Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2015, Article ID 898514, 12 pages
http://dx.doi.org/10.1155/2015/898514

Research Article

Supporting Technical Debt Cataloging with TD-Tracker Tool

Lucas Borante Foganholi, Rogério Eduardo Garcia, Danilo Medeiros Eler, Ronaldo Celso Messias Correia, and Celso Olivete Junior

Faculty of Science and Technology, São Paulo State University (UNESP), Roberto Simonsen Street, No. 305, 19060-900 Presidente Prudente, SP, Brazil

Correspondence should be addressed to Lucas Borante Foganholi; borante@gmail.com

Received 1 June 2015; Revised 8 August 2015; Accepted 27 August 2015

Academic Editor: Andrea De Lucia

Technical debt (TD) is an emergent area that has stimulated academic concern. Managers must have information about debt in order to balance time-to-market advantages and issues of TD. In addition, managers must have information about TD to plan payments. Development tasks such as designing, coding, and testing generate different sorts of TD, each one with specific information. Moreover, literature review pointed out a gap in identifying and accurately cataloging technical debt. It is possible to find tools that can identify technical debt, but there is not a described solution that supports cataloging all types of debt. This paper presents an approach to create an integrated catalog of technical debts from different software development tasks. The approach allows tabulating and managing TD properties in order to support managers in the decision process. It also allows managers to track TD. The approach is implemented by TD-Tracker tool, which can integrate different TD identification tools and import identified debts. We present integrations between TD-Tracker and two external tools, used to identify potential technical debts. As part of the approach, we describe how to map the relationship between TD-Tracker and the external tools. We also show how to manage external information within TD-Tracker.

1. Introduction

During software development it is usual to prioritize some activities, leaving others in the background, such as code optimization, use of best practices, defect correction, and documentation. The main reason for this is to reach the expectations imposed due to time or financial constraints. Therefore, postponing a technical activity creates technical debt. The term "technical debt" is used as a metaphor to refer to the likely long-term costs associated with software development and maintenance shortcuts taken by project stakeholders to deliver short-term business benefits [1–3].

Seaman and Guo [4] explain a tradeoff when dealing with technical debt. They argue that software managers need to balance between incurring TD and the associated costs when planning their projects. Only with TD information, it is possible to plan and decide which TD is going to be fixed. However, the lack of this information is a problem. Letouzey and Ilkiewicz [5] described some of these manager actions:

(i) Setting targets for debt and specifying what level and which debt types are acceptable for the project or organization.

(ii) Analyzing and understanding debt to estimate the potential impact and provide rationale for decisions.

(iii) Using TD as input for governance of application assets and analyzing an application's debt in correlation with other information such as business value or user perceived quality.

(iv) Institutionalizing previous practices and putting in place tools and processes to produce benefits of proactive TD management.

There are techniques and tools that could potentially be useful in the identification of TD, even if many of them were not developed for this purpose [6]. Techniques, such as code smells, design patterns, test results analysis, and test plans verification, can be classified accordingly with the software artifact related or TD type. Tools, such as *FindBugs*, *Sonar*,

and code test coverage tools, fully or partially automate techniques. Automatic approaches are capable of accommodating the change rates that major development projects introduce, but their reliance on statically predefinable models makes them incapable of modeling the entire requirement space [6]. Manual approaches capture the entire space [7], but due to their nature they consume a large amount of development resources which prohibits their frequent use.

A problem of technical debt context consists of catalog TD items originated from any software activity or TD type and using contextualized information when the debt is detected. It is possible to find tools that can identify technical debt, but no solution described in the literature review was found to support cataloging all types of debt.

Thus, the goal of this paper is to present a tool named TD-Tracker. The tool aims to create a catalog integrating TD from design/code, test, documentation, defect, and infrastructure activities in a properly designed structure linking stakeholders' observations about technical debt to related artifacts in the software activity. It is also intended to integrate with identification tools to import technical debt, aggregating automatic and manual approach benefits. TD-Tracker can concentrate information about technical debt of any type and from different project activities, allowing managers to track the entire TD life cycle.

The methodology used in this paper consists in describing an integrated catalog within a semiautomatic approach. The proposed approach is implemented by TD-Tracker and is intended to resolve technical debt by supporting its management on two levels. Stakeholders conducted micromanagement through TD issues under their responsibility, at implementation level and project level management by making TD-Tracker cater for TD information needed in the decision process. In this paper two studies are presented in order to show external integrations with *Sonar* and *GitHub*. The goal of both is to catalog information of technical debts provided by TD identification accomplished in external tools. Each tool identifies specific TD types.

This paper is organized as follows: in Section 2 the background used in the work is presented, in Section 3 the proposed approach is detailed, in Section 4 TD-Tracker tool which implements the proposed approach is described, in Section 5 an application of TD-Tracker with two external tools is demonstrated, in Section 6 the discussion about the proposed approach is presented, and in Section 7 the conclusion and future works are presented.

2. Background

In this section the theoretical background of technical debt identification and cataloging is presented.

2.1. Identification of Technical Debt. Guo and Seaman [8] proposed a classification of technical debt into four main types: design (or code), testing, defect, and documentation debt. In each type of TD it is possible to find techniques and tools to identify the debt.

Design or code debt can be identified by statically analyzing source code or inspecting code compliance to standards [4]. Izurieta et al. [6] mentioned techniques and tools that could potentially be useful in the identification activity. They presented four specific identification techniques: code smells, automatic static analysis, modularity violations, and design patterns, described in the following.

Code smells (a.k.a. bad smells), a concept introduced by Fowler et al. [9], describe choices in object-oriented systems that do not use principles of good-object oriented design. Another concept of code smells refers to potential violations of good object-oriented design principles [10]. Automatic approaches have been developed to identify code smells, as proposed by Marinescu [11]. Moha et al. [12] designed a framework for automatic detection of code smells, including a component for formally describing a code smell using a domain specific language. van Emden and Moonen [13] described jCOSMO tool, a code smell browser to detect and visualize smells in Java source code, focusing on two code smells (instanceof and typecast). Simon et al. [14] define a distance-based cohesion metric and a 3D visualization for detecting bad smells that violate high cohesion and low coupling principles. Schumacher et al. [15] focused on evaluating the automatic approaches regarding their precision and recall. Palomba et al. [16] described an approach on how code elements change overtime, exploiting change history information to detect instances of five different code smells through HIST. Former studies have shown that some code smells are correlated with defect- and change-proneness [17]. Another example of tool for code smells technique is CodeVizard [17].

