Journal of the Brazilian Computer Society

RESEARCH Open Access

VisminerTD: a tool for automatic identification and interactive monitoring of the evolution of technical debt items

Thiago S. Mendes[1,4*†], Felipe G. S. Gomes[1†], David P. Gonçalves[1], Manoel G. Mendonça[1,5], Renato L. Novais[3,5] and Rodrigo O. Spínola[2,5]

Abstract
Technical debt (TD) contextualizes problems faced during software evolution considering the tasks that are not carried out adequately during software development. Software TD is a type of debt that brings a short-term benefit, but which may have to be paid with interest later on in the software development life cycle. Its presence brings risks to the project and can reduce its quality. It is worthwhile to have automatic mechanisms to monitor it, as TD monitoring requires the analysis of large amounts of complex data. Therefore, the combination of software metrics and code comment analysis, in the identification, and information visualization techniques, in monitoring, present themselves as a promising strategy to manage TD. This work presents VisminerTD, a tool that allows the automatic identification and interactive monitoring of the evolution of TD items by combining software metrics, code comment analysis, and information visualization. To evaluate its applicability, a feasibility study was carried out considering JUnit 4 and Apache Ant software projects. The results indicated that VisminerTD can support software development teams in monitoring TD items. In addition, a second case study was performed to assess the feasibility of the proposed tool regarding its usefulness, ease of use, and self-predicted future use. The results provided positive evidence on the use of the proposed tool, indicating (i) that it can be useful in supporting TD Identification and TD monitoring activities and (ii) that it can bring gains in terms of comprehensiveness and efficacy when evaluating the desirable time to identify and monitor different types of debt. Given the current scenario characterized by limited options of tools that combine different information to support automatic identification and monitoring of the evolution of TD items in software projects, VisminerTD can approximate the state-of-the-art and the state-of-the-practice in the TD area, contributing to a wider dissemination of the concept.

Keywords: Technical debt, Technical debt identification, Technical debt monitoring, Software evolution, Software visualization

Introduction

The development of software systems is increasingly challenging, as they are getting bigger and more complex, delivering more functionalities, and interacting much more with other systems. The quality of software systems that goes through evolution activities often decreases over time when considering aspects such as their internal structure, adherence to standards, documentation, and ease of understanding for future maintenance [1–3]. One reason for this is that development activities are often carried out under severe constraints of time and resources. Due to these constraints, tasks need to be prioritized and many of them are left behind, thus generating technical debt (TD).

The concept of TD has helped professionals and researchers to discuss those issues associated with software evolution [4]. The concept of technical debt (TD) contextualizes problems faced during software evolution considering the tasks that are not carried out adequately during software development. Software TD is a type of debt that brings a short-term benefit (e.g., increased

*Correspondence: thiagosouto@ifba.edu.br
†Thiago S. Mendes and Felipe G. S. Gomes contributed equally to this work.
[1]Federal University of Bahia, UFBA, Av Adhemar de Barros, s/n, Instituto de Matemática, 40170-110 Salvador, Brazil
Full list of author information is available at the end of the article

development speed or shortened time to market), but which may have to be paid with interest later on in the software development life cycle [4–6]. Non-execution of tests, pending code refactoring, and outdated documentation are examples of TD. More recently, Avgeriou et al. [7] defined TD as "TD is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. TD presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability." The use of such TD concept facilitates the understanding of the costs incurred unintentionally or by decisions taken during the software development life cycle [8, 9].

It is common for a software project to incur TD during the development process. However, its presence brings risks to the project and hinders its management since managers will have to decide if the debt will be paid and, if so, which debt should be paid and when. Effects of TD can be noticed in different stages of software development due to different types of debt [10]. Its presence makes it difficult to add new functionalities, creates a favorable environment for the occurrence of defects, impacts on external quality, and reduces the maintainability of the code [11]. TD identification strategies can be performed either manually, automatically, or computer-assisted. According to Zazworka et al. [12], it is recommended to combine two approaches (manually and automatically). However, there are still few options that combine different information extracted by calculating software metrics, code smells, static code analysis, and source code comment analysis to support automatic identification and monitoring of the evolution of TD items in software projects [13]. Another aspect to be considered is consider using software visualization techniques to support the identification and management of TD items because such techniques can be used to support tasks of understanding, maintaining, and evolving software systems [14, 15]. However, these techniques have only been used in a very limited way with this purpose so far [10].

This work presents VisminerTD[1], a tool whose goal is to support the activities of identification and monitoring of TD using software visualization resources. VisminerTD implements a new TD identification strategy by combining information extracted from software metrics and source code comments, monitoring the evolution of TD items through different versions of the software.

VisminerTD uses software metrics, duplicated code occurrences, style problems, ASA issues, and code comments to find TD indicators. The combined analysis of TD indicators allows development teams to identify more precisely TD items in the source code, calling their attention to specific parts of the project. Once a TD item is detected, VisminerTD allows the user to follow the evolution of that item and its indicators over the project life cycle. Its visual metaphors support the monitoring of the evolution of TD items through the versions of a software, identifying when they incur, and if their indicators are increasing or if they are being paid off.

To evaluate the proposed tool regarding its support to TD identification and monitoring activities, we performed two feasibility studies. In the first study, two well-known open-source projects were used, JUnit 4 and Apache Ant, both mature, large, and with many contributors. The results of the study indicated that the tool can support software engineers in monitoring debt items. The study also confirmed that the current version of the tool can process information extracted from large software project repositories in a few minutes (about 20 min for two large repositories), performing tasks of repository mining and metrics calculation. In the second study, the proposed tool was evaluated through a feasibility study by using the Technology Acceptance Model (TAM) [16]. The results provided positive evidence on the use of VisminerTD and indicated that it is capable of supporting TD identification and TD monitoring.

In addition to this introduction, this article has six more sections. The "Background" section presents background information on TD, metrics, comments, and software visualization. "VisminerTD" section describes the architecture and functionalities of VisminerTD. The "Feasibility study I" section discusses the first feasibility study. The "Feasibility study II" section details the planning, execution, and results of the second case study. Then, related works are analyzed in the "Comparison to related works" section. Finally, the "Final remarks" section concludes the paper and indicates some future work.

Background

This section provides an overview of relevant background on TD and reviews past work on TD indicators considering software metrics, code smells, ASA issues, and source code comments. It closes by discussing software visualization in the context of this work.

Technical debt

TD is seen as an important part of software management [11]. According to Seaman and Guo [4], the management of TD can center on a TD list. This list should contain TD items that represent tasks that were left undone, but that run a risk of causing future problems if not completed. The main component of the debt, the principal, refers to the cost to eliminate the debt (i.e., the effort required to complete the task). Depending on the type of TD, this can translate into different kinds of activities, such as updating outdated documentation, refactoring code that is hard to maintain, or defining new test cases to improve their coverage. The second main component of TD is interest, which is composed of two parts: (i) the interest amount is the potential penalty in terms of increased effort and decreased productivity that will have to be paid in the future as a result of not completing these tasks in the present [4], including the extra cost of paying off the debt later, as compared to earlier, and (ii) the interest probability, because TD will not always bring negative impacts on future project activities. For example, the higher the probability that the artifact that contains the debt will undergo maintenance, the higher the probability that the interest will negatively impact the project.

Alves et al. [10], through a systematic mapping study, identified 15 different types of debt that could affect the software development and the relation with their respective indicators. The focus of this work is on supporting the identification and monitoring of nine types of debt as listed below. According to Alves et al. [10], these types are among the most common ones in the area:

- Architecture debt: refers to problems encountered in the software architecture, for example, violation of modularity, which can affect architectural requirements such as performance and robustness. Normally, this type of debt cannot be paid off with simple interventions in the code, requiring more extensive development activities [6, 17].
- Build debt: refers to issues that make the build task harder, and unnecessarily time consuming, such as when the build process needs to run ill-defined dependencies and the process becomes unnecessarily slow. It also incurs when the build process involves code that does not add customer value [18].
- Code debt: refers to problems found in the source code that can negatively affect the legibility. Usually, this debt can be identified by examining the source code for issues related to bad coding practices [19].
- Defect debt: refers to known defects, usually identified by testing activities or by the user and reported on bug tracking systems, that the Configuration Control Board (CCB) agrees should be fixed but, due to competing priorities and limited resources, have to be addressed at a later time [20].
- Design debt: refers to debt that can be discovered by analyzing the source code and identifying violations of the principles of good object-oriented design (e.g., very large or tightly coupled classes) [5, 21].
- Documentation debt: refers to problems found in the project documentation such as missing, inadequate, or incomplete documentation of any type [21].
- People debt: refers to people issues that, if present in the software organization, can delay or hinder some development activities. An example of this kind of debt is expertise concentrated in too few people, as an effect of delayed training or hiring [22].
- Requirement debt: refers to trade-offs made with respect to what requirements the development team needs to implement or how to implement them. Examples of this type of debt are partially implemented requirements or implementations that do not fully satisfy a non-functional requirement [6].
- Test debt: Refers to issues found in testing activities that can affect the quality of those activities. Examples of this type of debt are planned tests that were not run, or known deficiencies in the test suite (e.g., low code coverage) [21].

The next section discusses indicators that can be used to automatically find TD items.

Technical debt indicators

TD indicators allow to discover technical debt items by analyzing different artifacts created during the development of a software project. Most TD indicators proposed in the literature can be related to calculation of software metrics [10, 13].

Metrics are used to ensure control over software projects, products, and processes [23]. More specifically, they allow the assessment of attributes, features, or characteristics of software entities, making it possible to characterize, monitor, and control them. In the context of object-oriented programming, the set of metrics proposed by Chidamber and Kemerer [24] makes it possible to characterize, for example, the size, complexity, and coupling of the code.

In the literature, there are many studies that address the detection of code smells or code anomalies through code metrics as a way to identify problems in the source code [25]. Code smells emerge from choices, in the design of a system, that do not comply with widely accepted principles of good design [12]. They are an initial indication that there may be a serious problem in the system [26]. These problems may hinder the software evolution and maintenance process and lead to code refactoring [27]. Code smells are a well-known indicator of the presence of code and design debt [10].

Automatic Static Analysis (ASA) tools can also be used in the TD identification activity. ASA tools allow the analysis of source code in search for violations of good programming practices that can cause failures or hinder some quality dimension of the software (for example, its maintainability or efficiency). Some of these violations can be removed through refactoring to avoid future problems [12]. As an example, FindBugs [28] is a static open-source code analyzer that detects possible bugs in Java programs. Potential errors are divided into categories, and the tool provides tips to the developer about their possible impacts. Checkstyle is another ASA tool [29]. It is used in software development to verify if the source code complies with coding rules.

Source code comments are another valuable resource to help understand a software system [30]. They are widely used by maintainers involved in evolution management. Comments can describe issues that require future work, or indicate emerging issues and decisions that need to be taken on them. They facilitate human readability and provide additional information that summarizes the developer context [31]. Recently, some work has been performed considering the identification of TD items through the analysis of code comments [31–34]. In Farias et al. [33], the authors presented eXcomment, a tool for identifying and classifying TD through code comment analysis. The tool is based on text mining, a technology to extract useful information from large amounts of unstructured textual data. The tool uses a contextualized vocabulary composed of expressions and terms that locate TD indicatives in the source code [31].

As discussed above, software metrics, ASA issues, and code comments have been used as TD indicators. However, as far as we know, these indicators have been used mostly in isolation during the identification of TD items. VisminerTD, the tool presented in this paper, makes use of a new strategy that combines information generated by software metrics, code smells, style problems in Java code, ASA issues, and source code comments to improve the identification of TD items.

Software visualization

Software evolution of real projects with many developers commonly produces a large amount of data. Daily, developers have to perform maintenance tasks to keep the software up-to-date. A prerequisite for those maintenance activities is the software comprehension. Unfortunately, this is not a simple task: developers have to deal with several software modules, which are, in many cases, comprised of thousands of lines of code.

To deal with comprehension in this complex scenario, researchers and practitioners have been using software visualization (SoftVis). SoftVis can be defined as the mapping from any kind of software artifact (lines of code, method, class, etc.) to graphical representations [35, 36]. It is very helpful since it transforms intangible software entities and their relationships into visual metaphors that are easily interpreted by human beings [37].

The use of information and software visualization techniques [38, 39] have been used in software engineering as a possible solution to facilitate the understanding of the software, supporting the tasks associated with the maintenance and evolution of systems [40]. Software visualization uses visual resources to facilitate the understanding of information extracted from software metrics by software engineers [41]. Consequently, software visualization techniques are also a promising way to deal with debt items.

Researchers proposed different classification taxonomies for SoftVis. Diehl [39] divides software visualization into visualizing the structure, behavior, and evolution of the software. Other authors classified the visualization based on the type of the metaphors it uses to represent software [42, 43]. Software can also be visually analyzed from different perspectives [44].

In the context of software evolution, it is also important to consider visual strategies of analysis, that is defined as how the evolution is visually presented for analysis [45]. The strategy can allow the user to navigate in the history of a software element, making it possible to compare two versions of software modules, comparing two versions of a class, for example. Software evolution visualization tools should consider the combined use of different strategies as a promising approach [45].

In the next section, VisminerTD is presented, a multi-perspective and multi-visual strategy of analysis tool that combines information from software metrics, code smells, ASA issues, and source code comments to identify TD and allow the use of software visualization techniques to monitor existing debt items.

VisminerTD

VisminerTD is an open-source web tool that has multiple visual perspectives and different visual strategies of analysis to support development teams in activities of identifying and monitoring the evolution of TD items. VisminerTD runs on top of RepositoryMiner[2] (RM). The RM is responsible for metrics extraction and comments from source code repositories [46]. Figure 1 presents a component and connector architectural view of the whole solution, considering both VisminerTD and RM. They are further explained next.

RepositoryMiner

As previously discussed, all data presented in VisminerTD are extracted and analyzed by the RM, which is an extensible tool for mining software repositories to support automatic identification of TD. Developed as an open source project using Java and MongoDB [47], the main purpose of RM is to perform analysis of software repositories, through the extraction and combination of data related to software evolution. RM is distributed in the form of a JAR (Java ARchive) and provides an API (Application Programming Interface), so that its features can be easily accessed through Java applications.

The RM mining process bases on the list of TD indicators defined by Alves et al. [10] (e.g. code smells, ASA issues, style problems in code and software documentation issues). This means that the extracted data is related

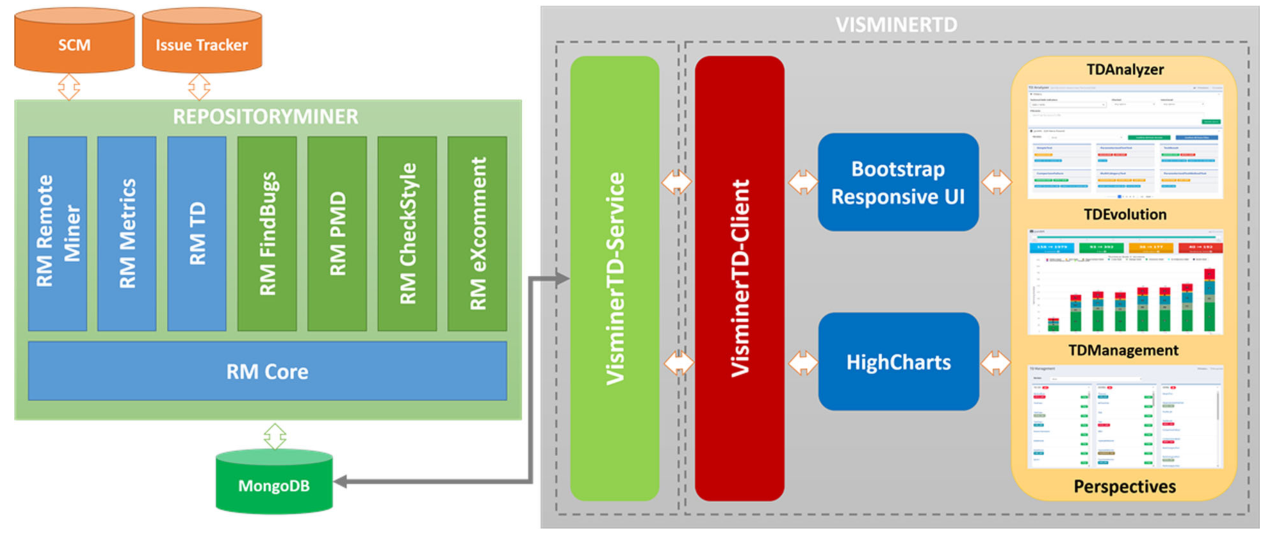


Fig. 1 VisminerTD architectural view. Presents a component and connector of VisminerTD architectural view

to the set of metrics, rules, and other information needed by the indicators. In this sense, the main RM features are:

- Mining local (e.g., projects stored on GIT) and remote software repositories (e.g., issues and milestones stored on github.com)
- Calculating 19 software metrics: AMW (Average Method Weight), ATFD (Access To Foreign Data), CYCLO (McCabe's Cyclomatic Number), FDP (Foreign Data Provider), LAA (Locality Attribute Accesses), LOC (Lines of Code), LVAR (Number of Local Variables), MAXNESTING (Maximum Nesting Level), MLOC (Method Lines of Code), NOA (Number of Attributes), NOAM (Number of Accessor Methods), NOAV (Number of Accessed Variables), NOM (Number of Methods), NOPA (Number Of Public Attributes), NProtM (Number of Protected Members), PAR (Number of Parameters), TCC (Tight Class Cohesion), WMC (Weighted Method Count), and WOC (Weight Of a Class)
- Detecting seven types of code smells: brain class, brain method, conditional complexity, data class, feature envy, god class, and long method
- Detecting duplicated code using the Copy/Paste Detector (CPD) functionality of the PMD tool [48]
- Detecting possible defects in Java projects through static code analysis using the FindBugs tool [28]
- Detecting style problems in Java code using the CheckStyle tool [29]
- Detecting comments in Java code that contain some indication of the existence of TD items using the eXcomment tool [33]
- Detecting nine TD types: architecture, build, code, design, defect, documentation, requirements, people, and test

The extraction of the metrics and code smells was implemented considering the definitions found in Fowler [26] and Lanza and Marinescu [25]. After the mining process, the data is stored in a MongoDB database. Fig. 2 presents a simplified view of the RM database model.

