Interactive Tag Cloud Visualization of Software Version Control Repositories

Gillian J. Greene and Bernd Fischer
CSIR, Meraka, Centre for Artificial Intelligence Research
Computer Science Division
Stellenbosch University, South Africa
ggreene@cs.sun.ac.za, bfischer@cs.sun.ac.za

***Abstract*—Version control repositories contain a wealth of implicit information that can be used to answer many questions about a project's development process. However, this information is not directly accessible in the version control archives and must be extracted and visualized. This paper describes ConceptCloud, a flexible, interactive browser for SVN and Git repositories. The main novelty of our approach is the combination of an intuitive tag cloud visualization with an underlying concept lattice that provides a formal structure for navigation. ConceptCloud supports concurrent navigation in multiple linked but individually customizable tag clouds, which allows for multi-faceted repository browsing and for the construction of unique visualizations. We describe the mathematical foundations and implementation of our approach, and use ConceptCloud to quickly gain insight into the team structure and development process of two projects.**

I. Introduction

Version control repositories contain a wealth of implicit information that can be used to answer many questions about a project's development process, such as "Who worked on these files?", "Which developers collaborate?", "What are the co-changed methods?", or "What has happened in this project while I was away?". Answering such questions is a daily task for software developers [1] but version control systems are not set up to make the necessary information easily accessible.

There are already many repository mining tools, such as Codebook [2], Hipikat [3], or the Information Fragments prototype [4], that use processed software repository data to show specific aspects of a project. However, when users do not yet know what information they are looking for or have no previous knowledge of a project, the task becomes one of exploratory search [5] rather than mining. Therefore, we propose an interactive tag cloud visualization of software repositories in order to let users explore the information implicitly contained in these repositories. Tag clouds support browsing or exploratory search tasks and have been found to be effective when the information discovery task is wide [6]. Our interactive tag clouds familiarize users with all of the repository's data, allowing them to construct more refined views as they explore and learn more about the repository.

Tag clouds are a simple visualization method for textual data where the importance of each tag (typically its frequency) is reflected in its size. We generate tags directly from the data that we extract from software repositories, instead of relying on user-generated labels for particular content, as commonly used in Web 2.0 applications. The data available in a version control archive is often large (500000+ revisions for Linux, see https://github.com/torvalds/linux) and so we allow the user to make incremental refinements in the tag cloud visualization in order to generate smaller, more detailed visualizations.

Navigation using tag clouds has previously been explored using a Bayesian approach [7]; however, navigation in our browser is supported by a novel combination of tag clouds and formal concept lattices [8], [9], [10]. Concept lattices have been shown to be useful for browsing data [11], [12], [13] but large lattices do not provide a suitable data visualization because the relationships between the concepts are difficult to identify in a large Hasse diagram.

Our navigation algorithm maintains a *focus concept* in the underlying lattice. We derive the tag cloud visualization from the current focus concept and update it after each navigation step. Navigation is driven by the user's selection (or de-selection) of tags in the tag cloud (as shown in Fig. 1). By using different objects in the formal contexts (see Section II-B) that are used to construct formal concept lattices, we are able to generate tag clouds that provide different perspectives on the same underlying data in the same familiar visualization. Existing repository visualizations implicitly hard-code the use of revisions as objects. In contrast, our foundation in formal concept analysis allows us to change the objects easily to get different insights on the repository and to generate visualizations that are distinctly not oriented towards time, distinguishing our approach from previous work.

In this paper we develop an intuitive tag cloud interface for the information represented in formal contexts (see Section IV), and a refinement-based navigation algorithm based on concept lattices (see Section II-D) that enables interactive repository browsing through this interface. We have implemented our approach in the ConceptCloud browser (available at www.conceptcloud.org) and used this to perform two case studies in Section V.

II. Navigation Framework

A. Formal Concept Analysis

Formal Concept Analysis (FCA) [8], [9], [10] uses lattice-theoretic methods to investigate abstract relations between objects and their attributes. Such *contexts* can be imagined

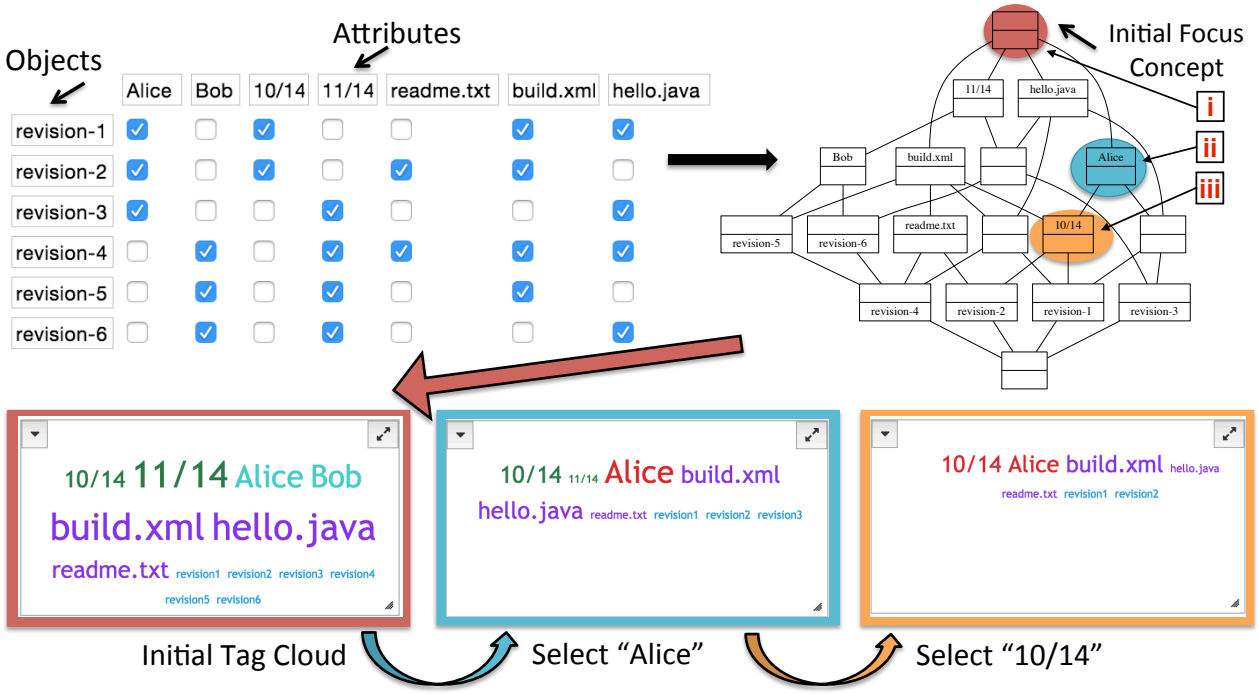


Fig. 1: Navigating Concept Lattices with Tag Clouds: tag clouds correspond to the matching colored concepts in the lattice.

as cross tables where the rows are objects and the columns are attributes (cf. Fig. 1).

