Reverse Engineering Co-maintenance Relationships Using Conceptual Analysis of Source Code

Scott Grant James R. Cordy David B. Skillicorn
Queen's University, Kingston, Canada
{scott, cordy, skill}@cs.queensu.ca

***Abstract*—In this work, we explore the relationship between topic models and co-maintenance history by introducing a visualization that compares conceptual cohesion within changelists. We explain how this view of the project history can give insight about the semantic architecture of the code, and we identify a number of patterns that characterize particular kinds of maintenance tasks. We examine the relationship between co-maintenance history and concept location, and visualize the distribution of changes across concepts to show how these techniques can be used to predict co-maintenance of source code methods.**

I. Introduction

Concept location is the act of identifying the set of source code fragments in a software system that implement a particular concept. Concepts intentionally have a very loose definition in program comprehension, to allow for a wide and subjective interpretation of the relationships between code fragments. In topic modelling, the definition of a topic is also intentionally vague. Concept models are not generally designed to directly relate to the human-oriented notion of concepts. They identify correlations based solely on the occurrence of terms in the documents, and it is assumed that these correlations will agree with our expectations.

This paper uses a visualization to explore the relationship between topic models and co-maintenance history, in an attempt to identify a human-oriented link between the concepts found by topic models and software code. We work at the changelist granularity to show that code fragments that are modified together often share a conceptual relationship as discovered by concept location techniques such as Latent Semantic Indexing and Latent Dirichlet Allocation.

II. Background

A topic model is a statistical model used to identify a set of latent topics in a data set. The fundamental premise behind a topic model is that there is some correlation between the tokens in the data set that can be explained by a mathematical relationship between them, and that these relationships can be extracted as topics. One example of a topic model is Latent Dirichlet Allocation (LDA) [2], a generative model that assumes the data set was derived from a prior topic distribution over the data. In general terms, LDA assumes the existence of a number of topics that can be used to relate elements of the data set to one another. If two pieces of data are strongly related to the same topic, they are likely to be very similar to one another. We would like to understand the nature of the similarity to make better use of topics during software development and maintenance.

The set of documents used in the concept models consists of the complete set of source code methods in the package. Each of these source code fragments has some degree of membership in each of the topics found by the concept model. In each of these models, we identify the largest value in the set of numbers that describe membership in the topics, and use that as the most significant concept for a document.

III. Method

In our visualizations in this paper, we examined three different systems: the Apache *httpd* webserver written in C, the *Django* web framework, written in Python, and the *Hadoop* distributed computing project, written in Java. In each case we used either *Subversion* or *Git* repository logs to extract checkins. To generate Latent Dirichlet Allocation models we used either *Mallet* [7] or *GibbsLDA++* [8], and to generate Latent Semantic Indexing models we used Matlab.

Our custom tool to generate these visualizations relies on two pieces of data. First, a snapshot of the project is taken and a model is generated from the extracted functions. A parser extracts all of the functions from the package, strips out comments, and splits apart the camel-case and underscore-separated compound tokens to give a larger set of meaningful words. Next, the full revision history is extracted from the source code repository logs. From these logs, all of the changelists that modify a source code function are identified. As we traverse back in the project, some functions are moved, renamed, or modified, and we attempt to associate as many of them as possible. For each of the changelists with meaningful code fragment changes, we generate a row of data in the visualization. We take the list of modified code fragments, identify the related concepts for each of the fragments, and plot the row based on those results. If there are many code fragments from the same concept, larger circles are drawn, and vice versa.

Each visualization is generated as an interactive HTML page, based on the input from a source code repository and a secondary source such as a topic model. The rows of the display are the list of relevant changelists over the history of the project, starting at the oldest, and progressing forward towards the newest revisions at the bottom. Each column in the display corresponds to one of the concepts described by the model. The circles of varying size and colour in each row