VisminerTD

VisminerTD uses the data processed by RM to visually support TD identification and monitoring. VisminerTD consists of two modules: VisminerTD-Client and VisminerTD-Service. VisminerTD-Client is a web application developed using Angular [49], Bootstrap [50], and HighCharts [51]. Whenever it is necessary to search for some information mined by the RM, VisminerTD-Client sends an HTTP request to VisminerTD-Service, passing the desired action and necessary parameters. VisminerTD-Service was developed as a REST (Representational State Transfer) service using Node.js [52], Express [53], and Mongoose [54]. The service receives the request, filters the action to be taken, and obtains/modifies the database information. VisminerTD-Service returns the request data, using JSON specification, which is used by VisminerTD-Client to process and present them. The amount of data regarding the identification and monitoring of TD is large, so, VisminerTD uses software visualization techniques to reach its goal. We used HighCharts to develop the set of views available in VisminerTD. Those views are explained next.

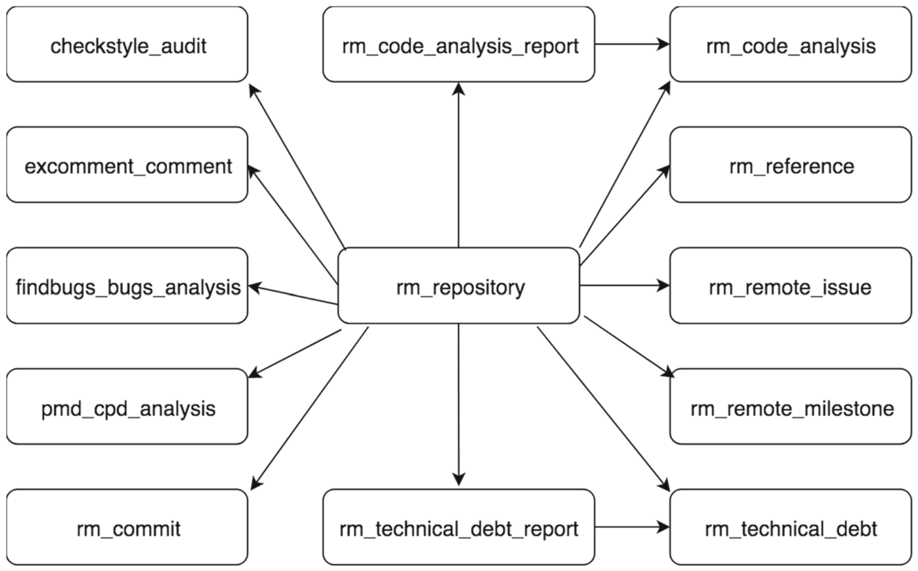


Fig. 2 RepositoryMiner database. Simplified RepositoryMiner database model view

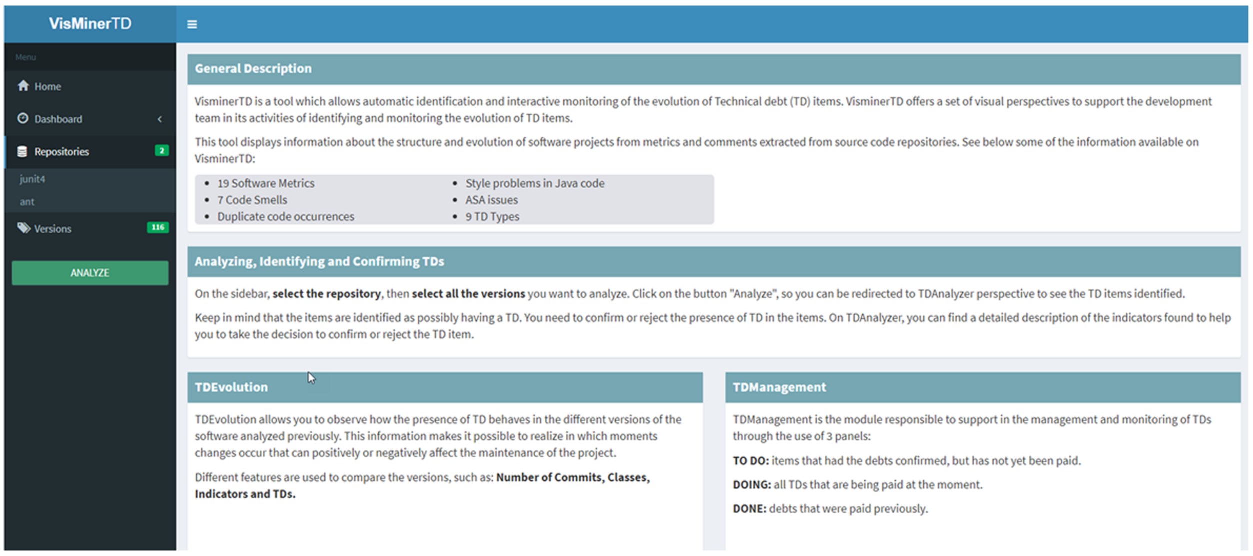
VisminerTD views

To support the identification and monitoring of TD items, VisminerTD consists of the following set of views: Home View, Sidebar View, TDAnalyzer View, TDEvolution View, and TDManagement View. The first two views are related to the configuration of the environment. The last three views are responsible for the identification and monitoring of technical debt.

The *Home View* (Fig. 3) shows a set of instructions about the general features of the tool. The *Sidebar View* (left side of Fig. 3) allows the selection of the repository to be analyzed, as well as their respective versions, and access to the tool modules. RM extracts the presented versions from the selected software repository. Only selected versions will be considered by VisminerTD.

TDAnalyzer View

The *TDAnalyzer View* allows the identification of TD items. It uses as input the RM extracted data (e.g., code smells, style problems in Java code, ASA issues, source code comments) as debt indicator information. The TDAnalyzer shows the TD items detected (Fig. 4).



General Description

VisminerTD is a tool which allows automatic identification and interactive monitoring of the evolution of Technical debt (TD) items. VisminerTD offers a set of visual perspectives to support the development team in its activities of identifying and monitoring the evolution of TD items.

This tool displays information about the structure and evolution of software projects from metrics and comments extracted from source code repositories. See below some of the information available on VisminerTD:

- 19 Software Metrics
- 7 Code Smells
- Duplicate code occurrences
- Style problems in Java code
- ASA issues
- 9 TD Types

Analyzing, Identifying and Confirming TDs

On the sidebar, **select the repository**, then **select all the versions** you want to analyze. Click on the button "Analyze", so you can be redirected to TDAnalyzer perspective to see the TD items identified.

Keep in mind that the items are identified as possibly having a TD. You need to confirm or reject the presence of TD in the items. On TDAnalyzer, you can find a detailed description of the indicators found to help you to take the decision to confirm or reject the TD item.

TDEvolution

TDEvolution allows you to observe how the presence of TD behaves in the different versions of the software analyzed previously. This information makes it possible to realize in which moments changes occur that can positively or negatively affect the maintenance of the project.

Different features are used to compare the versions, such as: **Number of Commits, Classes, Indicators and TDs.**

TDManagement

TDManagement is the module responsible to support in the management and monitoring of TDs through the use of 3 panels:

TO DO: items that had the debts confirmed, but has not yet been paid.

DOING: all TDs that are being paid at the moment.

DONE: debts that were paid previously.

Fig. 3 VisminerTD Home view. View that shows a set of instructions about the general features of the tool

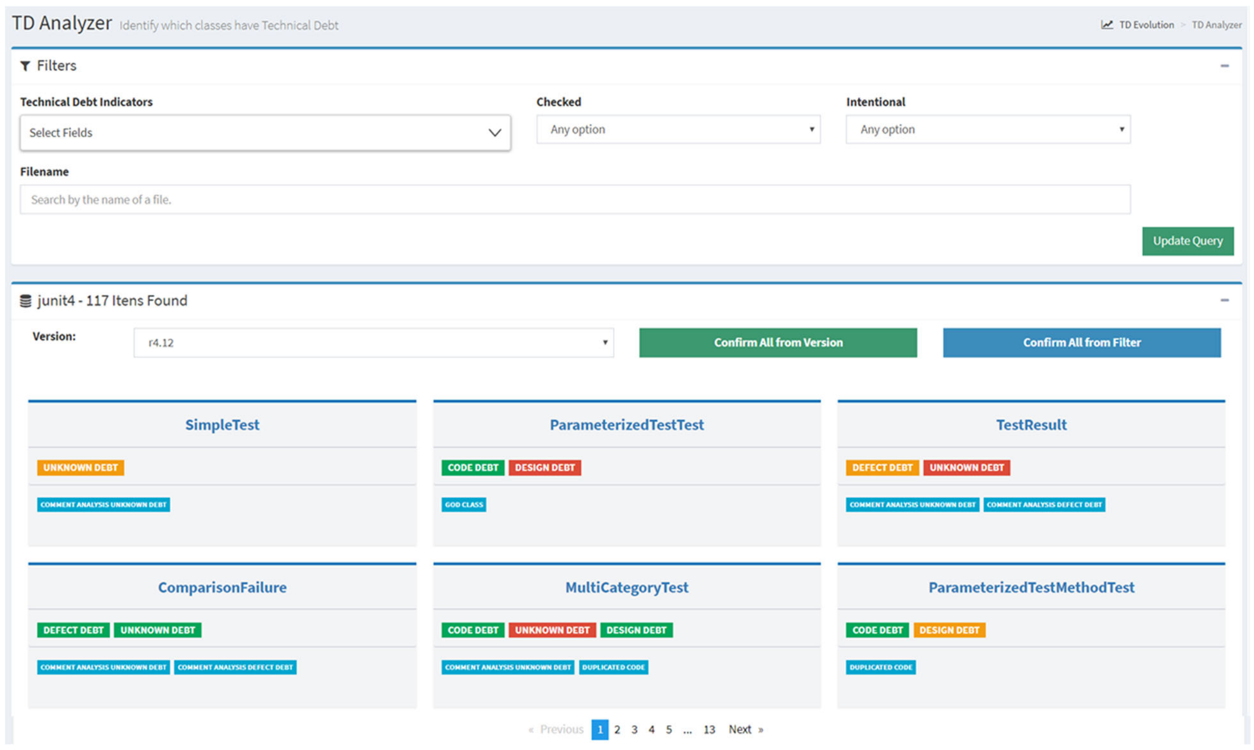


Fig. 4 TDAnalyzer perspective. Perspective that allows the identification of TD items

Initially, the user has to choose which versions of the software they want to analyze through the "Version" field. Remember that, only the selected versions in the Sidebar View are available in this field. After choosing a version, all TD items will immediately appear below, in the form of cards (Fig. 4).

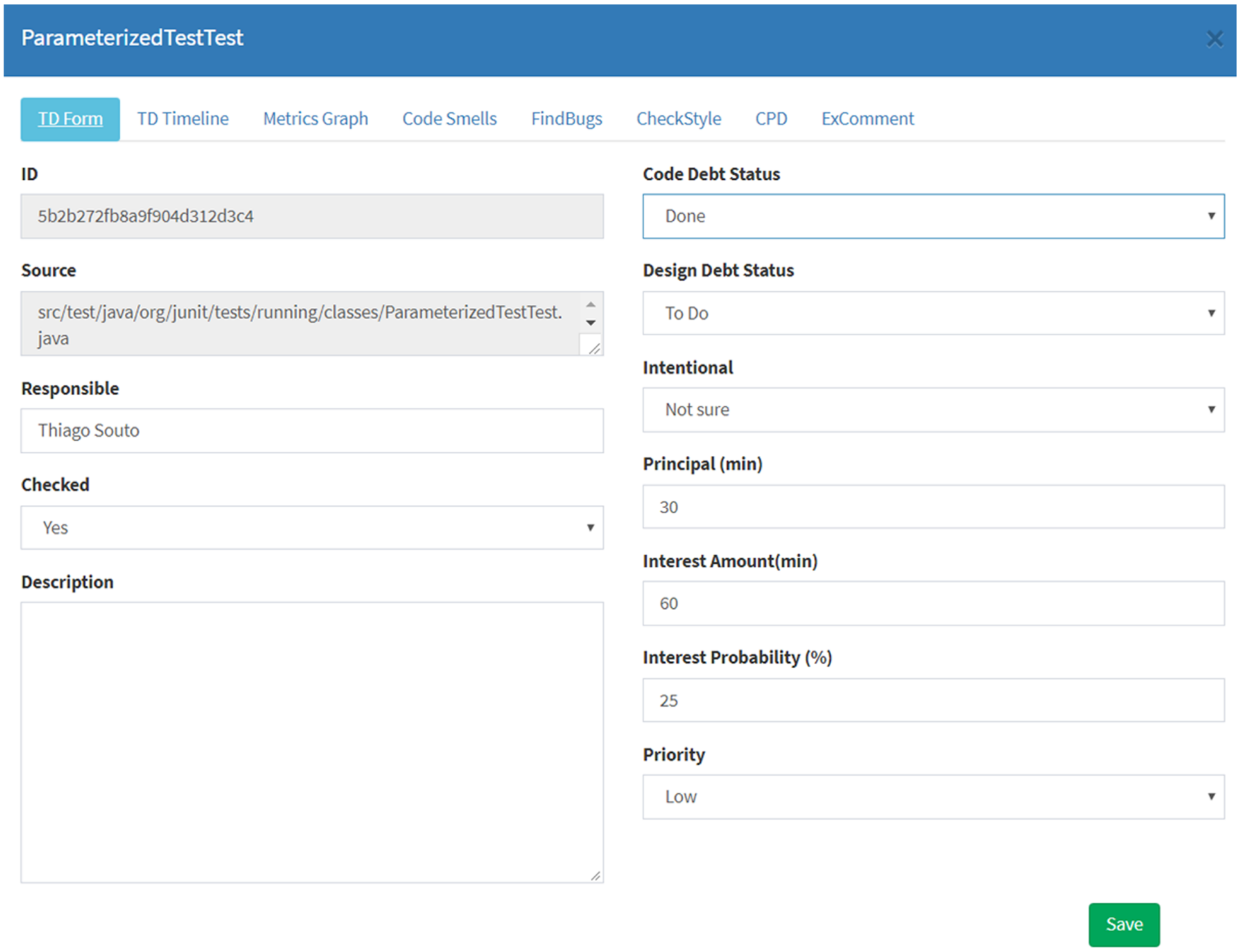
In addition to viewing all TD items for a version, the user can also use filters in the Filters box (Fig. 4, top) to refine the search, such as: (i) *Technical Debt Indicators:* to select the items according to the indicators found. The default shows all of them; (ii) *Checked:* to choose items that have been marked as checked, unchecked or both; (iii) *Intentional:* to select TD items that are intentional, unintentional, those that are not sure of intent or with any value; and (iv) *Filename:* to search for TD items by file name. The user can combine the filters if necessary.

The card contains the name of the file, the type(s) of debt, and the name of the TD indicators identified. The TD types on the cards have their colors set according to their status. The possible statuses are: (i) *Not Analyzed* (gray); (ii) *False Positive* (green); (iii) *To Do* (red); (iv) *Doing* (yellow); and, (v) *Done* (green).

The user can confirm or not each TD item identified automatically by RM. It can be done one by one, using the TD form (explained below), confirm all items indicated for a given version ("Confirm All from Version" button), or for all items according to the selected filters ("Confirm All from Filter" button). In any case, the VisminerTD changes the status of the TD types from "Not Analyzed" to "To Do".

The TDAnalyzer has a detail-on-demand view where the user can analyze detailed information of a TD item. The user can access the detail-on-demand view by clicking on the TD item name (i.e., the file name). The detail-on-demand view is composed of eight pages: TD Form, TD Timeline, Metrics Graph, Code Smells, FindBugs, CheckStyle, CPD, and eXcomment. These pages bring extra information about the identified TD indicator, allowing the user to decide if that debt should be considered a problem or not.

TD form In this page, the user has access to a form that allows the visualization and edition of information pertinent to the management of the TD item (Fig. 5). The construction of this form was based on the study of Guo et al. [55], which addresses the management of TD in software projects. The TD form contains the following fields: (i) *ID*: hashcode that uniquely identifies the TD item detected; (ii) *Source*: the file path in the repository; (iii) *Responsible*: the name of the person responsible for the analysis of the TD item; (iv) *Checked*: indicates whether the TD item has already been analyzed; (v) *Description*: general notes/observations with relevant information; (vi) *Debt Type Status*: for each type of TD identified indicates its status, and may have the following values: Not Analyzed



ID	5b2b272fb8a9f904d312d3c4
Source	src/test/java/org/junit/tests/running/classes/ParameterizedTestTest.java
Responsible	Thiago Souto
Checked	Yes
Description	
Code Debt Status	Done
Design Debt Status	To Do
Intentional	Not sure
Principal (min)	30
Interest Amount(min)	60
Interest Probability (%)	25
Priority	Low

Fig. 5 TD form page. Page that allows the visualization and edition of information pertinent to the management of the TD item

(not yet checked, it is the default value); False Positive (the item is not a TD), To Do (it is a TD to be paid), Doing (TD is being solved) and Done (TD has already been resolved); (vii) *Intentional*: indicates whether the TD item was intentional; (viii) *Principal*: the estimated amount of time (in minutes) required for correction; (ix) *Interest Amount*: the estimated amount of time (in minutes) required for correction in production code (in other words, in the future); (x) *Interest Probability*: indicator (percentage) of the chances of paying interest in the future; and finally, (xi) *Priority*: correction priority (high/medium/low) indicator of the item.

