Information and Software Technology 000 (2016) 1–19

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices

Gillian J. Greene*, Marvin Esterhuizen, Bernd Fischer

CAIR, CSIR Meraka, Computer Science Division, Stellenbosch University, Stellenbosch, South Africa

ARTICLE INFO

Article history:
Received 29 January 2016
Revised 2 December 2016
Accepted 5 December 2016
Available online xxx

Keywords:
Formal concept analysis
Tag clouds
Browsing software repositories
Interactive tag cloud visualization

ABSTRACT

Context: 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 repositories and must be extracted and visualized.

Objective: the main objective of this work is to develop a flexible and generic interactive visualization engine called ConceptCloud that supports exploratory search in version control repositories.

Method: ConceptCloud is a flexible, interactive browser for SVN and Git repositories. Its main novelty 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 scriptable construction of unique visualizations.

Results: 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 three projects. We perform a user study to determine the usability of ConceptCloud. We show that untrained participants are able to answer historical questions about a software project better using ConceptCloud than using a linear list of commits.

Conclusion: ConceptCloud can be used to answer many difficult questions such as "What has happened in this project while I was away?" and "Which developers collaborate?". Tag clouds generated from our approach provide a visualization in which version control data can be aggregated and explored interactively.

1. 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]. Developers also rely on examining the history of a software project to keep up with changes, understand coding decisions and debug [2]. In co-located teams new developers rely on members of the team to help them ramp-up [3] but in large open-source projects, where this is no longer possible, the repository information becomes a valuable resource for new developers as well [4].

While it is well-known that version control repositories are a valuable source of information, repository tools are not set up to provide insights into the history of a project directly and can only be used to see information about individual commits. Manually examining the last few commits to a software project in the version control repository is feasible for regular contributors of the project who are only seeking information about the most recent changes. However, manually examining the entire commit log of a repository in order to answer more complex questions becomes infeasible. Individual commits provide information about a single change to the project but only when a large number of commits is aggregated does the information become accessible to developers seeking answers to complex questions.

We develop ConceptCloud, an interactive tag cloud visualization engine for software repositories that aggregates commit data and lets users easily construct uniform visualizations of many different

* Corresponding author.
E-mail addresses: ggreene@cs.sun.ac.za (G.J. Greene), mhesterhuizen@cs.sun.ac.za (M. Esterhuizen), bfischer@cs.sun.ac.za (B. Fischer).

http://dx.doi.org/10.1016/j.infsof.2016.12.001
0950-5849/

aspects of the project history. ConceptCloud makes use of a novel combination of tag clouds and an underlying concept lattice [5] to support *exploratory search* [6,7] tasks on software repositories. When users have no previous knowledge of a project or have not yet formulated a direct query their task becomes one of exploratory search instead of *direct search* or *retrieval* [7]. While there are already approaches supporting specific retrieval tasks and visualizing aspects of software repositories [8], support for exploratory search in software repository data remains unavailable. The goal of our work is to build a flexible and interactive visualization engine that allows users to visualize different aspects of a project interactively and therefore supports exploratory search tasks, instead of presenting the user with one static, pre-configured view.

An exploratory search approach can provide an overview of the repository data and allow the user to further investigate any aspects of the project which they might find interesting. Therefore, exploratory search approaches can support new developers on a project in understanding the project history and team structure. An exploratory approach can also be used to answer more general questions (e.g., "Which developers collaborate?") which cannot be formulated as a single search query which would be possible if the question was more focused (e.g., "Who collaborates with Alice?").

Tag clouds (or word clouds) are a simple visualization method for textual data where the frequency of each tag is reflected in its size. We use a tag cloud visualization to present aggregated software repository data, as tag clouds support exploratory search tasks and have been found to be effective when the information discovery task is wide [9]. While our tag cloud visualization may not be the optimal visualization for all aspects of the data, it is flexible enough to visualize many aspects of the software project such as developer expertise (e.g., which developers have worked on particular files or directories and would be good candidates to ask questions about this functionality), co-changed methods in a software project, project activity (e.g., in which years and months has there been a lot of development, and on which parts of the system), or developer collaboration (e.g., which developers are working together on which parts of the project) in a uniform way. Our interactive tag clouds allow developers to aggregate commits into groups and filter commits that apply to a certain topic, which has been noted by developers to be useful [2].

We generate tags directly from the data that we extract from software repositories, instead of relying on user-generated labels as tags for particular content, as often done in Web 2.0 applications (such as Flickr's early tag cloud view). The data available in a version control archive is often large (for example, more than 500,000 revisions for the Linux [10] repository) and so we allow the user to make incremental refinements (i.e., navigate) in the tag cloud in order to generate smaller, more detailed visualizations. The navigation in our tag clouds is crucial for facilitating exploratory search tasks. Navigation using tag clouds has previously been explored using a Bayesian approach [11]; however, navigation in our browser is supported by a novel combination of tag clouds and concept lattices [5,12,13].

We conjecture that a concept lattice [5] provides a high level of internal structure for the repository data and therefore allows users to explore the data through multiple navigation paths. Concept lattices have been shown to be useful for browsing data [14–16] 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. Therefore, we make use of a concept lattice to facilitate navigation in the more intuitive and scalable tag cloud visualization.

Fig. 1 shows an overview of our approach. We construct a formal context from data in a version control archive (see Section 4.1) and generate a concept lattice directly from the context. Note that we have used a small illustrative example as larger context tables and lattices (for example, one derived from the Linux repository) become incomprehensible. Our refinement-based navigation algorithm (see Section 3.4) then enables interactive repository browsing through our tag cloud interface (see Section 4.1). Our navigation algorithm maintains a *focus concept* in the underlying lattice which represents the user's current tag selection. 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. Fig. 1(i) shows the initial focus concept generating the first tag cloud, after the selection of tag "Alice" the focus moves to (ii) and the tag cloud is updated.

By using different objects in the formal contexts (see Section 3.2) that are used to construct concept lattices, we are able to generate tag clouds that provide different perspectives on the same underlying data in the same familiar visualization. Our foundation in formal concept analysis allows us to change the objects easily to get different insights on the same repository.

We have implemented our approach in the ConceptCloud browser (available at www.conceptcloud.org) which includes advanced visualizations, such as multiple interlinked tag clouds. Section 5 shows the application of ConceptCloud to three different repositories.

In this paper, we extend our previous work [17] by providing a formalization for our formal context construction from repositories (see Section 2), combining multiple archives (such as issue databases and version control repositories) in the same context in order to support data fusion (see Section 3.2.5) and developing a browser scripting language for ConceptCloud to support advanced customizations (see Section 4.2.4). We have also conducted additional evaluation in the form of a user study (see Section 7).

2. Modeling software repositories

We use a simple repository model derived from Hindle and Germán's SCQL [18] to formalize how we construct the contexts that underpin our browser: a *repository* is simply a collection of *versions* of a set of *files* that are grouped into *revisions*. Note that we follow the SVN terminology [19] here. Hindle and Germán [18] refer to versions as revisions, while revisions are called modification requests; elsewhere revisions are called transactions.

A *version* $v \in \mathcal{V}$ denotes the abstract state of a *file* $f \in \mathcal{F}$ created by an *author* $a \in \mathcal{A}$ at a *time* $t \in \mathcal{T}$. We ignore the actual file contents and only use meta-data and abstract modifications. Versions constitute a *version history* if they are ordered by a precedence relation $\prec$ that holds only between versions of the same file and is compatible with the file creation times. We say that *vevolves into* v' if $v \prec v'$ holds; two versions v_1 and v_2 are *merged* into v if $v_1 \prec v$ and $v_2 \prec v$.

Definition 1. Let $\mathcal{V} \subseteq \mathcal{F} \times \mathcal{T} \times \mathcal{A}$ be a set of *versions* over files $\mathcal{F}$ and $\prec \subseteq \mathcal{V} \times \mathcal{V}$ be an irreflexive partial order. $(\mathcal{V}, \prec)$ is called a *version history* iff $v = (f, t, a) \in \mathcal{V}$, $v' = (f', t', a') \in \mathcal{V}$, and $v \prec v'$ imply $f = f'$ and $t < t'$.

A *revision* r is a set V of file versions that are committed to the *repository* $\mathcal{R}$ at time t by an author a; on commit, some meta-data (i.e., author, time, and an additional *log message* $l \in \mathcal{L}$) is stored together with the versions. We assume that each revision $r \in \mathcal{R}$ contains only one version of a file (which need not be the most recent version), and that each revision is uniquely determined by an abstract identifier $id(r)$.

Definition 2. Let $(\mathcal{V}, \prec)$ be a version history and $\mathcal{R} \subseteq \mathbb{P}(\mathcal{V}) \times \mathcal{T} \times \mathcal{A} \times \mathcal{L}$ be a set of *revisions*. $\mathcal{R}$ is called a *repository* iff $r = (V, t_r, a_r, l) \in \mathcal{R}$ and $v = (f, t_v, a_v) \in V$ imply $t_v \leq t_r$ and $v \not\prec v'$ for all $v' \in V$.

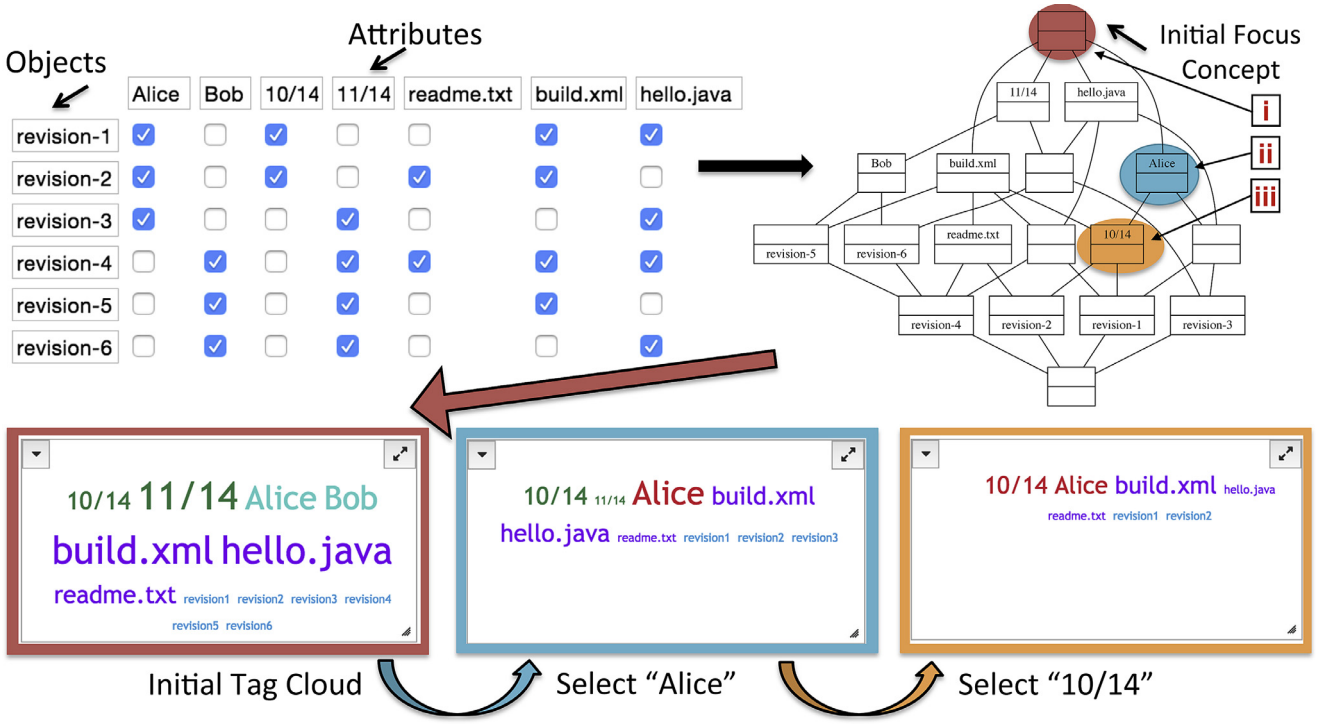


Fig. 1. Navigating concept lattices with tag clouds: tag clouds correspond to the matching colored concepts in the lattice (tag clouds from left to right correspond to concepts i, ii and iii respectively). Context table (top left) used to generate concept lattice (top right). Tag clouds are refined on each tag selection (selected tags shown in red). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

We can easily extend this basic model towards common revision control systems. For example, in CVS [20], the notions of versions and revisions are conflated; in our model we thus have for all revisions $r = (V, t, a, l) \in \mathcal{R}$ that $V = (f, t, a)$. Note that we do not model revision tagging explicitly, but assume that the tags are part of the log messages. In SVN, each revision can only contain the most recent version of a file, and only the commit author and time are recorded but not the file author or modification time. Hence, in our model we thus have for all $r = (V, t_r, a_r, l) \in \mathcal{R}$ and $v = (f, t_f, a_f) \in \mathcal{V}$ that $v \in V$ implies that $t_f = t_r$ and $a_f = a_r$. Note that we are only interested in the linear sequence of revisions and therefore do not model explicit branching and merging, but again assume that this information is encoded into the log messages, if requested. For distributed revision control systems such as Git we analyze a clone of the repository. Note that clones of the repository in different states will generate different contexts, as the contexts are generated using the commit information extracted from the repository. Therefore, if a repository is not up-to-date (i.e., has changes available to be pulled) then the generated context will differ from that of the up-to-date repository, as the list of commits differs.

