2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

Comparing Repositories Visually with RepoGrams

Daniel Rozenberg* Ivan Beschastnikh* Fabian Kosmale+ Valerie Poser+
Heiko Becker+ Marc Palyart* Gail C. Murphy*
*University of British Columbia
Vancouver, BC, Canada
{rodaniel, bestchai, mpalyart, murphy}@cs.ubc.ca
+Saarland University
Saarbrücken, Germany
{s9kofabi, s9vapose, s9hhbeck}@stud.uni-saarland.de

ABSTRACT

The availability of open source software projects has created an enormous opportunity for software engineering research. However, this availability requires that researchers judiciously select an appropriate set of evaluation targets and properly document this rationale. After all, the choice of targets may have a significant effect on evaluation.

We developed a tool called RepoGrams to support researchers in qualitatively comparing and contrasting software projects over time using a set of software metrics. RepoGrams uses an extensible, metrics-based, visualization model that can be adapted to a variety of analyses. Through a user study of 14 software engineering researchers we found that RepoGrams can assist researchers in filtering candidate software projects and make more reasoned choices of targets for their evaluations. The tool is open source and is available online: http://repograms.net/

1. INTRODUCTION

Software engineering (SE) researchers are increasingly using open source project information stored in centralized hosting sites, such as GitHub and Bitbucket, to help evaluate their ideas. For example, GitHub hosts tens of millions of projects, has about 9 million registered users and is one of the top 100 most popular sites [29]; all of these projects are potential targets to use in an evaluation.

Some SE studies target hundreds or thousands of projects in their evaluations [36, 11]. For these studies, projects can be selected solely on the basis of meta-data and simple metrics, such as the programming language used. A variety of tools exist to help researchers with selecting projects in this manner. For example, GHTorrent [21] is a database of meta-data of GitHub repositories that can be queried for large-scale project selection. Another example is Boa [17], a database and query language designed to support researchers in finding projects that match certain criteria.

However, many SE studies use just a handful of evaluation targets. In our reading of 114 papers from six major SE conferences we found that 84 of these papers performed an empirical evaluation on some artifacts of software projects and 63 of these 84 papers (75%) used 8 or fewer evaluation targets. For such studies, tools like GHTorrent and Boa are only useful in identifying an initial set of targets. Filtering the initial set down to a final selection of projects requires additional analysis; analysis that is not supported by existing tools.

MSR'16, May 14-15 2016, Austin, TX, USA

ISBN 978-1-4503-4186-8/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2901739.2901768

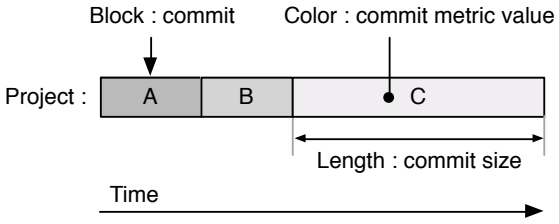

Figure 1: Repository footprint visual abstraction.

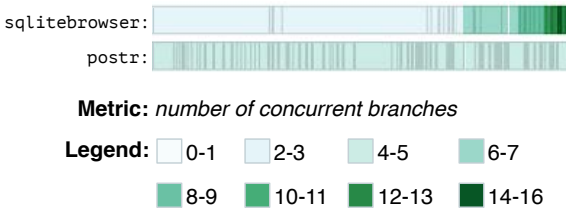

Figure 2: Two example repository footprints.

We propose RepoGrams, a new tool that helps support SE researchers to *qualitatively* study and compare several software project repositories. RepoGrams juxtaposes the evolution of several projects to support SE researchers in exploring potential evaluation targets. RepoGrams is built on an extensible, metrics-based, visualization model that can be adapted to a variety of analyses. RepoGrams displays a *repository footprint* abstraction for a repository (Figure 1), which represents the commits in a project repository as a sequence of blocks, one block per commit. The blocks are colored according to a user-defined metric: a block's color represents the corresponding commit's metric value. A block's length represents the corresponding commit's size (LoC changed). RepoGrams supports researchers in studying metrics related to project evolution and presents multiple metrics across multiple projects to facilitate comparison of projects.

For example, Figure 2 shows two footprints: a `sqlitebrowser` [4] footprint (top) and a `postr` [5] footprint (bottom). In this example the metric is the *count of the number of concurrent branches active when a commit was introduced*. From this figure we can make two observations: (1) `sqlitebrowser`'s footprint contains commits that are significantly larger than those in `postr`. We can see

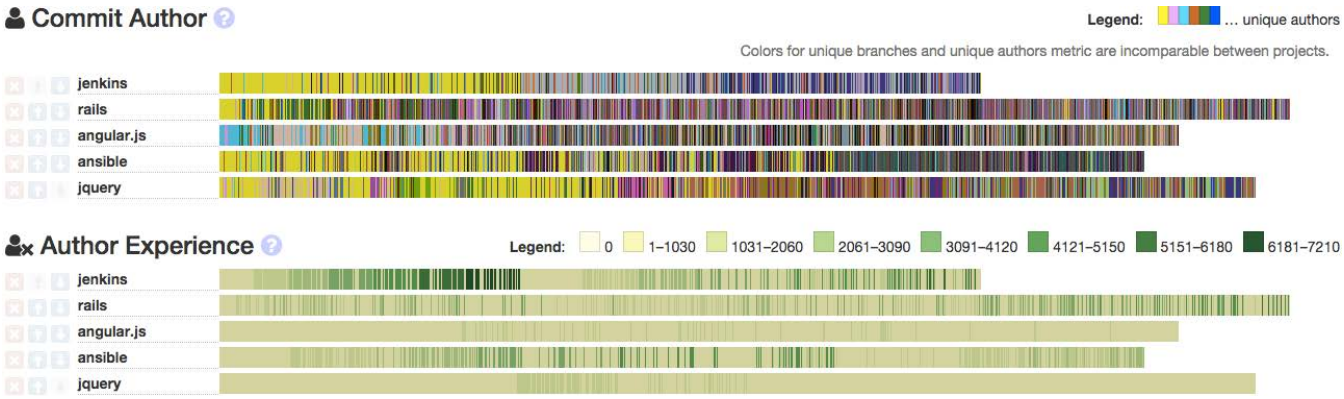

Figure 3: Five repositories used in a study on the impact of developer turnover on software quality, visualized with RepoGrams using its commit author metric (top) and author experience metric (bottom).

this by comparing the length of the commit blocks between the two footprints. (2) in contrast to `postr`, `sqlitebrowser` progressively uses more concurrent branches over time. We can see this by observing the footprints left to right: the footprint for `sqlitebrowser` becomes darker and it eventually has as many as 14-16 concurrent branches. In contrast, the footprint for `postr` does not change its color. Understanding these differences between the projects can help a researcher choose appropriate repositories to demonstrate the range of their approach; for instance, these two projects exhibit opposite patterns of branching use that may help demonstrate generalizability of a tool.

To evaluate RepoGrams we performed a user study and a case study. We conducted a user study with 14 active SE researchers from the MSR community and determined that they can use RepoGrams to understand and compare characteristics of a project's source repository. We also evaluated the effort involved in adding six new metrics to RepoGrams in a case study with two different individuals.