TD Timeline In this page, the user can get an insight into the evolution of the TD item across the various versions of the software (Fig. 6). The analysis can be performed from the initial versions of the software to the current version being analyzed (selected in the "Version" field in TDAnalyzer).

All TD indicators identified in the file in each of the software versions are displayed in the timeline. For each version, it presents its name, the TD types (colored according to their status in each version), and the indicators with their respective amounts of occurrences. A circle just below the name represents a comparison of that version with the previous one considering the amount of TD indicators. A gray circle means that the number of indicators remains the same as the previous version, a red circle means that TD indicators increased from one version to another, and the green color indicates that the number of indicators has decreased.

Metrics Graph The Metrics Graph page presents a line graph that shows the evolution of some of the metrics used to indicate TD throughout software versions (Fig. 7). Since a file can consist of one or more classes, the user can filter the class ("Classes" field) and which metric to display ('Metrics' field). The metrics supported so far are

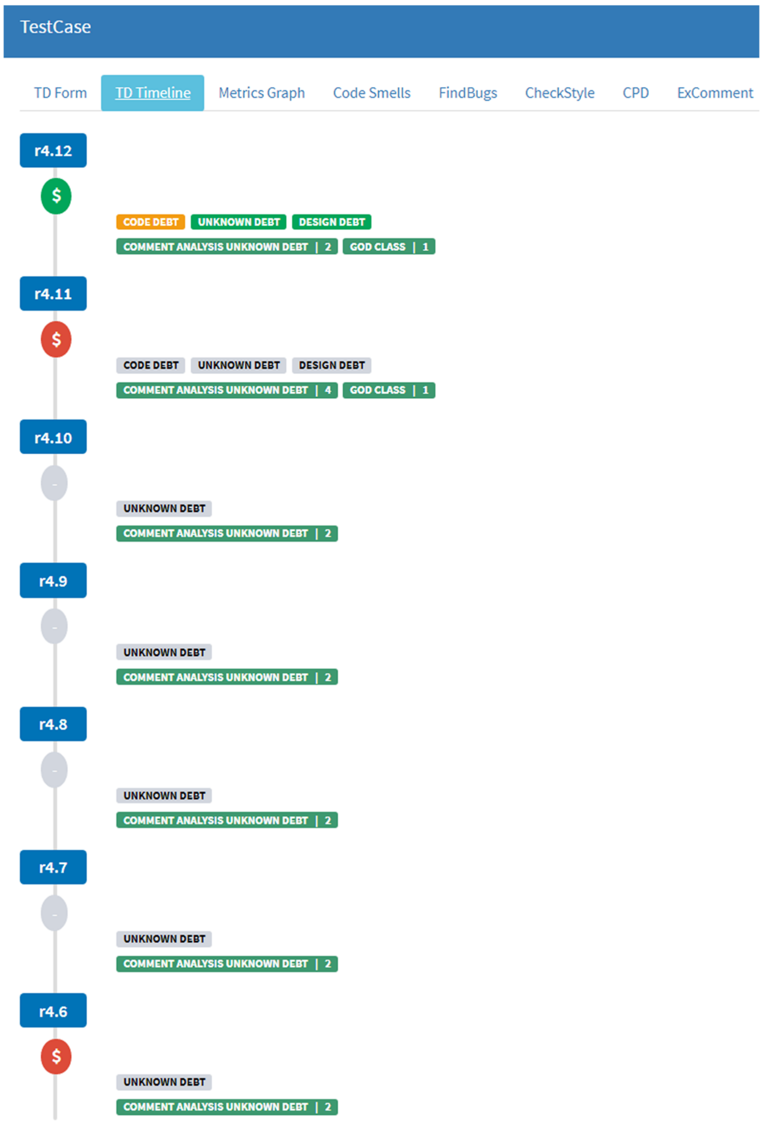


Fig. 6 TD Timeline page. Page that shows the evolution of the TD item across the various versions of the software

the ones related to classes: AMW, AFTD, LOC, NProtM, NOA, NOPA, NOM, NOAM, TCC, WOC, and WMC. The Metrics Evolution view shows analyzed data from the first version of the software to the version being analyzed. In this way, the user can analyze the evolution of all the calculated metrics and verify in which version there were changes in the values of the metrics. This visualization uses a temporal overview strategy [45], which shows several versions of software at the same time.

Code Smells The Code Smell page allows the user to view the code smells detected in the file (Fig. 8). The user can select classes ("Classes" field) and methods ("Methods" field). Then, the Code Smell view displays the metric values (or code smells, if applied) for each of the detected code smells. Only the classes and methods affected by at least one smell are displayed in the selection fields.

FindBugs This page (see Fig. 9) allows the user to visualize the information about possible bugs found by the FindBugs tool. VisminerTD shows each possible bug identified and its associated information (class, field, method, local variable). We also have other related information: (i) *Rank*: refers to the bug rank, the potential bugs are classified into four ranks: (a) scariest, (b) scary, (c) troubling, and (d) of concern, giving the programmer a hint about the impact or severity of the bug; (ii) *Priority*: refers to

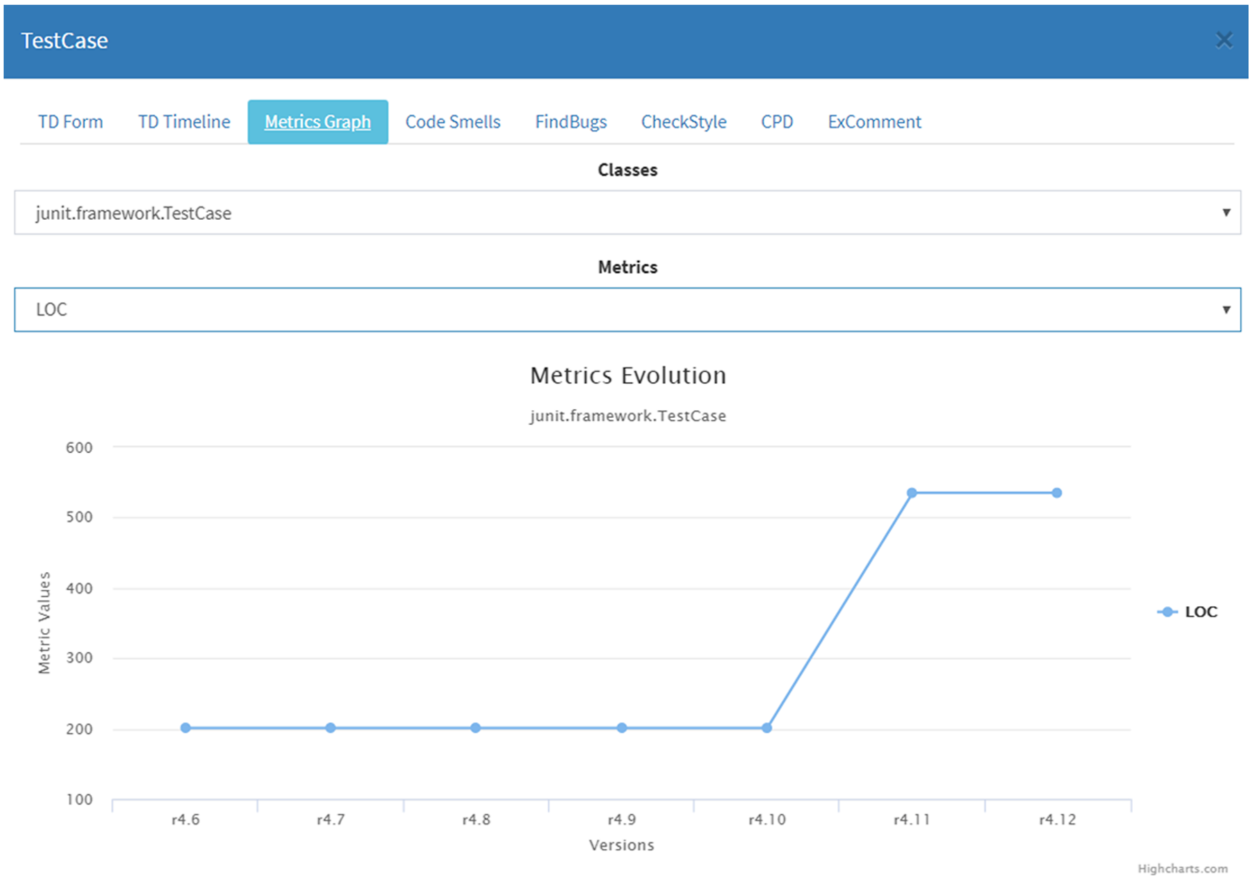
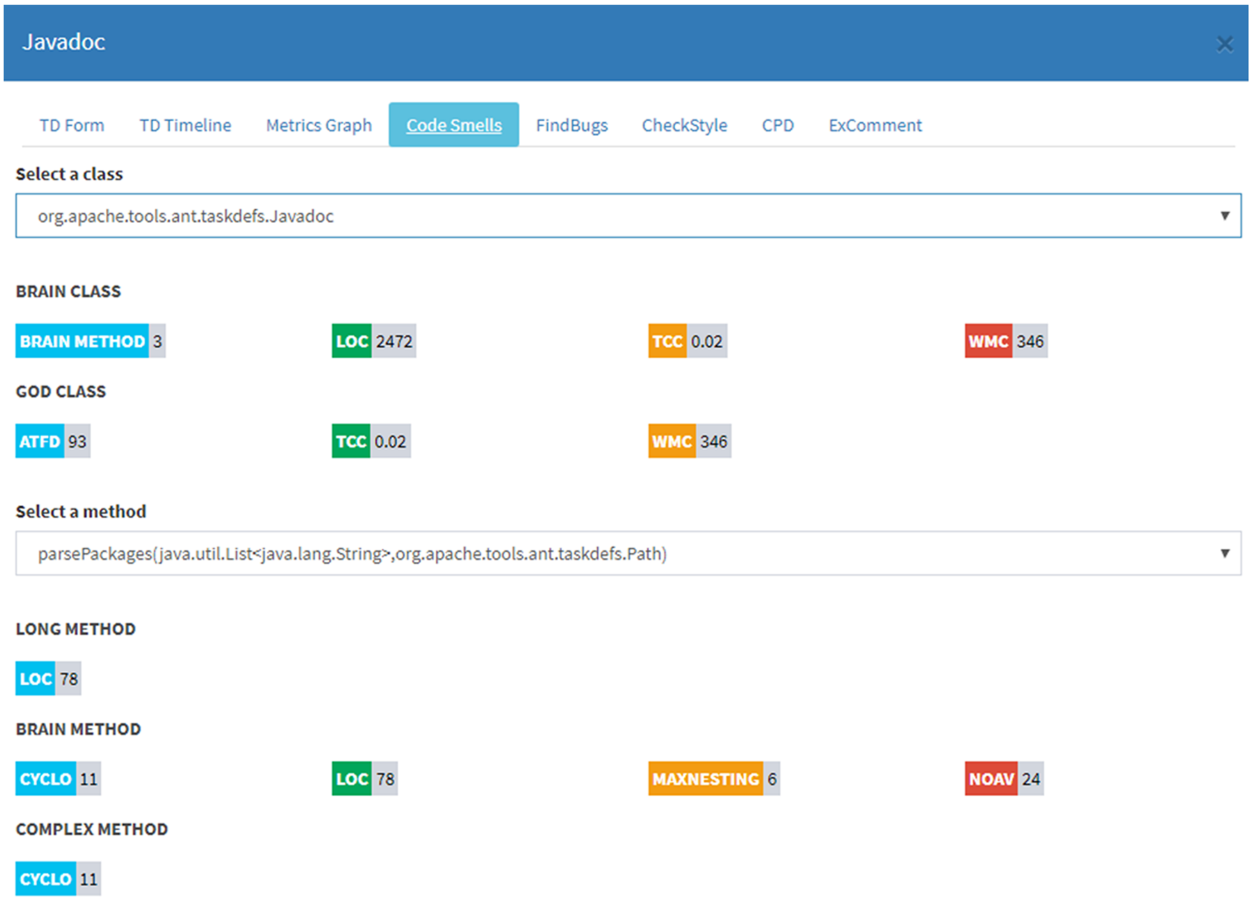


Fig. 7 TD Metrics Graph page. Page that presents a line graph to show the evolution of some of the metrics used to indicate TD throughout software versions

the priority of the problem resolution. The higher the priority the greater the impacts of the bug on the system; (iii) *Type*: the bug type. There are currently 425 classified bugs arranged in several categories; (iv) *Category*: the category of the bug, so far the categories are: "bad practice", "correctness", "experimental", "internationalization", "malicious code and vulnerability", "multithreaded correctness", "performance", "security" and "dodgy code"; (v) *Description*: a detailed description of the bug and its impact; and (vi) *Message*: a message to help the developer to solve the problem, usually presenting the location of the problem, and sometimes some tips on how to solve it.

CheckStyle This section shows the information extracted from the Checkstyle tool (Fig. 10). Through this page, the user can visualize the following information: (i) *Line*: the line where the problem occurred; (ii) *Column*: the column where the problem occurred, if it is zero, consider the entire line; (iii) *Severity*: the severity of the problem which may be: "error", "ignore", "info", and "warning"; (iv) *Checker*: the name of the rule used to identify the problem; and finally (v) *Message*: a message describing the violation in detail.

CPD This page shows all detected PMD/CPD duplicated code occurrences per file (Fig. 11). The duplication occurrences can be found within the same file or across different files in the project, where each duplication occurrence represents a duplicated segment of code. At the top of the screen (in "Total Duplication" field), the percentage of duplicated file is shown. Duplicate occurrences are shown below. The following information about occurrences is also provided: (i) *Tokens Threshold*: refers to the minimum amount of duplicate tokens present in a segment to consider it as duplicate; (ii) *Token Count*: refers to the number of duplicate tokens present in the occurrence; (iii) *Language*: the programming language considered for verifying duplicity; (iv) *Occurrences*: refers to the number of different files and/or sections of the same file where the duplication was found, and for each section, the fields below are shown; (v) *Filename*: the path of the file that contains the duplication; (vi) *Begin Line*: the line on which



Select a class

org.apache.tools.ant.taskdefs.Javadoc

BRAIN CLASS

BRAIN METHOD 3 LOC 2472 TCC 0.02 WMC 346

GOD CLASS

ATFD 93 TCC 0.02 WMC 346

Select a method

parsePackages(java.util.List<java.lang.String>,org.apache.tools.ant.taskdefs.Path)

LONG METHOD

LOC 78

BRAIN METHOD

CYCLO 11 LOC 78 MAXNESTING 6 NOAV 24

COMPLEX METHOD

CYCLO 11

Fig. 8 Code Smells page. Page that allows the user to view the code smells detected in the file

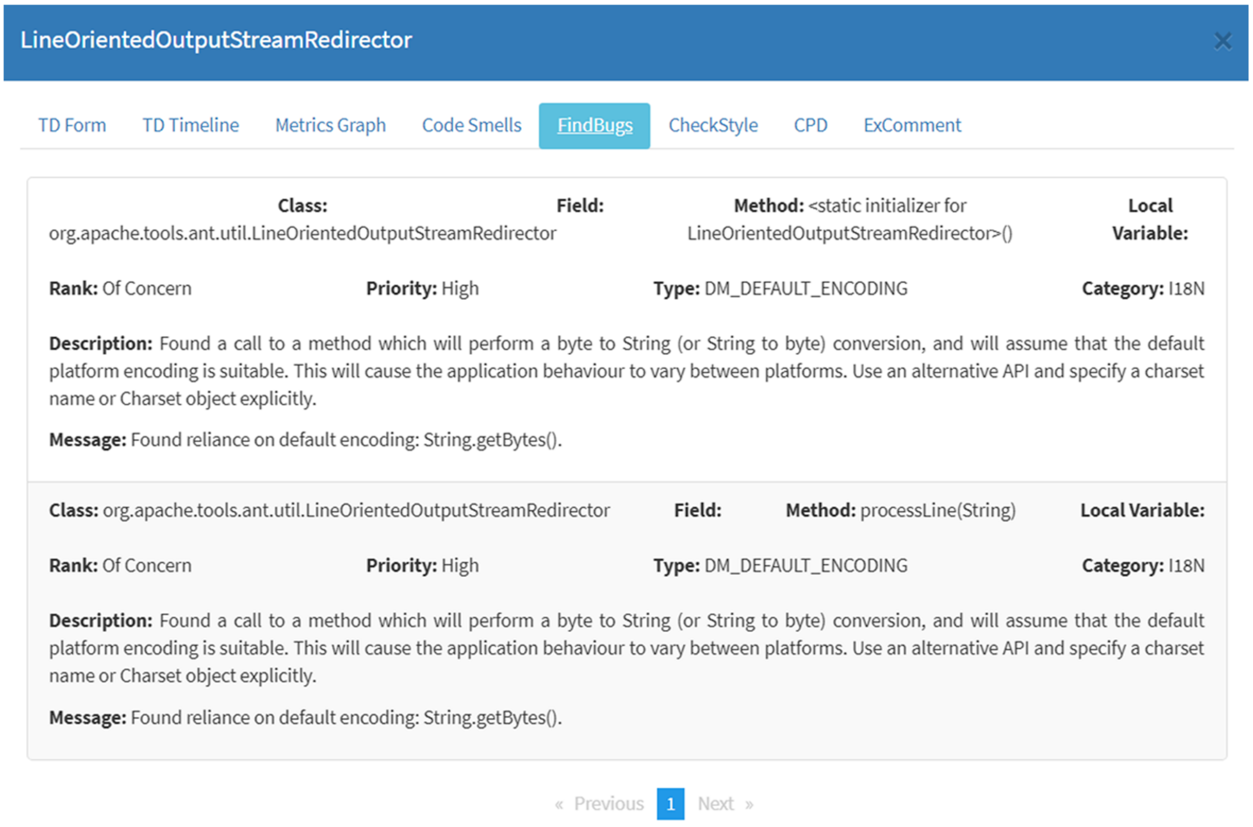
the duplicate portion begins; (vii) *End Line*: the line on which the duplicate portion ends; (viii) *Line Count*: the amount of duplicate lines; and (ix) *Duplication:* is the percentage of that duplicate stretch to the file; in other words, it is the ratio of the size of the stretch to the size of the file, where we consider size as the amount of characters.

eXcomment This page (Fig. 12) displays the list of comment information from the eXcomment tool presented in [31]. The user can view the following information regarding a comment: (i) *Comment text*: content of the comment in the source code, for example, "// ALL: Review all callers to make sure that they localize the title"; (ii) *Class/Method*: class and, in some cases, the name of the method where the comment is located; (iii) *Pattern*: set of word classes and TD code tags and software engineering terms (e.g., SE nouns, verbs, adverbs, adjectives, and tags), which led to the identification of that comment as a TD item, for example, "TODO", "Temporary", "Workaround", and "Bug". In a comment, there may be one or more patterns identified; (iv) *Theme*: set of vocabulary themes related to TD contexts, such as "Bad coding practices" and "Inadequate solution". It is important to note that a pattern can be related to more than one theme. The list of themes was created based on the list of TD indicators proposed by Farias et al. [33] and Alves et al. [10]; (v) *Total Score*: value calculated based on the occurrence and degree of importance of the patterns found in the source code comment. Patterns with larger scores are more likely to indicate TD items than patterns with smaller scores; and (vi) *TD Type*: TD type found based on the patterns and themes found, expressed as architecture debt, build debt, code debt, defect debt, documentation debt, design debt, requirement debt, test debt, or unknown debt.

