

6-2012

# IDENTIFYING CORE COMPONENTS IN SOFTWARE SYSTEMS

Phillip Meyer

*University of Nebraska at Omaha*

Follow this and additional works at: <https://digitalcommons.unomaha.edu/studentwork>

 Part of the [Computer Sciences Commons](#)

---

## Recommended Citation

Meyer, Phillip, "IDENTIFYING CORE COMPONENTS IN SOFTWARE SYSTEMS" (2012). *Student Work*. 2874.  
<https://digitalcommons.unomaha.edu/studentwork/2874>

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact [unodigitalcommons@unomaha.edu](mailto:unodigitalcommons@unomaha.edu).



# IDENTIFYING CORE COMPONENTS IN SOFTWARE SYSTEMS

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

University of Nebraska at Omaha

by

Phillip Meyer

June 2012

Supervisory Committee:

Dr. Harvey Siy

Dr. Sanjukta Bhowmick

Dr. William Mahoney

UMI Number: 1516935

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent on the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 1516935

Copyright 2012 by ProQuest LLC.

All rights reserved. This edition of the work is protected against unauthorized copying under Title 17, United States Code.



ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 - 1346

# Identifying Core Components in Software Systems

Phillip Meyer, MS

University of Nebraska, 2012

Advisors: Dr. Harvey Siy and Dr. Sanjukta Bhowmick

*Abstract* -- As large software systems are highly complex, they can be difficult for a developer to understand. If a core subset of a software system could be extracted which contains the most important classes and connections of the larger system, studying this core would be useful for efficiently understanding the overall system. In this research we examine research into core/periphery structures in networks, primarily focusing on the use of k-core decomposition. The extracted dependencies of three open source Java software systems provide the inputs, with forty different versions of these systems analyzed in total. We derive inter-class dependencies from these releases and represent them as undirected graphs. We extract the k-core values by recursively pruning the least connected nodes within the networks, leaving an inner core. The resulting coreness values are analyzed against centrality metrics and high-level communities detected by the Louvain method. Both system level and component level evolution of coreness values for these systems are studied. The validity of this approach for identifying software system cores is discussed and analyzed.

## Table of Contents

I.	Motivations.....	1
I.1	Cognition and Visualization.....	1
I.2	Maintenance and Reusability.....	2
II.	Related Research.....	4
II.1	[1979] Snyder and Kick .....	4
II.2	[1998] Tushman .....	5
II.3	[1999] Borgatti and Everett .....	6
II.4	[2005] Petter Holme .....	7
II.5	[2008] Haohua et al .....	8
II.6	[2008] Zimmermann and Nagappan .....	9
II.7	[2010] MacCormack et al.....	10
III.	Methodology.....	13
III.1	Overview of Research Steps.....	13
III.2	Workflow Diagrams.....	16
III.3	Tools Used.....	22
IV.	Examined Software Systems.....	23
IV.1	Hibernate-ORM.....	23
IV.2	Apache-Ant.....	24
IV.3	JHotDraw.....	25
V.	Dependency Extraction.....	27
VI.	Reproducing MacCormack.....	36
VII.	K-Core Decomposition.....	41
VII.1	K-Cores in Other Research Contexts.....	44
VIII.	Visualization of K-Cores.....	46
IX.	K-Core and Centrality.....	55
IX.1	Degree vs K-Core.....	55
IX.2	Closeness Centrality vs K-Core.....	58
IX.3	Betweenness Centrality vs K-Core.....	60
IX.4	Other Centrality Measures.....	63
X.	K-Core and Community Detection.....	65
XI.	K-Core Evolution.....	68
XI.1	System Level Evolution.....	68
XI.2	Component Level Evolution.....	73
XII.	K-Cores and Seniority.....	80
XIII.	Conclusions.....	84
XIV.	Future Work.....	86
XV.	References.....	90

## Figure Index

Figure 1: World Trade Matrices, Before and After.....	5
Figure 2: Workflow for Component Level Analysis.....	16
Figure 3: Workflow of Aggregate Analysis Steps.....	20
Figure 4: Inherited Members.....	28
Figure 5: Visible Outer-Class Members.....	30
Figure 6: Indirect Ancestors.....	31
Figure 7: Hibernate-ORM 4.0.0.CR4.....	36
Figure 8: Apache-Ant 1.8.0.....	36
Figure 9: JHotDraw 7.6.....	36
Figure 10: Hibernate-ORM Transitive.....	37
Figure 11: Apache-Ant Transitive.....	37
Figure 12: JHotDraw Transitive.....	37
Figure 13: Simple Graph Cores.....	41
Figure 14: Example K-Core Decomposition.....	42
Figure 15: Protein Interaction Network K-Cores.....	44
Figure 16: Hibernate 0.9.4 Lanet-VI.....	47
Figure 17: Hibernate-ORM 0.9.4 Inner 7-Core.....	48
Figure 18: Hibernate-ORM 4.1.6 Lanet-VI.....	49
Figure 19: Hibernate-ORM 4.1.0 Inner 16-Core.....	49
Figure 20: Apache-Ant 1.8.2 Lanet-VI.....	50
Figure 21: Apache-Ant Inner 11-Core.....	51
Figure 22: JHotDraw 7.6 Lanet-VI.....	51
Figure 23: JHotDraw Inner 11-Core.....	52
Figure 24: Scientific Collaboration.....	53
Figure 25: Internet Routers.....	53
Figure 26: Web Page Links.....	54
Figure 27: Random Graph.....	54
Figure 28: Degree and K-Core Hibernate-ORM.....	56
Figure 29: Apache-Ant Degree.....	57
Figure 30: JHotDraw Degree.....	57
Figure 31: Hibernate-ORM Closeness Centrality.....	58
Figure 32: Apache-Ant Closeness Centrality.....	59
Figure 33: JHotDraw Closeness Centrality.....	59
Figure 34: Hibernate-ORM Betweenness Centrality.....	61
Figure 35: Apache-Ant Betweenness Centrality.....	62
Figure 36: JHotDraw Betweenness Centrality.....	62
Figure 37: Hibernate-ORM System Level Evolution.....	68
Figure 38: Apache-Ant System Level Evolution.....	70
Figure 39: JHotDraw System Level Evolution.....	71
Figure 40: Plot K-Core and Vertice Count All Systems.....	72
Figure 41: K-Core Class History, Selected Hibernate Classes....	74
Figure 42: Correlated Hibernate "Type" Evolution.....	76
Figure 43: Correlated Hibernate "Parser" Evolution.....	76
Figure 44: Hibernate-ORM Correlated Evolution Heat-map.....	77
Figure 45: K-Core and Average Seniority.....	82

## Table Index

Table 1: Canonical types from MacCormack, 2010.....	11
Table 2: MacCormack Results on Examined Systems.....	39
Table 3: Hibernate-ORM Communities.....	66
Table 4: Apache-Ant and JHotDraw Communities.....	67
Table 5: Community/Package Prediction on Correlated Evolution..	79
Table 6: Average Seniority of Classes within K-Core.....	82

## I. Motivations

Real world software systems are composed of thousands of interconnected components. Depending on the programming language, these components could be classes, functions, aspects, packages or modules. However, not all of these components in a system are of equal importance. We collectively refer to the most important and vital components of the software system as the *core* of the system. If there existed an automated method of identifying the core of a complex software system, the knowledge of this core could be useful for several aspects of software engineering. These include the understandability, maintainability, and re-usability of the systems.

There are currently no industry accepted tools or techniques for discovering this essential software system core. In this research we examine existing concepts and techniques from the literature to discover a simple and easily automated approach to this identification.

### I.1 Cognition and Visualization

A software developer can be thought to possess two distinct types of knowledge; general and software-specific [Mayrhauser and Vans, 1995]. General knowledge is independent of the system they are attempting to understand, while software-specific knowledge applies to the individual software application. Part of this software-specific knowledge encompasses the mental model of a software system that one may hold.

As a new developer on an existing large software system, one can be faced with “information overload” when confronted with the task of studying the system as a whole.

There is just too much complexity to digest without a way to pare it down for consumption. One can work on a software project for a significant length of time without developing an accurate mental model of the subsystems and components that make up the system as a whole.

Adequate documentation of a software system can help alleviate this prohibitive learning curve, but this documentation often is lacking or non-existent. Many legacy software applications have little beyond the source code, with no experts remaining on the project to facilitate knowledge transfer.

It is in these situations where cognitive aids would be most useful for efficiently acquiring software-specific knowledge. A list of core components that is much smaller than the total system, yet contains the most important classes and connections of the system, could be one such cognitive aid. Building an accurate mental model of the more limited core is much easier, and the non-essential components are filtered out from consideration. Information overload is reduced or eliminated.

## **I.2 Maintenance and Reusability**

If one has a solid grasp over what classes compose the core of a software system, this knowledge could also aid in efforts to maintain the system. If they are aware of the relative importance of a component in relation to the system as a whole, a developer can quickly assess the potential impact of a potential change. A change to a critical core component would have a larger impact than updating a component which is only peripherally related to the system.

System reusability can be improved with an accurately identified core subsystem as well. For instance, if the system needed to be ported to a new programming language, it may be worthwhile to port the essential core of the system first. This decision would allow the core functionality of the system to be brought online sooner in the migration effort. The developers could also take extra caution when porting these core components, as a mistake here would be more costly than to a non-essential component.

Unit testing of the system would be impacted, as developers may want to focus on developing automated unit tests around the core components versus applying them in an arbitrary fashion. Writing test cases can be a time consuming process, and getting the most beneficial coverage for the least amount of work would be appealing for project management.

## II. Related Research

The concept of core/periphery organizational structures did not originate in the field of software engineering. It is an intuitive concept that has been used for a wide range of research fields; including product engineering, geography, biology, social sciences, and studies of corporate organizational structure. There also exist previous academic attempts to identify the important modules of software systems.

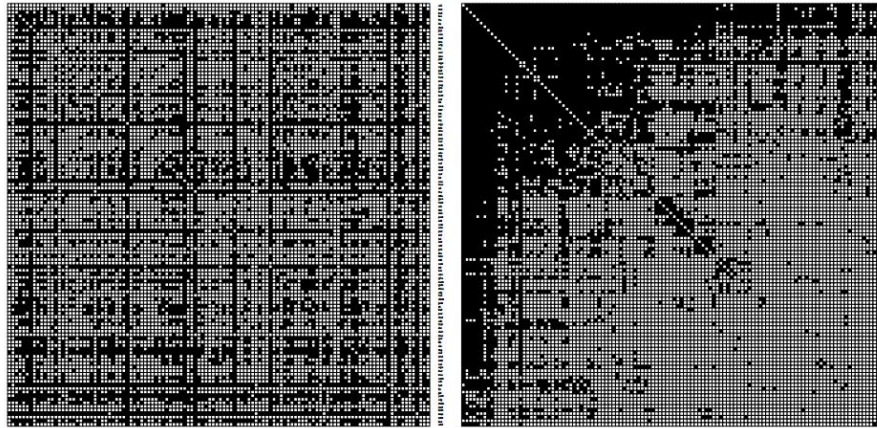
In preparing for this research, we discovered research related to ours in one of two possible directions; (1) Identifying the “most important”, “critical”, or “key” components of software applications and (2) Attempts to identify the core of a given network, not specific to software systems. Our work combines these two subjects, and so was influenced by papers from both.

These related works are presented here in chronological order. In addition, we devoted multiple weeks of study towards one particular related research effort which directly deals with identifying cores of software systems, the MacCormack et al. 2010 paper. We cover that research in further detail in section VI.

### ***II.1 [1979] Snyder and Kick***

One of the earliest references to core/periphery structures found in the literature is in the field of sociology, specifically in geopolitics. In the work done by Snyder and Kick, the team attempts to examine interactions between the nations of the world in order to group them into *core*, *periphery*, or *semi-peripheral* classifications. They do this based on modeling trade flows, military interventions, diplomatic relations and treaty

memberships as networks. They then used a *blockmodel* process on these networks (represented by matrices) to generate their classifications. Blockmodeling is a method of re-ordering the rows and columns in a matrix based on their structural equivalence.



*Figure 1: World Trade Matrices, Before and After*

Here we see the world trade matrix, with the alphabetical ordering on left, and the ordering after blockmodeling is applied on the right. Here the core nations would be towards the lower right quadrant, with the peripheral nations towards the upper left. This research appears to have been influential in its field, though has also faced significant criticism [Nolan, 1983].

## ***II.2 [1998] Tushman***

There are also several papers referencing the concept of core/periphery structures in the product engineering field. These were co-authored by Michael Tushman, in his research into *dominant designs*. In this context, it is observed that changes to core subsystems have a greater impact on the overall design of a product than other components [Tushman, 1998]. In this research, a core subsystem is defined as being one

that is “tightly coupled to other subsystems”. In contrast, a peripheral subsystem is one that is only loosely coupled to other subsystems. Examples of core subsystems in this context would possibly include the engine of a car, or the CPU of a computer. In contrast, examples of peripheral subsystems would then be less critical systems like the headlights of a car, or the CD/DVD drive on a computer.

Identification of core subsystems in engineering realms that deal with physical structures seems to be done entirely by intuition or post-hoc analytical processes. Examples are given of core subsystems with no formal identification criterion accompanying them.

### ***II.3 [1999] Borgatti and Everett***

In 1999, an influential paper was written by Stephen Borgatti and Martin Everett that first attempted to define core/periphery structures in a formal way. The authors also give two different conceptual models of the concept from which to develop core identification algorithms from.

The graph theoretical approaches presented by this research are very general, and can be used to identify core/periphery structures in any network. Examples of networks examined in the paper are diverse, including references between academic journals and the interactions between monkeys within a colony.

The two models of core/periphery structures presented by this paper are the *discrete* and *continuous* models. In the *discrete* model, the core is considered to be a sub-graph that is central to the graph. The goal of an algorithm then is to classify a node as

either belonging to the core or periphery sets of nodes. There is no middle ground. In the continuous model, the coreness of a node is instead a score along a range of possible values. A node can be “more core” than another.

The research admits a very close relationship between their concept of *core* and the more heavily researched graph theory concept of *centrality*. The key distinction made between the two is that the nodes found with algorithms that measure centrality do not necessarily result in a set of nodes that are at all connected. The goal then is to find the core of the graph as a well connected sub-graph, and not just a loose set of nodes. This research gives us a foundation of core identification that is distinct from simply finding the most central components.

## ***II.4 [2005] Petter Holme***

The research done by Petter Holme for the University of Michigan was our first exposure to the usage of *k-core decomposition* in analyzing the organization of graphs and networks. The details of this algorithm are explained in depth in a later section, as this approach ended up being the primary direction of our research.

In this paper the formal definition of core/periphery structures is referenced, and the K-Core algorithm from graph theory was applied as a simple means of satisfying this definition. The algorithm produces a core that is both well connected and central to the examined networks. Interestingly, the values produced by this algorithm can be considered as fulfilling both the discrete and continuous models of coreness. Each

component in the network is given an integer score which can be measured on a continuous scale, and the maximal k-core can be segregated from the rest of the graph and considered the discrete core of the system.

Holme extends the basic k-core decomposition algorithm by adding a step where each derived shell has its average closeness centrality calculated, and the shell with the highest average centrality considered the core. Our research identified this step as redundant for our examined software networks, as in each case the highest average centrality was the most innermost k-core.

In this research, networks from many different contexts are examined. These include geoFigureal networks (highways, pipelines, streets, airports), acquaintance networks, electronic networks, academic reference networks, food webs, neural networks, five different biochemical networks, and even one software dependency network. In the software dependency network based on the Linux operating system, the *package* was chosen as the component granularity.

From this point the focus of Holme's research is comparing these network types based on metrics computed from the derived coreness values. Here the networks with clear real world hubs such as airports and internet routing were found to have the highest *core-periphery coefficient* and *assortative mixing coefficient*. The Linux package dependency network had fairly low values for each, compared to the networks with a more well defined and obvious core.

## ***II.5 [2008] Haohua et al***

We discovered during the late stages of our research that we were not the first team to apply k-core decomposition to software systems for analysis. In 2008, a team from Shen'Yang China also used the k-cores of software systems for their research for studying software hierarchies. The research here looked at ten different software systems, each open source C/C++ systems. The K-Core values were computed, and some observations made about the output of the algorithm on these systems.

Our research goes deeper in a few different directions, such as the relationship between centrality and coreness in the examined systems. We also examine the evolution of k-core values within software systems, and the relationship between K-Core decomposition and community detection. This alternative direction serves to maintain the novelty of our research, and lends support the conclusions reached by Haohua et al.

## ***II.6 [2008] Zimmermann and Nagappan***

In this research, a diverse array of social network analysis measures were captured on the Windows Server 2003 software system. The team then used these metrics as a way to predict defects in the system at the binary file level. This component level is similar in concept to the Java class used in our research. Their results for each measure were statistically correlated with the defect tracking system in use by the developers, using Pearson and Spearman correlation coefficients.

The network analysis measures included both *global* and *ego* metrics. Here global refers to metrics that measured the node's relation to the entire software system, and ego metrics measure the relationship between a component and its immediate neighbors.

What they found was that some network analysis measures did offer some measure of predictive ability, though the ego network metrics proved more useful than the global network metrics. These predictive capabilities were more effective than similar efforts for predicting defects based on popular complexity metrics.

Most interesting to our research is that there exists a developer maintained list of critical binaries for Windows Server 2003. The team was able to predict these critical binaries with network analysis measures with a success rate of 60%, which is much better than the reported 30% reported for traditional complexity metrics.

This research showed that extensive work on conventional centrality measures had already been done on software dependency networks, which allowed us to focus on coreness measures instead. Later work by Nguyen et al. in 2010 reproduced this work on the Eclipse, an open source IDE (Integrated Development Environment) written in Java. They found that some results reached by Zimmermann were not applicable to their research context, while others did still provide some benefit for defect prediction.

## **II.7 [2010] MacCormack et al**

In a recent paper, a comprehensive analysis was done by a collaboration between MIT and Harvard Business School to determine whether software system design is dominated by core-periphery structures. Their research claimed that for the majority of software systems analyzed, core subsystems were present and detectable based on their invented algorithm.