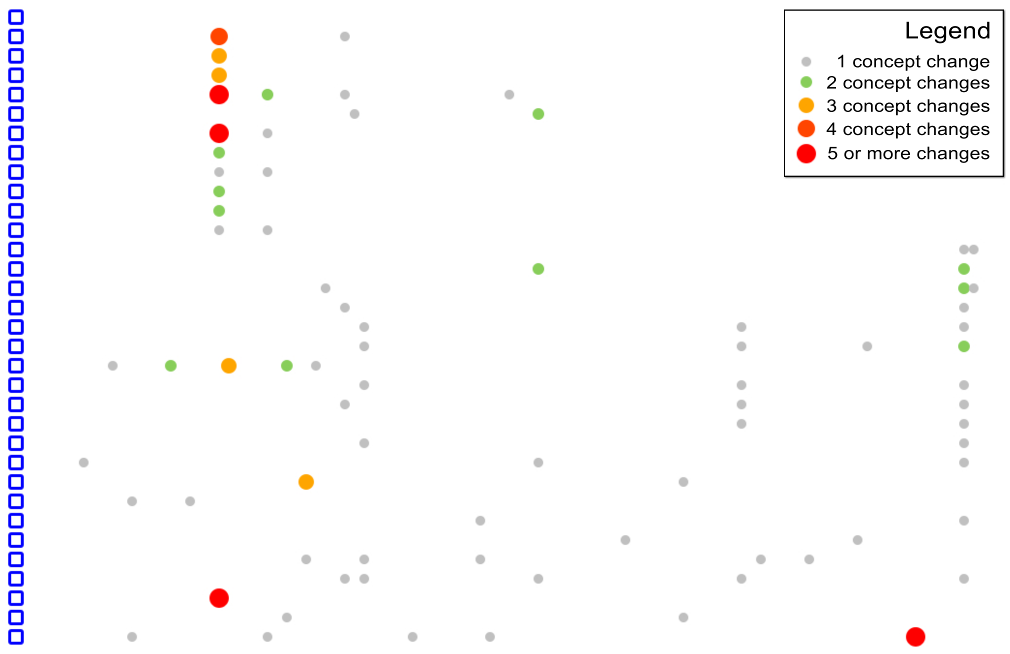

Figure 1. A full view of the visualization for approximately 35 consecutive changelists in httpd's history. Each blue square on the left defines a horizontal row corresponding to a changelist. Each column corresponds to a concept, and each coloured circle represents some number of modified code fragments from that concept. The size and colour of each circle shows how many code fragments from that concept were modified in this particular changelist.

are an indication that some code fragments were modified in that changelist. The horizontal location of the circle indicates the concept in which the modifications were made.

Figure 1 shows a demonstration of this visualization. We plot a portion of the change history for the Apache *httpd* webserver using an LDA model with 100 topics. Each row in the table corresponds to a changelist, with the oldest changes at the top of the table, and the most recent at the bottom. The circles are colour-coded and sized to indicate the number of code fragments they represent. In this demonstration, we use the largest red circles to indicate five or more code fragments that are associated with a concept, ranging down to a single small grey circle for one code fragment. Detailed information can be obtained by hovering the mouse over elements on the screen. The blue squares on the left side are changelist indicators, and hovering over one will give the revision id, author, date of revision, size of revision, and the checkin message. Hovering over any of the coloured circles will list the functions modified in that changelist that are associated with a shared concept.

In general, the results are quite sparse. Although there are many changelists with a small number of functions owned by different concepts, it appears that many of the larger changes are, for the most part, owned by some concepts that describe the modified area. This seems to indicate a correlation between co-maintenance history and conceptual clustering. If we had instead discovered a relatively small amount of conceptual clustering within changelists, it would be an indication that concepts did not capture a human-oriented perspective of revision history.

IV. Patterns

In our analysis of systems, we identified a number of patterns that appear regularly. In this section we classify and show by example how these patterns are found, and discuss why they appear and how they correspond to particular maintenance activities.

Vertical columns of circles indicate related source code methods in the traditional topic modelling sense, where two code fragments found in the same concept are likely to be highly conceptually related to one another. We have observed many instances of feature development that appear as coloured vertical bands, and believe this is an easy way to examine the history of a project with visual cues about where and when certain features were implemented.

Horizontal rows of circles indicate larger system-wide changes, or modifications that necessarily cut across concepts. The justifications for aspect-oriented programming are also appropriate here, and in the case of feature development, large vertical bands followed by shallow horizontal ones indicate feature development followed by implementation.

A. *Feature Development*

Iterative and incremental development is a common software development practice [5]. When combined with source code versioning systems, the addition of a new feature is often characterized by a long series of related changelists. When viewed in this visualization, a sequence of related changelists often shows up as a vertical column of coloured circles. This indicates a series of modifications to source code methods that are conceptually related in the topic

