Using Formal Concept Analysis to Construct and Visualise Social Hierarchies of Software Developers

Michel Wermelinger[1], Yijun Yu[1] and Markus Strohmaier[2]
[1] Department of Computing and Centre for Research in Computing, The Open University, UK
[2] Knowledge Management Institute, Graz University of Technology, and Know-Center Graz, Austria

Abstract

Interest in the human aspects of software engineering has grown in the past years. For example, based on activity logs in software artefact repositories, researchers are recommending who should fix a bug for a certain component. However, existing work largely follows ad-hoc approaches to relate software artifacts to developers and rarely makes those socio-technical relations explicit in a single structure. In this paper we propose a novel application of formal concept analysis, in order to overcome those deficiencies. As a case study, we construct and visualise different views of the developers who fix and discuss bugs in the Eclipse project.

1 Motivation

Software engineering is inherently a socio-technical endeavour, as pointed out by Conway [4]) among others. The rise of global software development projects and the research opportunities provided by the rich open source repositories have led to an increased interest in the social side of development, as the ICSE workshops on socio-technical congruence and human aspects of software engineering testify.

Although much work exists on exploring the information about developers and other contributors contained in software repositories, there is actually not much work on explicitly showing the overall socio-technical relations between people and software artifacts in an explicit way. Sometimes, the relations are implicit and given to users one at a time (e.g. based on the file the user is editing). Other times, the socio-technical relations are just an intermediate step to obtain further relations between only artifacts or only people (e.g. who collaborates with whom), and only those relations are shown. In the few cases where socio-technical relations are explicitly shown, they are usually drawn as graphs and, depending on the layout algorithm, it may not be easy for analysts to visually recognise any relevant connections, e.g. which developers worked on most source code files. One of the underlying reasons for this state of affairs is that due socio-technical relation graphs do not scale well due to the large amount of artifacts and people involved in most software projects.

We therefore asked ourselves if socio-technical relations could be presented in a different way, that would be explicit, compact, and yet intuitive. To avoid reinventing the wheel, we looked for existing and well-established network techniques and tools that could help reduce the learning curve for those wishing to analyse such relations. We decided to try applying formal concept analysis in order to automatically construct and visualise social structures in a more hierarchical way, which would immediately lead the user to the most important developers, namely those appearing at the top of the hierarchy. To put our idea to the test, we did an exploratory study of the social hierarchy of those developers that discuss and fix bugs in Eclipse. We constructed various hierarchies, both over the same and different Eclipse releases, in order to obtain, on one hand, different views of the same social reality and, on the other hand, the same view over an evolving social reality.

2 Related work

There has been much work on mining social networks of developers from source code [5, 6], e-mail archives [3], and bug reports [1]. However, all these works have a different aims from ours. Whereas we are interested in obtaining a general approach to explicitly construct and visualise socio-technical relations that will enable different questions to be answered, the cited researchers build custom graphs for the particular research question at hand and those graphs are either implicit (i.e. not shown to the user) or only have one type of nodes (either artifacts or people).

Nevertheless, we have taken an important lesson from the cited works: obtaining social structures from configuration management systems (like CVS) or from source code may leave many contributors out of the picture as they do not have commit rights on the repository or they do not contribute by writing code, but instead by discussing on e-mail lists, for example.

Formal Concept Analysis (FCA) is a graph-theoretic approach to categorization based on mathematical order and lattice theory [9]. Given a set of objects O, a set of attributes A, and a matrix stating which attributes each object has, FCA will first construct all *concepts*, i.e. all pairs $\langle o, a\rangle$ such that $o \subseteq O$ is the set of objects that share the attributes $a \subseteq A$. The objects o are the *extent* of the concept, whereas the attributes a are the *intent* of the concept. The concepts will then be organised into a lattice, following the intuition that general concepts have larger extents. Formally, $\langle o, a\rangle \leq \langle o', a'\rangle$ if $o \subseteq o'$. Since objects o' share attributes a', the subset o will obviously share the same attributes and possibly more. Hence, if $\langle o, a\rangle \leq \langle o', a'\rangle$ then $a \supseteq a'$. In other words, as we move upwards in the lattice, the extent increases and the intent decreases.