3. Navigation framework

In our model, we have a set of revisions and a set of attributes for each revision; the attributes are divided into separate categories such as author, date, or file name. Our goal in browsing is to retrieve a set of revisions which share a common attribute such as the same author, and then to refine this set gradually by adding more attributes. We use *formal concept analysis* (FCA) as framework to achieve this goal.

3.1. Formal concept analysis

Formal concept analysis (FCA) [5,12,13] uses lattice-theoretic methods to investigate abstract relations between objects and their attributes. Such *contexts* can be imagined as cross tables where the rows are objects and the columns are attributes (cf. Fig. 1).

Definition 3. 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 4. 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 is a concept, since adding another revision object loses common attributes, while adding another attribute loses common objects.

Definition 5. 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 6. 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 7 (Wille [5])**.** *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 8. 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, such as Lindig's algorithm [21].

3.2. 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 in the data will be taken as the object and which fields are suitable as attributes for that object. We use three different object types, namely revisions, files, and revision-file pairs (i.e, 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 into the data. We are able to combine multiple data sources in the same context to support data fusion as object types in the context table need not be homogeneous. We use a combination of issue and version control data, in the same context, to provide a more complete overview of a project.

3.2.1. Basic preprocessing

When we construct 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 use a function $\mathbb{W} : \mathcal{L} \to \mathbb{P}(\mathcal{W})$ that segments each log message into individual words $w \in \mathcal{W}$, removes words on a default stop list, and reduces each word to its stem, using the Apache Lucene implementation of Porter's [22] 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 use a function $\mathbb{D} : \mathcal{F} \to \mathbb{P}(\mathcal{F})$ that decomposes each file name into a set of all path prefixes, similar to recursively applying the Unix `dirname` command. For commit times, we use a function $\mathbb{T} : \mathcal{T} \to \mathbb{P}(\mathcal{T})$ that truncates the times at different precision levels (days, months, and years).

In addition, we also use aggregators (such as aggregating files with the same names, even across directories) to capture regularities that appear across the bins, e.g., similarities between identically named files such as README.txt in different directories. We use d, n, and t, respectively, to denote mappings from each time to the corresponding weekday, and from each file to its base name and type, respectively.

Note that we do not perform more complicated pre-processing steps such as word sense disambiguation [23] or identity merging [24]. We instead prefer to leave the user in control of such decisions.

3.2.2. 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, for example to identify when the most changes have been made to a project, which developers have worked on particular files and which directories have been development hotspots.

Definition 9. Let $\mathcal{R}$ be a repository, and $\mathcal{A}_R = \mathcal{W} \cup \mathcal{A} \cup \mathcal{T} \cup \mathcal{F}$. $\mathcal{C}_R = (id(\mathcal{R}), \mathcal{A}_R, \mathcal{I}_R)$ is called the *revision-based context of*$\mathcal{R}$ if for all $r = (V, t, a, l) \in \mathcal{R}$, $v = (f, t', a') \in V$, and $x \in \mathcal{A}_R$, we have $(r, x) \in \mathcal{I}_R$ iff

(i) $x \in \mathbb{W}(l)$, or
(ii) $x = a$, or
(iii) $x = d(t)$ or $x \in \mathbb{T}(t)$, or
(iv) $x = n(f)$ or $x \in \mathbb{D}(f)$, or
(v) $x = t(f)$.

3.2.3. 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 the set of all attributes derived from the commit meta-data.

Definition 10. Let $\mathcal{R}$ be a repository, and $\mathcal{A}_F = \mathcal{W} \cup \mathcal{A} \cup \mathcal{T} \cup id(\mathcal{R})$. $\mathcal{C}_F = (\mathcal{F}, \mathcal{A}_F, \mathcal{I}_F)$ is called the *file-based context of*$\mathcal{R}$ if for all $r = (V, t, a, l) \in \mathcal{R}$, $v = (f, t', a') \in V$, and $x \in \mathcal{A}_F$, we have $(f, x) \in \mathcal{I}_F$ iff

(i) $x \in \mathbb{W}(l)$, or
(ii) $x = a$, or
(iii) $x = d(t)$ or $x \in \mathbb{T}(t)$, or
(iv) $x = n(f)$ or $x \in \mathbb{D}(f)\backslash\{f\}$, or
(v) $x = t(f)$, or
(vi) $x = id(r)$.

Note that revision- and file-based contexts give complementary views on the repository. For example, the author tags from a revision-based context will be 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 will be 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 any of the same files. Selecting two author tags will then reveal the extent of their collaboration, i.e., all files they have both worked on. Therefore file-based contexts can be used to visualize the collaboration in the project, showing which developers work together and on which files.

Please cite this article as: G.J. Greene et al., Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices, Information and Software Technology (2016), http://dx.doi.org/10.1016/j.infsof.2016.12.001

Fig. 2. Multiple linked tag clouds of the JUnit Repository in ConceptCloud, showing changed files (top), authors (bottom) and years (left). The tag cloud is constructed from a revision-based context.

3.2.4. 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 [25] 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. 3). Therefore change-based contexts can be used to construct visualizations that depict the co-changed methods in the project as well as showing other method information, for example, which methods are development hotspots and in which time periods.

In our model, we assume a set $\mathcal{M}$ of abstract *modifications* (in the spirit of the atomic changes of Ren et. al [26]), and use $\Delta(v', v) \subseteq \mathcal{M}$ to denote the (non-symmetric) difference between two versions $v' \prec v$ of a file.

Definition 11. Let $\mathcal{R}$ be a repository, and $\mathcal{A}_C = \mathcal{W} \cup \mathcal{A} \cup \mathcal{T} \cup \mathcal{F} \cup id(\mathcal{R}) \cup \mathcal{M}$. $\mathcal{C}_C = (\mathcal{F} \times id(\mathcal{R}), \mathcal{A}_C, \mathcal{I}_C)$ is called the *change-based context of*$\mathcal{R}$ if for all $r = (V, t, a, l) \in \mathcal{R}$, $v = (f, t', a') \in V$, $v' \in V$ with $v' \prec v$, and $x \in \mathcal{A}_C$, we have $((f, r), x) \in \mathcal{I}_C$ iff

(i) $x \in \mathbb{W}(l)$, or
(ii) $x = a$, or
(iii) $x = d(t)$ or $x \in \mathbb{T}(t)$, or
(iv) $x = n(f)$ or $x \in \mathbb{D}(f)$, or
(v) $x = id(r)$, or
(vi) $x \in \Delta(v', v)$.

3.2.5. Combined contexts: bug reports and revision control data

Software development projects often make use of dedicated tools for different tasks, such as issue databases, task trackers, and source code repositories, or use a tool that provides a combination of these such as GitHub [27]. Moreover, archive entries can be linked across the different tools, by, for example, adding an issue identifier to the log message of a revision which references that issue. Ideally, visualization tools should be able to "fuse" the information from different archives for the same project into a single combined data structure, such as Hipikat's uniform artifact database [4] or Codebook's central graph [28].

Here, we combine data from multiple archives (or different features of GitHub) into a single context using multiple object types. In particular, we combine repository data and GitHub issue data into the same context. In the combined contexts we use the revisions and bug reports as objects (since the object types in the context table need not be homogeneous) and derive the attributes from both the revisions' pre-processed meta-data and the text from the bug reports. Therefore, where bug reports and revisions share a common attribute they will be grouped together in the same concepts, indicating the relation of the bug reports to the revisions. The combined context gives a more complete overview of the project activities.

Note that the objects in a combined context are a *union* of revisions and issue IDs; this is different to the construction of the change-based contexts where the objects are *pairs* of revisions and files. The combined context's attributes are the union of the original attributes for both the revisions and the issues, and each object keeps its own attributes. We merge corresponding attribute categories from the data sources, e.g., log messages and issue descriptions. This assumes that words have the same meaning in the different archives, but in return it provides us with implicit links between bugs and revisions that both talk about a specific topic (e.g., "Linux"), because their log messages and descriptions share a common attribute. The issues and revisions are therefore connected automatically, without the need to create any links, as for example described by Silwerski et al. [29]. However, for a data source such as GitHub, which stores explicit references between commits and issues, we are able to link these in the context table by using a "surrogate key" attribute which we assign to both the revision object and the issue object in the context table. A surrogate key is therefore, an additional attribute which serves exclusively to indicate an explicit link between the revision and the issue in the concept lattice. Section 5.2 provides examples of tag clouds generated from Git repositories and issues in the GitHub issue-tracking

Please cite this article as: G.J. Greene et al., Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices, Information and Software Technology (2016), http://dx.doi.org/10.1016/j.infsof.2016.12.001

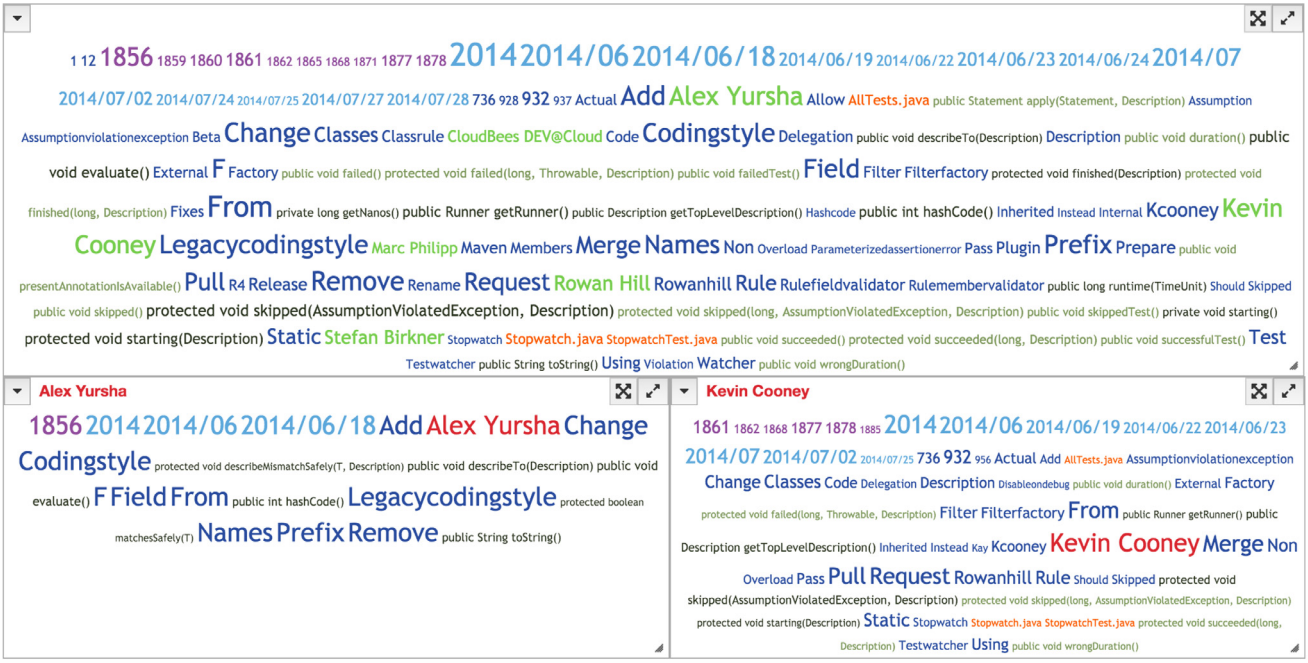


Fig. 3. JUnit: vacation cloud for David Saff constructed from a change-based context. Main tag cloud view (top). Changes by Alex Yursha (bottom left) and Kevin Cooney (bottom right). Alex Yursha and Kevin Cooney are selected with sticky tags. Only tags with occurrence greater than two are shown.

system. Combined contexts can be used to visualize which files have been changed when a bug has been fixed as well as showing the project activity both in terms of commits and issue reports.