Automatic static analysis (ASA) is a reverse engineering technique that consists of extracting information about a program from its source code using automatic tools [18]. ASA tools search issues based on violations on recommended programming practices and potential defects that might cause faults or might degrade some dimensions of software quality such as maintainability and efficiency, among other things. Some ASA issues can indicate candidates to TD. In this case, one in charge of cataloging can postpone the task of fixing it. In previous work Vetro et al. [19] analyzed the issues detected by *FindBugs* in two pools of similar small programs, *Sonar* [20].

There are other tools used in ASA studies related to code review such as *CheckStyle*, *Gerrit*, and *PMD*. *Checkstyle* is a static code analysis tool used in software development for checking if Java source code complies with coding rules [21]. A *Checkstyle* plug-in can provide overload syntax coloring or decorations in code editor, decorate the project explorer to highlight problem-posing resources, and add warnings and error outputs to the outputs. *Gerrit* is a free web-based software code review tool integrated with git and serves as a barrier between developers' private repositories and the official. Developers make local changes and then submit these changes for review. Reviewers make comments via the Gerrit web interface. For a change to be merged into the Android source tree, it must be verified and approved by a senior developer. *PMD* [22] is a source code analyzer that finds common programming flaws like unused variables,

empty catch blocks, and unnecessary object creation. The tool supports Java, JavaScript, PLSQL, Apache Velocity, XML, and XSL languages.

Izurieta et al. [6] described that, during software evolution, if two components are always changed together to accommodate modifications but they belong to two separate modules designed to evolve independently, then there is nonconformity. They may be caused by side effects of a quick and dirty implementation, or requirements might have changed so that the originally designed architecture could not be easily adapted. When such discrepancies exist, the software can deviate from its designed modular structure, which is called a modularity violation and it can indicate TD. Wong et al. [23] have demonstrated the feasibility and utility of this approach. In their experiment using *Hadoop*, they identified 231 modularity violations from 490 modification requests and 152 (65%) violations were conservatively confirmed. For modularity violations *CLIO* is another example of a suitable tool [23].

Design patterns [24] are popular because they claim, among others, to facilitate maintainability and flexibility of designs, reduce the number of defects and faults, and improve architectural designs. Software designs decay as systems and operational environments evolve and this can involve design patterns. Classes that participate in design pattern realizations accumulate grime nonpattern related code. Grime represents a form of TD, since the effort to keep the patterns cleanly instantiated has been deferred. In prior studies, Izurieta and Bieman [25] introduced the notion of design pattern grime and performed a study on three open source systems, *JRefactory*, *ArgoUML*, and *eXist*.

The four described techniques comprehend one type of TD: source code or design. There are other types of TD such as testing, defect, and documentation, which will be contextualized in the following.

Testing debts are tests that were planned but not implemented/executed or got lost. They are also test cases not updated for new/changed functionality or low code coverage [4]. They can be identified by different techniques such as comparing test plans to their results, checking planning tests and executed activities, and test code coverage [4]. The tools mentioned are not planned to identify TD but can be used to do that. Yang et al. [26] survey coverage-based tools and compared 17 tools based on three features: code coverage measurement, coverage criteria, and automation, which includes reporting. Shah et al. [27] present the consequences of exploratory testing of TD through systematic review. Exploratory testing is an approach that does not rely on formal test case definition; instead of designing test cases, the execution and evaluation of the software behavior are based on tester intuition and knowledge and can be used as source of TD [27].

Debts of documentation type originate from documentation that are not kept up to date. It can include API and software requirements and use case documentation. They can be identified by comparing change reports to documentation version histories. If modifications are done without accompanying changes to documentation, the corresponding not updated documentation is a TD. Forward and Lethbridge [28] identified, through a survey, tools used to deal with documentation in software projects, including automated generation of documentation. Traceability-based reading [29] is a scenario-based reading technique designed to uncover inconsistencies between multiple views on an object-oriented software system. It describes how to perform correctness and consistency checks among various UML models. Due to this specific focus on a limited set of important defects, the technique is to be used in combination with other reading techniques for complete coverage of a system description.

Defect debts are known defects that are not yet fixed [4], such as low priority or severity defects due to rarely manifested or presented workarounds. They can be identified by comparing test results to change reports; the defects found and not fixed are defect debt items [4]. The tools used to find source code debts and the tools that support test executions are capable of identifying defects. Snipes et al. [30] detailed a technique based on change control boards (CCB) to categorize and prioritize defects supporting manager decisions to fix/defer debts based on cost-benefit analysis.

2.2. Cataloging Technical Debt. The structure for cataloging technical debt in the proposed approach is based on the documentation structure introduced as part of the Technical Debt Management Framework (TDMF) [4, 31] by Seaman et al. This structure is extended in order to decompose entries into reusable components as well as to properly present technical debt in different software development activities.

The TDMF is a three-part approach for managing technical debt in software projects. It relies on a TD List (TDL) constructed in the first, technical debt identification, part. The list, which is similar to a task backlog, is populated with technical debt items, which correspond to single atomic occurrences of technical debt in the project.

A TD Item documents and upholds a set of information [4]. Each item represents a task left undone that incurs a risk of causing future problems if not completed. The original catalog contains the following:

(i) A description that explains the TD.

(ii) The type of TD, related to the software artifact.

(iii) Location of which part of the system the debt item is related to.

(iv) The reason for its acquisition or why that task needs to be done.

(v) An estimate of the TD principal, which indicates how much resources are required to pay it back to make this partition fully adhere to the design.

(vi) Interest probability, which estimates the probability of the issue occurrence.

(vii) Interest amount, resources required for extra work if the TD causes problems later.

Initially, when an item is created, the principal, expected interest amount, interest standard deviation, and correlations with other debt items can be estimated subjectively according to the maintainer's experience [4]. Since it is uncertain

whether extra effort will be required, they used expected interest amount and interest standard deviation to capture the uncertainty. The principal and interest of a technical debt item are estimated through measurement estimation of various development-related entities, principally effort. This rough estimation can be adjusted later using historical data or through other types of program analysis.