The team used the call-graphs parsed from a large variety of software systems,

including commercial and open source projects. Their analysis looked at the components at the class level, and a *Design Structure Matrix* (DSM) was used to represent the dependencies between these classes.

The MacCormick research gives four different possible classifications for a component, based on the transitive fan-in value (FIV) and fan-out value (FOV) of each class. The combined variables create a two dimensional spectrum, from which the components can be classified into four “canonical types”.

<i>Core Components:</i> High FIV, High FOV	Files with high visibility on both measures are “Core” files. They are “seen by” many files and “see” many files. They are often linked directly or indirectly to all other core files.
<i>Shared Components:</i> High FIV, Low FOV	Files with high FIV are “Shared” files. They provide shared functionality to many different parts of the system. These files are “seen by” many files, but do not “see” many files.
<i>Peripheral Components:</i> Low FIV, Low FOV	Files with low visibility on both measures are “Peripheral” files. They are neither “seen by” many files nor “see” many files. They typically execute independently of other files.
<i>Control Components:</i> Low FIV, High FOV	Files with high VFO are “Control” files. They direct the flow of program control to different parts of the system. These files “see” many other files, but are not “seen by” many files.

*Table 1: Canonical types from MacCormack, 2010*

Their analysis then classified each component based on the results of matrix multiplication operations applied to the matrix, designed to determine the FIV and FOV values.

They then determine the maximum possible values for the combined FIV/FOV, and consider a component core if the combined values exceed 50% of the possible value. The cutoff of 50% seems to be entirely arbitrary, but the team still uses this classification

to make many observations about the domination of core-periphery structures in software systems.

This classification into core or not-core buckets is binary in nature, which resembles the discrete model of Borgatti and Everett. Two classes could be almost identical, but if one of them has just one single dependency more than the other, it could be enough tip the classification from one bucket to the other.

This arbitrary cutoff seems to be a limitation of the analysis, but was crucial for their central question of determining whether core components exists in a particular system. Their research reports that a significant number of software systems have no core components at all, which is in contrast with our intuitive understanding.

Our first experimental direction was in reproducing this research on our representative systems. The results of which are discussed in detail in a later section

### **III. Methodology**

In this section, we will give a high level overview of the path our research took. The details of these steps are given in later sections, along with our experimental results. Figureal representations of the research work-flows are presented in this section, to aid with attempts to duplicate this work.

#### **III.1 Overview of Research Steps**

##### **1) Select Software Systems**

The first step in our research is to select which software systems we want to examine for the purpose of core identification. We chose three open source software systems that have had active development for a length of time suited for evolutionary study.

##### **2) Extract the Software Dependencies**

Next we use a dependency extraction tool to discover both the use and inheritance relationships from the examined systems. We compared the outputs and features of several tools for this purpose, and made detail the critical decisions made when it came to representing object oriented software dependencies in a data structure.

##### **3) Reproduce the MacCormack Algorithm**

As the MacCormack research appeared to closely represent our research goals, we first examine their approach as a possible core identification method. We analyze the results of their algorithm, and reject their work as not suitable for our purposes. The rationale for

this decision is presented.

#### **4) Decompose Dependency Graphs using K-Core Algorithm**

We then look to K-Core decomposition as our core identification method. Finding the results to be promising for our purposes and having a strong theoretical basis, we then focus our efforts on this approach. The algorithm finds cores that are manageable size that are well connected and central to the software systems.

#### **5) Create Visualizations of the Systems for K-Core Structure**

There exists a very well done visualization method based on K-Core decomposition for representing the network structure in two dimensions. We employ this algorithm on our selected software systems, and make observations from them. In addition the inner core is extracted to its own visualization Figure with the peripheral components removed, which is then examined.

#### **6) Compare with Detected Communities**

Previous research done by members of our team focus on community detection in software networks. We investigate the high level communities (detected with the Louvain method) discovered in our software systems, and then compare them to the classes which compose our inner cores.

#### **7) Examine System Level Core Evolution**

Each software system has a 10+ year history of release versions to draw from. We select around thirteen versions of each system, and then examine the evolution of the inner core between these release versions.

#### **8) Examine Component Level Coreness Evolution**

We also use the history of k-core values for each component in the system to attempt to discover insights into those classes. We find that many classes have correlated k-core evolution, and we examine the correlations to determine if they can be used to predict whether the correlated classes have the same community or package.

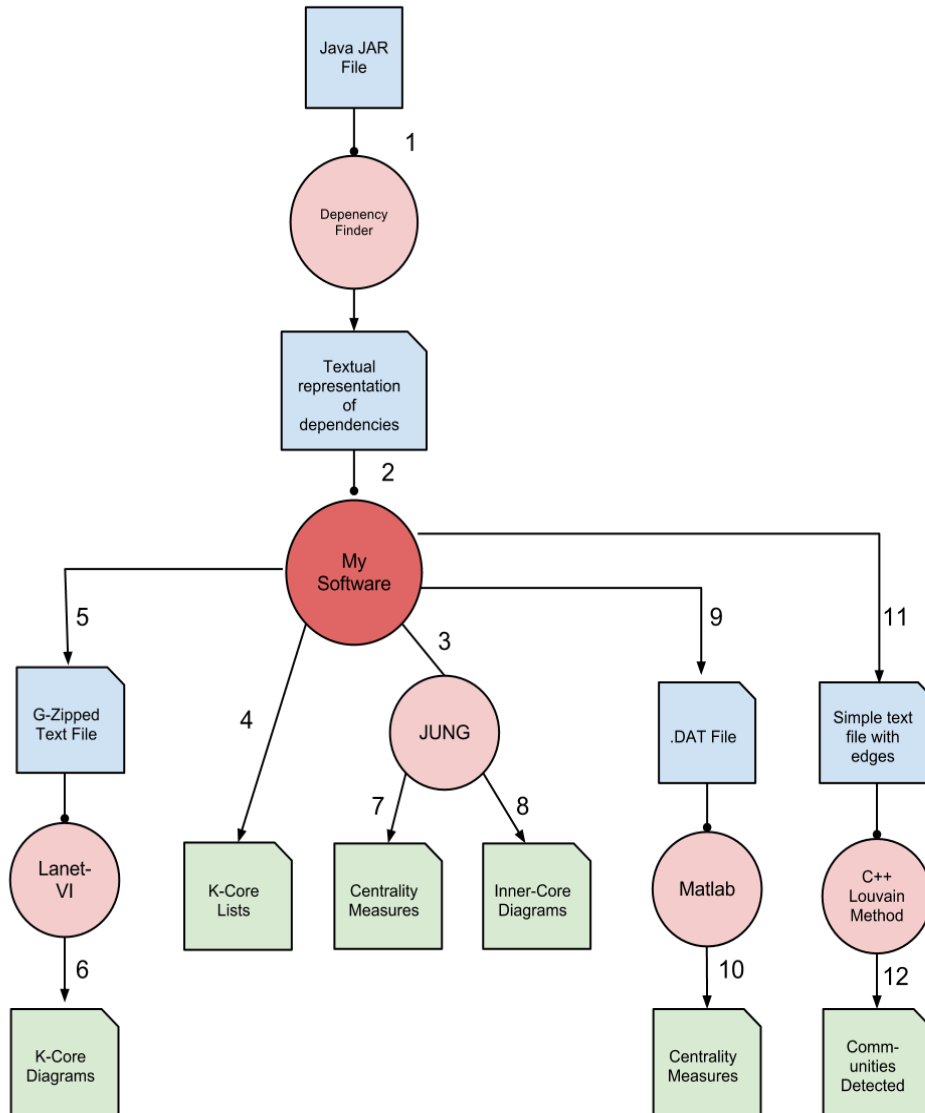
#### **9) Validate Cores with Seniority Metric**

We then use these same k-core histories to discover a new metric, the “component seniority”, and we compare these values to the k-core shells. This metric is simply how long a given class has existed in the software system. This metric is completely independent of network structure, and so serves to independently validate the importance of the classes within the innermost core.

#### **10) Discuss Implications of Research**

Finally, we discuss the above results and their implications to our initial research goals. We find that the results appear promising, but that more work to validate the usefulness of the core in a real world development environment would be beneficial. Possible future directions this research can take are then detailed.

### III.2 Workflow Diagrams



*Figure 2: Workflow for Component Level Analysis*

In Figure 2, the high level flow of information is displayed. This would be the same flow for each software system version, with the initial JAR file input differing.

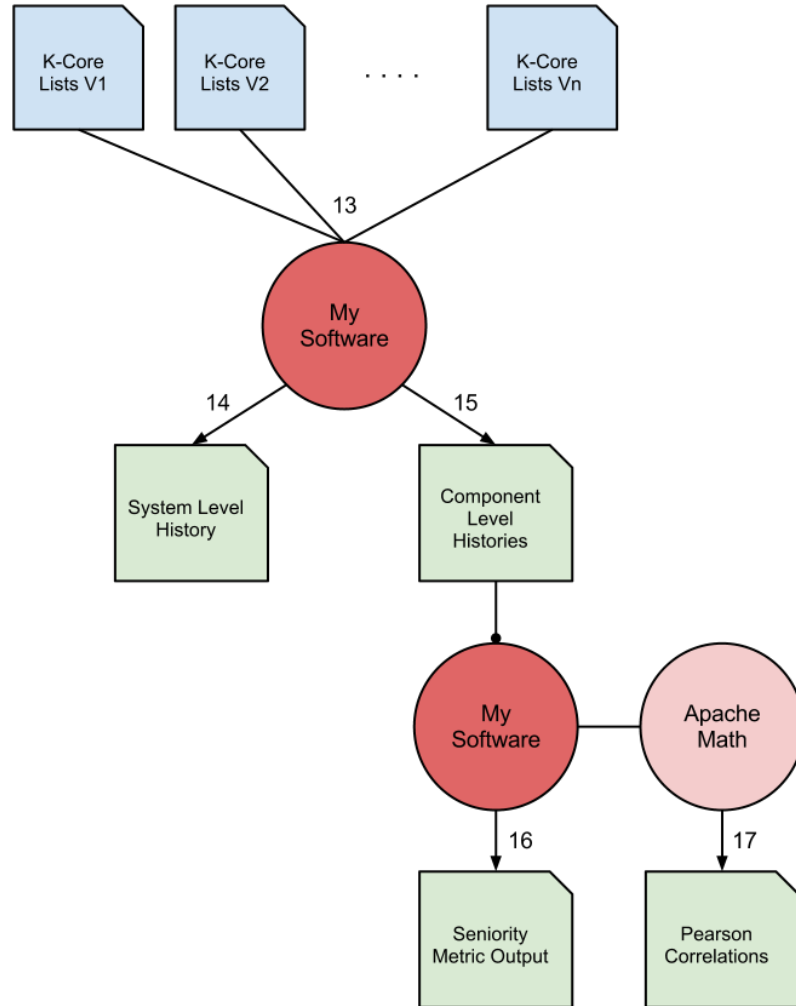
1. First, the *DependencyFinder* tool is used to output a text based representation of the software dependencies.
2. This text file is then inputted into my own program, which serves a number of functions, detailed below.
3. The first of these is to take these dependencies and represent them in a `Graph` object from the *JUNG* framework.
  - a) Each node is labeled from the fully qualified class name of the class.
  - b) Each dependency is an edge and is given a numeric identifier.
4. The software assigns each named class in the system a k-core value based on the k-core decomposition algorithm. This file is then used for analyzing the k-core evolution, both system and component level.
5. Each class is given a numeric label. A file is generated which represents each dependency as a single line, with the two numeric labels separated by a space. This file is then gzipped.
6. This file acts as the input to the *Lanet-VI* system. A portable network Figures (PNG) file is generated by the *Lanet-VI* system, visually representing the k-core structure of the system. The algorithm can be used through their website, or the

C++ source code can be downloaded and executed manually.

7. *JUNG* is used to calculate several centrality metrics, including Eigenvector Centrality, Closeness Centrality, and Degree. *Jung* provides implementations of these algorithms within the framework.
8. The `FRLayout` within *JUNG* is used to output the inner core of the system, with customization for nicer looking diagrams. Attraction and repulsion factors tweaked, and custom `Transformer` instances to render the nodes and edges appropriately for the scale of these software networks.
9. A text file is generated with a `.DAT` extension. This file is imported into *Matlab*, which is set up with the functions from the *Matlab Boost Graph Library* included.
10. These *Matlab BGL* functions are used to calculate the same centrality metrics listed under the *JUNG* steps, for validation of results. The Betweenness Centrality was also calculated with *Matlab BGL*, as the implementation of this algorithm is more efficient in this tool.
11. The dependencies in the network are written out in a simple text file, suitable for input into the publicly available C++ implementation of the Louvain method of community detection. This is just numbered nodes separated with spaces to

represent an edge.

12. The C++ algorithm for Louvain method of community detection is executed against this text input file. This algorithm was developed by Vincent Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. The algorithm outputs to a file that displays the hierarchical communities in a tree structure.



*Figure 3: Workflow of Aggregate Analysis Steps*

13. The results from step 4, the k-core outputs for each class, are inputted again to my software for the analysis of evolution over time. Each version of the examined system has its own input file of k-core values.

14. The system level evolution is captured by examining the maximum k-core for each inputted version and plotting over time.
15. The k-core value history is generated for each class. These histories are used for examining the component level evolution within the system.
16. The k-core histories for each component are used to generate the new “seniority” metric for each class in the system. Each shell in the final version has an average seniority score computed, based on the mean value for each class that ended up in that shell.
17. The *Apache Commons Math* library is used to compute the normalized Pearson correlation coefficients for the k-core histories of each class in the system. The top correlations are then checked for community or package prediction.

### III.3 Tools Used

DependencyFinder

<http://depfind.sourceforge.net/>

JUNG (Java Universal Network and Graph Library)

<http://jung.sourceforge.net/>

Lanet-VI

<http://lanet-vi.soic.indiana.edu/>

MATLAB

<http://www.mathworks.com/products/matlab/>

MATLAB BGL (Boost Graph Library)

<http://www.mathworks.com/matlabcentral/fileexchange/10922>

Apache Commons Math

<http://commons.apache.org/math/>

C++ Implementation of Louvain Method

<https://sites.google.com/site/findcommunities/>

## IV. Examined Software Systems

We decided that we would limit our initial research to three representative software systems. The number was kept manageably low in order to fully explore the evolution of these systems. All three systems are open source Java applications. Java is one of the premier commercial development language, and is within the expertise of our team. We selected systems that we have a familiarity to, to aid in us the interpretation of our results. Hibernate-ORM and Apache-Ant are projects used extensively by Phil Meyer, and JHotDraw had been studied in previous research by Dr. Siy and Dr. Bhowmick [Paymal, 2010].

### IV.1 *Hibernate-ORM*

The Hibernate framework is the premier object-relational mapping (ORM) solution in Java development. It was created by Gavin King, was eventually brought into the JBoss suite of application tools, and is now maintained by RedHat.

Most commercial Java systems have a relational database persistence data store and a middle tier of object oriented entity classes. In the past the developer would need to take the classes and manually write the code to generate the appropriate SQL to insert, update, delete, and queries based on these entities. This was an arduous and error-prone process.

The *entity bean* portion of the Enterprise Java Bean (EJB) specification was meant to facilitate this object/database interaction, but in practice the technology had many faults. The Hibernate framework was developed as an alternative to EJB

persistence, using a more light-weight solution that avoided many of the entity bean pitfalls. Over time the Hibernate framework was split into a few different independent projects, and it is the original object-relational mapping system that we study, now deemed “Hibernate-ORM”.

Of the three systems we examined for this research, this is the software system that grew the largest. The initial examined version (0.9.4) had a mere 136 named classes, while the final version (4.0.1.Final) had nearly 2500. We examine 13 different versions of the framework ranging from 2002 to 2012.

## ***IV.2 Apache-Ant***

The Apache Ant library is a tool for developers to build Java projects from the source code to the executables. Here “Ant” is an acronym for “Another Neat Tool”. The application was developed by the Apache Software Foundation, a non-profit corporation made up of a decentralized community of software developers. The foundation maintains dozens of open source software projects, and Ant is one of its most popular tools.

Apache Ant bases the Java build process around tasks and targets, typically defined in an XML file. This product was designed as an alternative and improvement over C++ Make files.

Example tasks would include ones that compile your code (javac) or bundle it up in a jar file for distribution. A typical build script will clean the existing build artifacts, compile the Java code, package the binaries into one or more java archive files, and then copy these out to a deploy directory. Ant is designed in a way that this is easily automated

and can be fired off automatically. Individual tasks within the build file can be run individually, or in sequence. Each task within the ant build file has its dependent tasks defined, running them in sequence where needed.

While still in heavy use by the Java community, more modern build tools like Apache-Maven have gained a strong following and are starting to become the preferred tools for this purpose. We examined 13 versions of the Apache-Ant project released over ten years, from 2000 to 2010.

### ***IV.3 JHotDraw***

The JHotDraw framework is another open source Java framework. This project intends to provide a well-designed and extensible framework that one can use for creating structured drawing editors. Appropriate contexts for using JHotDraw would include building an application for creating Pert diagrams or network layout graphs.

The framework provides a basic GUI editor with support for the most necessary features of a drawing editor, including undo, save, load and print functions. The users then extend this functionality for creating an editor specific for their needs.

JHotDraw is interesting from a design perspective in that it was consciously used for exploring design patterns, and was influential in the development of this object oriented design discipline. The framework was originally written in Smalltalk (and called HotDraw) by the University of Illinois in Urbana-Champaign. It has since become an open source project hosted by SourceForge.

We first examine version 5.2 after it was ported to Java in 2001. The JHotDraw

project underwent a major rewrite with the release of JHotDraw 7. In our research we examine 14 versions in total, with the most recent released in 2011.

## V. Dependency Extraction

A critical element to this research was the gathering of inter-class dependencies from the examined software systems. The output of any algorithm is only as good as its inputs, and core identification algorithms are no different. Despite the importance on good inputs, other related research on examining the network structure of a software system appears to take for granted the accuracy of the output of their dependency extraction tool.