FCA has been used in software engineering mainly to complement traditional static code analysis in order to obtain different relationships between code artifacts [7], e.g. to classify them into cross-cutting higher level features (concepts).

3 Proposed Approach

The novel approach we propose is to view software artifacts as objects and people as attributes. In that way, the concepts computed by FCA will be clusters of artifacts that are associated to the same people. Moreover, the lattice will implicitly correspond to a hierarchy, in which those people associated to more artifacts will appear in the top levels of the lattice, thus indicating their importance in the project. In other words, FCA will give us for free the clustering of artifacts and people, an ordering of those clusters, and an intuitive view of such ordering. Moreover, computing the lattice over different releases of the system will allow us to see how the clusters and their ordering evolves. All this, put together, can then be used for various purposes.

For example, consider that the objects are the source code files, the attributes are the developers, and the matrix states which developers worked on which files for a give period of analysis. Hence, each concept will group all files that, over that period analysed, were changed by the same group of developers. The top level concepts will show who are the developers working on most files and hence are likely to have the widest knowledge about the system. Inversely, the low level concepts will show those developers that specialise only on a few files and hence may have more in depth knowledge for those parts of the source code.

Furthermore, concepts with small intents (i.e. few developers) point to parts of the system that may be at risk of becoming legacy, if those developers leave the project. However, due to the way the concepts are ordered by FCA, a manager can quickly see which developers are likely to be the best replacement for those leaving, simply by looking at the intents of the immediate children of the critical concept. To see the reason, consider a concept $c = \langle\{fileA, fileB, fileC\}, \{John\}\rangle$. If John leaves the project, who can quickly replace him? All the immediate children $c_i \leq c$ in the lattice have an extent that most closely matches the extent of c, e.g. $c_1 = \langle fileA, fileB\}, \{John, Mary\}$ and $c_2 = \langle\{fileA, fileC\}, \{John, Peter\}$. Hence, the intent of each c_i includes those developers (besides John) who will have to become acquainted with the least number of files in order to match John's current expertise. They are thus the potentially best candidates to replace John in the project.

4 Exploratory Study

To explore the application of FCA to uncover social hierarchies, we chose to use bug reports in order to avoid the limitations associated to using source code or a configuration management system (Section 2). We selected Eclipse as case study because: we already had some experience mining it [8]; a Bugzilla database is available[1] for a sufficiently long history for social changes to become apparent; the lead of IBM allows some social continuity to be traced throughout history.

First we extracted, for each of the 101966 bug reports processed, its unique id, the Eclipse component for which it was reported, and the people associated to the bug: the reporter, the current assignee (i.e. the person fixing the bug), and the (zero or more) past discussants of the bug. Each different role can be understood to cover a different aspect of communication in software development. All these stakeholders are given as email addresses in the database. We have not yet filtered e-mail aliases, as this is just a preliminary exploration of the data set. However, as a rough estimate we computed how many e-mails shared the user name but had a different domain name (e.g. `user@ibm.com` and `user@gmail.com`) and found this to be the case for 7% of reporters and discussants, and for 3% of assignees.

With this information we constructed a graph consisting of three types of nodes: people P, bugs B and software components C. There is a directed arc from person p to bug b, if p reported, worked on, or discussed b. There is a directed arc from b to c if bug b was reported for component c. Next we created a bi-partite graph PC: an arc from person p to component c will be weighted with the number of bugs of c that p is associated with, in other words, the number of paths from p to c in the original PBC network.

We repeated the construction of the PBC and PC networks for several releases of Eclipse, selecting for each re-

[1] `http://msr.uwaterloo.ca/msr2008/challenge`

lease all bugs reported up to the release's date. The cumulative effect over releases allows us to see which developers become more involved (i.e. are associated to more components) and which ones remain at the same level.

To make a meaningful analysis, it is necessary to avoid 'noise' due to people that had only a very small intervention in the project. We therefore introduced a threshold k: arcs with a weight less than k will be removed from the PC network, and so will any nodes that become detached.