Definition 1 *A formal* context *is a triple* $(\mathcal{O}, \mathcal{A}, \mathcal{I})$ *where* $\mathcal{O}$ *and* $\mathcal{A}$ *are sets of objects and attributes, respectively, and* $\mathcal{I} \subseteq \mathcal{O} \times \mathcal{A}$ *is an arbitrary incidence relation.*

Definition 2 *Let* $(\mathcal{O}, \mathcal{A}, \mathcal{I})$ *be a context,* $O \subseteq \mathcal{O}$*, and* $A \subseteq \mathcal{A}$*. The* common attributes *of* O *are defined by* $\alpha(O) = \{a \in \mathcal{A} \mid \forall o \in O : (o, a) \in \mathcal{I}\}$*, the* common objects *of* A *by* $\omega(A) = \{o \in \mathcal{O} \mid \forall a \in A : (o, a) \in \mathcal{I}\}$.

For example, the common attributes of the objects *revision-1* and *revision-2* in Fig. 1 are *Alice*, *10/14* and *build.xml*.

Concepts are pairs of objects and attributes which are synonymous. They are maximal rectangles (modulo permutation of rows and columns) in the context table. For example, ({*revision1*, *revision2*}, {*Alice*, *10/14*, *build.xml*}) in Fig. 1(ii) is a concept, since adding another revision object loses common attributes, while adding another attribute loses common objects.

Definition 3 *Let* $\mathcal{C}$ *be a context.* $c = (O, A)$ *is called a* concept *of* $\mathcal{C}$ *iff* $\alpha(O) = A$ *and* $\omega(A) = O$*.* $\pi_O(c) = O$ *and* $\pi_A(c) = A$ *are called* c*'s* extent *and* intent*, respectively. The set of all concepts of* $\mathcal{C}$ *is denoted by* $B(\mathcal{C})$.

Concepts are partially ordered by inclusion of extents such that a concept's extent includes the extent of all of its subconcepts; the intent-part follows by duality.

Definition 4 *Let* $\mathcal{C}$ *be a context,* $c_1 = (O_1, A_1), c_2 = (O_2, A_2) \in B(\mathcal{C})$*.* c_1 *and* c_2 *are ordered by the* subconcept relation*,* $c_1 \leq c_2$*, iff* $O_1 \subseteq O_2$*. The structure of* $B(\mathcal{C})$ *and* $\leq$ *is denoted by* $\mathcal{B}(\mathcal{C})$.

The basic theorem of FCA states that the structure induced by the concepts of a formal context and their ordering is always a complete lattice. Such *concept lattices* have strong mathematical properties and reveal hidden structural and hierarchical properties of the original relation. They can be computed automatically from any given relation between objects and attributes. The greatest lower bound or *meet* and least upper bound or *join* can also be expressed by the common attributes and objects.

Theorem 5 ([8]) *Let* $\mathcal{C}$ *be a context. Then* $\mathcal{B}(\mathcal{C})$ *is a complete lattice, the* concept lattice *of* $\mathcal{C}$*. Its meet and join operation for any set* $I \subset B(\mathcal{C})$ *of concepts are given by*

$$\bigwedge_{i \in I} (O_i, A_i) = \left(\bigcap_{i \in I} O_i,\ \alpha(\omega(\bigcup_{i \in I} A_i)) \right)$$

$$\bigvee_{i \in I} (O_i, A_i) = \left(\omega(\alpha(\bigcup_{i \in I} O_i)),\ \bigcap_{i \in I} A_i \right)$$

Each attribute and object has a uniquely determined defining concept in the lattice. For example, the defining concept for *Alice* is indicated in blue in the concept lattice in Fig. 1(ii). The defining concepts can be calculated directly from the attribute or object, respectively, and need not be searched in the lattice.

Definition 6 *Let* $\mathcal{B}(\mathcal{O}, \mathcal{A}, \mathcal{I})$ *be a concept lattice. The* defining concept *of an attribute* $a \in \mathcal{A}$ *(object* $o \in \mathcal{O}$*) is the greatest (smallest) concept* c *such that* $a \in \pi_A(c)$ *(*$o \in \pi_O(c)$*) holds.*

It is denoted by $\mu(a)$ ($\sigma(o)$). We use the $\delta(x)$ to denote $\mu(x)$ if x is an attribute and $\sigma(x)$ otherwise.

Efficient algorithms exist for the computation of the concept lattices and the meet and join of concepts in the lattice.

B. Contexts from Repositories

In order to construct a concept lattice from repository data we need a context table. The first step in the construction of such a context table is to determine which field will be taken as the object and which fields are suitable as attributes. We use three object types, namely revisions, files and revision-file pairs (changes) in order to construct different types of contexts, which enables us to create different tag cloud visualizations for the same repository, providing new insights on the data.

1) Revision-Based Contexts: In a revision-based context we interpret the *revisions*, represented by their *revision number*, as objects and the commit meta-data (e.g., author or words from the log message) as attributes; each revision is associated with its own meta-data as attribute. This context type represents the canonical view of repositories. Its concepts are sets of revisions and their common attributes (e.g., all revisions that include a common set of files). It is useful to get a historical overview of a project.

2) File-Based Contexts: In a file-based context we interpret the *files* as objects but derive the attributes from the revisions' pre-processed meta-data; more precisely, each file receives all attributes from all revisions that involve the file. Concepts from such contexts are sets of files with common attributes (e.g., the set of all files on which a group of developers have all worked); in particular, each commit induces a concept: since a developer can only commit one set of files at any given time, the set of committed files is maximal with respect to all attributes derived from the commit meta-data.