In our research one crucial determination was in defining what constitutes a “component” in a Java application. The individual classes of the system was a likely choice, but one could also choose to look for core packages, as was done in other research [Holme 2005]. We chose to consider *named* Java classes as the discrete components to examine.

Several tools were considered for the purpose of dependency extraction for our research, including *Soot*, *Doxygen*, *JDepend*, *DependencyFinder*, and *SPARS-J*. The outputs and features were compared and analyzed for validity. In general, the extracted dependencies matched closely across tools. However, some small portion of the extracted relationships differed, representing the corner cases of Object Oriented design. In the end we chose the *DependencyFinder* tool, in part based on the choices made in these cases.

### Notes on *DependencyFinder*:

- Developed by Jean Tessier, a former software engineer for *Google* and *LinkedIn*.
- Parses the compiler generated .class files, which it can also read from JAR

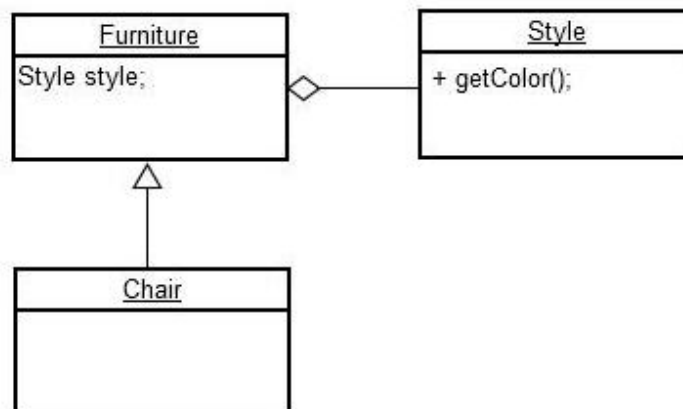
or ZIP files.

- This allows the tool to generate the dependency output from the released binaries and does not require the full source code.
- Outputs in XML and plain text format.
- Executes from command line, or from an included Swing based GUI for ease of use.

Here we detail the most common gray areas of dependency identification, and detail the choices that were made by the *DependencyFinder* tool:

#### ***V.1 Does a class have a direct dependency on the visible members of a superclass?***

Here we have one class that extends a parent class, and inherits the members of that superclass.



*Figure 4: Inherited Members*

In this example, **Chair** inherits access to the `style` class instance variable, which it may or may not use. It is clear that **Furniture** directly depends on the **Style**

class, but does `Chair`?

In *DependencyFinder*, the determination is made based off whether the subclass actually uses the visible member of the superclass. In the above example, if anywhere in the `Chair` class a method `style.getColor()` is called, for example, then the dependency is captured. If the superclass member is ignored completely, the `Chair` class is not considered dependent. Some dependency extraction tools examined (*SPARS-J*) include this as a dependency whether it uses the inherited members or not.

As inheritance relationships are heavily used in object oriented software systems, this decision can have wide ranging consequences. After careful consideration this choice was deemed to be preferred over the possible alternatives. As *DependencyFinder* works by parsing the `.class` files generated from compiling the Java code, its likely that a dependency on a superclass member only exists in the `.class` files if it is referenced somewhere within the `.java` file.

## ***V.2 Does an inner class depend on the members of the outer class.***

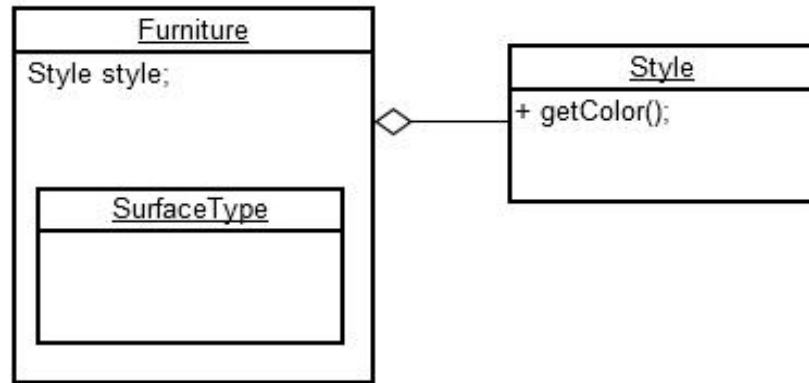


Figure 5: Visible Outer-Class Members

This question is related to the previous one, but it does not necessarily have the same answer. Technically, an inner class has access to the class members of the outer class, even though it does not inherit from the outer class. In this example, does `Furniture$SurfaceType` have a dependency on `Style`?

Again, *DependencyFinder* looks at whether the inner class actually uses the member that it has access to. In the above example, the dependency is found only if the `style` member is referenced directly within the inner class. We similarly find this decision to be the preferred resolution.

**V.3 Does a class directly depend on each of its ancestors in the class tree?**

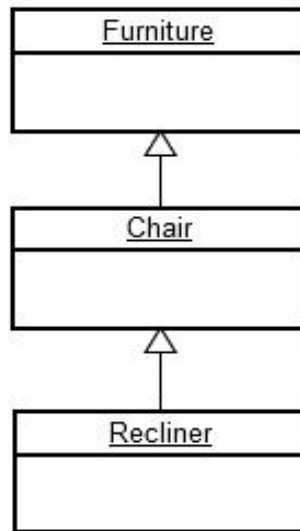


Figure 6: Indirect Ancestors

Here the question is whether a class depends directly on the parent of its parent, and so on. Similarly, does a class depend on the interfaces of its parent class? In the above example, the question is whether `Recliner` has a direct dependency on `Furniture`.

The *DependencyFinder* tool does not consider this to be a direct relationship, though at least one other tool (*SPARS-J*) does. This holds true for inherited interface declarations as well.

#### ***V.4 Do dependencies on anonymously passed method parameters get captured?***

This example involves a class passing a returned object directly into a method of another class, without this object being referenced anywhere as a variable. This is a scenario that is not easily visualized without code to reference, so a simple constructed

example is provided below. This example contains four classes, named A, B, C and D.

```
public class A {
    public void doTest() {
        B b = new B();
        C c = new C();
        c.use(b.get());
    }
}

public class B {
    public D get() {
        return new D();
    }
}

public class C {
    public void use(D d) {
        System.out.println(d);
    }
}

public class D {}
```

*Text 1: Anonymously passed method parameters*

In the above code segment, class A never directly references class D, but it does depend on it, as it is returned from `b.get()`, and passed into `c.use()`. To see the dependency exists, consider that the code in `doTest` could be re-factored as below, with no change in code behavior:

```

public void doTest() {
    B b = new B();
    C c = new C();
    D d = b.get();
    c.use(d);
}

```

*Text 2: Alternative without anonymity*

*DependencyFinder* does capture this dependency faithfully, though at least one other examined tool did not (*Soot*).

#### ***V.5 Are inner classes considered their own component or are they part of the outer class?***

Java allows inner classes defined within an outer class, and also lets you define a class anonymously in your code, so these also needed to be considered.

```

public class OuterClass {
    public void doSomething(){
        Runnable runnable = new Runnable(){
            public void run() {
                System.out.println("Running!");
            }
        };
        runnable.run();
    }
}

```

*Text 3: Anonymous Inner Class Example*

In the above example, there is an inner class which implements `Runnable` that is

defined within the same method in which the code is executed. The class was constructed in a way such that it has no name, and so cannot be referred to by another class within the software system. One could construct an example where this anonymous class could be passed into a method of an unrelated class to `OuterClass` and executed. However, to do so it would have to be a polymorphic reference to the `Runnable` interface, and the executing code would not know the name of the class it depends on at compile time.

```
public class OuterClass {  
    public void doSomething(){  
        Runnable runnable = new NamedInnerClass();  
        runnable.run();  
    }  
    class NamedInnerClass implements Runnable{  
        public void run() {  
            System.out.println("Running!");  
        }  
    }  
}
```

*Text 4: Named Inner Class Example*

In contrast, here is a named inner class that represents the same behavior as the anonymous inner class. However, as the above class is defined formally with a name, any unrelated class in the software system that has the appropriate permissions could access this class. Here then the dependency is known at compile time.

*DependencyFinder* outputs named inner classes like with the outer class, a dollar sign, then the inner class (`OuterClass$InnerClass`). Anonymous inner classes are given a number, corresponding to the order they are defined in the outer class

(OuterClass\$1).

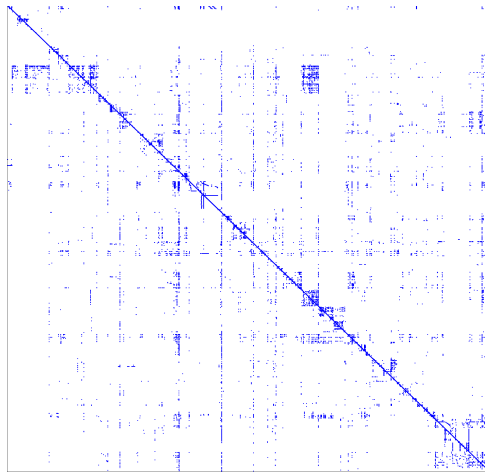
We decided to treat *named* inner classes as their own individual components from the outer class. However, the dependencies of an *anonymous* inner class were rolled into that of the class which defines them.

The reasons for this were because of (1) the impossibility of having a compile time dependency, and also because of (2) the nature of how these classes are typically used. When you use an anonymous inner class, you expect to execute the code of the class immediately. They are transitory in nature.

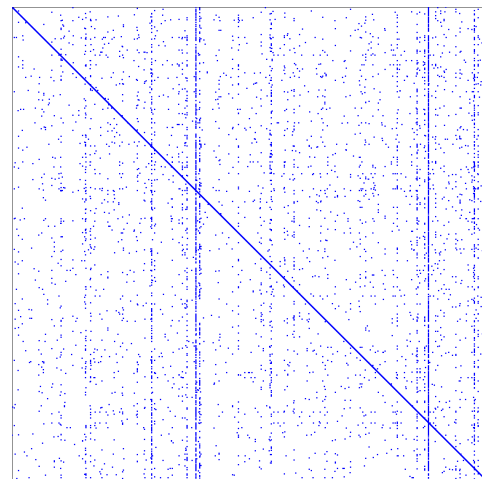
## VI. Reproducing MacCormack

Our first approach for finding the core of a software system was to reproduce the MacCormack algorithm. We applied this algorithm to selected versions of Hibernate-ORM and Apache-Ant, and JHotDraw, and then examined the results

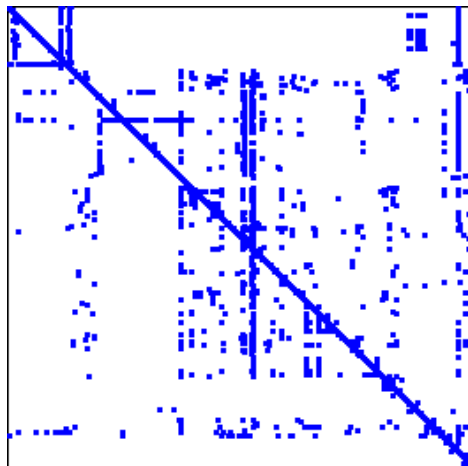
Here we extract the dependencies from the source code, representing them in adjacency matrices. These are not to scale, as the system sizes vary greatly:



*Figure 7: Hibernate-ORM 4.0.0.CR4*



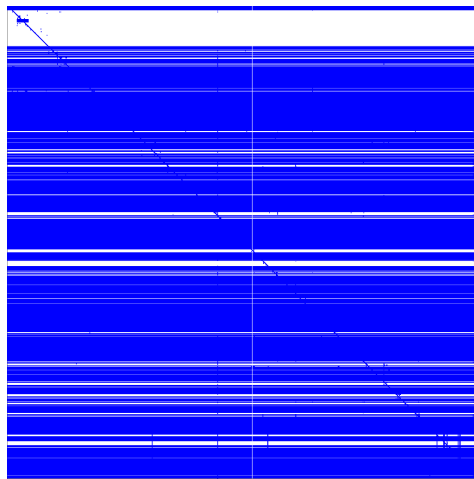
*Figure 8: Apache-Ant 1.8.0*



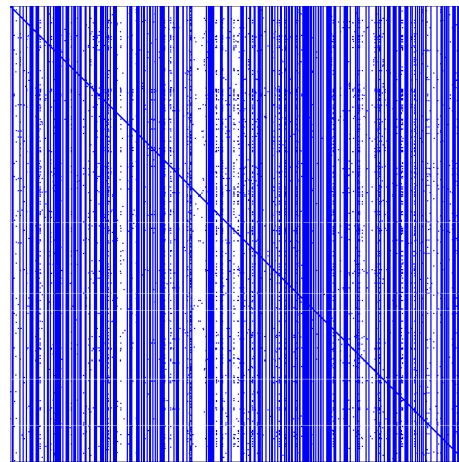
*Figure 9: JHotDraw 7.6*

In the above Figures, the X and Y axis are the classes within the systems, with the blue squares representing a dependency between classes. Here the diagonals relate to each class being dependent on itself.

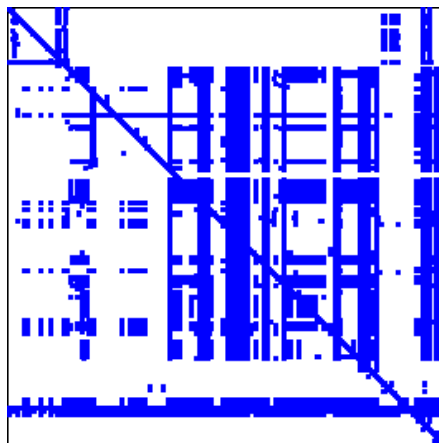
We then apply matrix multiplications to determine the transitive dependencies, with each successive multiplication representing one step of indirection. Each of these matrices is then summed together to form a final matrix that represents the transitive dependencies. The following graphs show the results of this process.



*Figure 10: Hibernate-ORM Transitive*



*Figure 11: Apache-Ant Transitive*



*Figure 12: JHotDraw Transitive*

What we see with these transitive graphs are some stark differences in the final summed matrices. Hibernate-ORM has primarily horizontal lines, indicating that most classes transitively “depend on almost everything”. This is explained later on in this section with the logging implementation in this system. Apache-Ant has a fair number of classes for whom “almost everyone depends on” transitively, such as the `Project` and `Task` classes, leading to visible vertical lines. Finally JHotDraw is a more sparse, indicating that many dependencies remain local even when expanded out, and don't end up leading to dependence on the whole system.

We then grouped each class into the four classification buckets (Core, Peripheral, Shared, Control) based on the calculate values. The results of this classification are shown in the tables below:

<b>Hibernate-ORM</b>		
<i>Classification</i>	<i>Number of Classes</i>	<i>Percent of System</i>
Core	1121	62.0%
Peripheral	10	0.6%
Shared	657	37.0%
Control	4	0.20%

<b>Apache-Ant</b>		
<i>Classification</i>	<i>Number of Classes</i>	<i>Percent of System</i>
Core	0	0.0%
Peripheral	893	79.60%
Shared	230	20.4%
Control	0	0.00%

<b>JHotDraw</b>		
<i>Classification</i>	<i>Number of Classes</i>	<i>Percent of System</i>
Core	0	0.0%
Peripheral	203	77.0%
Shared	33	13.0%
Control	28	11.0%

*Table 2: MacCormack Results on Examined Systems*

The results were not encouraging for this being a practical and useful method of identifying a core set of components. Of the three examined systems, only one had a core at all, and this was too large to be of any practical use.

We further examined the identified *core* of the Hibernate system to determine what set these classes apart from the classes put in the *shared* bucket. What we found cast further doubt on this method being worthwhile for identifying core components. In this system exists a logging class called

`org.hibernate.internal.CoreMessageLogger`. The class has a dependency on many other classes within the system, as it performs logging specific to the types of objects passed into the methods.

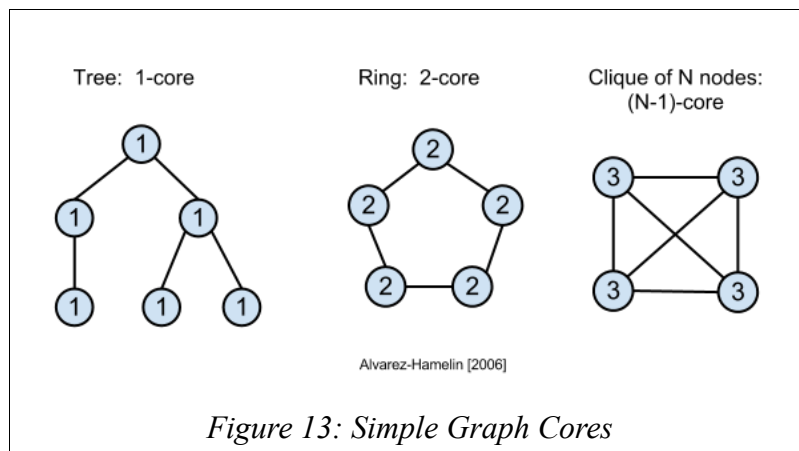
This logger is used throughout the Hibernate- system, so whichever classes used this logger also gained all the outbound transitive dependencies of the logger, which is the majority of the software system. In this way nearly every class is dependent on every other class transitively. This explains the mostly solid blue transitive dependency Figure. Whether a class ends up in the core of the Hibernate system is largely dependent on whether it uses this `CoreMessageLogger`. This is hardly a useful determiner of whether a class is important or not.

So this algorithm is also too brittle to small design decisions to be of any practical value. It appears that the MacCormack team spent too much time drawing conclusions from their aggregated results and not enough time examining the output of any single software system for validity.

## VII. K-Core Decomposition

Based on the influence of the Petter Holme research, the next approach we looked at for identifying the core of a software system was *k-core decomposition* [Seidman, 1983]. In this approach, an algorithm breaks a network down into a series of shells based on the inter-connectivity of the vertices. Applied to our research context, the innermost shell is then considered the core of the software system.