In summary, we make the following contributions:

* We designed and implemented RepoGrams, a tool to juxtapose the evolution of multiple projects. RepoGrams supports SE researchers in exploring the space of possible evaluation targets and is built on an extensible, metrics-based, visualization model that can be adapted to a variety of analyses.

* We evaluated RepoGrams in a user study with active SE researchers and determined that they can use RepoGrams to understand and compare characteristics of a project's source repository. We also report on the effort involved in extending RepoGrams with new metrics.

2. MOTIVATING EXAMPLES

To concretely describe how RepoGrams may be used, we describe the application of the tool to two examples drawn from previously published papers [18, 23]. We describe how RepoGrams can help to understand the diversity of the evaluation targets and how RepoGrams can be used to form hypothesis about target evaluation projects.

Developer turn-over in open-source projects.

Foucault et al. [18] studied developer turnover in open source projects, with a focus on the impact of developer turnover on software quality. The authors performed an in-depth quantitative study of five large projects, finding that the activity of developers joining the project negatively impacts software quality.

Figure 3 visualizes the RepoGrams repository footprints for the five projects used by Foucault et al. using two metrics: 1) the commit author metric that assigns a unique color to each committer of the project and 2) the author experience metric that indicates at the time of commit how many commits its author has already performed in this project. In both cases, for simplicity, we fixed the block length used in RepoGrams to a constant; as a result, commit size is not illustrated (this feature and the available metrics are detailed in the next section).

Using the RepoGrams footprint for Commit Author, one can see a diversity of patterns in the footprint, although all projects accumulate contributors over time. For example, `Angular.JS` [2] has a few dominant contributors at the start of the project, but the project quickly gains new committers. On the other hand, `Jenkins` [1] contains a single dominant contributor at the start (color yellow), who is then replaced by another dominant contributor (color grey), and later a third dominant contributor (color purple).

Although using the commit author metric on such large projects yields a complex picture, one can see from the figure that `Jenkins`, `Ansible` [3], and `jQuery` [6] had one dominant contributor at the beginning. This information could have been useful to the researchers (two of whom are authors on this paper) since for the study they had to select releases with a diversity of contributors. Since part of the study involved manually analyzing bugs associated with the releases studied, which was time-consuming, there was a desire to select the appropriate releases up-front. RepoGrams could have saved the researchers time by helping them to focus on sections of the project timeline that had the most commit author diversity.

Considering the same set of repositories through the author experience metric we can see that all five projects have a variety of patterns according to this metric. For example, the initial dominant contributor in `Jenkins` is much more dominant than contributors across all other projects (in terms of number of commits). By comparison, `Angular.JS` and `jQuery` have a much more evenly distributed author experience over time, while `Ansible` has several pronounced but distributed metric value peaks throughout its history.

The two RepoGrams footprints in Figure 3 demonstrate that the projects analyzed in the original study had diverse patterns of developer activity and experience. The authors could have used these footprints to help support the generality of their findings.

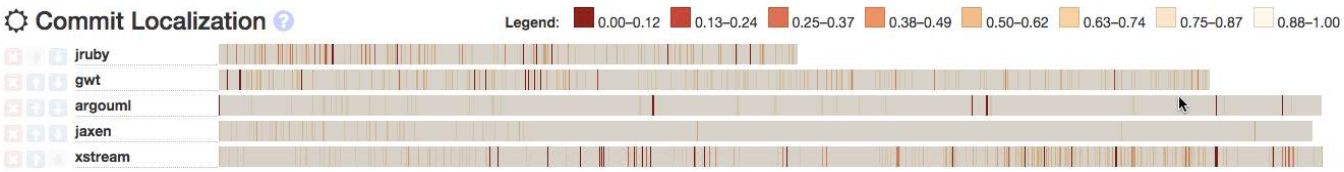

Figure 4: Three repositories used in a study of tangled code, visualized with RepoGrams using its commit localization metric.

Tangled code changes.

The role of tangled code changes and their impact on quality were studied by Herzig and Zeller [23]. Figure 4 illustrates the set of evaluation targets used in this study using the RepoGrams' commit localization metric. This metric computes the fraction of the number of unique project directories containing files modified by the commit. This value ranges from 0 (all the project directories contain a file modified by the commit) to 1 (all modified files in a commit are in a single directory). This metric captures commit modularity and can be used to identify cross-cutting commits or commits that mix multiple tasks (i.e., tangled code changes). As before, we fixed the block length in RepoGrams, so commit size is not illustrated.

From Figure 4, we can see that the `XStream` project (shown at the bottom of the footprint) has a continuous stream of outlier (tangled) commits with a low commit localization metric value. By contrast, the `Jaxen` project has almost no commits with a low metric value.

The authors in the study manually inspected over 7,000 change-sets across these five projects and manually determined which of these were associated with an issue and whether the change-set was tangled or not. Using this process the authors determined that according to the change-sets they studied, the projects had the following fraction of tangled changes (ordered from least to most): `ArgoUML` (5.8%), `Jaxen` (8.1%), `GWT` (8.4%), `jRuby` (9.3%), `XStream` (11.9%). Considering the footprint in Figure 4, one can make a similar assessment from the diagram. The RepoGrams footprint cannot be used to derive an exact fraction, but one can get a general sense for which projects have any tangled changes and how two projects compare in the number of tangled changes they contain.

Summary.

Both examples illustrate how RepoGrams may be used to evaluate project repositories using metrics that closely match features relevant to the study. Based on this information, a researcher may decide to add another project to improve the diversity of RepoGrams footprints, or this may help the researcher argue that the evaluation targets they have selected generalize to projects with similar RepoGrams footprints.

3. REPOGRAMS DESIGN

We designed RepoGrams to compare and contrast project repositories over time to help SE researchers in qualitatively filtering a set of potential evaluation targets. RepoGrams is qualitative in the sense that it presents data in a way that can be observed but not measured. For example, researchers can use RepoGrams to identify and compare visual patterns in the sequence of colors and the lengths of the commit blocks.

RepoGrams has three key features. First, RepoGrams captures project repository activity over time. Second, RepoGrams is designed to present multiple metrics side-by-side to help characterize informal qualities or artifacts relating to the software development activity in a project overall. Third, to support researchers in project selection RepoGrams supports comparison of metrics for about a dozen projects.

This last point is supported by a literature survey in which four of the authors considered a random set of 114 research track papers from five SE conference proceedings in 2012–2014. The five conferences were ICSE, FSE, ASE, MSR, and ESEM. For each paper we considered the number of distinct evaluation targets used in the paper's evaluation. We recorded the number of targets that the authors *claim* to evaluate as some targets can be considered to be a single project or multiple projects. For example, Android is an operating system with many sub-projects: one paper can evaluate Android as a single target, while another paper can evaluate Android's many sub-projects. We found that 84 papers (74%) performed an empirical evaluation on some artifacts of software projects, and 63 of these 84 papers (75%) evaluated their work with 8 or fewer evaluation targets. Evaluations with large datasets are uncommon.

The rest of this section details RepoGrams' design and implementation.

3.1 Overview

RepoGrams is a client-server web application. Figure 5 shows a screenshot of a session with three projects and two metrics.