3.3. 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 12. 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.

3.4. 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 (see 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.

Intuitively, deselection should be the inverse of selection: deselecting the last selected tag should move the focus back to its previous position. Because of the duality in the concept lattice, we would expect the de-selection operation to be implemented by the join in the lattice. However, using the join operation to deselect an attribute a would move the focus up in the lattice and effectively de-select all other currently selected attributes *except* a, which leads to counterintuitive results. We must therefore recompute the focus as the meet of the defining concepts of the remaining selected tags, in order to provide a de-selection operation which is the inverse of the selection operation.

3.5. Relation to information retrieval

Our lattice-based browsing approach is related to classical information retrieval (IR) [30,31]. 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. A concept in the lattice contains for each document in its extent, the set of terms that occur in the document in its intent. 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,

Please cite this article as: G.J. Greene et al., Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices, Information and Software Technology (2016), http://dx.doi.org/10.1016/j.infsof.2016.12.001

we can exploit the lattice structure, e.g., to update the focus incrementally, or to show which other tags are implied by (i.e. always occur along with) the current selection set.

4. ConceptCloud browser

We have implemented our approach in the ConceptCloud browser. The VISSOFT 2015 evaluated artifact [17] is available at vissoft15.conceptcloud.org/ and the continuously updated web application is available at www.conceptcloud.org. Our browser can automatically index Git and SVN repositories and create tag cloud visualizations from them. It also supports more advanced preprocessing and interface customizations.

ConceptCloud comprises three main components that extract meta-data from the revision control system, construct a context table in the desired format, and display the tag cloud 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 [32].

ConceptCloud currently supports extraction of meta-data and construction of context tables from SVN [19] and Git [33] 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 3.2. 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 [34] 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.

4.1. Tag cloud interface

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 [35]. 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 [36] 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 [37]; 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,

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

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

A variety of alternative tag layout methods have been proposed, such as tag flakes by Caro et al. [38]. Tag flakes are used in order to provide context for tags as basic tag clouds fail to show how the tags are related [38]. However, instead of using a more complex visualization that depicts the relationships between the tags, 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 contextual information for the currently 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 the cloud is only showing attribute tags from a subset of the total objects in the context table.

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.

Proposition 13. *Let $c \in \mathcal{B}(\mathcal{O}, \mathcal{A}, \mathcal{I})$ be a concept, $o \in \mathcal{O}$, and $t \in \mathcal{O} \cup \mathcal{A}$. Then (i) $o \in \tau(c) \Rightarrow \sigma(o) \leq c$, and (ii) $t \in \tau(c) \Rightarrow \delta(t) \wedge c \neq \bot$.*

Since the tag clouds can be very large we provide functionality in the interface to limit clouds to one particular category (e.g., commit authors), or to remove unwanted categories from them. The cloud can also be adjusted to show only a certain number of tags or to show only 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 already know which tag they want to select (such as their 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, Fig. 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 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, scaled according to how often they are changed together. Fig. 2 allows us to answer questions such as "Who has changed this file?" (i.e., expertise) , "Is this file still under development?" and "What other files should I be looking at if I want to change this file?" (i.e., co-changed files).

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. 4) which update simultaneously when another tag is selected in any viewer. Sticky tags therefore enable us to show mutually exclusive views in two tag clouds next to each other.

A tag is *implied* if it has not been selected explicitly, but corresponds to an attribute in the focus' intent. Implied tags reveal the repository's internal structure, similar to the way association rules reveal the implicit structure of shopping baskets [39] but without any additional cost.

4.2. Advanced visualization in ConceptCloud

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 4.2.3 below.

ConceptCloud also supports a number of advanced visualizations such as customizing a specific tag cloud or using a scripting language to automatically layout the ConceptCloud interface.

4.2.1. 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

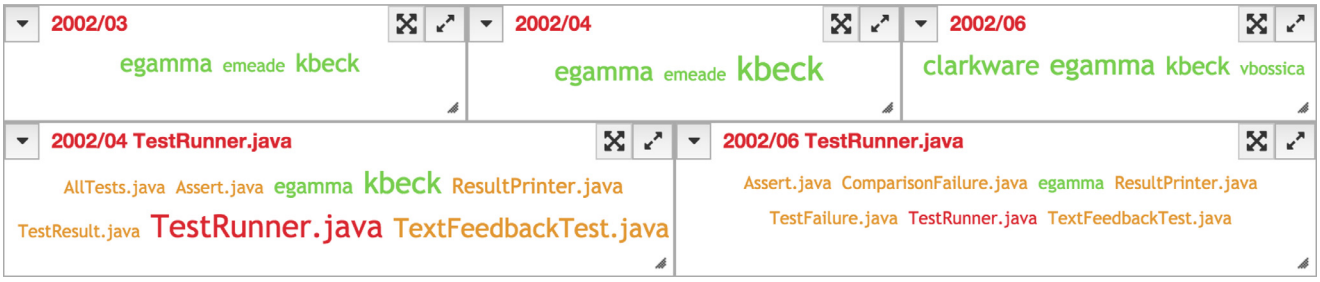


Fig. 4. JUnit: author clouds (top); changes to TestRunner.java (bottom). Tag Clouds constructed from a file-based context and months/files are selected as sticky tags.

to different categories than the tags from the remaining commits (such as "file of interest"), 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 easily identifiable in views even when the tag for that developer has not been selected.

4.2.2. 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 restrict the revision range from which the context table is constructed. Our pre-processing offers different ways of specifying the ranges of interest, such as processing only a certain number of commits or processing only commits falling between a specified start and end date.

4.2.3. Customized visualizations

The combination of personalization and filtering steps allows ConceptCloud to highlight answers to the question "What happened in my project while I was away?" with a *vacation cloud* as for example shown in Fig. 3. 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. 3). 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. 3 does not provide us with many detailed insights into the data and we refine the view by selecting tags in order to discover more insights. 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 to many files. Alternatively, selecting Cooney reveals that he has merged in several pull requests (cf. Fig. 3) 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 previously changed by Saff from those in other files. We can therefore see that different variants 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 also worked on (shown in dark grey).

4.2.4. Scripting tag cloud viewers

We have developed a scripting language, ConSL, in order to construct and lay out multiple viewers in ConceptCloud simultaneously. Scripts can be written in order to open viewers for specific categories, open viewers with sticky tags (i.e., selections that are unique to the view and cannot be modified in the view) and to customize the layout of the viewers in ConceptCloud's interface. While manually opening viewers from the ConceptCloud interface is useful for exploration of a repository, opening multiple viewers (such as all tags from a particular category) and manually laying them out can be time consuming. ConSL scripts provide a mechanism to easily recreate a particular viewer layout and can be used on multiple datasets so that the datasets can be compared using the same custom layout. The same script can also be loaded every time a dataset is loaded so that there is no need to manually configure the tag cloud layout on opening ConceptCloud. After executing a ConSL script the user can still perform all available customizations through the interface.

```
define author_view as tag_view {
        category = 'author ';
}
for tag_x in ['year'] {
        open author_view(tag_x);
}

layout author_view{
        width = '50
        menu = 'hidden';
}
```

Listing 1. Example of a script written in ConSL. The author view shows only author tags and the for-loop opens an author view for each year tag selection. Views are sized at 50% of the full screen width and viewer menus are hidden.

ConSL scripts are compiled and used to generate JavaScript code that is executed in the browser where ConceptCloud is loaded. ConSL provides four main operations: defining a view, for-loop constructs, opening a view and setting layout. A view can be defined with one, multiple or all categories of tags in the tag cloud. Views can then be opened with optional sticky tag selection arguments. For example, a view showing only the authors in the project can be defined and then this view can be opened with selection of year tag 2015, to open a view showing all project authors in 2015. A for-loop construct is provided to open multiple viewers with sticky tags from all tags in a certain category, such as a sticky tag for each year (see Listing 1), which can be tedious to achieve manually through the interface. ConSL's layout functionality allows the user to specify a precise layout and ordering for all the viewers. For example, Listing 1 shows a layout where each row will contain two viewers of equal width. The internal menus of each of these viewers will be also be collapsed. Alternatively, using the interface's drag and drop functionality to manually resize and layout multiple viewers can often lead to imprecise layouts. ConSL scripts can be loaded at the same time as a saved ConceptCloud context table, or from the tag cloud view. This enables users to load scripts after initial exploration of the dataset.

Please cite this article as: G.J. Greene et al., Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices, Information and Software Technology (2016), http://dx.doi.org/10.1016/j.infsof.2016.12.001

```
define author_cloud as tag_view {
        category = 'author';
}
define test_runner as tag_view {
        category = 'author, filename';
}
open author_cloud('2002/03');
open author_cloud('2002/04');
open author_cloud('2002/06');
open test_runner('2002/04, TestRunner.java');
open test_runner('2002/06, TestRunner.java');
layout author_cloud {
        views_in_a_row = 3;
}
layout test_runner {
        views_in_a_row = 2;
}
```

Listing 2. ConSL script for generating author by month view of the JUnit repository (Fig. 4).

5. Illustrative application examples

We apply our ConceptCloud browser to two open source repositories and one industrial application to demonstrate the insights that can be obtained using the browser. We repeat and expand on a previous case study on the JUnit repository in Section 5.1 to highlight the flexibility of our browser. We also show how the browser can be used to explore both combined version control and issue data simultaneously using the RubyGems repository in Section 5.2. We have also applied our browser to generate insights from a small industrial project (see Section 5.3) in order to evaluate the appropriateness of the insights that can be gathered with ConceptCloud.

5.1. JUnit repository

JUnit is a popular open-source testing framework for Java which has been used in previous studies [40,41]. Here we repeat Weissgerber's study [40], which investigates developer roles up until 2006, and extend it to a more current date. We show that we can easily extend the previous observations on the repository through our interface even though our interface was not specialized only to identify collaboration patterns. We also show that we can make the same observations using our ConceptCloud browser as the customized visualizations for each aspect presented in [40]. 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 egamma started the project in December 2000. In an author cloud for the first full year of development (2001) we see that developers kbeck and emeade join the project in 2001 but egamma remains the most prolific author in that year (cf. [40]).

Authors by month: Weissgerber et al. [40] look 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 (cf. Fig. 4, top). Selecting an author tag shows us which files the author has worked on in each month. Fig. 4 shows kbeck's contributes less and less in the given period. The cloud for June 2002 shows the addition of developers vbossica and clarkware to the project.