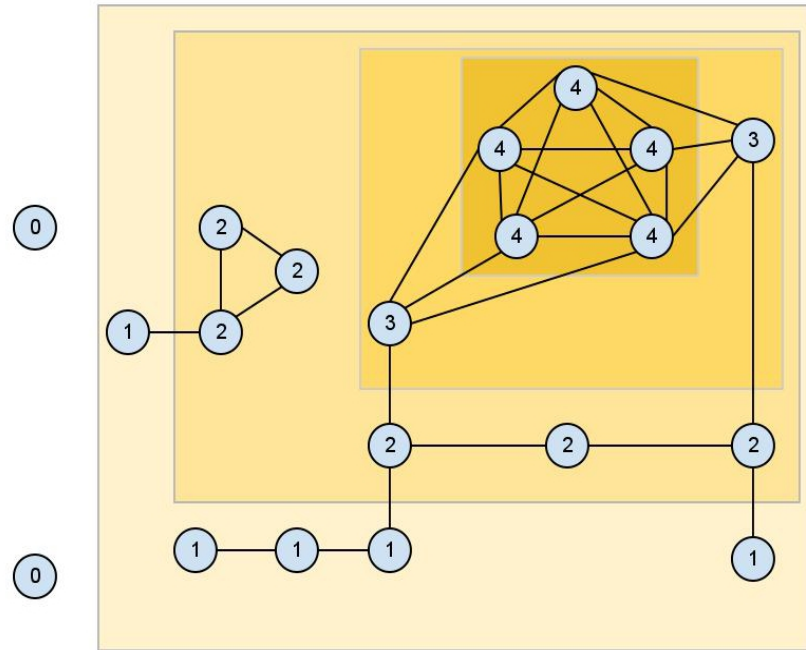
Formal definitions and implementation details of this algorithm can be found in [Batagelj and Zaversnik, 2002] and [Hamelin et al, 2006]. The intuitive definition of the *k*-core algorithm is fairly simple: Given undirected graph, recursively remove vertices with degree less than *k*. The resulting graph has all vertices with degree of at least *k*. The *maximal k*-core is the highest *k*-core that is not empty. The (*k*+1)-core would be empty.



Above we see some simple networks, and how they relate to a *k*-core score. In the tree example, each node is only connected to one other node, while a ring has each node connected to two others. A *clique*, defined as a sub-graph where each node is

connected to each other node, corresponds to a  $k$ -core of  $N$  minus 1. Here below a 4-clique is in fact a 3-core:

The example below shows the decomposition of a simple network, with the algorithm finding a series of  $k$ -cores like the layers of an onion. In the first pass, the 0-core is determined by removing any node not connected to any others. Then in the next pass any node connected to only one other node is removed, becoming the 1-core.



*Figure 14: Example K-Core Decomposition*

Here we see that while there existed nodes in the 1-core with degree of two that got included in the 1-core. This is because after the algorithm removes a node, it considers the network again, and if your neighbor was removed it doesn't count toward

your degree. In this way a series of nodes that are only connected to low degree nodes can be removed in series. It isn't enough to be connected to two other nodes, the nodes you connect to must themselves be connected to two other nodes as well.

This concept is used elsewhere in the graph theory literature as *degeneracy* or *coloring-numbers*. A classic example of the latter is when coloring a map, and choosing a number of colors such that no two connected countries will get the same color. This can be determined by calculating the  $k$ -cores of a graph that represents the map, and then the number of colors needed is simply one more than the maximal  $k$  value. In the above  $k$ -core example, only five unique colors would be needed for a map with nations representing the nodes in the graph.

One advantage of this decomposition algorithm over others is that it is very fast, with  $O(n)$  linear execution time. This is much faster than many other network metrics, such as betweenness-centrality.

There is the potential for the inner core of a network to be in fact two or more separate and unconnected cores. However, in examining the results of our software systems, each maximal core was one interconnected sub-graph.

#### Advantages of K-Core Decomposition:

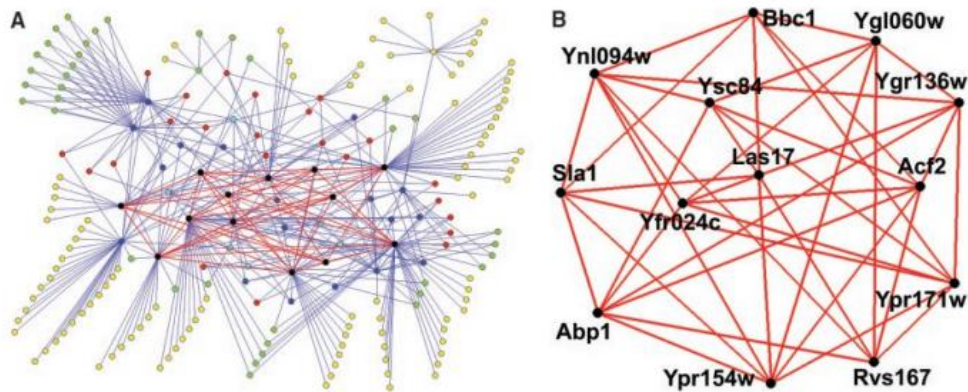
- Fast (Linear run-time)
- Intuitive algorithm
- Concrete graph theory foundations
- Meets formal definition of coreness
- Cores are of manageable size in respect to system size
- All systems will have a core
- Proven useful in other research fields

Due to favorable initial experimental results, the focus of our research shifted from finding a suitable algorithm to instead investigating the k-core approach for finding cores, and its implications for software systems.

## VII.1 K-Cores in Other Research Contexts

As mentioned above, a primary reason for giving the k-core decomposition an extended look for our purposes is that this network analysis tool has proven useful in other research contexts.

The most prominent example of this is the usage of the algorithm for the analysis of protein interaction networks. This research was initially done by Amy Hin Yan Tong et al 2002, by the University of Toronto.



*Figure 15: Protein Interaction Network K-Cores*

In this Figure, the protein interaction network is modeled on the left A. The colors of the nodes represent the k-core levels (6-core, black; 5-core, cyan; 4-core, blue; 3-core, red; 2-core, green; 1-core, yellow). The interactions among the maximal 6-core are drawn in red. This 6-core is then segregated out and represented on the right hand side as B.

This research was expanded upon in [Bader and Houge, 2003] and other later works. The results are clearly favorable for the suitability of k-core decomposition in identifying cores in this research context.

The other research in which k-core decomposition has received a lot of attention is in analyzing the connectivity of the internet. Examples of this can be found in [Gaertler and Patrignani, 2003] as well as [Carmi et al, 2007] and other works. In this context it is the various routers which handle requests that are modeled, and then examined by using the k-core algorithm to break the network into a hierarchy of shells.

## VIII. Visualization of K-Cores

Another advantage of k-core decomposition is that there exists a very nice visualization algorithm that uses this technique to render a network Figureally. This visualization algorithm is called Lanet-VI (**L**arge **N**etwork-**V**isualization). It was developed jointly by the Université of Paris-Sud, Indiana University, and CNRS. The algorithm represents a few different aspects of network structure within the generated image:

Degree: The size of the nodes in the image correlate to the *degree* of the node, based on a logarithmic scale.

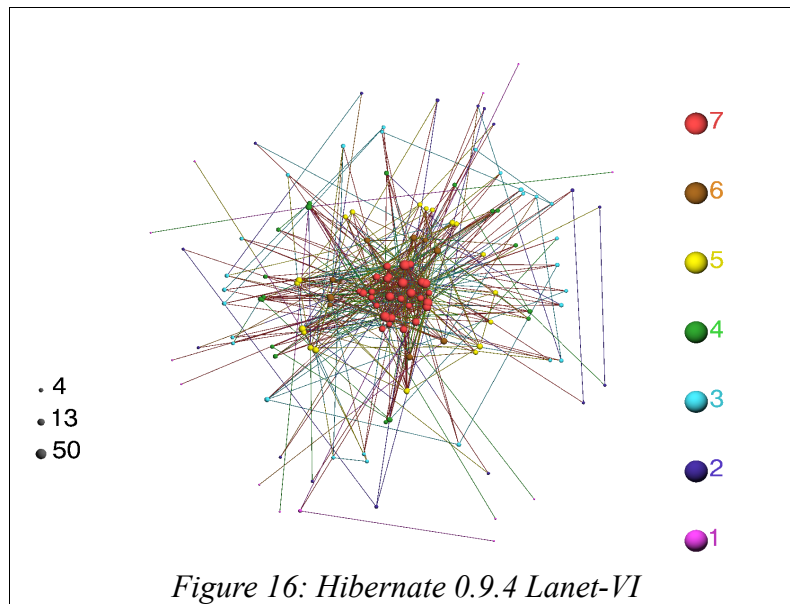
K-Core value: The color of the node, as well as the position of that node in the network, correlate to that node's k-core value. The nodes closer to the center of the graph are the nodes with the highest k-core value in that graph. The k-core values are also visually identifiable based on their position on the color spectrum between blue and red. The more red colored nodes have higher k-core values.

Clustering: The algorithm attempts to position nodes close together which are connected to each-other and exist within the same shell.

Connectivity: Some portion of the edges within the network are also rendered. By default, an optimal percentage of edges is determined, and then which edges are displayed is randomly determined. This helps visualize the dense connections of the inner shells, while not overwhelming the viewer with a mess of thousands of visible edges. This factor can also be set manually, specifying what percentage of edges to select, between 0% and 100%. Also, nodes within a shell that have more connections to the nodes of the inner shells are rendered slightly more inward than nodes only connected to outer nodes.

In addition to using the Lanet-VI tool to represent the system as a whole, we also drill down and create a visual representation of the core classes alone, with all peripheral

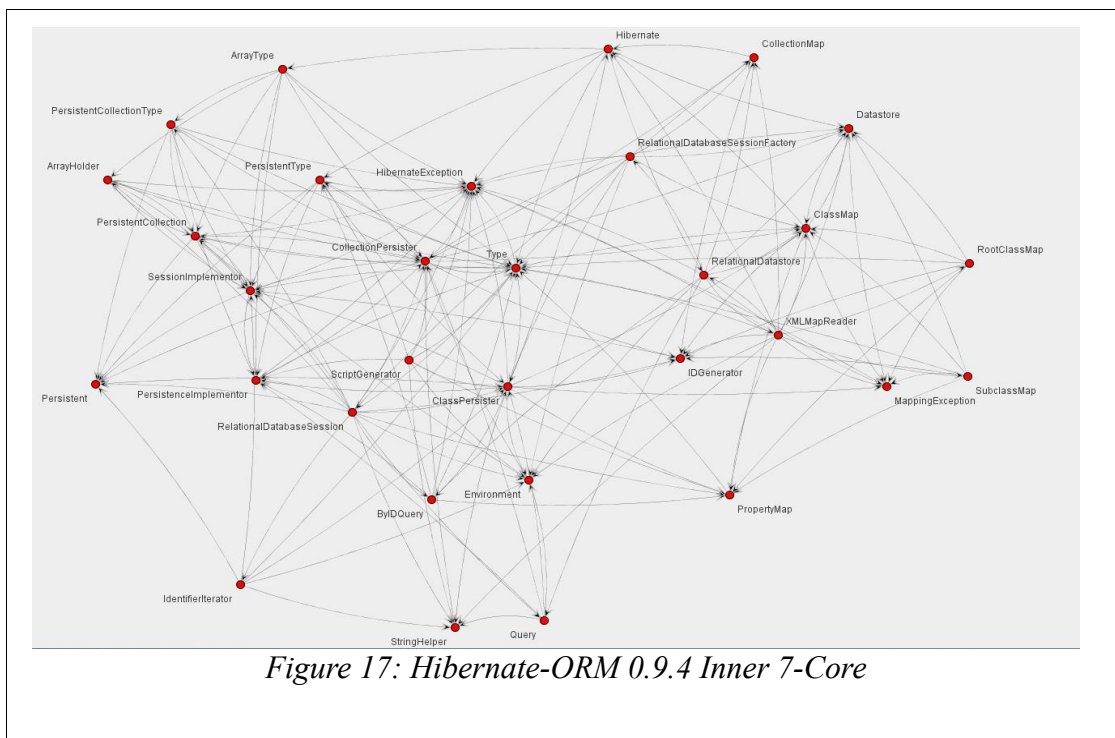
nodes and edges pruned. The nodes are arranged with the Fruchterman-Reingold force-directed algorithm for node layout using the JUNG framework.



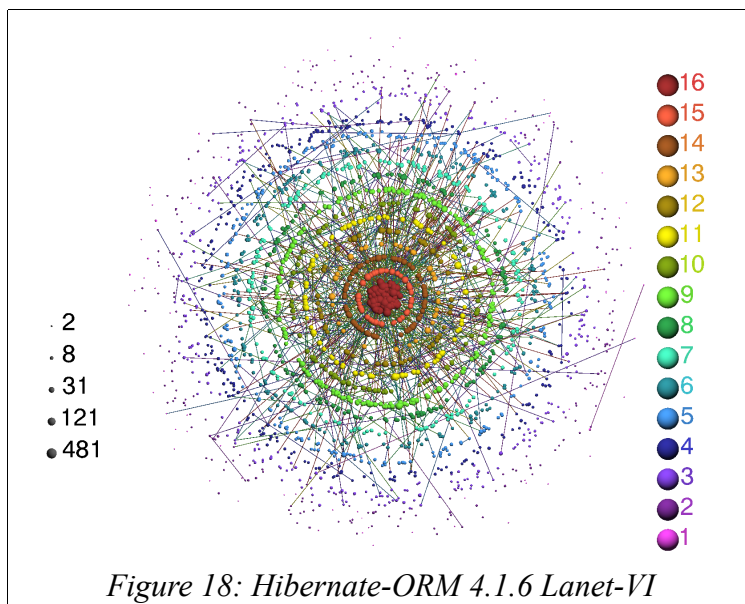
To start with, we examine one of the smallest software systems we studied, the initial release of Hibernate-ORM, 0.9.4.Beta. In the Lanet-VI visualization above, we see that the largest nodes have a degree of about 50. The maximal k-core is the 7-core in this instance, shown in red at the center of the Figure. With only 136 named classes in total, this system is fairly simple to study. Unlike later systems with more classes, the orbital shells are less clearly delineated, and most connections are faithfully rendered.

Then we break out this inner out and represent it with the Fruchterman-Reingold layout. This algorithm arranges nodes based on an attraction multiplier, a repulsion multiplier, and the number of iterations with which to run before stopping. In Figure 17

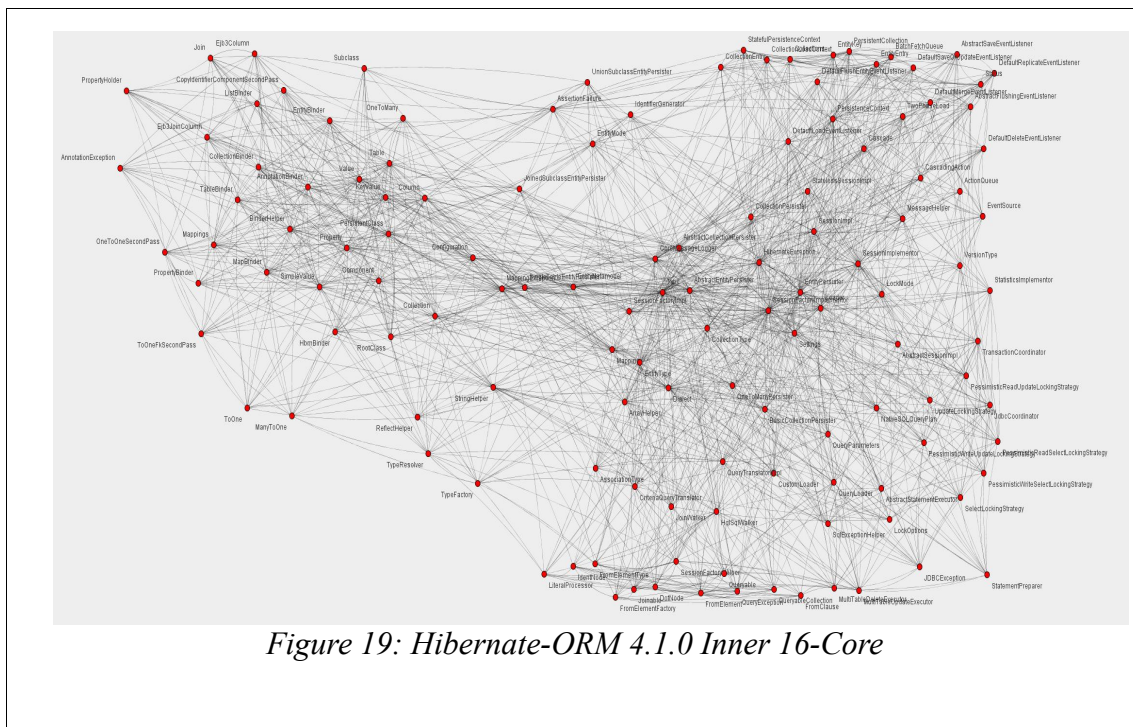
we see an easy to read graph, with the most central classes (`Type`, `HibernateException`, etc) representing the most connected core classes.



Next we will examine the largest software network we studied, the final version of the Hibernate-ORM framework, 4.1.6. Here we have nearly 2500 named classes to represent. The number of nodes represented in the same space gives a Figure with clearly arranged shell structure. The inner 16-core is the densely packed inner red cluster.

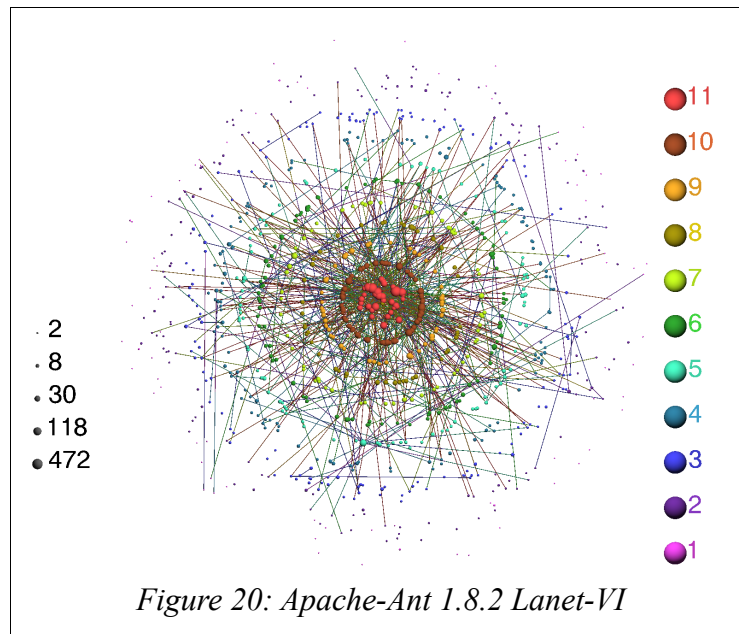


We break out this inner core to examine in Figure 19 below. As this core has 131 nodes, it is densely displayed by the FR-Layout we are using. When showing this sub-graph on a full page with ability to zoom, the information is more clearly captured.



