The Journal of Systems & Software 192 (2022) 111421

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Visualization of aggregated information to support class-level software evolution☆

Mona Rahimi [a],*, Michael Vierhauser [b]

[a] Northern Illinois University, 100 Normal Rd. DeKalb, IL, 60115, United States
[b] Department of Business Informatics–Software Engineering, Johannes Kepler University, Altenberger Str. 69, 4040 Linz, Austria

ARTICLE INFO

Article history:
Received 13 December 2021
Received in revised form 27 May 2022
Accepted 23 June 2022
Available online 25 June 2022

Keywords:
Code changes
Change visualization
Software evolution

ABSTRACT

Context: Software is inherently prone to constant change, especially in the source code, making it difficult for developers to keep track of changes performed over time and to fully understand their implications.
Objective: To this end, we present an Eclipse plug-in for visualizing heterogeneous information, collected from multiple sources, at different levels of granularity. This visualization provides a single graphical representation of a system's change histories over multiple versions, allowing developers to identify the previous and present dependencies in the system, while adding new or removing and modifying the current functionalities of a software. Summarizing and associating the relevant changes in a single graph, further supports developers, not familiar with the overall system, to conduct a self-study and explore the systems design and changes of its functionality over time.
Method: Our tool, DejaVu, initially infers and further visualizes *change scenarios* that have been applied to a given class in source code, across multiple versions of a software. DejaVu additionally augments the change information with prior commits from GitHub repositories, as well as associated issues from Jira issue tracking system.
Evaluation: As part of the evaluation, we conducted a controlled experiment, recruiting participants with research or industrial programming experiences. The participants were asked to investigate and assess a set of change stories with and without the use of the DejaVu. As such, we empirically evaluated the impact of DejaVu in alleviating developers' understanding of code class-level changes across multiple versions.
Results: Our results showed an average of 52% reduction in completion time and a 51% increase in correctness of several change-comprehension tasks once, users adopted DejaVu in comparison to the manual completion of the same tasks. A student's t-test verified the significant improvement in *time* and *correctness* of the tasks with *p-values* of 0.01 and 0.002.
Conclusion: Visualizing aggregated information from multiple sources provides developers with a more comprehensive intuition of the change and its rationale, facilitating software maintenance tasks.

© 2022 Elsevier Inc. All rights reserved.

1. Introduction

Evolution is an innate process associated with modern software systems as they are inherently prone to change due to maintenance activities (Lehman, 1996). New functionalities are introduced, code is refactored, and components are changed to meet new or changing requirements. This, in turn, means that software artifacts, particularly source code, are subject to constant change, making it hard for developers to keep track of or fully understand the nature of past changes (Maalej et al., 2014; Rugaber, 1992). To evolve a codebase, during development or maintenance, developers need to first explore the available information, typically spread across different sources of information, to understand a component's functionality, its past changes, and ideally the reasons why the changes have been made.

Previous studies showed the importance of developers being able to talk to the authors (i.e., fellow developers) of source code for gaining a better understanding of its meaning and purpose (Maalej et al., 2014). However, particularly in larger teams, developers often are not aware of who to contact (Maalej et al., 2014). Additionally, these activities can be rather time-consuming, taking up half of the time developers spend on software maintenance tasks (Murphy et al., 2006). As a response, researchers have proposed a wide variety of solutions to reduce the effort of developers in understanding a software system

☆ Editor: Burak Turhan.
* Corresponding author.
E-mail addresses: rahimi@cs.niu.edu (M. Rahimi), michael.vierhauser@jku.at (M. Vierhauser).

https://doi.org/10.1016/j.jss.2022.111421
0164-1212/© 2022 Elsevier Inc. All rights reserved.

and its changes. Examples range from program summarization techniques (McBurney and McMillan, 2016), feature localization (Poshyvanyk et al., 2007; Dit et al., 2013), to architecture reconstruction, reverse engineering (Garcia et al., 2013; Prete et al., 2010), and software visualization, including visualizing static structure, runtime behavior, and evolution of the software (Diehl, 2007).

However, one common shortcoming of these approaches is that the original rationale for the development and further, the intention for the changes, are physically separated from the software's source code. Searching for this information, scattered across different sources and platforms, makes maintenance tasks more difficult. As the result, developers sometimes have to rely solely on the source code to deduce the intentions of the original developments and change. The rationale behind the implemented source code and further changes, if not directly documented, remains hidden and needs to be explored before further activities can be performed (Alshakhouri et al., 2018). Furthermore, the majority of the approaches lack visualization at a proper level of change granularity (Hattori et al., 2013). Providing fine-grained information regarding the change not only helps developers to better understand the nature of the change but also supports them to rationalize the change.

To address these challenges, we have developed DejaVu to provide an integrated view of class-level changes, statement-level changes, and the reasons for these changes. DejaVu, an Eclipse plug-in, visualizes high-level changes applied to an individual class and a schema of the change across multiple subsequent versions of a software system. DejaVu further, upon request, provides more fine-grained information about the change, such as statement-level changes. It further aggregates information from commit messages and issue trackers, as a way to potentially capture the original developer's intention of the change. DejaVu, therefore, provides the following features:

- (i) it **infers** change scenarios that occurred to a certain class;
- (ii) it **visualizes** the inferred change scenarios in different levels of granularity (statement and class level) and the impact of the change on the associated artifacts (requirements or features); and
- (iii) it **augments** the change scenarios with the developer's original intentions and rationale alongside the related source code component (e.g., class).

DejaVu is based on our previous work Trace Link Evolver (TLE), a Java-based tool to automatically evolve source-to-requirements trace links in response to changes in an underlying system. DejaVu herein reuses TLE's algorithm to detect change scenarios that occurred in the artifacts, extracts scenarios for a certain class, and adds a visualization component, displaying the scenarios of the changes in the form of a single evolutionary graph. DejaVu further augments the graph with additional information, collected from source code repositories and issue tracking systems. Contributions of DejaVu are therefore two-fold:

- **Visualization:** We visualize the evolutionary history of a given code component through a comprehensive and user-friendly format. DejaVu visually displays the history of past changes of a certain code component and its associated features (requirements) across subsequent versions of a software system in the form of an evolutionary graph.
- **Augmentation:** To unite information about the change, the intention of the change, and the changed artifacts (source code and features) in a single development environment, DejaVu augments the evolutionary graph with the additional change information alongside the source code within the Eclipse IDE. This holistic view helps developers to better understand how a certain class has reached its current state.

To assess the usefulness of our augmented evolutionary graph, we conducted an empirical study with software developers. Results of this controlled experiment showed a significant reduction in completion time and an increase in the correctness of the completed tasks when our participants used DejaVu. All supporting artifacts of the study, including the DejaVu prototype, questionnaires, and additional study material can be accessed on our public repository.[1]

The primary benefit of DejaVu is the enhanced comprehension of the codebase and its evolution achieved through integrating multiple information sources into a single visual graph within an environment, commonly used by programmers. Presenting previous changes of a class, first at class-level and further at statements-level, to developers in a visual format improves their understating of the current state and role of the class in the code. Using DejaVu, developers can further retrieve commit messages, as well as trace issues relevant to a certain change. This process saves a great amount of developers' time, while compensating for the knowledge that should be otherwise collected from the original programmer of the code who may have left the project.

In this article, our previous work is summarized in Section 2, and DejaVu is presented in Section 3. We describe our empirical study, share our findings, discuss the results, and provide threats to validity in Sections 4, 5, and 6. Finally, Section 7 describes related work, and Section 8 concludes with potential ideas for future work.

2. Background

In this section, we first provide a brief overview of source code and requirements co-evolution and summarize the implementation of TLE,[2] and finally, explain how DejaVu extends TLE's functionalities.

2.1. Requirements and code co-evolution

Changes in software systems often reflect concurrently across various types of artifacts. To identify common patterns of change in the source code and corresponding requirements, we performed a systematic literature review, and analyzed source code and requirements changes in several open-source and closed-source systems. From our findings, we systematically identified 24 requirements and source code *change scenarios* at the class and method levels, as well as their associated properties. Table 1 provides a subset of these class-level co-evolutionary patterns that we selected to be relevant to the context of our current work. The remaining patterns are more fine-grained, at the method-level, and are out of scope of DejaVu and this study. In this paper, we focus on visualizing class-level changes, and throughout this paper, we refer to these patterns as *change scenarios*.

2.2. Trace Link Evolver (TLE)

Given the change scenarios, we implemented TLE to detect their occurrence through a set of pre-defined heuristics, coupled with refactoring detection tools, and informational retrieval algorithms. Each change scenario is defined as a set of lower-level properties, which are expected to hold if the scenario has

[1] https://github.com/jku-win-se/dejavu.

[2] The work presented here extends our three prior publications. First, a paper discussing the change scenarios used in TLE (Rahimi and Cleland-Huang, 2015). Second, a paper explaining the properties, heuristics, and algorithms TLE implements to identify change scenarios (Rahimi et al., 2016). Finally, an Empirical Software Engineering (EMSE) journal paper, in which we added a human-in-the-loop and evaluated several variants of TLE in multiple new environments (Rahimi and Cleland-Huang, 2018).

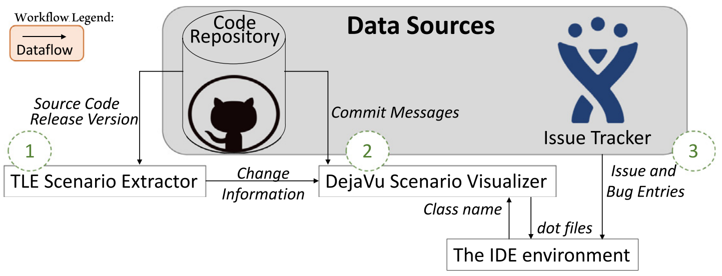


Fig. 1. DejaVu workflow: High-level view of DejaVu and its primary components.

Table 1
DejaVu class-level *Change Scenarios* inferred through a combination of refactoring tools and information retrieval techniques.

