Exploring Coordination Structures in Open Source Software Development

Chintan Amrit and Jos van Hillegersberg
Dept. of IS & CM
University of Twente
The Netherlands
{c.amrit,j.vanHillegersberg}@utwente.nl

Abstract

Coordination is difficult to achieve in a large globally distributed project setting. The problem is multiplied in open source software development projects, where most of the traditional means of coordination such as plans, system-level designs, schedules and defined process are not used. In order to maintain proper coordination in open source projects one needs to monitor the progress of the FLOSS project continuously. We propose a mechanism of display of Socio-Technical project structures as well as propose a metric based on these structures. With the help of the display and the metric we can locate and describe the extent of particular coordination problems in open source software development. Using the tool TESNA (TEchnical and Social Network Analysis) that we have developed; we cluster the software and produce a display of the different software clusters along with the people working on its constituting software class. We then demonstrate the technique on a sample FLOSS project that we think is on the brink of becoming inactive.

Introduction

Distributed self-organizing teams develop most Free/Libre Open Source Software (FLOSS). Developers from all over the world rarely meet face to face and coordinate their activity primarily by means of computer-mediated communications, like e-mail and bulletin boards [1, 2]. Such developers, users along with the associated user turned developers of the software form a community of practice [3]. The success or failure of open source software depends largely on the health of the health of such open source communities [4, 5]. Most of the literature on open source software development attests its success and only a relatively small but growing number of empirical studies exist that explain how these communities actually produce software [1, 6, 7]. Also, not everything is ok with open source development projects. Out of 153579 projects registered in source forge, only 12.24% of the projects had attained stable status (has a stable version of their software) when we last checked in July 2007. For an IT professional or FLOSS project

leader it seems to be crucial to know the status of the open source project in order to contribute or recommend the project [5]. To this extent we provide a set of STSCs which can be checked in order to see the coordination inconsistencies of the work being done in an FLOSS project. deSouza et al recognize socio-technical patterns of work assignment among the open source community members [6]. In this paper, we extend this research further by identifying socio-technical coordination problems, what we call Socio-Technical Structure Clashes or STSCs [8] based on one of these patterns. We provide a theoretical framework along with a technique and a metric for the identification of this STSC. We then show the occurrence of this particular FLOSS STSC in one open source project called JAIM, in order to demonstrate the technique. With our technique we hope to provide another method of assessing the health of FLOSS projects.

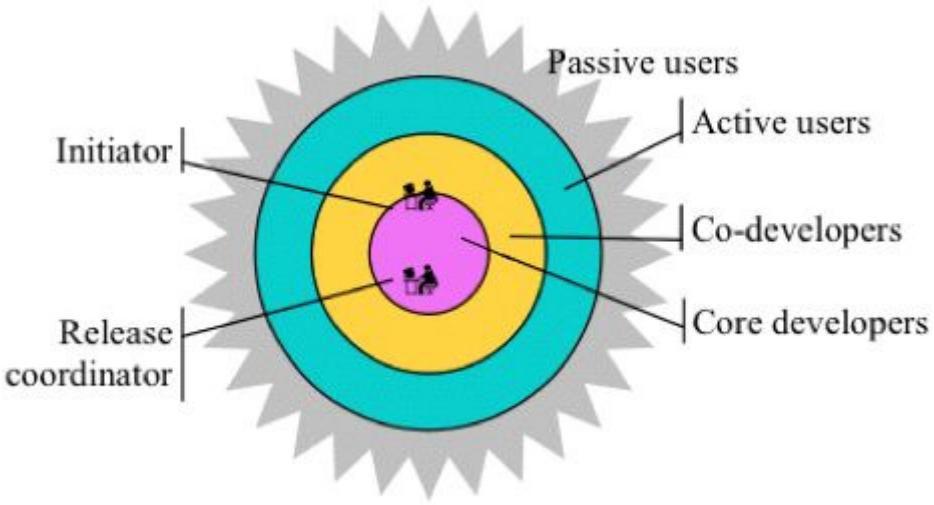

Figure 1: A healthy open source community (taken from [5])

OSS Community Structure

Although there is no strict hierarchy in open source communities, the structure of the communities is not completely flat. There does exist an implicit role based social structure, where certain members of the community take up assume certain roles by themselves based on their interest in the project [3]. According to Crowston and Howison [5] a healthy open source community has the structure as shown in Figure 1 with distinct roles for developers, leaders and users. Core developers are responsible for guiding and coordinating the development of a FLOSS project. These developers are generally involved with the project for a relatively long period, and make significant contributions to the development and evolution of the FLOSS system.