We see that the most central classes are often also the central classes of the 0.9.4.Beta release. We examine this evolution further in section XI.

In the next visualization diagram, we see that the Apache-Ant system has fewer classes, and appears to have less dense organizational structure than the later version of Hibernate-ORM. The maximal core is an 11-core, and the largest nodes have a degree similar to that of the Hibernate-ORM visualization.

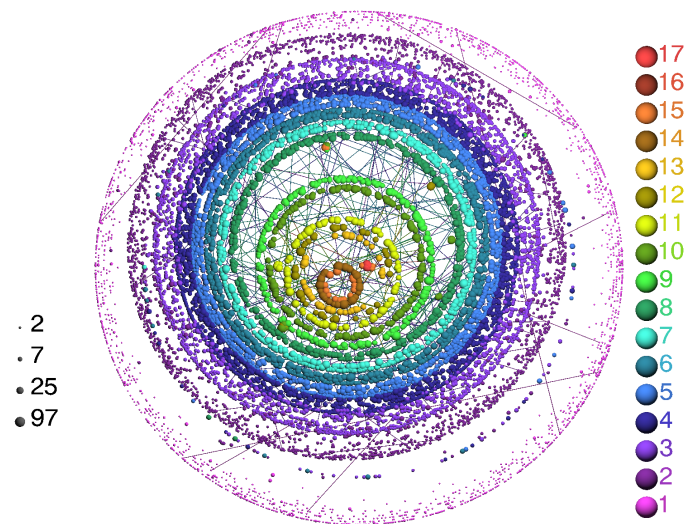


The maximal 11-core shown below is composed of only 64 nodes. This allows us to clearly see that the most central nodes within the core are the `Project` and `BuildException` classes. Like `HibernateException`, the `BuildException` is used extensively throughout the system and so has a huge number of connections. Seeing the core segregated out into its own network of classes, we can easily see just how tightly connected these software cores are.

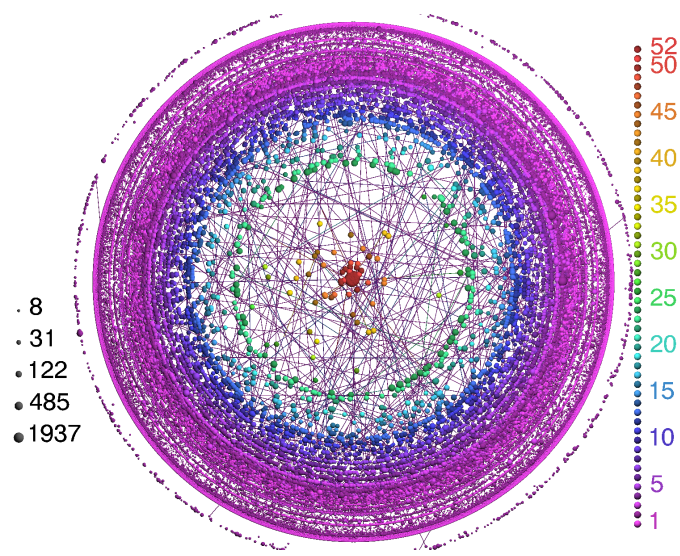
Figure 22: JHotDraw 7.6 Lanet-VI



Each software system appears to be similar in general structure by the Lanet-VI algorithm, as one might expect from their similar object oriented hierarchical structure. One can view non-software network visualizations on the Lanet-VI website to see just how different other network types can look.



*Figure 24: Scientific Collaboration*



*Figure 25: Internet Routers*

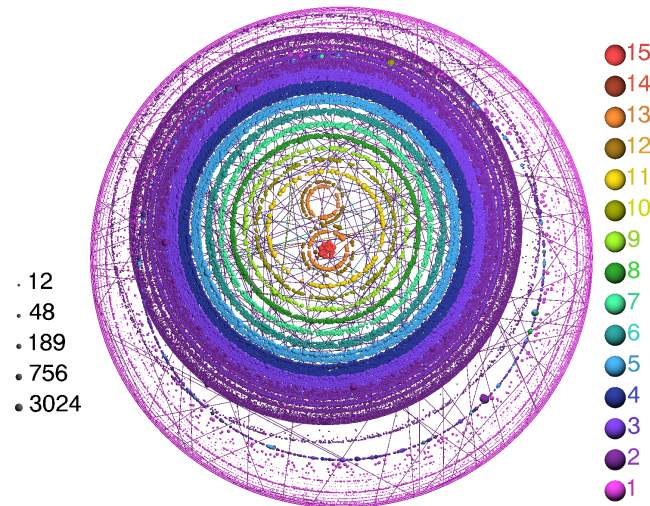


Figure 26: Web Page Links

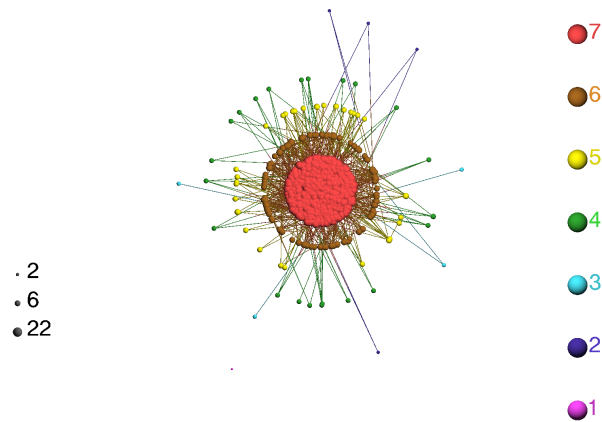


Figure 27: Random Graph

Here we see visualizations of scientific collaboration, internet routing, links on web pages, and a random graph generated with the Erdős-Rényi model. One interesting thing to note here is that while the World Wide Web network has millions of nodes, it still has a maximal coreness lower than the 16-core of Hibernate-ORM. We also see that some networks have multiple cores, while our software networks only have one core. The similarity between Java software system networks is itself a noteworthy feature.

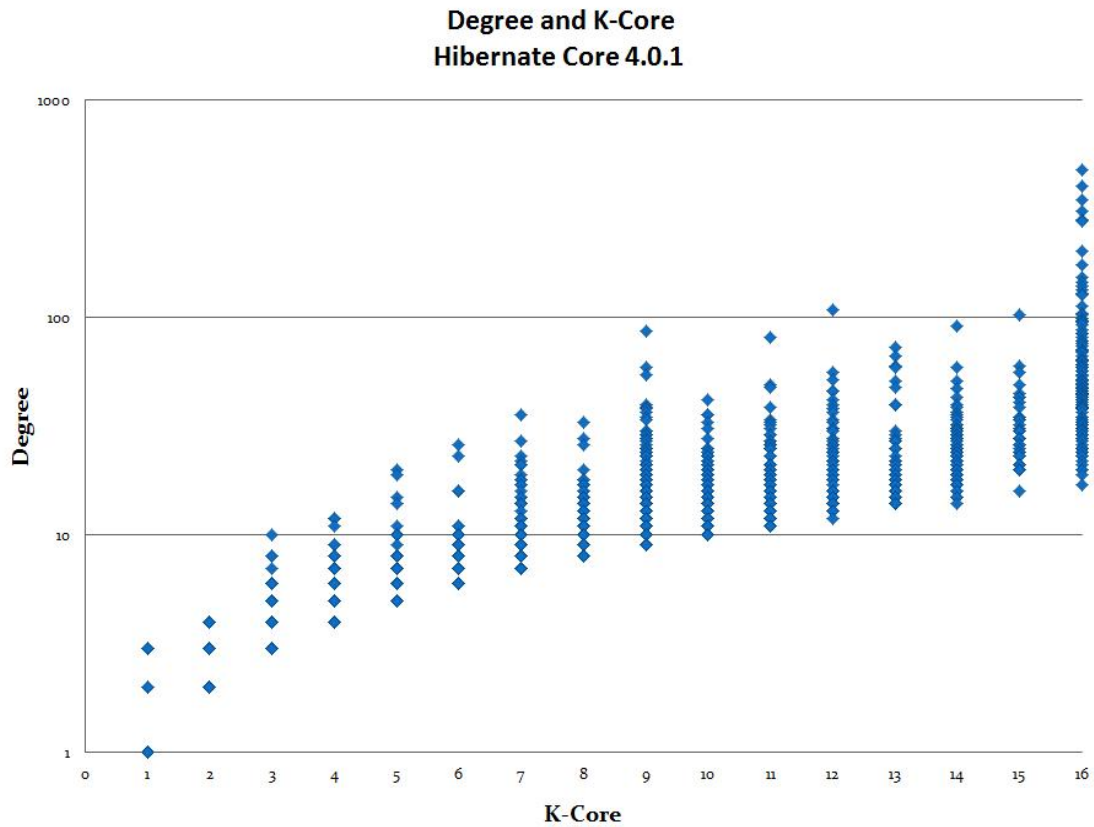
## IX. K-Core and Centrality

In the very early stages of our research, our initial concept of coreness was indistinguishable from the concept of centrality. The Borgatti and Everett paper first explained the relationship between the two measures. A coreness metric is a measure of centrality, but a centrality metric is not necessarily a measure of coreness. The core of a system cannot merely be composed of central nodes, but nodes which are also themselves tightly connected to each-other. In this section we explore the relationship between k-core degeneracy and three different centrality metrics: degree, closeness centrality, and betweenness centrality.

This analysis was primarily done with data outputted by algorithms contained within the Java Universal Network/Graph Framework (JUNG), based on the extracted class dependencies. JUNG is an open source Java framework developed to provide an easy to use API for graph and network programming. Also used was the Matlab Boost Graph Library (BoostGL) package, which provides a number of functions for graph and network algorithms.

### IX.1 Degree vs K-Core

Simply put, *degree* of a node is the number of other nodes that a node connects to. In the context of our research, this is the total number of classes which depend on the examined class, or which the class itself depends on. We plotted the degree and k-core levels for the latest versions of each of the software systems.

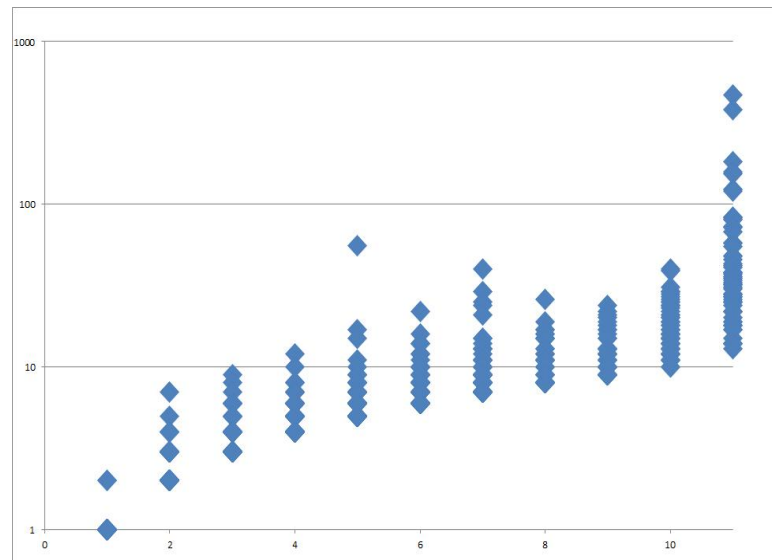


*Figure 28: Degree and K-Core Hibernate-ORM*

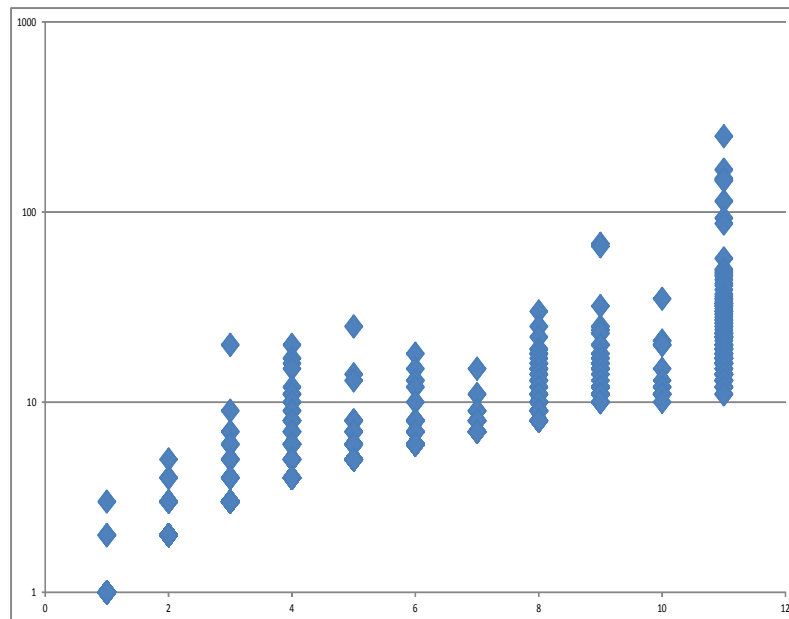
The above scatter plot demonstrates the relationship between degree and k-core degeneracy. The y-axis is logarithmic scaled, and you can clearly see that every node with degree of at least 150 is in the maximal k-core shell. This also shows the obvious lower-bound such that a 16-core contains classes with a degree of at least 16, as each class needs to be connected to at least 16 others.

Classes with a relatively high degree and only mid level k-core, like the outliers in the 9, 11 and 12 cores, would represent classes that have a lot of connections, but that these connections are to classes which are not themselves well connected. These would be classes with more local importance than global importance to the software system.

The plots for the latest version of Apache-Ant and JHotDraw look very similar, and are shown below. The trend seems to be that the maximal k-core contains the nodes with very high degree, while a handful of nodes with only moderately high degree are outliers.



*Figure 29: Apache-Ant Degree*



*Figure 30: JHotDraw Degree*

## IX.2 Closeness Centrality vs K-Core

The next centrality measure that we examined was closeness centrality. Closeness Centrality is defined as the total distance between a particular node in the system and all other nodes in that system. This involves, as a first step, calculating the shortest paths in the system.

$$C_C(v) = \sum_{t \in V \setminus v} 2^{-d_G(v,t)}.$$

In this formal definition,  $v$  and  $t$  are two vertices in the set of vertices  $V$ . Here then  $d_G(v,t)$  is the distance between the two nodes, as determined by the shortest path. It is a simple and intuitive measure of network centrality.

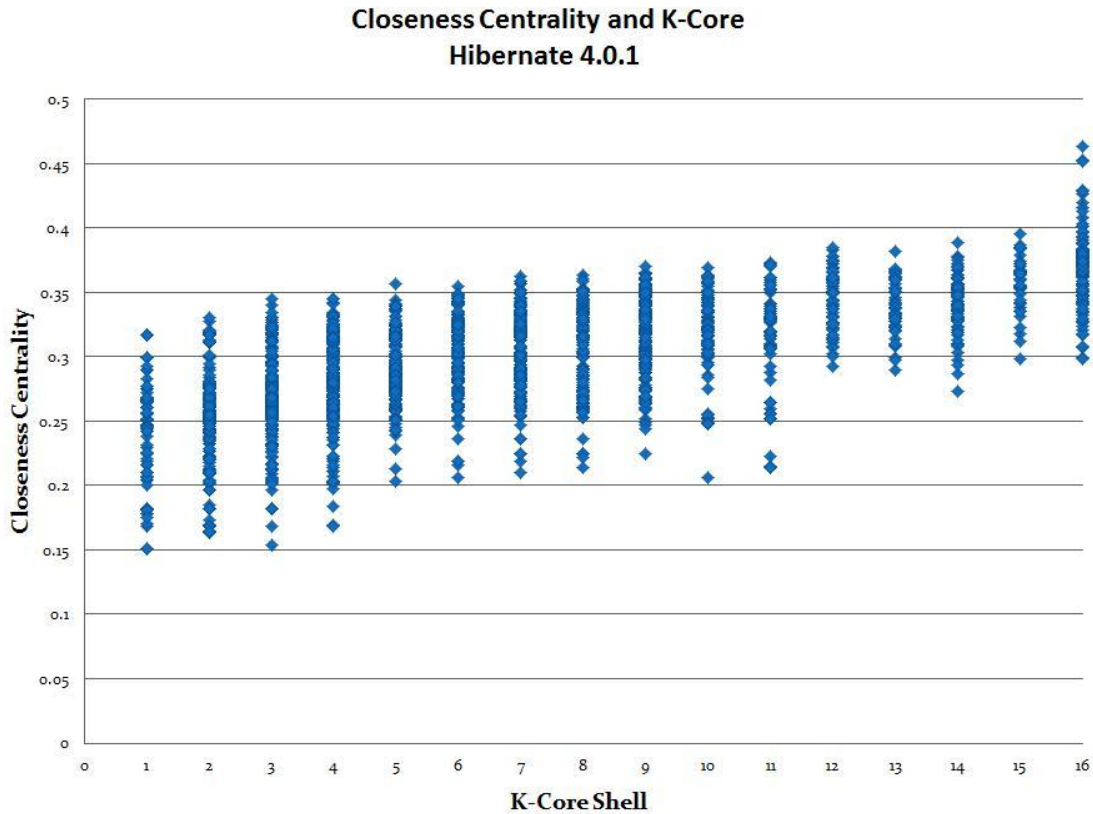


Figure 31: Hibernate-ORM Closeness Centrality

We plot the Closeness Centrality and the k-core shells in the Figure above. This plot shows a similar relationship as we discovered for the *degree*. This makes intuitive sense, as classes with a high degree will tend to be more central than those with a lower degree. The relationship here again seems to be a clear lower bound on the centrality of the maximal k-core, along with each class with an exceptionally high centrality score belonging to the 16-core.

