between all possible component combinations that form the context. Use of the shortest paths approach is justified in that the graph formed in Section III-B is fundamentally a static call graph. The shortest found path between two elements is the least interfered demonstration of a relationship for them. In the case of several shortest paths existing for a pair they are all are considered. The weights and directions are considered irrelevant during this process firstly because the information is already present in the graph layout and secondly because reference frequency above zero is enough evidence to indicate existence of a relationship.

Figure 1c represents the complete graph with a query specific sub-graph extracted from it. The separated and highlighted graph is the result of applying the shortest path approach presented in the previous paragraph. The input to this approach encompasses a context containing two program elements `B.a` and `B.b`. In addition to the query itself, the highlight records all directly related elements for all paths. The related elements can be left undisplayed leading to a less obfuscated visualization but this leaves out the often interesting immediate neighborhood for this context. Carrying the neighboring nodes is always done at the cost of clarity.

For displaying the nodes' immediate neighborhood we use the following colors. For edges, shortest found paths are marked with black so as to clearly indicate relations within the queried context, those representing dependencies to context nodes are marked red and those representing dependencies originating from context nodes are marked green.

Now, the red edges capture the 'change group' for the query context. That is, if changes were made to nodes in the query context and the changes would not be contained within the nodes themselves, additional changes would propagate through the relationships indicate by the 'change group'.

Similarly, the green edges capture the possible 'root cause set'. This is the set to which the context nodes are directly dependent on to. For example, if a query is utilized to discover a cause for a problem within a certain context, then the 'root cause set' should also be considered as the functionality implemented in the query context is directly dependent on to and affected by this set. Further coloring, such as gradient coloring, for the nodes can be used to indicate superimposed rankings (like PageRank in Figure 1c).

Alpha blending (in Figure 1c) is used to fade out non-considered parts and to enable comparison between the sub-graph and the host graph. Allowing relative distances to be observed for the sub-graph constitutes a major contribution for this paper. For a force-optimized graph that captures cohesion and coupling, observing small distances between its nodes conveys information about high cohesion for it. Similarly, at a higher level, if a sub-graph forms a hub that is clearly distinguishable from the host graph then the sub-graph has captured a context which should be loosely coupled.

IV. Application

We wanted to trial our approach on a large open source software project with a well documented development history. Due to previous experience with the Eclipse project it was selected. Eclipse Foundation develops an open-source integrated development environment (IDE) that supports editing and building a multitude of languages. In the following, we discuss composition of data, building a system wide and project specific implementation graphs, using bug reports as query contexts, and the extraction of context specific graphs.

A. *Eclipse*

The Eclipse platform architecture is built on the concept of plug-ins. Functionality is provided through them and they can be extended to introduce user-defined additions. The Eclipse foundation manages development of core plug-ins and divides them into sets called products. The core release of Eclipse with Java-language support encompasses two products: `JDT` and `Platform`. We consider the Eclipse release version 3.0 for our trials.

We capture cohesion and coupling in the Eclipse system by building a single large graph to encompass the entire system as well as smaller, more focused graphs. In building the graphs we follow the process described in Section III-B.

We utilize the services of our DebtFlag plug-in (Section II-D) to first discover all source components at the valid abstraction level. The plug-in uses Eclipse's AST processor to accomplish the task. Since Eclipse is implemented using the object-oriented Java language, we identify that all functionality is implemented in classes. Classes are described by their interfaces which are constructed from members of varying visibility. Java Language Specification [23] dictates that these can be either variables or methods. This corresponds to the sought after abstraction level and we capture it by generating a node in the graph for all such occurrences. The DebtFlag is utilized again to discover all direct use relations between any two discovered interface parts. All such occurrences are modeled as edges where the weight carries information regarding invocation frequency for this pair and this direction.

Applying this process for the entire Eclipse's version 3.0, yields us with a graph containing 121K nodes and 1.50M weighted directed edges between them. Iterative construction of this graph took 36 minutes when running 4 threads on an Intel Core i5-2410M @ 2.3GHz and 8GB RAM machine. After completion, the PageRank values can be calculated for each node to change their coloring to reflect this. Figure 2 presents this graph after 1421 iterations of the ForceAtlas2 algorithm (see Section III-B). The continuous algorithm was stopped after no movement was perceived between graph layout iterations. The layout took 17 minutes with the Gephi toolkit [24] when running 8 threads on an Intel Core i7-2600K @ 4.7GHz and 16GB RAM machine.