Selecting the file TestRunner.java, shows that egamma and kbeck have changed this file in April 2002 and only egamma has made changes in June 2002 (cf. bottom of Fig. 4). We also see that there is a group of files which have been changed at the same time.

Differences in visualizations: while the visualization presented by Weissgerber et al. [40] is an author file graph which shows for each author lines connecting the author to a specific file (which is represented as a dot in the graph) our visualization shows the different tag sizes for the developers according to their amount of contribution. In the author file graph visualization the amount of nodes connected to a developer can be used to assess the amount of their activity, whereas in the tag cloud their tag size directly corresponds to the amount of activity. Additionally, by selecting author names in the tag cloud the names of the corresponding files that these authors have been changing will be shown in the tag cloud. It is unclear how the author file graph presents the names of the files which have been changed. The author file graph [40] also allows the identification of developer collaboration: if two developers are linked to the same node they have collaborated on a file. In our tag cloud view from a file-based context the selection of a particular author would update all other author tags to show only authors that have been collaborating with the selected author in a size that represents the amount of collaboration. Therefore, in our tag cloud view the identification of collaboration is interactive and it is also scalable, since the tags for all collaborating developers can be shown at the same time. Using the sticky tag function, comparisons between different groups of collaborators can also be easily drawn, by comparing the tag clouds. The file author matrix [40] shows a grid-like summary of which developers have been working on which files across the project, where each pixel color indicates the amount of activity on a file. In our tag cloud visualization files can be selected to see which authors have been working most actively on a file and authors can also be selected to indicate on which files they have been working. A summary view across a group of developers (or files) can be created by making sticky tag viewers for the group of developers and comparing the tag clouds created.

Conclusions: ConceptCloud allowed us to gather the same insights as the dedicated tool presented by Weissgerber et al. [40]. However, ConceptCloud does not produce a static picture but allows the user to refine the analysis, and access the other information (e.g., log messages) that remains available.

5.2. Rubygems repository

We constructed the combined context for commits and issues from the RubyGems GitHub repository [42] to show how we can combine issue and repository data in the same tag cloud. The GitHub issue tracking system provides links between issues and commits that either close an issue or reference it. We extract these links, using the GitHub API, to create explicit links between issues and commits in our tag cloud, but we also extract keywords from the issues and commit messages and use these to create implicit links between issues and commits that discuss the same topics. For other issue tracking systems that do not include explicit links between issues and commits we would still be able to extract implicit links from the commit messages.

Linked issues and commits appear in the same tag cloud, showing which files have been changed in order to close an issue. For example, Fig. 5 shows the tag cloud containing information for issue 227 which was closed by commit 3642. We can immediately see that files rubygems.rb and specification.rb were fixed in relation to the bug reported about inactive gems. We see here tags #227 as well as tag 227, where #227 represents the issue object and 227 is part of the commit message for commit 3642. We see also tags r3642 and 3642, where 3642 represents the revision object and r3643 is used as a link between both the revision and

Fig. 5. RubyGems: tag cloud for commit 3642 which closes issue 227 in RubyGems.

```
define main as tag_view {
        category = 'message';
}
define files_view as tag_view {
        category = 'filename';
}
define committers as tag_view {
        category = 'author';
}
define reporters as tag_view {
        category = 'issue_reporter';
}
open main('Gem, Install');
open files_view('Gem, Install');
open committers('Gem, Install');
open reporters('Gem, Install');
layout main {
views_in_a_row = 1;
menu = 'hidden';
}
layout files_view {
        views_in_a_row = 2; menu = 'hidden';
}
layout committers {
        views_in_a_row = 0.5;
        width = '15
}
layout reporters {
        views_in_a_row = 0.5; menu = 'hidden';
}
```

Listing 3. ConSL script for the view in Fig. 6.

the issue objects. Therefore, while 3642 and #227 occur only once, r3642 occurs twice (and appears bigger) in the tag cloud as it is an attribute which applies to both the issue and the revision objects in the tag cloud.

Additionally, we can explore all commits and issues that discuss a particular topic such as gem install. Fig. 6 shows the main files (orange tags), committers (green tags) and GitHub issue reporters (maroon tags) that are associated with the keywords "gem" and "install". We can see the main files changed that fix issues mentioning gem install and also files changed in commits where the commit message mentions gem and install. We can further restrict the cloud to showing only commits that have closed a bug report (by selecting the bug report status closed tag) mentioning words gem and install (Fig. 7). We see that Eric Hodel is the only author that makes commits closing issues that mention gem and install and these commits only occur in 2013 and 2014. This indicates that while other authors have also made commits mentioning gem and install, Eric Hodel is responsible for this area as he has either fixed issues referring to gem install or has been responsible for closing these issues when merging a pull request from another developer.

Conclusions: ConceptCloud can be used successfully to combine multiple data sources to get more detailed information on a specific project. We can fuse issue and repository data into the same underlying context table and explore commits that are related to specific issues in the tag cloud interface.

5.3. Industrial application

We used ConceptCloud to analyze the Git repository of a small, local non-profit organization. This project develops an educational service comprising of a mobile app, backend, and data analytics. The goal of this application was determine whether the insights that we gather using ConceptCloud are appropriate and can be confirmed by the project manager. Since this is a small localized development team the insights that we gather might not be surprising to the project manager, but we aim to validate the accuracy of our observations. We analyzed the project from its start (08/2015) up until the app's release to the Google Play Store (01/2016). Note that we use only abbreviations of the developers' commit names here to preserve their anonymity.

Project contributors: by creating author viewers for each month from the revision-based context of the project (Fig. 8) we saw that the project started towards the end of August 2015 with only three developers. In September all three developers contributed in similar amounts to the project. In October two more developers (CM and P9) joined and overall commit activity of the developers greatly increases. Additional contributors RS and S also joined the project in November, and this team remained relatively stable with only the addition of PW in December. The team structure changed again in January with SM, PW and RS leaving but two new (and therefore less-active) contributors, HW and F joining the project. In each month the developers (excluding the new additions) appeared to be sharing the workload uniformly. We see that project was expanding but also that there was a high developer churn. The project manager confirmed that contractor SM was replaced by two new full time developers in January.

We can also observe other apparent small contributors (with one to three commits) which on further investigation appear to be alternative aliases (particularly GitHub usernames) for some of the contributors (such as a developer editing the ReadMe file directly on GitHub and the commit being recorded with their GitHub username). These alias characteristics could also be incorporated to identity merging techniques as identity merging in projects is a difficult problem [43].

Developer collaboration: by creating the file-based context of the project we can observe collaborations between the developers. Selecting developer LS (Fig. 9) showed that he often collaborated with developers SM and P9, however there are a small amount of files common to other team members as well. Developers F and HW have also collaborated with LS since they joined the project. If we select an additional tag for developer AV (who showed up only as a small tag) and show which files both AV and LS have changed, we saw that the gitignore was the only file common to most of the development team, which indicates that they also have different project focuses.

If we select the tags for LS's collaborators SM and P9 and show the tag cloud for the changed directories (Fig. 10(a)), we see a directory structure that indicates that these developers worked on the Android client. This cloud confirms that developers F and HW had also begun working on the Android client.

The directory cloud of developer AV (Fig. 10(b)), who collaborated mostly with S and CM shows a very different directory structure (appearing to be concerned with backend development). Therefore, we see a clear separation of responsibilities among the development team. However, when we investigate the collaboration clouds of S and CM individually we see that these three developers each worked on a number of files that are not touched by other team members. If one of these developers were to leave the team there would be a large number of files that no other team

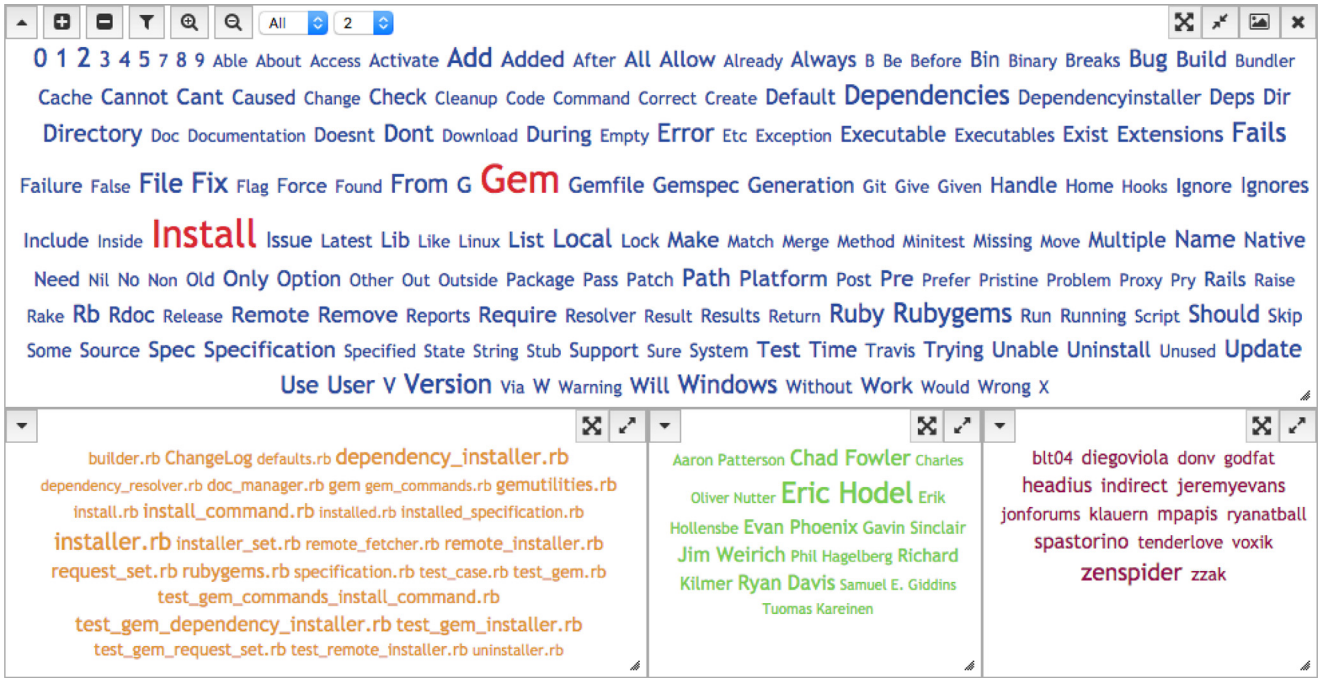


Fig. 6. RubyGems: main changed files, committers and bug reporters from commits and issues mentioning Gem Install. Tag clouds constructed from a combined context of GitHub issues and commits.

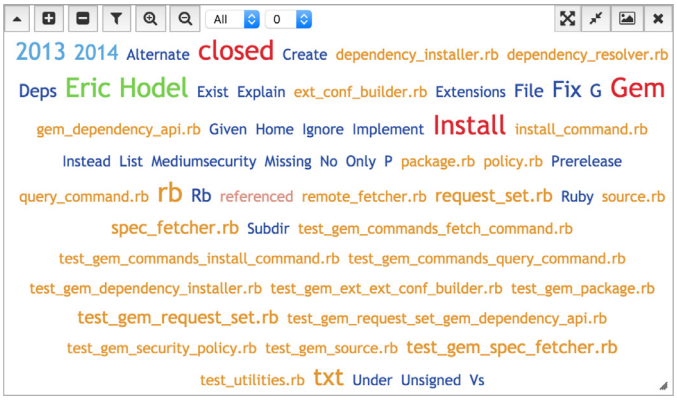


Fig. 7. RubyGems: changed files, committers from commits closing issues mentioning Gem Install.