As could be observed, TDAnalyzer has several features to support the identification of TD. Different pages, some of them with specific visualizations, help the user to decide the existence of a TD, by confirming it or not. Both TDEvolution and TDManagement benefit from the users' decisions while using TDAnalyzer.

TDEvolution View

The *TDEvolution View* allows the understanding of how software is evolving over time regarding identified TD



Class: org.apache.tools.ant.util.LineOrientedOutputStreamRedirector **Field:** **Method:** <static initializer for LineOrientedOutputStreamRedirector>() **Local Variable:**

Rank: Of Concern **Priority:** High **Type:** DM_DEFAULT_ENCODING **Category:** I18N

Description: Found a call to a method which will perform a byte to String (or String to byte) conversion, and will assume that the default platform encoding is suitable. This will cause the application behaviour to vary between platforms. Use an alternative API and specify a charset name or Charset object explicitly.

Message: Found reliance on default encoding: String.getBytes().

Class: org.apache.tools.ant.util.LineOrientedOutputStreamRedirector **Field:** **Method:** processLine(String) **Local Variable:**

Rank: Of Concern **Priority:** High **Type:** DM_DEFAULT_ENCODING **Category:** I18N

Description: Found a call to a method which will perform a byte to String (or String to byte) conversion, and will assume that the default platform encoding is suitable. This will cause the application behaviour to vary between platforms. Use an alternative API and specify a charset name or Charset object explicitly.

Message: Found reliance on default encoding: String.getBytes().

Fig. 9 Findbugs page. Detailing of information found by Findbugs

items (Fig. 13). Using this view, the user can visualize differences between two selected versions and identify when positive or negative changes occur and whether they affect the project evolution or not. TDEvolution uses a differential absolute strategy [45], which compares two versions of software at a time (Fig. 13, top).

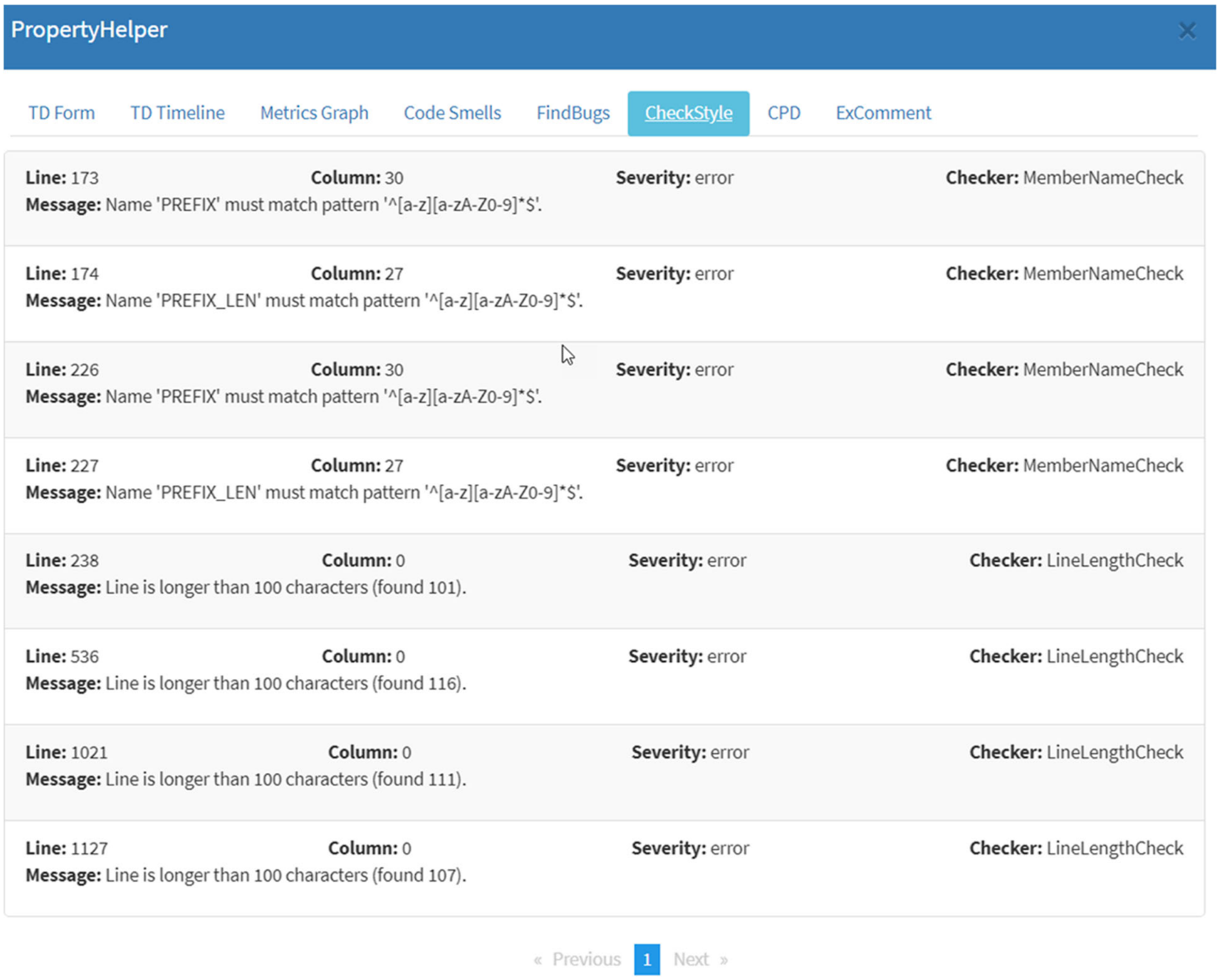
Among the information that can be compared from one version to another, we have the number of commits, source files, indicators (sum of the quantities of TD indicators found), and debts (sum of the quantities of TD types found) identified so far. We used a Stacked Column view to illustrate the evolution of TD throughout the versions. In addition to the comparison between two selected versions, TDEvolution View also shows the sum of the quantities of TD types found for each one of the versions between the two selected versions (Vertical bars at the bottom of Fig. 13). In the case of Fig. 13, the slider is placed for two subsequent versions, therefore, only two vertical bars are presented. This temporal overview strategy [45] view helps to analyze the evolution of the quantity of debts in various versions of the software separated by their types. The Stacked Column view divided by TD types helps a lot to compare releases.

This view only shows the count of TD types with status “To Do”, “Doing”, and “Not Analyzed”. The status “Done” and “False Positive” are not considered; this way the user can see the amount of TD left behind. Each color in the graph represents a type of TD, where the amount of occurrences are shown in its respective color (box), and above each bar, the sum of occurrences of all TD types in the version are displayed. To control the range of versions, the user can use a slider above the graph that goes from the oldest to the newest version of the software, from the left to the right.

Therefore, TDEvolution view helps users to monitor the evolution of TD in their projects.

TDManagement View

The TDManagement View is a module also responsible for assisting in TD monitoring (Fig. 14). The TD items are presented using the Kanban concept [56] with three panels (TO DO, DOING, and DONE). Each TD type of the file, identified by the type and name of the file, is represented as a card. In this view, the user can visualize the TD items of the project that had their debt types marked “To Do”, “Doing”, or “Done” in TDAnalyzer.



Line: 173 **Column:** 30 **Severity:** error **Checker:** MemberNameCheck
Message: Name 'PREFIX' must match pattern '^[a-z][a-zA-Z0-9]*$'.

Line: 174 **Column:** 27 **Severity:** error **Checker:** MemberNameCheck
Message: Name 'PREFIX_LEN' must match pattern '^[a-z][a-zA-Z0-9]*$'.

Line: 226 **Column:** 30 **Severity:** error **Checker:** MemberNameCheck
Message: Name 'PREFIX' must match pattern '^[a-z][a-zA-Z0-9]*$'.

Line: 227 **Column:** 27 **Severity:** error **Checker:** MemberNameCheck
Message: Name 'PREFIX_LEN' must match pattern '^[a-z][a-zA-Z0-9]*$'.

Line: 238 **Column:** 0 **Severity:** error **Checker:** LineLengthCheck
Message: Line is longer than 100 characters (found 101).

Line: 536 **Column:** 0 **Severity:** error **Checker:** LineLengthCheck
Message: Line is longer than 100 characters (found 116).

Line: 1021 **Column:** 0 **Severity:** error **Checker:** LineLengthCheck
Message: Line is longer than 100 characters (found 111).

Line: 1127 **Column:** 0 **Severity:** error **Checker:** LineLengthCheck
Message: Line is longer than 100 characters (found 107).

Fig. 10 CheckStyle page. Information of all code style problems found by CheckStyle

By clicking on the "pay" button, the status of the TD changes from "TO DO" to "DOING". On the same token, by clicking on the "paid" button, the status of the TD item switches from "DOING" to "DONE". This way, the user can manage each TD item separately. The user can select in the "Version" field the version that they want to monitor. From this view, the user can also access the TD Timeline (Fig. 6) and see the TD items evolution.

Feasibility study I

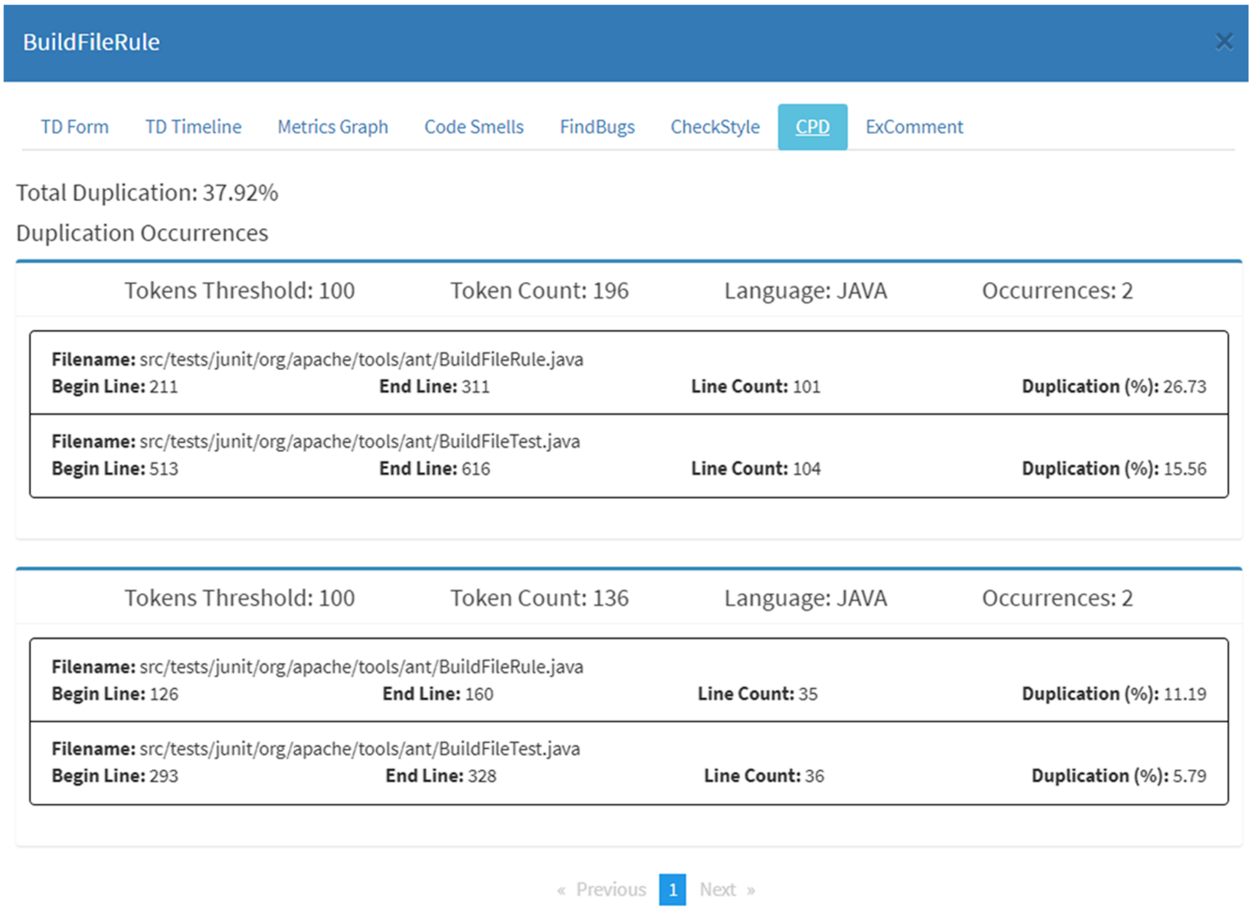
This section presents the study carried out with the objective of investigating the feasibility of using the developed tool. We evaluated whether VisminerTD, through the use of software visualization, allows one to perform activities of *identification* and *monitoring* of TD items.

Study objective

This study was performed aiming to analyze the VisminerTD: *with the purpose of* characterizing its effectiveness of use; *with respect to* support activities of identification and monitoring of TD; *from the point of view of* researchers; and *in the context of* software development projects. Thus, through this feasibility study, we intend to investigate if (i) VisminerTD's features are able to perform activities of TD identification and monitoring using real software projects and (ii) VisminerTD makes it possible to analyze the evolution of TD items.

Project context

We selected as software objects two real projects: JUnit (*github.com*
unit-team
junit4) and Apache Ant (*github.com*
apache
ant). JUnit is an open-source framework that supports the creation of automated tests in Java. JUnit4 is available on GitHub and has 21 versions available, 138 collaborators, and 2225 commits. Apache Ant is a tool developed in



Total Duplication: 37.92%

Duplication Occurrences

Tokens Threshold: 100 Token Count: 196 Language: JAVA Occurrences: 2

Filename: src/tests/junit/org/apache/tools/ant/BuildFileRule.java
Begin Line: 211 **End Line:** 311 **Line Count:** 101 **Duplication (%):** 26.73

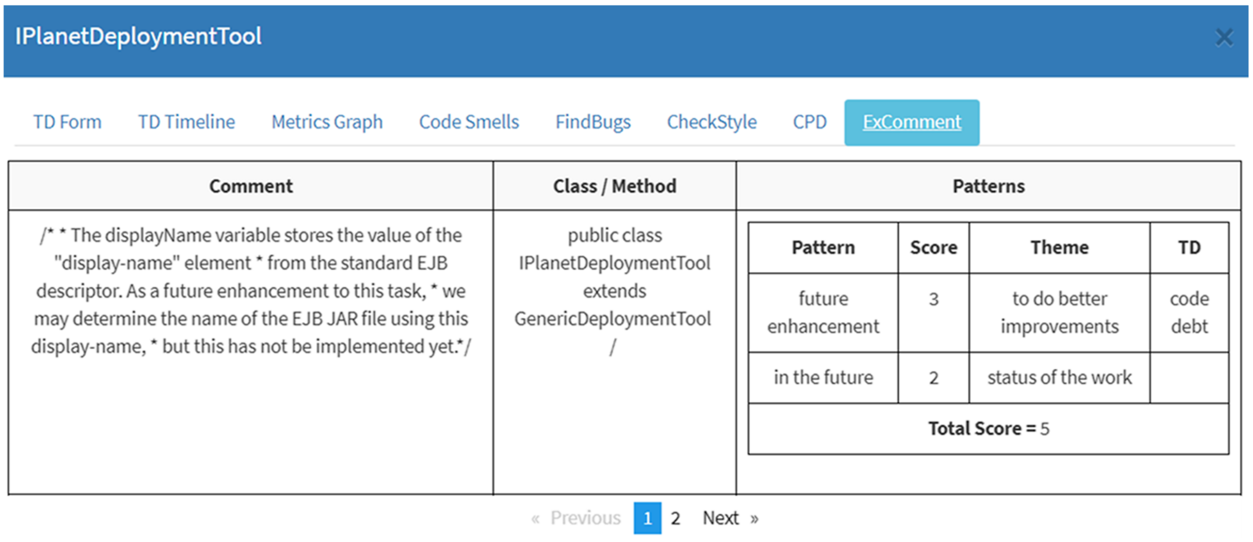
Filename: src/tests/junit/org/apache/tools/ant/BuildFileTest.java
Begin Line: 513 **End Line:** 616 **Line Count:** 104 **Duplication (%):** 15.56