We used `awk` and the relational calculator Crocopat [2] to write scripts that, given a subset of the three roles, a release number, and a value for k, will output a comma separated value representation of the node adjacency matrix of $PC(k)$ for those people that fulfill the given roles. This output file is fed into the FCA tool `ConExp`[2] (short for Concept Explorer) to generate the concept lattice.

For example, the lattice for $PC(10)$ at release 1.0, and only taking assignees into account, is represented in Figure 1. Because of the way concepts are ordered in a lattice, ConExp uses *reduced labelling* to avoid cluttering the diagram, i.e. it only shows for each concept the objects (resp. attributes) the concept has in addition to its descendants (resp. ancestors). For example, the extent of the concept labelled with assignee Kai-Uwe Maetzel is just {`platform:ui`,`jdt:ui`}, the union of the extents of the descendants. Similarly, the intent of the concept labelled with object `jdt:ui` is {Kai-Uwe Maetzel, André Weinand, akiezun, ..., Dirk Baeumer}, the union of its ancestors' attributes and its own.

We point out that a static screenshot does not do justice to ConExp, which is an interactive tool that allows users to properly explore the lattice. For example, pop-up windows can show the complete extent of any node, without users having to do the union in their head. It is also possible to hide the object or attribute labels or drag them to the side, to make the lattice less cluttered.

Returning to Figure 1, we can see that there is actually no proper hierarchy, the lattice being rather flat: most developers were assigned to a single component. The exceptions are Kues, Radloff, Maetzel, Weinand, and Klicnik, each one having worked on two components. Some components have only one single developer assigned to more than 10 bugs, while others have six or more. This might be just an indication that some components require many more bug fixes than others, but it might also be cause for concern if those single developers with expertise for a given component leave the project. The use of FCA to cluster developers around artifacts can quickly point out potential problematic hotspots with too many or too few developers, but whether there is actual cause for concern can only be established by consulting other information sources. Last but not least, Figure 1 clearly shows some geographic clustering: all IBM Switzerland developers handle bugs in `jdt:ui` and IBM France only handles bugs in `jdt:core`.

If we now fast forward to the date of release 3.0, and increase the threshold to 100 bug reports in order to take the accumulation of bugs into account, but keep looking at the same role (assignees), we obtain a lattice (omitted for space constraints)that, interestingly, has not changed much in certain respects. For example, the geographical division of labour is largely kept, and most developers still specialise on a single component, but for example Daniel Megert has 'climbed up the social ladder' and moved to the top level of the hierarchy, contributing to at least 100 bug fixes for each of three components. It is also interesting to see that the three top bug handlers for `platform:debug` also deal with `platform:ant` and `jdt:debug` which may point to certain particular characteristics of those components.

Finally, keeping the threshold and release but switching to the discussant role, we get a completely different lattice that has fewer objects and attributes than the assignee hierarchy. Moreover, a quick browsing confirms that the active discussants are largely a subset of the active developers. Together, these facts imply that a developer does not discuss all bugs they are assigned to. Hence, only few people discuss more than 100 bugs for a single component and therefore less people and components appear in this lattice. For example, Kai-Uwe Maetzel, who heavily contributed to two components, does not appear in the discussant hierarchy. It is also interesting to note that most developers do not just discuss the bugs of the components they specialise in. For example, John Arthorne and Erich Gamma heavily discuss `platform:ui` bug reports, besides those for the components they fix. This may point to tight dependencies between those pairs of components.

5 Concluding remarks

This paper makes two contributions: a new idea, namely the novel application of formal concept analysis (FCA) in order to obtain social hierarchies, and some emergent results about the Eclipse project. The results so far are promising about the kinds of information and relationships that are easily apparent from looking at the various lattices we constructed, showing different views of the same release or comparable views of different releases. General socio-technical evolution patterns can't be formulated about Eclipse's overall development at this point of our preliminary exploration, but we will continue our study.