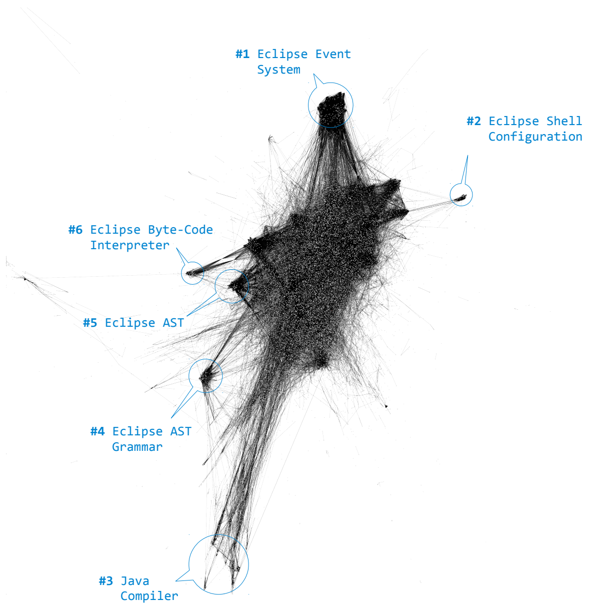


Fig. 2: Complete cohesion and coupling graph for the Eclipse system. Distinguishable hubs are highlighted

Since the system wide graph (Figure 2) is very large, we extract a smaller, more focused, graph encompassing a single project. For our example, we chose the *Debug* project, which is responsible for the `JDT.Debug` and `Platform.Debug` components. This graph is formed by extracting their elements from the system wide graph. Figure 3 contrasts the extraction against the system wide graph.

B. *Extracting Sub-Graphs*

Bugs represent an area for which resolving context cohesion and coupling is of especial interest as this communicates about the ease of chance for it. Constructing the graphs for Eclipse's version 3.0 allowed us to survey the Eclipse bug database in order to identify candidate bugs. In the following, we present an example bug, derive a context of interest from it and finally extract a sub-graph to present cohesion and coupling for this context.

In Section IV-A we formed a graph for the Eclipse Debug project. We query the Eclipse bug tracker for a bug declared for this project and for version 3.0. Bug number 148965 was chosen for this example. Initial documentation for this bug declares two problem elements `CompositeSourceContainer.findSourceElements(..)` and `PackageFragmentRootSourceContainer.findSourceElements(..)`. To identify cohesion and coupling for this context we proceed as described in Section III-C to form a sub-graph.

In this case, a direct connection exists between the two context elements resulting in finding a single shortest path with length one. Figure 4 displays the extraction with context nodes highlighted against the Debug project's graph in Figure 3.

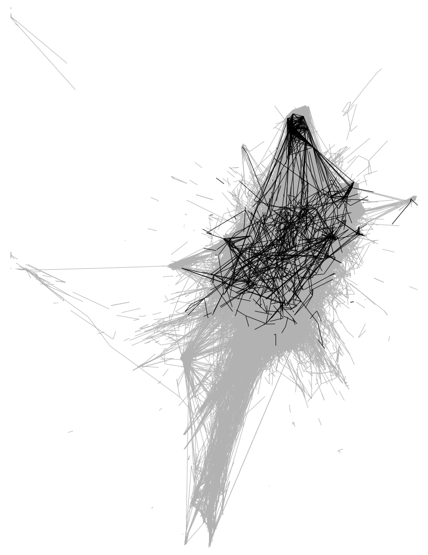


Fig. 3: Cohesion and coupling graph for the Debug project

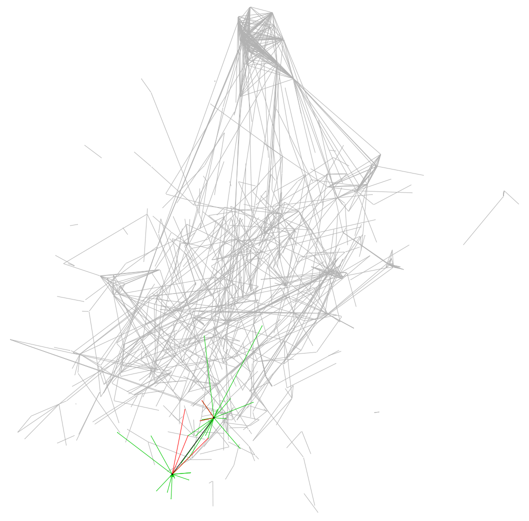


Fig. 4: Bug #148965 highlighted from the Debug project graph

V. Analysis

In Section III we introduced a process to form and layout a cohesion and coupling aware software implementation graph. In Section IV we applied this process for the Eclipse implementation to derive three graphs with increasing accuracy. In this section, we provide an analysis of the graphs in order to distinguish advantages and challenges related to the presented approach.