In those FLOSS projects that have evolved into their second generation there exists a council of core members that take the responsibility of guiding development. Such a council replaces the single core developer in second-generation projects like Linux, Mozilla, Apache group etc.

- **Project Leaders**: The Project Leader is generally the person responsible for starting the FLOSS project. This is the person responsible for the vision and overall direction of the project.

- **Co-developers**: Also known as peripheral developers, occasionally contribute new features and functionality to the system. Frequently, the core developers review their code before inclusion in the code base. By displaying interest and capability, the peripheral developers can move to the core.

- **Active users**: contribute by testing new releases, posting bug reports, writing documentation and by answering the questions of passive users.

Each FLOSS community has a unique structure depending on the nature of the system and its member population. The structure of the system differs on the percentage of each role in the community. In general, most members are passive users, and most systems are developed by a small number of developers [1]. In this research we take a different approach than the one adopted by Crowston et.al. [5]. Instead of trying to locate the core-periphery structure of the developers [5] and then draw conclusions about the health of the project, we look at the core-periphery structure of the software and see how movement across this structure relates to the health of the project. We also show how this movement can be monitored using visualizations as well as a metric. We now describe what we mean by STSC in the context of FLOSS projects.

STSCs in FLOSS projects

A Socio-Technical Structure Clash (STSC) occurs if and when a Socio-Technical Pattern exists that indicates that the social network of the software development team does not match the technical dependencies within the software architecture under development [8]. STSCs are indicative of coordination problems in a software development organization. Some of these problems (or STSCs) concerning development activities have been collected and described by Coplien et al. [9] including a set of what they call Process Patterns to deal with these coordination problems. As the term process patterns is also used in business process management and workflow, we prefer to use the term Socio-Technical Patterns to refer to those patterns involving problems related to both the social and technical aspects of the software process [8]. de Souza et al. identify a core periphery shift by mining software repositories. The core-periphery shift in a healthy FLOSS project is when the peripheral developers move from the periphery of the project to the core, as their interest and contribution in the project increases [6]. Note that our notion of Core-Periphery is from the perspective of the software and not from the perspective of the social structure of the developers as described by Crowston, Wei et.al. [10]. So, in this sense we can be adding one more technique of defining Core-Periphery developers [10]. We come up with the following STSCs based on this Socio-Technical Pattern:

Pattern Format	Core-Periphery Shifts Pattern [6]
A problem growing from the Forces	When developers move from development of the core modules of the software to development of the peripheral modules.
The current structure of the system giving the context of the problem	The different part of the software the developers are working on
Forces that require Resolution	When core developers move on to developing peripheral parts of the software.
The solution proposed the problem	Get more developers interested in the core part of the software
Discusses the context resulting from applying the pattern. In particular, trade-offs should be mentioned	Make sure that more people are interested in the core part of the software project.
The design rationale behind the proposed solution. Patterns are often coupled or composed with other patterns, leading to the concept of pattern language.	The core of the FLOSS project is vital to its performance and hence needs more work in order to reach stability.

Table 1: Socio-Technical Pattern for FLOSS projects.

1) If the developers working in the periphery of the project do not move towards the centre core of the project.
2) If the developers working on the core of the project move to working on the periphery of the project.

From now on, we would refer to these STSCs as FLOSS STSC 1, 2 respectively. The pattern is described in greater detail (in the pattern format) in Table 1.

Detection of STSC in FLOSS

In order to detect the STSC related to FLOSS projects we used a clustering algorithm based on the algorithm by Fernandez [11] and later on used by MacCormack et al. [12]. We implemented this algorithm (see Appendix) to cluster the software components into a number of clusters, as explained in the following subsection. The resulting software clusters are the red clusters seen in Figure 4. We then included the author information of the components (extracted from the project's software repository (SVN)) in the same diagram and displayed the authors of the individual code modules as authors of the respected clusters (in which the code modules lay), as seen in Fig 4 where the developers are shown as blue circles. As this clustering method is based on the dependencies between, the software components, the central cluster would represent the most dependent components of the software, or in other words the software core. Thus, the structure of the clustered software graph would represent the actual core and periphery of the software architecture.