As exposed by Guo and Seaman [8], different types of technical debt may require different forms of measure. For example, the number of known but not fixed defects is a measurement of defect debt. The difference between expected code coverage of the test suites and their actual coverage can be used to measure testing debt. They also mentioned that it is necessary to use compatible units of measurement for all types of technical debt in order to easily monitor and compare with each other. For that, they used principal and interest to measure all types of technical debt. In this paper we use three-point scale (High/Medium/Low) as the unit of the metrics.

After explaining the measurement step, technical debt is tracked, is quantified, and is thus ready to be used for the decision making process. At this point it is necessary to contextualize decision making, but it is not detailed since the work is limited to TD identification and cataloging process, also to describe a tool for tracking TD. Guo et al. [31] described one scenario in which the technical debt list is used to facilitate decision making in release planning. For this planning, the project manager sorts TD according to their cost and benefit and determines the set that should be paid to minimize the cost and maximize the benefits of the project. Various approaches [8, 32, 33] have been proposed for technical debt prioritization and decision making.

3. The Proposed Approach

In this section the proposed approach of identification and cataloging technical debts is described. The approach has three activity groups: *Identification*, *Cataloging*, and *Managing*, as depicted in Figure 1.

The *identification* group initiates the proposed approach and copes with technical debt identification activities for the types design (or code), testing, documentation, and defect. These types summarize all defined types proposed by [31]. Each type of technical debt represents a different activity since the process or tool used in identification of this activity can be different. There is no input artifact or previous process in this group. The output of this activity is a list containing candidates to TD. When the list is created they are classified as candidates because they need to be confirmed by a manager or someone else performing the TD collector role. Some collector actions can include decision to fix TD before catalog or a proper review of the identified item, which can lead to an exclusion from the catalog. As indicated in the previous section, there are techniques and tools for identification of each kind of technical debt mentioned.

Technical debt *cataloging* is the second group in the proposed approach and starts after *identification*. This group contains two distinct activities: manual cataloging and semi-automated cataloging. They can be used in different situations

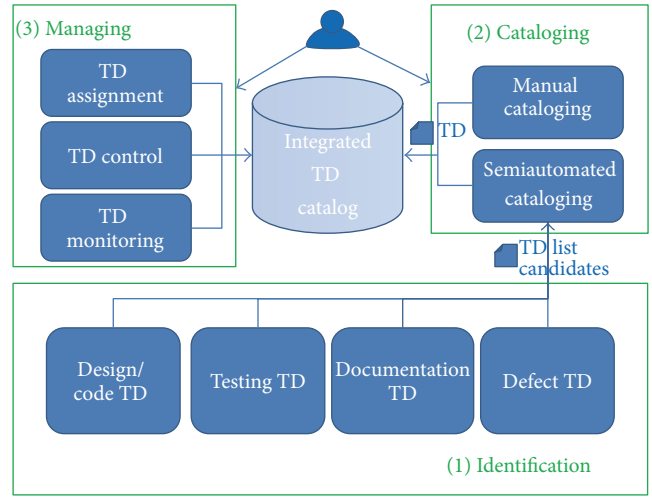

Figure 1: Proposed approach.

and applied accordingly to the use of techniques and tools mentioned in the previous group activity. Both activities must be performed by a collector and are detailed in the following. Collector is a role performed by a user and consists of collecting (in a manual or semiautomated way) the TD candidates list and defining which will be cataloged as TD. The output of *cataloging* is a register of technical debt.

Manual cataloging is the activity used to catalog any technical debt manually, regardless of debt type, origin, or way of identification. Manual approach can analyze wide areas in software development activities, besides the large amount of resources (time or people) required. Zazworka et al. [34] show that different stakeholders are aware of different debts in their project, indicating that TD elicitation should consider an inclusion of project team members. Zazworka et al. also indicates that aggregation is an effective approach to combine the identification results of these different team members.

Semiautomated cataloging uses the candidates of TD list generated in *identification* group as input and catalogs the item from the TD list. The collector analyzes the TD candidates list and, for each TD item, rejects or registers TD item, fulfilling the required information. This information can be provided by the integration if the catalog attributes (structure) are previously mapped. In this case, the only action performed by the collector is to check and approve the candidates as TD items.

For both *cataloging* activities the integrated catalog is used and its structure supports any type of technical debt. The structure is based on the item structure described in Subsection 2.2 proposed by Seaman and Guo [4]. It is extended in the proposed approach in order to catalog all types of technical debt and properly present technical debt at the next group activity. It also makes it possible to track, control, and classify pending or finished items, TD sort and prioritization. Item structure is described in Table 1.

The last group in the proposed approach is *Managing*. It comprehends TD assignment (or reassignment), TD control,

Table 1: Technical debt template, adapted from [4].

ID	TD identification number
Date	Date of TD identification
Responsible	Person or role who should fix this TD item
Type	Design, documentation, defect, testing, or other type of debt
Project	Name of project or software application
Location	List of files/classes/methods or documents/pages involved
Description	Describes the anomaly and possible impacts on future maintenance
Estimated principal	How much work is required to pay off this TD item on a three-point scale: High/Medium/Low
Estimated interest amount	How much extra work will need to be performed in the future if this TD item is not paid off now on a three-point scale: High/Medium/Low
Estimated interest probability	How likely is it that this item, if not paid off, will cause extra work to be necessary in the future on a three-point scale: High/Medium/Low
Intentional	Yes/No/Don't Know
Fixed by	Person or role who really fix this TD item
Fixed date	Date of TD conclusion
Realized principal	How much work was required to pay off this TD item on a three-point scale: High/Medium/Low
Realized interest amount	How much extra work was needed to be performed if this TD item was not paid off at moment of detection, on a three-point scale: High/Medium/Low

and TD monitoring activities. The activities use the TD catalog list as input and support manager decisions. TD assignment activity is used to perform attribution of TD to a responsible. TD control activity is used to manage TD, modifying properties and deleting or changing TD status. TD monitoring activity is used to track TD's life cycle and due dates and, when some changes are needed, can use previous activity to perform any modification.

4. TD-Tracker Tool

TD-Tracker is a tool that implements the approach (protocol) described in this paper. TD-Tracker uses the integrated catalog as metadata in order to register technical debt properties. It was developed using Java language and public community maintained frameworks to improve development productivity and reuse.

The tool architecture uses MVC (Model View Controller) design pattern. For persistence (or model) *Hibernate* framework is used, which enables the writing of applications whose data outlives the application process and applies to any relational databases (via JDBC). For interface (or view) layer the framework used is *Vaadin* because of its popularity and automation on browser-server communication.