Workflow.

To use the tool the user starts by importing some number of project repositories. She does this by either adding a particular candidate projects' Git repository URL to RepoGrams or by adding a random GitHub repository (① in Figure 5). The server clones each added repository and computes metric values for each commit in the repository. Next, the user selects one or more metrics (② in Figure 5) that are relevant to her research. This causes the server to transmit the pre-computed metric values to the client for display. The metric values are assigned to colors and the interface presents the computed project repository footprints to the user (③ in Figure 5) along with the legend for each metric (④ in Figure 5).

Repository footprint.

RepoGrams visualizes a set of metrics over the commits of a set of repositories as a continuous horizontal line that we call a *repository footprint*, or footprint for short (Figure 5 shows six repository footprints). The footprints are displayed in a stack to facilitate comparison between projects/metrics. A footprint is composed of a sequence of *commit blocks*, each of which represents one commit in a repository. RepoGrams serializes the commits across all branches of a repository into a footprint using the commits' timestamps. The metric value computed for a commit determines the block's color.

Block width.

The width of a each commit block can be either a constant value, a linear representation of the number of LoC changed in the commit, or a logarithmic representation of the same. We also support two **normalization** variants: project normalized and globally normalized. In a *project normalized* block width all widths

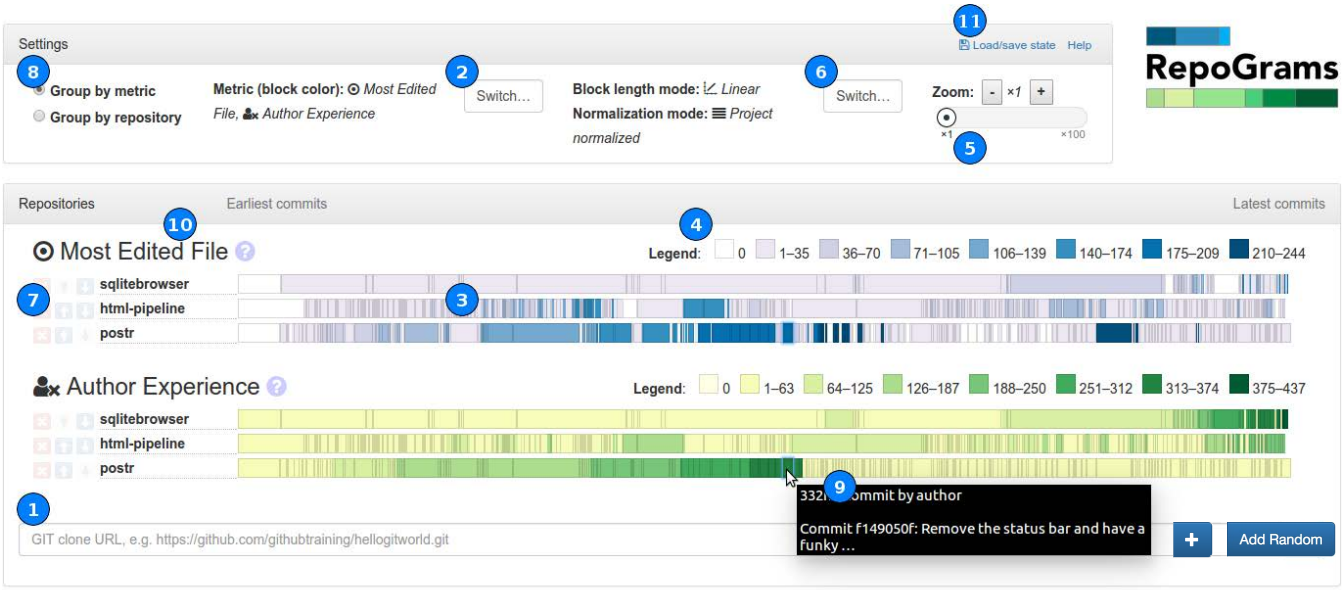

Figure 5: The RepoGrams interface and its key features: (1) input field to add a repository into view (with an option to add a random repository from GitHub), (2) button to select the metric(s), (3) a *repository footprint* corresponding to a specific project/metric combination. The color of a *commit block* represents the value of the metric on that commit, (4) the legend for commit values in the selected metric(s), (5) zoom control, (6) button to change the block length representation, (7) buttons to remove or change the order of repository footprints, (8) control to switch between grouping by metric/project (see Figure 7), (9) tool-tip with an exact metric value and the commit message (truncated), (10) metric name and description, (11) import/export functionality and help documentation.

are normalized per project to utilize the full width available in the browser window. This mode prevents meaningful comparison between projects if the user is interested in contrasting absolute commit sizes. The footprints in Figure 5 use this mode. With the second normalization mode, *globally normalized*, block widths are resized to be relatively comparable across projects. Figure 6 illustrates the six possible block widths.

Block color.

A commit block's color is determined by a mapping function that is implemented as part of a metric. RepoGrams currently supports three color mapping variants: *fixed buckets*, *uniform buckets*, and *discrete buckets* (see Section 3.2.1). Each of these functions maps a commit's value in a metric to one of several metric value *buckets*, and each of these buckets corresponds to a color for the corresponding commit block.

The interface shows a color legend next to each selected metric (④ in Figure 5). For example, the second metric in Figure 5 is *Author Experience*. Using this example we can see that the most experienced author in the `sqlitebrowser` repository committed 375–437 commits in this repository, according to the latest commits in that project (right-most *commit blocks*). In contrast, no author committed over 250 commits in the *html-pipeline* repository, and no author committed over 374 commits in the *postr* project.

Supported interactions.

The RepoGrams interface supports a number of interactions. The user can (1) scroll the footprints left to right and zoom in and out to focus on either the entire timeline or a specific period in the projects' histories (⑤ in Figure 5), (2) change the block length mapping (⑥ in Figure 5), (3) remove a project or change the order in which the repository footprints are displayed (⑦ in Figure 5), (4) change the footprint grouping (⑧ in Figure 5) to group footprints by metric or by project (see Figure 7), and (5) hover over or click on an individual commit block in a footprint to see the commit metric value, commit message, and link to the commit's page on GitHub (⑨ in Figure 5). Finally, RepoGrams supports an import/export functionality (⑪ in Figure 5), allowing users to store and share their sessions with others.

Figure 5 shows a metric-grouped view (Figure 7(a)) of three projects and two metrics. The most edited file metric (top three repository footprints) represents the maximum number of times that a file in a commit has been previously modified. This metric indicates that edits to files in `sqlitebrowser` are more spread across the system; there is no small set of frequently modified files. This is in contrast to `html-pipeline` and `postr`. The author experience metric (bottom three footprints in Figure 5) counts the number of commits that a commit's author has previously contributed to the repository. This metric shows that the `postr` project had an active committer with over 300 commits until about half-way through the footprint; this committer then suddenly stopped contributing and does not appear later in the history.

3.2 Metrics

As of this writing, RepoGrams has twelve built-in metrics, listed in Table 1. We selected these metrics using our team's past experiences with empirical evaluations, and based on suggestions of SE researchers in our user study (Section 4.1).

