Exploranative Code Quality Documents

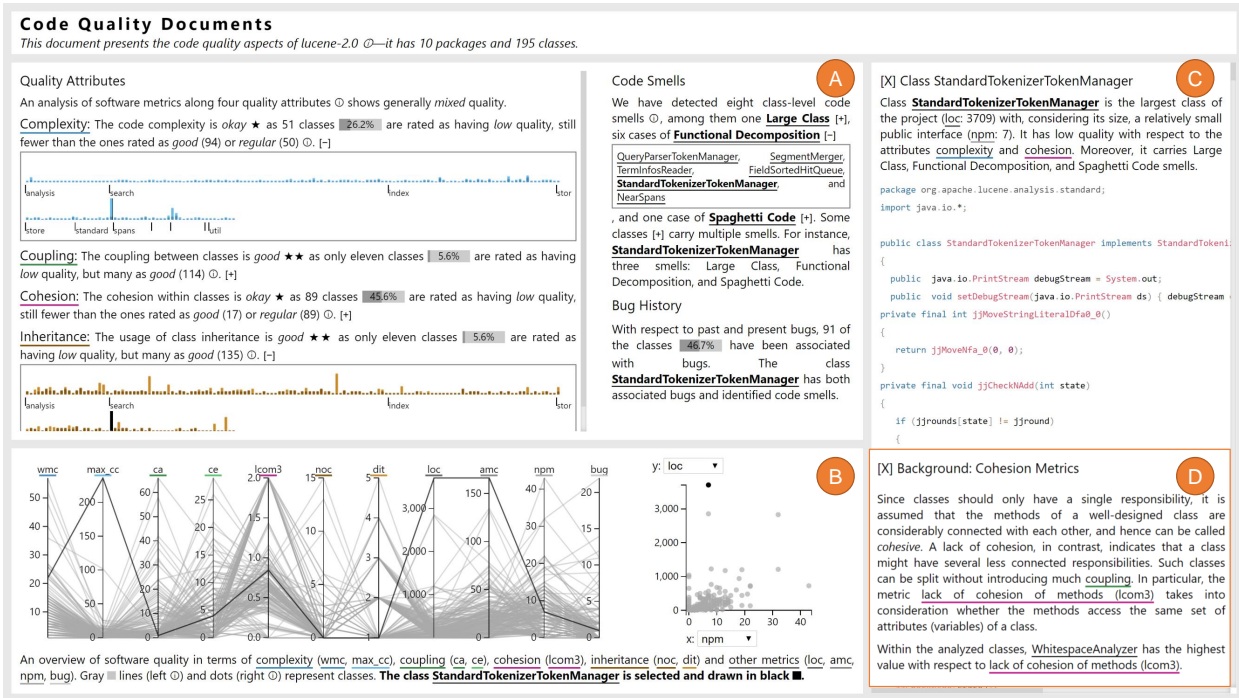Haris Mumtaz, Shahid Latif, Fabian Beck, and Daniel Weiskopf

Fig. 1. Exploranative code quality document for *Lucene 2.0*. Ⓐ Textual overview in terms of quality attributes, code smells, and bugs, which includes embedded visualizations. Ⓑ Overview visualizations: parallel coordinates plot and scatterplot. Ⓒ Source code of a class provided in the details view. Ⓓ Description of a quality attribute alternatively presented in the details view.

Abstract—Good code quality is a prerequisite for efficiently developing maintainable software. In this paper, we present a novel approach to generate *exploranative* (explanatory and exploratory) data-driven documents that report code quality in an interactive, exploratory environment. We employ a template-based natural language generation method to create textual explanations about the code quality, dependent on data from software metrics. The interactive document is enriched by different kinds of visualization, including parallel coordinates plots and scatterplots for data exploration and graphics embedded into text. We devise an interaction model that allows users to explore code quality with consistent linking between text and visualizations; through integrated explanatory text, users are taught background knowledge about code quality aspects. Our approach to interactive documents was developed in a design study process that included software engineering and visual analytics experts. Although the solution is specific to the software engineering scenario, we discuss how the concept could generalize to multivariate data and report lessons learned in a broader scope.

Index Terms—Code quality, interactive documents, natural language generation, sparklines

✦

1 INTRODUCTION

To create high-quality software, the stakeholders involved need to obtain information about code quality. Existing visualization approaches provide an overview of metrics related to code quality, but they often lack to put data into context. Findings related to different code quality aspects need to get connected, explanations might be required for stakeholders less experienced in software quality, and making transparent

- *H. Mumtaz and D. Weiskopf are with VISUS, University of Stuttgart, Germany. E-mail: {haris.mumtaz, daniel.weiskopf}@visus.uni-stuttgart.de.*
- *S. Latif and F. Beck are with paluno, University of Duisburg-Essen, Germany. E-mail: {shahid.latif, fabian.beck}@paluno.uni-due.de.*

the workings of analysis algorithms will increase trust into the quality assessment. In contrast, some tools integrated into the software development chain already make a step in this direction, however, consider different quality aspects separately and use only simple visualizations. Our goal is to go a step ahead and integrate visualizations with contextual information. We argue that textual descriptions are key to communicate context and guide the data exploration process. Additional textual explanations can make the interface fully self-explanatory, allow the integration of abstract information that is hard to explain visually, and facilitate blending domain terminology with data descriptions.

We suggest a visual analytics solution that combines text and visualization. The idea is to automatically generate textual descriptions that guide the users through the main findings of a code quality analysis. We augment the generated text with visualizations that also allow users to interactively explore different aspects of source code quality. With this, we adopt the ideas of *exploranation* (explanation and exploration) [58, 61] and *explorable explanations* [56] for a visual analytics

application in a professional context. We describe our underlying concept as a general approach to represent multivariate data before we tailor it in a design study process to the specific application of code quality analysis.

In the visual analytics system shown in Figure 1, we use a multi-view interface to present different perspectives on the data. The overview panels on the left (Figure 1 A and B) show code quality information in the form of text and visualizations. The main text (Figure 1 A) summarizes code quality attributes based on four important indicators (left) and discusses code smells and the bug history. We use visualizations embedded in the text (A: small icons and bars in line with text and bar charts between the text lines) as well as traditional visual representations of multivariate data (B: a parallel coordinates plot and a scatterplot) to report metric values for the classes of an object-oriented system. While the bar charts show a subset of related metrics per class (grouped by packages) in stacked bars, the parallel coordinates plot displays all metrics for all classes (i.e., each line represents a class); the scatterplot adds a different perspective for a user-selected pair of metrics. All visualizations are interactive and linked with each other to support further exploration. Details are provided on demand in the right panel (Figure 1 C and D). For instance, the user can learn about the specifics of a class (C: textual description and source code) or get background information about a specific code smell (D: explanation and example). The navigation between the views is supported by multi-directional *vis–text* interactions. This means that the text is interactively linked with the visualization and vice versa; for example, hovering a class in the text or in any visualization highlights the other representatives of the same class across all views. With these features, the system guides through the result of the code analysis and supports active exploration of the data. Overall, our paper provides the following main contributions:

- A general concept building on textual explanations, explorable visualizations, and consistent linking between them to represent multivariate data as interactive documents.

- A visual analytics system that allows users to learn and understand code quality aspects in an interactive environment driven by textual explanations and visual exploration.

- An interaction model that provides close integration of text and visualization by interactive linking.

- Lessons learned from our design study that could impact the design of data-driven interactive documents.

We have followed a design process involving four authors with a diverse background in software engineering, visual analytics, interactive documents, and text generation. We have also conducted a formative study with different external experts to evaluate and improve our system in an iterative manner. Our system is available as a web application at `https://vis-tools.paluno.uni-due.de/cqd`. We also provide the system as well as the material and data of our formative study as supplemental material [38].

2 RELATED WORK

Our overview of related work covers aspects of software engineering, visualization, and visual analytics, as well as interactive documents.

Code Quality – Code quality is multi-faceted and covers, for instance, testability, maintainability, and readability. To examine these aspects, certain quality attributes (e.g., coupling, complexity, size) are quantified by underlying software metrics; for instance, McCabe's software complexity metrics measure readability aspects of the code [33]. For object-oriented systems, a popular set of metrics is the CK suite introduced by Chidamber and Kemerer [15] and the QMOOD metrics (Quality Model for Object-Oriented Design) [8]. Many approaches employ such metrics suites to distinguish parts of the source code in terms of good, acceptable, or bad quality [18, 46] or to identify code smells (problematic properties and anti-patterns of the code) [40].