TD-Tracker (download of Beta version and installation guide is available on http://www2.fct.unesp.br/grupos/lapesa/tdr) is a web-based application and uses an internal *Tomcat* as web container, which allows the execution from a web application archive (war) file as a Java archive (jar). As a web-based software all advantages are inherited from this kind of application and its detailed benefits are discussed in [35]. Since TD-Tracker uses *Vaadin 7*, the web browsers supported by this version of the framework are also supported by the tool, but the tool was intensively tested with *Google Chrome* browser and its use is strongly recommended.

The database structure of TD-Tracker uses the proposed integrated catalog (or TD catalog) as an entity and its attributes as columns. It also has an entity for users (a.k.a responsible), one entity to register the information of connection with external tools and another for mapping the relationship between TD-Tracker catalog and external application properties. These connection and relationship maps are the key to configuring an integration, allowing TD-Tracker semiautomated process to retrieve information from TD identification tools. As explained before, different tools are used for different kinds of technical debt and an integrated catalog is only possible when you have the external TD identification tools connected with TD-Tracker. In the current version TD-Tracker supports four of the most popular database engines [36], *MySQL*, *Oracle*, *Postgres*, *and SQL Server*, and integrates with external database through direct connection.

The layout of the application contains a left side steady menu used to navigate among the programs (or functionalities). Each program represents one or more activities of the proposed approach. The menu also contains the logged user information such as *Name*, *Email*, *Edit Profile* option, and application logout. TD-Tracker has access and permission control; to manipulate any content it is necessary to have a login and log into the application. The user has two types of profile, *User* and *Administrator*, and only administrators can add new users.

The architecture of the solution is depicted in Figure 2 and contains *Integration*, *TD Cataloging*, *Authentication*, and *User* components.

Integration corresponds to external integration process and allows configuring, testing, and saving the information

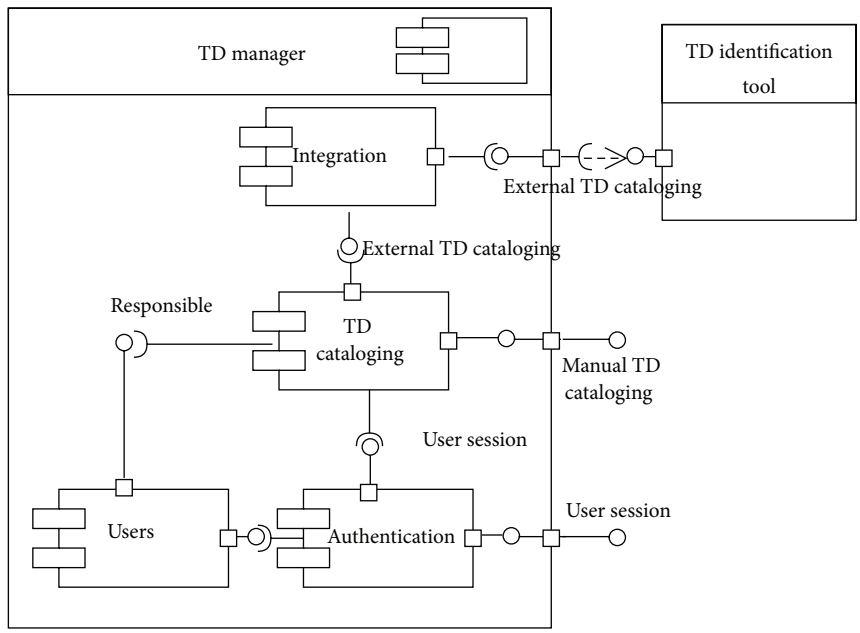

Figure 2: TD-Tracker architecture.

needed to connect. In current release, TD-Tracker has two kinds of integration: database direct connection and using *Github* API. The first is intrusive and made through database mapping and connection, which involves setting all required information for TD-Tracker map and connecting directly to an external TD identification tool databasee. The second integration uses an API to connect with *GitHub*, reported to be the largest repository host in the world, in order to extract all issues from any public or private project.

For database connection, the first step is to set the credentials of the external database to connect. To accomplish that it is necessary to fulfill integration properties which comprehend *Alias*, *Server*, *Port*, *Database Name*, *Database Type*, *User*, and *Password*. After informing all the required fields, it is possible to test the connection. This configuration is hard since it demands knowledge of identification of TD tool database credentials.

After configuring the connection it is indispensable to map a relationship between TD catalog properties and identification tool fields. For that, the user needs to provide an external table name and fields, associating with the existing properties of TD-Tracker catalog, establishing a relationship between the tools. At this point, the user can check results of the integration map under construction and visualize which information is correct or needs changes.

For *GitHub* connection, the first step is to set the credentials to connect with a specific project on *GitHub*. The fields used to create this integration are *Alias*, *User*, *Password*, and *Project*. After informing all the required fields, it is also possible to test the connection. Note that for this configuration it is necessary to have one user created on *GitHub* and the name of the project to access its issues. No other relationship is needed to map *GitHub* connection and the user can check results of the integration map for a project retrieving foreign

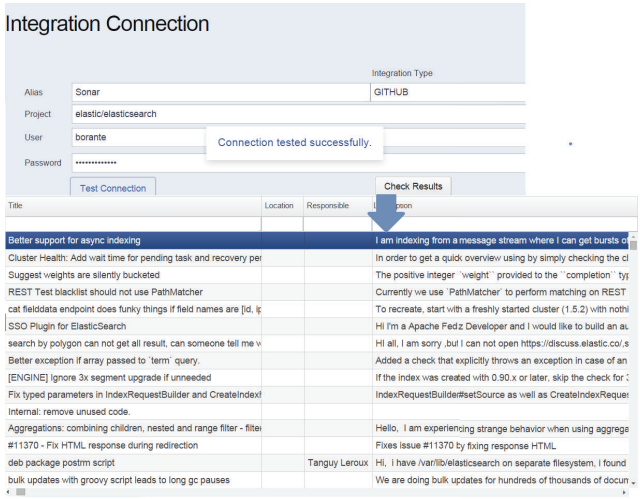

Figure 3: Integration between TD-Tracker and *GitHub* for *ElasticSearch* project.