We do not claim that lattices should replace the more common 'flat' social networks seen in existing work, because it does not always make sense to organise data in a hierarchical way. However, using FCA has several fundamental advantages over the bi-partite or nested graphs used so far, which usually have one node for each artefact and

[2] `http://sourceforge.net/projects/conexp`

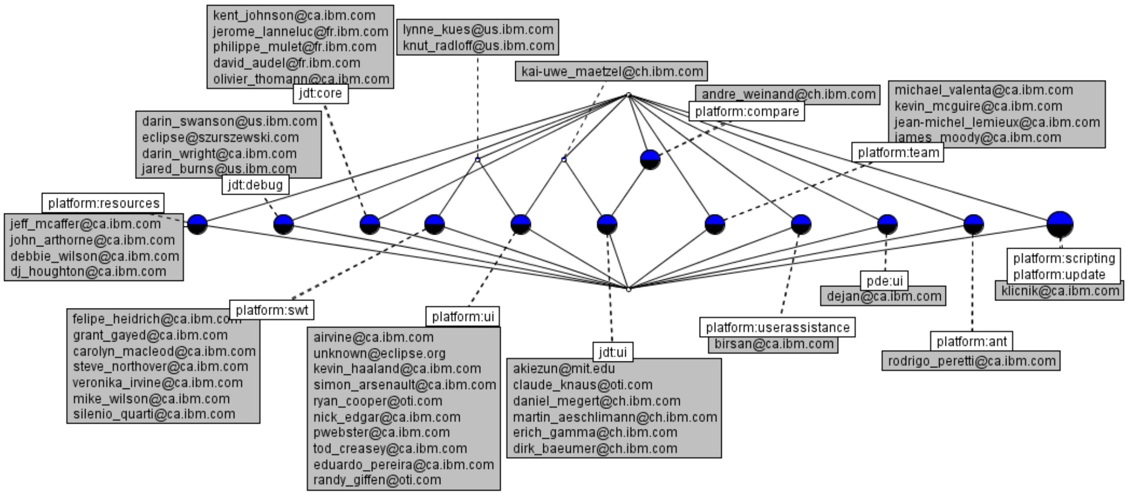

Figure 1. Social hierarchy of assignees at release 1.0 with $k = 10$

person.

1. By clustering multiple artifacts and people into the same node, lattices are much more compact than the corresponding graphs.

2. By merging artifacts and their associated people into the same concept, the socio-technical relations become much clearer than in a bi-partite graph that requires lots of arcs to depict the same relations.

3. Lattices have a systematic layout that intuitively maps the vertical dimension to our mental expectations about hierarchies, thereby reducing the learning curve necessary to meaningfully explore the lattices. By contrast, understanding a bi-partite graph (e.g. finding the most important people) may be difficult due to the layout algorithm used.

4. The approach is general and not dependent on the artifacts considered and how they are associated to people. By contrast, in existing approaches the graphs have different semantics and are visualised differently depending on the artifacts and socio-technical relations analysed.

We therefore hope that not only developers and managers will find this flexible approach of relating software artifacts and people useful to solve practical questions in their daily work (e.g. who should replace someone leaving the project), but that it will also help researchers to further explore and make explicit the deep socio-technical relationships that exist in software engineering.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. 28th Int'l Conf. on Software Engineering*, pages 361–370, 2006.

[2] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Trans. Software Eng.*, 31(2):137–149, 2005.

[3] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Chapels in the bazaar? Latent social structure in OSS. Presented at the 1st Workshop on Socio-Technical Congruence, Leipzig, Germany, 2008.

[4] M. Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[5] C. de Souza, J. Froehlich, and P. Dourish. Seeking the source: software source code as a social and technical artifact. In *Proc. Int'l ACM SIGGROUP Conf. on Supporting group work*, pages 197–206. ACM, 2005.

[6] J. D. Herbsleb and R. E. Grinter. Splitting the organization and integrating the code: Conway's law revisited. In *Proc. 21st Int'l Conf. on Software Engineering*, pages 85–95. IEEE Computer Society Press, 1999.

[7] P. Tonella. Formal concept analysis in software engineering. In *Proc. 26th Int'l Conf. on Software Engineering*, pages 743–744. IEEE Computer Society, 2004.

[8] M. Wermelinger, Y. Yu, and A. Lozano. Design principles in architectural evolution: a case study. In *Proc. 24th IEEE Int'l Conf. on Software Maintenance*, 2008.

[9] R. Wille. *Formal Concept Analysis as Mathematical Theory of Concepts and Concept Hierarchies*, pages 1–33. 2005.