A. *System Wide Graph*

The system wide graph was formed in Section IV-A and depicted in Figure 2. Due to the vastness of Eclipse's implementation, we needed to combat against obfuscation (see Section II-B) when displaying it. This is done via applying a preview ratio: the layout is derived, as described previously, for all components but only a fifth of them are displayed. For dense graphs, this procedure retains global size and measure information while making the graph more approachable. When displaying query contexts in smaller graphs, this ratio may not be applied as position of every node carries valuable information.

Inspecting the received layout (in Figure 2), we first note that the graph consists from a number of hubs. Going over the nodes in the hubs, we note that single hubs capture program elements that are closely related: program elements are either from the same class or from a combination of classes responsible for implementing a shared functionality. This behavior is expected from a graph capturing cohesion and coupling.

There is high hub density in the main body of the system wide graph. Taking into account that the graph captures the core implementation for the Eclipse system we can expect a volume of nodes to be close to one another. The high density does however make the centre of the graph rather obfuscated and without dynamic highlighting or further node reduction it is very difficult to observe singular hubs in this area. We do however observe a number of hubs protruding from the centre mass.

Figure 2 has six of these outer hubs marked with numbers. Number one contains elements that form the Eclipse user interface and resource control event system. Second contains elements responsible for Eclipse's shell configuration. Third contains elements interacting with Java compilers. Fifth and fourth contain the Eclipse AST and its grammar respectively. Finally, sixth contains the Eclipse Java byte-code interpreter. All these hubs share a common trait in implementing a very specific functionality and we further argue that the hubs distance from the graphs centre correlates with the level of independence—that is high cohesion and low coupling—perceived for each hub.

B. *Sub-Graphs*

We extracted the Debug project from the system wide graph and presented the results in Figure 3. As mentioned, this project consists from two parts `JDT.Debug` and `Platform.Debug`. The former implements a language independent debugging model, where as the latter extends on this to provide Java debugging.

Form of the Debug project's graph can be explained as follows. The centre mass consist mainly from implementing the debugging tools in the user interface. Longer reaching edges capture queries to the AST, the process and memory controllers as well as configuration of the debugging shell. All these interface parts rely on Eclipse's interface event system to function and thus a large distinct hub exists at the top of the graph (see Section V-A). The project graph leads us to argue that the Debug project does not make unexpected references, it is tightly coupled with Eclipse's core implementation and this results in a low level of intra-project cohesion.

The context specific graph for Eclipse's bug no. 148965 is presented in Figure 4. The underlying graph is a scaled up version of the Debug project graph and the query can be seen highlighted at the bottom. Surveying this graph, we note that the separated context is at the edge of the project graph. We would not automatically interpret this as the context being a less cohesive part of the project but it does lead to another observation.

Since the context is at the edge we can observe that some of the green edges could be coming from outside the project graph. Since the green edges indicate possible root cause elements (see Section III-C), this can mean that the actual reason for the bug in question is in another project and it is just first observed for an element in the Debug project. Fortunately, similar behavior can not be observed for the red edges. This means, that when the bug's elements are modified, further spawned direct changes in dependents are constrained within the project.

Lastly, a notion about the context's cohesion measures. Our example query consisted from two program elements `CompositeSourceContainer.findSourceElements(..)` and `PackageFragmentRootSourceContainer.findSourceElements(..)`. While their signatures as well as the found shortest path of length one indicate close relation, the geometric distance between the elements—relative to the project graph—remains large. Observing the geometric distance makes it evident that these two components, despite their similarities, are actually coming from two different plug-ins `Platform.Debug` and `JDT.Debug` and as such may require divergent context knowledge when modified.