The type column in Table 1 lists the type of mapping function used to assign a color to a metric value (described later in this section). The LoC column in the table lists the lines of code involved in the server-side calculation of a metric. Client-side display logic and meta-data are not included in this count. The top six metrics in Table 1 were developed in the first iteration of our prototype.

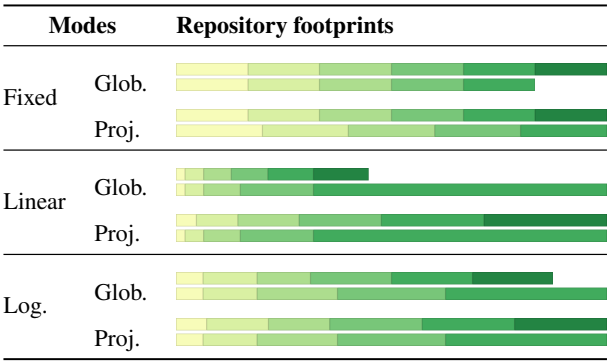

Figure 6: An illustration of six supported block widths using footprints for two different repositories with the same metric. The top repository has 6 commits with 1, 2, 3, 4, 5, and 6 LoC changed, in that order. The bottom repository has 5 commits with 1, 2, 4, 8, and 16 LoC changed, in that order.

The bottom six metrics were added by two developers to evaluate metric development time, the Dev. time column (see Section 4.2).

We briefly describe one example metric from each of the three information categories that we used to formulate our metrics:

Code metrics.

Information about *code* or artifacts inside the repository. For example, the *Commit localization* code metric computes the fraction of the number of unique project directories containing files modified by the commit. This value ranges from 0 (all the project directories contain a file modified by the commit) to 1 (all modified files in a commit are in a single directory). This metric captures commit modularity and can be used to identify cross-cutting commits or commits that mix multiple tasks [32].

Process metrics.

Information about the *process* followed by the project. For example, the *Commit message length* metric computes the number of words in a commit message. Commit messages are used to document changes in the repository and projects have varying rules on how to write commit messages (e.g., must reference an issue in a task tracker, must be at most 1 sentence, etc). This metric can expose trends in a project's documentation policy, and can be used to identify the commits that prompted a developer to write more detailed commit messages.

Social metrics.

Information about *contributors* to the project, or their relationships. For example, *Author experience* counts the number of commits that a commit's author has previously contributed to the repository. This metric can assist researchers who study code ownership (e.g., [9, 34]) as this metric can categorize projects based on the different patterns in the experience of their authors.

We note that the existing metrics are not intended to be comprehensive. RepoGrams' power lies in its extensibility model (as described in Section 4.2). Additionally, although RepoGrams relies on an underlying set of quantitative metrics, the way in which the results of these metrics are surfaced is qualitative.

3.2.1 Representing metric values as colors

We decided to use colors to represent metric values for two reasons: (1) RepoGrams is intended to support exploration and colors make it easy to spot patterns such as alternation and gradient, (2) colors provide efficient use of space due to their compact encoding. The difficulty with colors is that the human perceptual system can differentiate between just a handful of colors [30, chapter 10.3]. Our strategy for most metrics is to use a multi-hue progression of eight colors that use the same base color, and to map metric values to these eight colors by bucketing some set of metric values and associating each bucket with one color[1]. Our prototype currently supports three kinds of bucketing strategies:

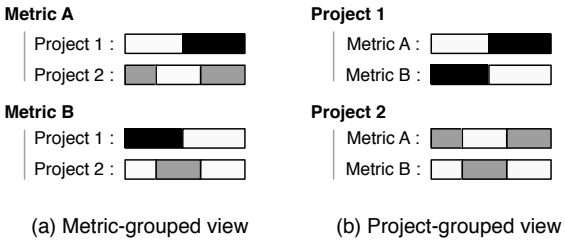

Figure 7: Two ways of grouping footprints: (a) the metric-grouped view facilitates comparison of different projects along the same metric, and (b) the project-grouped view facilitates comparison of different metrics for the same project.

Fixed buckets.

Metric values are bucketed into eight predefined buckets. For example, the *commit message length* metric has predefined buckets for commit messages with 0–1, 2–3, 4–5, 6–8, 9–13, 14–21, 22–34, and 35+ words. Each commit block is colored based on the color of the bucket corresponding to its commit message length. The colors for the buckets is a multi-hue progression (see the color legends for the two metrics, ④ in Figure 5).

Uniform buckets.

Metric values are bucketed into at most eight equally sized buckets to contain all metric values across all commits in the current view. For example, consider the *languages in a commit* metric, and a repository containing a commit that modified files in 18 different programming languages, and containing no commits that modify files in 19 or more languages. The uniform buckets for this metric will then look as follows: 0–2, 3–4, 5–6, 7–8, 9–11, 12–13, 14–15, and 16–18. As with fixed buckets, the colors assigned to these buckets is a multi-hue progression.

Discrete buckets.

A single metric value is assigned to one bucket. For example, the *commit author* metric assigns each author to a unique color (up to 728 authors).

The fixed and uniform buckets and the associated colors are designed to be appropriate for color-blind users. However, the discrete buckets, which encode the commit author and branches used metrics, are not accessible to the color-blind, as these may encode a very large number of distinct colors.

Next, we describe our evaluation of RepoGrams.

4. EVALUATION

We developed RepoGrams to help SE researchers answer complex questions about repository histories and to help them filter and

[1]In our implementation we used the ColorBrewer2 web tool [10] to identify the colors for each metric.

	Name	Info. category	Description	Color buckets	LoC	Dev. time
Initial set of metrics	Commit localization	Code	The fraction of the number of unique project directories containing files modified by the commit.	Fixed	13	—
	Most edited file	Code	The maximum number of times that a file in a commit has been previously modified.	Uniform	11	—
	Languages in a commit	Code	The number of unique programming languages used in a commit based on filenames.	Uniform	15	—
	Branches used	Process	Associate each branch with a unique value. A commit is assigned the value of the branch it is associated with.	Discrete	5	—
	Number of branches	Process	The number of branches that are concurrently active at a commit point.	Uniform	47	—
	Commit message length	Social	The number of words in a commit's log message.	Fixed	6	—
Added by Dev1	POM files	Code	The number of POM files (project configuration files used by Maven [8]) changed in the commit.	Uniform	6	30 min
	Commit age	Code	The elapsed time between a commit and its parent commit.	Fixed	7	48 min
	Commit author	Social	Associate each author with a unique value. A commit is assigned the value of its author.	Discrete	34	52 min
Added by Dev2	Files modified	Code	The number of files modified in a commit, including new and deleted files.	Fixed	6	42 min
	Merge indicator	Process	The number of parent commits from which the commit is derived ($\geq$2 parents indicates a merge).	Uniform	5	44 min
	Author experience	Social	The number of commits that the commit's author has previously contributed to the repository.	Uniform	8	26 min

Table 1: Metrics in the current RepoGrams prototype.

choose evaluation targets. To determine to what degree RepoGrams serves this purpose, we posed the following research questions:

- **RQ1**: Can SE researchers use RepoGrams to *understand* characteristics of a project's source repository?
- **RQ2**: Can SE researchers use RepoGrams to *compare* characteristics of a project's source repository?
- **RQ3**: Will SE researchers consider using RepoGrams to select evaluation targets for experiments and case studies?