Tokens Threshold: 100 Token Count: 136 Language: JAVA Occurrences: 2

Filename: src/tests/junit/org/apache/tools/ant/BuildFileRule.java
Begin Line: 126 **End Line:** 160 **Line Count:** 35 **Duplication (%):** 11.19

Filename: src/tests/junit/org/apache/tools/ant/BuildFileTest.java
Begin Line: 293 **End Line:** 328 **Line Count:** 36 **Duplication (%):** 5.79

Fig. 11 CPD page. Information about duplication occurrences detected by CPD



Comment	Class / Method	Patterns
/* * The displayName variable stores the value of the "display-name" element * from the standard EJB descriptor. As a future enhancement to this task, * we may determine the name of the EJB JAR file using this display-name, * but this has not be implemented yet.*/	public class IPlanetDeploymentTool extends GenericDeploymentTool /	see below

Pattern	Score	Theme	TD
future enhancement	3	to do better improvements	code debt
in the future	2	status of the work	
Total Score = 5			

Fig. 12 eXcomment page. Information from comments that indicate TD extracted by the eXcomment tool (Apache Ant)—Class: IPlanetDeploymentTool.java

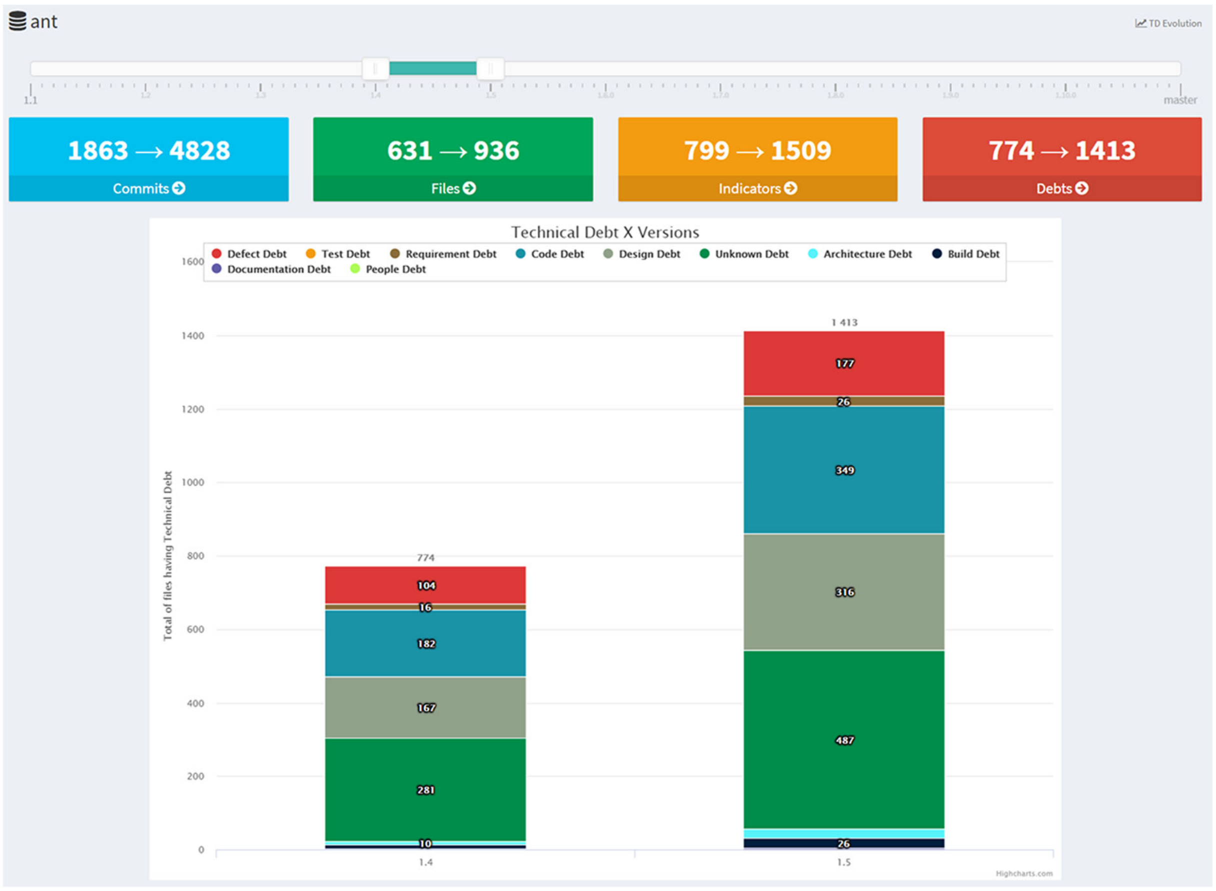
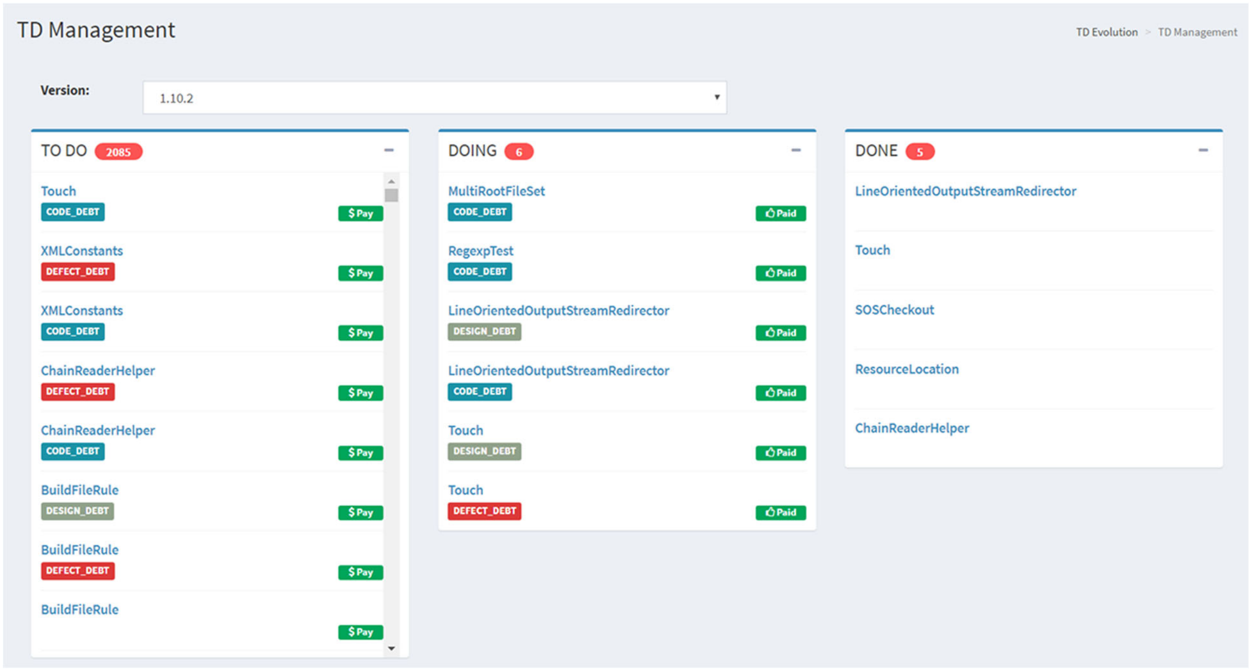


Fig. 13 TDEvolution perspective. View that allows the understanding of how software is evolving over time regarding the identified TD items


TD Management

TD Evolution > TD Management

Version: 1.10.2

TO DO 2085

- Touch CODE_DEBT $ Pay
- XMLConstants DEFECT_DEBT $ Pay
- XMLConstants CODE_DEBT $ Pay
- ChainReaderHelper DEFECT_DEBT $ Pay
- ChainReaderHelper CODE_DEBT $ Pay
- BuildFileRule DESIGN_DEBT $ Pay
- BuildFileRule DEFECT_DEBT $ Pay
- BuildFileRule $ Pay

DOING 6

- MultiRootFileSet CODE_DEBT Paid
- RegexpTest CODE_DEBT Paid
- LineOrientedOutputStreamRedirector DESIGN_DEBT Paid
- LineOrientedOutputStreamRedirector CODE_DEBT Paid
- Touch DESIGN_DEBT Paid
- Touch DEFECT_DEBT Paid

DONE 5

- LineOrientedOutputStreamRedirector
- Touch
- SOSCheckout
- ResourceLocation
- ChainReaderHelper

Fig. 14 TDManagement perspective. View that allows the monitoring in details of TD items in the project

Java, used for the automation of the compilation process. It is available on GitHub and has 115 versions available, 36 collaborators, and 14,215 commits. Both projects were selected because they are widely used in academic (for empirical studies) and industrial (to support some development activities) environments. Besides, JUnit has already been analyzed by Mendes et al. [57], so it would be possible to make comparisons of performance in the mining process. The Ant project is about six times bigger than JUnit. Both projects are mature, large, and with many contributors.

Design

The study procedure defines an activity of verifying the internal structure of JUnit and Ant projects aiming to identify possible parts of them that did not follow good development practices and, therefore, could affect their maintainability (possibly TD items). Thus, the study considers the following steps:

1 One of the participants installed and configured the RM on their laptops, performed the mining process for both projects (JUnit and Ant), and saved the results in a database available to all participants.
2 Participants installed and configured VisminerTD and accessed the information previously mined in step 1.
3 Participants selected the JUnit project repository and, then, the Ant project. After, they used the TDAnalyzer view to search for files with TD indicators in each project. Next, they selected the files that had the largest amount of TD indicators.
4 Participants used the TDEvolution view to analyze the evolution of TD occurrence throughout several project versions. They also analyzed specific cases where there was a significant increase in the number of debt items between two different versions.
5 Finally, participants performed simulations of the use of TDManagement to verify its operation. They also analyzed some timelines of TD items to observe if there were situations in which the number of debt indicators has increased or decreased over the time in a same file.

The study was performed by three researchers of the project (a Ph.D. student, a Master's student, and an Undergraduate student) with more than 8 years of experience in software development. In the following sections, we describe the activities performed in the use of VisminerTD and also some information about how the tool supports each of them.

Results

Initially, we analyzed JUnit and Apache Ant repositories using the RM. For both projects, we considered 19 object-oriented metrics, 7 code smells, duplicate code, ASA issues, and source code comments. In this study, style problems were not considered because of two major reasons. First, we could not detect style problems in Apache Ant due to parsing errors in many files. The CheckStyle tool simply aborts its analysis when a parser error occurs. And second, the large amount of style problems in JUnit because it has no well-defined code styling, some parts of its code (the newer ones) follows the Google Java Style (considered in the analysis) while older parts follows a different custom style.

The mining time of the JUnit project has improved considerably when compared to this same task reported in Mendes et al. [57]. The previous analysis took about 30 min; meanwhile with the new version of VisminerTD, it took only three and half minutes, showing an improvement of almost 1000% in performance for the data mining activity. For Apache Ant, the mining time was about 16 min. The used computer is the same used in [57] and has the following specification: Intel Core i5-2520M processor, 8 GB of RAM, and Linux Mint 18.3 Sylvia operating system.

The next step was to filter the collected information. Nine versions of JUnit and 12 versions of Apache Ant, including master, have been selected to have their information displayed in TDAnalyzer module. After, all items that could indicate the presence of TD were listed as cards, and it was possible to confirm or reject the indicated items (if they were a TD item or not). In VisminerTD, this activity is performed card by card, clicking on the item name. By doing this, we can have access to detailed information about the TD item, its indicators, and the different types of debt that have been identified in that file. After analyzing the information, the user can confirm that an item actually has that type of debt or discard that item indicating that it is a false positive. For the purpose of this study, all items were confirmed as TD items.

In JUnit, VisminerTD found 805 files containing some kind of debt and 1250 indications of presence of debt considering 9 versions of the software (the selected versions were: r3.8.2, r4.6, r4.7, r4.8, r4.9. r4.10, r4.11, r4.12 and master (commit 660a373 on February 9, 2018)), resulting in an average of 89 files and 139 TD indicators per version. When selecting the master version, for example, we noticed that the tool identified 133 files with some debt. By analyzing different information about metrics and software comments available in VisminerTD, participants identified that the artifact with more types of debt in the master version was "Assert", with 1036 lines of code and five different types of debt. For this specific TD item, the tool reported 23 different TD indicators. An occurrence of God Class was detected, indicating the presence of code and design debt according to Alves et al. [58]. In addition, 15 code comments were found: (i) seven of them indicating the presence of defect debt; (ii) five indicating the presence of code debt, (iii) one case indicating the presence of test debt; and (iv) nine of them pointing to the presence of debt in general (it was not possible to define a specific type).

In Apache Ant, VisminerTD found 6,496 files with some debt and 14,026 indications of presence of debt considering the 12 versions analyzed (the selected versions were: 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.10.2 and master (commit 52a0ec8 on February 19, 2018)). This resulted in an average of 541 files and 1,169 TD indicators per version. As an example, VisminerTD identified five types of debt (code, design, defect, requirements, and unknown) in the "Javadoc" file in the version 1.10.2. This file has 2259 lines of code and the following indicators of the presence of debt were detected: 7 Complex Methods, 3 Brain Methods, a God Class, 5 Slow Algorithms, 6 ASA issues and 41 comments with evidence of debt (code, design, defect, requirements and unknown).

The next step in using VisminerTD is to access the TDEvolution module (see Fig. 15), that shows how the presence of TD in the different versions of JUnit is. When analyzing the view provided by TDEvolution, it was verified that, from the first to the second version of JUnit, there was a significant increase in the number of files and TD indicators (see releases 3.8.2 and 4.6 in Fig. 15). This change occurred because there was a difference of about 5 years from version 3.8.2 (commit a0f0ee1 on December 28, 2004) to version 4.6 (commit b5e9885 on April 13, 2009). From the second version, the view presented an upward and gradual behavior, with a small decrease of the values in the master version. Concerning the comparison among versions, the number of commits increased 14 times and the number of classes increased more than four times, from the initial version to master, while the number of indicators and debts had a gradual increase. However, in version 4.12, there was a significant increase compared to version 4.11. The main differences are in defect debt that went from 22 to 33 items and in debt items in general (unknown type) that changed from 66 to 89 items.

In Apache Ant, versions from 1.1 to the most current were selected for analysis. Observing Fig. 16, we can see

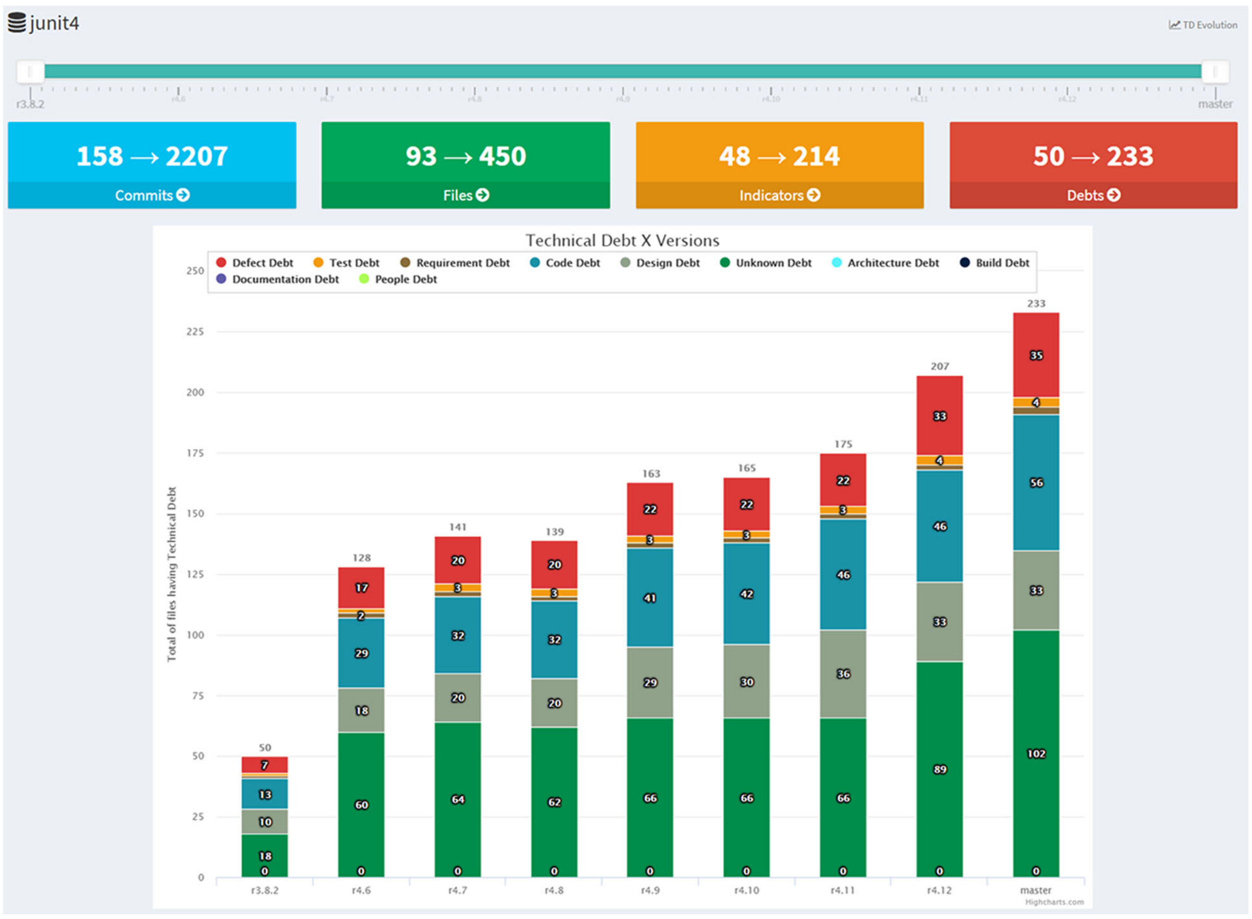


Fig. 15 Feasibility study—JUnit. JUnit project evolution analysis using TDEvolution view