It has to be noted that this break up of core and periphery is based on software dependencies and could be different from that which was designed (especially in commercial software development, which generally follows traditional process of designing, developing and testing cycles). In this paper we trace the co-evolution of the project and the communities [3] and show the method of detecting FLOSS related STSCs by looking at the author-cluster figures (Fig 4 – 7, see Appendix) at equal intervals in the development lifetime of the project. To make the detection mechanism more automated than simply the observation of the clusters, we define two ways of measuring this metric. The metrics are based on the representation of the cluster graph and the author cluster graph (Fig 4) as Matrices as shown in the following subsection.

Measuring the Core Periphery Shift STSC metric

Dependency matrices have been used in engineering literature to represent the dependency between people and tasks. Li et al. [13] use dependency matrices to analyze dependencies between components in a CBS. Cataldo et.al. define what they call *Task Dependency Matrix* and *Task Assignment Matrix* that represents the task dependencies and the people working on specific tasks respectively [14].
In this research we define a *Software Dependency Matrix* to represent the dependencies between the software classes (we take the class to the smallest unit of measurement as in MacCormack et.al. [12])
The *Software Dependency Matrix* would represent:

- Function call dependency
- Inheritance dependency
- Aggregation dependency

We also use a matrix similar *Task Assignment Matrix* [14] we call this matrix *Developer Software Matrix* that represents the particular software modules the developers are working on. The developers are represented by the rows of the matrix while the columns represent the software modules the developers are working on.
In order to better understand the Core-Periphery Shift (Table 1) pattern we cluster the software based on the dependencies of the software modules using the algorithm described by Fernandez [11] and used by MacCormack et.al. [12]. The clusters formed from this clustering process represent the amount of dependency in the modules. The larger a particular cluster is the more number of closely dependent modules the cluster would have.
After clustering we define the *Cluster Dependency Matrix* to represent the connections or dependencies between software module clusters. The corresponding *People Cluster Matrix* represents the people working on the clusters. We also have the *Cluster Size Matrix* which is the matrix of the sizes of the clusters in the *Cluster Dependency Matrix*.
The procedure to calculate the core-periphery shift consists of the following steps:

1. Identifying the core and the periphery of the *Cluster Dependency Matrix*
2. Reordering the *Cluster Dependency Matrix* in the descending order of Core-ness.
3. Reordering the *People Cluster Matrix* in the same order as the *Cluster Dependency Matrix.*

4. Calculating the core-periphery metric

In order to identify the core and the periphery of the *Cluster Dependency Matrix* we realize that the core-ness of a particular cluster depends not only on the size of the cluster but also the dependencies of the particular cluster with other clusters. We hence multiply the *Cluster Dependency Matrix* with the *Cluster Size Matrix.* The resulting matrix gives us an indication of the core and the periphery clusters with the larger entries being more core than the smaller entries. So if we arrange the columns of this matrix in the descending order we would have the clusters in the descending order of core-ness. Now we can also re-arrange the corresponding columns of the *People Cluster Matrix* so that the columns of both the matrices are arranged in the descending order of core-ness from left to right. Each row of this *People Cluster Matrix* represents the different clusters the particular person identified by the row number is working on and the row can be converted to a Boolean vector.
We now use this re-arranged *People Cluster Matrix* to calculate the core-periphery metric in the following two ways:

1. The binary number corresponding to the Boolean vector (raising each digit to the corresponding power of two)
2. The average distance from the most core cluster