Note that revision- and file-based contexts give complementary views on the repository. For example, the author tags from a revision-based context are scaled according to the number of revisions that the author has committed over the project lifetime; during browsing only one author tag can be selected at a time since each revision has only one author. In a file-based context, the author tags are scaled according to how many files a particular author has changed. Selecting an author tag will reveal all *collaborators*, i.e., all other authors who have also changed the same files. Selecting two author tags will then reveal the extent of their collaboration, i.e., all files they have both worked on.

3) Change-Based Contexts: In a change-based context we use *pairs of files and revisions* as objects, so that for example (*hello.java*, *revision-1*) and (*hello.java* , *revision-3*) become separate objects in the context. This allows us to use the content of the files as additional attributes, which we cannot do with revision- or file-based contexts. In our implementation we focus on the changes (rather than the entire contents), and use a lightweight fact extractor [14] to get the signatures of the changed methods from each file. We could therefore have, for example the attributes *public int equals()*, *public static void main()*, and *Alice* associated with the object (*hello.java*, *revision-1*) to represent the fact that *revision-1* by *Alice* changes the methods *equals()* and *main()*. Selecting a method tag *m* then produces a tag cloud which contains all other methods that have been *co-changed* with *m*, scaled according to how often they have been changed together (cf. Fig. 3a).

C. Tag Clouds from Concepts

We visualize repository data with a tag cloud that we construct from the focus concept in the lattice. Since a concept comprises a set of objects and a set of attributes, it is tempting to use the attributes (i.e., the intent) as the tag cloud. However, this produces degraded clouds because (i) the intent only contains the attributes common to all objects, and (ii) each attribute only occurs once so that all tags would have the same size. Instead, we use the intents of the extents; more precisely, we collect all attributes of the defining concept of each object in the extent of the focus concept; we also add the objects themselves, to allow their direct selection in the tag cloud.

Definition 7 *The* tag cloud *from a concept $c = (O, A) \in B(\mathcal{C})$ is defined as $\tau(c) = O \uplus \biguplus_{o \in O} \pi_A \sigma(o)$.*

Here $\uplus$ denotes multiset union. By construction, the objects in the tag cloud induce subconcepts of the concept from which the tag cloud was derived; moreover, all tags have a non-bottom meet with that concept.

D. Navigating Concept Lattices with Tag Clouds

The browser maintains a *focus concept*, from which it renders the tag cloud as described above; when the user selects (or deselects) a tag, the browser updates the focus and re-renders the tag cloud. The focus, or more precisely, its extent contains the subset of objects in the repository that share all currently selected tags. The initial focus (corresponding to an empty selection set) is therefore the lattice's top element, whose extent contains the entire repository (c.f. Fig 1(i)).

Navigation is refinement-based: when the user selects another tag, the browser updates the focus by computing the meet of that tag's defining concept and the old focus, rather than recomputing it from the full selection set.

Intuitively, deselection should be the inverse of selection: deselecting the last selected tag should move the focus back to its previous position. It can however not be implemented by the join operation, because this can lead to over-generalizations and thus to counterintuitive results. For deselection, we must therefore recompute the focus as the meet of the defining concepts of the remaining selected tags.

E. Relation to Information Retrieval

Our lattice-based browsing approach is related to classical information retrieval (IR) [15], [16]. The context table can be seen as a boolean version of the document-term matrix, while the concept lattice can be seen as representation of the usual indexes: for each document the attributes of the object is the set of terms that occur in the document, and for each term the set of objects in its introducing concept is its inverted index

entry. If we see the selected tags as a conjunctive query, then the focus' extent is the query's result.

The tag cloud can also be seen as the aggregation of the boolean term frequencies for each document in the query result, scaled according to the size of the document collection. The concept lattice provides us with an efficient way to compute this tag cloud; a computation from only the inverted index would be impractically inefficient: we would first need to retrieve all documents indexed by the selected tags, then iterate over the entire vocabulary and compute the size of the intersection of each term's inverted index with the query's result. Hence, any efficient IR-based implementation must use the same information in essentially the same way as our lattice-based implementation. However, we can exploit the lattice structure, e.g., to update the focus incrementally, or to show which other tags are implied by the current selection set.

III. ConceptCloud Tool

We have implemented our approach in the ConceptCloud browser which is a web application available at www.conceptcloud.org. It comprises three main components that extract meta-data from the revision control system (see below for more details), construct a context table in the desired format (see Section II-B), and display the tag cloud (see Section IV) of the resulting lattice. ConceptCloud automates the process of creating a tag cloud visualization from a version control archive and its user interface supports customization of the tag clouds. The browser is generic and can show tag clouds of different context types. It is also completely automatic: there are no manual pre-processing steps, and the user only needs to enter the URL of the repository. A more detailed description of the tool architecture and usage is available in [17].

ConceptCloud currently supports extraction of meta-data and construction of context tables from SVN [18] and Git [19] repositories, both locally and remotely. For Git repositories, the hashes are converted into sequential revision numbers. Both extractors support the revision-, file-, and change-based contexts, as described in Section II-B. The construction of change-based contexts requires the identification of methods changed in consecutive versions, which requires the extraction to be language-aware. Such contexts are currently limited to Java files. The generated context tables can be saved in XML format so that they can be loaded again without extraction.

For the lattice construction, we use a method based on the Colibri/Java library [20] which constructs concepts on the fly. We thus never need to compute the full lattice and are able to render an initial tag cloud relatively quickly.

When we construct the context tables we pre-process the meta-data that we extract from the revision control system, in particular the log messages, file names, and commit times from each revision in the repository. We segment each log message into individual words, remove words on a default stop list, and reduce each word to its stem, using the Apache Lucene implementation of Porter's [21] stemming algorithm. Since the stem is not necessarily a proper word we take the most frequently used word that evaluates to a given stem as representative in the cloud.

We group both file names and commit times into increasingly coarser bins. For file names, we decompose each filename into a set of all path prefixes, similar to recursively applying the Unix dirname command. For commit times, we truncate the times at different precision levels (days, months, and years). In addition, we also use aggregators to capture regularities that appear across the bins, e.g., commit patterns or similarities between identically named files such as README.txt in different directories.

Note that we do not perform more complicated pre-processing steps such as word sense disambiguation [22] or identity merging [23]. We instead prefer to leave the user in control of such decisions and plan to extend ConceptCloud's GUI to support merging and splitting of tags.