TD candidates (or *GitHub* issues). This form of connection and checking results action is presented in Figure 3.

Integration component, as presented in Figure 2, has two lines that create a relation between *TD Cataloging* and an external TD identification tool. It is also possible to see that the relation with the external tool through External TD Candidate and the connection with *TD Cataloging* is through external cataloged TD.

TD Cataloging component is where the register of technical debt resides and can be inputed manually or retrieved after integration. It has a synchronization activity made through a foreign database connection from a TD identification tool,

detailed in the following. The component also contains manager activities such as TD monitoring, control, and assignment.

For manual cataloging, the user has to fulfill TD attributes presented in the catalog described in the proposed approach. The collector can input manually a TD from an external candidates list or after detecting it in an isolated case; it is only possible after logging into application. For this kind of cataloging, TD-Tracker validates required attributes, not allowing them to be saved if they are not detailed.

For semiautomatic interface it is necessary to create a database integration mapping of the relationship between TD-Tracker and the external tool. The interaction between *TD Cataloging* and *Integration* component exists in this kind of integration. To make it work it is necessary to configure the connection information and create the relationship between the fields of the integrated catalog and external database table, as explained in the *Integration* component. Configuration is a prerequisite for synchronization, an action performed by a controller or manager. Synchronization is composed of two steps: database or API connection and external information extraction.

After synchronization, external information is presented in a grid in TD-Tracker interface where it is possible to filter and sort the external TD identified by the integrated tool and import to TD-Tracker. A filter and sorting mechanism is used to select on TD candidates list the debts that should be cataloged, granting TD control. All fields mapped with TD catalog in the configuration stage can be used as filters, allowing managers or controller to limit the number of candidates to analyze and apply an external strategy within displayed data. The grid is composed of the fields mapped in the configuration stage and information presented in the grid, for some fields such as responsible, TD type, and the estimations, can be overridden.

An attribute to register external information ID is also created in the semiautomatic interface, avoiding duplication when cataloging TD and alerting when the external key is marked for integration twice. For the proper use of this activity it is vital to configure the external field ID when mapping integration properties.

The assignment activity is only possible after including the responsible person as an application user. In manual cataloging a list of existing users can be used to find a specific person when assigning a TD for a person. TD control activity can be performed while TD is not finished or deleted; it remains available to the manager to deal with TD relevance, estimated effort to fix the debt, and TD interest and probability to cause extra work in the future. It can also control the date when TD was fixed and the efforts realized to fix it. The dates and realized efforts can also be performed by the user responsible for the technical debt.

TD monitoring activity can support the decision process when it is possible to filter and find pending technical debts, which can affect the plan of the next project interaction or delivery. TD-Tracker can help the manager to prioritize TD based on estimated information of each TD. The tool does not implement any prioritizing algorithm but the properties, estimated principal, interest amount, and interest probability, can be combined and used in decision techniques by managers. TD-Tracker also keeps historical TD information, which can be used to estimate TD in similar projects or software activities and a proper estimation can help managers' decisions.

Authentication component corresponds to application authentication and permissions to access the functionalities. It integrates with *TD Cataloging* component through user session, which is required to perform any kind of cataloging. *Authentication* also integrates with *User* component to validate the credentials and create a user session. *User* component is related to profile and used as a responsible or fixed by person when associating with a TD in *TD Cataloging*, as presented by the line in Figure 3.

The implementation of integrated catalog and an overview of the tool are presented in next section.

5. TD-Tracker Application

Two different scenarios covered by TD-Tracker are presented in this section. In the first scenario, the integration with *Sonar* through database mapping and extraction is described. In the second scenario *GitHub* integration through a public API is described. The reason for presenting two scenarios is to show that integration is feasible for design and defect TDs.

5.1. Integration with Sonar and OpenRefine Project. In this integration the use of TD-Tracker to catalog TD from open source project named OpenRefine [37] is presented. This tool is developed for working with messy data, which includes cleaning, transforming from one format into another, extending with web services, and linking to databases. The project is being developed using Java and contains 36.883 LOCs (Lines of Code) distributed in 496 classes, a wiki as documentation [37] and some issues cataloged on *Github* repository.

Sonar is a world wide adopted (ASA) tool developed to analyze source code. The tool is based on rules to identify source code defects or refactoring points. It contains a total of 516 rules when integrated with *FindBugs*, in the version 3.7.4, and for Java projects. These rules are classified into standard severities such as info, minor, major, critical, and blocker, which can be customized. The rules are also grouped as bad practices, correctness and performance issues, design flaws, code issues, security, and so forth. In this application *Sonar* was locally installed and configured to use *Postgres* database as its issues repository.

After configuring [37] project using *Eclipse*, a very common IDE for Java projects, and compiling all the applications, the project was added into *Sonar* to perform the automatic syntax analysis. The 6596 issues identified by *Sonar* for this project represent TD candidates and were revised before performing the integration with TD-Tracker. Minor issues represent more than 41% and major issues more than 56%, comprehending 98% of the total amount identified. Minor issues usually are not fixed because of their relevance and major issues because of the high effort spent in fixing and the low probability of causing extra work.

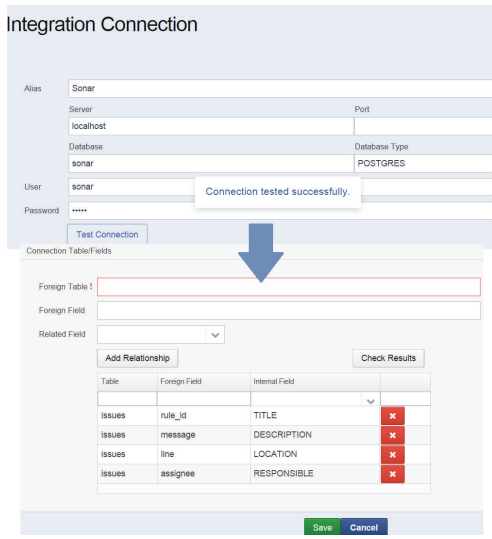

Figure 4: Mapped integration between TD-Tracker and *Sonar*.