ID	Change scenario
CS_1	New functionality
CS_2	Extracted class
CS_3	Added merged classes
CS_4	Promoted method
CS_5	Extracted subclass
CS_6	Extracted superclass
CS_7	Obsolete functionality
CS_8	Divided class
CS_9	Deleted merged classes
CS_{10}	Renamed class

occurred. These properties are defined according to both **structural** and **semantic** changes in the codebase and requirements. For instance, consider the case of CS_3, in which an added class C_{i+1}, in the new version $i + 1$, is created through merging two of the existing classes C'_i and C''_i in the previous version i of the software. To assess this scenario, the six properties defined in Table 2 are assessed. Here we refer to an *antecedent* class as one that existed in the original version i, regardless of whether it still exists in the current version. Further, we refer to an associated class as a class *associated* with another class through structural dependencies including associations, aggregations, and composition relationships. Artifact Property 1, AP_1, states that a new class is added. AP_2 states that antecedent classes should exist as being semantically similar to the newly added class. In our work, we used cosine similarity as a measure of semantic similarity (Rahimi and Cleland-Huang, 2018). AP_3 states that requirements that are semantically similar to the newly added class are a subset of requirements union of any two antecedent classes. AP_4 states in case of the scenario occurrence, two antecedent classes exist whose methods union is a superset of methods in the new class. AP_5 states that classes associated with the new class are a subset of associated classes with the two antecedent classes. Finally, AP_6 states that the two identified antecedent classes no longer exist in the new version. When these properties hold true change scenario CS_3 is recognized.

To further evolve code-to-requirements trace links automatically according to the detected change scenarios, TLE uses a set of *link evolution heuristics* to specify which links should be created, deleted, modified, and maintained, when a change scenario is detected. For instance, in case of identifying CS_3, TLE uses the heuristic $l(C_{i+1}, R_{C'_i} \cup R_{C''_i})$ to create trace links between the newly added class C_{i+1} (in version $i + 1$) and requirements that used to be linked to the two antecedent classes in version i. Additionally, TLE applies the heuristics $!l(C'_{i+1}, R_{C'_i})$ and $!l(C''_{i+1}, R_{C''_i})$ to remove the stale links associated with the antecedent classes C'_i and C''_i in the new version.

Finally, we evaluated the efficacy of TLE and its multiple variants in an extensive series of experiments in different environments: a greenfield Unmanned Aerial Vehicle (UAV) system, and a large open-source system, Apache Cassandra. The results showed that TLE outperforms other approaches (Rahimi and Cleland-Huang, 2018). The reason is that TLE reasons about the changes that previously occurred and evolves trace links according to the detected change scenarios, while other standard approaches generate links from scratch for the newer version, often based on semantic similarities, without the knowledge of the past changes.

Please note in order to keep the focus on the original work in this paper, we only included a small example, one change scenario, the associated properties, and the relevant link evolution heuristics. For other scenarios, artifact properties, and link evolution heuristics, as well as details regarding our evaluations and experiments please refer to our previous works (Rahimi and Cleland-Huang, 2015; Rahimi et al., 2016; Rahimi and Cleland-Huang, 2018).

3. DejaVu

DejaVu builds on the algorithm provided by TLE for detecting change scenarios. However, our focus is mainly on visualizing class-level changes, as well as an upon-request demonstration of statement-level changes from GitHub and relevant issues from Jira. Table 1 lists class-level change scenarios which DejaVu is able to detect and visualize across contiguous versions of a software. To detect these scenarios DejaVu only requires subsequent versions of the source code and requirements. If the user is interested in visualizing the evolution of trace links as well, then an initial set of requirements-to-code links is also required. The following section describes DejaVu workflow and details of its implementation.

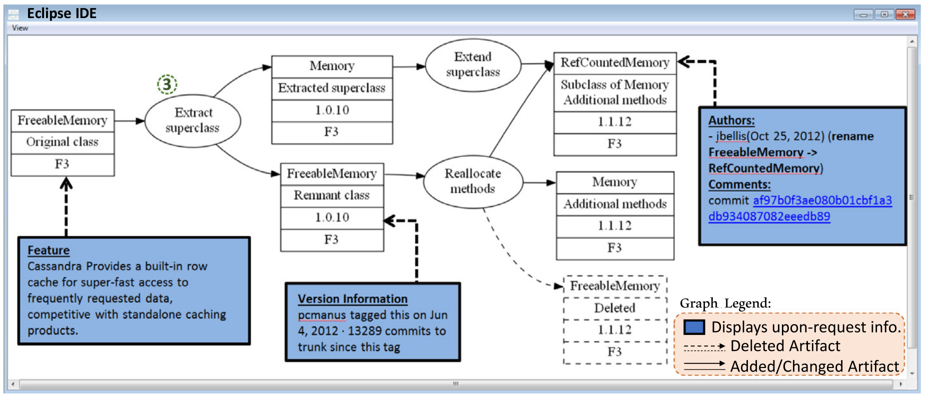
3.1. DejaVu workflow

DejaVu is comprised of three main components:
(i) Fig. 1-①: a component to infer a set of change scenarios that happened to a certain class across multiple versions. This component uses the same scenario extractor algorithm, which TLE uses and explained earlier in Section 2.2. The component then passes the identified information to a visualizer component.
(ii) Fig. 1-②: a user interface for visualizing scenarios. This component takes a class name from the user and retrieves relevant scenarios to the given input from the TLE-provided information containing a list of changed classes, the respective change scenarios for each version, contributing classes, and their roles in scenarios. Subsequently, the scenarios are merged and integrated into a single file which is then used to visualize a single evolutionary graph, illustrated in Fig. 2. (iii) Fig. 1-③: extensions for connecting additional sources of information (GitHub commits and Jira issues) to change scenarios in the graph. This component gathers fine-grained information from GitHub and Jira to augment the change scenarios in the graph, with information on statement-level changes, commit descriptions, commit messages, and information about the author(s) of the change. To consolidate the retrieved information from various sources, DejaVu displays the changed information next to the changed code in the Eclipse IDE.

Table 2
A subset of our Heuristics (Artifact Properties) to Determine if change Scenario CS_3 has Occurred (Given Properties are Discussed in the Text).

ID	Artifact properties (cS_3: Added Merged classes)
AP_1	$\exists\ c \in C_{i+1} \mid \nexists c \in C_i\ \ \wedge \dots$
AP_2	$\dots \exists\ c'c'' \in C_i \mid Sim(c, c') \wedge Sim(c, c'') \wedge \dots$
AP_3	$\dots \forall\ r \in R_c \forall\ r' \in R_{c'} \forall\ r'' \in R_{c''} \mid Sim(r, c) \subseteq Sim(r', c') \cup Sim(r'', c'') \wedge \dots$
AP_4	$\dots \forall\ m \in M_c \forall\ m' \in M_{c'} \forall\ m'' \in M_{c''} \mid m \subseteq m' \cup m'' \wedge \dots$
AP_5	$\dots \forall\ d \in D_c \forall\ d' \in D_{c'} \forall\ d'' \in D_{c''} \mid d \subseteq d' \cup d'' \wedge \dots$
AP_6	$\dots \nexists c'c'' \in C_{i+1}$

C_i is the class set in version i;
C_{i+1} is the class set in version $i+1$;
R_c, M_c, D_c are requirement set, method set, and between-class association set of class c;
$Sim(a, b)$: a & b have cosine similarity; $A(a, b)$ a & b are associated;



Feature
Cassandra Provides a built-in row cache for super-fast access to frequently requested data, competitive with standalone caching products.

Version Information
pcmanus tagged this on Jun 4, 2012 · 13289 commits to trunk since this tag

Authors:
- jbellis(Oct 25, 2012) **(rename FreeableMemory -> RefCountedMemory)**
Comments:
commit af97b0f3ae080b01cbf1a3db934087082eeedb89

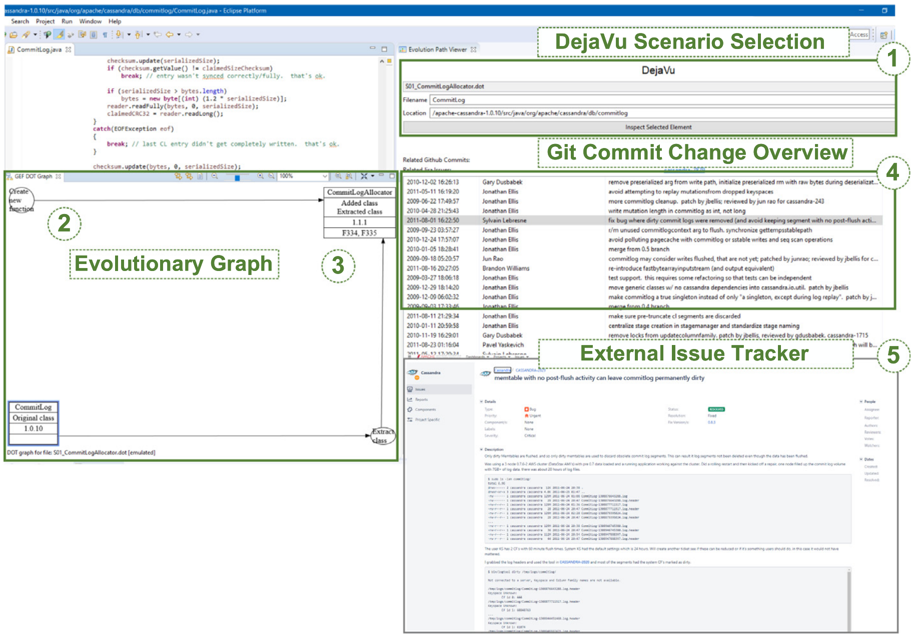
Fig. 2. An example of the generated graph as a holistic view of multiple change scenarios. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3.2. Visualizing class-level evolution

The first design decision for efficient visualization of the gathered information was to merge and display all change scenarios, for a given class and its change-contributing classes, in a single graph. The reason was to provide the user with the entire change information in a single view. A holistic view of a chain of changes, initiated in the given class, better reveals the rationale for applying the changes. Additionally, individual display of scenarios first, overpopulates the screen, confusing the user, and second, since a class could participate in more than one scenario, the individual display results in multiple visualizations of the same classes, negatively impacting the performance. For instance, in Fig. 2 class `Memory.java` is an extracted superclass in version 1.0.10, while the same class is also extended in a later version. Individual visualization of scenarios would result in generating class `Memory.java` twice, increasing the cost of visualization, and overcrowding the view. However, integrating scenarios into a single graph not only is space-saving, but also improves readability of a large amount of related information by revealing interrelations and removing duplicates. This design decision maximized the information DejaVu visualized in a minimized space.

Our second design decision relates to the graph visualization granularity level. As shown in Fig. 2, the initial change visualization is at the class level in the graph. We noticed that integrating additional fine-grained information about the change, such as statement-level information, in the same view overpopulates the graph, impeding developers from finding useful information. To prevent information overload, we decided to provide users with more details only upon request. Blue rectangles in Fig. 2 are placeholders representing additional click-request details that DejaVu provides, such as change author, change comments, associated commits, version details, and feature descriptions.