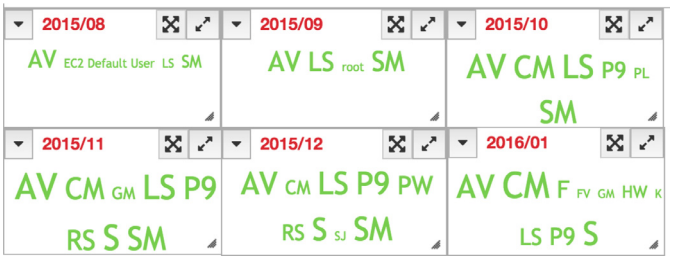


Fig. 8. Industrial application study: developer contributors over project from project start to first release. Tag clouds generated from a revision-based context. Months are selected as sticky tags.

member would be familiar with. Therefore, we see that the backend team has a very low "bus factor" [44].

Contributer RS did not appear as one of the main collaborators of AV's team (backend development) or LS's team (android application) and on further investigation of RS's changed directories we see that he contributed mostly images to the project.

Fig. 9. Industrial application study: collaboration with developer LS. Tag cloud build from a file-based context.

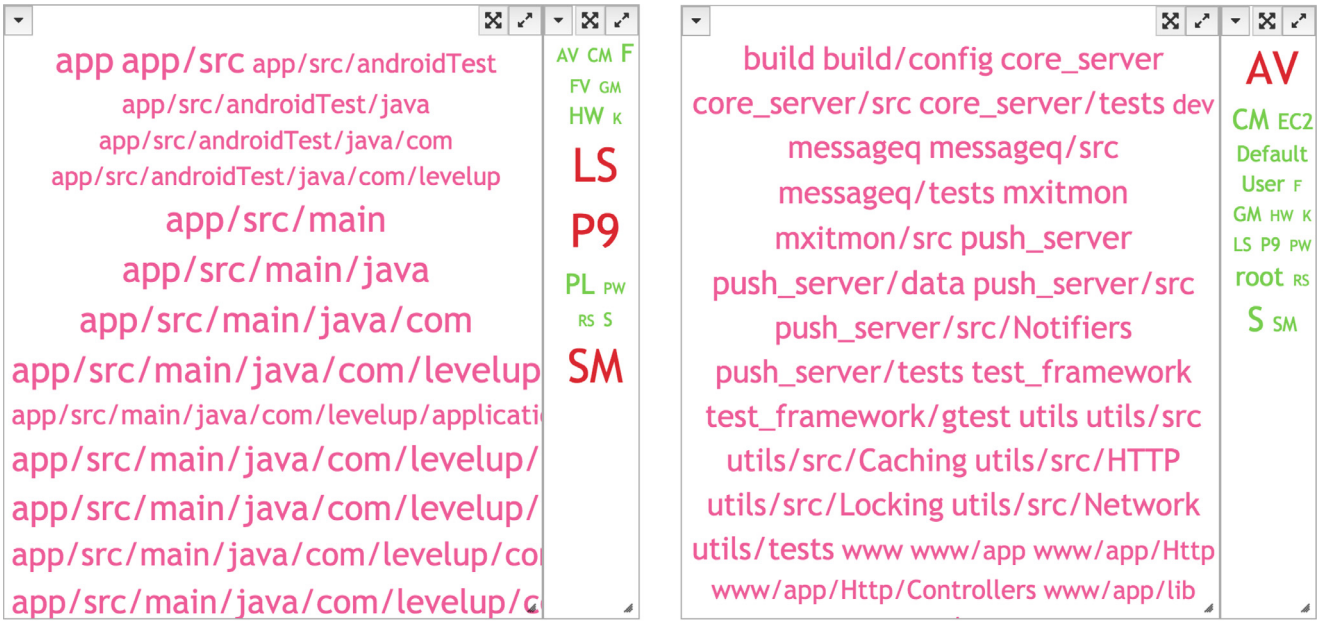
Commit activity: comparing the revision-based and file-based views on the weekdays on which the developers have been making commits (Fig. 11) we see that the most commit activity occurs between Tuesdays and Thursdays, with less activity on Mondays and Fridays and very little over the weekends (Fig. 11, left). This is consistent with what we expect from a full-time commercial development team.

Observing the number of files changed on each weekday (Fig. 11, right) shows that while less commits are made on Fridays these commits generally touch more files. This would be consistent with developers committing their changes before the weekend in fewer but larger commits. The project manager also indicated that bi-weekly sprint planning takes place on a Friday, which could also explain the fewer but larger commits observed on Fridays.

Commit messages: examining the most frequent words used in commit messages in the first full month of the project (09/2015) and comparing those to the commit messages in 01/2016 (Fig. 12) we see that the initial activity was largely concerned with Facebook and Database integration. In the last month examined (01/2016) we see that the activity is more centered around bug fixes and the user interface changes (Images, Styling), which indicates that the project was about to be released.

5.3.1. Threats to validity

The study on the industrial application has been performed by the first author. However, to mitigate risks the first author had no previous knowledge of the project or development team. We



(a) (b)

Fig. 10. Industrial application study: directories and collaboration of developers (a) LS, SM and P9 and (b) AV. Tag cloud build from a file-based context.

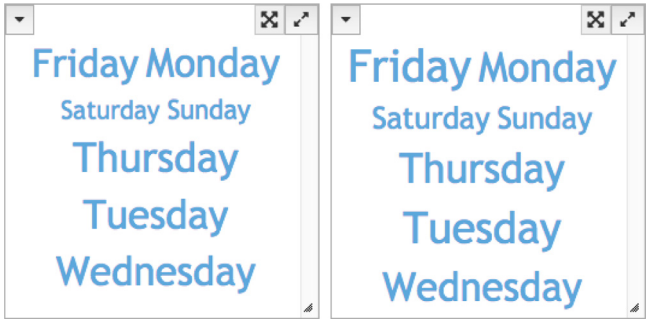


Fig. 11. Industrial application study: weekdays of developer commits. (left) tags sized according to number of commits (right) tags sized according to number of files changed.

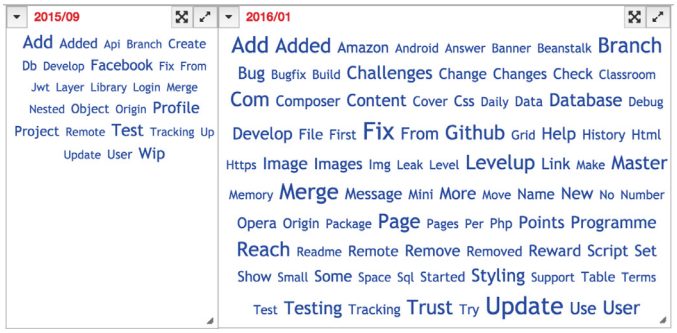


Fig. 12. Industrial application study: popular keywords from commit messages in the first and last month analyzed .

subsequently verified all observations with the project manager in order to establish their correctness.

5.3.2. Discussion

We have applied ConceptCloud to an industrial project to determine whether the insights that we can gather are appropriate and can be confirmed by the project manager. For this application, the team was small and co-located so the insights might not be surprising to them. However, we have seen that there are many valuable insights, such as team collaboration, areas of expertise and activities, contained in the version control repository. Using ConceptCloud we were able to gather these insights which would be very valuable for new developers starting on the team and teams in which the collaboration patterns or activities are not obvious to the project manager. We could identify the different roles of developers in the team by examining the directory structure of the files they committed, which indicated what parts of the system members were working on. We were also able to identify when developers joined and left the team and how the different team members collaborate.

5.4. Conclusions

Using ConceptCloud we are able to get an overview of the collaboration and work patterns in both open source and industrial projects. We see that the different context types give us different views on the project (e.g., collaboration vs. commit activity) and that additional project information such as the issue-reports can be merged into the same context to provide more detailed information on a project.

Comparing our observations from an industrial project to those made from open-source projects we observe that the commit activity in the industrial project is much more regular and the contributions are shared relatively evenly among the contributors. The development team of the commercial project is also separated into smaller groups ($+/-$ 3) that work consistently on one aspect of the project. In contrast, in the open-source projects we observe one main contributor who has a much higher activity than the others during their involvement; when this contributor leaves the project another developer takes over this role.

6. Performance evaluation

FCA is commonly associated with high run-times and so we evaluate ConceptCloud's performance on a variety of repositories to illustrate the feasibility of our approach. In particular, we used ConceptCloud on a medium-sized server with 64GB RAM and two

Table 1
Metrics for revision-based contexts.

Project	Type	$\|O\|$	$\|A\|$	$\|I\|$	Indexing time (s)	Initial cloud creation time (s)
Subversive[a]	SVN	1511	8222	88,090	55.5	1.8
JUnit[b]	Git	1905	5959	66,242	8.0	1.9
AngularJS[c]	Git	5547	9055	133,436	116.2	2.8
Spring[d]	Git	9017	40,332	540,813	43.4	14.8
Valgrind[e]	SVN	10,989	29,009	348,136	176.6	40.0
Django[f]	Git	18,471	38,821	583,701	58.4	11.0
Moodle[g]	Git	69,550	154,834	2,222,486	333.2	45.7
DPorts[h]	Git	155,627	196,850	2,917,269	2,049.9	892.8

[a] https://dev.eclipse.org/svnroot/technology/org.eclipse.subversive/. Analyzed December 2006 to September 2014.
[b] https://github.com/junit-team/junit. Analyzed December 2000 to September 2014.
[c] https://github.com/angular/angular.js. Analyzed December 2013 to September 2014.
[d] https://github.com/spring-projects/spring-framework. Analyzed July 2008 to September 2014.
[e] svn://svn.valgrind.org/valgrind/trunk Analyzed March 2002 to September 2014.
[f] https://github.com/django/django. Analyzed July 2005 to September 2014.
[g] https://github.com/moodle/moodle. Analyzed November 2001 to September 2014.
[h] https://github.com/DragonFlyBSD/DPorts. Analyzed October 2012 to September 2014.

Xeon 8-core 2.0Ghz CPUs to analyze several Git and SVN repositories in order to evaluate its performance.

We created revision-based contexts (using local clones of Git repositories and remotely accessing the SVN repositories). Table 1 summarizes the characteristics of and runtimes for these repositories, showing 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 time to create the context table (i.e., indexing) and to draw the repository's full tag cloud.

We see that the indexing times (including the extraction of all of the log information for the repositories) are only a few seconds for smaller repositories, and a few minutes for medium-sized ones; even the largest repository with 155,627 revisions requires only 34 min. 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 initial cloud creation times are given for the full tag cloud for the repository, which contains $|O| + |A|$ tags. The table thus gives an indication of the cloud computation in the worst case; in practice, we can limit the number of tags shown to substantially improve this. However, the initial tag cloud is cached and so can be generated off-line in a pre-processing step. Subsequent loads of the initial tag cloud from cache are instantaneous.

Tag clouds become smaller with subsequent navigation steps and are therefore created substantially faster. Overall, navigation is instantaneous for small and medium repositories, with some degradation on the initial clouds for very large repositories.

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

To reduce drawing time for larger repositories we could limit the number of tags shown in the initial tag cloud to only those that apply to a larger portion of the revisions in the repository and then show the full tag set when the user has made selections to refine the tag cloud. For large repositories that are indexed repeatedly, our approach allows us to incrementally update the context table (and therefore the concept lattice) so that updates can be performed quickly and the initial indexing needs to only be performed once.

7. User study

We performed a user study in order to evaluate whether untrained users are able to answer questions about the history of a software project using ConceptCloud more or less effectively than with current widely-used interfaces. In particular we compare ConceptCloud to the default list-view of commits as implemented in GitK and the GitHub interface, which is graph-based. Both GitHub and linear list commit views, such as GitK, are widely used in practice and we therefore use these interfaces as the controls for comparison against ConceptCloud. Linear list commit views are implemented in many popular Git GUIs (such as SourceTree[1] and TortoiseGit[2]), but we use GitK as it is packaged standard with Git. GitK provides a searchable linear list of commits and shows the diffs between two revisions. GitHub's interface is widely used in order to visualize the history of a software project and provides graph views of user's activity in repositories. GitHub also provides a code search interface. GitK, GitHub and ConceptCloud present the same underlying information through different interfaces. We therefore compare the effectiveness of our tag cloud interface to that of a searchable list interface and an interactive graph-based interface. Since the participants in our study had never used our ConceptCloud browser before, we also investigate whether the browser can be used successfully by untrained users.