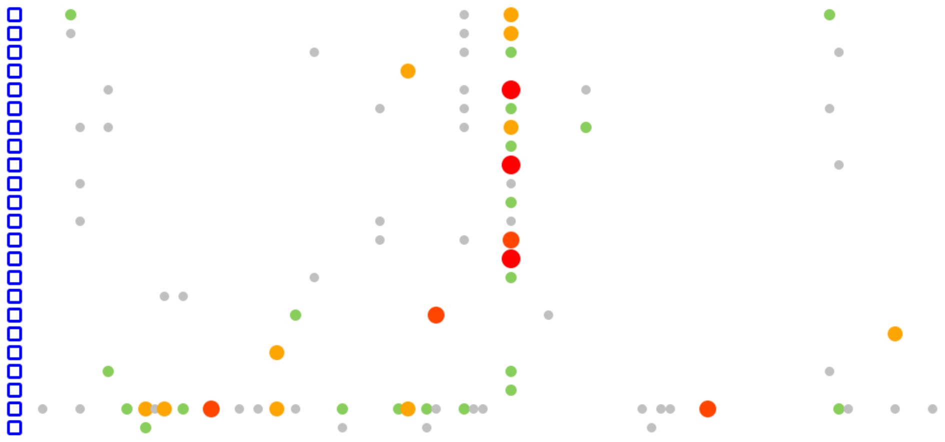

Figure 2. A view of feature development, characterized by a long vertical band of related changes, as seen in the center of the diagram. This set of changes involved an overhaul of *mod_include*'s filter parser. Shortly after the feature was implemented, a large change spanning many concepts was added. This new changelist was a system-wide move from the old feature to the new one.

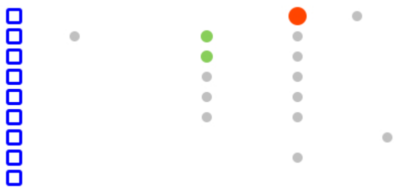

Figure 3. A subset of the visualization showing an example of a commit followed by patches. This pattern is seen throughout the code, and indicates a single larger checkin, followed by small optimizations or bug fixes. This example, starting from the orange circle on the right (revision 984188) added a feature to *mod_proxy* that improved communication. The next five checkins fixed comments and code, added some additional checks, and refactored code slightly.

model, and is a good demonstration that topic models like LSI or LDA are capturing co-maintenance relationships.

In Figure 2, the upper-most changelist (revision 101036) is described as the start of a major overhaul of *mod_include*'s filter parser. A set of consecutive changes introduces a code wrapper, removes old code, refines the API, and improves the efficiency and cleanliness of the code. Interestingly, shortly after the vertical feature implementation line, a horizontal pattern can be seen (revision 101154) that switches to the new API. This large change necessarily touches many concepts, and is described in the actual changelist as "switch to APR 1.0 API". After a long run of feature changes, a larger aspect-like change was submitted to implement the feature in the code, and so the entire feature addition and move to the new system can be seen in the visualization.

If we return to Figure 1, the view of the system also gives at least one example of a feature in development. The long run of vertical red, green, and orange dots in the upper left is described in the changelists as a series of modifications to the *apr_send* functionality (revision 88625), and the related code changes that go with it. The long vertical run on the bottom right corner, and the associated modifications found in other concepts, are described as a significant modification to the FTP proxy code (revisions 88721 and forward).

B. Commit, Patch

In our visualization, and in our review of the changelist history, it is common to find instances of a single large conceptual change followed by several smaller ones. Figure 3 provides an example with two instances of larger checkins followed by a series of smaller revisions. Examples like this often show up visually as a larger green, orange, or red circle, followed by a trail of grey circles. This pattern is seen throughout the code, and indicates a single larger checkin, followed by a sequence of small optimizations or bug fixes.

In the history represented by Figure 3, a modification was made to *mod_proxy* that improved communication handling (large red circle). The next five checkins fixed comments and code, added some additional checks, and refactored code slightly. Other examples include data structure modifications that require additional fixes after submission, or feature propagations that can't be made in a single large submission.

This type of modification is often seen as a subset of the feature development pattern from Section IV-A, where incremental changes are made, tested, and patched. In fact, it may be that this pattern is just a specialized version of feature development at a much smaller scale.

C. Co-maintained Concepts

One surprising artifact of our visualization was the relatively low frequency of co-maintained concepts. More specifically, it is much more common to find changes that are either localized to a single topic, or spread out across many topics. It is very rare to find a changelist with a large number of method modifications that are evenly distributed across two or three concepts. Instead, changes with a large number of method modifications are usually concentrated in a single concept (possibly with some sparse distribution in other concepts), or widely spread out like an aspect.