We investigated these questions through a user study with 14 SE researchers in which each researcher used RepoGrams to answer questions about repository footprints (Section 4.1).

One issue raised by SE researchers in our user study was the need to define custom metrics. We therefore posed and investigated one more research question:

- **RQ4**: How much effort is required to add metrics to RepoGrams?

To answer RQ4 we conducted a case study with two individuals each of whom implemented three new metrics (Section 4.2).

Our evaluation methodology and results are further detailed on a companion web-site [37].

4.1 Study with SE researchers

To investigate RQ1–RQ3, we performed a user study with researchers from the SE community. This study had two parts: first, participants used RepoGrams to answer a series of questions about individual projects and comparisons between projects (using preselected footprints); second, participants were interviewed about RepoGrams. For this study we recruited participants from a subset of authors from the MSR 2014 conference, as these authors likely have experience with repository information. Some of the authors forwarded the invitation to their students who we included in the study.

The study had 14 participants: 5 faculty, 1 post doc, 6 PhD students, and 2 masters students. Participants were affiliated with institutions from North America, South America, and Europe. All participants have had research experience in analyzing the evolution of software projects and/or evaluating tools using artifacts from software projects.

We raffled off a gift card to incentivize participation. The study was performed in one-on-one sessions with each participant: 5 participants were co-located with the investigator and 9 sessions were performed over video chat.

4.1.1 Methodology

A session in the study began with a short demonstration of RepoGrams and gathering of demographic information. A participant then worked through nine questions presented through a web-based questionnaire.

The first three questions were aimed at helping a participant understand the user interface and various metrics (5 min limit total). This ensured that each participant gained a similar level of experience with the tool. The remaining six questions tested whether a participant could use RepoGrams to *understand* and *compare* a variety of project source repository characteristics (RQ1 and RQ2). Questions in this section were of the form *"group the repositories into two sets based on a characteristic"*, where the characteristic was implied by the chosen metric (3–7 min limit per question). Ta-

#	Question	# repository footprints	Answer distribution
4	Which of the following statements is true? *There is a general {`upwards / constant / downwards`} trend to the metric values.*	1	
5	Categorize the projects into two clusters: (a) projects that use Maven (include .pom files), (b) projects that do not use Maven.	9	
6	Categorize the projects into two clusters: (a) projects that used a single master branch before branching off to multiple branches, (b) projects that branched off early in their development.	5	
7	Categorize the projects into two clusters: (a) projects that have a correlation between branches and authors, (b) projects that do not exhibit this correlation.	8	
8	Categorize the projects into two clusters: (a) projects that have one dominant contributor, based on number of **lines of code changed**, (b) projects that do not have such a contributor. A dominant contributor is one who committed at least 50% of the LoC changes in the project.	3	
9	*Same as 8, with number of **commits** instead of number of **lines of code changed**.*	3	

Table 2: Main set of questions from the user-study with SE researchers. The last column summarizes the answers with a graphic. Each participant's answer is represented by a block; blocks of the same color denote identical answers.

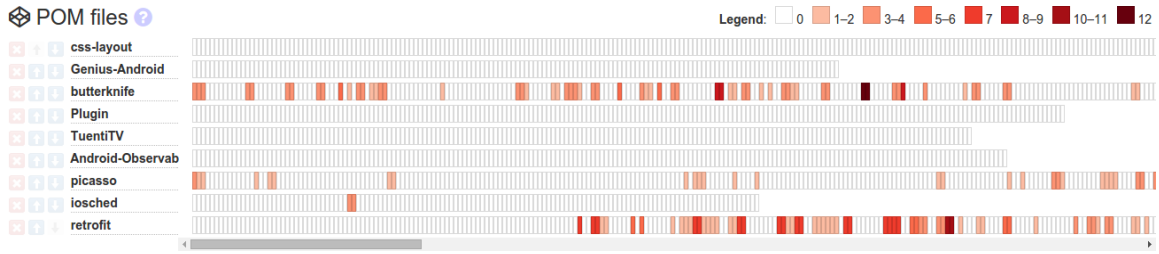

Figure 8: Repository footprints used in question 5.

ble 2 details these questions.

We then interviewed each participant in a semi-structured interview to investigate RQ3 (Section 4.1.3).

For the study we chose the top 25 trending projects (pulled on February 3rd, 2015) for each of the ten most popular languages on GitHub [45]. From this set we systematically generated random permutations of 1–9 projects for each question until we found a set of projects whose repository footprints fit the intended purpose of the questions. One author came up with ground truth for the questions based on the selected projects. The other authors then independently answered the questions and verified that their answers matched the ground truth. The final set of project repositories in the study had a min / median / max commit counts of 128 / 216 / 906, respectively.

4.1.2 Study results

To give an overall sense of whether SE researchers were in agreement about the answers posed, we use a graphic to capture the distribution of answers in the right-most column of Table 2. In this graphic, each participant's answer is represented by a block; blocks of the same color denote identical answers. For example, for question 6, twelve participants chose one answer and two participants chose a different answer each; a total of three distinct answers to that question. We now overview the six questions and elaborate on a few of them.

Question 4 asked the participants to recognize a trend in the metric value in a single repository. The majority of participants (12 of 14) managed to recognize the trend almost immediately by observing the visualization.

Question 5 asked the participants to identify repositories that have a non-zero value in one metric. The participants considered 9 repository footprints (Figure 8) with the *POM files* metric[2]: a metric value of n indicates that n POM files were modified in a commit. This metric is useful for quickly identifying projects that use the Maven build system and commits in those projects that change the Maven configuration. All except one participant agreed on the choice of nine repositories. This question indicates that RepoGrams is useful in distinguishing whether a software project exhibits a characteristic over time. Using this metric in combination with other metrics an SE researchers can filter out irrelevant projects that do not exhibit a characteristic that they care about in a potential evaluation target.

The right-most column of Table 2 shows widespread agreement amongst the researchers for questions 4 and 5. These questions are largely related to interpreting metrics for a project. This quantitative agreement lends support to RQ1. More variance in the answers resulted from the remaining questions that target RQ2; these questions required more advanced interpretation of metrics and inter-project comparison.

Question 6 asked the participants to identify those repositories in which the repository footprints started with a sequence of commit blocks of a particular color. The participants considered 5 repository footprints. The metric was *branches used*: each branch is given a unique color, with a specific color reserved for commits to the master branch. All five footprints contained hundreds of colors.

The existence of a leading sequence of commit blocks of a single

[2]POM stands for "Project Object Model". It is an XML representation of a Maven [8] project held in a file named `pom.xml`.

color in a *branches used* metric footprint indicates that the project used a single branch at the start of its timeline or the project was imported from a centralized version control system to Git. All participants agreed on three of the footprints and all but two agreed on the remaining two footprints. This indicates that RepoGrams is useful in finding long sequences of colors, even within footprints that contain hundreds of colors.

Question 7 asked the participants to identify those repositories in which the repository footprints for two metrics contained a correspondence between the colors of the same commit block. The participants considered a total of 8 repository footprints (see Figure 9), with two metrics for four project. The two metrics were *commit author* and *branches used*. The commit author metric assigns all commits by the same author a unique color. The branches used metric was described in question 6 above.