In this study, we aim to answer the following research questions:

RQ1: is a rich exploratory interface, such as our interactive tag cloud interface, accessible to untrained users?
RQ2: does our interactive tag cloud interface allow users to achieve higher correctness than the familiar linear list view of commits when answering questions about the history of a software project in a set time period?
RQ3: does our interactive tag cloud interface allow users to achieve higher correctness than a graph-based interface, such as the one provided by GitHub, when answering questions about the history of a software project in a set time period?

[1] https://www.sourcetreeapp.com/ .
[2] https://tortoisegit.org/ .

Please cite this article as: G.J. Greene et al., Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices, Information and Software Technology (2016), http://dx.doi.org/10.1016/j.infsof.2016.12.001

Table 2
Question set for user study, (a) Ruby Gems (b) Backbone (c) Retrofit.

(a)	*RubyGems:*
1	Who is the contributor with the most commits on the Ruby Gems project?
2	In which year were the most commits made to the project?
3	Which file types has Charlie Somerville changed in his commits?
4	Which contributors have worked on the file lib/rubygems/psych additions.rb?
5	Who has been making the most changes on the project since Samuel E. Giddins last worked on it?
6	When was this repository created?
b)	*Backbone:*
1	In which month was the most activity on the project?
2	Who was the most active developer in this month?
3	Who is the most prolific author of the backbone/test directory?
4	Who was the last person to change the file backbone.js?
5	Which file has been changed the most in this project?
6	Who has made the most changes to the images in the project (jpg, png)?
7	Who has changed the most files that Brad Dunbar has also changed?
c)	*Retrofit:*
1	Where are the tests for the main project located?
2	Who has edited the .yml files?
3	Who contributed the most to this project in its first year?
4	Who has worked on JacksonConverter.java?
5	Who merged pull request #1017?

7.1. Experimental setup

We used a between-subjects design to conduct the experiment, where each participant uses only one of the three tools to answer questions about the software development process in specified projects. We constructed three questions sets, based on three different software repositories that were also available on GitHub. All participants were asked to answer three question sets using a tool (GitK, GitHub or ConceptCloud) which was randomly assigned to them. We then evaluated the correctness of the answers supplied by the participants. Each participant was supplied with a user manual, detailing how their tool showed the history of software projects. We marked all of the question answers that were submitted by the participants and calculated their results. We investigate the hypothesis that there is no difference between the correctness results obtained by the participants over all three tools.

Our user study took place in a computer lab at Stellenbosch University. All participants took part at the same time to avoid communication about the tasks. Participants were not permitted to communicate during the study.

Resources for the user study, including question sets, sample solutions and the versions of the repositories used, are available at www.conceptcloud.org/userstudy15.

7.2. Population

We performed our user study with students in our third year Software Engineering class of 2015. Previous courses required the students to submit assignments using Git repositories, so all were all familiar with Git. The participating group consisted of 47 students in total. Participation was voluntary for all students.

7.3. Tasks

We developed three question sets using three different repositories available on GitHub, namely RubyGems [42], Backbone [45] and Retrofit [46] (see Table 2). We selected these repositories as they are popular projects available on GitHub, and they differ in size. At the time of the user study the RubyGems repository was the largest with 6388 revisions, Backbone consisted of 3130 revisions and Retrofit had 998 revisions. We used repositories of different sizes so that the results of our study would not be biased towards one repository size.

Table 3
Descriptive statistics for average percentages obtained with each of the three tools across all questions.

	GitK	ConceptCloud	GitHub
Mean	0.52	0.71	0.67
Sd	0.21	0.10	0.10
Min.	0.27	0.55	0.53
Max.	0.84	0.90	0.85
Range	0.57	0.36	0.32

The question sets were developed by exploring the repositories equally using GitK [47], GitHub [27] and ConceptCloud. Question sets included questions about the location of files, collaboration of users, expertise of the contributors as well as the history of the projects. The question answers were then verified using all three tools to make sure that the results were consistent. All questions were weighted equally. We used all three tools to generate the question sets because the different tools have different strengths and weaknesses and using only one tool would have made the questions easier to answer for the participants assigned to a specific tool.

Participants were given 15 min to answer each question set, (6, 7 and 5 questions respectively) after which they were given the next question set and corresponding repository. Participants were made aware of this time limit at the beginning of the user study and before each new question set was started. Participants were asked to answer as many questions as they could in the time provided and to move on from a question when they were unable to answer it.

7.4. Analysis and results

We used the R package for analysis of the experimental results. We performed the Shapiro and Wilk [48] test to determine whether participants' scores were normally distributed, in order to determine what further analysis could be performed. We obtained a p-value of 0.06, and at a confidence level of 0.05 we cannot reject the null hypothesis and conclude that the data is normally distributed.

7.4.1. Summary statistics

Fig. 13 shows a summary of average correctness percentages achieved by participants for each question set, in the order that the question sets were answered (Ruby Gems, Backbone and Retrofit). In the first question set users of GitHub performed the best, and for the second and third question sets users of ConceptCloud performed the best. Fig. 14 shows a box-and-whisker plot for the average scores obtained across all questions for each tool. We see that the median as well as the minimum value for participants using ConceptCloud is the highest, followed by GitHub and then GitK. The range of results of participants using GitK is the highest, with some participants achieving high averages and others achieving much lower results than those using either GitHub or ConceptCloud. Fig. 15 shows the box-and-whisker diagrams for the percentages obtained across each of the question sets for each of the tools. Participants using ConceptCloud achieved higher median percentages for each new question set, which indicates there might have been some learning effect observed over the different question sets. However, participants using GitK or GitHub performed worse in the second question set and then again better in the third question set.

Please cite this article as: G.J. Greene et al., Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices, Information and Software Technology (2016), http://dx.doi.org/10.1016/j.infsof.2016.12.001

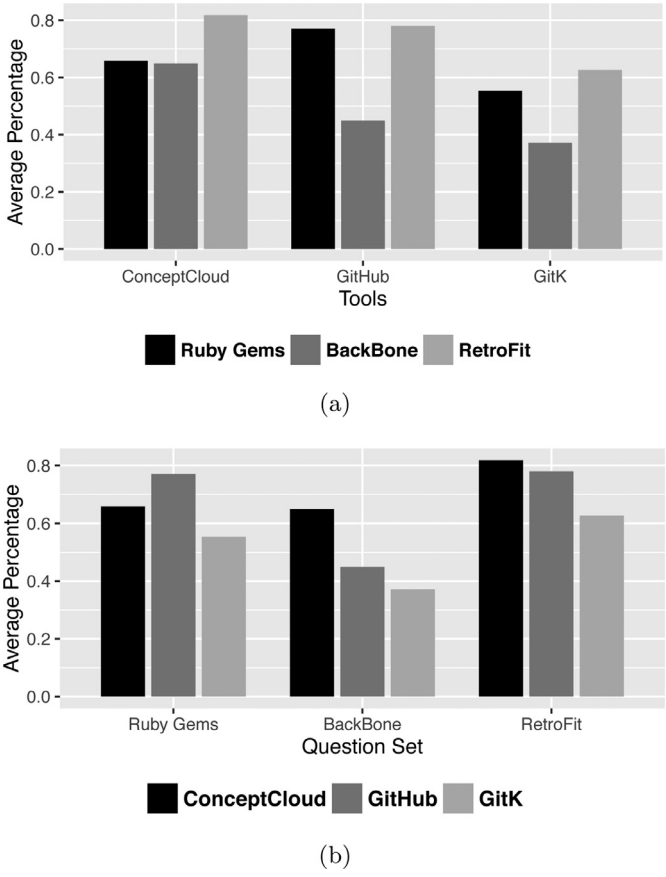


Fig. 13. Average percentage obtained by participants, across all three tasks, using ConceptCloud, GitK or GitHub. (a) Bars from left to right indicate: ConceptCloud, GitHub and GitK (b) Bars from left to right indicate: Ruby Gems, Backbone and Retrofit Questions.

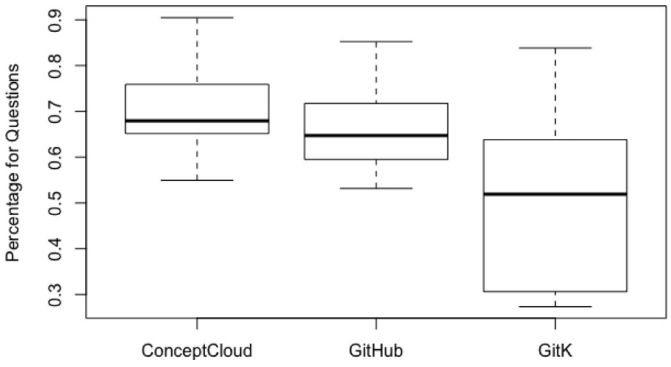


Fig. 14. Box and whisker plots for average percentages obtained using ConceptCloud, GitK or GitHub.

Table 4
P-values for Tukey test.

Tool comparison	*P*-value
GitHub–ConceptCloud	0.6343916
GitK–ConceptCloud	0.0006546
GitK–GitHub	0.0059474

7.4.2. Statistical significance

We performed a two-way ANOVA test on the correctness obtained by each participant across all three question sets to determine if there was any statistically significant difference in the correctness obtained by users of the different tools. We tested our null hypothesis that the results of the participants would be the same over all three tools. We formulated this null hypothesis so that we would be able to conclude whether there was any difference in the performance of the tools, rather than only investigating whether one tool was better than another. Since we have more than two tools to compare we perform a two-way ANOVA test as opposed to a *t*-test because the *t*-test only accounts for the comparison of two tools. We first checked for interaction effects of the tools and the question sets. We found that the interaction effects were not statistically significant ($p = 0.11$). Therefore there is no evidence that the variation of correctness between the three tools depends on the question set. This therefore allows us to compare the performance of participants using each tool across all three question sets to draw conclusions on the accuracy obtained with each of the tools. We obtained a *p*-value of 0.000548 from the ANOVA test for the comparison of the tools. We therefore rejected the null hypothesis at a significance level of 0.05 and concluded that the mean values of percentages obtained by participants differed statistically significantly over the three tools. We further performed a post-hoc Tukey test [49] to determine in which tool comparisons statistically significant differences exist (GitHub vs. GitK etc.). The p-values obtained for all comparisons are listed in Table 4. Using a significance level of 0.05 we find that the difference between results obtained using GitK and ConceptCloud as well as GitK and GitHub are statistically significant. A graph plot of the confidence intervals is given in Fig. 16. Therefore we can conclude that participants using ConceptCloud or GitHub were able to answer questions about software projects statistically significantly better than those using GitK.

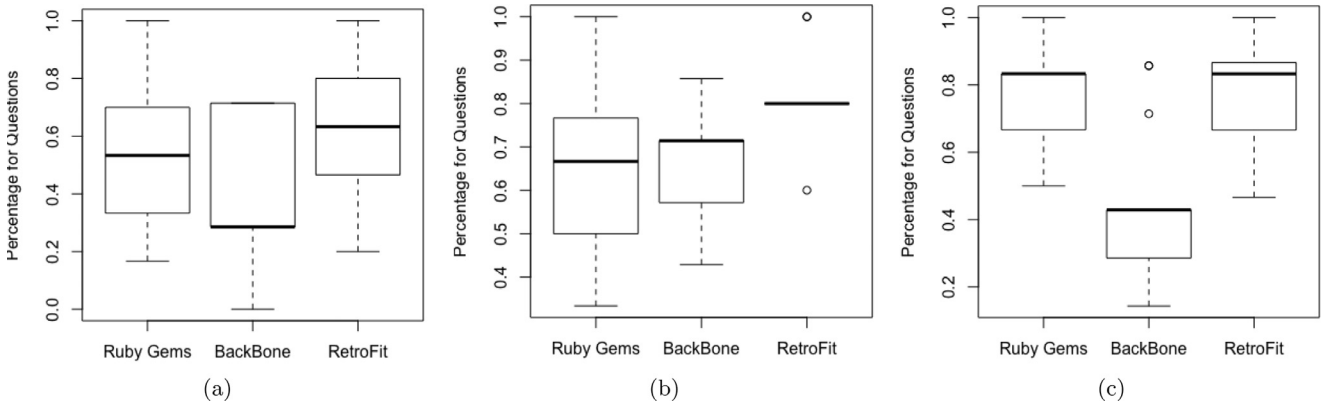


Fig. 15. Average percentage obtained by participants using (a) GitK, (b) ConceptCloud, (c) GitHub across all Three Question Sets.