We had considered the possibility that large pairs of parallel vertical columns would emerge, indicating a strong relationship between two concepts. This does show up from

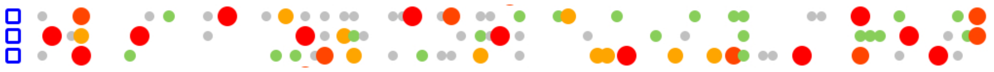
Figure 4. An example of aspect development. These three changelists, starting with revision 95149, made a modification to the behaviour of several related logging functions. These functions are found throughout the code, and as a result, we see a very wide range of concepts affected by the modification.

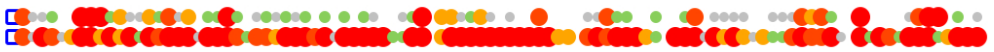
Figure 5. An example of a global change. These two changelists (revisions 332305 and 332306) involved a global removal of tabbed whitespace in favour of spaces. Another similar changelist (revision 88019) renamed various functions.

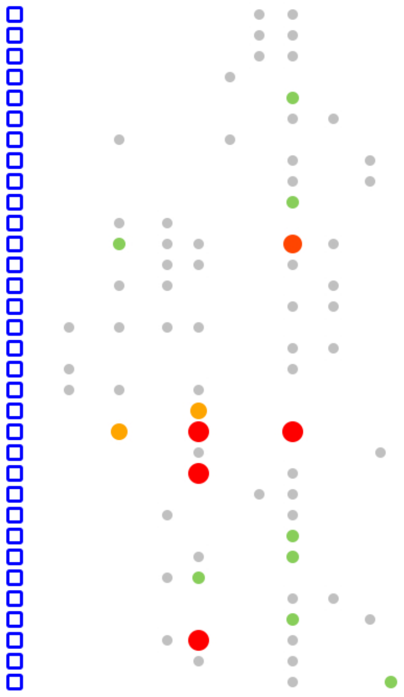
Figure 6. A subset of the visualization showing an example of co-maintained concepts. In this example, the developers were working with some communication code dealing with a proxy worker pool (revision 104557 and onward). A number of code cleanup submissions were made, along with some data structure changes. The largest vertical line along the right side of the diagram is made up of modifications to the actual worker pool code (*ap_proxy_get_balancer*, *ap_proxy_get_worker*, and so on), and other vertical lines include proxy connection code and socket functions.

time to time, as in Figure 6, but it is much more frequent to see changes isolated to a single concept, or distributed across a wide range of concepts. In Figure 6, it appears that the development of a new feature that is closely tied to old functionality in some way, such as the addition of a proxy worker pool in *httpd* instead of a single worker, may result in a distinct concept. Occasionally such pairs or groups of vertical runs will occur, but it is rare. It seems that developers much more commonly work on either a single conceptual area or on a cross-cutting one when making single checkins.

D. Aspects

Aspect-Oriented Programming is a paradigm that explicitly attempts to capture the notion of cross-cutting concepts (or business concerns) as separate *aspects*. In this context, a changelist that modifies an aspect might be seen as one that modifies code fragments across a wide range of concepts.

Figure 4 shows one example of aspect development with a fairly wide distribution across topics. In the figure, three changelists (revisions 95149, 95150, and 95151) are made to modify the behaviour of the *ap_log_error* and *ap_log_prerror* logging functions. These functions are distributed throughout the code, and as a result, we see a very wide range of concepts affected by the modification. Logging is a classic example of a cross-cutting concern, and from our observations, aspects cross topics as well.

Other research has also suggested that aspects may be latent topics with high scattering entropy [1], which may suggest that true source code aspects would fall under a single topic. Although we have observed no clear example of this yet, it is a compelling idea.

E. Global Changes

Larger system-wide changes often appear as wide horizontal bars that modify many concepts. These are usually not indicative of feature or concept changes, but instead relate to system wide syntax changes or large renamings. In *httpd*, we only find a small number of instances of these large changes. Each one deals with a superficial system-wide modification, such as the removal of tabbed spacing, and holds relatively little value from a maintenance standpoint.

Figure 5 shows an example of a large global change. It stands out very clearly in the visualization, and ends up affecting nearly all of the concepts discovered by the model. From our observations, these global changes have no meaningful functional impact, even though they are the most distinct visual features in the display.

V. Model Comparison

In Figure 7, two images of the same approximately thirty changelists are presented as seen in our visualization for LSI (above) and LDA (below). Each model was generated using 100 topics, and the same snapshot of the code was used to generate each model.

When we looked at the average cluster size for models generated on our data sets, on average, LDA achieved a higher number of code fragments per cluster than LSI. For example, in *httpd*, the average cluster size across the entire code history using LDA was 1.76, compared to an average cluster size using LSI of 1.35.