Software Metrics Visualization – Visualizations of software metrics have already been used to investigate source code quality—we

cannot report all approaches here but give a brief overview. For instance, Emden and Moonen [55] present a graph-based approach to automatically detect code smells and investigate how code smells can be integrated into a code inspection tool. Murphy-Hill and Black [39] integrate a software metrics visualization with a source code view. Erdemir et al. [17] provide a graph-based visualization tool encoding multivariate metrics in the node glyphs of the graph. Diverse software maps and cities [7, 13, 59] arrange the structure of a software project on a 2D map and visually encode software metrics on top of this structure (often in the form of 3D buildings). Mumtaz et al. [37] support the interactive detection of code smells using parallel coordinates plots and scatterplots. Also, several tools are available that build on visualizations to assist developers in analyzing code quality in terms of software metrics. For instance, SonarQube [5] controls and manages the code quality in several ways, such as continuous inspection and issue detection. The platform shows issues like code smells, bugs, etc. using lightweight visualizations. It also helps developers collaborating with a shared vision of code quality. There are some code analysis tools that accomplish similar tasks as SonarQube, for instance, Checkstyle [2] and PMD [4]. In contrast to these visualization approaches and tools, we focus on providing more context to the data and explain the findings and their background in detail. We are not aware of any approach that uses a sophisticated text generation approach for this purpose.

Embedded Visualization – Visualizations included into the lines or paragraphs of a text are known as *sparklines* [54], *word-sized* [12], or *word-scale* graphics [20]. They allow close and coherent integration of the textual and visual representations of data. Some approaches apply these in the context of software engineering and embed them into the code to assist developers in understanding a program. Harward et al. [21] and Sulír et al. [52] suggest augmenting the source code with visualizations to keep track of the state and properties of the code. Beck et al. [9, 10] implement embedded visualizations for understanding program behavior and performance bottlenecks. Similarly, Hoffswell et al. [22] and Swift et al. [53] augment source code with visualizations to aid understanding of runtime behavior. We embed visualizations into natural language text (not into source code) to support better understanding of the quality of the source code.

Natural Language Generation – Natural language generation allows us to automatically generate text from raw or pre-processed data [44]. There are plenty of approaches that focus on automated text generation [19]. Few solutions also deal with the combined generation of text and visualization. Such combinations have been applied, for instance, to operational or maintenance-related instructions for mechanical devices [57], explanatory captions of diagrams for information graphics [35], reports of brain imaging data [23], weather forecast reports using predictive data analysis [42, 50], evaluation of learning analytics in education [43], and author profiles of the visualization community [27]. Automated text generation has also been employed in the context of software engineering, for instance, for describing software models, such as class diagrams or use case diagrams, where they are augmented with textual descriptions [14, 29, 30, 34]. Even more related to our solution are those approaches that deal with textual reports for source code, such as code documentation and summarization [32, 36, 48], commit messages [16], or reports on the runtime behavior of the code [11]. However, none of these approaches discusses code quality. Also, the results are usually not presented as highly interactive documents that support the exploration of the data.

Text–vis Linking – The interactive linking of text and visualizations has only been explored to some extent. Beck and Weiskopf [12] propose an abstract interaction model for documents containing text, word-sized graphics, and regular visualizations; all three types of data representations are linked via brushing-and-linking. Latif et al. [28] describe an authoring solution for web documents to produce some of those interactions. Our interaction model also uses and extends the model by Beck and Weiskopf. Kim et al. [25] advocate for *text–text* linking to facilitate document reading. In their approach, the linking is supported between text in the main body and text in tables. Few of the systems that generate both text and visualization—for instance, *VIS Author Profiles* [27] and *interactive Map Reports* [26]—discuss

interactions, but still focus more on explanations and offer limited data exploration. *Voder* [49], in contrast, focuses more on interactions and supports the data exploration process by offering short descriptions about key findings in the data. However, it does not generate a comprehensive report with longer descriptions.

In summary, although existing approaches present source code information, they lack in putting data into context and providing explanations. None of the systems, also outside the software engineering community, supports exploranation as a process blending explanations and explorations in a way that we envision leveraging the interactive combination of textual and visual descriptions. We are inspired by the abstract idea of interactive linking of text and visualizations by Beck and Weiskopf [12] to support exploranation. We adopt CK, QMOOD, and McCabe's metrics (listed in Table 1) and use them in combination with pre-defined thresholds to analyze and present source code quality.

3 EXPLORANATIVE DOCUMENTS FOR MULTIVARIATE DATA

With our approach, we want to support *active reading* [6] of multivariate data. The readers should not be restricted to passively consume explanations but be facilitated to actively explore the data as well. They can read with a specific focus, break the content apart, and analyze its meaning. In contrast to traditional visual analytics interfaces, which focus on exploration, we provide more guidance to the users in the form of textual explanations, while—to some extent—preserving the explorative power of an interactive visualization. While our concrete implementation is tailored to software metrics and related code quality characteristics, our general approach applies to multivariate data in a broader sense and might even extend to wider classes of interactive documents. In this section, we introduce the generic concept, before the following sections present the details of the tailored solutions.

We build on ideas from Ynnerman et al.'s concept of *exploranation* [61] and Victor's *explorable explanations* [56]. Ynnerman et al. introduce *exploranation*—a coinage stemming from the terms *exploration* and *explanation*—for scientific communication, for instance, to show visitors of a museum a visualization of an ancient mummy, guide them to interesting aspects but also let them explore it; related earlier work by Weiskopf et al. [58] combines visual explanation and exploratory visual experiments for similar scenarios. We adopt this idea for a visual analytics scenario. Although our targeted professional audience is narrower, we also want to make the data easy to access. Some of Ynnerman et al.'s design principles also apply to our scenario, in particular, that *explorative microenvironments* blend with *signposted narratives*. With respect to the visual representations used, Victor's suggestions for *explorable explanations* are even closer to our work as they also focus on interactive documents. We employ *explorable examples* and *contextual information*, as two of three suggested categories of explanations.

Our approach of exploranative documents for multivariate data comprises the following building blocks.

(I) Textual Explanations – The main feature that distinguishes our approach from most other visual analytics solutions is the use and integration of automatically generated textual explanations. We discern different types of explanations: (i) **Data-driven explanations** summarize the data and results of data analysis (e.g., identification of patterns or clustering) while pointing to remarkable observations and giving examples. (ii) **Educational explanations** provide background on the domain concepts reflected in the data. (iii) **Methodological explanations** give details about how the analysis was performed and the reason why the system came to certain conclusions. While the data-driven explanations are the focus of the documents, the two other types provide important context. The users can obtain, on demand, for what parts of the data summary they require background information.

(II) Explorable Visualizations – In addition to the textual explanations, the explorative component is mostly contributed by interactive visualizations. (i) **Overview visualizations** should have a consistent location in the interface and be visible all the time. In the specific solution, we will build on parallel coordinate plots and scatterplots as such overview visualizations, but the approach is open to any appropriate visualization of multivariate data (e.g., scatterplot matrices,

tabular-like representations, multivariate glyphs, projection methods). (ii) **Embedded detail visualizations**, in contrast, enrich the text with further information and just show subsets or aspects of the data. There can be regular visualizations that scroll with the respective text, but also word-sized representations embedded in the text (*sparklines* [12, 54]). The better these visualization are integrated with the text, the easier it will be for users to explore them along reading the text. In general, the exploration process can happen visually by users deciding to look at and investigate certain elements of the visualization. On top of that, interactions to subselect the data and pick out individual elements further extend the users' abilities with respect to exploration.

(III) Consistent Linking – Like in a traditional multi-view visual analytics system, a challenge is to maintain a clear linkage between the different views. However, in our case, the linking becomes even more difficult because data is described on very different levels of abstraction and with different modalities (text and visualization). We apply the concept of (i) **vis–text interaction** [12], which suggests linking all three representations—textual, visual, and embedded visual ones—in a bidirectional way using hover and click interactions. For instance, hovering a word in a text, the related entities can be highlighted across all views and also in textual descriptions. (ii) **Consistent color coding** is used to further clarify relationships between the different textual and visual descriptions of related data. We suggest applying a consistent color coding of the different variables across all representations. Similar variables can be grouped by hue and get assigned a different brightness.

With these concepts, we support active reading and the exploranation process. The users are first confronted with a summary text and associated overview visualizations. One group of readers, especially first-time users, might follow mainly the provided narrative and start reading at the top left. Whenever something is unclear from the high-level summary, they can explore the required background on demand. After having read the main text, they might switch to exploring the data further using visualizations. In contrast, another group of users that might be more experienced could immediately start the data exploration process. While some information can be directly gained from the visualizations, for other insights occasionally reading the textual explanations can provide support. The textual summaries might also point them to interesting findings that they might have missed otherwise.