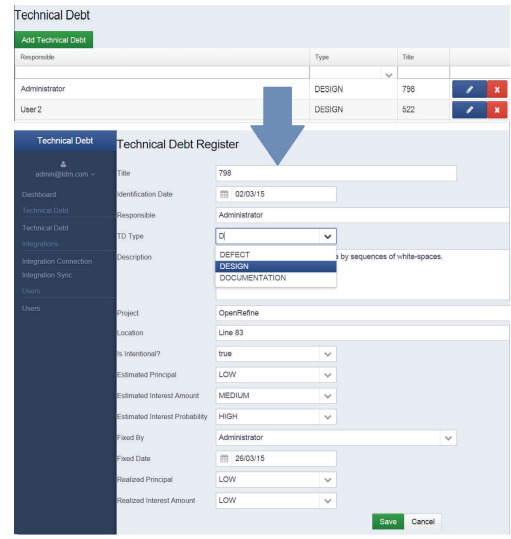

Figure 5: Implemented integrated catalog.

Table 2: Mapped relationship between TD-Tracker and *Sonar*.

TD-Tracker fields	Sonar fields
Date	issue_creation_date
Responsible	assignee
Project	root_component_id
Location	component_id
Description	message

The first step in TD-Tracker, for integrating a TD catalog, was to create a connection between TD-Tracker and *Sonar*. The integration connection attributes explained in Section 4 were defined as *Postgres* database, *localhost* server, and *Sonar* credentials to connect to the database, as presented in Figure 4. In Figure 4 it is also possible to see how TD-Tracker creates the relationship, explained in the following.

After successfully testing the connection, it is crucial to map the external table and fields to import TD candidates list. In this step, it is fundamental to know the characteristic of the *Sonar* database, which means knowing the table that contains issue information and which columns of that table are related to the proposed catalog properties. By investigating *Postgres* database, the "Issues" table on *Sonar* schema was found and it contains all identified information for *OpenRefine* project. Using table "Issues" it is possible to create the relationship between the two applications with fields as shown in Table 2.

The previous steps establish the integration, using these parameters TD-Tracker can extract the information through Integration Sync interface, and the manager can filter or sort this information to register technical debt into the integrated catalog. As explained in the previous section, only the marked external information is retrieved from the identification tool and that is the reason for creating such filters and sort options. In the *OpenRefine* project issues registered into *Sonar* only the critical severity was filtered and marked, resulting in 30 TD candidates. After reading and concluding that only 2 of the 30 issues were TD, both were cataloged, 1 assigned to user Administrator and 1 to User2.

With the information inserted into the catalog, managers are allowed to take control of it through the *TD Monitoring* interface, which is the same for manual registers of TD. In Figure 5 an example of an integrated technical debt of design type assigned to the responsible named *Administrator* is presented. It is possible to see two different screens; the first (superior corner) is a list of remaining TD only displaying the 2 cataloged TDs; in this screen manager can filter by responsible, TD Type, and title. In the second screen (inferior corner) the detailed TD information is presented, explained in the following.

The cataloged debt occurred on project *OpenRefine* in line 83 of class *com.google.refine.browsing.facets.ListFacet* and was defined as design TD to the responsible *Administrator* on 02/03/15. The TD description was *Incorrect lazy initialization of static field com.google.refine.ProjectManager.singleton in com.google.refine.RefineServlet.destroy( )* and the estimation was defined as follows: low estimated principal, which implies low difficulty of fixing; medium interest amount, which means medium difficulty if it is decided to postpone implementation to fix it; high interest probability, which stands for a lot of extra work when necessary to fix it in the future.

In this example, it was also possible to show information with regard to fixing TD such as the date when the implementation finishes, the author who fixed the TD, and the real effort to fix: realized principal and interest amount. The first one remained low, same as in estimation, and the realized interest amount was changed from medium to low difficulty. It means that real difficulty was easier than expected initially. This kind of information is useful to the person who performed the collector role, since it can be used as historical information and provide a better base of estimation for the next time. It is important to point out that if the estimation of interest amount was defined as low, it probably changes

Table 3: Cataloged technical debts from *ElasticSearch* issues in *GitHub*.

Title	Description
Reindex from _source by document ID or query	Be able to ask the system to reindex from the saved JSON by document ID or query. This is useful once we have ES style plugins for manipulating documents that might later change and therefore cause you to want to reindex some set of documents. #490 and #491 would let you query by a set of documents indexed before the required change. If you are going to store the JSON, you can take advantage of that by reindex requests. This might also allow the system to handle schema changes in the future more automatically by reindexing to the new analyzer over time in batch.
Changes API #1242	There should be an integration point for ES and external application where the external applications should be notified of any document changes or updates that happen in ES. CouchDB have a good implementation on it and it would be great if ES can also incorporate something similar or same. CouchDB change notification feature http://guide.couchdb.org/draft/notifications.html
Terms facet gives wrong count with n_shards > 1 #1305	With only one shard the following query gives the correct counts no matter what the size parameter is set to. However, with more than one shard the size parameter affects the accuracy of the counts. If it is equal to or greater than the number of terms returned by the facet query (5 in this case) then it works fine. However, the terms at the bottom of the list start to display low counts as you reduce the size parameter.

the order in project manager list and could not be included as TD to be fixed.

5.2. Integration with GitHub and ElasticSearch Project. In this second scenario the use of TD-Tracker is presented to catalog TD from open source project named *ElasticSearch* [38]. The tool is an API for real-time search and analytics capabilities. It supports multilingual search, geolocation, contextual *did-you-mean* suggestions, autocomplete, and result snippets. The project has different implementations in multiple languages, but the chosen project is being developed using Java. It has 124 releases, 12,163 commits done by 436 contributors, complete documentation references (the documentation, accessed on 2015-05-27, can be found at https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html), and some issues cataloged on *Github* repository.

Github is a web-based *Git* repository hosting service, which offers all of the distributed revision control and source code management functionality. *GitHub* provides a web-based graphical interface, desktop, and mobile integration. It has several collaboration features such as wikis, task management, and bug tracking and feature requests for every project. The repository offers plans for private repositories and free accounts, which are usually used to host open source software projects. *GitHub* reports having over 9 million users and more then 21.1 million repositories, making it the largest code hoster in the world.

For this TD-Tracker application, *ElasticSearch* was chosen because of its relevance on open source projects developed using Java with more then 11,000 positive evaluations on *GitHub*. The project has 7,222 issues created on its bug tracker repository, 6,266 of them closed and 966 open, used as TD candidates.