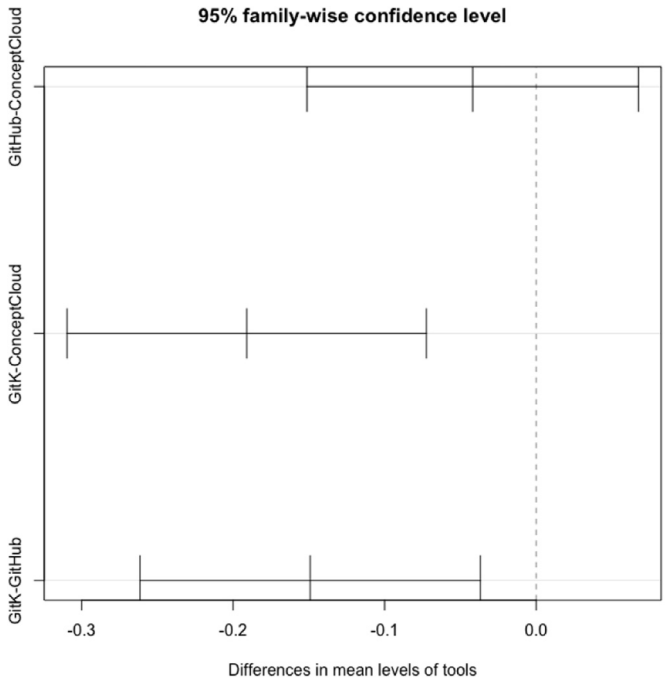


Fig. 16. Confidence Intervals obtained from Tukey test. The GitHub–ConceptCloud interval includes 0, indicating no difference between the means for those two groups.

7.4.3. Question types

We further investigated which user group had the highest result on each question to understand what types of questions were better answered through each interface.

We found that participants using GitHub were best able to answer questions about the activity on the project ("Who is the contributor with the most commits?") as well as which users were the last to change a specific file or who has worked on a particular file. These results are to be expected as the GitHub activity charts prominently show the years and months in which the most commits have been made as well as including an activity chart for each developer. On the GitHub code search interface, specific files can be searched for and these include a list of contributors, so GitHub also makes this information prominent.

Participants using GitK were best able to answer questions about changes occurring after a developer has made their last commit as well as when a project originated. Since the linear list view provides an chronologically ordered list of commits this type of information is easy to obtain from scrolling through the commit list.

Participants using ConceptCloud were best able to answer questions about a user's activity in a specific time period, as well as which files have been changed the most and which developers are changing certain types of files ("Who has made the most changes to the images in the project (jpg, png)?"). ConceptCloud allows users to select a specific time period (month, year etc.) and observe the size of the developers' tags (commits) in this time period. ConceptCloud includes the changed file types as tags, so information about what type of work a developer is doing on a project (front-end etc.) is simple to obtain.

7.5. Discussion

With respect to our research questions we find for RQ1 that untrained users were able to make use of the ConceptCloud tool to answer questions about the history of a software project. On average, participants using ConceptCloud received a correctness percentage of 71% which indicated that they were able to use the tool to answer questions about the project history with a fair amount of accuracy.

In response to RQ2 and RQ3 we observe that while participants using ConceptCloud achieved the highest average over all question sets, these were only statistically significantly better than the results obtained using the linear list view provided by GitK. We can therefore conclude the ConceptCloud interface allows users to answer questions better than a linear list view which is common in many repository GUI tools (RQ2). However, no statistical significance was observed in comparison with GitHub, so we cannot conclude that the ConceptCloud interface allows users to answer questions about a software project better than a graph-based interface (RQ3).

While all participants were able to answer various types of questions through the different interfaces we also see evidence that each interface makes specific activities more prominent.

7.6. Threats to validity

Our user study was conducted using a centralized lab server and so it is possible that some participants experienced slower loading times than others. However, since all of the participants took part in the user study at the same time the load on the servers would have been largely consistent for all the participants in the lab at the time.

The repositories used were of varying sizes and so the results of a specific tool might be influenced by the size of the repository. However, the average correctness percentages were actually the worst for the middle-sized repository (Backbone) for all tools and so we do not see a direct trend showing higher correctness with either smaller or larger repositories for any of the tools. There were also no statistically significant interaction effects between the question set (i.e., repository used) and the tool used which allowed us to compare the performance of each tool over all question sets.

The questions sets that we constructed could have been biased towards one type of visualization. However, to mitigate this risk we constructed questions using observations from all three tools equally and also verified that all questions could be answered correctly using all tools.

Questions were marked by the first author, however the sample solutions were verified using all tools prior to the marking process and so the questions have all been marked using answers that were consistent across all the tools.

The participants might not be representative of a real-world sample of software developers. However, all participants were also involved in their own software development projects and were familiar with Git.

Our sample size is limited, due to the size of our Software Engineering class, however we have made conclusions from our study as much as our sample size has allowed.

The participants might have shown a bias towards our tool as it was developed at their university. However, we have never measured the participants tool preferences, only their performance and since each participant has used only one tool (due to a between-subjects design) their performance should not be affected by any tool preferences.

8. Related work

Our work is related to topics of visualization, navigation using tag clouds and applications of formal concept analysis to software. We discuss a subset of techniques used to visualize version control repositories and bug repositories that can be most closely compared to our approach (see Section 8.1). We also discuss applications of tag clouds directly to software visualization (see Section 8.2) and other navigation techniques used with tag clouds

Please cite this article as: G.J. Greene et al., Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices, Information and Software Technology (2016), http://dx.doi.org/10.1016/j.infsof.2016.12.001

for wider applications (e.g., clinical trial data, see Section 8.3). In Section 8.4, we discuss previous applications of formal concept analysis to tasks in software engineering that are most closely related to the goals our approach, for example, detection of co-changed methods and methods related to a particular bug report.

8.1. Visualizing software and bug repositories

8.1.1. Team structure and developer expertise visualizations

Girba et al. use an "ownership map" visualization [50] in order identify developer interaction and development patterns using the CVS log of a project. Girba et al. also identify several behavioral patterns of developers, such as teamwork, takeover, and cleaning, and show how these can be identified in their ownership map visualization. These collaboration patterns could also be observed in our tag clouds constructed from a file-based context. While the ownership map visualization serves to provide an overview of the project developer patterns in a single visualization, our tag cloud views are aimed at allowing users to interactively explore the contributions. Therefore, while the collaboration patterns might not be visible in a single tag cloud view, our approach aims to support users in exploring the information at varied levels of detail. The user can then also continue exploring other aspects of the project when they have identified interesting collaboration patterns.

Alonso et al. [51] also use a tag cloud visualization to display information from CVS version control repositories. Their "expertise cloud visualization" creates a tag cloud of committers that are identified using a rule-based classification of CVS log information. Users are then able to select the names in this cloud to display a cloud of the developers' expertise. The expertise cloud visualization [51] 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. In contrast our underlying concept lattice only limits the available tag selections to tags that will not cause an empty tag cloud to be displayed.

Weissgerber et al. [40] develop a transaction overview visualization, file-author matrix, and author-file graph to allow identification of team structure, developer collaboration, and project activity over a certain time period from data contained in the version control system. Section 5.1 compares these visualization techniques to the tag cloud view provided by ConceptCloud in the context of the JUnit case study.

8.1.2. Co-evolution of production and test code

Zaidman et al. [52] 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 test code. In our tag clouds we can distinguish between production and test code by observing the project's directory structure.

8.1.3. Centralized data structures and visualizations for multiple software development artifacts

Codebook [28] 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 artifacts 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 archives, bulletin boards, mails, and directory information. Direct queries in a specific format can be given to Codebook to answer different types of questions. Results are displayed in a web interface that provides a ranked result list including images of people associated with artifacts. Results from our user study indicate that list-based interfaces do not support exploration tasks effectively. Codebook has been built with the aim of supporting multiple information needs from software development archives. While the Codebook data storage is flexible enough to support users in answering different types of questions, the applications built on top Codebook are aimed at answering specific questions. With ConceptCloud we aim to have a single application that is flexible enough to support users in answering different types of questions, rather a centralized data-structure which can be used as the base for different applications. However, our context tables can also be seen as a central data structure for storing multiple types of project information.

Hipikat [4] 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 results are displayed in a Hipikat list view Eclipse plugin. However, the goal of Hipikat is more to recommend relevant items to project newcomers and not to provide them with an interface through which to explore the artifacts. Cubranic et al. [4] also note that project artifacts are not easily accessible to developers as searching the archives requires them to know the correct search terms for finding relevant information. In our work we also argue that searching the software development archives does not support all use cases, as to be able to conduct a search the developer already needs to have some information about the archive. In our approach we aim to make the information contained in software development archives accessible to users for interactive exploration so that they can access the information even before they have formulated a direct query. This is a different approach to the recommendations provided in [4] and supports users in exploring the full archives in an unbiased way.

Cubranic et al. [4] also note that while a list-based presentation of results (as used by Hipikat) is very common "when the user's purpose is exploratory browsing of a collection, such a flat-list presentation does not indicate relationships within the results, only to the query itself.". We propose interactive tag clouds as an alternative view, as they allow users to explore query results in an aggregated form and support users in further filtering the results and identifying relationships between them.

Information fragments [53] 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 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. The information fragments prototype is aimed at answering specific questions that developers ask on a day-to-day basis and not on allowing exploration of the underlying archives. While our approach can be used to answer the questions identified by Fritz and Murphy [53] it is specifically aimed at supporting exploration of the underlying archives even for users who have not yet formulated a direct query. While the list-based interface presented in the information fragments prototype groups items together, to show for example which developers have been working on a section of the code, our tag clouds present this type of information through navigation, where the user can select the relevant file or directory and observe the developers that have made changes to it. The "queries" that can be composed through our tag cloud interface are also more flexible in that different

Please cite this article as: G.J. Greene et al., Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices, Information and Software Technology (2016), http://dx.doi.org/10.1016/j.infsof.2016.12.001

kinds of information (e.g., years, files and developers) can be selected at the same time.

8.2. Tag cloud visualizations of software

There have been applications of tag cloud visualizations directly to software for different purposes.

Guido [54] 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. [55] use a tag cloud to visualize the structure of Java class names. Emerson et al. use tag clouds to visualize Java methods and explore several different tag cloud layouts using the TAGGLE tool [56]. 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.

8.3. Tag clouds and navigation

Mesnage and Carmen [11] 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. Gwizdka and Bakelaar [57] 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. This approach is not directly applicable to our tag clouds since we use refinement navigation where multiple tags can be selected. Hernandez et al. [58] use multiple linked tag clouds to browse semi-structured clinical trial data. 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.

8.4. Software and formal concept analysis

Poshyvanyk and Marcus [59] use a combination of latent semantic indexing and concept lattices to find methods that are relevant to a bug report. Girba et al. [60] 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 [61,62] but these only consider an individual program, not a repository.

9. Conclusions and future work

In this paper, we have developed an interactive browser for revision control archives. We use a novel combination of concept lattices and tag clouds, to make the information implicitly contained in repositories accessible to users. 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?”.