The resulting approach can be classified as a visual analytics solution that puts emphasis on presentation, storytelling, and dissemination. In terms of the sensemaking process described by Pirolli and Card [41], it covers the sensemaking loop (i.e., build a case, tell a story, search for support, and reevaluate) rather than the foraging loop (i.e., search and filter, read and extract, search for information, and search for relations).

4 CODE QUALITY ANALYSIS

In this section, we discuss the application scenario and explain the data processing required for it.

4.1 Targeted Users

The targeted users of our system are mainly the stakeholders who are concerned with source code and its quality. Being well-informed about the quality of source code can help them in taking steps to improve various aspects that need attention. The group includes mainly software developers who work with the code on a daily basis; they need to read and understand the code to be able to extend and maintain it. But the group also extends to testers, software architects, product owners, or project managers. For example, stakeholders like product owners and project managers can use our system to assess the overall quality of a project and use the gained information to prioritize quality concerns. In essence, this target user group is much wider than just code quality experts and highly experienced developers. For this reason, we believe that integrating guidance and explanations will provide valuable support for most of the users and allows them to draw actionable conclusions. However, just presenting a static report would not suffice because the prepared summaries and explanations can only be a starting point for investigating a detected problem in detail.

Table 1. Class-level software metrics (name and acronym) used for code quality analysis, grouped by quality attributes.

Quality Attribute	Software Metric	Acronym	Description
Complexity	Weighted methods per class	wmc	The sum of all method complexity values for a class.
	Maximum cyclomatic complexity	max_cc	The maximum of all the method-level complexity values of a class.
Coupling	Afferent coupling	ca	The number of other classes that depend on a class (incoming dependencies).
	Efferent coupling	ce	The number of other classes on which a class depends (outgoing dependencies).
Cohesion	Lack of cohesion of methods	lcom3	It checks whether the methods access the same set of variables of a class.
Inheritance	Depth of inheritance	dit	The inheritance levels for a class.
	Number of children	noc	The number of immediate descendants of a class.
Other	Average method complexity	amc	The average size of the methods in a class.
	Lines of code	loc	The total lines of code present in a class.
	Number of public methods	npm	The number of methods declared as public in a class.
	Number of bugs	bug	The number of bugs that have been associated with a class.

4.2 Data

Software metrics provide significant information on code quality. We employ software metrics belonging to object-oriented metrics known as Chidamber and Kemerer metrics suite [15], QMOOD metrics [8], and McCabe's complexity metric [33]. The metrics are at class-level abstraction; they quantify quality attributes of the classes of a software project. Specifically, we work with a subset of 11 metrics in total (Table 1). The selected metrics are often employed to measure quality attributes, for instance, the *coupling between objects* metric measures the degree of interdependence between classes, which is consistent with how coupling is traditionally defined. Excessive coupling results in a weak modular design and limits the maintainability of a class. This means that the coupling properties of a class need to be analyzed to ensure better modularity and maintainability. Similarly, other metrics measure different characteristics of the classes that need to be monitored for better software quality. We selected the object-oriented metrics that quantify four different quality aspects: complexity, coupling, cohesion, and inheritance. The metrics listed in "Other" category (in Table 1) reflect general properties like size or are required for the detection of code smells. To show how bug-prone the code has been, we include the *number of bugs* associated with each class at a respective point during the development process.

4.3 Code Quality Analysis

From the related work, we observe that there are many class-level metrics that can be linked to code quality [37,40,51]. These approaches employ thresholds to detect code quality issues. Software metrics are also used to express abstraction level quality characteristics, such as coupling, complexity, cohesion, and inheritance [18,46]. In our system, we also apply software metrics with the thresholds defined by Filó et al. [18] to measure these quality attributes. We rate the quality level (good, regular, or bad) of a component or the severity level of a problem (high, medium, or low).

Code smells provide information on implementation decisions or choices that might degrade code quality [63]. Again based on related work [40], we detect four types of common class-level code smells: *Large Class*, *Functional Decomposition*, *Spaghetti Code*, and *Lazy Class* using class-level metrics. *Large Class* is the one that has many fields and methods, resulting in many lines of code [63]. A class with many private fields and methods is associated with *Functional Decomposition* [63]. A class with *Spaghetti Code* has long methods without proper structure [63]. A class with little to no functionality is a *Lazy Class* [63]. Since we have class-level metrics, it is possible to compute these code smells using predefined thresholds.

Based on the metrics and this analysis, the content of the code quality document comprises three parts: first, quality attributes covering coupling, complexity, cohesion, and inheritance; second, code smells in terms of *Large Classes*, *Functional Decomposition*, *Spaghetti Code*, and *Lazy Classes*; and third, information about bug history.
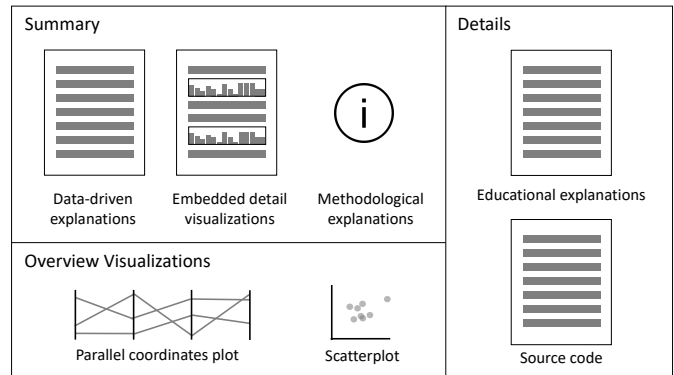

Fig. 2. Abstract representation of the interface structure.

5 EXPLORANATIVE CODE QUALITY DOCUMENTS

Using the described software quality analysis (Section 4) and our generic approach (Section 3), we have developed a visual analytics solution named *Code Quality Documents*. Our system is implemented as a web application using the libraries *D3js*, *JQuery*, *JQuery Sparklines* and builds on the standard *D3js* implementations of parallel coordinates plots and scatterplots. In this section, we first give an overview of the interface, then explain the individual components, and finally describe the interaction model that links the different components.

5.1 Interface Structure

We designed a multi-view interface with different panels for overview descriptions and details. Figure 2 shows an abstract representation of the interface, which maps the building blocks described in Section 3 to the different panels—specific examples of the interface can be found in Figure 1 and Figure 3. The summary panel presents the main data-driven textual explanations (I.i) summarizing the results of code quality analysis (Section 4.3). In addition, it contains embedded visualizations (II.i) and methodological explanations can be retrieved on demand (I.iii). The panel for the overview visualizations (II.i) contains a parallel coordinates plot and a scatterplot. The details view provides educational explanations (I.ii) or class details on selection.

The data-driven text describes code quality along with quality attributes, code smells, and bugs. The overview text contains detailed embedded visualizations and a list of classes corresponding to certain categories of code smells. By default, these details are not expanded in this panel. An icon [+] indicates that details are available and can be expanded. This dynamic expansion is similar to the concept of interline in fluid documents [62], where additional details are shown within the lines of the text. An info icon ⓘ hints at the presence of methodological explanations. Hovering this icon presents a tooltip (similar to a popup in fluid documents [62]) that describes the methodology used to come up with the respective detail. In Figure 4, the tooltip shows the

Code Quality Documents
This document presents the code quality aspects of xerces-1.2 ⊘—it has 18 packages and 293 classes.

Quality Attributes

An analysis of software metrics along four quality attributes ① shows generally *mixed* quality, with issues mainly regarding the cohesion within classes.