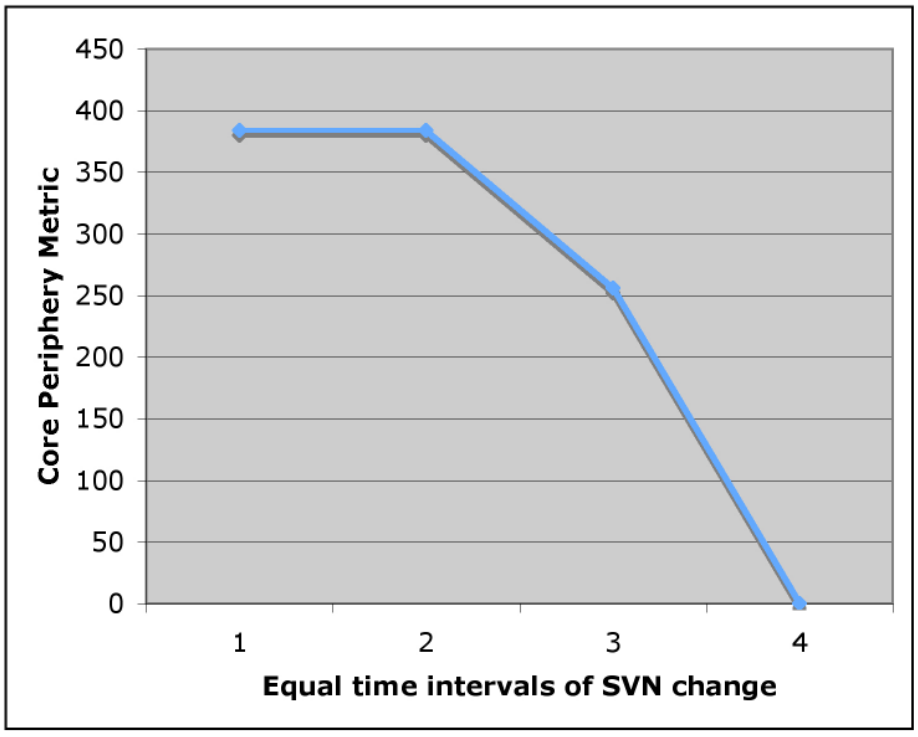

Figure 2: Binary Number Core-Periphery Metric for JAIM

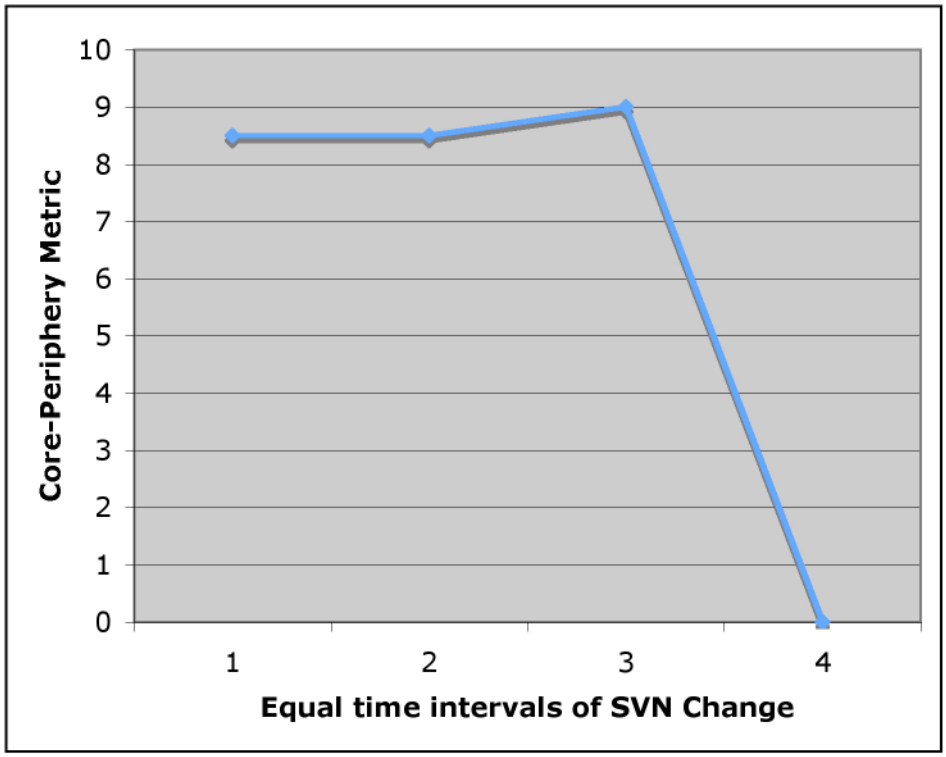

Figure 3: Average Distance Core-Periphery Metric for JAIM

The first metric (from now on called the *Binary Core-Periphery* metric) is got by simply getting the decimal number (by raising the digits to the power of two and summing up) corresponding to the binary number represented by the Boolean vector. The second metric (from now on called the *Average Distance Core-Periphery* Metric) based on the average distance function $N\text{-}j/m$, where m is the number of 1's present in a row of the matrix (the number of clusters each developer works on), j is the position of the cluster the person is working on and N is the total number of clusters. We can sum up the numbers for each row (for both the metrics) we get the total metric for a week W_t (where t is the number of the week). We then take an average of the vector values W_t/n, where n is the number of developers working that week. Now, a change in the developer's task for a week, i.e. movement towards the core or towards the periphery can easily be deduced by taking the difference of this number every week, i.e. $W_t/n - W_{t+1}/m$ would give the net core or periphery shift. If this difference produces negative values in the vector, this denotes movement to the periphery whereas positive values denote movement to the core.