IV. Tag Cloud Visualization

We make use of a tag cloud visualization that can be customized to show different views on the repository. Multiple different visualizations for different metrics were found to confuse users [24]. We therefore propose one uniform visualization that can be used to explore various different aspects of a version control archive.

The simplest and most popular tag cloud layout [25] is as an alphabetically sorted list of tags in a roughly rectangular shape which was found by Schrammel et al. to perform better than random or semantic layouts [26]; we use this layout because it simplifies textual search within the tag cloud. We scale each tag i between the given minimum and maximum font sizes f_{min} and f_{max}, according to its weight t_i in relation to the minimum and maximum weights in the context table, t_{min} and t_{max}; hence,

$$\mathrm{size}(i) = \left\lceil \frac{(f_{max} - f_{min}) \cdot (t_i - t_{min})}{t_{max} - t_{min}} \right\rceil + f_{min} - 1$$

for $t_i > t_{min}$ and $\mathrm{size}(i) = f_{min}$ otherwise.

A variety of alternative tag layout methods have been proposed, such as tag flakes by Caro et al. [27]. Tag flakes are used in order to provide context for tags as basic tag clouds fail to show how the tags are related [27]. However, instead of using a more complex visualization that depicts the relationships between the tags, such as tag flakes or a large Hasse diagram of a concept lattice, we use incremental refinement in the tag cloud to provide context and structure to the tag clouds. By selecting a tag in the tag cloud the resulting cloud will provide background for the selected tag.

The initial tag cloud shown in ConceptCloud includes tags from all attributes and objects in the context table (using the top concept in the lattice as the focus). This allows the user to select any tag from the extracted repository information. Tags in the initial tag cloud will be at their largest size because we scale all tags according the maximum and minimum tags in this cloud. Making selections in the initial tag cloud will result in clouds with smaller tags (cf. Fig. 1), indicating that

Fig. 2: Multiple linked tag clouds of the JUnit Repository in ConceptCloud

the cloud is only showing attribute tags from a subset of the total objects in the context table.

Since the initial tag cloud can be very large we provide functionality to limit the tag cloud to one particular category (e.g. commit authors), or to remove unwanted categories from the cloud. The cloud can also be adjusted to show only a certain number of tags or to show tags that occur more than a given number of times. Since all the tags are textual, users are also able to search in the tag cloud to find a tag if they do initially know which tag they want to select (such as their own commit name).

Customized visualizations can be created from the initial tag cloud by selecting relevant tags and by moving categories of tags into separate viewers. For example, Figure 2 shows a view of the year, filename and author clouds for the JUnit repository where the filename tag "AllTests.java" has been selected. The visualization quickly shows in which years this file has been changed, who has changed this file and what other files are often changed in the same commit as this one.

Viewers can also be opened with a "sticky" tag that always remains selected and cannot be deselected. This enables us to open multiple parallel viewers with different tag selections in the same category (such as months, cf. Fig. 5) which update simultaneously when another tag is selected in any viewer.

A tag is *implied* if it has not been selected explicitly, but corresponds to an attribute in the focus' intent. Implied tags thus reveal the repository's internal structure, similar to the way association rules reveal the implicit structure of shopping baskets [28] but without any additional cost.

In addition to the interface customizations that can be performed on the tag cloud there are also two customizations that can be performed during construction, namely *personalization* and *filtering*. A combination of these two customizations allows us to produce a "vacation cloud" as described in Section IV-C below.

A. Personalization in Tag Clouds

We can personalize a tag cloud for a particular developer by identifying all tags that apply to that developer (e.g., files they have changed) in our pre-processing step. We then assign these tags to different categories than the tags from the remaining commits, and render them in a different color. In the personalized tag cloud, the files that have been changed by that particular developer will thus be identifiable in views even when the tag for that developer has not been selected.

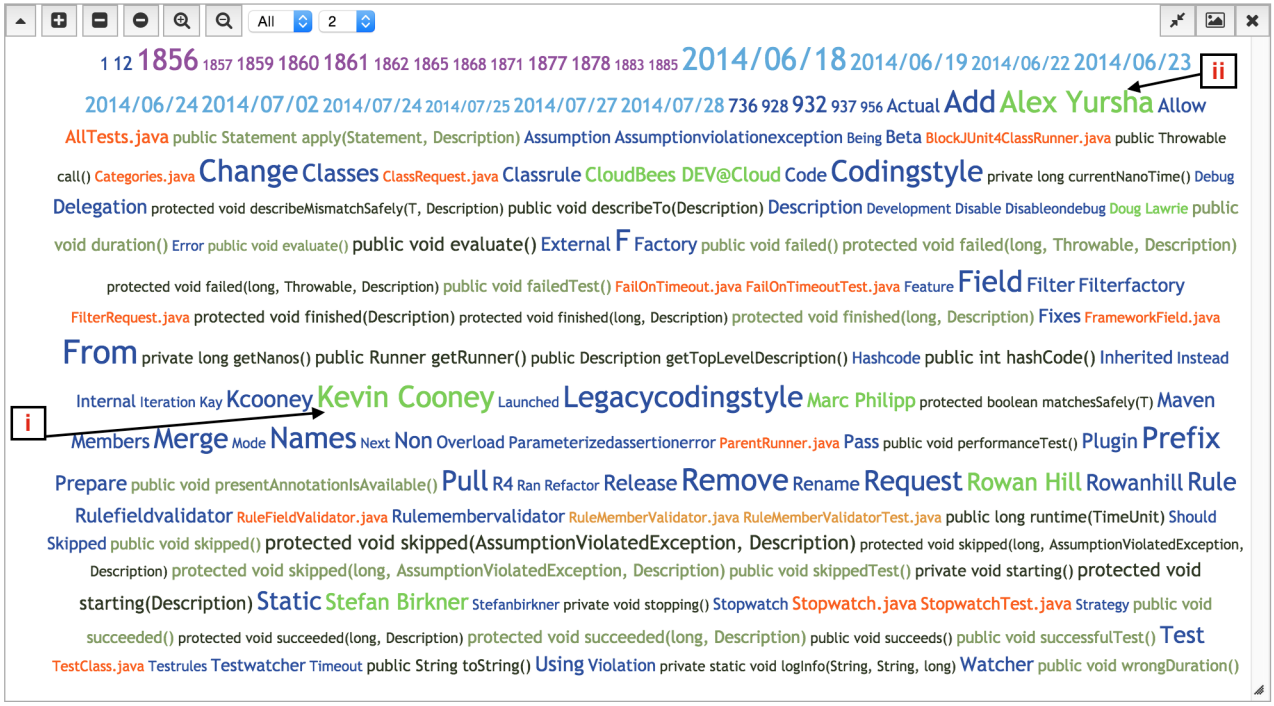
B. Filtering Tag Clouds

If we want to analyze only a particular section of a repository (e.g., only the portion since we started working on the project) we can simply restrict the revision range from which the context table is constructed. Our pre-processing offers different ways of specifying the ranges of interest.