Complexity: The code complexity is *okay* ★ as 79 classes ▓ 27% are rated as having *low* quality, still fewer than the ones rated as *good* (153) or *regular* (61) ①. [−]
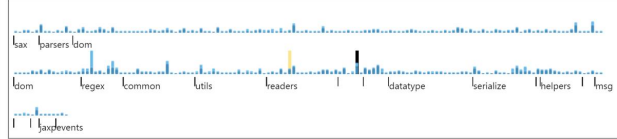
Coupling: The coupling between classes is *very good* ★★★ because only the small number of fourteen classes ▓ 4.8% is rated as having *low* quality, but many as *good* (234) ①. [+]

Cohesion: The cohesion within classes is *low* ⚠ because the high number of 221 classes ▓ 75.4% is rated as having *low* quality ①. [+]

Inheritance: The usage of class inheritance is *okay* ★. Although not a high number of classes (13) ▓ 4.4% is rated as having *low* quality, many are just classified as *regular* (148) and fewer as *good* (132) ①.
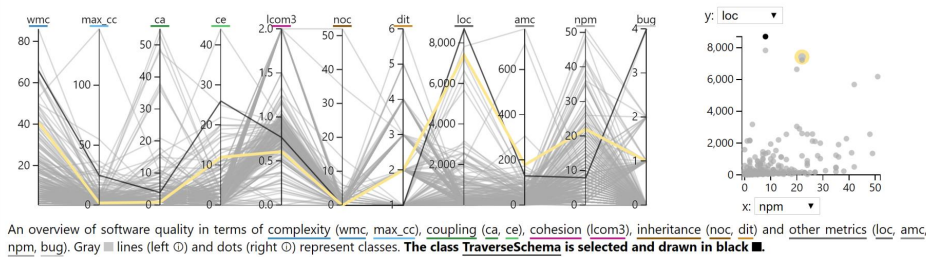
Code Smells

We have detected eighteen class-level code smells ①, among them five **Large Classes** [+], seven cases of **Functional Decomposition** [+], and six cases of Spaghetti Code [+]. Some classes [−]

UTF8Reader, XMLCharacterProperties, XMLMessages, **TraverseSchema**, and UTF8CharReader

carry multiple smells. For instance, UTF8Reader has two smells: Large Class and Spaghetti Code. Comparatively high metric value of lines of code (loc) indicates bad quality in **TraverseSchema**.

Bug History

With respect to past and present bugs, 43 of the classes ▓ 14.7% have been associated with bugs. The classes UTF8Reader and **TraverseSchema** have both associated bugs and identified code smells.

[X] Class TraverseSchema

Class **TraverseSchema** is the largest class of the project (loc: 8696) with, considering its size, a relatively small public interface (npm: 8). It has low quality with respect to the attributes complexity, coupling, and cohesion. Moreover, it carries Large Class and Functional Decomposition smells.

```java
private class ParticleRecoverableError extends Exception {
    ParticleRecoverableError(String s) {super(s);}
}

private class GroupInfo {
    int contentSpecIndex = -1;
    int scope = -1;
}

private class ElementInfo {
    int elementIndex;
    String typeName;

    private ElementInfo(int i, String name) {
        elementIndex = i;
        typeName = name;
    }
}

//REVISIT: verify the URI.
public final static String SchemaForSchemaURI = "http://www.w3.o

private TraverseSchema( ) {
    // new TraverseSchema() is forbidden;
}

public void setFullConstraintCheckingEnabled() {
```

An overview of software quality in terms of complexity (wmc, max_cc), coupling (ca, ce), cohesion (lcom3), inheritance (noc, dit) and other metrics (loc, amc, npm, bug). Gray ▬ lines (left ①) and dots (right ①) represent classes. **The class TraverseSchema is selected and drawn in black ■.**

Fig. 3. Various instances of *vis–text* interactions. A persistent highlighting (click on *TraverseSchema*) marks the related elements with bold font in text (*text–text*), a black line in the parallel coordinates, a black dot in the scatterplot (*text–vis*), and black background in embedded detail visualizations (*text–emvis*). Similarly, a non-persistent highlighting (hover on *UTF8Reader*) marks the corresponding elements in yellow. The details panel shows the class-specific description and source code of the persistently selected class, *TraverseSchema*.

Complexity: The code complexity is *okay* ★ as 79 classes ▓ 27% are rated as having *low* quality, still fewer than the ones rated as *good* (153) or *regular* (61) ①. [−]

We use thresholds values of weighted methods per class (wmc) and maximum cyclomatic complexity (max_cc) for categorizing complexity as *low*, *regular*, or *good*.

Low: wmc > 34 OR max_cc > 4
Regular: not *low* AND (wmc > 11 OR max_cc > 2)
Good: all other cases

The detailed embedded visualization can be expanded by clicking on the [+] icon. Each bar in this visualization represents a class and the classes are grouped by packages.

Coupling: The cou... only the small number of fourteen classes ▓ 4.8% is rated as having

Fig. 4. Methodological explanation of classifying complexity in terms of low, regular, or good.

software metrics and their thresholds used to classify low, regular, or good complexity. We argue that highlighting different quality issues with methodological explanations can assist users in making decisions to improve the quality.

Educational explanations are related to the domain of code quality. The names of the quality attributes and the software metrics carry a thick colored border at the bottom. Clicking on these terms brings up background information in the details view. The description also includes project-specific examples to better communicate the concept. For instance, Figure 1ⓓ shows the educational explanation for the quality attribute *cohesion*; *WhitespaceAnalyzer* is provided as an example of a class having the highest value of *lack of cohesion of methods*. In a similar way, explanations on code smells can be accessed. Clicking on a class anywhere in the system opens the source code of that class

in the details view preceded by a class-specific explanation providing a short summary of problems in the class, if any.

5.2 Natural Language Generation

To automatically produce textual descriptions, we need to generate text from data. We aim at a self-explanatory and understandable text that has a natural flow and appears almost as if written by a human. To this end, we employ template-based natural language generation [19], a simple, but effective and easy-to-use generation method.

Our text generation process is based on previous work [11, Section IIIA] [27, Section 4.2] and can be modeled as directed decision graphs—one for each subsection of the report. The decision graph consists of *start*, *stop*, *decision*, and *text* nodes. Any traversal of the graph from *start* to *stop* node results in a report section. *Decision* nodes control the flow of the graph based on the values of decision variables. *Text* nodes consist of predefined templates with variables in them filled with the information coming from data. Traversing a *text* node adds a phrase or sentence to the report section. Even in the *text* nodes, we make extensive use of conditions, not just to account for different grammatical cases such as plural or singular but also to describe different cases and provide reasons for an analysis result. For instance, we not only list the number of classes that have low quality with respect to any of the four quality attributes but explain reasons for the rating. The same rating can even have different reasons (Figure 3: *"The code complexity is* okay *as 79 classes are rated as having low quality, still fewer than the ones rated as* good *(153) or* regular *(61)."* and *"The usage of class inheritance is* okay. *Although not a high number of classes (13) is rated as having low quality, many are just classified as* regular *(148) and fewer as* good *(132)."* We leave out sentences if no results are available and handle special cases. For instance, a special case for this paragraph is when no code smells were found: *"We have not detected any class-level code smells in the project—congratulations."*
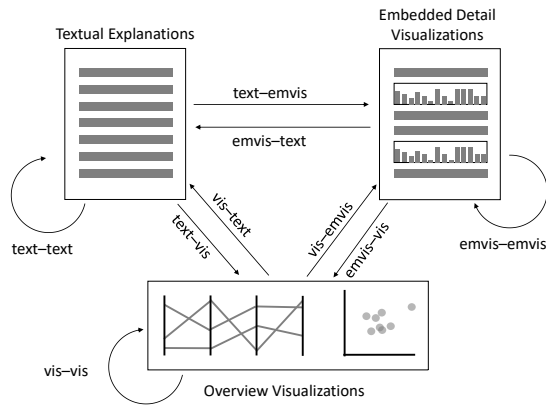
Fig. 5. Abstract representation of the interaction model. We have bi-directional interactions among text, embedded visualizations (emvis), and overview visualizations (vis).

5.3 Visualizations

For data exploration and context of the textual descriptions, we provide two overview visualizations: a parallel coordinates plot and a scatter-plot. These visualizations are useful in discerning important patterns and relationships between metrics [45]. We argue that users can find these visualizations useful in understanding the code quality and comparing the properties of different classes. To obtain an overview of all the metrics, the parallel coordinates plot is helpful, whereas the scatterplot supports the identification of relationships between two metrics.

In addition to these visualizations, the embedded detail visualizations complement the text generated in the document. We employ small bar charts to represent the metric values of one category for all classes, structuring the classes with respect to the packages in which they are contained. Furthermore, we use word-sized bar charts to indicate percentage values. The values always refer to problematic cases (e.g., low quality or bug-prone classes) and are given relative to the overall number of classes. Small star icons and warning symbols provide a quick hint of the respective rating for each quality attribute.

We use consistent color coding to couple the visualized metrics. For instance, in Figure 3, the colors of complexity metrics in the caption of the parallel coordinates plot match the color coding of the complexity bar chart in the quality attributes section. The metrics are grouped in terms of the quality attributes—two metrics of the same group are associated with the same hue but a different brightness.