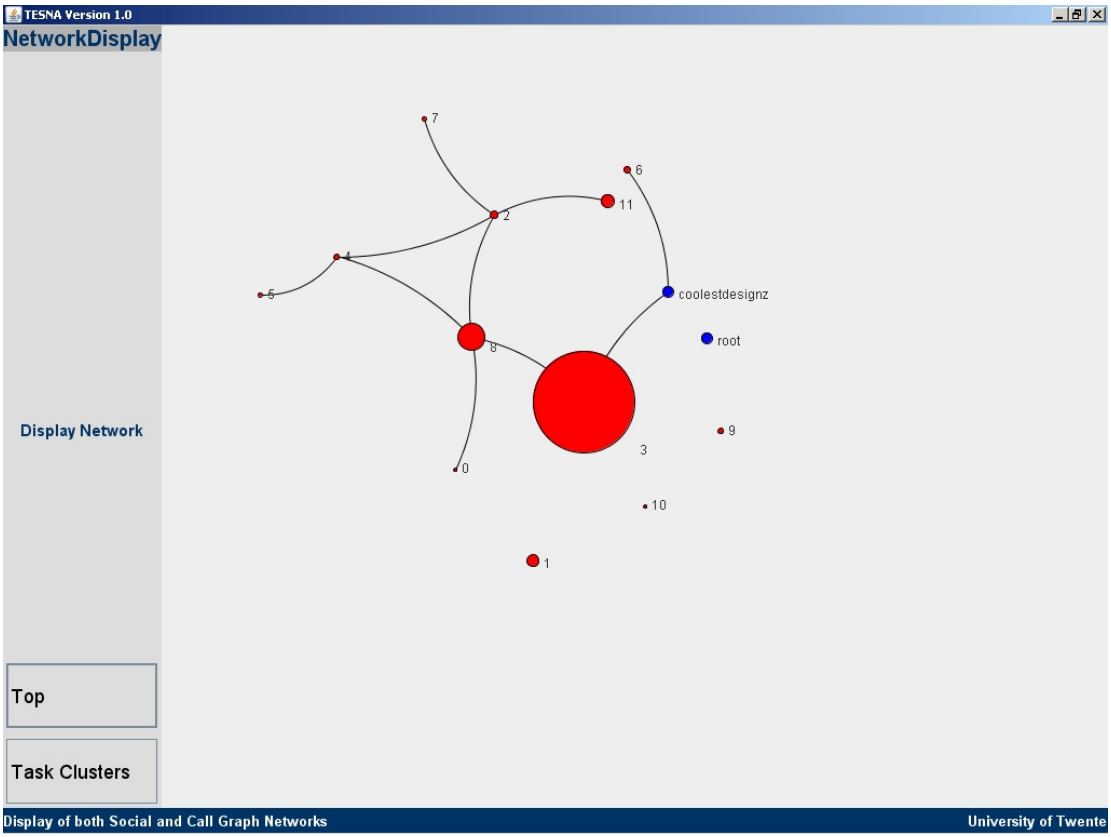

Figure 4: Class Clusters along with Author information for revision 3

Empirical Data

The purpose of this research is to help the software project manager aware of the software development process. So this research is beneficial to projects which are not doing too well (not healthy [5]). So we avoided projects which are completely inactive with no stable deliverables (projects that have already failed). In order to show the occurrences of the two STSCs we searched Sourceforge and selected a project called JAIM. The reason behind selecting JAIM was that it has a low activity percentage for the last year, and has no write activity in the repository (SVN) after 2006. The public forums had 4 messages of which two were welcome messages. The mailing lists were empty. So were the Bug reports, support requests, patches and feature requests. This lack of activity showed that the project was heading towards becoming inactive. Therefore, we wanted to investigate the task structures over the period of the project, and see if we can find the two STSCs in this project. TESNA can only display the dependencies between java modules, hence any of the .html documentation or java script files cannot be seen in the class cluster displays. Therefore, developers working on any of the non-java files will appear to be unconnected in the author-cluster displays. The software releases did not change the dependencies between the components in the releases 3 through 10, so the call graph and consequently the cluster graph of the software remained the same. So we just had to add the changed

author information in the TESNA tool in order to get the picture of which author was changing which module, belonging to the particular cluster.