The final output represents the history of the detected changes in the form of a graph with the root being the input class. The scenarios are ordered by system's version numbers from the oldest on the left to the newest on the right. Vertically, the newly-added classes appear first on the top of the graph, while the deletions are represented with dashed lines and are displayed last at the bottom of each layer of the graph. The associated scenarios, having one or more entities in common, are placed next to each other to simplify the visualization of their relations. The graph layout algorithm, adopted here, namely 'dot', is a filter for drawing directed acyclic graphs, aiming edges in the same direction (top to bottom, or left to right) and then attempts to avoid edge crossings and reduce edge length (Ellson et al., 2001). To minimize the display area, we represented each change scenario (listed in Table 1) as an *oval* in the graph, only containing a short description of the change. However, the participating classes in the scenarios contained multiple associated information: (i) the name of the class (ii) its role of the class in the change scenario (iii) the version in which the scenario occurred (iv), and finally, features associated with the class. Therefore, selecting a *rectangle* for class representation enabled us to optimally include and separate each class property within lines. To distinguish between the existing and deleted artifacts, we used dashed lines for the deleted elements and solid lines for the existing ones. For instance, the graph in Fig. 2 depicts the evolution history of a given class `FreeableMemory.java` across 27 subsequent versions of Apache Cassandra. In the Cassandra documentation, requirements are referred to as *features*, therefore we use requirement and feature interchangeably. The graph shows that in version 1.0.10 a new class `Memory.java`, was extracted as a superclass from the original class. Furthermore, in version 1.1.12, the recently created superclass, `Memory.java`, was extended by a subclass, `RefCountedMemory.java`, and methods from the class `FreeableMemory.java` were re-allocated across the superclass and



```
checksum.update(serializedSize);
if (checksum.getValue() != claimedSizeChecksum)
    break; // entry wasn't synced correctly/fully.  that's ok.

if (serializedSize > bytes.length)
    bytes = new byte[(int) (1.2 * serializedSize)];
reader.readFully(bytes, 0, serializedSize);
claimedCRC32 = reader.readLong();
}
catch(EOFException eof)
{
    break; // last CL entry didn't get completely written.  that's ok.
}

checksum.update(bytes, 0, serializedSize);
```

S01_CommitLogAllocator.dot
Filename: CommitLog
Location: /apache-cassandra-1.0.10/src/java/org/apache/cassandra/db/commitlog

DOT graph for file: S01_CommitLogAllocator.dot [emulated]

2010-12-02 16:26:13	Gary Dusbabek	remove preserialized arg from write path, initialize preserialized rm with raw bytes during deserializat...
2011-05-11 16:19:20	Jonathan Ellis	avoid attempting to replay mutationsfrom dropped keyspaces
2009-06-22 17:49:57	Jonathan Ellis	more commitlog cleanup. patch by jbellis; reviewed by jun rao for cassandra-243
2010-04-28 21:25:43	Jonathan Ellis	write mutation length in commitlog as int, not long
2011-08-01 16:22:50	Sylvain Lebresne	fix bug where dirty commit logs were removed (and avoid keeping segment with no post-flush acti...
2009-09-23 03:57:27	Jonathan Ellis	r/m unused commitlogcontext arg to flush. synchronize gettempsstablepath
2010-12-24 17:57:07	Jonathan Ellis	avoid polluting pagecache with commitlog or sstable writes and seq scan operations
2010-01-05 18:28:41	Jonathan Ellis	merge from 0.5 branch
2009-09-18 05:20:57	Jun Rao	commitlog may consider writes flushed, that are not yet; patched by junrao; reviewed by jbellis for c...
2011-08-16 20:27:05	Brandon Williams	re-introduce fastbytearrayinputstream (and output equivalent)
2009-03-27 18:06:18	Jonathan Ellis	test support. this requires some refactoring so that tests can be independent
2009-12-29 18:14:20	Jonathan Ellis	move generic classes w/ no cassandra dependencies into cassandra.io.util. patch by jbellis
2009-12-09 06:02:32	Jonathan Ellis	make commitlog a true singleton instead of only "a singleton, except during log replay". patch by j...
2009-09-03 17:33:46	Jonathan Ellis	merge from 0.4 branch
2011-08-11 21:29:34	Jonathan Ellis	make sure pre-truncate cl segments are discarded
2010-01-11 20:59:58	Jonathan Ellis	centralize stage creation in stagemanager and standardize stage naming
2010-11-19 16:29:01	Gary Dusbabek	remove locks from updatecolumnfamily. patch by jbellis, reviewed by gdusbabek. cassandra-1715
2011-08-23 01:16:04	Pavel Yaskevich	[illegible]

Cassandra / CASSANDRA-2829
memtable with no post-flush activity can leave commitlog permanently dirty

Details
Type: Bug
Priority: Urgent
Component/s: None
Labels: None
Severity: Critical
Status: RESOLVED
Resolution: Fixed
Fix Version/s: 0.8.3

Description
Only dirty Memtables are flushed, and so only dirty memtables are used to discard obsolete commit log segments. This can result it log segments not been deleted even though the data has been flushed.

Fig. 3. DejaVu as part of the Eclipse IDE.

subclass. Finally, the original class `FreeableMemory.java` was deleted in the same version. If the requirements-to-code trace matrix is provided for the initial version, DejaVu evolves the links and displays the associated requirement within each class as well, otherwise, this space remains empty. For instance, the original class `Memory.java` is initially associated with *Feature 3 (F3): "Cassandra provides a built-in row cache for super fast access to frequently requested data."*. However, the graph shows that this feature is further re-associated with the extracted superclass, subclass, and the reallocated method.

As a use case scenario consider version 1.0.11, in which only the two first levels of the evolutionary graph exist. A developer tends to implement a new memory feature, called RefCounted, using a reference counting method to improve performance. Providing the DejaVu-generated graph, the developer now knows that a class `Memory.java` is extracted as a superclass for an existing class `FreeableMemory.java` (by another developer in past versions). This information offers the developer to place `RefCountedMemory.java` within the same hierarchy for inheriting the appropriate super functionalities.

3.3. Augmenting the evolution

DejaVu is designed to be interactive so that upon request more change details are extracted and displayed to the user. Thus, to assist developers to understand the change rationale, for each participating class in a scenario, additional information is collected from the web. When a class is selected in the graph, a list of existing commits, on GitHub repository, will be shown to users alongside the image. Users can click on a commit to open the related page in their default web browser for further investigations.

Additionally, commit messages are parsed to identify whether or not they contain links to Jira issues. If they do then Jira links are extracted from the issue tracking framework and provided in the interface. Users can then click-select to extend the link further in the browser. This information is extremely helpful for understanding how classes in a system have evolved into their current states. For instance, Fig. 4 lists *commit messages* associated with the `FreeableMemory.java` change scenario in Fig. 1-②. As discussed, this scenario refers to the extraction of `Memory.java` to be a superclass for `FreeableMemory.java`. The associated commit message states *"add serializing cache provider"*, providing the rationale of this change. Further, Fig. 5 represents the associated Jira issue stating *"Cassandra-1969: To Improve GC performance"* explaining the ultimate intention of this change. Further the description of the issue states *"Java BB.allocateDirect() will allocate native memory out of the JVM and will help reducing the GC pressure in the JVM with a large Cache. From some of the basic test it shows around 50% improvement than doing a normal Object cache..."*

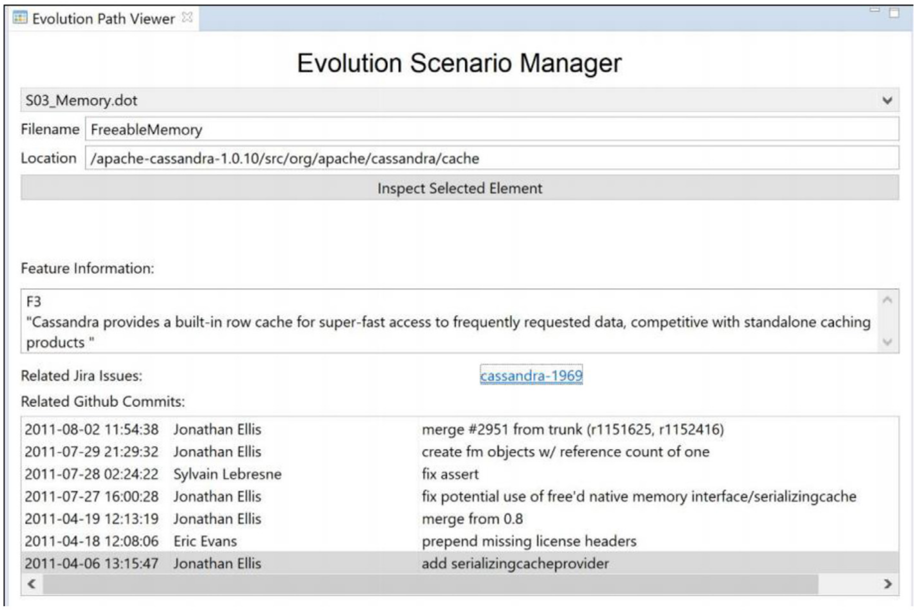
3.4. Implementation

We used Graphviz (2022), an open-source framework providing a wide range of graph-based visualizations (Riesco et al., 2009; Shaw et al., 2016; Saito et al., 2016; Bohnet and Döllner, 2006), to generate graph representations from change scenarios. Later JGit (2022) was used to parse git repositories and collect commits for corresponding classes in the scenarios. All components are implemented in Java and the information is displayed within Eclipse IDE, where we implemented DejaVu as a plugin to visualize scenarios (Graphviz files), commit information, and issue entries alongside the source code. The reason for this decision was to provide a framework in which developers can first understand the code and second modify the code within the same environment.

DejaVu's implementation is particularly cost-effective in terms of the number of user interactions, as well as time performance. The Visualizer only requires a class name, which the user is interested in its change history, as input through the Eclipse console. The TLE-generated report is then parsed and the dot-file is generated for the given class within seconds. To display a change graph and more detailed information regarding the change one can select the generated dot-file in the drop-down menu in DejaVu scenario selection. The graph is generated and the additional information is retrieved within a few seconds.