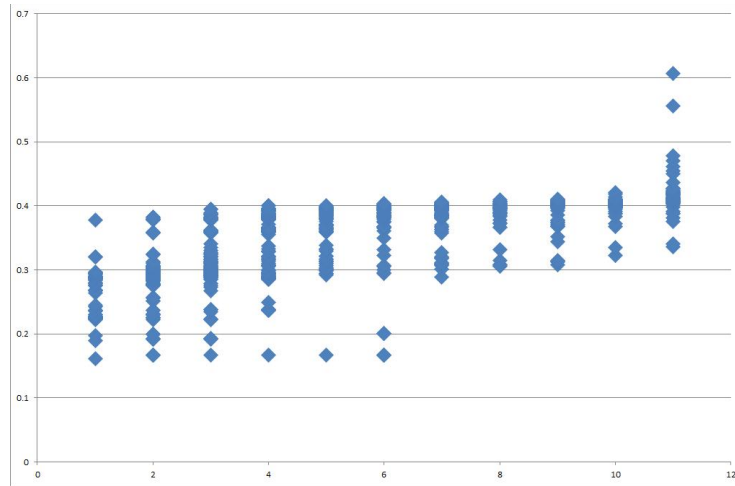


Figure 32: Apache-Ant Closeness Centrality

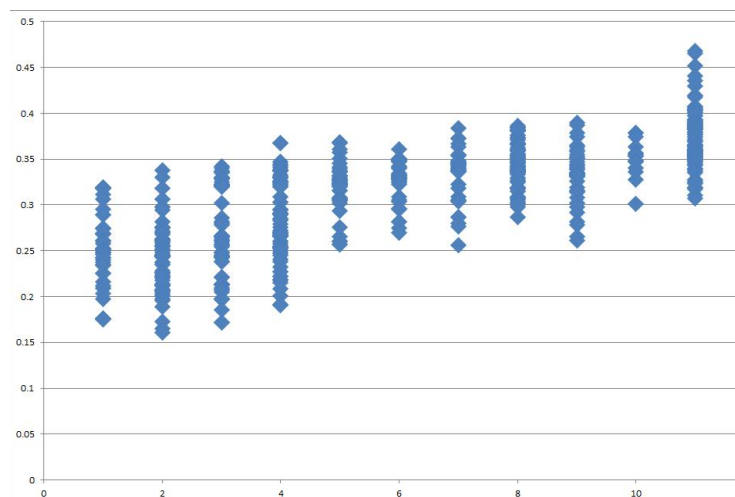


Figure 33: JHotDraw Closeness Centrality

The above plots again show that for the latest versions of the other two software products, the general pattern matches very closely. Apache-Ant is has fewer obvious outliers, and JHotDraw appears to have a more obvious concentration of high centrality in the maximal k-core.

### IX.3 Betweenness Centrality vs K-Core

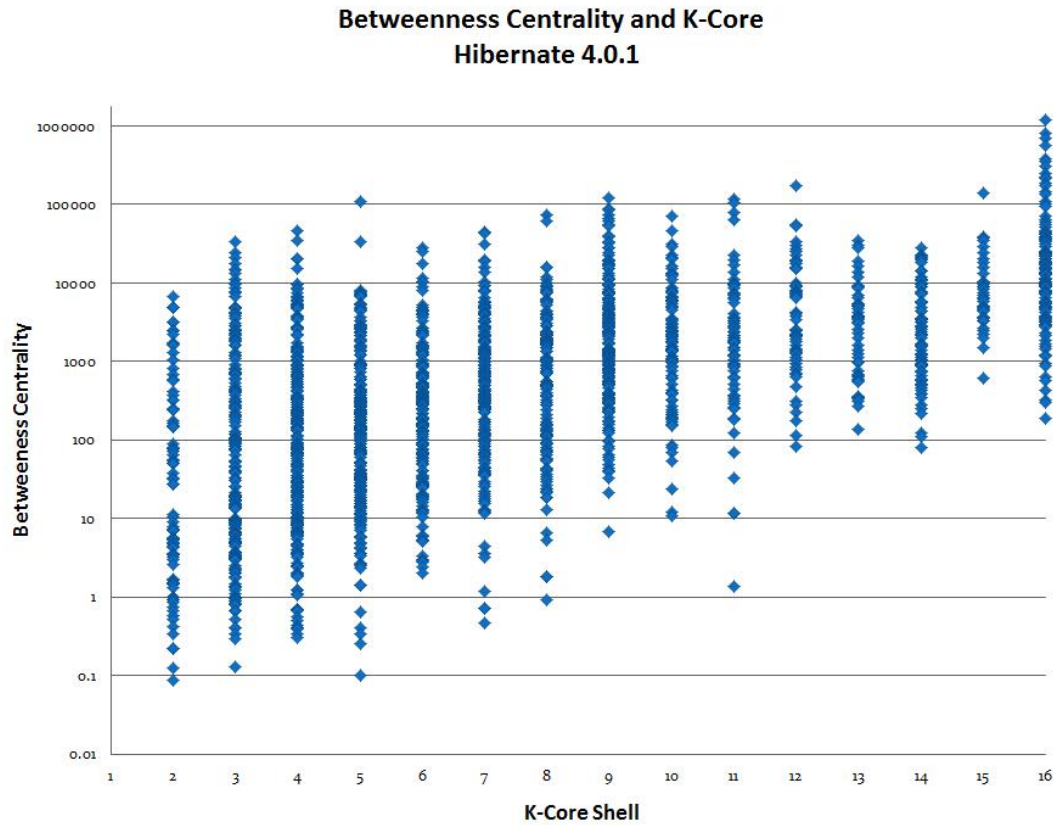
The final centrality measure we examined in relation to k-core degeneracy was betweenness centrality. Like closeness centrality, the first step in this algorithm is to determine the shortest paths in the network. Unlike closeness centrality, this measure only cores about the shortest paths in the network that *pass through* the measured vertice.

So a vertice can have a high centrality by other measures, having a high degree and many connections, but still have a low betweenness centrality if there is another vertice (perhaps a neighbor) that is even better connected to important nodes in the system. The measure can be thought of as measuring the extent to which a vertice serves as a *bridge* in the network. If two parts of a software system are largely segregated, a class that served as the link between the sub-graphs would have a very high betweenness centrality.

The formal definition for betweenness centrality is expressed as:

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Here  $v$  is the examined node, while  $s$  and  $t$  are different nodes from  $v$  and from each-other.  $O_{st}$  is the total number of shortest paths from  $s$  to  $t$ , and  $O_{st}(v)$  is the number of shortest paths that pass through  $v$ . The sum of this ration for each node in the network derives the final score.



*Figure 34: Hibernate-ORM Betweenness Centrality*

In figure 34, the betweenness centrality axis is logarithmically scaled. The relationship presented in this graph is a little more interesting, in that there is a large range of centralities in the lower k-core shells. As you get closer to the maximal k-core, very similar observations can be made as with degree and closeness centrality.

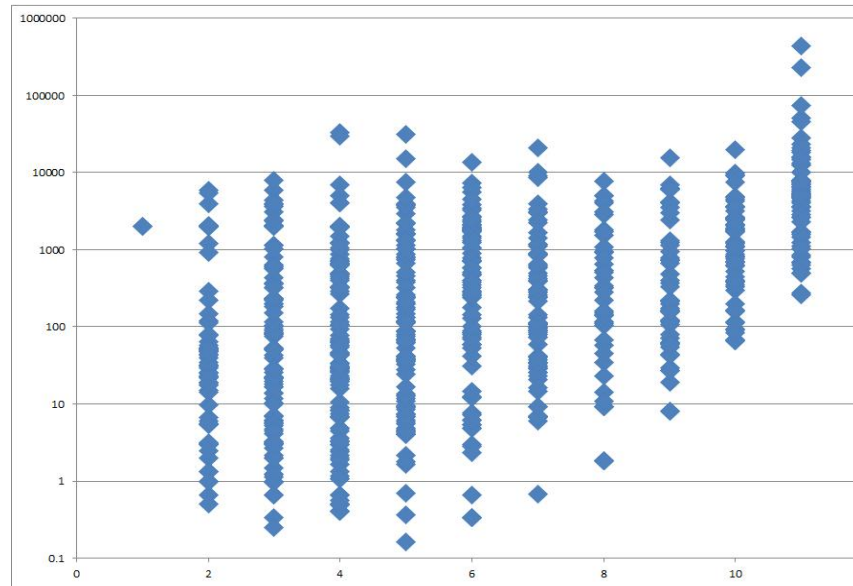


Figure 35: Apache-Ant Betweenness Centrality

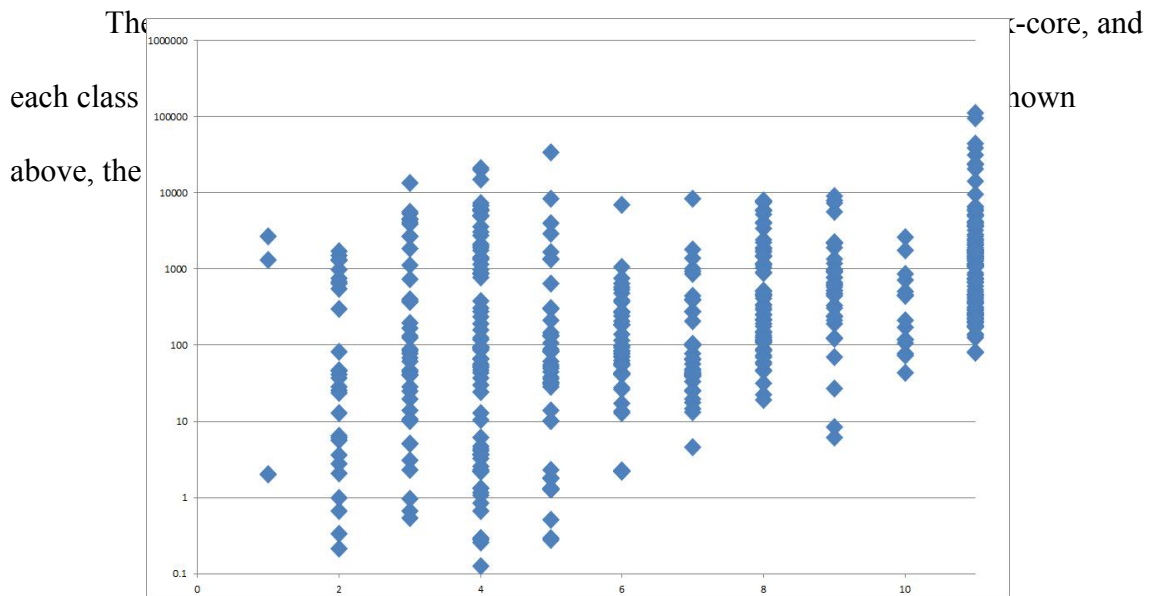


Figure 36: JHotDraw Betweenness Centrality

From examining these three centrality metrics, we conclude that there is a clear relationship between the centrality of a class and its k-core score. This validates that this maximal k-core is not an arbitrary selection of classes. These are classes that have importance to the network structure, based on having a large number of connections to

other classes, and being central in the dependency network structure. However coreness is more than just a reflection of centrality, and is worthy of its own independent examination.

## IX.4 Other Centrality Measures

Our research also calculated the *eigenvector centrality* values for our software systems using the JUNG framework. The eigenvector centrality relies on reasonably advanced linear algebra mathematics. At a high level, it gives connections to important nodes more weight than connections to unimportant nodes. It is similar to Google's famous Page Rank algorithm.

When examined for our software systems, it was found that the eigenvector centrality results calculated by JUNG had a near perfect correlation with degree. Based on this similarity, we felt that the analysis of degree was sufficient for covering this measure.

The final centrality measure we considered was *katz centrality*. In this measure, the centrality is determined by the number of neighbors a node has, as well as the nodes connected indirectly by those neighbors. At each step away from the measured node, there is an attenuation factor which does not give as much weight to distant connections.

It proved difficult to find a freely available implementation of this algorithm that was suitable for our purposes. We considered implementing our own version of the algorithm, but it did not seem an appropriate use of time, as the concept is not very different from closeness centrality or eigenvector centrality, and would likely have given

quite similar results. If in the future a suitable algorithm is made available, it could be interesting to analyze this metric as well.

## X. K-Core and Community Detection

In complex network analysis, one important avenue of study is in community detection. Here a diverse set of algorithms exist that take a network and identify closely connected groups of nodes to identify as existing as part of the same community.

We used the *Louvain Method* of community detection for our research. This is a widely used algorithm based on a greedy optimization routine. In the first pass of this method, low level communities are detected. Subsequent passes group these communities into higher level communities, if such a grouping would increase the overall *modularity* of the overall community output. This aggregation leaves a set of high level communities, which we then compare with the output of our k-core decomposition algorithm.

Hibernate-ORM 4.0.1				
<i>Label</i>	<i>Total Classes</i>	<i>Classes In Core</i>	<i>% of System</i>	<i>% of Core</i>
A	320	22	13.2%	16.8%
B	377	22	15.6%	16.8%
C	305	18	12.6%	13.7%
D	209	16	8.6%	12.2%
E	172	9	7.1%	6.9%
F	124	8	5.1%	6.1%
G	222	8	9.2%	6.1%
H	200	7	8.3%	5.3%
I	270	6	11.1%	4.6%
J	80	5	3.3%	3.8%
K	60	5	2.5%	3.8%
L	13	2	0.5%	1.5%
M	24	2	1.0%	1.5%
N	47	1	1.9%	0.80%

Table 3: Hibernate-ORM Communities

Apache-Ant 1.8.2					JHotDraw 7.6.0				
<i>Label</i>	<i>Total Classes</i>	<i>Classes In Core</i>	<i>% of System</i>	<i>% of Core</i>	<i>Label</i>	<i>Total Classes</i>	<i>Classes In Core</i>	<i>% of System</i>	<i>% of Core</i>
A	171	15	17.2%	24.2%	A	101	23	16.6%	21.9%
B	100	8	10.1%	12.9%	B	126	20	20.7%	19.0%
C	159	7	16.0%	11.3%	C	92	19	15.1%	18.1%
D	117	7	11.8%	11.3%	D	89	15	14.6%	14.3%
E	80	5	8.1%	8.1%	E	74	10	12.2%	9.5%
F	82	5	8.3%	8.1%	F	48	4	7.9%	3.8%
G	50	4	5.0%	6.5%	G	29	4	4.8%	3.8%
H	18	3	1.8%	4.8%	H	17	3	2.8%	2.9%
I	68	3	6.9%	4.8%	I	12	2	2.0%	1.9%
J	60	2	6.0%	3.2%	J	12	2	2.0%	1.9%
K	22	2	2.2%	3.2%	K	1	1	0.2%	1.0%
L	18	1	1.8%	1.6%	L	2	1	0.3%	1.0%
M	3	0	0.3%	0.0%	M	2	1	0.3%	1.0%
N	25	0	2.5%	0.0%	N	2	0	0.3%	0.0%
O	19	0	1.9%	0.0%	O	2	0	0.3%	0.00%

*Table 4: Apache-Ant and JHotDraw Communities*

One might expect the maximal k-core to be such an identified community, as this set of nodes is closely connected. However, our results indicate that instead, the maximal k-core is composed of classes from many different communities. While its true that the inner core is tightly connected, for most nodes in the k-core, there are other nodes in the network that are more truly “neighbors”, at least by this Louvain method.

In fact, in analyzing the inner k-cores of our examined networks, it was the case that nearly all of the high level communities detected had representation in the the core. Communities without representation have very low member counts. One could hypothesize that in fact the most important members of each community will tend to make up the inner core. This would be a happy discovery for our purpose of identifying a

core subset of classes, as one would prefer not to miss out on an entire community of classes that aren't represented in the core.

The Louvain method found a remarkably similar number of communities for each of the three systems (13,14 and 14), despite their highly varied number of vertices. We see in these tables that for the most part, the representation of a community in the core is proportional with the total size of that community in the system. There are a few noteworthy outliers such as community *I* from the Hibernate-ORM system, which has less representation than expected. This variance can likely be explained by statistical noise.

## XI. K-Core Evolution

The reason we examined so many versions of the three selected software systems is because we were intent on studying the evolution of the k-core values between release versions of the products. Before tracing evolution at the class level, first we examine the evolution of the k-core shells at a system level.

### XI.1 System Level Evolution

For analyzing system level evolution, we recorded the size and  $k$  value of the maximal k-core for each version and compared this with the total number of vertices (classes) in the total system.

<b>Hibernate-ORM</b>				
<i>Version</i>	<i>Release Date</i>	<i>Total Vertices</i>	<i>Vertices In Max K-Core</i>	<i>Max K-Core Level</i>
4.0.1.Final	1/11/2012	2429	131	16
4.0.0.CR4	9/30/2011	2403	130	16
3.6.9.Final	12/15/2011	2425	106	16
3.6.1.Final	2/3/2011	2404	106	16
3.5.0.CR2	2/25/2010	2196	98	16
3.2.7.GA	6/3/2009	1265	118	15
3.2.6.GA	2/7/2008	1245	117	15
3.2.2.GA	1/24/2007	1209	115	15
3.1.2	1/28/2006	1048	72	15
2.1.8	1/30/2005	549	61	12
2.1.2	2/4/2004	515	59	12
2.0.beta1	1/29/2003	427	38	11
0.9.4	1/29/2002	136	31	7

*Figure 37: Hibernate-ORM System Level Evolution*

Here we see that Hibernate started as a very modest software system, having only 136 named classes in the first release that we analyzed. The system has enjoyed a very active development life, growing to nearly 2,500 named classes, by far the largest of the three systems we examined. As the class count grew, so too did the max k-core level. However, as the class count grew, the k-core count grew at a much more modest rate. The maximal k-core value seems to have a pretty clear logarithmic growth rate. This makes intuitive sense, as we can think of the maximal k-core as being at the top of a pyramid structure, with lesser cores making up the lower levels. Each new level of the pyramid would need many more nodes to further elevate the peak.