Discussion

Using the tool TESNA 1 that we have developed we generated the author-cluster diagrams. We analysed the various write revisions in the repository (SVN) in order to detect STSCs mentioned in section 3. The latest revision of the project was 10, so we analysed revisions at equal intervals namely revisions 3, 5, 7 and 10. We have used TESNA to analyse the author-cluster diagrams qualitatively. We think approach is best suited in detecting such STSCs. The first thing we notice in the author-cluster diagrams is that only 3 developers contributed to the development in the revisions 3, 5, 7 and 10 we observed namely, coolestdesignz, root and dingercat though 6 developers were listed as the developers in the project had 6 developers listed. Of these developers coolestdesignz is listed as the project administrator. We notice that no developer other than dingercat has attempted to take responsibility of any module. In revision 3 (Fig. 4), we notice the developer coolestdesignz altering files in the core cluster 3 and the peripheral cluster 6. The developer root is also visible but is not visible changing any file in any cluster in the diagram as he/she would have changed/contributed to a non-java file. Since the file(s) is not in java, it is an .html or java script file which would mean that root is a peripheral or co-developer for this revision. Also, in Figures 4 to 7 (see Appendix), we see the occurrence of FLOSS STSC 1 (section 2) as the peripheral developer root does not move to the core. We notice FLOSS STSC 2 as core cluster developers coolestdesignz and dingercat move to the peripheral region, as coolestdesignz disappears after revision 3 (Fig 4), while dingercat moves from being a core developer to being a co-developer in revision 10 (Fig 7). On calculating the *Binary Core-Periphery Metric* and the *Average Distance Core-Periphery Metric* for the four time intervals, we see a characteristic downward trend in both graphs indicating a shift away from the core. The downward trend in the Binary Core-Periphery metric (Fig. 2) is more pronounced as this metric is more sensitive to core-periphery shifts (as the value decreases exponentially) as compared to the *Average Distance Core-Periphery Metric* (Fig. 3) . Thus we see that for this project the *Binary Core-Periphery Metric* gives a better indication of people moving from core to the periphery of the project.

Conclusion

In this paper, we have tried to come up with STSCs based on FLOSS literature. We have showed a technique (a clustering based display mechanism) that can be used to detect STSCs in FLOSS projects as well as a metric to detect the extent of the STSC. We have tried to demonstrate this technique by detecting STSCs in an open source project JAIM. The project JAIM is in the beta stage of development and has all the signs of joining the ranks of an inactive and failed project in the Sourceforge database. Through the detection of STSCs, we plan to provide the project leader (of JAIM for example) as well as potential interested developers with one more indicator the health of the open source project. In this paper, we display a technique using the actual mining of data from the code repositories to detect STSCs.

Though, there are other ways of detecting the health of an open source community [4, 5], the techniques they use do not display the actual task allocation of the FLOSS project. The other contribution in this paper is to look at the software code and try and define the core and the periphery of the code based on class and function dependencies rather than from the software design (which is not available generally in FLOSS projects). This may have also inadvertently provided one more technique to determine the Core or Periphery developers. Research along the lines of Crowston et.al. [10] is required in order to validate this technique.

We have also come up with two ways of calculating the Core-Periphery metric which could indicate the extent of the STSCs described in this paper. We hypothesize that the greater the number and extent of the STSCs found in the project the worse would be its health. Future work could deal with honing the core-periphery metric by testing it on different FLOSS projects. We could also involve the identification and detection of many more STSCs in different open source projects enabling project managers to manage the FLOSS development process in a better way.