VI. Discussion

The previous Section V made a number of observations based on visual analysis. Here we discuss their implications in more detail. Regarding the analysis of the system wide graph, we observed that the approach is capable of separating hubs that had distinct functional goals. This is an important initial indication of the approach's autonomous capability to highlight structures that are of interest from the perspective of cohesion and coupling. We demonstrate metric results for this in the next section.

Further, we were able to make observations regarding the Debug project's integration into rest of the system in addition to deriving additional information for a bug in the Debug project. These observations show that the single visual presentation used was capable of letting the user explore structures ranging from thousands to just singular program elements. However, use of the 'preview ratio' for the system wide graph indicates that, in case of a very large system, the presentation is prone to edge congestion which obfuscates the visualization. Improvements to this are currently pursued and some possibilities are discussed in future work.

Regarding the context queries, the resulting sub-graph for bug 148965 could be used to argue that 1) the possible root cause for this problem maybe coming from outside the hosting Debug project, 2) all modifications spawned from fixing the bug would be limited to the hosting project and 3) the geometrical distances between the context elements indicated that the bug encompassed elements that were not very closely related. All these observations were made based on visually available information. We interpret lower implementation technique and context knowledge requirements for the made observations as an indication of the approach's intuitiveness.

VII. Evaluation

We discussed our observations about the approach's ability to distinguish structures of interest from the perspective of cohesion and coupling. In Section V-A we presented the entire Eclipse system (in Figure 2) and we discussed our expectations of cohesion and coupling correlating with the six distinguished hubs. To provide initial evaluation for our approach, we calculate well established measures of cohesion and coupling for these hubs.

Table I records cohesion and coupling measures for each distinguished hub (see Figure 2) as LCOM4 and total couplings values. In addition to specific hub values there are the values for all resources in Eclipse. All versions of LCOM (introduced in Section II-A) produce inversed results: lower ones indicate higher cohesion. The value range of LCOM4 is $[1.0, \infty]$. The total couplings measure is the sum of afferent A_c and efferent couplings E_c.

TABLE I: Cohesion and coupling measures for hubs

Hub	#	LCOM4	Couplings
Eclipse Event System	1	1,0583333333	41
Eclipse Shell Integration	2	1,0	39
Java Compiler	3	1,0	259
Eclipse AST Grammar	4	1,0333333333	47
Eclipse AST	5	1,35	49
Eclipse Byte-Code Intrp.	6	1,0230769231	44
Mean for all resources		**1,0606837607**	**36,3646723647**
St.dev. for all resources		**0,171359571**	**49,7891359905**

Inspecting the LCOM4 values for hubs in Table I we first note that most of them are below the average and very close to the bottom value of 1.0. This would seem to indicate that these hubs indeed capture element sets that display high cohesion and represent the more cohesive areas of the entire implementation. The fifth hub is an exception to this in being much less cohesive than its counterparts. On visual inspection, we can see that the hub in question is closer to the center mass than the rest of the hubs. We interpret this as Eclipse AST being more coupled to the core system functionalities and thus being a less cohesive part on its own account.

We explain this phenomenon as follows. The average values are calculated for all resources required by the Eclipse system. The LCOM4 metrics indicate that the average resource is generally less cohesive than a distinguished hub. This indicates that the average resource implements more partial functionalities, while a distinguished hub implements a more complete and independent functionality. Thus, the system's coupling to a hub can be expected to be larger since all communication to access a functionality is mainly between the hub and the system. While in the case of an average resource, the communication extends to all resources that implement the complete functionality. This is apparent for hub number three. The Java Compiler is a very independent unit and its communication consists only from a handful of library objects (e.g. ASTs) to receive source code and to deliver the compilation results. While the libraries are very distinctive and unique dependencies, they are composed from several hundreds of definitions (e.g. language syntax). Since the hub has captured the compilers functionality rather well, this is highlighted in the large coupling value.

VIII. Conclusions and Future Work

In this paper, we have demonstrated an approach to capturing information for software implementation contexts by way of utilizing cohesion and coupling aware graphs laid out using force-optimization. We applied the approach introduced in Section III to the Eclipse platform in Section IV. The received graphs were analyzed, discussed, and evaluated in Sections V, VI, and VII respectively. This section concludes the paper by discussing challenges, advantages and future work related to the presented approach.