4. Empirical study

We conducted a controlled experiment to empirically evaluate the applicability and usefulness of DejaVu. We were particularly


Evolution Path Viewer

Evolution Scenario Manager

S03_Memory.dot

Filename: FreeableMemory

Location: /apache-cassandra-1.0.10/src/org/apache/cassandra/cache

Inspect Selected Element

Feature Information:

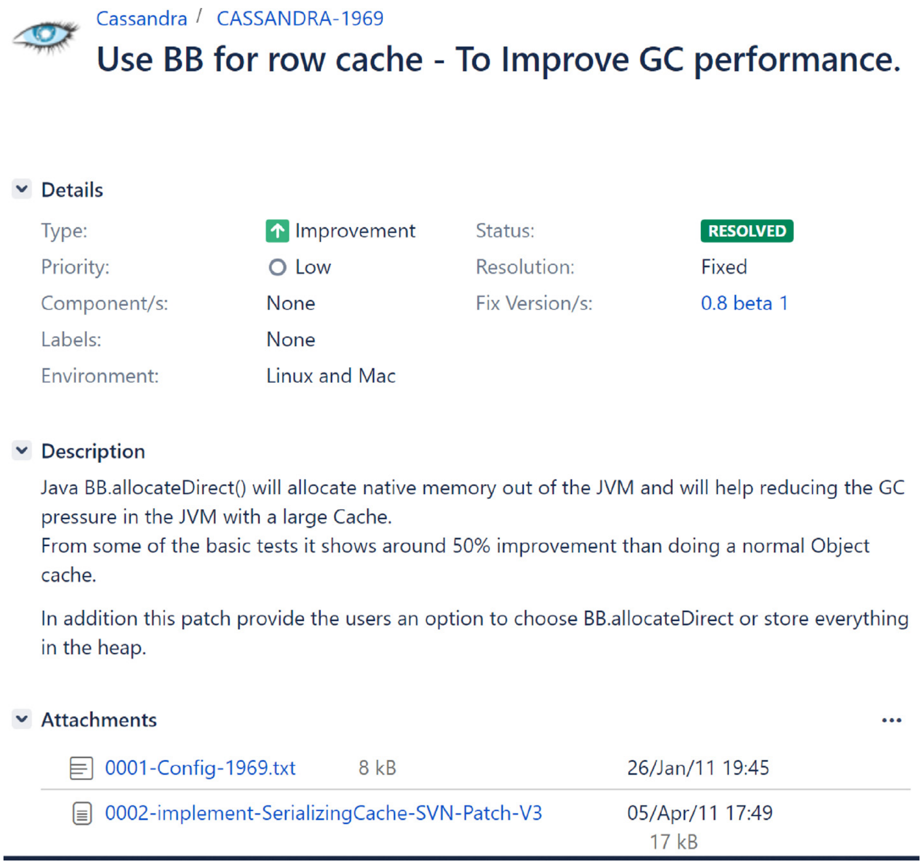
F3
"Cassandra provides a built-in row cache for super-fast access to frequently requested data, competitive with standalone caching products "

Related Jira Issues: cassandra-1969

Related Github Commits:

```
2011-08-02 11:54:38   Jonathan Ellis      merge #2951 from trunk (r1151625, r1152416)
2011-07-29 21:29:32   Jonathan Ellis      create fm objects w/ reference count of one
2011-07-28 02:24:22   Sylvain Lebresne    fix assert
2011-07-27 16:00:28   Jonathan Ellis      fix potential use of free'd native memory interface/serializingcache
2011-04-19 12:13:19   Jonathan Ellis      merge from 0.8
2011-04-18 12:08:06   Eric Evans          prepend missing license headers
2011-04-06 13:15:47   Jonathan Ellis      add serializingcacheprovider
```

Fig. 4. DejaVu External Issue Tracker for a specific issue and the linked entry in Jira.


Cassandra / CASSANDRA-1969

Use BB for row cache - To Improve GC performance.

Details

Type:	Improvement	Status:	RESOLVED
Priority:	Low	Resolution:	Fixed
Component/s:	None	Fix Version/s:	0.8 beta 1
Labels:	None		
Environment:	Linux and Mac		

Description

Java BB.allocateDirect() will allocate native memory out of the JVM and will help reducing the GC pressure in the JVM with a large Cache.
From some of the basic tests it shows around 50% improvement than doing a normal Object cache.

In addition this patch provide the users an option to choose BB.allocateDirect or store everything in the heap.

Attachments

0001-Config-1969.txt	8 kB	26/Jan/11 19:45
0002-implement-SerializingCache-SVN-Patch-V3	17 kB	05/Apr/11 17:49

Fig. 5. DejaVu Jira Issue Entry related to the change detected by DejaVu.

interested in two factors: first, if and to what extent DejaVu **accelerates** the process of comprehending the modified or evolved parts in the source code, and second if using DejaVu also **improves the quality** of understanding these changes. According to these two factors, we further derived two research questions:

RQ1: *Does DejaVu accelerate the comprehension of evolution in a software system*? We assessed the time needed to complete a variety of tasks with or without DejaVu. Our hypothesis is that providing developers with DejaVu results in spending less time on completion of the assigned tasks of exploring specific

Table 3
List of tasks: C, C', and C'' are replaced with specific class names, v_i and v_{i+1} with old and new versions, respectively.

T_i	Tasks (Description of Change Scenario CS_i)
T_1	New class C is added in v_{i+1}, indicating new functionality f_{i+1}.
T_2	New class C is extracted in v_{i+1} from an existing class C' in v_i.
T_3	Existing classes C', C'' in v_i are merged to a single new class C in v_{i+1}.
T_4	Existing method m in v_i is promoted to a new class C in v_{i+1}.
T_5	New subclass C in v_{i+1} is extracted from existing superclass C' in v_i.
T_6	New superclass C in v_{i+1} is extracted from existing C' in v_i.
T_7	Existing class C is deleted in v_{i+1}, indicating obsolete functionality f_i.
T_8	Existing class C in v_i is divided into two or more parts C', C'' in v_{i+1}.
T_{10}	Existing class C in v_i is renamed C' in v_{i+1}.

Table 4
Examples of potentially improved maintenance tasks in the software new version v_i. The T_i IDs refer to the same task IDs in Table 3.

T_i	Maintenance activities in new version v_{i+1}
T_1	Modifying f_{i+1}; New functionality f'_{i+1} interdependent with f_{i+1}.
T_2	Changing C_{i+1} requires change in C'_{i+1}(C_{i+1} Provenance).
T_3	Changing C_{i+1} requires change in C', C''_{i+1}(C_{i+1} Provenance).
T_4	Changing C_{i+1} requires change in other classes used to call m in v_i.
T_5	Developing a new class which belongs to the same hierarchy.
T_6	Developing a new class which belongs to the same hierarchy.
T_7	Implementing f_i, Developing f'_{i+1} interdependent with f_i.
T_8	Changing C', C''_{i+1} may require change in C_{i+1}(C', C''_{i+1} Provenance).
T_{10}	Changing C'_{i+1} may require change in other classes used to call C in v_i.

change scenarios. Our null hypothesis, therefore, is that using DejaVu does not significantly reduce the completion time of such activities.

RQ2: *Does DejaVu increase the quality of the comprehension of evolution in a software system?* We investigated the rate of successfully completed tasks of verifying change scenarios. Our hypothesis is that providing developers with DejaVu will positively impact the correctness of the completed tasks, while the null hypothesis considers no significant improvement in the correctness of tasks.

4.1. Research method

To answer the research questions, we selected a subset of DejaVu change scenarios as tasks to be evaluated by participants in our study. We investigated two aspects in which using DejaVu is potentially useful for developers:

Product Evolution Comprehension: We investigate if DejaVu's inference and visualization of change, properly describe the evolution of a certain component over time, across multiple versions of a software system. In this regard, the goal of DejaVu is to foster a deeper and more in-depth understanding of changes that have happened to a component in an integrated development environment.

Product Comprehension: In addition to providing change-related information, the goal of DejaVu is to connect the dots between information provided by various tools and data sources. DejaVu tends to make this information visible to the user for a better understanding of the component's intended functionality, design decisions made, and rationales behind them.

DejaVu's Intended Audience: The aggregation of the relevant change scenarios, which occurred across several versions of a software – and likely by multiple developers – in particular, assists the developers involved in the project. For instance, the task of responding to a feature request is accelerated, once the new feature somewhat relates to the existing functionalities of the system. For instance, both functionalities could share a similar super functionality in a mutual superclass; they could have similar sub-functions in common, and they both could attempt to satisfy a mutual system-level requirement. DejaVu reveals the implied relations among classes, supporting developers to better understand the relation of the need-to-be-developed feature to the existing features, as well as the feature's impact on the current elements of the system. In addition, demonstrating a holistic view of a system's history, DejaVu helps to bring the project newbies up to speed, since fixing bugs and adding and modifying functionalities require developers to first understand the existing codebase.

4.1.1. Study design and setup

The purpose of qualitative methods is to describe and clarify human-lived experience of a phenomenon (Polkinghorne, 2005; Kitchenham et al., 2002; Seaman, 1999). We, therefore, selected a qualitative research method to empirically evaluate DejaVu's usefulness according to the experience of a representative human-developers sub-population. Our study was structured based on the guidelines by Runeson et al. (2012) and Ko et al. (2015):

Terminology: The study was initiated by carefully selecting a number of *stories*. A *"story"* contains all change scenarios that DejaVu detected for a particular class. The term *"change scenario"* refers to the same high-level scenario of changes that TLE is able to detect (cf. Table 1). The change scenarios in each story were selected for assessment by our study's participants. For the purpose of simplifying the study, we provided the descriptions of change scenarios in simple natural language and referred to them as *"task"*s. These descriptions are provided in Table 3, although in the study, C, C', v_i, and v_{i+1} were replaced with the actual class names and version numbers of the system under test. Though these tasks are not necessarily maintenance tasks, they still provide the prerequisite knowledge for maintaining the change-participating classes and their associated features. For instance, Task 1 in Table 3 notifies developers that the intention of adding a new class C in version $i+1$ was to provide a new functionality f_{i+1}. As such, later over time, once the functionality f_{i+1} requires modification, the developer will know that the change requires to be initiated in the class C; or once the developers tend to further add a new functionality f'_{i+1} to the product, which is interdependent with f_{i+1}, the developer will be informed to initially investigate the class C for potential relations between the two functionalities. For instance, if the two functionalities address the same system-level requirement or share a few sub-functionalities, then there are potential opportunities for code reuse. Additionally, if f_{i+1} inherits parts of the mutual functionalities from a superclass, then the developer will identify the superclass, required to be extended by the new functionality. As another example, the second maintenance task in Table 4 represents a scenario, in which a developer is assigned to modify an existing feature, which is implemented in a class named C_{i+1}. If the developer is notified of the class provenance, called C'_i, by DejaVu, then she will be able to propagate the change, if necessary, to the possible remnants of the feature in the new version of the base class, C'_{i+1}.

To summarize, advising developers of the class provenance, as well as the implicit hierarchical and functional dependencies between classes, will improve their analysis of change impacts and potential code reuse during the maintenance activities. Table 4 provides more examples of the maintenance activities, potentially alleviated with the information, which DejaVu tasks in Table 3 provide.