References

1. Mockus, A., R.O.Y. T Fielding, and J. D Herbsleb, *Two Case Studies of Open Source Software Development: Apache and Mozilla.* ACM Transactions on Software Engineering and Methodology, 2002. **11**(3): p. 309-346.
2. Raymond, E., *The Cathedral and the Bazaar.* Knowledge, Technology, and Policy, 1999. **12**(3): p. 23-49.
3. Ye, Y. and K. Kishida, *Toward an understanding of the motivation of open source software developers.* Proceedings of the 25th international conference on Software engineering, Portland, Oregon, IEEE Computer Society, 2003: p. 419-429.
4. Crowston, K. and J. Howison, *The social structure of open source software development teams.* OASIS 2003 Workshop (IFIP 8.2 WG), 2003.
5. Crowston, K. and J. Howison, *Assessing the health of open source communities.* Computer, 2006. **39**(5): p. 89-91.
6. de Souza, C., R. B., J. Froehlich, and P. Dourish, *Seeking the source: software source code as a social and technical artifact*, in *GROUP '05: Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*. 2005: New York, NY, USA. p. 197--206.
7. Yamauchi, Y., et al., *Collaboration with Lean Media: how open-source software succeeds.* Proceedings of the 2000 ACM conference on Computer supported cooperative work, 2000: p. 329-338.
8. Amrit, C. and J. van Hillegersberg, *Detecting Coordination Problems in Collaborative Software Development Environments.* Information Systems Management, 2008. **25**(1): p. 57 - 70.
9. Coplien, J., O. and N. Harrison, B. , *Organizational Patterns of Agile Software Development*. 2004: Upper Saddle River, NJ, USA.
10. Crowston, K., et al., *Core and periphery in Free/Libre and Open Source software team communications.* Proceedings of the 39th Annual Hawaii International Conference on System Sciences-Volume 06, 2006.
11. Fernandez, C.I.G., *Integration Analysis of Product Architecture to Support Effective Team Co-location.* ME thesis, MIT, Cambridge, MA, 1998.
12. MacCormack, A., J. Rusnak, and C.Y. Baldwin, *Exploring the structure of complex software designs: An empirical study of open source and proprietary code.* Management Science, 2006. **52**(7): p. 1015-1030.
13. Li, B., et al., *Matrix-based component dependence representation and its applications in software quality assurance*, in *ACM SIGPLAN Notices*. 2005: New York, NY, USA. p. 29--36.
14. Cataldo, M., et al., *Identification of coordination requirements: implications for the Design of collaboration and awareness tools*, in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. 2006, ACM Press: Banff, Alberta, Canada.