A match in colors between the two metrics in this question would indicate that committers in the project follow the practice of having one branch per author. This type of information can be useful in choosing evaluation targets for a research project that studies code ownership or the impact of committer diversity on feature development, such as [9].

For this question the number of colors in a pair of footprints for a repository ranged from a few (<10) to many (>20). The majority of participants (12 of 14) agreed on a choice of first, second, and fourth footprint pairs. But, they were about evenly split (8 vs. 6) on the third pair of footprints for the `stackit` project (see Figure 9). In fact, some sections of `stackit`'s repository history do exhibit this correlation, but not the entire history.

Questions 8 and 9 asked the participants to estimate the magnitude of non-continuous regions of discrete values. The participants were split in their answers, though more than half of the participants answered identically.

4.1.3 Semi-structured interview

IQ1: do you see RepoGrams being integrated into your research/evaluation process? If so, can you give an example of a research project in which you could use RepoGrams?

Of the 14 participants 11 noted that they want to use RepoGrams in their future research: *"I would use the tool to verify or at least to get some data on my selected projects"* [P12][3] and *"I would use RepoGrams as an exploratory tool to see the characteristics of projects that I want to choose"* [P9]. They also shared past research projects in which RepoGrams could have assisted them in making a more informed decision while choosing or analyzing evaluation targets. The remaining 3 participants said that they do not see themselves using RepoGrams in their research but that either their students or colleagues might benefit from the tool.

IQ2 what are one or two metrics that you wish RepoGrams included that you would find useful in your research? How much time would you be willing to invest to implement these new metrics?

Most participants found the existing metrics useful: *"Sometimes I'm looking for active projects that change a lot, so these metrics [e.g., Commit age] are very useful"* [P8]. However, they all suggested new metrics and mentioned that they would invest 1 hour to 1 week to add their proposed metric to RepoGrams. In Section 4.2 we detail a case study in which we add three of the proposed metrics to RepoGrams and show that this takes less than an hour per metric.

The participants also found that RepoGrams helped them to identify general historical patterns and to compare projects: *"I can use RepoGrams to find general trends in projects"* [P3] and *"You can find similarities . . . it gives a nice overview for cross-projects comparisons"* [P13]. They also noted that RepoGrams would help them make stronger claims in their work: *"I think this tool would be useful if we wanted to claim generalizability of the results"* [P4].

IQ3: what are the best and worst parts of RepoGrams?

One of our design goals was to support qualitative analysis of software repositories. However, multiple participants noted that the tool would be more useful if if exposed statistical information: *"It would help if I had numeric summaries."* and *"When I ask an exact numeric question this tool is terrible for that. For aggregate summaries it's not good enough"* [P6]

Another noted design limitation is the set temporal ordering of commits in the repository footprint abstraction: *"Sometimes I would like to order the commits by values, not by time"* [P7] and *"I would like to be able to remove the merge commits from the visualizations."* [P14]. Related to this, a few participants noted the limitation that RepoGrams does not capture real time in the sequence of commit blocks: *"the interface doesn't expose how much time has passed between commits, only their order."* [P7]

In response to what worked well, participants noted that *"this tool would give us a nice overview in terms of developer contributions"* (P4), and thought it would be useful for inspecting project activity: *"Sometimes I'm looking for active projects that change a lot, so these metrics [e.g., Commit age] are very useful"* (P8). Finally, several people thought that the tool could be useful for discerning representative project groups: *"there's potential there to try to visualize project to see if they're better representative of some set of different groups"* (P1).

IQ4: how would you approach solving one or more of the main tasks without using RepoGrams?

The participants proposed two general approaches as alternatives to using RepoGrams. The most common proposed approach was to write a custom script that clones the repositories and performs the analysis.

Other participants proposed to import the meta-data of the repositories into a spreadsheet and perform a manual analysis. A few participants mentioned that GitHub exposes several repository visualizations, such as a histogram of contributors to a repository. However, these visualizations are per-repository, do not facilitate comparison, and cannot be extended with new metrics.

4.1.4 Summary

This study shows that SE researchers can use RepoGrams to understand characteristics about a project's source repository (RQ1) and that they can, in a number of cases, use RepoGrams to compare repositories (RQ2), although the researchers noted areas for improvement. Through interviews, we determined that RepoGrams is of immediate use to today's researchers (RQ3) and that researchers were interested in specialized metrics.

4.2 Effort in adding new metrics

The SE researchers who participated in the first study had a strong interest in new metrics. Because researchers tend to have unique projects that they are interested in evaluating, it is likely that this interest is true of the broader SE community, as well. Thus, we evaluated the effort in adding new metrics to RepoGrams (RQ4).

The metrics were implemented by two junior SE researchers: (Dev1) a masters student who is a co-author on this paper, and (Dev2) a fourth year CS undergraduate student. Dev1 was, at the

[3]We use numbers to identify the 14 anonymous participants.

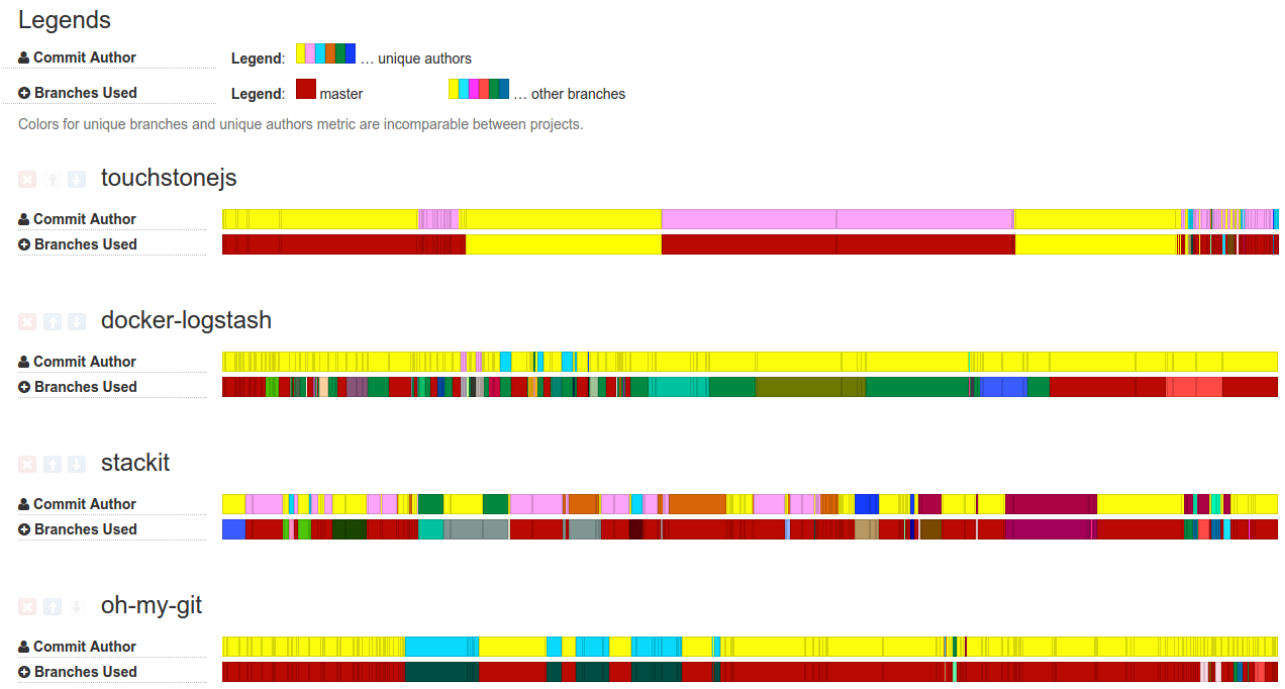

Figure 9: Repository footprints used in question 7.

time, not directly involved in the programming of the tool and was only slightly familiar with the code base. Dev2 was unfamiliar with the project code base. Each developer added three new metrics (bottom six rows in Table 1).