The first step in TD-Tracker, to integrate a TD catalog, was to create a connection between TD-Tracker and *GitHub*. For this, an existing *GitHub* user was necessary, as explained in the tool section. After configuring user *borante*, its password, and a repository project (*ElasticSearch*), it is possible to test the connection and check information about the issues created for the project. It is important to mention that this step is only for parameters configuration, not for real synchronization.

After configuring the parameters, the next step is to use *Integration Sync* interface to perform synchronization. TD-Tracker uses the parameters to extract TD candidates through API and presents it on a grid for manual selection. The manager can filter or sort all gathered information in this grid to register technical debt into the integrated catalog. The selected external information is retrieved from the identification tool and cataloged. In the *ElasticSearch* project issues registered into *GitHub* only the oldest was filtered (issues before 2012) and marked to sync, resulting in 8 TD candidates. The reason to use this filter is to catalog defect debts pending for too long (more than 3 years). After excluding the issues with some project developers already assigned, only 3 of the 8 issues became TD and were cataloged as presented in Figure 6. In this application example, all of the cataloged issues were assigned to user *Administrator*.

Unlike design issues presented in previous scenario, the 3 cataloged debts occurring on project *ElasticSearch* came from problems founded and not fixed, becoming defect debt. As a common scenario when dealing with bugs, the person who identifies does not know the code and, consequently, the exact location of the bug. With this, none of the previously cataloged TD has the location defined when synchronized. In Table 3, the three cataloged technical debts are presented.

6. Discussions

6.1. Approach. This initial picture will contribute to future research efforts concerned with continuously monitoring and managing TD in software projects, a larger subject that is out of the scope of this study. In this study we address the TD problem holistically for any kind of TD from identification to registering and managing the cataloged information. We also provided useful information on techniques and tools used

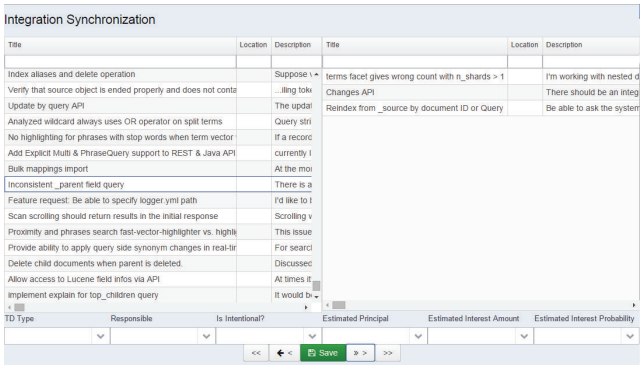

Figure 6: Synchronization between TD-Tracker and issues mapped for *ElasticSearch* on *GitHub*.

on identification activity. The tool application presented in this paper focused on an integration with *Sonar* but raises a question for further tradeoff analysis studies into how tools can help to point to TD that is worth being managed.

Besides some background mentioned regarding the portfolio approach proposed by [8], there is no definition in our extended catalog which establishes the correlation between two technical debt items. It is known that this correlation depends on the type of changes imposed on the system and it cannot be fixed. However, this lack of information (correlation between TDs) is planned as a topic for future work. This work is being developed to refine and expand the approach proposed in this paper.

6.2. TD-Tracker. It is possible to find several works related to decision making process in TD, such as [30, 32, 33]. TD-Tracker tool aims to help managers control TD and support their decision, establishing a relationship to these works. However, integrated catalog is an extension of previous work [4] with additional information to cope with manager needs and it can change from person to person, which is implied in expanded integrated catalog in future versions.

TD-Tracker may cause accumulation of technical debt. The integrated catalog is designed to be complete and intuitive but it cannot be fast enough to capture all information of technical debts and update them. Using another tool to manage the debt could make the responsible of the debt not update the information.

It is possible to extend TD-Tracker for software development companies, in order to get more people involved on technical debt scenario and its management. For that it is fundamental to provide a refined version of the academic application described and introduce the TD-Tracker for manual or integrated technical debt management process in their projects. With continued development, it is expected to discover ways to further support the projects' technical debt management through enhancements in the tool.

A unified integrated catalog is the key to control different processes and artifacts from a project. It allows managers to keep a list of closed and pending technical debts, helping with project planning, control, and execution. An integrated catalog also enables comparison of TD attributes for prioritization or applies techniques to rank and schedule TD payments. The tool also allows the users to handle multiple projects simultaneously, expanding TD cataloging and control. It also simplifies monitoring of identified TD in a single tool, independent of technical debt type.

7. Conclusions

The tool is designed to facilitate integration between different types of TD. It is possible through a unique integrated repository that supports all kind of debts and can group any information related to TD. TD-Tracker design focused on easily integrated databases and assists relationship mapping between external tools.

The approach produces and maintains a TD list according to the captured TD. TD List provides valuable information for existing software components. It can be helpful generating a product backlog list, providing a list of needed updates on documentation, prioritizing defects, and evidencing faults in testing area.

Two applications scenarios were presented; each one deals with a different kind of technical debt source code/design and defect. No application example of testing and documentation TD was presented due to the similarity of the steps when using database direct connection. It means that any integration can be created on database direct connection using the same steps as the first application example. Of course it is necessary for the TD identification tool to provide the connection credentials.

TD-Tracker also can capture human-detected technical debt. In addition to the integration, it is possible to catalog any kind of technical debt identified by stakeholders. Since they are fully aware of all active requirements and development conventions, they can find additional technical debts, not identified by any tools. This ensures that information can be cataloged, regarding the integration.

By grouping technical debt identification from different project development levels (code, test, and document), TD-Tracker tool ensures that the project is conducted while aware of technical debt's presence. This allows any stakeholder to avoid unintentionally increasing the number of TD by checking cataloged items of any project area. It also allows decreasing TD interest value by tackling technical debt in areas where the project is currently conducted.

When cataloging intentional debt, especially for defect debt, many of the items could be requirements that were not fully implemented. The intentionality of these items indicates that a decision was made not to finish the implementation of those requirements, most likely due to time constraints, which makes these instances conceptually different from defects caused by unintentional programming mistakes. For this action it is indispensable that the collector possess or receives the correct information to properly register postponed implementation.

TD-Tracker is being used by a small software development company for Java developers and testers teams. The project manager reported that TDs have been cataloged and

paid, but we still have no amount of data to apply any prioritizing technique, since the sampled TD set is not statistically significant. With continued development, getting more people involved on technical debt scenario and its management, it is expected to discover ways to further support the projects' TD management through enhancements in the TD-Tracker.