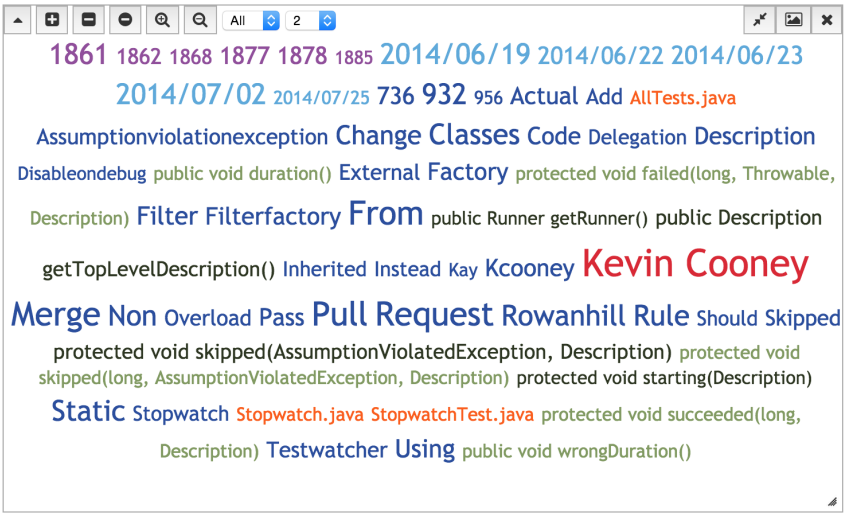
C. Customized Visualizations

The combination of filtering and personalization steps allows ConceptCloud to answer the question "What happened in my project while I was away?" with a *vacation cloud* as for example shown in Fig. 3a. This is constructed from a change-based context where file and method tags have been personalized to the developer (here David Saff) and revisions have been filtered by the date of his last commit.

The initial tag cloud shows in which revision most files were changed (1856), when most changes happened (2014/06/18), or which developers have made most changes (Alex Yursha and Kevin Cooney, cf. Fig. 3a (i) and (ii)). Tag colors indicate the corresponding categories and selected tags are shown in red. The words from the commit messages indicate that most changes were either pull requests or stylistic in nature, as indicated by prominent tags such as "Change", "Codingstyle", "Legacycodingstyle", or "Remove". However, the overall view of the changes in Fig. 3a does not provide us with many insights into the data and we refine the view by selecting tags



(a) JUnit: vacation cloud for David Saff.



(b) Changes by Alex Yursha (left) and Kevin Cooney (right).

Fig. 3: JUnit: vacation cloud for David Saff

in order to discover more. Selecting a developer gives a more detailed view of their changes and selecting one of the most active developers, Yursha, reveals that he has only committed one revision that contains stylistic changes. Alternatively selecting Cooney reveals that he has merged in several pull requests (cf. Fig. 3b) which contain changes to files that Saff has previously worked on (such as "AllTests.java"). Selecting further tags (e.g., "From" and "Rowanhill") brings out further details (e.g., about the pull requests from Hill). The cloud also shows how often files and methods have been changed; it uses different colors to distinguish changes in files also changed by Saff from those in other files. We can therefore see that the method "skipped" was a development hotspot during Saff's absence; we can further see that variants with different signatures were added (shown in light grey), on top of the changes to the variants that Saff has worked on (shown in dark grey).

V. Illustrative Case Studies

We perform case studies to show how ConceptCloud's visualizations can be used to gain insight into the development process of one commercial and one open-source project.

A. Ruby on Rails Web Application

The second author analyzed the Git repository of a small commercial development project which implements in Ruby on Rails a crowd-funding website that handles donations for solar lights. The repository contains 490 revisions in 219 files with approximately 8900 lines of code that were developed over a period of two months by seven developers.

The goal of this case study was to see whether the Concept-Cloud browser can be used to quickly gain an understanding of an unknown project's organizational structure. We therefore created a revision-based context, browsed the tag cloud for approximately two hours and then checked our observations with the project manager who confirmed them.

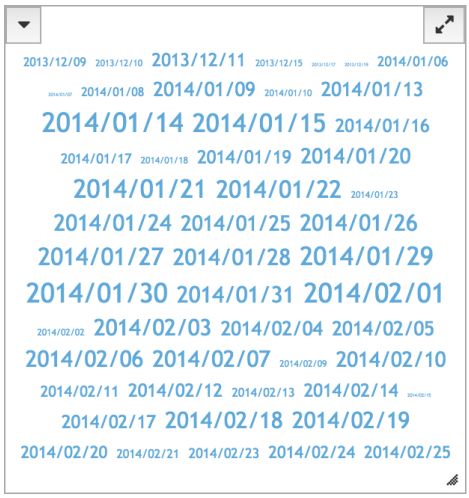
Project Activity: The commit time view (cf. Fig. 4a) shows that the project had two major activity spikes (2014/1/14-15 and 2014/1/29-30); these coincide with project demonstrations but the log messages do not reveal this. The second spike is trailed by a number of commits on Saturday 2014/2/1, by developer PP who works mostly on Saturdays (cf. Fig. 4f).

Developer Activity: The author view (cf. Fig. 4b) shows that the project involves seven developers, with three developers (CK, DR, PP) evenly sharing the main work load (80% of the revisions) on the project. Selecting the respective names of two developers (AE, GM) reveals (in the commit time view) that these were only active towards the end of the project.

Developer Expertise: Selecting the keywords "merged", "pull" and "request" in the main cloud shows (in the author view, cf. Fig. 4c) that most Git pull requests were merged by DR, followed by CK. This indicates that these two developers are the main architects of the system. Selecting DR and opening a directory view (cf. Fig. 4e) shows that DR is mostly merging requests concerning files in the app/controllers, app/models, and app/views directories and is thus responsible for the system's functionality, while CK works mostly in the app/views and app/assets/stylesheets directory and is thus responsible for the system's appearance. However, deselecting the keywords and just selecting the app/assets/stylesheets directory indicates that PP is actually implementing most of the visual functionality.