Program visualization and the mechanics utilized in it still prove to be challenging and our approach is no exception from this. Especially, the initial setup for it remains somewhat cumbersome. The production of graphs requires access to source code, a preset AST parser, a layout engine and a visualization library. However, after the initial setup, further context queries can be served automatically. Another challenge lies in defining the query contexts. Results produced by the approach are directly dependent on the provided contexts and as such the level of expertise in defining them correlates with attained sub-graph quality.

The advantages do however outweigh the remaining challenges. Use of force-optimization gives the ability to present cohesion and coupling in a very intuitive manner. The found natural layout provides visual emphasis for structures of importance. This allows even inexperienced users to make observations regarding software modifiability. Context queries to such graphs produce a medium in which it is efficient and easy to communicate about matters that would otherwise call for rigorous analysis of program dependency structures.

Ability to visualize and explore the system should prove useful when large and obfuscated systems are explored. The approach is being integrated into our DebtFlag tool [20] in order to introduce it as part of daily development activities. This also allows us to introduce interaction capabilities like dynamic highlighting, direct source code access and metrics driven partitioning to reduce edge congestion and increase clarity. Regarding research use, we are very interested in conducting studies to see if the highlighted structures can be associated with well known architectural patterns (e.g. Model-View-Controller) and problems related to them. This could also allow their identification even from fully obfuscated sources.

References

[1] T. Ball and S. G. Eick, "Software visualization in the large," *Computer*, vol. 29, no. 4, pp. 33–43, 1996.

[2] B. Price, R. Baecker, and I. Small, "An introduction to software visualization," in *Software Visualization*, J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, Eds. Cambridge MA, MIT Press, 1998, pp. 4–26.

[3] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[4] S. R. Chidamber and C. F. Kemerer, *Towards a metrics suite for object oriented design*. ACM, 1991, vol. 26, no. 11.

[5] A. Henderson-Sellers, Z. Yang, and R. Dickinson, "The project for intercomparison of land-surface parameterization schemes," *Bulletin of the American Meteorological Society*, vol. 74, no. 7, pp. 1335–1349, 1993.

[6] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, 1994.

[7] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems," in *Proceedings of the International Symposium on Applied Corporate Computing*, vol. 50, 1995, pp. 75–76.

[8] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[9] P. Caserta and O. Zendra, "Visualization of the static aspects of software: a survey," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 7, pp. 913–933, 2011.

[10] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*. IEEE, 2007, pp. 92–99.

[11] G. Langelier, H. Sahraoui, and P. Poulin, "Visualization-based analysis of quality for large-scale software systems," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 214–223.

[12] R. Koschke, "Software visualization for reverse engineering," in *Software Visualization*. Springer, 2002, pp. 138–150.

[13] Y. Hu, "Efficient, high-quality force-directed graph drawing," *Mathematica Journal*, vol. 10, no. 1, pp. 37–71, 2005.

[14] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, 2003, pp. 77–ff.

[15] M. Jacomy, S. Heymann, T. Venturini, and M. Bastian, "Forceatlas2, a continuous graph layout algorithm for handy network visualization," *Medialab Center of Research 560*, 2011.

[16] T. M. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software: Practice and experience*, vol. 21, no. 11, pp. 1129–1164, 1991.

[17] A. Noack, "Energy models for graph clustering." *J. Graph Algorithms Appl.*, vol. 11, no. 2, pp. 453–480, 2007.

[18] J. Barnes and P. Hut, "A hierarchical o (n log n) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[19] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Tech. Rep. 66, 1999.

[20] J. Holvitie and V. Leppänen, "DebtFlag: Technical Debt Management with a Development Environment Integrated Tool," in *Managing Technical Debt (MTD), 2013 Fourth International Workshop on*. IEEE, 2013.

[21] J. Holvitie, M.-J. Laakso, T. Rajala, E. Kaila, and V. Leppänen, "The role of dependency propagation in the accumulation of technical debt for software implementations," in *13th Symposium on Programming Languages and Software Tools*, k. Kiss, Ed. University of Szeged, 2013, p. 6175.

[22] S. Mäkelä and V. Leppänen, "Client-based cohesion metrics for java programs," *Science of Computer Programming*, vol. 74, no. 5, pp. 355–378, 2009.

[23] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java (TM) Language Specification, The (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[24] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," 2009. [Online]. Available: http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154