Task Selection: To select appropriate tasks for the study, we analyzed multiple stories, inferred by DejaVu from Java classes across 27 subsequent versions of Apache Cassandra (Apache Cassandra Database, 2022), a distributed database management system. We selected Apache Cassandra because it represents an open-source, well-documented, and non-trivial-sized system. For the selection of stories our inclusion criteria were stories with tasks that (i) are feasible to be assessed in a designated time; (ii) require no domain knowledge about Cassandra; (iii) refer to change scenarios, more familiar to programmers; and finally (iv) are better representatives of real tasks developers encounter during maintenance and bug fixing activities. As a result, we selected six distinct stories with a different number of tasks (size) and with different levels of difficulty (complexity) for verification to reflect realistic maintenance activities a developer would typically encounter.

Participant's Bias: To prevent biased decisions, we additionally included incorrect scenarios, which have not occurred in the source code. For instance, we included tasks with false scenarios, such as specifying classes that were not, in fact, merged in the newer version. At the beginning of the study sessions, we informed participants of the presence of these mock change scenarios in the stories to prevent them from agreeing with all tasks before sufficient investigation.

We designed our study as a combination of within-subject and between-subject studies to limit the existing risks in each design. The two conditions (treatments) in our experiment refer to completing the assigned tasks once with the use of DejaVu, and once again without the use of the tool. In within-subject studies, both conditions are assessed by the same participant. However, Within-subject design is known to be biased, due to subjects' learning effects, meaning that participants may potentially exhibit improved performance in completing tasks under the second condition, simply due to their prior practice with the same task under the first condition. However, between-subject design, which tests each condition by different participants, is also potentially influenced by differences in participants' background.

Thus to minimize both effects, we designed a hybrid method. In our design, the beginning participant was asked to randomly select half of the stories – without looking at their contents – to be evaluated with the use of the tool, while the remaining half were evaluated without using the tool. Then with the next participant, we rotated the tasks, to be completed with or without the use of the tool. For instance, if a participant, P_1, randomly decided to verify tasks within stories S_1 and S_2 when using the tool but (tasks within) S_3 and S_4 without the tool, further we asked the next participant, P_2, to complete the stories in the reverse order. Therefore, P_2 evaluates S_1 and S_2 without using the tool but S_3 and S_4 with the tool. Facing tasks for which no tool was provided, the participants were able to manually inspect the source code, compare versions of a class using the integrated diffing tool or inspect class hierarchies. As such, all stories were equally biased and unbiased of participants' learning effects. At the same time, the stories were also completed by a variety of participants with different backgrounds. Moreover, we rotated the sessions starting with or without using the tool. For instance, if participant P_1 started her session with stories for which she was using the tool, we later assigned participant P_2 to initiate her session with stories without the tool. This way, both conditions were equally influenced by the participants' learning effect. Table 5 provides an overview of our assignments.

The systematic rotation of task assignments removed potential bias and improved the validity of our experiment in three ways: (i) First, having half of the participants start their session with DejaVu and the other half without the tool counterbalanced participants' learning experiences. (ii) Second, the rotations caused each participant to be exposed to both treatments (with and without DejaVu). Thus, the results are not biased by different participants' backgrounds. (iii) Third, (tasks within) each story was completed with DejaVu half of the time (4 instances) and without the tool for the other half (4 instances). Therefore, we had an equal number of both treatments for each story.

Table 5
Rotation of Story(S)-Assignments to Participants(P): "Tool/NoTool" refer to DejaVu.

Part.:	Started:	S_1	S_2	S_3	S_4
P_1	***Tool***	NoTool	NoTool	Tool	Tool
P_2	***NoTool***	Tool	Tool	NoTool	NoTool
P_3	***Tool***	Tool	Tool	NoTool	NoTool
P_4	***NoTool***	NoTool	NoTool	Tool	Tool
P_5	***Tool***	Tool	NoTool	Tool	NoTool
P_6	***NoTool***	NoToo	Tool	NoTool	Tool
P_7	***Tool***	Tool	Tool	NoTool	NoTool
P_8	***NoTool***	NoTool	NoTool	Tool	Tool

Completing Tasks: Providing the source code versions, specified in each task, participants were asked to (i) read each task, (ii) investigate whether or not the change scenario has occurred to the given class, and (iii) consequently agree or disagree with the task through selecting one of the provided options as "correct", "incorrect", or "don't know". Participants selected "don't know" when they were unable to precisely verify the task after the time limit was up (10 min for each task). For instance, to complete T_5, given in Table 3, participants were expected to evaluate the given change scenario for the specified class C, in the new version v_{i+1}. For this, participants potentially looked into the newer version of the source code, v_{i+1}, to identify whether or not the declaration of class C contains the phrase: *extends* C'. The presence of the phrase could result in participant's agreement with the task, while the phrase's absence could illustrate an incorrectly detected scenario by DejaVu and lead to participant's disagreement. As another example, to assess T_3, participants could potentially search for the main functionalities of two old classes, C' and C'', within the new class C in the newer version. While T_3 may require further investigation and additional inference in comparison to T_5, the readability, simplicity, and single-responsibility design of Cassandra's classes, as well as a rich body of developer's comments in the code, facilitated such investigations for participants.

4.2. Pilot study

Prior to our study, to validate the feasibility of completing tasks, we conducted a pilot study. As the pilot subject, we selected a senior researcher without any previous experience with DejaVu and Cassandra, but with several years of industry experience. During the pilot study, he provided feedback regarding the tool, as well as the stories and tasks. This, for example, included comments on minor usability issues, ambiguities during the briefing, and difficulty level of the assigned tasks. We used this feedback to update DejaVu and the study materials accordingly. The results of the pilot study led to the removal of tasks related to trace link maintenance, as we found that completing such tasks requires additional domain knowledge about the system. The rest of the tasks were found to be feasibly completed within the designated time. Moreover, we added additional descriptions regarding the tool, its capabilities, and how it can be used, to the initial briefing material. We further rephrased parts of the post-study questionnaire to address parts that were unclear or hard to understand for participants. Ultimately, due to time constraints, we reduced the number of scenarios from six to four to meet the requirement of a maximum study duration of 45 min to 1 h per participant. For the same reason, we limited the completion time to 10 min for each task. The pilot subject did not take part in the actual study.

4.2.1. Study process and data collection

- **Participants:** The adequacy of probabilistic sample size is determined based on a concept called sample *"saturation"*, the point at which no new information is observed in the data.

However, estimating the sufficient number of participants upfront or waiting to reach saturation in empirical studies with non-probabilistic data is not possible. Thus, multiple research groups conducted studies to systematically document the degree of data saturation and variability of various sample sizes. One group found that saturation occurred within the first twelve interviews, although basic elements for meta-themes (patterns that reflect the meaning behind a set of similar insights, learnings, or data points) were present as early as six interviews, leading to 80% of thematic saturation. Variability within the data followed similar patterns (Guest et al., 2006). Another study, recommends six to eight interviews for homogeneous samples and proposes to focus on the heterogeneity of the samples with respect to research objectives instead of the participants' number (Kuzel, 1999). Commonly, between 3 to 10 subjects are recommended for studying the logic of participants' (Dukes, 1984; Riemen, 1986).
We, therefore, recruited eight participants to take part in our study. We carefully selected our participants to be representative, and thus, sufficient for drawing conclusions. The participants included three full-time Computer Science (CS) faculties, four CS Ph.D. students, and one Senior Professional in industry. A summary of our participants' background is represented in Table 6. Our participants had an average of 11 years of experience in programming, five out of the eight participants had at least one year of experience in industry. Participants mentioned a number of different programming languages they were familiar with, while all had expertise in Java and were familiar with Eclipse, to be able to understand the examples in our study. We systematically followed the process below for each individual subject.

- **Briefing:** At the start of each session, the participant was given an approximate 10 min tutorial, when we provided background information about TLE, different types of change scenarios, objectives of the study, and a short demo of the tool and its functionalities. We first answered all questions related to the tool, the scenarios, and tasks before proceeding with the study.
- **Tasks:** Each participant received four stories to assess.[3] Table 7 provides an overview of the stories and their tasks, assigned to the participants. Each task was completed under both conditions, with and without using DejaVu:
  Tasks Completed with DejaVu: For each story to be evaluated with DejaVu, the participant selected the class name associated with the story in the drop-down menu, Fig. 3-①. DejaVu scenario selection console is shown in better resolution in Fig. 4. In doing so, an evolutionary graph displayed change scenarios across the existing versions in which the aforementioned class participated, Fig. 3-②. Inspecting a change scenario (described in a task) participant selected the class in the evolutionary graph and clicked on DejaVu-provided feature "Inspect Selected Element", Fig. 3-③. This action opened the class-corresponding source code in the relevant version within the Eclipse environment. Users inspected the change further by selecting to view the associated GitHub commits, Fig. 3-④, and Jira issues, Fig. 3-⑤. An example of a Jira Issue is depicted in Fig. 5.
  Tasks Completed without DejaVu Tool-support: In this case, the participant completed the same tasks only using standard features provided by the Eclipse IDE, as well as any other external tool she often uses in her daily activities. Similarly, here, we imported the versions of the source code, which were required to investigate the change scenarios, in Eclipse. We additionally provided participants with a list of Cassandra's features in an Excel sheet. We ensured the participants, in both conditions, had access to the same information, while the only variable was that DejaVu was additionally provided. Users were asked to evaluate the tasks in the same way they do in their daily activities. We observed users assessed tasks through searching for the participating classes in the relevant version. We precisely logged the completion time of tasks for each participant.
- **Data Collection:** During the study, we asked participants to "think-aloud", i.e., say what they were doing or aiming to do, comment when they do not know how to proceed, and comment on what they liked/didn't like. The moderator took notes collecting information regarding participants' interaction with DejaVu and the IDE, and participants' comments. We encouraged participants to ask questions and make comments and upon which, we paused the timer if they were raised during the process of completing tasks.
  When the activity of evaluating stories was completed, we first collected participant's task sheets, containing their "correct", "incorrect", or "don't know" answers to each of the ten given tasks along with their completion time. As such, we collected a total of 80 data points for both conditions, with and without DejaVu (10 tasks per participant).
  Participants were then provided with a questionnaire including detailed questions regarding: (i) their demographic and background, (ii) alternative tools they may have used for change comprehension, (iii) additional features they need to improve their experience with DejaVu, and finally (iv) tool usefulness in better understanding the code, changes, and rationales behind the change. Later, the moderator performed semi-structured interviews on the utility and usability of DejaVu to receive direct feedback from the participants. Finally, both types of the collected data, qualitative (questionnaire) and quantitative (time and correctness), were further analyzed to draw empirical conclusions based on the experience of our human sub-population with DejaVu. A summary of our findings and participants' comments is provided in Section 5.