The number of classes within the maximal k-core grows along with the vertex count, but this relationship too is not linear. While the first version has a core composing approximately 23% of the total system, our final version has a core of only about 5% of the total. It is important that our derived core be of manageable size even with very large software systems, and so this would appear to be a beneficial property of the k-core decomposition.

It appears that when the system gains a new k-core level, the number of nodes in the maximal k-core drops down some. This is reminiscent of the electron shells of an atom, where the number of electrons in the outermost valence shell drops as the number of shells increases.

<b>Apache-Ant</b>				
<i>Version</i>	<i>Release Date</i>	<i>Total Vertices</i>	<i>Vertices In Max K-Core</i>	<i>Max K-Core Level</i>
1.8.2	12/27/2010	997	64	11
1.8.1	5/7/2010	792	33	11
1.8.0	2/8/2010	787	33	11
1.7.1	6/27/2008	709	79	10
1.7.0	12/19/2006	698	79	10
1.6.5	6/2/2005	558	61	9
1.6.2	7/14/2004	533	55	9
1.6.0	12/18/2003	507	59	10
1.5	7/10/2002	387	60	9
1.4.1	10/11/2001	248	31	9
1.3	3/3/2001	154	31	8
1.2	10/24/2000	154	31	8
1.1	7/19/2000	83	16	6

*Figure 38: Apache-Ant System Level Evolution*

The Apache-Ant software system also started out very small, with only 83 named classes. It too has grown over the years, but has only grown to a about 1000 classes. The growth of the maximal k-core value has not been as reliably growing as the Hibernate system, with one noticeable drop from 10 to 9 before climbing back up to 10.

<b>JHotDraw</b>				
<i>Version</i>	<i>Release Date</i>	<i>Total Vertices</i>	<i>Vertices In Max K-Core</i>	<i>Max K-Core Level</i>
7.6	1/9/2011	679	123	11
7.5.1	8/1/2010	677	72	12
7.4.1	1/17/2010	641	59	12
7.3	10/18/2009	631	68	12
7.2	5/23/2009	613	68	12
7.1	3/25/2008	409	42	11
7.0.9	6/23/2007	477	61	11
7.0.8	1/10/2007	308	44	10
7.0.7	11/12/2006	278	58	9
7.0.6	8/27/2006	278	58	9
5.4b1	2/1/2004	343	68	10
6.0b1	2/1/2004	339	59	10
5.3	2/9/2002	238	78	9
5.2	2/19/2001	170	47	9

*Figure 39: JHotDraw System Level Evolution*

Finally, the JHotDraw system appears to have a more erratic growth more similar to the Apache-Ant system than Hibernate-ORM. The total named class count starts off higher than the initial versions of the other two systems, but the final version ends up to be the smallest of the three examined systems.

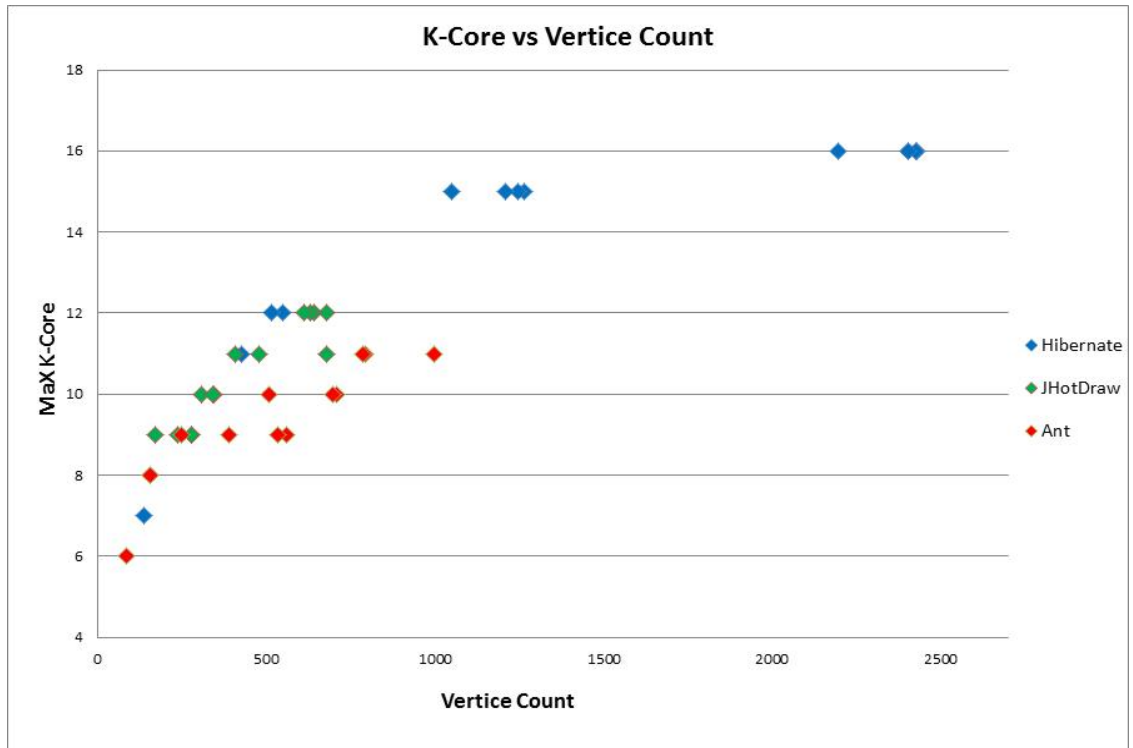


Figure 40: Plot K-Core and Vertice Count All Systems

This chart demonstrates the relationship between the number of vertices (classes) in the system and the K value of the maximal k-core. A clear logarithmic growth can be seen in the Hibernate-ORM project. The Apache-Ant and JHotDraw systems also seem to follow a similar curve, but don't grow quite as fast or high in system size, so are less obvious.

## **XI.2 Component Level Evolution**

We gathered the k-core history of each class in each of the three software systems, and analyzed these histories for insights.

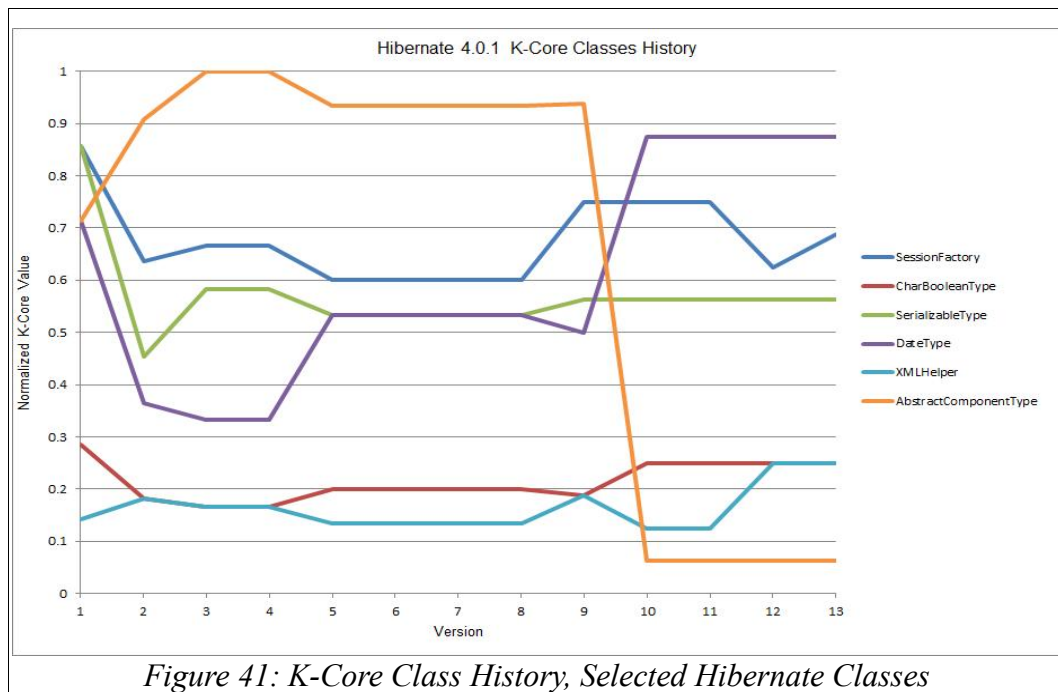
The primary implementation challenge in doing so was in preserving the history of a class when it was moved from one package to another, a fairly common occurrence.

The simplest example of this renaming is the changing of the packages names, as a project redefines itself over the years. For instance,

```
cirrus.hibernate.type.YesNoType was renamed to  
net.sf.hibernate.type.YesNoType before finally ending named  
org.hibernate.type.YesNoType.
```

Non-trivial cases to solve included instances where a class is moved to a different package in the same version where another class is created with the same name in a different package. Here we needed to determine which class to assign the previously recorded history to. These gray areas could usually be solved by examining the connections of the two like named classes. In a few cases manual determination was made by examining the source code.

In making observations about the k-core history, it is first useful to normalize the values. Each project underwent dramatic growth over the course of their development history. Consequently, the maximal k-core for the first version could be the 6-core, while in the latest version it would be the 16-core. We normalize the values by dividing them by the maximum k-core value for that version. A history of 1.0 throughout each version would indicate that the class was in the maximal k-core in each version.

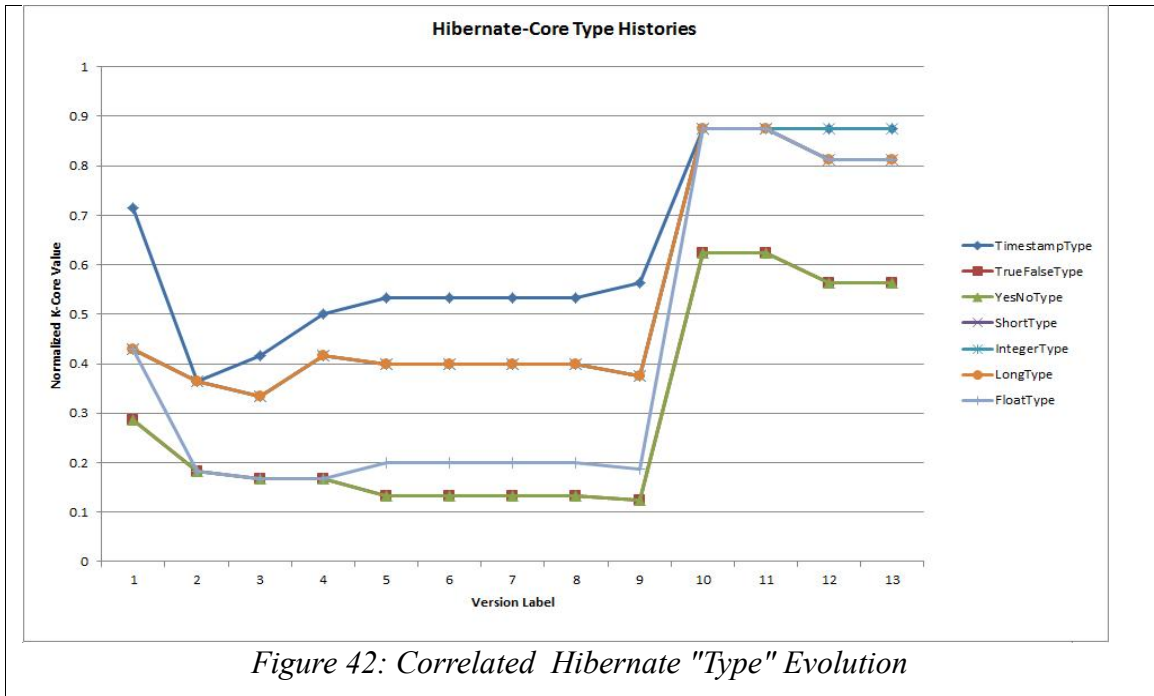


The first analysis attempted from these k-core histories was to determine if dramatic differences in k-core level between versions could be traced to design changes within the software system. In this first sampling of Hibernate-ORM history, we see that despite being in an upper k-core shell for most of the history of the product, the `AbstractComponentType` class underwent a dramatic decrease in its k-core value between versions 9 and 10 (3.5.0.CR2 → 3.6.1.Final). When examining the source code

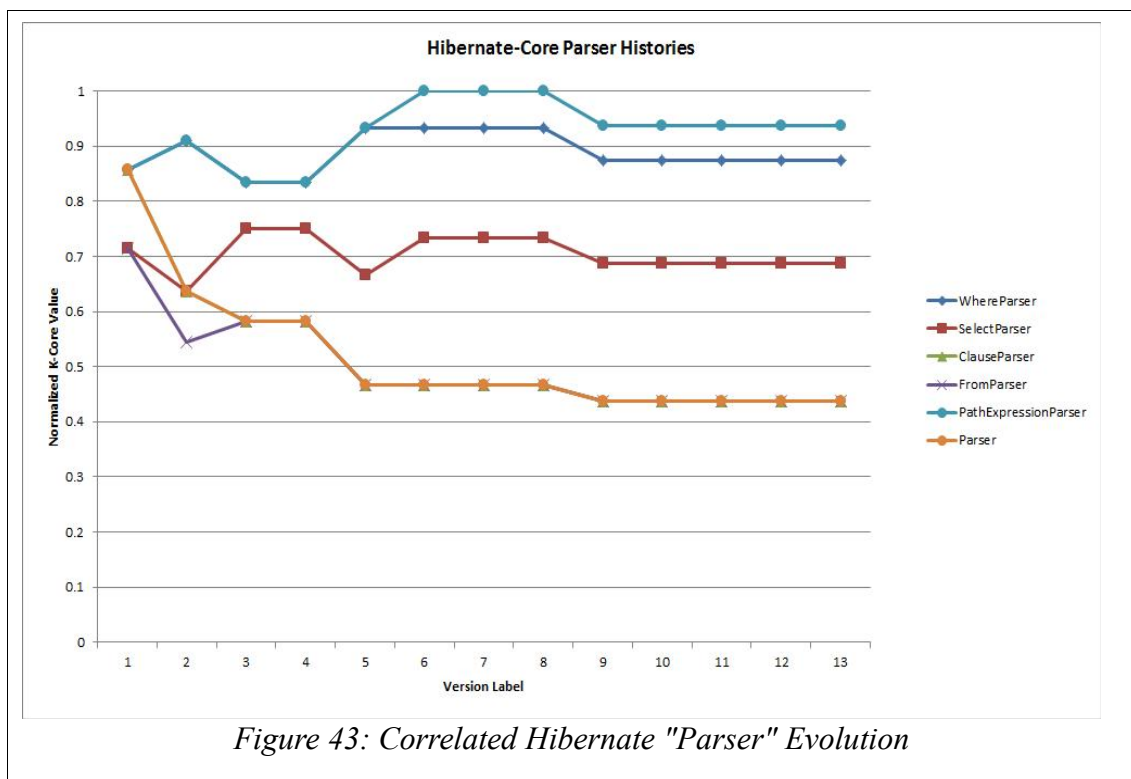
for these two versions, we see that `AbstractComponentType` was deprecated in version 10. All of the functionality from this class was moved to `CompositeType`.

The near inverse of this change can be seen in the `DateType` class. For most of the product's life it was in a mid-level k-core, but then between versions 9 and 10 there was a sharp increase towards a higher k-core shell. When the code is investigated for this class, we see that there was an introduction of multiple “Dialect” classes into the Hibernate system, which the framework uses to tailor generated SQL to the particular database types. The majority of these new dialects depends on this `DateType` class, increasing its relative importance to the system.

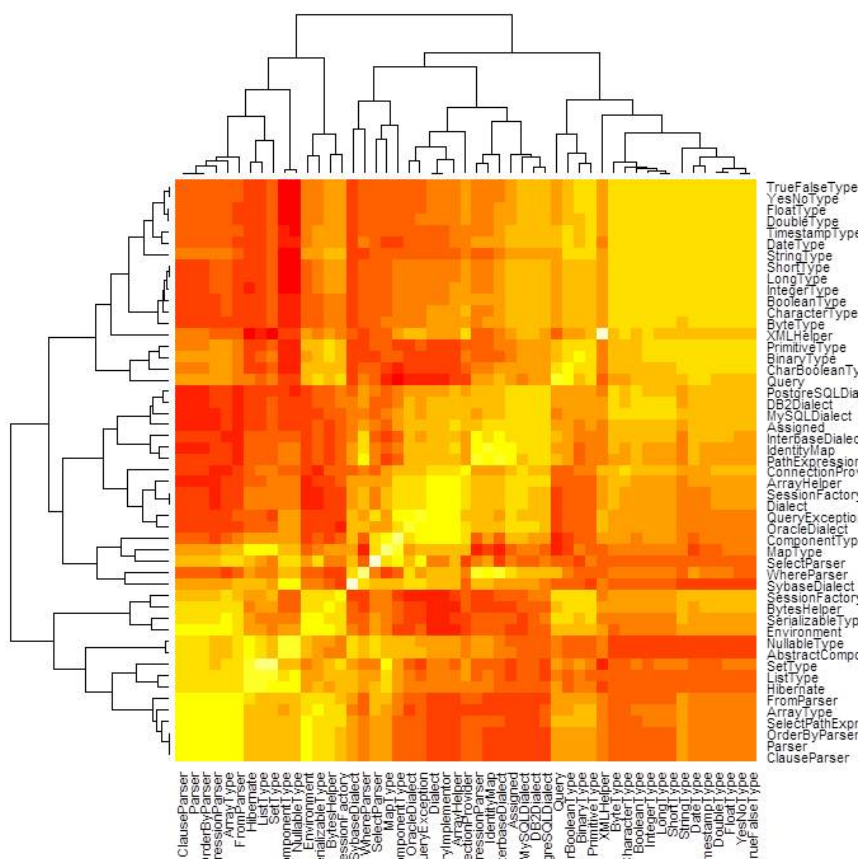
When the histories for the k-core evolution were plotted, we discovered that some classes appeared to have undergone a correlated evolution. Two examples of this correlated evolution phenomenon are plotted below. The first set of these classes we found were the “Type” classes in the Hibernate-ORM system. These classes, like the `DateType` mentioned above, are used by the system to map object data to the corresponding database column type. The second case of correlated evolution clearly visible was in the “Parser” classes. These classes are used by the system for translating the Hibernate Query Language (HQL) into the corresponding SQL instructions.