Selecting individual keywords (e.g., "video" or "facebook") shows that the related parts of the system functionality were implemented by one of the lead developers, often with the support of a second developer to integrate the functionality into the web pages (e.g., "video" by PP and GM, cf. Fig. 4d).

Conclusions: We were able to extract the team structure and developer responsibilities from the repository; this information is helpful for new team members. The category-specific tag clouds were instrumental in gaining insight about certain aspects of the project (e.g., developers) and finding this information much more quickly. The multi-faceted visualization allowed us to analyze and explore different aspects of the information concurrently, which made observations more obvious. The full tag cloud containing information from all categories gives us a complementary unified view from which we can re-direct our exploration after a navigation step. Overall, our insights come from the incremental nature of the



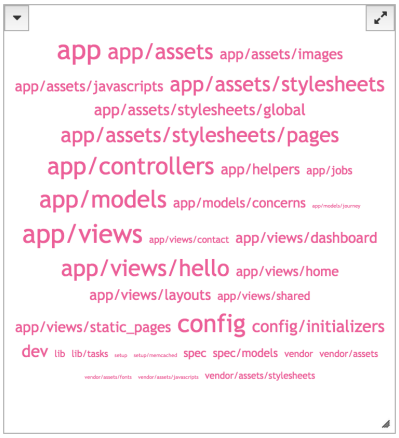
(a)

(b)

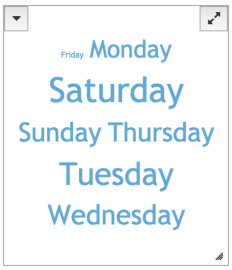
(c)

(a) Cloud from all commit dates. (b) Cloud of all developers. (c) Developers of "merged pull request"

(d) Cloud after selection of tag "Video".



(e)



(f)

(e) Directory cloud of DR. (f) Weekdays of developer PP.

Fig. 4: Ruby on Rails website

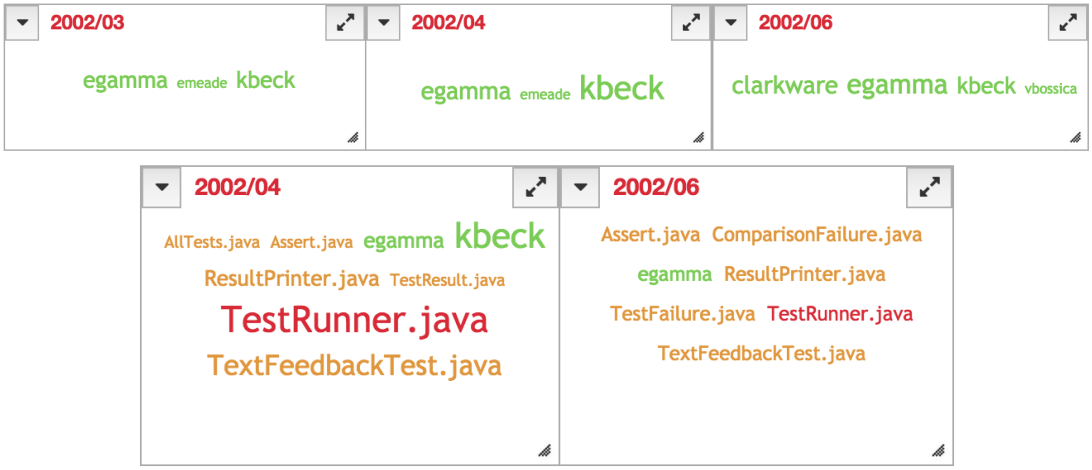


Fig. 5: JUnit: author clouds (top); changes to TestRunner.java (bottom).

exploration: each step reveals a new view into the repository and each view in turn suggests the next steps, through its most prominent tags.

B. JUnit

JUnit is an open-source testing framework for Java which has been used in other case studies [29], [30]. We are able to repeat Weissgerber's case study, which investigates developer roles up until 2006, here as the date ranges used are provided in [29]. We created the revision-based context for the JUnit project from its first revision in 03/12/2000 up until 26/02/2014 (1772 revisions).

Overview: In order to get an initial view of the project we open a commit time view and restrict it to years. This shows that project activity increases dramatically from the first full year in 2001 until 2007 and remains relatively steady thereafter. Selecting the year tag 2000 in the full cloud shows us that developer EG (we follow [29] and only use developer initials) started the project in December 2000. In an author cloud for the first full year of development (2001) we see that developers KB and EM join the project in 2001 but EG remains the most prolific author in that year (cf. [29]).

Authors by Month: [29] looks specifically at the file changes made in the months March to June 2002. To repeat this we open viewers with "sticky tags" for March, April, and June 2002 (there was no commit in May 2002) and limit these to show only author and file tags (cf. Fig. 5, top). Selecting an author tag shows us which files the author has worked on in each month. Fig. 5 shows KB's contribution reduces in the given period. The cloud for June 2002 shows the addition of developers VB and CW to the project.

Selecting the file "TestRunner.java", shows that both EG and KB have changed this file in April 2002 and just EG in June 2002 (cf. Fig 5 (bottom)). We also see that there are a group of files which have been changed at the same time.

Conclusions: ConceptCloud allowed us to draw the same conclusions as the dedicated tool presented in [29]. However, in contrast to [29], it does not simply produce a static picture but allows the user to refine the analysis, and access the other information (e.g., log messages) that remains available.

TABLE I: Metrics for revision-based contexts

Project	Type	$\|O\|$	$\|A\|$	$\|I\|$	Indexing time(s)	Drawing time(s)
Case Study 1	Git	492	1,516	14,561	3.4	0.6
Subversive	SVN	1,511	8,222	88,090	55.5	1.8
JUnit	Git	1,905	5,959	66,242	8	1.9
AngularJS	Git	5,547	9,055	133,436	116.2	2.8
Spring	Git	9,017	40,332	540,813	43.4	14.8
Valgrind	SVN	10,989	29,009	348,136	176.6	40
Django	Git	18,471	38,821	583,701	58.4	11
Moodle	Git	69,550	154,834	2,222,486	333.2	45.7
DPorts	Git	155,627	196,850	2,917,269	2,049.9	892.8

VI. Performance Evaluation

We used ConceptCloud on a medium-sized server (64GB RAM, 2 Xeon 8-core 2.0Ghz CPUs) to analyze several Git and SVN repositories in order to evaluate its performance. We created revision-based contexts; we used local clones of the Git repositories but accessed the SVN repositories remotely. Table I summarizes the characteristics of and runtimes for these repositories. It shows the number of revisions $|O|$, the number of attributes $|A|$, and the size of the incidence relation (i.e., the number of object/attribute pairs) $|I|$, as well as the times taken to create the context table (i.e., indexing) and to draw the full tag cloud for the repository.