[3] Due to the limited space not all stories are provided in the paper. All stories and descriptions are available at https://github.com/jku-win-se/dejavu/tree/main/studymaterial.

Table 6
Participants' experience: years of programming (Prog.); years in industry (Inds.)

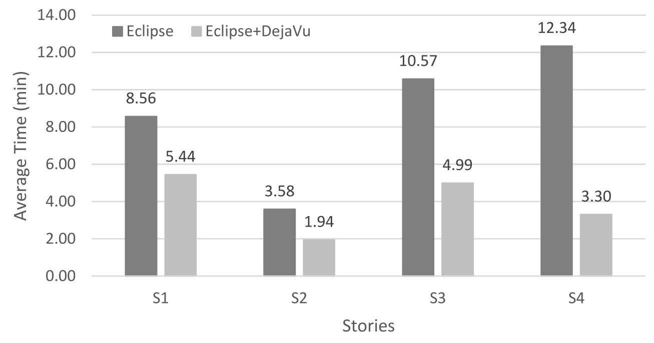
ID	Role	Yr. Prog.	Yr. Inds.	Main Prog. Languages
P1	Researcher	15	0	*Java*, C++, Python
P2	Researcher	8	0	*Java*, JavaScript, Python
P3	PhD Student	4	2	*Java*, Python
P4	Researcher	20	1	*Java*, JavaScript, Python
P5	PhD Student	8	3	*Java*, Python
P6	Senior software Eng.	15	3	*Java*, Python, C++
P7	PhD Student	9	0	*Java*, C++
P8	PhD Student	10	1	*Java*, C#

4.3. Results

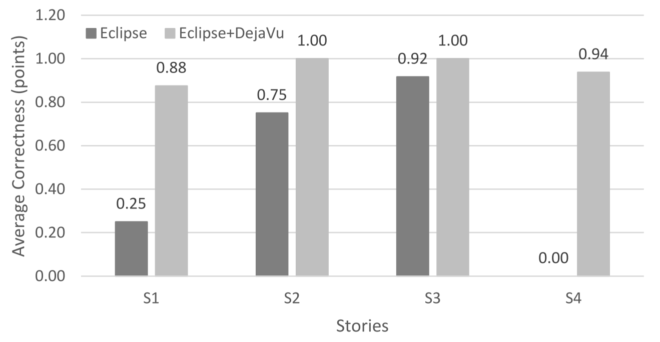
The entire sessions, including briefing, investigating all tasks, filling the questionnaire, and the semi-formal interview on average took between 45–60 min. Two studies went 15 min over one hour as the questions, comments, and post-study interview took longer than anticipated. After all studies were completed, we used the collected data to answer our two research questions.
RQ1 – (Time): After collecting the time, spent on each task, we created a corresponding "results matrix" of participants and tasks. For summarizing purposes, we then computed the completion time for each individual story and normalized the total completion time by the number of participants. Fig. 6(a) shows

Table 7
Distribution of *Tasks* and their associated *Change Scenarios* (CS) across *Stories* (S).

	Task	CS_1	CS_2	CS_4	CS_5	CS_6	CS_8	CS_9
S1	1.1		●					
	1.2	●						
S2	2.1				●			
S3	3.1					●		
	3.2						●	
	3.3				●			
S4	4.1					●		
	4.2							●
	4.3							●
	4.4			●				



(a) Average completion time of stories per participant.



(b) Figure 5: Average correctness of stories per participant

Fig. 6. The results of the study for participants with and without the DejaVu.

the average time each participant spent on completing each story, with and without using DejaVu, while Table 8 provides statistics of the collected data. Without the tool, it took participants on average 8.76 min to complete a single story, while using DejaVu this time was significantly reduced to 3.91 min. In cases where participants completed tasks without using the tool, the extra time was often spent on activities such as manually searching for changed classes in the appropriate version package, locating the relevant features in the excel sheet, and investigating changes further through searching within Jira and GitHub repositories on the web. More details in this regard are discussed in Section 5. As shown in Fig. 6(a), for half of the stories, S_3 and S_4, completion time was reduced over 50% (73% for S_4 and 52% for S_3). For the other half, S_1 and S_2, the time decreased over 35% (45% for S_2 and 36% for S_1). As expected, the reduction in time is more significant in larger stories (stories with more tasks), such as S_3 and S_4 with respectively three and four tasks.

RQ2 – (Correctness): Prior to the study, we created an "answer set". We have scrutinized Cassandra's release notes and commit

Table 8
Statistics of the results.

	RQ_1		RQ_2	
	Tool	*NoTool*	*Tool*	*NoTool*
Min.	3.58	1.94	0.00	0.88
Max.	12.34	5.44	0.92	1.00
Mean	8.76	3.91	0.47	0.95
Stdv.	3.27	1.39	0.36	0.05
Q_1	7.31	2.95	0.18	0.92
Median	9.57	4.14	0.50	0.97
Q_3	11.01	4.18	0.79	1.00

logs to determine the exact relationships between the changed classes in DejaVu-identified change scenarios. The "answer set" distinguishes between change scenarios that in fact happened in the system and those which were incorrectly detected by DejaVu. This means that the mistakenly-detected scenarios were corrected in the answer set, which is being referred to as the ground truth. Further, using our collected data from the empirical study, we created an initial "results matrix", for participants and tasks. In the "results matrix", the *incorrect* and *don't know* answers were assigned *zero* points, while *one* point was assigned if the answer was correct, meaning that participant's answer matched the "answer set". We then normalized the total points for each story by its size (number of tasks). These results are reflected in Fig. 6(b), showing that correct answers to S_1 improved from 25% to 88%. In case of S_2 and S_3, correctness raised to 100% from previously 75% and 92%. While the improvement of S_3 correctness is not remarkable, the S_3 completing time is significantly reduced. As shown, no participant was able to correctly verify the consisting tasks of S_4 without the use of the tool, due to the complexity of tasks in this story. Using DejaVu, we observed 100% improvement in correct answers to our largest story, S_4. In total, using DejaVu has increased the average correctness of answers from 47% to 95% over all four stories.

Finally, we twice ran the parametric Student's t-test with a significance level of 0.05. First, to assess whether the improvement in completion time is significant (using time as the dependent variable). Second, to verify the significant improvement of correctness (using correctness as the dependent variable). We selected these statistical tests because our dataset completion time and correctness were normally distributed and had equal variances in both of our populations (using no Tool-support and using DejaVu). The *p*-value of 0.01 (time) and 0.002 (correctness) confirms the statistically significant improvement of the results. In conclusion, we can reject our null hypotheses: DejaVu did not significantly accelerate (RQ_1) or increased comprehension (RQ_2) of the evolution.

5. Discussion

To investigate the usefulness of DejaVu for understanding source code change, we asked participants about their experience with similar tools or techniques which support their daily work. While various source code visualization approaches have been developed (Van Rysselberghe and Demeyer, 2004; Yoon et al., 2013; Telea and Auber, 2008), all our participants stated that they have not used, and are not aware of any other tools for better visualizing changes in the source code. All participants mentioned that they are primarily using the capabilities provided by their development environments (such as Eclipse or IntelliJ for Java). This includes, for example, visualizing differences between versions of source code, class hierarchies, and dependencies. One participant, for example, mentioned: *"I'm mostly using what is provided by the IDE, [...] using git-diffing to look at changes"*. This, however, is typically limited to a certain source of information, e.g., Git, to

Table 9
Usefulness in What, How, Why changes happened: ⊙ helpful, + very, − not.

Part.:	Code *(what)*	Changes *(how)*	Rationale *(why)*
P_1	⊙	+	⊙
P_2	⊙	+	⊙
P_3	⊙	+	⊙
P_4	−	+	⊙
P_5	+	+	⊙
P_6	⊙	⊙	+
P_7	−	+	⊙
P_8	⊙	+	⊙

inspect the history of changes, and compare different versions. While a few IDEs provide some limited form of visualization (e.g., line by line comparison and highlighting changes in color), the feature is often only usable for comparing two versions. For the tasks completed without DejaVu in several cases, participants spent additional time searching in the wrong package. One participant specifically stated: *"paying attention to multiple version numbers was confusing"*. For instance, to investigate a change that happened to class C between two subsequent versions v_i and v_{i+1} they needed to investigate the same class in both versions. Another change scenario in version v_{i+2}, for instance, requires them to open the third version of the package and search for participating classes. Thus, switching between multiple versions became confusing, especially in stories with a larger number of tasks. On the other hand, while using DejaVu the proper class in the corresponding version opens only by selecting the class in the graph. Therefore users do not need to search for the version numbers. In case of manual tasks, we also observed users occasionally searching through the web for relevant GitHub commits or Jira issues to better understand change scenarios.

We asked participants to rate the usefulness of DejaVu on a three-level Likert scale (very helpful, helpful, not helpful) with regards to understanding three aspects: (a) the code relevant to a particular change (*what* has changed?); (b) the change itself (*how* have things changed?); and (c) the rationale behind the change (*why* have things changed?). The results are shown in Table 9. The post-interviews were not only meant to assess the usefulness of DejaVu but to also facilitate a subsequent discussion regarding further positive and negative aspects of the visualization and potential improvements that participants would like to see to consider using the tool in their daily work.

While almost all participants (seven out of eight) acknowledged that DejaVu was "very helpful" for understanding the changes (*how*) and "helpful" for understanding *why* certain code classes changed, they did not have consensus on the usefulness of DejaVu in terms of understanding *what* has changed. In this case, only one participant called DejaVu "very helpful", while five called it "helpful" and two referred to it as "not helpful".

• **What?** The participants mentioned, that in order to understand *what* parts have changed, they would need additional information, beyond what was shown in the visualization. One participant mentioned that DejaVu depicts the *change on an abstract level, which is ok*, but in order to fully see what has changed *"more details, like line-based changes, for example, as a pop-up in addition to the classes in the graph"* or *"showing the name(s) of the method(s) that have changed"* would be beneficial. While DejaVu does provide a direct link to the respective class affected by the changing scenario and further statement-level changes in the GitHub repository, we observed that participants found it hard to make use of this information as it was shown in a different window, while the attention was on the graph-based visualization.

• **How?** For the second aspect, understanding *how* parts of the code have changed, seven out of eight participants found DejaVu to be very "helpful". Participants particularly liked the change scenarios that were clearly shown in the graph and the visualization of various changes over time. They found it easy to follow the graph and see how things have changed.