5.4 Interaction Model

To connect textual and visual descriptions, consistent interactive linking is important (see also Section 3). We build on the concept of *vis–text* interaction introduced by Beck and Weiskopf [12]. In addition to the local interactions discussed above (i.e., interactions that only locally impact the interface, such as details blended in on demand), a global interaction model is intended to link the various visual representations. In slight adaption to the model of Beck and Weiskopf, we discern the textual explanations (*text*), overview visualizations (*vis*), and embedded visualizations (*emvis*). As shown in Figure 5, these representations are interactively linked to each other in a bi-directional way.

As these components contain different representations of the same class-level multivariate data, an essential interaction is the brushing and linking of data points across all representations. Hovering over a class name anywhere in the text triggers a transient selection (a non-persistent selection). It highlights the corresponding poly-line in the parallel coordinates plot (*text–vis*), the dot in the scatterplot (*text–vis*), the bars in the embedded visualizations (*text–emvis*), and other occurrences of the class name in the text (*text–text*). Figure 3 shows the effect of hovering over the class name UTF8Reader; the linked parts are highlighted with yellow color. Apart from the other instances of the hovered class in text, we also mark the corresponding code

smells. The *Large Class* and *Spaghetti Code* are highlighted since *UTF8Reader* contains both of these code smells (Figure 3). Hovering over a bar in the embedded visualization and a dot in the scatterplot has a similar effect and triggers *emvis–vis/vis–vis*, *emvis–text/vis–text*, and *emvis–emvis/vis–emvis* interactions.

The transient selection shows as long as the interactive element is hovered and provides a quick way of cross-referencing different representations. To make the highlighting persistent, the interactive elements can be clicked; the parts related to the clicked element are highlighted with black color in the visualizations and with bold font in the text. For instance, Figure 3 shows the persistent selection corresponding to the class TraverseSchema. This helps in getting a comparative overview of two different classes with respect to various aspects; one glance at Figure 3 is sufficient to tell that *UTF8Reader* and *TraverseSchema* have one code smell (*Large Class*) in common. In addition, we can quickly observe that *UTF8Reader* has less complexity (embedded visualization for complexity), fewer *lines of code* (scatterplot), and fewer bugs (parallel coordinates plot) than *TraverseSchema*.

Clicking on a code smell, aside from showing an educational explanation in the details view, highlights the set of classes that contain that code smell in the parallel coordinates plot and scatterplot (*text–vis*)—Figure 6 shows the result of clicking *Functional Decomposition*. This helps in understanding the pattern of metric values for the classes having different code smells. Since the scatterplot illustrates the relationship between any two software metrics, we update the dimensions of the scatterplot on persistent interactions according to the context. For example, clicking on *Functional Decomposition* will update the scatterplot dimensions to *weighted methods per class* and *number of public methods* as these metrics are used to identify the smell (see Figure 6). Moreover, users can explore the relationships between other metrics. Since lines are hard to select in a parallel coordinates plots (they are thin and often occlude each other), we provide a persistent range selection on the axes (brushing interaction with mouse press and hold). On every persistent selection, the caption of the figures adapts accordingly to describe the selected elements (see Figure 6). In contrast to legends, the textual captions allow for the inclusion of contextual and methodological information (e.g., data filtering criterion), which helps in making the interactive visualizations more self-explanatory.

6 DESIGN PROCESS AND EVALUATION

We followed a design process in which the authors of this paper worked in close collaboration as a team. Initially, the team was composed of three members, but later another member was included because of a specific skill-set. The fourth member was included in the fourth quarter of the design process. The team has a blend of different backgrounds and expertise. One member has expertise in software engineering; one member works in the domain of software visualization; the third has experience in visual analytics and visualization in general; the fourth member (included later) has skills in automated text generation and interactive documents. The team members were located at different institutes, therefore, frequent visits were arranged. One member spent a week at a lab where other members usually collaborate. In addition, we organized regular meetings to share ideas and discuss the outcomes of the design decisions for our system. Some of the implementation work was completed in a pair programming setting. The process went through a period of approximately 9 months with a focus on designing a visual analytics system for a domain-specific problem.

Along the design process, we conducted a formative evaluation (i.e., an evaluation that focuses on testing and improving intermediate prototypes)—in the fourth quarter of the design process—to validate the problem analysis and our visual analytics solution. We invited participants who were not part of the design process team and are not authors of this paper. From our formative study, we obtained user feedback and observed usage strategies. We performed a qualitative analysis of the participants' feedback—received through a questionnaire—where they expressed their views (positive and negative) about our system. In the questionnaire, we formulated tasks to analyze the usability and usefulness of our system. In particular, through our formative study, we assessed how different views coordinate with each other; how *vis–text*
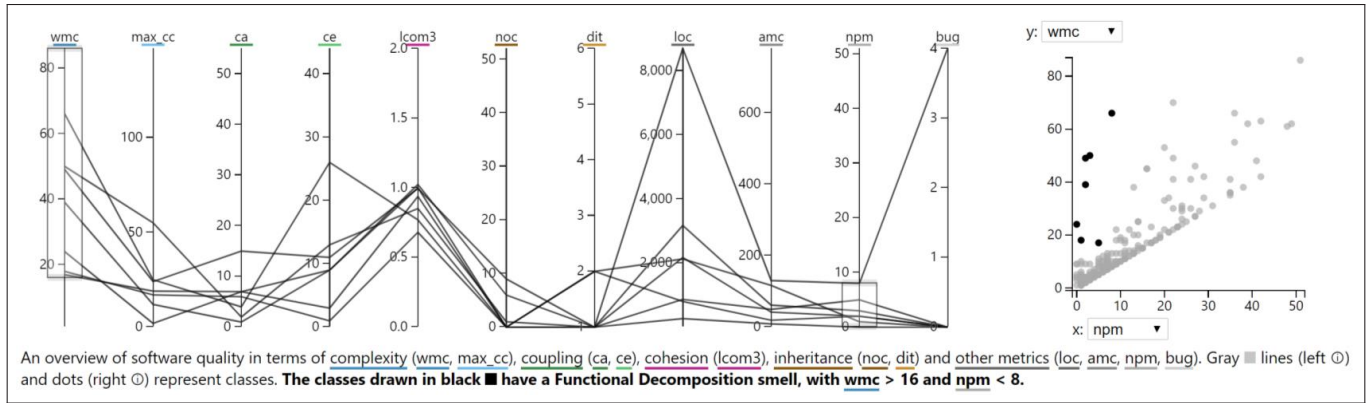
Fig. 6. A selection of the classes in *Xerces 1.2* that have a *Functional Decomposition* smell; they are highlighted in the parallel coordinates plot and the scatterplot. The caption of the visualizations adapts to describe the selection.

interaction support the exploration process; how text, visualizations (embedded and non-embedded), and their interactive linking help in the code quality analysis process. The results helped us refine and improve our system. We conducted two iterations within the formative study. The data from the user study is provided in the supplemental material [38].

First Iteration – In terms of textual explanations, the first tested version of the system had data-driven explanations, some educational explanations, but no methodological explanations. From the interaction perspective, the system had a few one-way *text–vis* interactions and a few *emvis–emvis* interactions. Consistent linking was implemented minimally. The first iteration of the formative study was organized with four participants (3 PhD students and a postdoc), two with a software engineering background and the other two working in visualization. We decided for this mix of experts to receive educated feedback both on the content of the documents as well as the data presentation. The study included three phases and took approximately 45 minutes per participant. In the first phase, participants were asked to use the system, identify different aspects of code quality discussed in the document, and summarize them. The second phase involved reviewing different features of the system and providing detailed feedback on each. As the third phase, we interviewed the participants, asked general questions, and concluded the session. We performed a qualitative analysis of the participants' feedback. Although we received quantitative feedback (through Likert scale) as well, we were more interested in textual responses because they helped us identify areas for improvement. We summarize the results as follows, also discussing our improvements that we implemented as a reaction to the feedback.

All participants found the data-driven explanations helpful and complementing the visualizations (*"It is nice to see discussion on some quality aspects of code. The tool analyzes the data well; interactions and details of code are useful."*). However, due to the lack of educational and methodological explanations, they were unclear about certain descriptions (*"I have [had] a hard time trying to figure out the meanings of the acronyms [metrics] in parallel coordinates plot and scatter plot [...]"*); two participants disagreed that the document is self-explanatory due to lack of educational and methodological explanations (*reaction*: we consistently added educational and methodological explanations). One participant argued to exclude details about the project from the main text because it distracted the focus from the aspects discussed in the document (*reaction*: we moved this text to the header to clarify that it is meta information and not a regular part of the document).