We see that the indexing times are only a few seconds for smaller repositories, and a few minutes for medium-sized ones; even the largest repository with 155627 revisions requires only 34 minutes. Note that these times are not directly related to either the size or the density (i.e., $|I|$) of the context tables but are to a large extent determined by the (lexical) pre-processing.

The drawing times are given for the full tag cloud for the repository, which contains $|O| + |A|$ tags. The table thus gives an indication of the initial load time in the worst case; in practice, we can limit the number of tags shown to substantially improve this. Tag clouds become smaller with subsequent navigation steps and will therefore be drawn substantially faster; we also use caching to improve performance further. Overall, navigation is instantaneous for small and medium repositories, with some degradation on the initial clouds for very large repositories.

Note also that drawing the initial cloud also requires us to compute the defining concepts of all objects; however, since we use an incremental lattice construction approach and therefore never actually compute the full lattice, we do not experience the large runtimes normally associated with FCA.

VII. Related Work

A. Visualizing Software Repositories

Zaidman et al. [31] develop a change-history view and a growth-history view to study the co-evolution of production and test code. The change history view is a plot of the changed files over the revisions of a project's repository distinguishing between production and text code.

Girba et al. use an "Ownership Map" visualization [32] in order identify developer interaction in a project and development patterns using the CVS log of a project. Girba et al. also identify several behavioral patterns of developers, such as a teamwork, takeover and cleaning and show how these can be identified in their ownership map visualization.

Alonso et al [33] also use a tag cloud visualization to display information from version control (CVS) repositories. Their "expertise cloud visualization" creates a tag cloud of commiters that are identified using rule-based classification on CVS log information. Users are then able to select the names in this cloud to display a cloud of the developer's expertise. The expertise cloud visualization [33] differs from that of ConceptCloud as the different types of information can only be displayed in separate clouds, meaning that the combinations of tags a user can select are limited, as opposed to our underlying concept lattice which only limits the available tag selections to tags that will not cause an empty tag cloud to be displayed.

Codebook [2] is a social network inspired toolset to analyze information implicitly contained in software repositories. Its central data structure is a graph, where the nodes represent the different artifact and actors (e.g., change set, developer), and the edges represent the different relations between these (e.g., contains, committer). This graph is built from different sources including revision control systems, bulletin boards, mails, and directory information. Results are displayed in a web interface that provides a simple list including images of people associated with the artifacts.

Hipikat [3] also monitors multiple information sources (Bugzilla, CVS, email, newsgroups) and builds a uniform artifact database. It has a number of heuristics (based on text similarity and activity times) to create links between the artifacts, and provides lists of related artifacts on request. Hipikat queries are made using the Eclipse IDE and the results are also displayed in a Hipikat list view Eclipse plugin.

Information Fragments [4] provide answers to developer's questions by combining subsets of relevant project information. Information Fragments are comprised of nodes of different types, such as a team member or work item. Node types are similar to tag categories in ConceptCloud. The presentation of results in [4] uses an Eclipse plugin and supports a counting feature to get an overview of the number of occurrences of nodes, to get for example, the number of items a developer has been working on. Our tag cloud automatically gives the user an overview of the number of occurrences of each item as the tags are sized according to occurrence frequency.

B. Tag Cloud Visualizations of Software

Guido [34] includes a tag cloud to visualize names of types, variables, parameters and methods in source code. Selecting nodes in the graph visualization that Guido also provides will highlight the corresponding tags in the tag cloud and selecting a name in the tag cloud will highlight corresponding source code elements in the graph view. The visualizations are linked in Guido similarly to the multiple tag clouds that update simultaneously in ConceptCloud. Anslow et al. use a tag cloud to visualize the structure of Java class names in [35]. Emerson et al. use tag clouds to visualize Java methods and explore several different tag cloud layouts using the TAGGLE tool [36]. TAGGLE extends basic tag cloud views and allows highlighters to be associated with tags so that if a tag is selected related tags in the cloud will be highlighted. Tag clouds in TAGGLE are customizable, as they are in ConceptCloud, with TAGGLE additionally allowing tag layouts to be changed.

C. Tag Clouds and Navigation

Mesnage and Carmen use a Bayesian approach for navigation in tag clouds that allows tags related to one or more selected tags to be shown in the cloud, where previously clouds could only be created for one selected tag [7]. Gwizdka and Bakelaar look at displaying a tag cloud history, which allows users to keep track of their previous navigation steps, when clouds are used for pivot navigation [37]. This approach is not directly applicable to our tag clouds since we use refinement navigation where multiple tags can be selected. Hernandez et al. use multiple linked tag clouds to browse semi-structured clinical trial data [38]. These tag clouds are generated from the results of an initial search query and each represent one facet (e.g. medical condition), of the data. A multi-faceted view can also be created in ConceptCloud by moving tag categories into separate tag clouds.

D. Software and Formal Concept Analysis

Poshyvanyk and Marcus [39] use a combination of latent semantic indexing and concept lattices to find methods that are relevant to a bug report. Girba et al. [40] use concept analysis to detect co-change patterns in revision control systems. Objects are packages, classes, or methods, while properties are

the validity of expressions over certain metrics of the objects (e.g., number of classes, methods, or statements); the specific expression is determined by which co-change pattern is to be detected. Similar ideas could be integrated into our approach.

There have also been direct applications of formal concept analysis to source code analysis and re-engineering [41], [42] but these only consider an individual program, not a repository.

VIII. Conclusions and Future Work

In this paper we have developed an interactive browser for revision control repositories which uses a novel combination of formal concept lattices and tag clouds to make the information implicitly contained in these repositories accessible to the user.

Our browser can thus be used to answer many difficult questions such as "What has happened in this project while I was away?", "Which developers collaborate?", or "What are the co-changed methods?". We have used the ConceptCloud browser to repeat a previous case study [29] and to make observations about the internal structure of a small commercial development project.