• **Why?** Finally, with regards to *why* code classes have changed, once again the majority of participants found DejaVu to be "helpful" (one *very helpful* and 7 *helpful*). While we used the integration of GitHub and Jira, links were not particularly emphasized in our evaluation, to avoid creating extra complexity in *stories*, these features were appreciated by the participants. Similar to the first aspect, participants mentioned additional details, directly shown in the graph, would be helpful. Two participants specifically mentioned the direct integration of GitHub commits info into the graph as one potential improvement (instead of separately opening the commit in the web browser). *"An additional info dialog would be nice"* (displaying GitHub information directly within the graph view).

In general, what we observed was that having the visualization and additional information DejaVu provided, participants had a clear starting point for answering their questions related to change scenarios. With the prototype implementation of DejaVu users had an easy way to get a quick high-level overview of the changes and dig deeper as needed. This was also confirmed by our results from the study (cf. Section 4), when using DejaVu, participants performed their tasks significantly better in a shorter amount of time. Based on our usability questionnaires and our discussions with participants, we came up with three major areas of improvement for the tool to further increase its usefulness.

• **Improve the visual appearance of Change Scenarios:** Participants mentioned that it could become challenging to easily distinguish the different change scenarios presented in the evolutionary graph, once the scenario gets larger. Especially with multiple changes across several versions, the image could become rather crowded. With respect to this, one participant explicitly mentioned that while the change scenarios are helpful some *additional color-coding similar to diff-tools* would make it easier to understand various changes.

• **Provide additional details within the graph:** While participants found the scenarios visualization in the graph to be a good starting point for exploring changes, a few demanded a more detailed level of information in addition to the view provided. Therefore, one implication on usefulness is to *provide additional details upon request directly in the graph*. This includes direct links to respective lines in the code that have changed, a direct comparison (visualized source-diffs) between different versions of the classes (e.g., before and after the change), or integrating additional scenarios on the method level. To avoid overcrowding the image, the additional information will be provided upon user's request.

• **Tighter integration within the tool:** Finally, our participants recommended a more interconnected view between the graph visualization and additional information. While information such as feature descriptions and GitHub details were shown in a separate window, their usage and relation to the respective class or change scenario were not always obvious. We, therefore, aim to directly incorporate this information inside the graph and link them directly to the respective change-related element.

In conclusion, DejaVu was deemed helpful by all the participants and the majority provided helpful comments and suggestions for further improvements.

6. Threats to validity

A threat to *construct validity* is the potential bias caused by the system we used for this study. We selected Apache Casandra as this was open source, publicly available, Java-based system.

While participants were not familiar with the system, all eight participants have several years of experience programming with Java which was important in order to understand the code and the changes performed in the different versions. A further threat stems from the fact that we only used a single system for the study. This stems from the lack of publicly available open-source datasets that contain sufficient information (source code, multiple versions, trace links, etc.) for inferring (and visualizing) *change scenarios* across multiple different versions. While we only used one system, we were still able to identify several different change scenarios in this system that were used for the study (cf. selection criteria). Another threat to validity is related to the "answer sets" that we used for evaluating *change scenarios* in Cassandra. To mitigate this, we only selected *change scenarios*, supported by evidence extracted from Cassandra's documentation and commit logs. External vetting was not possible, because open-source developers were not keen to undertake the task of validating *change scenarios*.

Regarding *external validity*, we selected a representative system in terms of size and complexity which is actively maintained and developed by the open source community. While we cannot claim generalizability for different programming languages and technologies, we are confident that the approach is suitable for other Java-based systems as well.

There are also threats to *internal validity* meaning that the results might have been influenced by our treatment. Subjects were selected based on their experience. While the number of subjects may seem relatively small, we aimed to include participants with extensive programming and industry experience. More than half of the participants had industry experience and all participants had several years of experience in working with Java-based software systems.

Regarding *conclusion validity*, some results and findings are not based on statistical information but on qualitative data. However, we have shown that for both RQ1 and RQ2 there is a statistically significant improvement in both, time spent to complete a task, and the correctness of the answers.

7. Related work

We divided the related work into two categories, those supporting software changes (i) standalone, and (ii) requiring a specific development environment:

• **Standalone:** The simplest form of change support includes repositories for source code control such as Concurrent Versions System (CVS), Subversion, and Git, which only highlight changes on statement-level granularity, and lack a graphical representation and high-level view of the change. In addition, repositories store code only when developers commit their changes, thus missing the intermediate changes performed locally by developers. In a study software visualization techniques were used to recover software evolution from code repositories (Lanza, 2001). However, the focus was only on visualizing the evolution of classes without reasoning about the change or integrating additional change information, e.g., change authors, issues, and commit messages related to the change.

Several standalone tools provide support for higher-level changes. For instance, Holt et al. improve visualization through color-coding changes in architectural structures such as new and deleted parts of a software (Holt and Pak, 1996). At the system level, Steinbrückner and Lewerentz (2010) captures and compares the system's structure in different versions, while Grisworld et al. argue that using a map metaphor alleviates the burden of managing cross-cutting changes across the system (Griswold et al., 2001). However, these studies leave out support for detailed changes. In a research, authors visualize class hierarchies although the visualization is limited to changes to inheritance hierarchies and the distribution of change across the classes of a hierarchy (Gîrba et al., 2005). A tool, *Code flows*, visualizes structural changes of C++ systems at multiple levels, however it does not provide additional change information (Telea and Auber, 2008).

• **Environment-dependent:** Several work characterized changes through monitoring a project development environment, requiring special setups for developers to perform their daily tasks which can become inconvenient. For instance, one study characterized change through monitoring a project for low-level changes (Robbes and Lanza, 2007). In a different study, authors implemented a framework for collaborative development, called Syde, which monitors runtime changes, and store them in an internal repository in form of Abstract Syntax Trees (Hattori and Lanza, 2010). Further, the same authors developed Replay, an Eclipse plug-in, allowing developers to explore Syde's repository of monitored changes (Hattori et al., 2011, 2013). We found multiple studies which particularly tailored their approach to specific environments. For instance, authors integrated static, dynamic, evolutionary, and plug-in information into a framework, called Ferret, to answer a set of predefined *conceptual queries* about a program (De Alwis and Murphy, 2008). However, in their empirical study participants were unable to use the evolution-related queries, due to compatibility issues. Many other researchers showed how using data generated during the programming activity and merging information from heterogeneous sources can provide valuable information about the evolution of a project. For instance, Alshakouri et al. visualize product artifacts within software development process however the tool is tailored for their research problem (Alshakhouri et al., 2018). Another work proposes to extend IDEs with a (Twitter-like) micro-blogging facility in order to make program comprehension knowledge accessible (Guzzi et al., 2010). While the implementation is not yet available, Fernandez et al. introduce a visualization tool for Git commits that relates the code authorship to commit frequency. However, the tool does not provide support for change visualization (Fernandez and Bergel, 2018). Another work retrieves information from multiple sources to provide a high-level view of a system and color-codes components according to the type of the change. However, types of source code changes are not identified and no further information about the change is provided (Dal Sasso et al., 2015). There are multiple plugins that augment CVS data with different forms of visualization. For instance, CVSscan supports line-oriented displays (Voinea et al., 2005); GEVOL displays Java programs stored within a CVS using a temporal graph visualizer (Collberg et al., 2003); REFVIS prototype visualizes a few refactoring patterns in CVS repositories but does not provide any additional information (Gorg and Weißgerber, 2005); and Visual Code Navigator provides three views of CVS code at a different level of detail. In the *evolution* view, which is designed to display the changes of the code, changes are color-coded for each developer (Lommerse et al., 2005).

Clonecompass is more recently developed, focusing on the analysis of performance metrics at different granularity levels of multiple versions of a software system. *Kam1n0* is another assembly code clone-based search engine to support the analysis of vulnerabilities in binaries (Wang et al., 2019). Another visualization tool uses a matrix layout for the analysis of run time metrics and source code changes, e.g., execution graphs (Alcocer et al., 2019). Furthermore, two recent theses propose run time source code visualization, both requiring several execution environments to monitor and record the changes (Han, 2021; Hock and Nakayama, 2021).

There are several recent research approaches that analyze and display the code change histories, while their primary objective is to exploit the visualization for feature and bug localization tasks (Razzaq et al., 2018; Qayum et al., 2022). A few used machine learning approaches and provided graph-based visualization of the code changes. For instance, Zhang et al. (2020) proposed a deep learning-based bug localization model, namely KGBugLocator, which consists of a knowledge graph and NLP-based search methods to extract information from the graph. Similarly, another work proposes a graph-based visualization within a platform (a compiler), namely ProgQuery, allowing developers to write a modified program for analysis. The tool then computes a few syntactic and semantic links between the code elements in order to visualize the code in a graph, and extract different types of knowledge/features from the graph (Rodriguez-Prieto et al., 2020; Ortin et al., 2020).

In comparison, DejaVu does not require environment monitoring, infers high-level changes, provides request-based access to lower-level changes, and retrieves further change-related information from external repositories.

8. Conclusion

We presented DejaVu for multi-level visualization of source code changes, which in turn, provides developers with a better understanding of software evolution. DejaVu generates an evolutionary graph representing change patterns that occurred to a specific class across subsequent versions of a software system. The patterns are further augmented with additional information extracted from GitHub and Jira, providing a holistic view of a system's changes. In an empirical study using DejaVu improved the correctness and completion time of several tasks by more than 50%. We plan to further improve the visual appearances of DejaVu, such as improving the visualization of larger graphs, providing additional change information, and developing extra features, e.g., feature to focus and zoom into certain areas to provide detailed descriptions of certain changes when requested. Furthermore, we will integrate other tools in DejaVu to collect and provide additional information for certain change scenarios.

CRediT authorship contribution statement

Mona Rahimi: Conceptualization, Methodology, Software, Visualization, Investigation, Writing – original draft. **Michael Vierhauser:** Software, Investigation, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Alcocer, J.P.S., Beck, F., Bergel, A., 2019. Performance evolution matrix: Visualizing performance variations along software versions. In: Proc. of the 2019 Working Conference on Software Visualization. IEEE, pp. 1–11.

Alshakhouri, M., Buchan, J., MacDonell, S.G., 2018. Synchronised visualisation of software process and product artefacts: Concept, design and prototype implementation. Inf. Softw. Technol. 98, 131–145.

Apache Cassandra Database, 2002. https://cassandra.apache.org, [Online; accessed Mai-01-2022].

Bohnet, J., Döllner, J., 2006. Visual exploration of function call graphs for feature location in complex software systems. In: Proc. of the 2006 ACM Symposium on Software Visualization. ACM, pp. 95–104.