The plots for these two types of classes are shown on the following page in Figure 42 and 43 respectively.



The figure below is a heat-map generated from these Pearson coefficients. In this visualization, the lighter colors correspond to high correlations, those closer to 1.0. The darker colors are lower correlation scores, ones that indicate a low degree of co-evolution.



*Figure 44: Hibernate-ORM Correlated Evolution Heat-map*

As only a very small percentage of classes in the systems have existed since the very first version of the software, we then weighted these correlations by how many combined versions they've existed in. Correlations that have only existed for the last few versions were treated as less interesting as correlations that exist throughout

many version changes. We then took the highest correlations and examined them to determine if classes with a similar k-core history might also belong to the same communities. Various cutoffs were examined, between 75% correlation (coefficient of 0.75 or higher) and 97% (coefficient of 0.97 or higher). The results are shown in the table below.

<b>Hibernate-ORM</b>					
	<b>75%</b>	<b>85%</b>	<b>90%</b>	<b>95%</b>	<b>97%</b>
<i>Number of Correlations</i>	14929	2109	567	85	43
<i>% In Same Package</i>	6.5%	12.1%	23.8%	62.4%	90.6%
<i>% In Same Community</i>	10.2%	10.5%	11.3%	8.2%	4.7%

<b>Apache-Ant</b>					
	<b>75%</b>	<b>85%</b>	<b>90%</b>	<b>95%</b>	<b>97%</b>
<i>Number of Correlations</i>	3073	790	337	92	65
<i>% In Same Package</i>	28.0%	43.4%	41.5%	44.6%	52.3%
<i>% In Same Community</i>	9.7%	9.5%	9.2%	10.9%	10.8%

<b>JHotDraw</b>					
	<b>75%</b>	<b>85%</b>	<b>90%</b>	<b>95%</b>	<b>97%</b>
<i>Number of Correlations</i>	346	20	8	5	1
<i>% In Same Package</i>	7.5%	35.0%	25.0%	40.0%	100.0%
<i>% In Same Community</i>	15.3%	15.0%	12.5%	20.0%	0.0%

*Table 5: Community and Package Prediction based on Correlated Evolution*

What we found was that these correlations did not appear to have any useful prediction capabilities for whether these classes were in the same *Louvain* assigned community. However, it did appear that classes that had correlated k-core evolution were typically in the same *packages*. In Java, a package is a grouping of classes, typically

organized by serving a similar function. There are hundreds of packages in the Hibernate system, so to have over 60% of correlated k-core histories belong to the same package is statistically significant.

## XII. K-Cores and Seniority

From the investigations into the k-Core histories emerged the idea of examining the *seniority* of the classes in the software system, and how it might relate to the cores extracted from decomposition algorithm. Here we are defining *seniority* as a measure of how long an individual class has existed in the software system.

Classes introduced in the very first examined release version would receive the maximum possible score equal to the number of these versions. A class introduced in the very last version would receive a score of one.

The central conjecture here is that in most organizational structures, the most important members will tend to be those that have been around the longest. We reason that the same is true of software systems, with the most critical classes tending to be those that were developed during the early life of the project, and which have persisted through design changes and refactoring efforts since then.

The most important thing about this metric is that it is completely independent of network structure. This means that if a correlation is found, it cannot merely be dismissed as being a metric that measures a similar quality, which could be argued for the centrality metrics.

To examine this metric in relation to k-core decomposition, took the latest versions of the three systems, and calculated the average seniority of each class contained within the k-core shells of that system. This average seniority is shown in the table and graphs below:

K-Core	Hibernate-ORM	Apache-Ant	JHotDraw
0	3.18	2.37	2.99
1	5.62	5.68	5.76
2	4.65	6.23	5.10
3	5.16	6.11	5.54
4	5.75	6.12	5.53
5	5.85	7.06	7.21
6	5.78	4.58	6.91
7	5.78	5.44	6.00
8	5.99	7.49	7.71
9	5.75	7.30	7.62
10	6.33	7.43	8.41
11	6.77	<b>10.08</b>	<b>9.27</b>
12	8.14		
13	9.09		
14	9.38		
15	8.49		
16	<b>9.29</b>		

*Table 6: Average Seniority of Classes within K-Core*

What we found is that there is a clear direct correlation between our seniority metric and the k-core shells of our software systems. The correlation is not perfect, as the Hibernate-ORM project has a slightly higher average seniority of the 14-core than it has in its 16-core. However, overall the average seniority increases in direct proportion to the k-core levels. This is especially clear in the JHotDraw and Apache-Ant projects, where the maximal 11-core has a large increase over any other k-core in that system.

In examining why it might be the case that Hibernate-ORM does have as clear of a trend as the other two products, it was discovered that a significant number of classes in the maximal core were introduced when the Hibernate-Annotations project was merged into the Hibernate-ORM baseline in version 3.6. These merged in classes are actually more senior than they appear, but development of these classes occurred in a different project umbrella. This explains why the average seniority for the 16-core of Hibernate-ORM is slightly lower. Even so, the trend is clear, as seen in the graph below:

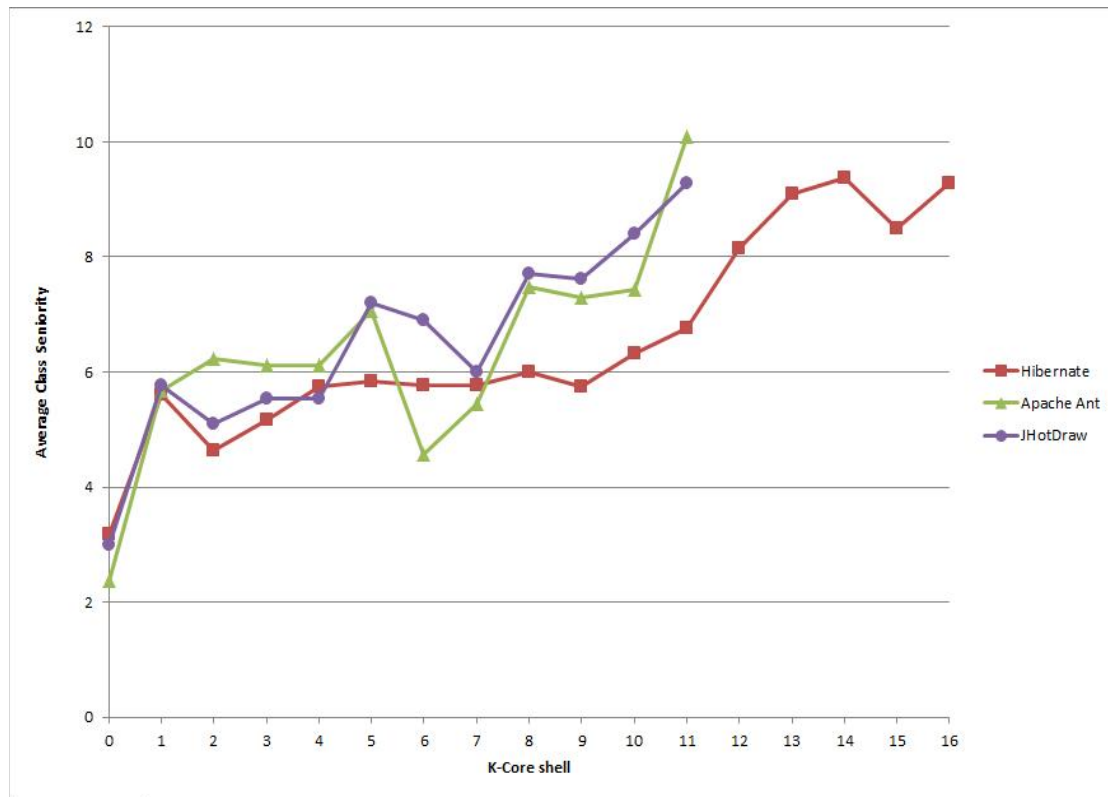


Figure 45: K-Core and Average Seniority

The above plot represents the same information as seen in Table 8. With a few outliers, the inner k-core shells have a much higher likelihood of containing the most love-lived classes in the software system.

These results provide some degree of validation that the decomposition algorithm extracts a core that contains important classes, and not just by network analysis measures.

### **XIII. Conclusions**

In this research we set out to discover or create an automated way to extract the core classes of a complex software system. After examining the related literature, we first reproduced the MacCormack et al methodology against a sample of software systems (Hibernate, Ant, JHotDraw). What we found is that the results were not suitable for our research goals. We next examined the methodology of Holme, where k-core decomposition was the algorithm for extracting a core.

Having favorable initial results with this method, we examined the relationship between social network metrics for centrality and the derived k-core shells. We found that the maximal k-core contained classes with a lower-bound centrality score for each metric that was greater than any other k-core shell. We also determined that the most central classes to the dependency network for each system were in this maximal k-core, partially validating that the most inner k-core contained important nodes and connections to the system.

Next the relationship between the maximal k-core and alternative community detection (Louvain method) was analyzed. We found that the maximal k-core was typically composed of a sampling of the most central nodes from each of the high level communities. This relationship between the centrality measures and the k-core values was remarkably similar across each of the three examined systems.

Our next avenue of exploration was in tracing the evolution of the k-core shells between versions of each of the three systems, a completely novel research path. We traced both the system level k-core evolution and the component level evolution for each

class. Having noticed that certain classes underwent an apparent co-evolution, we derived the correlation coefficients between classes in the system. We determined that these correlations typically occurred between classes in the same packages, which fulfill similar functions to the system.

Finally we used these k-core histories to determine the length of life in the software system for each class, termed the *seniority*. The average seniority for each class within the k-core shells of the latest versions of the software systems was compared, and we discovered a clear correlation between this seniority metric and the k-core level the classes ended up in. Classes that have been in the system longer are typically found in higher k-core shells. This provides a validation to the importance of these inner-most classes completely independent of network structure.

A software developer can use this k-core decomposition technique to analyze their systems for gaining system knowledge. The algorithm is fast and easily automated, and when combined with a tool like *DependencyFinder*, can be run against either the source code or binaries. The core of the software system can be presented to the developer for analysis. The overall system can be visualized with the Lanet-VI algorithm, and the core itself can be shown visually with any number of network display tools (such as JUNG).

This inner core meets the theoretical definition of coreness, and features the most central and tightly connected classes of the software system. It is our belief that studying this core will efficiently impart system knowledge. A developer using this decomposition and visualization method should arrive at greater understanding sooner than without employing this research tool.

## **XIV. Future Work**

### **XIV.1 More Software Systems**

Any of the observations made in our research would be strengthened by the application to more software systems. The process for extracting dependencies, applying the k-core decomposition algorithm, and then generating the relevant analysis data was done in a repeatable way. This allowed us to apply this process to 40 different software versions. A new software system would not be any different. The ability of DependencyFinder to extract directly from Java binaries assists the ease of this process greatly.

### **XIV.2 More Diverse Systems**

These three systems were all open source Java systems. This was an intentional choice, but the validity of the results would be broadened by examining systems that vary by some key characteristics.

- *Commercial Systems*
  - Closed source commercial systems are developed by organizations with different priorities and methodologies from open source systems.
- *Larger Systems*
  - While the Hibernate framework was our largest system developed, there are many systems that are larger by an order of magnitude or two. Examples of these large systems would include operating systems. While

it may be useful to extract a core of around 150 classes from a system with 2500 classes, would it be more or less useful to extract a core from a much larger system?

- *Different Programming Languages*
  - Java is a language where many of the dependencies between classes are obscured by instead having your class depend on Interface objects. These interfaces are then implemented by the concrete class that calls your code. Languages that don't have these interface intermediaries have more tightly coupled code. The cores extracted from these systems could look very different.

### **XIV.3 Empirical Studies**

The process of extracting a useful core is an interesting challenge, but how does one evaluate the usefulness of the generated core? Our research attempts this validation based on the importance of the classes in the system. However, what would truly validate this research is if a developer was given such a list of core classes, and we were able to evaluate how helpful it truly is. Measures of successful cognition could be applied to a developer who studied the system based on the identified core components versus a control where the developer was left to study the system without such aids. Given the scope constraints of our study, we were not able to pursue this sort of empirical testing.

### **XIV.4 Validation with Experts**

Another way to validate the results would be if we could compare our results of identified core classes against an expert generated list of core or critical classes. Attempts

to find existing documented class lists on Java products proved fruitless. We contacted posted to both the Hibernate and Apache-Ant developer's mailing lists asking for possible interest in providing a list of what classes they feel comprise the core of the system. We unfortunately received no interest. The manual work required for this task would be significant, so it is perhaps not surprising that no one wished to undertake such an endeavor. The difficulty in producing this core class list by hand can favorably contrasted with a program that performs the k-core decomposition within mere seconds.

#### **XIV.5 Comparison with other Software Metrics**

A possible future research path could involve the further comparison with k-core values against other software engineering metrics, such as those which measure complexity. There are a wide variety of such metrics, including ones specifically designed for object oriented software.

Additionally, one could compare these k-core values against measures of churn (how often the class is changed). Intuitively, one could make the case that the core of a software system would be made up of classes changed most often, as they are the most critical. The counter-argument for this supposition would be that the core is the most dangerous to make changes to, and so should not be changed needlessly. It may be interesting to discover which of these intuitions, if either, is correct.

Similarly, comparing these results with defect tracking systems could produce interesting results. It could be predicted that changes to core classes produce more defects than changes to more peripheral classes. This analysis would require a very good defect tracking system that ties each defect to the change which produced it.

#### **XIV.6 Weighted Connections Between Classes**

In our research we considered each dependency between two classes as being binary. This ignores that a class could reference a single method of a related class once, or it could call many different methods many times. One possible approach for modeling the software system would be to consider each of these references to be its own independent dependency. The k-core degeneracy algorithm would then consider these edges as just as valid as between classes, leading to classes that are more tightly connected making it further into the core.

This modification to the algorithm could help capture the complexity of the classes, and to truly find a core that is tightly connected. No dependency extraction tool we considered has this fine of detail in the number of connections, but the *DependencyFinder* code is open source and could possibly be tweaked for this purpose.

Similarly, our model of the software system could treat use relations as being weighted differently from inheritance relationships. If we have a software system with a core identified by system experts, various models could be applied and tested to determine which best fits.

## XV. References

- [Borgotti and Everett, 1999] S. P. Borgatti and M. G. Everett. Models of core / periphery structures. *Social Networks*, 21:375–395, 1999
- [Carmi et al, 2007] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, E. Shir. A model of Internet topology using k-shell decomposition. *Proceedings of the National Academy of Sciences*, vol. 104, no. 27, 11150-11154, 2007.
- [Bader and Hogue, 2003] G. D. Bader, C.W. V. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4(2), 2003
- [Gaertler and Patrignani, 2003] M. Gaertler, M. Patrignani. Dynamic Analysis of the Autonomous System Graph. *International Workshop on Inter-domain Performance and Simulation*, 13-24, 2004.
- [Holme, 2005] P. Holme. Core-periphery organization of complex networks. *Physical Review E*, 2005.
- [Inoue et al, 2005] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, 31(3), 213–225, 2005.
- [Kusiak et al, 1994] A. Kusiak, N. Larson, J. Wang. Reengineering of design and manufacturing processes. *Computers and Industrial Engineering*, 26(3), 521–536, 1994.
- [Hamelin et al, 2005] J.I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, A. Vespignani. K-Core Decomposition: A tool for the visualization of large scale networks. <http://arxiv.org/abs/cs/0504107>, 2005
- [Hamelin et al, 2008] J.I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, A. Vespignani. K-Core decomposition of internet graphs: hierarchies, self-similarity and measurement biases. *Networks and Heterogeneous Media* 3, 371, 2008
- [Han Yan Tong et al, 2002] A. H. Y. Tong, B. Drees, G. Nardelli, G. D. Bader, B. Brannetti, A Combined Experimental and Computational Strategy to Define Protein Interaction Networks for Peptide Recognition Modules, *Science (New York, N.Y.)*. 295(5553): 321-4.

- [Haohua et al, 2008] Z. Haohua, Z. Hai, C. Wei, Z. Ming, L. Guilan, Z Haohua. Visualization and Cognition of Large-scale Software Structure using the k-core Analysis. *International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, 954-957, 2008
- [MacCormack et al, 2010] A. MacCormack, C Baldwin, J. Rusnak. The architecture of complex systems: Do core-periphery structures dominate? *Harvard Business School Working Paper 10-059*, 2010.
- [Mayrhauser and Vans, 1995] A. Mayrhauser, A. M. Vans. Program Comprehension During Software Maintenance and Evolution. *Journal Computer*, vol. 28, issue 8, 1995
- [Myers, 2003] C. R. Myers. Software systems as complex networks: Structure, function and evolvability of software collaboration graphs. *Phys. Rev. E*, vol. 68, 2003.
- [Nguyen et al, 2010] T. H. D. Nguyen, B. Adams, A E. Hassan. Studying the impact of depdency network measures on software quality. *IEEE International Conference on Software Maintenance*, 1-10, 2010
- [Paymal et al, 2001] P. Paymal, R. Patil, S. Bhowmick, H. Siy. Empirical Study of Software Evolution Using Community Detection, 2010
- [Seidman, 1983] S. B. Seidman. Network structure and minimum degree, *Social Networks*, 5, 269-287, 1983.
- [Synder and Kick, 1979] D. Snyder, E. Kick. Structural Position in the World System and Economic Growth. *American Journal of Sociology* 84: 1096-1126.
- [Tushman and Murmann, 1998] J. P. Murmann, M. L. Tushman. Dominant Designs, Technology Cycles, and Organizational Outcomes. Staw, B. and Cummings, L.L (eds.) *Research in Organizational Behavior*, JAI Press, Vol 20, 1998.
- [Zimmermann and Nagappan, 2008] T. Zimmermann, N. Nagappan, Predicting Defects using Network Analysis on Dependency Graphs. *ICSE '08 Proceedings of the 30th International conference on Software Engineering*, 531-540, 2008