By changing the type of objects in the context table (e.g., revisions, files etc.) we are able to provide complementary views on the same underlying data and observe collaboration patterns of the developers. By using changes (i.e., revision-file pairs) as objects we are able to easily identify the co-changed methods in a project. Additionally, our context tables can be used as a centralized data structure for multiple sources of information, such as version control data and bug reports.

Our tag clouds provide a visualization in which version control data can be aggregated and explored interactively to support developers in tasks such as keeping up with project changes. Our interface is customizable through the use of a scripting language, which can be used to repeatedly access a constructed view on the dataset. Our interactive visualization supports users in exploratory search tasks when they have no previous knowledge of a project.

We have used the ConceptCloud browser to repeat a previous case study [40] and to make observations about the internal structure of a small commercial development project. We have also performed a user study to determine the usability of ConceptCloud and to compare its effectiveness in allowing users to answer historical questions about a project to that of other existing information representations. Through our user study we conclude that untrained users are able to make use of our ConceptCloud browser to answer questions about the history of a software project.

In future, we plan to conduct an additional user study which compares our ConceptCloud browser to other tools mentioned in related work (which index repositories as well as additional information sources such as email archives) to determine how the different tools perform in both search and exploratory search tasks. We are currently working on building a generic framework from our ConceptCloud browser so that this can be used to visualize a variety of semi-structured data archives (such as academic paper data) [63]. We are also applying ConceptCloud in a different domain and conducting another user study in which we specifically evaluate the learning effects present when using the tool.

Acknowledgments

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

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.infsof.2016.12.001.

References

[1] J. Sillito, G.C. Murphy, K. De Volder, Questions programmers ask during software evolution tasks, in: Proceedings of the SIGSOFT '06/FSE-14 International Symposium on Foundations of Software Engineering, 2006, pp. 23–34.
[2] M. Codoban, S. Srinivasa Ragavan, D. Dig, B. Bailey, Software history under the lens: a study on why and how developers examine it, in: Proceedings of the International Conference on Software Maintainance and Evolution (ICSME), 2015.
[3] S.E. Sim, R.C. Holt, The ramp-up problem in software projects: a case study of how software immigrants naturalize, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, 1998, pp. 361–370.
[4] D. Cubranic, G.C. Murphy, J. Singer, K.S. Booth, Hipikat: a project memory for software development, IEEE Trans. Softw. Eng. 31 (6) (2005) 446–465.
[5] R. Wille, Restructuring lattice theory: an approach based on hierarchies of concepts, in: Ordered Sets, Reidel, 1982, pp. 445–470.
[6] R.W. White, R.A. Roth, Exploratory search: beyond the query-response paradigm, Synth. Lect. Inf. Concept. Retr. Serv. 1 (1) (2009) 1–98.
[7] G. Marchionini, Exploratory search: from finding to understanding, Commun. ACM 49 (4) (2006) 41–46.
[8] H. Kagdi, M.L. Collard, J.I. Maletic, A survey and taxonomy of approaches for mining software repositories in the context of software evolution, J. Softw. Maint. Evol. Res. Pract. 19 (2) (2007) 77–131.
[9] J. Sinclair, M. Cardew-Hall, The folksonomy tag cloud: when is it useful? J. Inf. Sci. 34 (1) (2008) 15–29.
[10] Linux github repository, (https://github.com/torvalds/linux).
[11] C.S. Mesnage, M.J. Carman, Tag navigation, in: Proceedings of the SoSEA 2nd International Workshop on Social Software Engineering and Applications, ACM, 2009, pp. 29–32.

[12] B. Ganter, R. Wille, Formal Concept Analysis - Mathematical Foundations, Springer, Berlin, 1999.
[13] B.A. Davey, H.A. Priestley, Introduction to Lattices and Order, 2nd. ed., Cambridge University Press, Cambridge, 2002.
[14] B. Fischer, Specification-based browsing of software component libraries, Autom. Softw. Eng. (ASE) 7 (2) (2000) 179–200.
[15] C. Lindig, Concept-based component retrieval, in: Proceedings of IJCAI, 1995, pp. 21–25.
[16] C. Carpineto, G. Romano, A lattice conceptual clustering system and its application to browsing retrieval, Mach. Learn. 24 (2) (1996) 95–122.
[17] G.J. Greene, B. Fischer, Interactive tag cloud visualization of software version control repositories, in: Proceedings of the IEEE 3rd Working Conference on Software Visualization (VISSOFT), IEEE, 2015, pp. 56–65.
[18] A. Hindle, D.M. Germán, SCQL: a formal model and a query language for source control repositories, ACM SIGSOFT Softw. Eng. Notes 30 (4) (2005) 1–5.
[19] C.M. Pilato, B. Collins-Sussman, B.W. Fitzpatrick, Version Control with Subversion - the Standard in Open Source Version Control, O'Reilly Media, Inc, Sebastopol, California, 2008.
[20] J. Vesperman, Essential CVS, O'Reilly Media, Inc., Sebastopol, California, 2006.
[21] C. Lindig, Fast concept analysis, Work. Concept. Struct.Contrib. ICCS 2000 (2000) 152–161.
[22] M.F. Porter, An algorithm for suffix stripping, Prog. Electron. Lib. Inf. Syst. 14 (3) (1980) 130–137.
[23] R. Navigli, Word sensedisambiguation: a survey, ACM Comput. Surv. 41 (2) (2009) 10:1–10:69.
[24] G. Robles, J.M. Gonzalez-Barahona, Developer identification methods for integrated data from various sources, SIGSOFT Softw. Eng. Notes 30 (4) (2005) 1–5.
[25] G.C. Murphy, D. Notkin, Lightweight lexical source model extraction, ACM Trans. Softw. Eng. Methodolol. 5 (3) (1996) 262–292.
[26] X. Ren, F. Shah, F. Tip, B.G. Ryder, O. Chesley, Chianti: a tool for change impact analysis of java programs, in: Proceedings of the Object-Oriented Programming, Systems, Languages and Applications, OOPSLA, 2004, pp. 432–448.
[27] Github, (http://github.com).
[28] A. Begel, Y.P. Khoo, T. Zimmermann, Codebook: discovering and exploiting relationships in software repositories, in: Proceedings of the International Conference on Software Engineering (ICSE), 2010, pp. 125–134.
[29] J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes? in: Proceedings of the International Workshop on Mining Software Repositories (MSR), ACM, 2005, pp. 1–5.
[30] C. Van Rijsbergen, Information Retrieval.
[31] C. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval
[32] G.J. Greene, B. Fischer, Conceptcloud: a tagcloud browser for software archives, in: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2014, pp. 759–762.
[33] J. Loeliger, M. McCullough, Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development, O'Reilly Media, Inc., Sebastopol, California, 2012.
[34] D.N. Götzmann, Colibri/java, 2007, (http://code.google.com/p/colibri-java/).
[35] C. Anslow, S. Marshall, J. Noble, R. Biddle, Sourcevis: collaborative software visualization for co-located environments, in: Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT), 2013, pp. 1–10.
[36] S. Lohmann, J. Ziegler, L. Tetzlaff, Comparison of tag cloud layouts: task-related performance and visual exploration, in: Proceedings of the International Conference on Human-Computer Interaction (INTERACT), 2009, pp. 392–404.
[37] J. Schrammel, M. Leitner, M. Tscheligi, Semantically structured tag clouds: an empirical evaluation of clustered presentation approaches, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2009, pp. 2037–2040.
[38] L.D. Caro, K.S. Candan, M.L. Sapino, Navigating within news collections using tag-flakes, J. Vis. Lang. Comput. 22 (2) (2011) 120–139.
[39] M.J. Zaki, M. Ogihara, Theoretical foundations of association rules, in: Proceedings of the 3rd ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, 1998.
[40] P. Weissgerber, M. Pohl, M. Burch, Visual data mining in software archives to detect how developers work together, in: Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR), 2007, pp. 9–17.
[41] S. Thummalapenta, T. Xie, Spotweb: detecting framework hotspots and coldspots via mining open source code on the web, in: Proceedings of the International Conference on Automated Software Engineering (ASE), 2008, pp. 327–336.
[42] Rubygems, (https://github.com/rubygems/rubygems).
[43] M. Goeminne, T. Mens, A comparison of identity merge algorithms for software repositories, Sci. Comput. Program. 78 (8) (2013) 971–986.
[44] Bus factor, (http://deviq.com/bus-factor/).
[45] Backbone, (https://github.com/jashkenas/backbone).
[46] Retrofit, (https://github.com/square/retrofit).
[47] Gitk, (https://git-scm.com/docs/gitk).
[48] S.S. Shapiro, M.B. Wilk, An analysis of variance test for normality (complete samples), Biometrika 52 (3/4) (1965) 591–611.
[49] J.W. Tukey, Comparingindividual means in the analysis of variance, Biometrics 5 (2) (1949) 99–114.
[50] T. Girba, A. Kuhn, M. Seeberger, S. Ducasse, How developers drive software evolution, in: Proceedings of the International Workshop on Principles of Software Evolution, 2005, pp. 113–122.
[51] O. Alonso, P.T. Devanbu, M. Gertz, Expertise identification and visualization from cvs, in: Proceedings of the International Working Conference on Mining Software Repositories (MSR), 2008, pp. 125–128.
[52] A. Zaidman, B. Van Rompaey, S. Demeyer, A. Van Deursen, Mining software repositories to study co-evolution of production and test code, in: Proceedings of the International Conference on Software Testing, Verification, and Validation, IEEE, 2008, pp. 220–229.
[53] T. Fritz, G.C. Murphy, Using information fragments to answer the questions developers ask, in: Proceedings of the International Conference on Software Engineering (ICSE), ACM, 2010, pp. 175–184.
[54] R. Cottrell, B. Goyette, R. Holmes, R. Walker, J. Denzinger, Compare and contrast: visual exploration of source code examples, in: Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), 2009, pp. 29–32.
[55] C. Anslow, J. Noble, S. Marshall, E. Tempero, Visualizing the word structure of java class names, in: Proceedings of the Object-Oriented Programming Systems Languages and Applications (OOPSLA), 2008, pp. 777–778.
[56] J. Emerson, N. Churcher, C. Deaker, From toy to tool: extending tag clouds for software and information visualisation, in: Proceedings of the Australian Software Engineering Conference, 2013, pp. 155–164.
[57] J. Gwizdka, P. Bakelaar, Tag trails: navigation with context and history, in: Proceedings of the CHI'09 Extended Abstracts on Human Factors in Computing Systems, ACM, 2009, pp. 4579–4584.
[58] M.-E. Hernandez, S.M. Falconer, M.-A. Storey, S. Carini, I. Sim, Synchronized tag clouds for exploring semi-structured clinical trial data, in: Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds (CASCON), ACM, 2008, pp. 4:42–4:56.
[59] D. Poshyvanyk, A. Marcus, Combining formal concept analysis with information retrieval for concept location in source code, in: Proceedings of the International Conference on Program Comprehension (ICPC), 2007, pp. 37–48.
[60] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, R. Daniel, Using concept analysis to detect co-change patterns, in: Proceedings of the IWPSE Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting, 2007, pp. 83–89.
[61] G. Snelting, Reengineering of configurations based on mathematical concept analysis, ACM Trans. Softw. Eng. Methodol. 5 (2) (1996) 146–189.
[62] G. Snelting, F. Tip, Reengineering class hierarchies using concept analysis, SIGSOFT Softw. Eng. Notes 23 (6) (1998) 99–110.
[63] G.J. Greene, A generic framework for concept-based exploration of semi-structured software engineering data, in: Proceedings of the Automated Software Engineering (ASE), IEEE, 2015, pp. 894–897.

Please cite this article as: G.J. Greene et al., Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices, Information and Software Technology (2016), http://dx.doi.org/10.1016/j.infsof.2016.12.001