Dev1 added the *POM files*, *commit author*, and *commit age* metrics. Prior to adding these metrics Dev1 spent 30 minutes setting up the environment and exploring the code. The *POM files* metric took 30 minutes to implement and required changing 16 LoC. Dev1 then spent 52 minutes and 48 minutes developing the *commit author* and *commit age* metrics, changing a similar amount of code for each metric.

Dev2 implemented three metrics based on requests made by SE researchers (Section 4.1.3): *files modified*, *merge indicator*, and *author experience*. Prior to adding these metrics Dev2 spent 39 minutes setting up the environment and 40 minutes exploring the code. These metrics took 42, 44, and 26 minutes to implement, respectively. All metrics required changing fewer than 30 LoC.

4.2.1 Summary

The min/avg/med/max times to implement the six metrics were 26 min / 40 min / 43 min / 52 min. These values compares favorably with the time that it would take to write a custom script to extract metric values from a repository (an alternative practiced by almost all SE researchers in our user study). The key difference, however, is that by adding the new metric to RepoGrams the researcher gains two advantages: (1) the resulting project pattern for the metric can be juxtaposed against project patterns for all of the other metrics already present in the tool, and (2) the researcher can use all of the existing interaction capabilities in RepoGrams (changing block lengths, zooming, etc).

5. DISCUSSION

We now briefly discuss the limitations of the RepoGrams tool and how we plan to address these in our future work.

Supporting other project information. RepoGrams currently supports Git repositories. However, Software projects may have bug trackers, Wikis, and other resources that are increasingly studied by SE researchers. It would be helpful to extend RepoGrams to support analysis of these other resources over time along with the repository history. We plan to integrate this information into RepoGrams using the GitHub API, taking into account concerns pointed out in prior work [25].

End-to-end support for evaluation target selection. RepoGrams is designed for qualitative repository analysis, supporting researchers in comparing and filtering a handful of evaluation targets. Complementary approaches, such as Boa [17], support selection and filtering of projects at scale. We are interested in integrating RepoGrams with these existing approaches to provide researchers with a complete, end-to-end, solution for selecting high-quality evaluation targets in their research.

Robust bucketing of metric values. Uniform bucket sizing currently implemented in RepoGrams has several issues. For example, a single outlier metric value can cause the first bucket to become so large as to include most other values except the outlier. One solution is to generate buckets based on different distributions and to find outliers and place them in a special bucket.

Supporting custom metrics. SE researchers in our user study (Section 4.1) wanted more specialized metrics, that were, unsurprisingly, related to their research interests. We are working on a solution in which a researcher would write a metric function in Python and submit it to the server through the browser. The server would integrate and use this user-defined metric to derive repository footprints. We plan to explore the challenges and benefits of this strategy in our future work.

Scalability. RepoGrams can visualize huge repositories, such as the Rails repository (visualized in Figure 3), which contains over 60K commits. However, repositories with long histories result in dense footprints that are difficult to navigate. We plan to include new features to improve RepoGram's scalability. For example, RepoGrams could allow a user to generate a footprint for a window of

commits, rather than for the entire history.

Other use-cases. RepoGrams targets SE researchers. However, it can be also used by managers to track project activity, or by developers to understand a project's practices (e.g., use of branches). For example, a member of Software Carpentry[4] wanted to use RepoGrams to determine if lesson repositories have similar or different contributorship patterns as code and/or documentation.

RepoGrams may be also useful in SE education. We performed a user study with 91 senior undergraduates in a software engineering class to understand whether individuals less experienced with software repositories could comprehend the *repository footprint* concept. We found that RepoGrams can be used successfully by this less experienced population. However, when individuals rely on the visualization without an understanding of the metric underlying the visualization, mis-interpretation of the data may occur.

6. RELATED WORK

The problem of helping SE researchers perform better evaluations has been previously considered from three different angles. First, there are ongoing efforts to curate high-quality evaluation targets in the form of bug reports [35, 7], faults [24], source code [41], and other artifacts. Such databases artifacts promote comparison between proposed techniques and scientific repeatability. Second, researchers have developed tools like GHTorrent [21], Lean GHTorrent [22], Boa [17], and MetricMiner [39] to simplify access to, filtering, and analysis of projects hosted by sites like GitHub. These tools make it easier to carry out evaluations across a large number of projects. Recent work by Nagappan et al. has proposed improvements for sampling projects [31]. Their notion of sample coverage is a metrics-based approach for selecting and reporting the diversity of a sample set of SE projects. Finally Foucault et al. proposed another approach [19] based on double sampling to improve representativeness. Unlike these four strands of prior work, RepoGrams supports SE researchers in narrowing down the set of evaluation targets. In particular, RepoGrams supports qualitative analysis and comparison of the evolution of projects' source code repositories across several metrics.

RepoGrams also builds on a broad set of prior work on visualization of software evolution [16] and software metrics [28]. We now overview the most relevant work from this space.

Novel visualizations span a broad range: revision towers [40] presents a visual tower that captures file histories and helps to correlate activity between files. Gevol [12] is a graph-based tool for capturing the detailed structure of a program, correlating it against project activity and showing how it evolves over time. Chronos [38] assists developers in tracking the changes to specific code snippets across the evolution of a program. RelVis [33] visualizes multivariate release history data using a view based on Kiviat diagrams. The evolution matrix [26] supports multiple metrics and shows the evolution of system components over time. Chronia [20] is a tool to visualize the evolution of code ownership in a repository. Spectographs [46] shows how components of a project evolve over time and like RepoGrams extensively uses color. Other approaches to visualizing software evolution are further reviewed in [13]. Some key features that differentiate RepoGrams from this rich prior work are its focus on commit granularity, no assumptions about the repository contents, support for comparison of multiple projects across multiple metrics, and a highly compact representation.

A more recent effort, The Small Project Observatory [27], visualizes ecosystems of projects. RepoGrams differs in its emphasis on a single unifying view for all metrics and a focus on supporting SE researchers.

The evolution radar [15] visualizes logical coupling between modules and files in a repository using shared commits between files as a coupling measure. This logical granularity is more fine-grained than what RepoGrams presents, but it is also a specific measure. Also, unlike RepoGrams, this tool is not designed for comparing multiple project histories.