The participants suggested to have more interactions to make the system more intuitive. The missing interactions were *vis–text* and *emvis–text* (*"I did not find a link from the visualization to the text. But the other way round [it] works nicely."*). Although we had some *text–vis* interactions present in the system, it was interesting that participants were expecting to click on visual representations of classes. The scarcity of interactions was hindering the exploration process. They also pointed

out the lack of *emvis–vis* linking (*reaction*: we systematically added interactive linkage between all representations of the same data). The participants emphasized that the overview visualization should not scroll out but be visible at all times (*"The placing (initially 60% off-screen, scrolling is needed) is bad [...]"*; *reaction*: we moved the overview visualizations to a separate panel).

We received mixed responses on the parallel coordinates plot and scatterplot, but the participants did not use the visualizations for exploration yet (*"Maybe they [parallel coordinates plot and scatterplot] would be useful to assess some hypotheses, but for exploratory analysis it is not clear how to use them [...]"*). We assume the main reason was a lack of interactive linking, especially between the two visualizations and with the textual explanations (*reaction*: we consistently added such interactive links). The participants also suggested to include legends and captions in the visualizations to make them self-explanatory because sometimes they were unable to understand the changes happening in the visualizations (*reaction*: we added captions and provide legends as part of the captions and in the details panel).

Second Iteration – In the second iteration of our formative evaluation, the system incorporated all types of textual explanations, however, still missing a few explanations. Furthermore, most of the transient interactions through consistent linking of text and visualizations were provided, without the full implementation of the persistent interactions yet. We invited three participants (two PhD students and one postdoc) who participated in the first phase to identify improvements in the system and, for a fresh perspective, one new participant (PhD student) to evaluate the system. The new participant has a background in software engineering and is currently conducting information visualization research. We followed a similar procedure as in the first iteration, however, removing the first phase because we did not expect new insights if repeating it; the duration of the study was reduced to about 30 minutes per participant. We also updated the interview questions to focus on the specific improvements and the general applicability of our approach.

All participants agreed that the system was improved overall. We received positive feedback on the layout, textual descriptions, coordinated views, interactions, and linking (*"The layout is more compact. I can see all the information without having to scroll, which benefits the analysis when highlighting elements of the coordinated views. The interface looks clean, which facilitates to concentrate on the visualizations."*). The participants confirmed that the system supports exploration well and textual explanations are useful and well connected. However, one of the participants suggested to include more information about bugs (*reaction*: *none*—we do not have access to more details; retrieving these and extending the description accordingly remains future work). All participants, except one, expressed that the document is self-explanatory. He wanted more methodological and educational explanations on metrics (*"[...] more information why certain metrics are grouped together and what are the background of those [metrics]."*; *reaction*: we added more detailed explanations on metrics). It was also

Fig. 7. Low quality in *HTMLElementImpl* found through active exploration of the embedded detail visualizations of coupling, cohesion, and inheritance.

unclear to him that labels under the small bar charts are package names (*reaction*: we now provide an explanation of the bar chart in the respective methodological explanation, see Figure 4). One participant did not find horizontal embedded bars (that show percentages) useful because they do not add much to the text (*reaction*: we reduced the number of shown bars and improved their consistency; they now always refer to problematic classes as a fraction of the total number of classes). The majority of the participants accentuated that the system can be useful for educational purposes (e.g., teaching students about code-related issues).

Despite incrementally improving, some limitations remain in our visual analytics system. Currently, the system analyzes the software quality but does not explicitly guide the users in correcting measures. For instance, it discusses different code smells in a project without suggesting suitable refactorings. We also could not include more information about bugs besides *number of bugs* because finding access to relevant information (e.g., bug appearance timeline, active duration, resolution) is difficult. Although we have optimized for self-explanation, misinterpretation of textual descriptions and visualizations is still possible. However, through the interactive linking of text and visualizations, we hope potential misunderstanding can be mitigated. In terms of evaluation, the participants had appropriate knowledge of visualization and code quality to provide substantial feedback on visualization design, usability, and code quality analysis. Additional valuable insights could be obtained through a larger-scale user study with an appropriate blend of targeted users having industrial experience.

7 APPLICATION EXAMPLES

With two application examples, we demonstrate how our exploranative documents can assist in analyzing code quality. This section also implicitly reflects on how users can work with our interactive document and the support they receive from it to achieve various analysis tasks. We demonstrate the working of our exploranative code quality documents using two software projects: *Xerces 1.2* (an XML handling framework) [1,24] and *Lucene 2.0* (a search library) [3,31]. We demonstrate how textual explanations point to a specific issue in the code, and how exploration helps in analyzing problems.

In the *Xerces 1.2* project, our system identifies 18 code smells (Figure 3). From the textual explanation, we read that *UTF8Reader* has *Large Class* and *Spaghetti Code*. *UTF8Reader* becomes more interesting when we realize that this class has had a large number of bugs as well. Looking into the code of the class, we find a confirmation of these issues self-admitted by the developers in the class description comment, also providing a reason: "*Some blatant v[io]lation of good object oriented programming rules, ignoring boundaries of modularity, etc., in the name of good performance.*" This is an example that shows that our approach helps the users focus on most problematic classes.

Another class highlighted in Figure 3 is *TraverseSchema* because it also carries two code smells (*Large Class* and *Functional Decomposition*) and has a history of bugs. In the class description, we read that *TraverseSchema* is the largest class in the system and its quality is low with respect to complexity, coupling, and cohesion (see Figure 3, details panel). Exploration through consistent linking helps locate this class in the parallel coordinates plot and the scatterplot (see Figure 3, black line and dot); it shows a similar pattern as *UTF8Reader* (see Figure 3, yellow line and dot). Investigating the source code of *TraverseSchema*, we confirm the existence of a *Functional Decomposition* smell because it has several private but few public methods.

During an exploration of the bar charts in the quality attributes section, we find that *HTMLElementImpl* has a complex inheritance structure indicated by an extremely high *number of children* (Figure 7; 52 children, while the threshold of low quality is only more than three children). The extending classes are also part of the *dom* package and can be investigated through the bar chart. We head on to explore the quality of *HTMLElementImpl* from other quality attributes as well. We notice that the related metrics of coupling and cohesion are also high, meaning low quality in these attributes, however, the complexity metrics are in acceptable range (see Figure 7). Similar information is also mentioned in the summary presented with the source code.

We use the parallel coordinates plot and the scatterplot to learn more about outliers and the interplay of metrics. For instance, in Figure 6, a positive correlation can be observed between *weighted method per class* (wmc) and *number of public methods* (npm) in the scatterplot, however, a few classes form interesting outliers with high wmc values but low npm values. These unusual characteristics correspond with the *Functional Decomposition* smell (highlighted as black dots in the scatterplot in Figure 6, right). As the other metric values for this set of classes are shown accordingly in the parallel coordinates plot (Figure 6, left), we also observe that one of the classes is associated with large file size (loc) and a high number of bugs.

In the *Lucene 2.0* project (Figure 1), we observe good quality for coupling and inheritance, whereas quality is lower for complexity and cohesion. We see an outlier (*StandardTokenizerTokenManager*) in the complexity bar chart, and when clicking it (see Figure 1), we observe it highlighted also in the explanations of code smells and the bug history section. The class that has three code smells (*Large Class*, *Functional Decomposition*, and *Spaghetti Code*) and has been associated with bugs. *StandardTokenizerTokenManager* also turns out to be a prominent outlier in the parallel coordinates plot and scatterplot.

We then switch to code smell analysis and notice that *Functional Decomposition* is the most occurring code smell, while we only see single instances of *Large Class* and *Spaghetti Code*. We further investigate the classes that carry *Functional Decomposition* by exploring the overview visualizations and source codes to discern the common reason for the code smell. We recognize that the majority of the classes

that have *Functional Decomposition* share a problem: more private methods and fewer public methods; moreover, they have large class sizes.

8 LESSONS LEARNED

Finally, we summarize our experience gained in designing the system as lessons learned. We focus on aspects that generalize beyond the specific application in software engineering to interfaces producing interactive documents that consist of generated textual and visual data descriptions.

Overview always! – While, in traditional text documents, text and graphics are arranged in a fixed layout, we observed that loosening this tight coupling is beneficial for interactive documents like ours. Especially, overview visualizations should always stay in view and not scroll out with the text (like tested in the first iteration of our formative evaluation). Also, texts that provide a high-level summary might act as anchors for the users to return to and should stay visible. We used the section describing code smells in this way. Such overview elements are important for interactive highlighting as well: hovering an element anywhere in the interface, related items in the overview representations get highlighted. For instance, with such highlighting in the code smell section, we can easily indicate by which code smells a highlighted class is affected. The always visible overview representations, together with a stable layout, provide a reliable skeleton for the user and allows other content to change dynamically without confusing the user's mental map. This lesson relates to Shneiderman's information-seeking mantra [47], which emphasizes on overview first and details later.