Collberg, C., Kobourov, S., Nagra, J., Pitts, J., Wampler, K., 2003. A system for graph-based visualization of the evolution of software. In: Proc. of the 2003 ACM Symposium on Software Visualization. ACM, pp. 77–ff.

Dal Sasso, T., Minelli, R., Mocci, A., Lanza, M., 2015. Blended, not stirred: Multi-concern visualization of large software systems. In: Proc. of the 3rd IEEE Working Conference on Software Visualization. IEEE, pp. 106–115.

De Alwis, B., Murphy, G.C., 2008. Answering conceptual queries with Ferret. In: Proc. of the 30th International Conference on Software Engineering. ACM, pp. 21–30.

Diehl, S., 2007. Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software. Springer Science.

Dit, B., Revelle, M., Poshyvanyk, D., 2013. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. Empir. Softw. Eng. 18 (2), 277–309.

Dukes, S., 1984. Phenomenological methodology in the human sciences. J. Religion Health 23 (3), 197–203.

Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G., 2001. Graphviz - open source graph drawing tools. In: Proc. of the 2001 International Symposium on Graph Drawing. Springer, pp. 483–484.

Fernandez, A., Bergel, A., 2018. A domain-specific language to visualize software evolution. Inf. Softw. Technol. 98, 118–130.

Garcia, J., Krka, I., Mattmann, C., Medvidovic, N., 2013. Obtaining ground-truth software architectures. In: Proc. of the 35th International Conference on Software Engineering. IEEE, pp. 901–910.

Gîrba, T., Lanza, M., Ducasse, S., 2005. Characterizing the evolution of class hierarchies. In: Proc. of the 9th European Conference on Software Maintenance and Reengineering. IEEE, pp. 2–11.

Gorg, C., Weißgerber, P., 2005. Detecting and visualizing refactorings from software archives. In: Proc. of the 13th International Workshop on Program Comprehension. IEEE, pp. 205–214.

2022. Graphviz - graph visualization software. https://graphviz.org, [Online; accessed Mai-01-2022].

Griswold, W.G., Yuan, J.J., Kato, Y., 2001. Exploiting the map metaphor in a tool for software evolution. In: Proc.of the 23rd International Conference on Software Engineering. IEEE, pp. 265–274.

Guest, G., Bunce, A., Johnson, L., 2006. How many interviews are enough?: An experiment with data saturation and variability. Field Methods 18 (1), 59–82.

Guzzi, A., Pinzger, M., Van Deursen, A., 2010. Combining micro-blogging and IDE interactions to support developers in their quests. In: Proc. of the 2010 IEEE International Conference on Software Maintenance. IEEE, pp. 1–5.

Han, Z., 2021. Real-Time Software Execution Visualization: Design and Implementation (Dissertation). University of California, Irvine.

Hattori, L., D'Ambros, M., Lanza, M., Lungu, M., 2011. Software evolution comprehension: Replay to the rescue. In: Proc. of the 19th International Conference on Program Comprehension. IEEE, pp. 161–170.

Hattori, L., D'Ambros, M., Lanza, M., Lungu, M., 2013. Answering software evolution questions: An empirical evaluation. Inf. Softw. Technol. 55 (4), 755–775.

Hattori, L., Lanza, M., 2010. Syde: A tool for collaborative software development. In: Proc. of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. ACM, pp. 235–238.

Hock, P., Nakayama, K., 2021. Proposal and Development of a Source Code Visualization Tool to Present Code Entities as Interconnected Windows on Demand to Improve Source Code Comprehension (Thesis).

Holt, R., Pak, J.Y., 1996. GASE: visualizing software evolution-in-the-large. In: Proc. of the 4th Working Conference on Reverse Engineering. IEEE, pp. 163–167.

JGit, 2022. https://www.eclipse.org/jgit, [Online; accessed Mai-01-2022].

Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K., Rosenberg, J., 2002. Preliminary guidelines for empirical research in software engineering. IEEE Trans. Softw. Eng. 28 (8), 721–734.

Ko, A.J., Latoza, T.D., Burnett, M.M., 2015. A practical guide to controlled experiments of software engineering tools with human participants. Empir. Softw. Eng. 20 (1), 110–141.

Kuzel, A., 1999. Doing Qualitative Research, second ed. (Chapter 2).

Lanza, M., 2001. The evolution matrix: Recovering software evolution using software visualization techniques. In: Proc. of the 4th International Workshop on Principles of Software Evolution. ACM, pp. 37–42.

Lehman, M.M., 1996. Laws of software evolution revisited. In: European Workshop on Software Process Technology. Springer, pp. 108–124.

Lommerse, G., Nossin, F., Voinea, L., Telea, A., 2005. The visual code navigator: An interactive toolset for source code investigation. In: Proc. of the 2005 IEEE Symposium on Information Visualization. IEEE, pp. 24–31.

Maalej, W., Tiarks, R., Roehm, T., Koschke, R., 2014. On the comprehension of program comprehension. ACM Trans. Softw. Eng. Methodol. 23 (4), 1–37.

McBurney, P.W., McMillan, C., 2016. An empirical study of the textual similarity between source code and source code summaries. Empir. Softw. Eng. 21 (1), 17–42.

Murphy, G.C., Kersten, M., Findlater, L., 2006. How are java software developers using the eclipse IDE? IEEE Softw. 23 (4), 76–83.

Ortin, F., Rodriguez-Prieto, O., Pascual, N., Garcia, M., 2020. Heterogeneous tree structure classification to label java programmers according to their expertise level. Future Gener. Comput. Syst. 105, 380–394.

Polkinghorne, D.E., 2005. Language and meaning: Data collection in qualitative research. J. Couns. Psychol. 52 (2), 137.

Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., Rajlich, V., 2007. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. Trans. Softw. Eng. 33 (6), 420–432.

Prete, K., Rachatasumrit, N., Sudan, N., Kim, M., 2010. Template-based reconstruction of complex refactorings. In: Proc. of the 2010 IEEE International Conference on Software Maintenance. IEEE, pp. 1–10.

Qayum, A., Khan, S.U.R., Akhunzada, A., et al., 2022. FineCodeAnalyzer: Multi-perspective source code analysis support for software developer through fine-granular level interactive code visualization. IEEE Access 10, 20496–20513.

Rahimi, M., Cleland-Huang, J., 2015. Patterns of co-evolution between requirements and source code. In: Fifth International Workshop on Requirements Patterns. IEEE, pp. 25–31.

Rahimi, M., Cleland-Huang, J., 2018. Evolving software trace links between requirements and source code. Empir. Softw. Eng. 23 (4), 2198–2231.

Rahimi, M., Goss, W., Cleland-Huang, J., 2016. Evolving requirements-to-code trace links across versions of a software system. In: Proc. of the International Conference on Software Maintenance and Evolution. IEEE, pp. 99–109.

Razzaq, A., Wasala, A., Exton, C., Buckley, J., 2018. The state of empirical evaluation in static feature location. ACM Trans. Softw. Eng. Methodol. 28 (1), 1–58.

Riemen, D.J., 1986. The essential structure of a caring interaction: Doing phenomenology. Nurs. Res. Qual. Perspect. 85–105.

Riesco, M., Fondón, M.D., Alvarez, D., 2009. Using graphviz as a low-cost option to facilitate the understanding of unix process system calls. Electron. Notes Theor. Comput. Sci. 224, 89–95.

Robbes, R., Lanza, M., 2007. Characterizing and understanding development sessions. In: Proc. of the 15th IEEE International Conference on Program Comprehension. IEEE, pp. 155–166.

Rodriguez-Prieto, O., Mycroft, A., Ortin, F., 2020. An efficient and scalable platform for java source code analysis using overlaid graph representations. IEEE Access 8, 72239–72260.

Rugaber, S., 1992. Program comprehension for reverse engineering. In: Proc. of the 1992 AAAI Workshop on AI and Automated Program Understanding. pp. 106–110.

Runeson, P., Host, M., Rainer, A., Regnell, B., 2012. Case Study Research in Software Engineering: Guidelines and Examples. John Wiley & Sons.

Saito, S., Iimura, Y., Tashiro, H., Massey, A.K., Antón, A.I., 2016. Visualizing the effects of requirements evolution. In: Proc. of the 38th International Conference on Software Engineering Companion. ACM, pp. 152–161.

Seaman, C.B., 1999. Qualitative methods in empirical studies of software engineering. IEEE Trans. Softw. Eng. 25 (4), 557–572.

Shaw, P.D., Kennedy, J., Graham, M., Milne, I., Marshall, D.F., 2016. Visualizing Genetic Transmission Patterns in Plant Pedigrees (Ph.D. thesis). Citeseer.

Steinbrückner, F., Lewerentz, C., 2010. Representing development history in software cities. In: Proc. of the 5th International Symposium on Software Visualization. ACM, pp. 193–202.

Telea, A., Auber, D., 2008. Code flows: Visualizing structural evolution of source code. Comput. Graph. Forum 27 (3), 831–838.

Van Rysselberghe, F., Demeyer, S., 2004. Studying software evolution information by visualizing the change history. In: Proc. of the 20th International Conference on Software Maintenance. IEEE, pp. 328–337.

Voinea, L., Telea, A., Van Wijk, J.J., 2005. CVSscan: visualization of code evolution. In: Proc. of the 2005 ACM Symposium on Software Visualization. pp. 47–56.

Wang, Y., Weatherston, J., Storey, M.-A., German, D., 2019. CloneCompass: Visualizations for exploring assembly code clone ecosystems. In: Proc. of the 2019 Working Conference on Software Visualization. IEEE, pp. 88–98.

Yoon, Y., Myers, B.A., Koo, S., 2013. Visualization of fine-grained code change history. In: Proc. of the 2013 IEEE Symposium on Visual Languages and Human Centric Computing. IEEE, pp. 119–126.

Zhang, J., Xie, R., Ye, W., Zhang, Y., Zhang, S., 2020. Exploiting code knowledge graph for bug localization via bi-directional attention. In: Proc. of the 28th International Conference on Program Comprehension. ACM, pp. 219–229.

Mona Rahimi is currently an assistant professor at the department of computer science at the Northern Illinois University. Rahimi received her Ph.D. from the University of Notre Dame and joined the University of Toronto for postdoctoral research. Her research interests fall in the intersection of software engineering (SE) and artificial intelligence (AI), including intelligent software engineering (AI for SE) and engineering intelligent software (SE for AI).

Michael Vierhauser is a postdoctoral researcher in the Secure and Correct Systems Lab, Johannes Kepler University Linz, Austria. Vierhauser received his Ph.D. in computer science from Johannes Kepler University Linz. His research interests include safety–critical and cyber–physical systems and runtime monitoring.