We see several avenues for future work. We plan to add functionality for indexing bug databases and combining these contexts with contexts from revision control repositories, to provide a more complete overview of a project, as done by Codebook or Hipikat. We are currently working on a scripting language so that specific layouts of viewers to be scripted; this will simplify building custom visualizations (such as the authors by year view) on top of the generic user interface. We also plan to use ConceptCloud to visualize other aspects of software such as class structure and method calls.

Acknowledgements

This research is funded in part by a STIAS Doctoral Scholarship, NRF Grant 93582 and the MIH Media Lab.

References

[1] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *FSE*, 2006, pp. 23–34.

[2] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: discovering and exploiting relationships in software repositories," in *ICSE (1)*, 2010, pp. 125–134.

[3] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE TSE*, 31(6), pp. 446–465, 2005.

[4] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," in *ICSE (1)*. ACM, 2010, pp. 175–184.

[5] R. W. White, B. Kules, S. M. Drucker, and m. schraefel, "Introduction," *Commun. ACM*, 49(4), pp. 36–39, Apr. 2006.

[6] J. Sinclair and M. Cardew-Hall, "The folksonomy tag cloud: when is it useful?" *Journal of Information Science*, 34(1), pp. 15–29, 2008.

[7] C. S. Mesnage and M. J. Carman, "Tag navigation," in *SoSEA 2009*, ACM, 2009, pp. 29–32. .

[8] R. Wille, "Restructuring lattice theory: an approach based on hierarchies of concepts," in *Ordered sets*, I. Rival, Ed. Reidel, 1982, pp. 445–470.

[9] B. Ganter and R. Wille, *Formal concept analysis - mathematical foundations*. Springer, 1999.

[10] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order (2. ed.)*. Cambridge University Press, 2002.

[11] B. Fischer, "Specification-based browsing of software component libraries," *ASE*, 7(2), pp. 179–200, 2000.

[12] C. Lindig, "Concept-based component retrieval," in *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, 1995, pp. 21–25.

[13] C. Carpineto and G. Romano, "A lattice conceptual clustering system and its application to browsing retrieval," *Machine learning*, 24(2), pp. 95–122, 1996.

[14] G. C. Murphy and D. Notkin, "Lightweight lexical source model extraction," *ACM TOSEM.*, 5(3), pp. 262–292, 1996.

[15] C. Van Rijsbergen, *Information retrieval*, Butterworths, 1979.

[16] C. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, Cambridge University Press, 2008.

[17] G. J. Greene and B. Fischer, "Conceptcloud: A tagcloud browser for software archives," in *FSE*, 2014, pp. 759–762.

[18] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick, *Version Control with Subversion - The Standard in Open Source Version Control*. O'Reilly, 2008.

[19] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.", 2012.

[20] D. N. Götzmann, "Colibri/java," http://code.google.com/p/colibri-java/, 2007.

[21] M. F. Porter, "An algorithm for suffix stripping," *Program: electronic library and information systems*, 14(3), pp. 130–137, 1980.

[22] R. Navigli, "Word sense disambiguation: A survey," *ACM CSUR*, 41(2), 2009.

[23] G. Robles and J. M. Gonzalez-Barahona, "Developer identification methods for integrated data from various sources," *SIGSOFT SEN*, 30(4), pp. 1–5, 2005.

[24] C. Anslow, S. Marshall, J. Noble, and R. Biddle, "Sourcevis: Collaborative software visualization for co-located environments," in *VISSOFT*, 2013, pp. 1–10.

[25] S. Lohmann, J. Ziegler, and L. Tetzlaff, "Comparison of tag cloud layouts: Task-related performance and visual exploration," in *INTERACT (1)*, 2009, pp. 392–404.

[26] J. Schrammel, M. Leitner, and M. Tscheligi, "Semantically structured tag clouds: An empirical evaluation of clustered presentation approaches," in *CHI*, 2009, pp. 2037–2040.

[27] "Navigating within news collections using tag-flakes," *Journal of Visual Languages and Computing*, 22(2), pp. 120 – 139, 2011.

[28] M. J. Zaki and M. Ogihara, "Theoretical foundations of association rules," in *DKMD*, 1998.

[29] P. Weissgerber, M. Pohl, and M. Burch, "Visual data mining in software archives to detect how developers work together," in *MSR*, 2007, pp. 9–.

[30] S. Thummalapenta and T. Xie, "Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web," in *ASE*, 2008, pp. 327–336.

[31] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," in *ICST*. IEEE, 2008, pp. 220–229.

[32] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, "How developers drive software evolution," in *IWPSE*, 2005, pp. 113–122.

[33] O. Alonso, P. T. Devanbu, and M. Gertz, "Expertise identification and visualization from cvs," in *MSR*, 2008, pp. 125–128.

[34] R. Cottrell, B. Goyette, R. Holmes, R. Walker, and J. Denzinger, "Compare and contrast: Visual exploration of source code examples," in *VISSOFT*, 2009, pp. 29–32.

[35] C. Anslow, J. Noble, S. Marshall, and E. Tempero, "Visualizing the word structure of java class names," in *OOPSLA Companion* , 2008, pp. 777–778.

[36] J. Emerson, N. Churcher, and C. Deaker, "From toy to tool: Extending tag clouds for software and information visualisation," in *Australian Software Engineering Conference*, 2013, pp. 155–164.

[37] J. Gwizdka and P. Bakelaar, "Tag trails: navigation with context and history," in *CHI'09*. ACM, 2009, pp. 4579–4584.

[38] M.-E. Hernandez, S. M. Falconer, M.-A. Storey, S. Carini, and I. Sim, "Synchronized tag clouds for exploring semi-structured clinical trial data," in *CASCON*,. ACM, 2008, pp. 4:42–4:56.

[39] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *ICPC*, 2007, pp. 37–48.

[40] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel, "Using concept analysis to detect co-change patterns," in *IWPSE*, pp. 83–89, 2007.

[41] G. Snelting, "Reengineering of configurations based on mathematical concept analysis," *ACM TOSEM.*, 5(2), pp. 146–189, 1996.

[42] G. Snelting and F. Tip, "Reengineering class hierarchies using concept analysis," *SIGSOFT SEN*, 23(6), 1998, pp. 99–110.