Consider brushing text, really. – From a visualization perspective, texts in an interface might appear being *dead*—they are not part of a visualization and cannot be interacted with. However, we have shown that text can be integrated into interactions almost like any other element of a visualization. Brushing a marked text element, the related visual representations get highlighted across all visualizations, and vice versa. Opportunities where such interactions make sense appear naturally whenever describing a data-related entity (in our case: a class, a software metric, a code smell, etc.). Also, methodological and educational explanations as well as further data-driven details can be interactively linked with a text. For the text we generate, we tried to identify all elements the users might want to explore further and decided what would be an appropriate interactive linking. This lesson links to the interaction model proposed by Beck and Weiskopf [12].

Captions! And make them dynamic. – Although not difficult to implement, many visual analytics systems even lack basic captions for the visualizations shown in different views. We decided to add captions to make the interface more self-explanatory. And it quickly turned out that these captions should to be adaptive: When something changes in the visualization based on an interaction, the caption needs to change accordingly as it should always accurately describe what is currently shown. Template-based natural language generation provides the means for implementing such adaptive captions.

Pointers, everywhere. – Although we defined them as separate categories in Section 3, data-driven, educational, and methodological explanations should somewhat blend in practice. A purely data-driven explanation might read cryptically, but with a few educational or methodological hints, the text would provide the necessary pointers to understand it more easily. Also, for instance, an educational explanation can profit from examples from the data, like we have integrated in the respective background text. These hints can be used as hyperlinks to the more detailed explanations. Still, we recommend, when authoring the texts, strictly discerning between the categories of text and also reflecting this categorization in the user interface by a consistent layout. For instance, we presented *methodological explanations* only in tooltips of info icons and educational explanations, marked with the term *background*, in the details view on the right. This consistency allows users to learn where to look for certain information.

You just learn on the side. – Like most visual analytics systems, our approach is also built to support users in understanding the specific data shown. With every data analysis, the users gain experience and might also have general insights with respect to the overall analysis procedure or domain. In contrast to other interfaces, we actively support this *learning on the side* through methodological and educational explanations. Through activating these explanations, the interface is adapting—not automatically but with only little extra effort—to the individual information needs of the user.

9 CONCLUSION AND FUTURE WORK

We have introduced an approach that automatically generates interactive documents to describe multivariate data and tailored the approach in a design study for reporting the code quality of a software project. The approach is *exploranative* as it *explains* the data and background in a textual way as well as it supports the *exploration* of the data through interactive visualizations. The textual and visual representations are consistently linked with each other. While our design process and formative evaluation have already provided insights into usability and aspects of visual design, a next step is to conduct a larger-scale user study with representatives of our targeted users (see Section 4.1). A goal of the study will be to test in a realistic setting whether actionable results can be derived from our documents across participants with different levels of expertise in code quality analysis.

We have introduced the general concept of *exploranative documents for multivariate data* (Section 3) and have demonstrated this concept in detail with one example from software engineering. Still, future work is to show the applicability of the concept for other data analysis scenarios. For instance, in the context of *Industry 4.0*, comparable documents could summarize multivariate performance and maintenance indicators of the machines of a modular production line. Besides visualizing multivariate raw data, textual descriptions of analysis results could directly hint at potential issues before the issues manifest themselves in an expensive halt of the production line. A comparable interactive document could be the result of another design study process, then involving manufacturing researchers and professionals.

We consider our contributions in extending interactive textual documents with respect to explorative analysis as a step toward blurring the borderline between textual and visual data representations. The interaction model introduced to connect visual and textual elements goes beyond previous work and showcases how text generation and brushing-and-linking techniques can play together in a multi-view system. We believe that texts, well-integrated with visualizations, can make data analysis more accessible and easier to understand for a wide audience. With further integration, eventually, *text* and *visualization* will become only two points in a continuum (with any point in between possible) instead of being treated as two separate modalities.

As part of future work, we are interested in investigating the educational aspects of visual analytics in more detail. For instance, learning research literature [60] discusses *worked examples*—step-by-step demonstrations—and *instructional explanations* that both share similarities to the textual explanations used in our approach. Although learning is usually not the primary goal of a visual analytics system, it still would be highly relevant to study how such systems educate their users. Follow-up research questions are whether adding educational and methodological explanations to an interactive visualization has positive educational effects and how users like and deal with textual descriptions when they have already become familiar with the text.

REFERENCES

[1] Apache Lucene Java downloads. `http://archive.apache.org/dist/lucene/java/`. Accessed: 2019-07-22.

[2] Checkstyle. `https://checkstyle.sourceforge.io/`. Accessed: 2019-07-22.

[3] Mirror of Apache Xerces2 Java. `https://github.com/apache/xerces2-j/tree/xerces_j_1`. Accessed: 2019-07-22.

[4] PMD: An extensible cross-language static code analyzer. `https://pmd.github.io/`. Accessed: 2019-07-22.

[5] SonarQube. https://www.sonarqube.org/. Accessed: 2019-07-22.

[6] M. J. Adler and C. Van Doren. *How to Read a Book*. Simon and Schuster, 1972.

[7] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz. Software Landscapes: Visualizing the structure of large software systems. In *Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization (VisSym)*, pp. 261–266. Eurographics Association, 2004. doi: 10.2312/VisSym/VisSym04/261-266

[8] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering (TSE)*, 28(1):4–17, 2002. doi: 10.1109/32.979986

[9] F. Beck, F. Hollerich, S. Diehl, and D. Weiskopf. Visual monitoring of numeric variables embedded in source code. In *Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 1–4. IEEE, 2013. doi: 10.1109/VISSOFT.2013.6650545

[10] F. Beck, O. Moseler, S. Diehl, and G. D. Rey. In situ understanding of performance bottlenecks through visually augmented code. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pp. 63–72. IEEE, 2013. doi: 10.1109/ICPC.2013.6613834

[11] F. Beck, H. A. Siddiqui, A. Bergel, and D. Weiskopf. Method Execution Reports: Generating text and visualization to describe program behavior. In *Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 1–10. IEEE, 2017. doi: 10.1109/VISSOFT.2017.11

[12] F. Beck and D. Weiskopf. Word-sized graphics for scientific texts. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 23(6):1576–1587, 2017. doi: 10.1109/TVCG.2017.2674958

[13] J. Bohnet and J. Döllner. Monitoring code quality and development activity by software maps. In *Proceedings of the Workshop on Managing Technical Debt (MTD)*, pp. 9–16. ACM, 2011. doi: 10.1145/1985362.1985365

[14] H. Burden and R. Heldal. Natural language generation from class diagrams. In *Proceedings of the International Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)*, pp. 8:1–8:8. ACM, 2011. doi: 10.1145/2095654.2095665

[15] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering (TSE)*, 20(6):476–493, 1994. doi: 10.1109/32.295895

[16] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 275–284. IEEE, 2014. doi: 10.1109/SCAM.2014.14

[17] U. Erdemir, U. Tekin, and F. Buzluca. E-Quality: A graph based object oriented software quality visualization tool. In *Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pp. 1–8. IEEE, 2011. doi: 10.1109/VISSOF.2011.6069454

[18] T. G. Filó, M. Bigonha, and K. Ferreira. A catalogue of thresholds for object-oriented software metrics. In *Proceedings of the International Conference on Advances and Trends in Software Engineering (SOFTENG)*, pp. 48–55, 2015.

[19] A. Gatt and E. Krahmer. Survey of the state of the art in natural language generation: Core tasks, applications and evaluation. *Journal of Artificial Intelligence Research (JAIR)*, 61:65–170, 2018. doi: 10.1613/jair.5477

[20] P. Goffin, W. Willett, J.-D. Fekete, and P. Isenberg. Exploring the placement and design of word-scale visualizations. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 20(12):2291–2300, 2014. doi: 10.1109/TVCG.2014.2346435

[21] M. Harward, W. Irwin, and N. Churcher. In situ software visualisation. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*, pp. 171–180. IEEE, 2010. doi: 10.1109/ASWEC.2010.18

[22] J. Hoffswell, A. Satyanarayan, and J. Heer. Augmenting code with in situ visualizations to aid program understanding. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, pp. 532:1–532:12. ACM, 2018. doi: 10.1145/3173574.3174106

[23] P. Jordan, N. L. Green, C. Thomas, and S. Holm. TBI-Doc: Generating patient & clinician reports from brain imaging data. In *Proceedings of the International Natural Language Generation Conference (INLG)*, pp. 143–146. Association for Computational Linguistics, 2014.