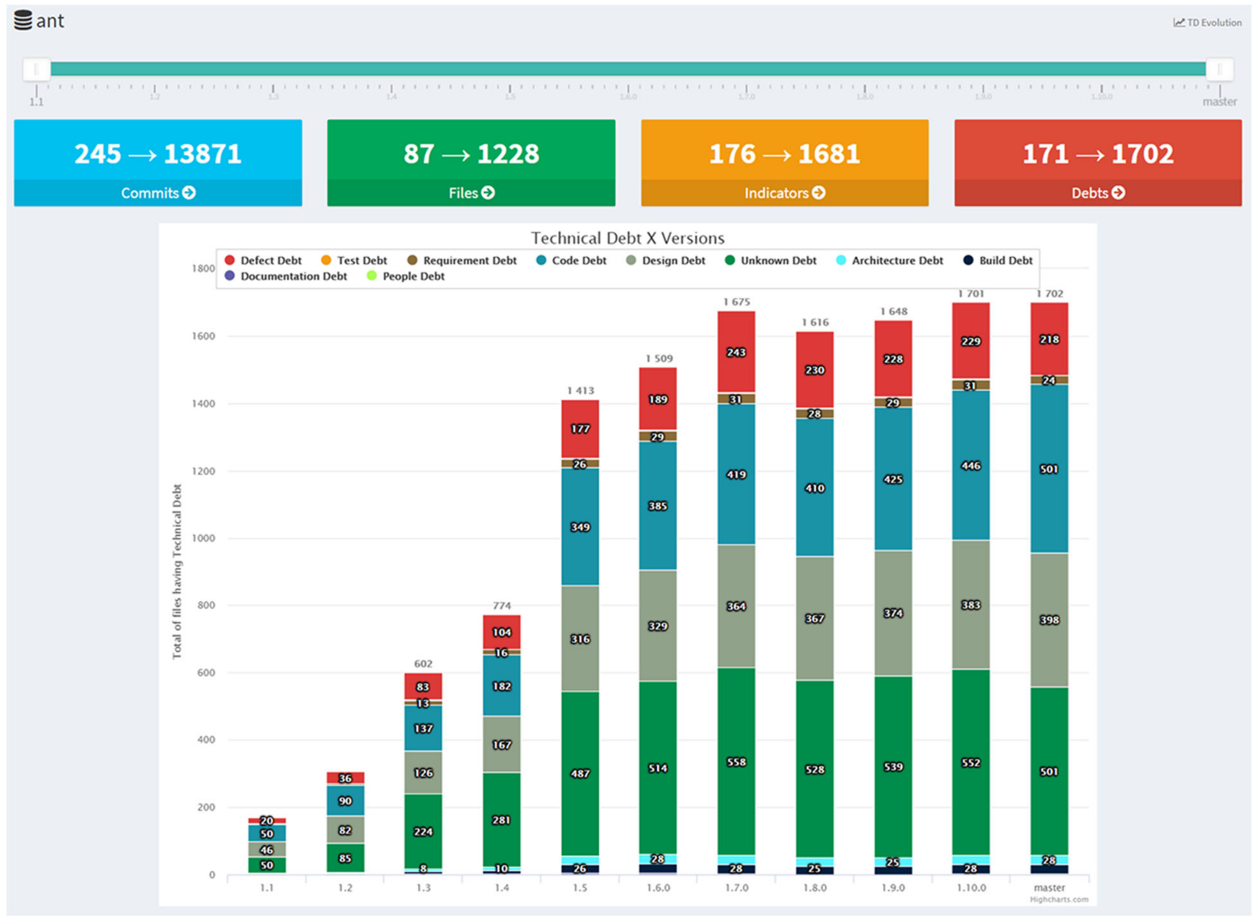


Fig. 16 Feasibility study—Apache Ant. Apache Ant project evolution analysis using TDEvolution component

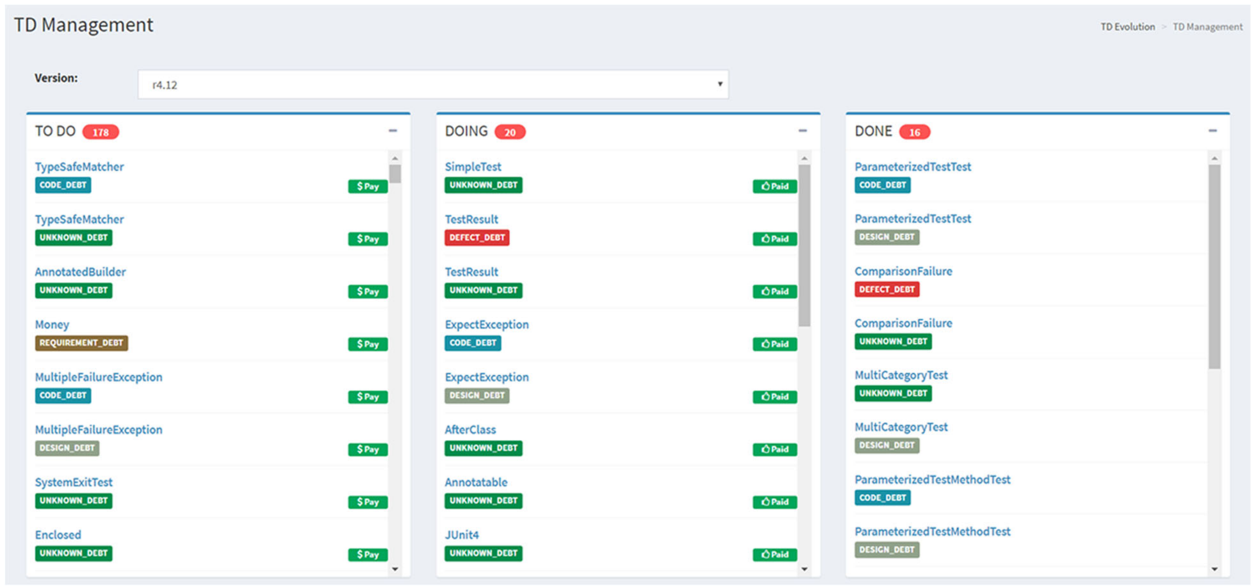
that the project size has increased constantly, and this behavior has been accompanied by an increase in the occurrence of different types of debt. Specifically, from version 1.4 to 1.5, the number of commits increased from 1863 to 4828, the number of files from 631 to 936, and we also detected the appearance of 639 debts and 710 TD indicators.

The last phase of the study was the analysis of the TDManagement module, which allows the user to monitor the current status (detected items, items under payment activities, or items that have been already paid off) of debts items in the project (see Fig. 17). In the study, 178 debts from version 4.12 of the JUnit project were mapped to the TO DO panel, 20 to the DOING panel, and 16 to the DONE panel. In the TO DO panel, all detected debt items that have not been selected for payment or have not been already paid off are listed. When the software engineer decides that a debt item should be paid, they can move the card to the DOING panel. This panel lists all the debt items that are currently being solved. Once this task is completed, the item is marked as paid off by moving the card to the DONE panel. During this process, the amount of items in each status can be checked through a counter located at the top of each panel.

In this module, it is also possible to obtain a detailed view of the current status of a TD item, as well as its evolution throughout the different versions of the software. Clicking on a particular item will show a detail-on-demand view as well as seen in the TD Analyzer, where the user can see a timeline presenting the history of that TD item. For example, in Fig. 18, we can see the TD Timeline view for the Javadoc file. It presented variations in the last three versions: new TD indicators appeared in version rel/1.10.0, and some of them were paid off in the master version.

Discussion

In this study, it was verified that VisminerTD allows software engineers to identify TD items by combining information from different TD indicators (software metrics, code smells, duplicated code, style problems, as well as information extracted from code comment analysis)



TD Management

TD Evolution > TD Management

Version: r4.12

TO DO 178	DOING 20	DONE 16
TypeSafeMatcher CODE_DEBT $ Pay	SimpleTest UNKNOWN_DEBT Paid	ParameterizedTestTest CODE_DEBT
TypeSafeMatcher UNKNOWN_DEBT $ Pay	TestResult DEFECT_DEBT Paid	ParameterizedTestTest DESIGN_DEBT
AnnotatedBuilder UNKNOWN_DEBT $ Pay	TestResult UNKNOWN_DEBT Paid	ComparisonFailure DEFECT_DEBT
Money REQUIREMENT_DEBT $ Pay	ExpectException CODE_DEBT Paid	ComparisonFailure UNKNOWN_DEBT
MultipleFailureException CODE_DEBT $ Pay	ExpectException DESIGN_DEBT Paid	MultiCategoryTest UNKNOWN_DEBT
MultipleFailureException DESIGN_DEBT $ Pay	AfterClass UNKNOWN_DEBT Paid	MultiCategoryTest DESIGN_DEBT
SystemExitTest UNKNOWN_DEBT $ Pay	Annotatable UNKNOWN_DEBT Paid	ParameterizedTestMethodTest CODE_DEBT
Enclosed UNKNOWN_DEBT $ Pay	JUnit4 UNKNOWN_DEBT Paid	ParameterizedTestMethodTest DESIGN_DEBT

Fig. 17 Feasibility study—TDManagement. Using TDManagement in the version r4.12 of Ant project

and monitor them through the use of different visual metaphors.

During the study execution, participants detected some instances of TD items that were pointed out only by code comments. This indicates that the eXcomment is relevant for identification and that such kind of information is relevant and can complement information that originated from software metrics to support development teams in tasks of TD identification.

Concerning the support for TD management activities, the tool allows to store the following information about each identified TD item: status, responsible for its insertion, if the debt is intentional or not, principal, interest amount, interest probability, and others. In addition, VisminerTD also has visual views that facilitate the monitoring of the evolution of debt items. It is possible, for example, to check the growth of the amount of TD items throughout the different versions of a software. More specifically, the tool has a slider bar that allows the comparison of the number of commits, source files, TD indicators and TD types, between two or more versions of the project.

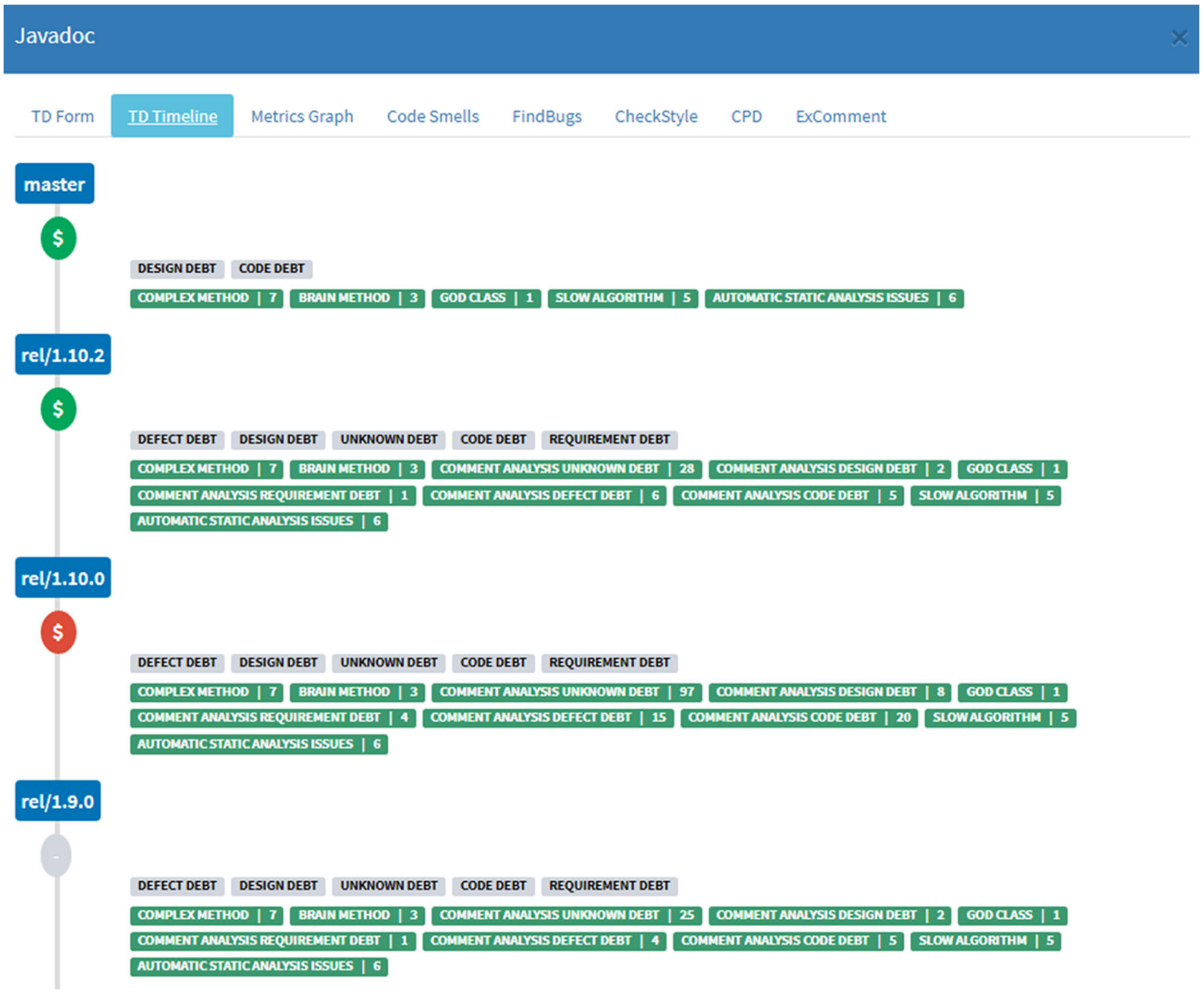
This work has implications for both practitioners and researchers. For practitioners, VisminerTD is available for use in VisminerTD-Client[3] and VisminerTD-Service[4]. Given the current scenario characterized by very limited options of tools to support activities of TD identification and monitoring, the tool presented in this paper approximates the state-of-the-art and the state-of-the-practice in the TD area, contributing to a wider dissemination of the concept. For researchers and practitioners, VisminerTD is an open-source project; thus, they are free to hold their version of the tool and, for example, implement the necessary adaptations for using in practice, implement and evaluate new visual metaphors to support TD-related activities. Finally, specifically for researchers, the use of VisminerTD indicated that the use of a combination of TD indicators (metrics and source code comments) is feasible and should be better investigated in the future to better understand questions related to the cost-benefit of this kind of analysis and the level of complementarity and overlapping among different indicators.

Study limitations

The study was performed by participants with different levels of experience. Two of them have professional experience in the industry. However, all participants have about 5 years of experience on the development of software solutions that involve concepts of software metrics and visualization. It is also important to highlight that the participants of the study are also the developers of the tool. So, they know exactly how to use the tool, and this may be an issue when considering other participants.

We did not consider style problems in the analyses. The parser problems encountered in the Apache Ant project could not be solved without change the CheckStyle source code since the tools was designed to abort its analysis when a parser problem occurs. Meanwhile, the styling inconsistencies in JUnit made the number of TD items and indicators increase by a lot. Thus, to avoid this difference between these two scenarios, we removed the CheckStyle from the analysis.

As another limitation, we considered two open-source software projects to perform the feasibility study. Despite



master

DESIGN DEBT | CODE DEBT

COMPLEX METHOD | 7 · BRAIN METHOD | 3 · GOD CLASS | 1 · SLOW ALGORITHM | 5 · AUTOMATIC STATIC ANALYSIS ISSUES | 6

rel/1.10.2

DEFECT DEBT | DESIGN DEBT | UNKNOWN DEBT | CODE DEBT | REQUIREMENT DEBT

COMPLEX METHOD | 7 · BRAIN METHOD | 3 · COMMENT ANALYSIS UNKNOWN DEBT | 28 · COMMENT ANALYSIS DESIGN DEBT | 2 · GOD CLASS | 1 · COMMENT ANALYSIS REQUIREMENT DEBT | 1 · COMMENT ANALYSIS DEFECT DEBT | 6 · COMMENT ANALYSIS CODE DEBT | 5 · SLOW ALGORITHM | 5 · AUTOMATIC STATIC ANALYSIS ISSUES | 6

rel/1.10.0

DEFECT DEBT | DESIGN DEBT | UNKNOWN DEBT | CODE DEBT | REQUIREMENT DEBT

COMPLEX METHOD | 7 · BRAIN METHOD | 3 · COMMENT ANALYSIS UNKNOWN DEBT | 97 · COMMENT ANALYSIS DESIGN DEBT | 8 · GOD CLASS | 1 · COMMENT ANALYSIS REQUIREMENT DEBT | 4 · COMMENT ANALYSIS DEFECT DEBT | 15 · COMMENT ANALYSIS CODE DEBT | 20 · SLOW ALGORITHM | 5 · AUTOMATIC STATIC ANALYSIS ISSUES | 6

rel/1.9.0

DEFECT DEBT | DESIGN DEBT | UNKNOWN DEBT | CODE DEBT | REQUIREMENT DEBT

COMPLEX METHOD | 7 · BRAIN METHOD | 3 · COMMENT ANALYSIS UNKNOWN DEBT | 25 · COMMENT ANALYSIS DESIGN DEBT | 2 · GOD CLASS | 1 · COMMENT ANALYSIS REQUIREMENT DEBT | 1 · COMMENT ANALYSIS DEFECT DEBT | 4 · COMMENT ANALYSIS CODE DEBT | 5 · SLOW ALGORITHM | 5 · AUTOMATIC STATIC ANALYSIS ISSUES | 6

Fig. 18 Feasibility study—Timeline view. Showing the Timeline view for Javadoc (Apache Ant)

the fact that they are commonly used in both academic and industrial settings, are from different application domains, and have different sizes, the obtained results cannot be generalized to all cases.