Appendix

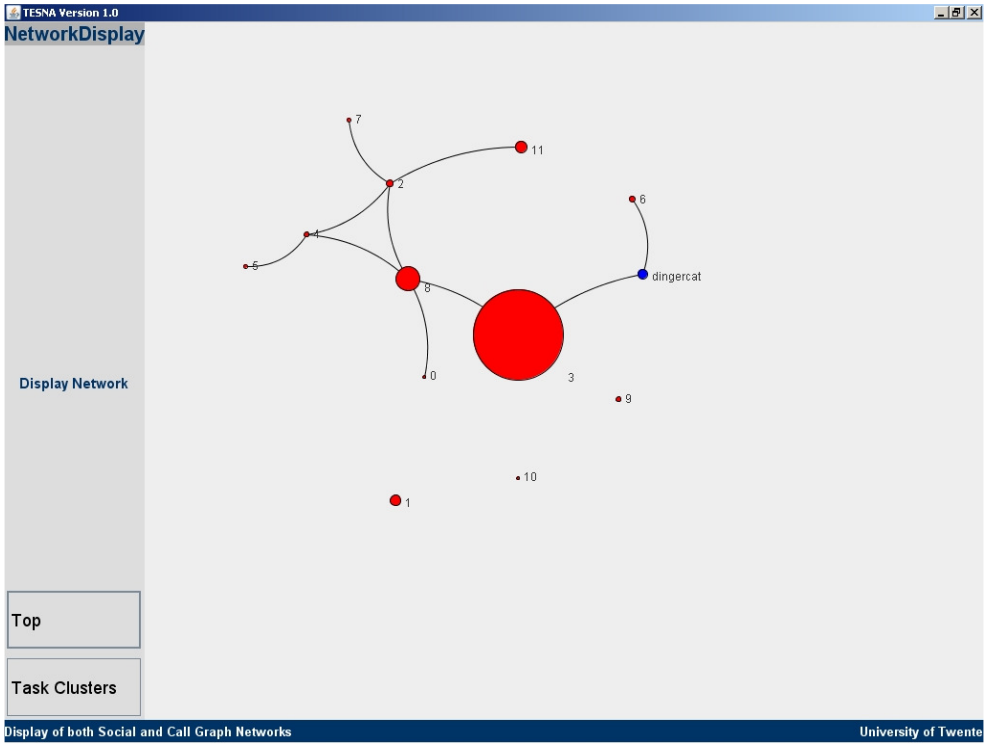

Figure 5: Class Clusters along with Author information for revision 5

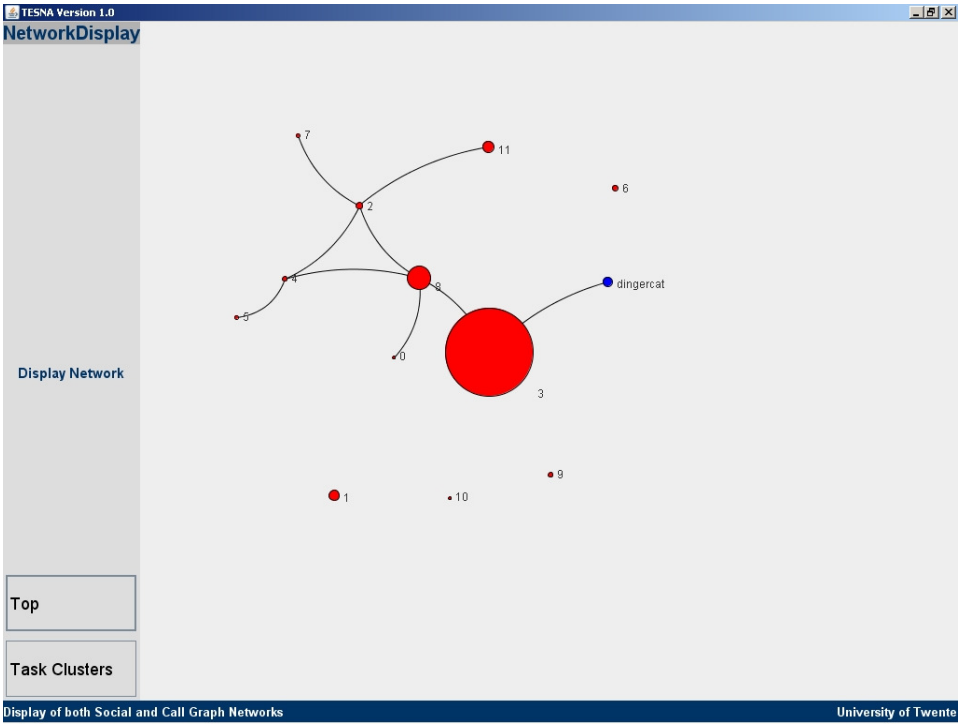

Figure 6: Class Clusters along with Author information for revision 7

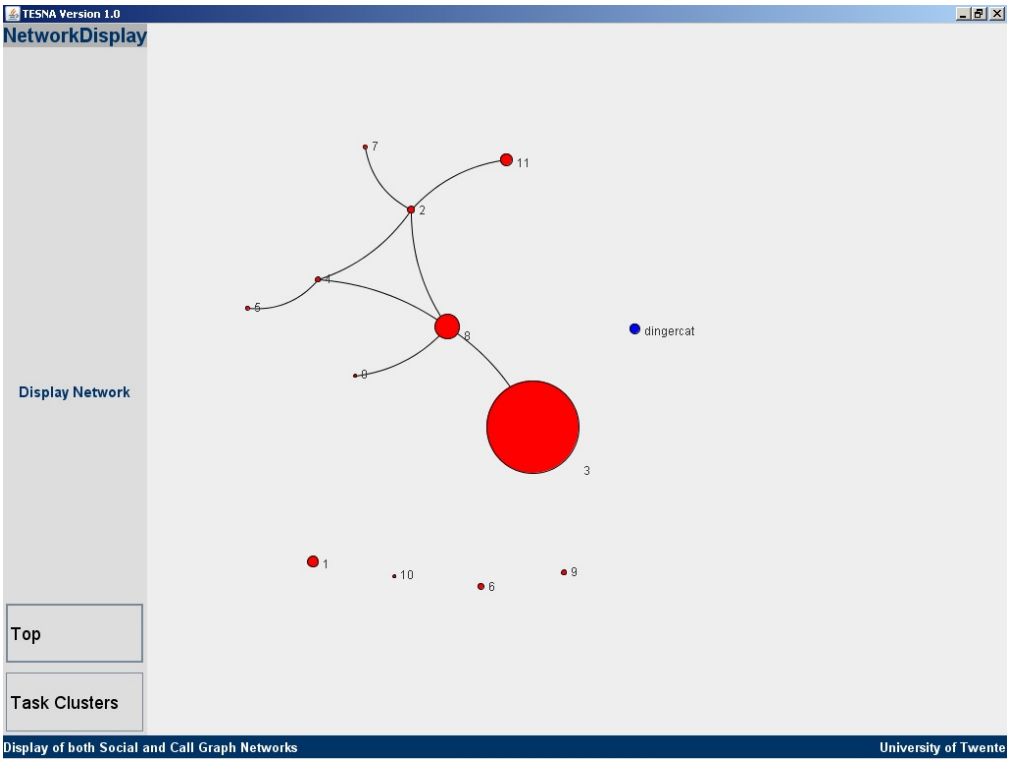

Figure 7: Class Clusters along with Author information for revision 10