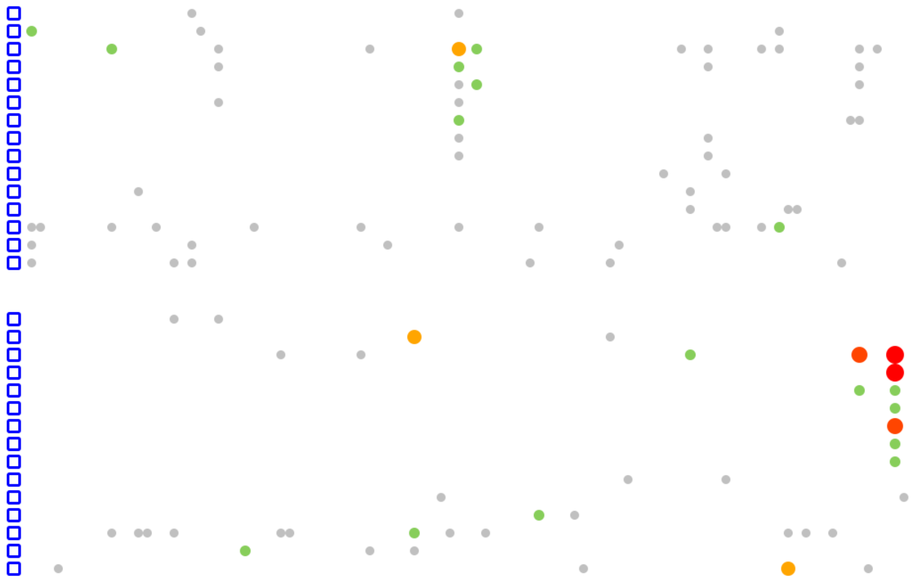

Figure 7. Two visualizations of a historical subset of the same software system, modelled with Latent Semantic Indexing with 100 topics (above) and Latent Dirichlet Allocation (below).

There are many reasons why LSI may be less effective at clustering based on co-maintenance in this example. The ideal number of topics to capture co-maintenance relationships may (and probably does) vary between LSI and LDA. However, we experimented with a range of topic counts for each system, and were unable to obtain better clusters using LSI when compared with LDA.

VI. Related and Future Work

Techniques that aid in the visualization of code clones share many similarities with our technique. In Livieri et al. [6], a thorough examination of code clones over large projects is undertaken. In the study, a colourized heatmap is provided for code clone coverage, with scatterplots for easier review. Göde and Koschke [4] have also devised some very clever visualizations to show the evolution of clones and clone groups over time. Our own visualization came about from a related demonstration by Cordy for the use of structural scatterplots in observing source code clones [3].

The majority of our research work using this visualization has centered on the relationship between co-maintenance history and the topics identified by Latent Dirichlet Allocation. We wanted to demonstrate that code fragments that were maintained together were likely to be conceptually related to one another. The maintenance patterns we have observed are also very interesting, and we plan to do more research to identify what other patterns may exist, and what the known patterns actually tell us about the kinds of changes occurring in the source code.

VII. Conclusions

We have introduced a visualization designed to explore the clustering of features over the changelist history of a project. With the visualization, we observed a relationship between topic models and co-maintenance history by identifying and cataloguing patterns that characterize a number of particular kinds of maintenance.

Acknowledgements

This work is supported by the Natural Sciences and Engineering Research Council of Canada, by the Ontario Graduate Scholarship Program, and by an IBM Center for Advanced Studies Fellowship.

References

[1] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya, "A theory of aspects as latent topics," in *Proc. OOPSLA '08*, 2008, pp. 543–562.

[2] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, 2003.

[3] J. R. Cordy, "Exploring large-scale system similarity using incremental clone detection and live scatterplots," in *Proc. ICPC '11*, June 2011, pp. 151–160.

[4] N. Göde and R. Koschke, "Studying clone evolution using incremental clone detection," *J. Softw. Maint. and Evol.: Res. and Practice*, 2011 (to appear).

[5] C. Larman and V. Basili, "Iterative and incremental developments. a brief history," *Computer*, vol. 36, no. 6, pp. 47 – 56, June 2003.

[6] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder," in *Proc. ICSE '07*, 2007, pp. 106–115.

[7] A. K. McCallum, "MALLET: A Machine Learning for Language Toolkit," http://mallet.cs.umass.edu, 2002.

[8] X.-H. Phan and C.-T. Nguyen, "GibbsLDA++, A C/C++ Implementation of Latent Dirichlet Allocation (LDA) using Gibbs Sampling for Parameter Estimation and Inference," http://gibbslda.sourceforge.net.