In order to overcome some of these limitations, we conducted a second feasibility study to obtain feedback on the use of the tool from the point of view of other subjects.

Feasibility study II

The evaluation was an academic case study applying our tool using real project data to simulate TD identification and monitoring meetings. The case study description is based on available guidelines [59].

Study objective

This study aims to analyze the proposed tool with the purpose of characterizing its viability of use with respect to the usefulness, ease of use, and self-predicted future use, from the point of view of students and professionals in the context of identification and monitoring of TD items.

For the study, the identification and monitoring meeting was performed with the support of VisminerTD. We intended to investigate if participants have a positive perception regarding the use of the tool. It is not the purpose of this study to analyze the accuracy and/or combination of the TD indicators. This analysis would require the participation of the team involved in the development of the project, which we intend to do in a next study.

As the objective of this study is related to the perception about the adoption of a new technology, we conducted the evaluation using the Technology Acceptance Model (TAM) [16], which has been extensively used [60]. TAM considers three constructs—perceived usefulness, ease of use, and self-predicted future use—which are measured by a set of questions. We adapted our questions from the ones used by Ali Babar et al. [61].

Project context

The study consists of analyzing the applicability of the proposed tool through the simulation of a TD identification and TD monitoring meeting. The objective is to evaluate, considering a list of candidate TD items from the JUnit project automatically identified by the RepositoryMiner, which items should be analyzed and monitored throughout the software development lifecycle, thereby minimizing the negative impact of debt accumulation on the project.

Procedure and instrumentation

To carry out the study, we created a virtual machine (VM) using the Ubuntu 18.10 operating system with LXDE desktop environment, and we allocated to this VM, 2 processors and 2 GB of RAM. The VM was executed in computers with Intel Core i5 4950 processors and 8 GB of RAM. We choose this approach, using a VM, because it made easier to create and redistribute the study environment to the participants, besides, to ensure an approximate experience between them.

In the VM, we have installed the VisminerTD and all other softwares needed to run it (e.g., MongoDB). We also included a database with the analysis of JUnit project, which contains information about the versions r3.8.2, r4.6, r4.7, r4.8, r4.9. r4.10, r4.11, r4.12, and master (commit 660a373 on February 9, 2018). The analysis performed did not consider the information from CheckStyle. This happened because the JUnit has no well-defined code styling and some parts of its code (the newer ones) follows the Google Java Style while older parts follows a different style. Thus, considering the large amount of data generated by the analysis (due the code styling inconsistencies) and the effort that would have been necessary to make the project more "analyzable", we have removed the CheckStyle from the analysis.

The case study was conducted with the following general steps:

1 The researchers trained the participants on TD concepts, software metrics, code smells, static program analysis, and source code comments analysis.
2 The researchers trained the participants about VisminerTD tool used to simulate the TD identification and monitoring meeting.
3 Using VisminerTD, the participants, in pairs, carried out five tasks of identification and monitoring of TD items.
4 Participants filled in the final evaluation form individually.

Before starting the study (step 1), each participant filled in a consent and a characterization form. After, we conducted a brief training on TD in order to familiarize the participants with the involved concepts. At this time, seeking to simulate TD identification and monitoring meetings, we organized the participants in pairs. The participants could discuss among them, but no communication was allowed outside the pairs.

During step 2, we gave a training to explain how to use the VisminerTD to identify and monitor TD items. The pairs used VisminerTD to perform five tasks (Table 1) that simulated different scenarios (step 3). Through these steps, we seek to show to the participants all the features available in the tool.

At the end of this activity, participants completed, individually, an evaluation form about their perceptions on the VisminerTD (step 4). In this form (Table 3), based on TAM, each participant analyzed statements related to the usefulness, ease of use, and self-predicted future use of the VisminerTD indicating the option that best represented their point of view, according to the following five point scale: (1) I totally agree; (2) I agree partially; (3) Neutral; (4) Partially disagree; and (5) Strongly disagree. At the end of the form, participants described their perceptions regarding the positive and negative points of the tool and suggestions for improvements.

Results

Characterization of participants

The participants are undergraduate students coursing a Software Engineering discipline. In total, 28 undergraduate students participated in the study. Seventy-three percent of them indicated have some experience with software development (18% have more than 2 years of experience). Not all participants had experience with software development; however, theoretical concepts were presented in the software engineering discipline.

Participants also indicated their level of experience in seven specific areas of the software development process according to the following scale: (1) none, (2) studied in class, (3) practiced in classroom projects, (4) used in personal projects, and (5) used in projects in the industry. The results are presented in Table 2. We can see that for all presented areas, there are participants with experience in the industry.

VisminerTD evaluation

The answers of the participants to the TAM questions are shown in Table 3. This table presents an overall scenario of acceptance regarding usefulness, ease of use, and self-predicted future use.

In all the statements analyzed with respect to usefulness, more than 87% of participants agreed with the affirmations. Moreover, the results indicate that, when using the VisminerTD, we can expect the following benefits: improved performance, productivity, and efficiency through ease to identify and monitor TD items

Table 1 List of tasks performed for TD identification and monitoring

#	Task
1	*Task 1 (TDAnalyzer)* 1. In version r4.12, identify the two debt items that contain "God Class" and "Duplicated Code" indicators, occurring in the same item. 2. For each item: (a) Inform the metric values used to detect "God Class". (b) Find the *total amount of duplication*. (c) On the *TDForm* tab, change the status of the debts found from "Not Analyzed" to "Doing" and click "Save".
2	*Task 2 (TDAnalyzer)* 1. Indicate the *number* of different types of TD present in the "Assert" item in the *master* version. 2. Using the *Timeline* tab, from "Assert" item, determine whether there has been any change in the number of indicators from version *4.12* to *master*. If so, name the indicators. 3. Go to *TDForm* tab, also from "Assert" item, change the *status* of the debts found from "Not Analyzed" to "Done", then click the "Save" button.
3	*Task 3 (TDAnalyzer)* 1. In the *master* version, select the debt type, "Defect Debt/Comment Analysis", and click on "Update Query". 2. Enter the *number of items found*, then click on the button "Confirm All from Filter".
4	*Task 4 (TDManagement)* 1. From the TD items identified in the TDAnalyzer, select the master version and simulate the payment of the debt for *ParametrizedTestTest* (*CODE_DEBT*) and *BaseTestRunner* (*DESING_DEBT*), until both are in the *DONE* panel; 2. Find the *total amount of items in all 3 panels* (TO DO, DOING, and DONE).
5	*Task 5 (TDEvolution)* 1. Use the slider to limit viewing the versions *r4.10* and *r4.11*. 2. Find the *difference* of the number of items having a *Code Debt* from version *r4.10* to version *r4.11*. 3. Investigate whether there has been any abrupt change in the amount of TD items between software versions, and in what versions this has happened. If so, why do you think this happened?

Table 2 Experience of participants

Knowledge area	Level of experience				
	1	2	3	4	5
Project management	7	13	3	1	4
Monitoring and correction of software defects	8	8	2	3	7
Software maintenance	8	9	1	3	7
Software architecture	8	13	1	3	3
Software design	6	11	4	3	4
Software documentation	11	9	3	4	1
Requirement specification	6	14	4	2	2
Implementation	1	7	8	4	8
Software testing	4	11	5	4	4

using visual techniques. In addition, the tool has a good coverage of different types of TD and provides a large amount of information in an automated way on the indicators.

In assessing the ease of use of the VisminerTD to "easy to learn" characteristic, 20 participants agreed and 4 participants disagreed indicated that they found easy to learn how to use the tool. When evaluating the interaction with the tool, 23 indicated that the interaction is clear and understandable. Regarding to become skillful and remember how to use the tool, 20 participants indicated that it would be easy to become skilled while 26 stated that it would be easy to remember how to identify and monitor TD items using VisminerTD.

Regarding the self-predicted future use, when asked about the preference in identifying and monitoring the TD using VisminerTD, the answers were well distributed

Table 3 TAM constructs, questions, and Likert Scale Frequency Count

		Strongly agree	Agree	Neutral	Disagree	Strongly disagree
#	Usefulness					
U1	Using the proposed tool in my job, I would be able to identify and monitor of TD items more quickly. (Quick)	21	7			
U2	Using the proposed tool, I would improve my performance in identifying and monitoring TD items. (Job Performance)	22	5	1		
U3	Using the proposed tool, I would increase my productivity. (Increase Productivity)	15	11	2	1	
U4	Using the proposed tool, I would improve my effectiveness in identifying and monitoring TD items. (Effectiveness)	21	6	1		
U5	Using the proposed tool would make identification and monitoring TD items easier. (Makes job easier)	22	5	1		
U6	I find the proposed tool useful to management of TD items. (Useful)	20	7	1		
#	Ease of Use					
E1	Learning to operate the proposed tool would be easy for me. (Easy to learn)	10	10	4	4	
E2	My interaction with the proposed tool would be clear and understandable. (Clear and understandable)	11	12	5		
E3	I would find it easy to use the proposed tool to do what I want it to do. (Controllable)	14	9	5		
E4	It would be easy to become skillful in using the proposed tool. (Skillful)	12	8	6	2	
E5	It would be easy to remember how to perform TD identification and monitoring using the proposed tool. (Remember)	18	8	2		
E6	I find the support easy to use. (Easy to use)	13	11	3	1	
#	Self-predicted future use					
S1	Assuming the proposed tool is available on my job, I predict that I will use it on a regular basis in the future.	16	10	2		
S2	I prefer using the proposed tool for conducting identification and monitoring of TD items than not using it.	12	6	8	1	1

across the options. This may suggest that although VisminerTD is well accepted, since 26 participants indicated that they would make use of the tool in the future, it is not yet ready for use.

Participants were also asked to indicate positive and negative impressions on the use of the tool. The positive points were (10 participants) "Ease of visualizing the results", "Good visually, intuitive and simple"; (6) "gain in performance and productivity", "Speed of identification of the debts"; (2) "Coverage Issues", "Quantity of information"; (2) "Automates the identification of TDs", "Automates the management of TDs"; (1) "Nice environment (Illustrative) and with dynamic update (in real time)"; (1) "Practicality, ease and good information filters". On the other hand, the participants also pointed out that "the tool have is a few complex" (2 participants) and "need the assistance of a researcher" (2 participants), "missing filters in TDManagement view" (1) and "the layout of some tool filters is confusing" (1).

Discussion

This study provided positive evidence on the use of the proposed tool, indicating that VisminerTD may be useful in supporting TD identification and TD monitoring activities (96% of participants indicated that they could use the tool to identify and monitor TD). The results also pointed out that more than 81% of the participants agreed that by using VisminerTD, they have gains in agility, productivity, performance, and efficacy when evaluating TD items. In the point of view of the participants, it is easy to learn and use VisminerTD, and the interaction with the tool is clear.

Although VisminerTD has shown itself to be an interesting and useful support for TD identification and TD monitoring, it is yet in its early stages of development and needs improvements to be ready for use. Participants suggested some improvements such as adding more filters on some views (e.g., TDManagement), changing the order of how TD items appear on the TD Analyzer view, the need to have support for other programming languages. This finding stimulated us to define some future directions on this research, as described in the "Final remarks" section of this paper.

Threats to validity

The study participants were chosen by convenience and are undergraduate students. However, some of them have experience in software development in industry. Thus, although the results are not generalizable, they provide an initial evidence on the investigated topic.

As discussed previously, we did not use the data from CheckStyle in the analysis used to carry out the study. This can be considered a threat to validity, since it was not possible to evaluate the view related to CheckStyle (CheckStyle page in the TDAnalyzer). However, we believe that the large amount of data, come from CheckStyle analysis, would compromise the participant's perception about the VisminerTD.

Finally, the TD items used for analysis were extracted from a real software project, but the number of items can be considered small. This is a threat assumed in this study. However, we believe that even if we had more TD items to evaluate, the participants' perception on the benefits of VisminerTD would tend to be positive as well.

Comparison to related works

The systematic mapping study performed by Alves et al. [10] indicated that the identification and monitoring of TD by means of software visualization techniques still demand tool support. The authors reported that only three primary studies about the use of visualization techniques in the TD identification activity were identified. In this section, we perform a comparative analysis between VisminerTD and the following related tools: SonarQube, DebtFlag, and Cast.

SonarQube [62] is one of the best-known tools for code quality inspection. SonarQube is an open-source software capable of extracting a wide variety of reports on a system, such as software metrics, code smells, TD, and dependencies between classes. This tool has a strategy of identifying TD different from that used in VisminerTD. First, it does not work with source code comment analysis. Second, VisminerTD allows the user to automatically perform analysis on several versions of the software, a functionality not present in SonarQube so far.

Although SonarQube has support for more programming languages, the VisminerTD can also be extended to support more programming languages through its generic Abstract Syntax Tree. We are currently working to support C/C++ in the future releases.

DebtFlag [63] is a tool that allows marking code to "link" debt items to the point of the project in which it is located in the code. The tool consists of two parts, an Eclipse Integrated Development Environment (IDE) plugin, called DebtFlag, and a web application. The DebtFlag plugin is responsible for capturing TD items through the Eclipse IDE, tracing its propagation and supporting its micromanagement, while the web application provides a dynamic presentation of the TD items detected by the DebtFlag plugin. Through this approach, the tool focuses on allowing the developer to manage TD items at the implementation level as well as at a higher level.

Although it does not provide the possibility of observing TD propagation, VisminerTD extracts a larger amount of information, as well as combining them. VisminerTD is built to be flexible, IDE-independent, extensible, and persistent-support. In this way, data can be analyzed using more robust mechanisms: this is especially useful in large data sets. In addition, the modular structure of the tool, due to the two modules RM and VisminerTD, also facilitates its integration with other tools for identifying and monitoring TD items.

CAST Application Intelligence Platform (AIP) [64] provides an approach for measuring TD on a regular basis. Through its use, development teams can identify structural flaws adding them to the TD of the application. This tool relies on some other CAST proprietary tools to perform its analysis. CAST AIP also has a strategy of identifying TD different from that used in VisminerTD. First, it does not work with source code comment analysis. Second, while VisminerTD can identify nine types of debt, CAST AIP identifies only two types (code and architecture debt).

Table 4 presents a comparison between VisminerTD and the other related tools. The main characteristics analyzed were (i) *License*; (ii) *Programming languages:* the supported programming languages; (iii) *New parsers:* allows the addition of new parsers; (iv) *Repository mining:* helps to mining local and remote repositories; (v) *Object-oriented metrics:* computes object-oriented software metrics; (vi) *Style problems:* detects coding style problems; (vii) *ASA issues:* detects ASA issues; (viii) *Code Smells:* detects code smells; (ix) *Custom TD thresholds:* allows to set TD identification thresholds; (x) *Multiple versions analysis:* allows the analysis of multiple versions automatically; (xi) *Source metrics analysis:* analyzes source code metrics for TD identification; (xii) *Code comments analysis:* analyzes source code comments for TD identification; (xiii) *Availability:* the tool is available for download; and (xiv) *Standalone:* the tool can run without an IDE.

As it can be observed in Table 1, VisminerTD is a tool with more functionalities among the analyzed characteristics, standing out in relation to the related works by (i) having a generic Abstract Syntax Tree allowing the creation of new parsers for mining source codes in other programming languages; (ii) accessing information from GIT and GITHUB; (iii) integrating with different tools that allow the combination of several types of metrics (e.g.,

Table 4 A comparison between VisminerTD and the other related tools

Characteristics	VisminerTD	SonarQube	DebtFlag	CAST AIP
License	Free	Free	Not Found	Commercial
Programming languages	Java	Java and 20 others	Java	Java and 50 others
New parsers	Yes	Yes	No	No
Repository mining	Yes	No	No	No
Object-oriented metrics	Yes	Yes	No	Yes
Style problems	Yes	Yes	No	Yes
ASA issues	Yes	Yes	No	Yes
Code Smells	Yes	Yes	No	No
Custom TD thresholds	Yes	Yes	Yes	Yes
Multiple versions analysis	Yes	No	No	No
Source metrics analysis	Yes	Yes	No	No
Code comments analysis	Yes	No	No	No
Availability	Yes	Yes	No	Yes
Standalone	Yes	Yes	No	Yes

code smells, bugs, issues, and source code comments); and (iv) allowing the identification of different types of TD indicators during software evolution.

Regarding the Table 1, we only considered a small amount of features. It is worthy mentioning that the tools SonarQube and Cast AIP have several other features that are not in the table; furthermore, they also analyze more programming languages. Although that the VisminerTD is just in its initial state, we intend to improve the VisminerTD adding support for more programming languages and more features in the future releases, especially features related to the combination of data from TD identification and software repositories, such as, an adviser for who must to pay a TD based in its commit history. We did not find functionalities like these in other tools available.

Final remarks

VisminerTD supports the identification and monitoring of debt items using software visualization techniques. Currently, the tool allows the analysis of 19 software metrics, detection of seven code smells, ASA issues, duplicate code occurrences, style problems in Java code, and the identification of nine types of debt (architecture, build, code, defect, documentation, design, requirement, people, and test debt). Moreover, it allows the joint analysis of data from metrics and code comments to support the identification of debt items.

The studies presented in this work provides initial evidence on the feasibility of using VisminerTD to analyze data from real software projects. The next steps of this research involve the execution of empirical studies in industrial environments to investigate how VisminerTD can be inserted in practical scenarios and what is the perception of software practitioners on its use.