The CVSgrab tool [44, 43] creates a visualization based on files in the repository: a stripe in the visualization is a single file, and a metric is applied to its different versions, which results in a view similar to a RepoGrams footprint. The file stripes are then stacked to present a complete project view. In many ways RepoGrams is a simplification of this file-based view: RepoGrams shows a single stripe, corresponding to commits in the repository, rather than a file. This simplified view loses information, but makes the resulting visualization more concise and makes it possible to compare several histories, which is difficult to do with CVSgrab.

ConcernLines [42] is a tool to visualize the temporal pattern of software concerns. It plots the magnitude of concerns on a timeline and uses color to distinguish high and low concern periods. RepoGrams can be extended with a metric that counts concerns expressed in commit messages or in code comments. Fractal Figures [14], visualizes commit authors from software repositories in CVS, using either one of two abstractions. RepoGrams's repository footprint abstraction is similar to Fractal Figures' abstraction called TimeLine View, which assigns a unique color to each commit author and lays all commits as colored squares on horizontal lines. Similarly to RepoGrams, each horizontal line represents a single software repository, and progression from left to right represents the passage of time. RepoGrams includes support for multiple metrics based on the artifacts exposed by the source repository; it also includes a metric that assigns a unique color per author.

7. CONCLUSION

The widespread availability of open source repositories has had significant impact on SE research. It is now possible for a study to consider hundreds of projects with thousands of commits, hundreds of authors, and millions of lines of code. Unfortunately, more is not necessarily better or easier. To properly select evaluation targets for a research study the researcher must be highly aware of the features of the projects that may influence the results.

To help with this issue we developed RepoGrams, a tool for analyzing and comparing software repositories across multiple dimensions. The key idea is a flexible repository footprint abstraction that can compactly represent a variety of user-defined metrics to help characterize software projects over time. We evaluated RepoGrams in a user study with 14 SE researchers and found that it helped them to answer advanced, open-ended, questions about the relative evolution of software projects. The tool is open source and is available online: http://repograms.net/

Acknowledgments

We would like to thank Maike Maas, Sebastian Becking, and Marc Jose who helped build the initial RepoGrams prototype. We would also like to thank Stewart Grant for implementing three of the metrics. Finally, a big thank you to everyone who participated in our user studies and helped us with evaluating the tool. This research is supported by an NSERC discovery award.

[4] http://software-carpentry.org

8. REFERENCES

[1] An extendable open source automation server. https://jenkins-ci.org.

[2] AngularJS âĂŤ Superheroic JavaScript MVW Framework. https://angularjs.org.

[3] Ansible is Simple IT Automation. https://www.ansible.com/.

[4] DB Browser for SQLite project. https://github.com/sqlitebrowser/sqlitebrowser.

[5] Flickr uploading tool for GNOME. https://github.com/GNOME/postr.

[6] jQuery. https://jquery.com/.

[7] Summarizing Software Artifacts. https://www.cs.ubc.ca/cs-research/software-practices-lab/projects/summarizing-software-artifacts.

[8] Welcome to Apache Maven. http://maven.apache.org/.

[9] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 4–14, 2011.

[10] C. A. Brewer. ColorBrewer2. http://colorbrewer2.org/.

[11] K. Chen, P. Liu, and Y. Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *ICSE*, 2014.

[12] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, pages 77–ff, New York, NY, USA, 2003. ACM.

[13] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger. Analysing software repositories to understand software evolution. In *Software evolution*, pages 37–67. Springer Berlin Heidelberg, 2008.

[14] M. D'Ambros, M. Lanza, and H. Gall. Fractal figures: Visualizing development effort for cvs entities. In *Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 3rd IEEE International Workshop on*, pages 1–6, 2005.

[15] M. D'Ambros, M. Lanza, and M. Lungu. The Evolution Radar: Visualizing Integrated Logical Coupling Information. In *MSR*, 2006.

[16] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2010.

[17] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE*, 2013.

[18] M. Foucault, M. Palyart, X. Blanc, G. C. Murphy, and J.-R. Falleri. Impact of Developer Turnover on Quality in Open-source Software. In *ESEC/FSE*, 2015.

[19] M. Foucault, M. Palyart, J.-R. Falleri, and X. Blanc. Computing contextual metric thresholds. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014.

[20] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, IWPSE '05, pages 113–122, Washington, DC, USA, 2005. IEEE Computer Society.

[21] G. Gousios. The ghtorent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[22] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 384–387, New York, NY, USA, 2014. ACM.

[23] K. Herzig and A. Zeller. The Impact of Tangled Code Changes. In *MSR*, 2013.

[24] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, San Jose, CA, USA, July 23–25 2014.

[25] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, New York, NY, USA, 2014. ACM.

[26] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 37–42, New York, NY, USA, 2001. ACM.

[27] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The Small Project Observatory: Visualizing Software Ecosystems. *Sci. Comput. Program.*, 75(4):264–275, Apr. 2010.

[28] T. Mens and S. Demeyer. Future trends in software evolution metrics. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 83–86, New York, NY, USA, 2001. ACM.

[29] C. Metz. How github conquered google, microsoft, and everyone else. http://www.wired.com/2015/03/github-conquered-google-microsoft-everyone-else/.

[30] T. Munzner. *Visualization Analysis and Design*. CRC Press, 2014.

[31] M. Nagappan, T. Zimmermann, and C. Bird. Diversity in software engineering research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 466–476, New York, NY, USA, 2013. ACM.

[32] D. L. Parnas. Classics in Software Engineering. chapter On the Criteria to Be Used in Decomposing Systems into Modules, pages 139–150. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[33] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 67–75. ACM, 2005.

[34] F. Rahman and P. Devanbu. Ownership, Experience and Defects: A Fine-grained Study of Authorship. In *ICSE*, 2011.

[35] S. Rastkar, G. C. Murphy, and G. Murray. Summarizing software artifacts: A case study of bug reports. In *ICSE*, 2010.

[36] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, New York, NY, USA, 2014. ACM.

[37] D. Rozenberg, I. Beschastnikh, F. Kosmale, V. Poser, H. Becker, M. Palyart, and G. C. Murphy. RepoGrams evaluation details. http://repograms.net/msr2016.

[38] F. Servant and J. A. Jones. History slicing: Assisting

code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 43:1–43:11, New York, NY, USA, 2012. ACM.

[39] F. Sokol, M. Finavaro Aniche, and M. Gerosa. Metricminer: Supporting researchers in mining software repositories. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 142–146, Sept 2013.

[40] C. Taylor and M. Munro. Revision towers. In *Visualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop on*, pages 43–50, 2002.

[41] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, Dec. 2010.

[42] C. Treude and M. Storey. Work item tagging: Communicating concerns in collaborative software development. *Software Engineering, IEEE Transactions on*, 38(1):19–34, Jan 2012.

[43] L. Voinea and A. Telea. An Open Framework for CVS Repository Querying, Analysis and Visualization. In *MSR*, 2006.

[44] S. L. Voinea and A. Telea. CVSgrab: Mining the History of Large Software Projects. In *EuroVis*, 2006.

[45] J. Warner. Top 100 most popular languages on github. https://jaxbot.me/articles/github-most-popular-languages, July 2014.

[46] J. Wu, R. Holt, and A. Hassan. Exploring software evolution using spectrographs. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 80–89, Nov 2004.