TD-Tracker allows multiple TD identification tools to be used instead of only one of the mentioned and the adoption of different tools is strongly recommended, since they point to different problems in a software artifact [20]. The use of a single tool or single indicator (e.g., single code smells) will only in rare cases point to all important TD issues in a project. As a result, project teams need to make intentional decisions about which of the TD indicators are of most relevance to them, based on the quality goals of their project, as suggested in [39].

In order to enhance productivity in the industrial sector we expect to further improve and validate TD-Tracker in such environments. For that, we have planned an extensive series of case studies with an experimental evaluation on open source software. These studies will reach closed and ongoing open source software projects with desirable characteristics such as access to software documentation, existence of repository issues, and test plans. Closed projects are also in the scope of validation because we intend to apply TD-Tracker in its previous released product versions, studying the life span and historical information of technical debt.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] W. Cunningham, "Object-oriented programming systems, languages, and applications," in *The WyCash Portfolio Management System*, 1992.

[2] N. Brown, Y. Cai, Y. Guo et al., "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*, pp. 47–52, ACM, Santa Fe, NM, USA, November 2010.

[3] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi, "Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, pp. 51–54, 2013.

[4] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," *Advances in Computers*, vol. 82, pp. 25–46, 2011.

[5] J.-L. Letouzey and M. Ilkiewicz, "Managing technical debt with the SQALE method," *IEEE Software*, vol. 29, no. 6, pp. 44–51, 2012.

[6] C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman, and F. Shull, "Organizing the technical debt landscape," in *Proceedings of the 3rd International Workshop on Managing Technical Debt (MTD '12)*, pp. 23–26, IEEE, Zürich, Switzerland, June 2012.

[7] J. A. Kupsch and B. P. Miller, "Manual vs. automated vulnerability assessment: a case study," *CEUR Workshop Proceedings*, vol. 469, pp. 83–97, 2009.

[8] Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *Proceedings of the 2nd working on Managing technical debt (MTD '11)*, pp. 31–34, Honolulu, Hawaii, USA, May 2011.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[10] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*, Springer, Secaucus, NJ, USA, 2005.

[11] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pp. 350–359, September 2004.

[12] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: a method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[13] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering*, pp. 97–106.

[14] F. Simon, F. Steinbrückner, and C. Lewerentz, "Metrics based refactoring," in *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pp. 30–38, March 2001.

[15] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10)*, ACM, 2010.

[16] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE '13)*, pp. 268–278, November 2013.

[17] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 22nd Workshop on Managing Technical Debt (MTD '11)*, pp. 17–23, ACM, May 2011.

[18] D. Binkley, "Source code analysis: a road map," in *Proceedings of the Future of Software Engineering (FOSE '07)*, pp. 104–119, May 2007.

[19] A. Vetro, M. Morisio, and M. Torchiano, "An empirical validation of FindBugs issues related to defects," in *Proceedings of the 15th Annual Conference on Evaluation and Assessment in Software Engineering (EASE '11)*, pp. 144–153, April 2011.

[20] N. Zazworka, A. Vetro', C. Izurieta et al., "Comparing four approaches for technical debt identification," *Software Quality Journal*, vol. 22, no. 3, pp. 403–426, 2014.

[21] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of production & test code," in *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST '08)*, pp. 220–229, IEEE, Lillehammer, Norway, April 2008.

[22] PMD, July 2015, https://pmd.github.io/.

[23] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, pp. 411–420, IEEE, Honolulu, Hawaii, USA, May 2011.

[24] Y.-G. Gueheneuc and H. Albin-Amiot, "Using design patterns and constraints to automate the detection and correction of

inter-class design defects," in *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39 '01)*, pp. 296–305, IEEE, Santa Barbara, Calif, USA, July-August 2001.

[25] C. Izurieta and J. M. Bieman, "A multiple case study of design pattern decay, grime, and rot in evolving software systems," *Software Quality Journal*, vol. 21, no. 2, pp. 289–323, 2013.

[26] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009.

[27] S. M. A. Shah, M. Torchiano, A. Vetrò, and M. Morisio, "Exploratory testing as a source of technical debt," *IT Professional*, vol. 16, no. 3, Article ID 6475929, pp. 44–51, 2014.

[28] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies," in *Proceedings of the ACM Symposium on Document Engineering (DocEng '02)*, pp. 26–33, ACM, New York, NY, USA, November 2002.

[29] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pp. 47–56, ACM, Denver, Colo, USA, November 1999.

[30] W. Snipes, B. Robinson, Y. Guo, and C. Seaman, "Defining the decision factors for managing defects: a technical debt perspective," in *Proceedings of the 3rd International Workshop on Managing Technical Debt (MTD '12)*, pp. 54–60, June 2012, http://www.scopus.com/record/display.url?eid=2-s2.0-84864135572&origin=inward&txGid=7200AE1F63AC9EDA2F928E7752950AC2.CnvicAmOODVwpVrjSeqQ%3a1.

[31] Y. Guo, C. Seaman, R. Gomes et al., "Tracking technical debt—an exploratory case study," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM '11)*, pp. 528–531, September 2011.

[32] C. Seaman, Y. Guo, N. Zazworka et al., "Using technical debt data in decision making: potential decision approaches," in *Proceedings of the 3rd International Workshop on Managing Technical Debt (MTD '12)*, pp. 45–48, IEEE, Zürich, Switzerland, June 2012.

[33] K. Schmid, "A formal approach to technical debt decision making," in *Proceedings of the 9th International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA '13)*, pp. 153–162, ACM, New York, NY, USA, June 2013.

[34] N. Zazworka, R. O. Spínola, A. Vetro, F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE '13)*, pp. 42–47, ACM, Porto de Galinhas, Brazil, April 2013.

[35] A. Charland and B. Leroux, "Mobile application development: web vs. native," *Communications of the ACM*, vol. 54, no. 5, pp. 49–53, 2011.

[36] DB-Engines Ranking, March 2015, http://db-engines.com/en/ranking.

[37] OpenRefine, March 2015, http://openrefine.org/.

[38] Elasticsearch, 2015, https://www.elastic.co/products/elasticsearch/.

[39] F. Shull, "Perfectionists in a world of finite resources," *IEEE Software*, vol. 28, no. 2, pp. 4–6, 2011.