Considering this research as a starting point, some perspectives of future work are as follows: (i) evolve VisminerTD based on the users' perception regarding its use; (ii) replicate the second study in other academic/industry scenarios; (iii) investigate when/how to use the proposed strategy in the context of a software development process and what impacts (e.g., in terms of effort) it brings to the development team; (iv) improve both RepositoryMiner and VisminerTD (e.g., adding more features and support for more programming languages), and (v) evaluate/evolve the accuracy of the RepositoryMiner automatic TD identification process considering industry projects and their development teams. We also intend to investigate how we could improve the proposed strategy to support organizational decisions regarding TD payment considering TD items prioritization and TD indicators agglomeration.

Endnotes

[1] https://visminer.github.io

[2] https://github.com/visminer/repositoryminer

[3] https://github.com/visminer/visminertd-client

[4] https://github.com/visminer/visminertd-service

Abbreviations

AIP: Application Intelligence Platform; AMW: Average Method Weight; API: Application Programming Interface; ATFD: Access To Foreign Data; CCB: Configuration Control Board; CPD: Copy/Paste Detector; CYCLO: McCabe's Cyclomatic Number; FDP: Foreign Data Provider; IDE: Integrated Development Environment; JAR: Java ARchive; JSON: JavaScript Object Notation; LAA: Locality Attribute Accesses; LOC: Lines of Code; LVAR: Number of Local Variables; MAXNESTING: Maximum Nesting Level; MLOC: Method Lines of

Code; NOA: Number of Attributes; NOAM: Number of Accessor Methods; NOAV: Number of Accessed Variables; NOM: Number of Methods; NOPA: Number Of Public Attributes; NProtM: Number of Protected Members; PAR: Number of Parameters; REST: Representational State Transfer; RM: RepositoryMiner; TCC: Tight Class Cohesion; TD: Technical debt; WMC: Weighted Method Count; WOC: Weight Of a Class

Acknowledgements
This work was partially supported by CNPq Universal 2014 grant 458261/2014-9 and SECTI-UFBA Cooperation Agreement 2012-1.

Funding
This work was partially supported by the National Council for Scientific and Technological Development (CNPq), grant 840003/2017-0.

Availability of data and materials
The authors declare that all data and materials are available at VisminerTD Site. The "Instructions" section in site describes how to replicate the performed study and execute the RepositoryMiner, VisminerTD-Service, and VisminerTD-Client tools.

VisminerTD Client

- Project name: VisminerTD Client
- Project home page: https://visminer.github.io/
- Archived version: https://doi.org/10.5281/zenodo.1195755
- Operating system: Platform independent
- Programming language: TypeScript
- Other requirements: Node.js 8.9 or higher and Angular 5
- License: Apache License 2.0
- Any restrictions to use by non-academics: license needed

VisminerTD Service

- Project name: VisminerTD Service
- Project home page: https://visminer.github.io/
- Archived version: https://doi.org/10.5281/zenodo.1195789
- Operating system: Platform independent
- Programming language: JavaScript
- Other requirements: Node.js 8.9 or higher and MongoDB 3.2 or higher
- License: Apache License 2.0
- Any restrictions to use by non-academics: license needed

RepositoryMiner

- Project name: RepositoryMiner
- Project home page: https://visminer.github.io/
- Archived version: https://doi.org/10.5281/zenodo.1195815
- Operating system: Platform independent
- Programming language: Java
- Other requirements: Java 7 or higher and MongoDB 3.2 or higher
- License: Apache License 2.0
- Any restrictions to use by non-academics: license needed

Authors' contributions
This paper aims to present the work generated from the doctorate degree of TSM (author). This paper results from a collaborative work. MGMN was the advisor and ROS co-advisor of TSM in his doctorate degree. MGMN and ROS contributions are mainly based on the academic orientation, as a partner for making decisions during the doctorate degree and writing the article. FGG, DPG, and RLN helped with the development of the tool and writing the article. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details
[1]Federal University of Bahia, UFBA, Av Adhemar de Barros, s/n, Instituto de Matemática, 40170-110 Salvador, Brazil. [2]Salvador University, Av Tancredo Neves, 2131, Caminho das Árvores, Salvador, Brazil. [3]Federal Institute of Bahia, R. Emídio dos Santos, s/n, Sala A303 - Barbalho, 40301-015 Salvador, Brazil. [4]Federal Institute of Bahia-Santo Amaro, Tv. São José, s/n - Bomfim, 44200-000, Santo Amaro, Brazil. [5]Fraunhofer Project Center at UFBA, Av. Luiz Viana Filho - Loteamento Colinas do Jaguaribe, Lote M, Salvador, Brazil.

Received: 20 March 2018 Accepted: 20 November 2018
Published online: 16 January 2019

References

1. Lientz BP, Swanson EB, Tompkins GE (1978) Characteristics of application software maintenance. Commun ACM 21(6):466–471
2. Lehman MM, Belady LA (1985) Program Evolution : Processes of Software Change. Lehman MM, Belady LA (eds). Academic Press London, Orlando
3. Parnas DL (1994) Software aging. In: Proceedings of the 16th International Conference on Software Engineering. IEEE Computer Society Press, London. pp 279–287
4. Seaman C, Guo Y (2011) Chapter 2 - measuring and monitoring technical debt. Adv Comput 82:25–46. Elsevier. https://doi.org/10.1016/B978-0-12-385512-1.00002-5. http://www.sciencedirect.com/science/article/pii/B9780123855121000025
5. Izurieta C, Vetró A, Zazworka N, Cai Y, Seaman C, Shull F (2012) Organizing the technical debt landscape. In: Proceedings of the Third International Workshop on Managing Technical Debt. MTD '12. IEEE Press, Piscataway. pp 23–26. http://dl.acm.org/citation.cfm?id=2666036.2666040
6. Kruchten P, Nord RL, Ozkaya I (2012) Technical debt: From metaphor to theory and practice. IEEE Softw 29(6):18–21. https://doi.org/10.1109/MS.2012.167
7. Avgeriou P, Kruchten P, Ozkaya I, Seaman C (2016) Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). Dagstuhl Rep 6(4):110–138. https://doi.org/10.4230/DagRep.6.4.110
8. Cunningham W (1992) The wycash portfolio management system. SIGPLAN OOPS Mess 4(2):29–30. https://doi.org/10.1145/157710.157715
9. Fowler M Technical Debt Quadrant. http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html. Accessed 23 Feb 2018
10. Alves NSR, Mendes TS, de Mendonça MG, Spínola RO, Shull F, Seaman C (2016) Identification and management of technical debt: A systematic mapping study. Inf Softw Technol 70:100–121. https://doi.org/10.1016/j.infsof.2015.10.008
11. Spínola RO, Vetró A, Zazworka N, Seaman C, Shull F (2013) Investigating technical debt folklore: Shedding some light on technical debt opinion. In: Managing Technical Debt (MTD), 2013 4th International Workshop On. pp 1–7. https://doi.org/10.1109/MTD.2013.6608671
12. Zazworka N, Spínola RO, Vetró A, Shull F, Seaman C (2013) A case study on effectively identifying technical debt. In: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering. EASE '13. ACM, New York. pp 42–47. https://doi.org/10.1145/2460999.2461005
13. Li Z, Avgeriou P, Liang P (2015) A systematic mapping study on technical debt and its management. J Syst Softw 101:193–220. https://doi.org/10.1016/j.jss.2014.12.027
14. Novais R, Simões P, Mendonça M (2012) Timeline matrix: an on demand view for software evolution analysis. In: II Brazilian Workshop on Software Visualization. Natal, Brazil. pp 9–16. http://reuse.cos.ufrj.br/wbvs2012/papers/wbvs02.pdf
15. Magnavita R, Novais R, Silva B, Mendonça M Using evowave for logical coupling analysis of a long-lived software system. Workshop de Visualizaçáo de Software, Evoluçáo e Manutençáo / VII Congresso Brasileiro de Software, Maringá, Brazil. pp 1–8. https://vem2016.ufba.br/artigos/Session3_VEM_2016_paper_9.pdf
16. Davis FD (1989) Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS Q 13(3):319–340. https://doi.org/10.2307/249008
17. Brown N, Cai Y, Guo Y, Kazman R, Kim M, Kruchten P, Lim E, MacCormack A, Nord R, Ozkaya I, Sangwan R, Seaman C, Sullivan K, Zazworka N (2010) Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER '10. ACM, New York. pp 47–52. https://doi.org/10.1145/1882362.1882373

18. Morgenthaler JD, Gridnev M, Sauciuc R, Bhansali S (2012) Searching for build debt: Experiences managing technical debt at google. In: Proceedings of the Third International Workshop on Managing Technical Debt, MTD '12. IEEE Press, Piscataway. pp 1–6. http://dl.acm.org/citation.cfm?id=2666036.2666037
19. Bohnet J, Döllner J (2011) Monitoring code quality and development activity by software maps. In: Proceedings of the 2nd Workshop on Managing Technical Debt, MTD '11. ACM, New York. pp 9–16. https://doi.org/10.1145/1985362.1985365
20. Snipes W, Robinson B, Guo Y, Seaman C (2012) Defining the decision factors for managing defects: A technical debt perspective. In: Proceedings of the Third International Workshop on Managing Technical Debt, MTD '12. IEEE Press, Piscataway. pp 54–60. http://dl.acm.org/citation.cfm?id=2666036.2666046
21. Guo Y, Seaman C (2011) A portfolio approach to technical debt management. In: Proceedings of the 2Nd Workshop on Managing Technical Debt, MTD '11. ACM, New York. pp 31–34. https://doi.org/10.1145/1985362.1985370
22. Seaman C, Spínola R (2013) Managing technical debt. In: (Short Course) XVII Brazilian Symposium on Software Quality, SBC, Salvador, Brazil
23. Wohlin C, Runeson P, Host M, Ohlsson MC, Regnell B, Wesslen A (2000) Experimentation in software engineering: an introduction. Kluwer Academic Publishers, Norwell. https://doi.org/10.1007/978-3-642-29044-2
24. Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493
25. Lanza M, Marinescu R (eds) (2006) Object-Oriented Metrics in Practice. Springer, New York. https://doi.org/10.1007/3-540-39538-5
26. Fowler M (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston
27. Fontana FA, Ferme V, Spinelli S (2012) Investigating the impact of code smells debt on quality code evaluation. In: Proceedings of the Third International Workshop on Managing Technical Debt, MTD '12. IEEE Press, Piscataway. pp 15–22. http://dl.acm.org/citation.cfm?id=2666036.2666039
28. FindBugs: Find Bugs in Java Programs. http://findbugs.sourceforge.net/. Accessed 23 Feb 2018
29. CheckStyle: CheckStyle. http://checkstyle.sourceforge.net. Accessed 23 Feb 2018
30. Corazza A, Maggio V, Scanniello G (2015) On the coherence between comments and implementations in source code. In: 2015 41st Euromicro Conference on Software Engineering and Advanced Applications. pp 76–83. https://doi.org/10.1109/SEAA.2015.20
31. Farias MADF, Neto MGDM, Silva ABD, Spínola RO (2015) A contextualized vocabulary model for identifying technical debt on code comments. In: IEEE 7th International Workshop on Managing Technical Debt (MTD). Bremen, Germany. pp 25–32. http://doi.ieeecomputersociety.org/10.1109/MTD.2015.7332621
32. Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference On. pp 91–100. https://doi.org/10.1109/ICSME.2014.31
33. Farias MAF, Santos JA, da Silva AB, Kalinowski M, Mendonça M, Spínola RO (2016) Investigating the Use of a Contextualized Vocabulary in the Identification of Technical Debt: A Controlled Experiment. In: ICEIS. pp 369–378. https://doi.org/10.5220/0005914503690378
34. Maldonado E, Shihab E, Tsantalis N (2017) Using natural language processing to automatically detect self-admitted technical debt. IEEE Trans Softw Eng PP(99):1–1
35. Koschke R (2003) Software visualization in software maintenance, reverse engineering, and re-engineering: A research survey. J Softw Maint 15(2):87–109. https://doi.org/10.1002/smr.270
36. Roman GC, Cox KC (1992) Program visualization: The art of mapping programs to pictures. In: Proceedings of the 14th International Conference on Software Engineering, ICSE '92. ACM, New York. pp 412–420. https://doi.org/10.1145/143062.143157
37. Novais RL, de Mendonça Neto MG (2014) Software evolution visualization: Status, challenges, and research directions. In: Ghani I, Kadir WMNW, Ahmad MN (eds). Handbook of Research on Emerging Advancements and Technologies in Software Engineering, chap 26. IGI Global, Hershey. pp 597–611
38. Chen C (2004) Information Visualization - Beyond the Horizon. Springer, New York
39. Diehl S (2007) Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software. Springer, Secaucus
40. Storey M-AD, Cubranić D, German DM (2005) On the use of visualization to support awareness of human activities in software development: A survey and a framework. In: Proceedings of the 2005 ACM Symposium on Software Visualization, SoftVis '05. ACM, New York. pp 193–202. https://doi.org/10.1145/1056018.1056045
41. Novais RL, Torres A, Mendes TS, Mendonça M, Zazworka N (2013) Software evolution visualization: A systematic mapping study. Inf Softw Technol 55(11):1860–1883. https://doi.org/10.1016/j.infsof.2013.05.008
42. Keim DA (2002) Information visualization and visual data mining. IEEE Trans Vis Comput Graph 8(1):1–8. https://doi.org/10.1109/2945.981847
43. Ferreira de Oliveira MC, Levkowitz H (2003) From visual data exploration to visual data mining: A survey. IEEE Trans Vis Comput Graph 9(3):378–394. https://doi.org/10.1109/TVCG.2003.1207445
44. Carneiro G, Gomes de Mendonça Neto M (2013) Sourceminer - a multi-perspective software visualization environment. In: ICEIS 2013 - Proceedings of the 15th International Conference on Enterprise Information Systems, vol. 2. SciTePres, Angers, France. pp 25–36
45. Novais R, Santos JA, Mendonça M (2017) Experimentally assessing the combination of multiple visualization strategies for software evolution analysis. J Syst Softw 128:56–71. https://doi.org/10.1016/j.jss.2017.03.006
46. Gomes F, Mendes T, Carvalho L, Spínola R, Novais R, Mendonça M (2017) Repositoryminer – uma ferramenta extensível de mineração de repositórios de software para identificação automática de dívida técnica. Congresso Brasileiro de Software: Teoria e Prática (CBSoft) - Salão de Ferramentas, Fortaleza-Brazil
47. MongoDB: For giant ideas. https://www.mongodb.com. Accessed 23 Feb 2018
48. PMD: An extensible cross-language static code analyzer. https://pmd.github.io. Accessed 23 Feb 2018
49. Angular: One framework for mobile and desktop. https://angular.io/. Accessed 23 Feb 2018
50. Bootstrap: The most popular HTML, CSS, and JS library in the world. http://getbootstrap.com. Accessed 23 Feb 2018
51. HighCharts: Highcharts makes it easy for developers to set up interactive charts in their web pages. https://www.highcharts.com/. Accessed 23 Feb 2018
52. NodeJS: NodeJS. https://nodejs.org/en/. Accessed 23 Feb 2018
53. Express: Fast, unopinionated, minimalist web framework for Node.js. https://expressjs.com. Accessed 23 Feb 2018
54. Mongoose: Elegant mongodb object modeling for node.js. https://mongoosejs.com. Accessed 23 Feb 2018
55. Yuepu G, Spínola R, Seaman C (2014) Exploring the costs of technical debt management: a case study. Empir Softw Eng:1–24. https://doi.org/10.1007/s10664-014-9351-7
56. Silva D, Santos F, Neto P (2012) Os benefícios do uso de Kanban na gerência de projetos de manutenção de software. VIII Simpásio Brasileiro de Sistemas de Informação (SBSI), Trilhas Tácnicas, São Paulo, Brazil. pp 337–347
57. Mendes TS, Gonçalves DP, Gomes F, Spínola R, Novais R, Mendonça M (2017) VisminerTD: Uma Ferramenta para Identificação Automática e Monitoramento Interativo de Dívida Técnica. In: V Workshop de Visualização de Software, Evolução e Manutenção / VIII Congresso Brasileiro de Software, Fortaleza. http://vem2017.ufu.br/artigos/Mendes_et_al_2017.pdf
58. Alves NSR, Ribeiro LF, Caires V, Mendes TS, Spínola RO (2014) Towards an ontology of terms on technical debt. In: Proceedings of the 2014 Sixth International Workshop on Managing Technical Debt. IEEE Computer Society, Washington. pp 1–7. https://doi.org/10.1109/MTD.2014.9
59. Runeson P, Host M, Rainer A, Regnell B (2012) Case Study Research in Software Engineering: Guidelines and Examples. 1st ed. Workshop de Visualização de Software, Evolução e Manutenção / VII Congresso Brasileiro de Software, Maringá, Brazil. ISBN: 978-1-118-18100-3
60. Turner M, Kitchenham B, Brereton P, Charters S, Budgen D (2010) Does the technology acceptance model predict actual use? a systematic literature review. Inf Softw Technol 52(5):463–479. https://doi.org/10.1016/j.infsof.2009.11.005
61. Babar MA, Winkler D, Biffl S (2007) Evaluating the usefulness and ease of use of a groupware tool for the software architecture evaluation process. In: First International Symposium on Empirical Software Engineering and

Measurement (ESEM 2007). pp 430–439. https://doi.org/10.1109/ESEM.2007.48

62. SonarQube: The leading product for continuous code quality. https://www.sonarqube.org/. Accessed 23 Feb 2018
63. Holvitie J, Leppanen V (2013) Debtflag: Technical debt management with a development environment integrated tool. In: 4th International Workshop on Managing Technical Debt (MTD). pp 20–27. https://doi.org/10.1109/MTD.2013.6608674
64. Software C Software Intelligence for Digital Leaders. https://www.castsoftware.com/. Accessed 23 Feb 2018