[24] M. Jureckzo. Lucene, July 2010. doi: 10.5281/zenodo.322458

[25] D. H. Kim, E. Hoque, J. Kim, and M. Agrawala. Facilitating document reading by linking text and tables. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST)*, pp. 423–434. ACM, 2018. doi: 10.1145/3242587.3242617

[26] S. Latif and F. Beck. Interactive map reports summarizing bivariate geographic data. *Visual Informatics (VI)*, 3(1):27–37, 2019. doi: 10.1016/j.visinf.2019.03.004

[27] S. Latif and F. Beck. VIS Author Profiles: Interactive descriptions of publication records combining text and visualization. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 25(1):152–161, 2019. doi: 10.1109/TVCG.2018.2865022

[28] S. Latif, D. Liu, and F. Beck. Exploring interactive linking between text and visualization. In *Proceedings of the Eurographics/IEEE VGTC Conference on Visualization: Short Papers (EuroVis)*, pp. 91–94. Eurographics Association, 2018.

[29] B. Lavoie, O. Rambow, and E. Reiter. The ModelExplainer. In *Demonstration Notes of the International Natural Language Generation Workshop (INLG)*, pp. 9–12, 1996.

[30] B. Lavoie, O. Rambow, and E. Reiter. Customizable descriptions of object-oriented models. In *Proceedings of the Conference on Applied Natural Language Processing (ANLC)*, pp. 253–256. Association for Computational Linguistics, 1997. doi: 10.3115/974557.974594

[31] J. Marian. Defect prediction: Xerces, July 2010. doi: 10.5281/zenodo.268474

[32] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pp. 279–290. ACM, 2014. doi: 10.1145/2597008.2597149

[33] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering (TSE)*, SE-2(4):308–320, 1976. doi: 10.1109/TSE.1976.233837

[34] F. Meziane, N. Athanasakis, and S. Ananiadou. Generating natural language specifications from UML class diagrams. *Requirements Engineering (RE)*, 13(1):1–18, 2008. doi: 10.1007/s00766-007-0054-0

[35] V. O. Mittal, S. F. Roth, J. D. Moore, J. Mattis, and G. Carenini. Generating explanatory captions for information graphics. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1276–1283. Morgan Kaufmann Publishers Inc., 1995.

[36] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for Java classes. In *Proceedings of the IEEE International Conference on Program Comprehension (ICPC)*, pp. 23–32. IEEE, 2013. doi: 10.1109/ICPC.2013.6613830

[37] H. Mumtaz, F. Beck, and D. Weiskopf. Detecting bad smells in software systems with linked multivariate visualizations. In *Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 12–20. IEEE, 2018. doi: 10.1109/VISSOFT.2018.00010

[38] H. Mumtaz, S. Latif, F. Beck, and D. Weiskopf. Data and tool: Explorative Code Quality Documents, 2019. doi: 10.5281/zenodo.3336019

[39] E. Murphy-Hill and A. P. Black. An interactive ambient visualization for code smells. In *Proceedings of the International Symposium on Software Visualization (SOFTVIS)*, pp. 5–14. ACM, 2010. doi: 10.1145/1879211.1879216

[40] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum. Maintainability defects detection and correction: A multi-objective approach. *Automated Software Engineering (ASE)*, 20(1):47–79, 2013. doi: 10.1007/s10515-011-0098-8

[41] P. Pirolli and S. Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of the International Conference on Intelligence Analysis (ICIA)*, pp. 2–4, 2005.

[42] A. Ramos-Soto, A. J. Bugarin, S. Barro, and J. Taboada. Linguistic descriptions for automatic generation of textual short-term weather forecasts on real prediction data. *IEEE Transactions on Fuzzy Systems (TFS)*, 23(1):44–57, 2015. doi: 10.1109/TFUZZ.2014.2328011

[43] A. Ramos-Soto, B. Vazquez-Barreiros, A. Bugarín, A. Gewerc, and S. Barro. Evaluation of a data-to-text system for verbalizing a learning analytics dashboard. *International Journal of Intelligent Systems (IJIS)*, 32(2):177–193, 2017. doi: 10.1002/int.21835

[44] E. Reiter, R. Dale, and Z. Feng. *Building Natural Language Generation Systems*. MIT Press, 2000.

[45] C. Schulz, N. Rodrigues, K. Damarla, A. Henicke, and D. Weiskopf. Visual exploration of mainframe workloads. In *Proceedings of the SIGGRAPH Asia Symposium on Visualization (SA17VIS)*, pp. 4:1–4:7. ACM, 2017. doi: 10.1145/3139295.3139312

[46] R. Shatnawi, W. Li, J. Swain, and T. Newman. Finding software metrics threshold values using ROC curves. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 22(1):1–16, 2010. doi: 10.

1002/smr.404

[47] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *The Craft of Information Visualization*, pp. 364–371. Elsevier, 2003. doi: 10.1016/B978-155860915-0/50046-9

[48] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 43–52. ACM, 2010. doi: 10. 1145/1858996.1859006

[49] A. Srinivasan, S. M. Drucker, A. Endert, and J. Stasko. Augmenting visualizations with interactive data facts to facilitate interpretation and communication. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 25(1):672–681, 2019. doi: 10.1109/TVCG.2018.2865145

[50] S. Sripada, N. Burnett, R. Turner, J. Mastin, and D. Evans. A case study: NLG meeting weather industry demand for quality and quantity of textual weather forecasts. In *Proceedings of the International Natural Language Generation Conference (INLG)*, pp. 1–5. The Association for Computer Linguistics, 2014.

[51] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. *Information Systems Journal (ISJ)*, 12(1):43–60, 2002. doi: 10.1046/j.1365-2575.2002.00117. x

[52] M. Sulír, M. Bačíková, S. Chodarev, and J. Porubän. Visual augmentation of source code editors: A systematic mapping study. *Journal of Visual Languages & Computing (JVLC)*, 2018. doi: 10.1016/j.jvlc.2018.10.001

[53] B. Swift, A. Sorensen, H. Gardner, and J. Hosking. Visual code annotations for cyberphysical programming. In *Proceedings of the International Workshop on Live Programming (LIVE)*, pp. 27–30. IEEE, 2013. doi: 10. 1109/LIVE.2013.6617345

[54] E. R. Tufte. *Beautiful Evidence*. Graphics Press, first ed., 2006.

[55] E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pp. 97–106. IEEE, 2002. doi: 10.1109/WCRE.2002.1173068

[56] B. Victor. Explorable explanations. `http://worrydream.com/ ExplorableExplanations/`, 2011. Accessed: 2019-03-28.

[57] W. Wahlster, E. André, W. Finkler, H.-J. Profitlich, and T. Rist. Plan-based integration of natural language and graphics generation. *Artificial Intelligence (AI)*, 63(1-2):387–427, 1993. doi: 10.1016/0004-3702(93) 90022-4

[58] D. Weiskopf, M. Borchers, T. Ertl, M. Falk, O. Fechtig, R. Frank, F. Grave, A. King, U. Kraus, T. Müller, H. Nollert, I. R. Mendez, H. Ruder, T. Schafhitzel, S. Schär, C. Zahn, and M. Zatloukal. Explanatory and illustrative visualization of special and general relativity. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 12(4):522–534, 2006. doi: 10.1109/TVCG.2006.69

[59] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 551–560. IEEE, 2011. doi: 10.1145/1985793.1985868

[60] J. Wittwer and A. Renkl. How effective are instructional explanations in example-based learning? A meta-analytic review. *Educational Psychology Review (EPR)*, 22(4):393–409, 2010. doi: 10.1007/s10648-010-9136-5

[61] A. Ynnerman, J. Löwgren, and L. Tibell. Exploranation: A new science communication paradigm. *IEEE Computer Graphics and Applications (CG&A)*, 38(3):13–20, 2018. doi: 10.1109/MCG.2018.032421649

[62] P. T. Zellweger, S. H. Regli, J. D. Mackinlay, and B.-W. Chang. The impact of fluid documents on reading and browsing: An observational study. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pp. 249–256. ACM, 2000. doi: 10.1145/332040.332440

[63] M. Zhang, T. Hall, and N. Baddoo. Code bad smells: A review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 23(3):179–202, 2011. doi: 10.1002/smr.521