JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2012; **00**:1–21
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

A Method for Investigating the Quality of Evolving Object-Oriented Software Using Defects in Global Software Development Projects

Ronald Jabangwe*†, Claes Wohlin, Kai Petersen, Darja Šmite, Jürgen Börstler

Blekinge Tekniska Högskola, 371 79 Karlskrona

SUMMARY

Context: Global software development (GSD) projects can have distributed teams that work independently in different locations or team members that are dispersed. The various development settings in GSD can influence quality during product evolution. When evaluating quality using defects as a proxy the development settings have to be taken into consideration.
Objective: The aim is to provide a systematic method for supporting investigations of the implication of GSD contexts on defect data as a proxy for quality.
Method: A method engineering approach was used to incrementally develop the proposed method. This was done through applying the method in multiple industrial contexts, and then using lessons learned to refine and improve the method after application.
Results: A measurement instrument and visualization was proposed incorporating an understanding of the release history and understanding of GSD contexts.
Conclusion: The method can help with making accurate inferences about development settings because it includes details on collecting and aggregating data at a level that matches the development setting in a GSD context, and involving practitioners at various phases of the investigation. Finally, the information that is produced from following the method can help practitioners make informed decisions when planning to develop software in comparable circumstances.

Received . . .

KEY WORDS: Global Software Development; Software Quality; Software Evolution; Object-oriented Software; Source Code Quality; Defect Analysis

1. INTRODUCTION

Global software development (GSD) has become increasingly popular, since it increases the availability of skilled resources, facilitates around-the-clock development and potentially saves costs [1, 2]. GSD projects face a number of additional challenges compared to collocated projects, among others assuring software quality [3]. Furthermore, it is not uncommon for the development settings to change during software evolution as a result of, for example, offshoring activities [3]. Such changes might have implications on software quality and should therefore been taken into account when studying evolving software systems.

In this paper, development settings are used to refer to project arrangement of teams and team members that are common in GSD. Teams can be distributed or dispersed, where the former is loosely coupled while the latter is coupled. Figure 1, which is adapted from [4], provides an

*Correspondence to: Blekinge Tekniska Högskola, 371 79 Karlskrona
†Email: ronald.jabangwe@bth.se

exemplar. In the figure, D1 and D2 are an example of distributed teams, whilst D3 and D4 are dispersed teams.

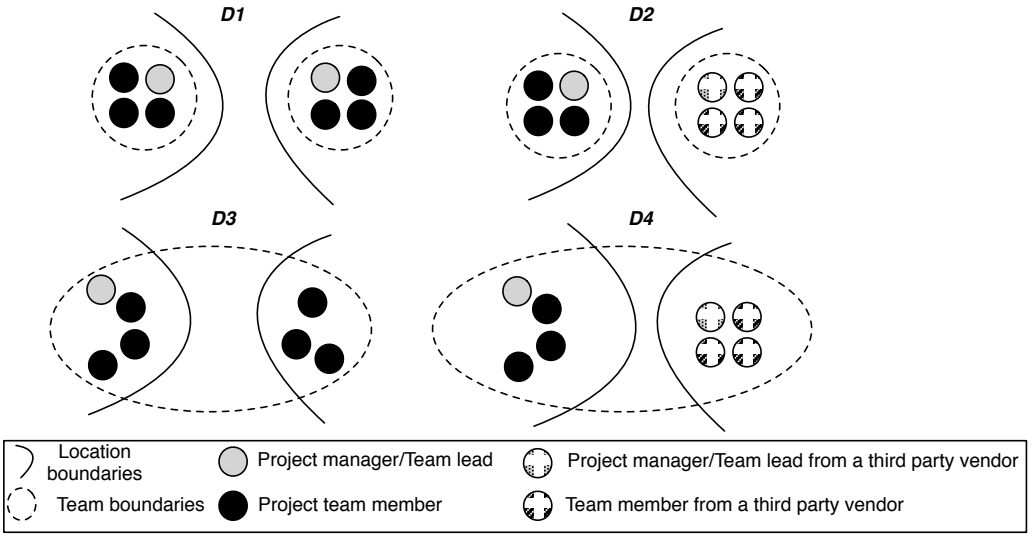


Figure 1. Example of Project Arrangements in GSD (Adapted from [4])

Studies on the quality of software developed in distributed environments are often based on defect data [5, 6, 7], like pre-release defects, post-release defects or defects reported during integration. Since GSD projects can be organized in many different ways [8, 4], certain defects are more useful for understanding the implications of certain development settings. For example, defects reported during integration would be more useful if teams in different locations are working independently on separate source code components for the same software product release.

Studies on software evolution and quality (e.g., [9]) seldom take events in the development setting into consideration as this may not be their primary concern, although the development setting may play a key role in product quality [10, 11]. The development setting, as well as changes to it, should therefore be taken into account when investigating the quality for evolving systems. For GSD projects this is particularly important, because the nature of collaboration between development locations has implications on quality [12].

Existing quality models characterize quality using various characteristics such as reliability, efficiency and portability [13, 14, 15]. Furthermore, they are quite general and provide little help when performing quality related investigations in GSD contexts.

The method presented in this paper seeks to address this issue. In terms of quality characteristics such as those found in quality models, the method involves the use of defect data, which can be traced to the quality characteristic reliability. It takes into account the development setting as well as GSD related events that change the development setting. It also details data collection and aggregation at a level that matches the development setting in the GSD context. The method focuses on object-oriented programming languages and is based on previous empirical research by the authors, e.g., [3, 16]. A method engineering approach [17] was adopted in the incremental development of the method. This involved a literature review, lessons learned from applying the method, and discussions with practitioners in industry.

The method is based on lessons from empirical studies presented in [3, 16, 18] and [19], and the data collection methods for each study are presented in detail in each study. Included in the present paper, apart from the complete method, are details of how the method was developed and how it was refined, as well as how the lessons learned from each individual empirical study helped in the development of the method. The individual empirical studies provide in-depth case study findings that allowed us to gain valuable insights about defect history in the context of GSD for evolving software systems. The studies do not comprise or present the method. Thus, the lessons from each

DOI: 10.1002/smr

individual study provide an incomplete representation of the method presented in the present paper. We, therefore, gained insights through the individual studies, and this enabled us to construct a systematic and coherent method by following a method engineering approach. Therefore, the current version of the method presented in this paper is a synthesis and aggregation of lessons learned from GSD empirical studies presented in [3, 16, 18] and [19]. Although we do not invent the method fragments and the fundamental properties of the method (e.g., the importance of understanding context and the goal-oriented approach), the novelty of the method is in how it combines them into a coherent structure and the insights that are obtained in following the method.

Overall, we believe this present paper makes a substantial contribution by proposing the first systematic method for investigating quality, using defects, for software systems that are developed, as well as evolve, in GSD contexts. The expectation is that practitioners and researchers alike can also use the method to gain insight into their own GSD contexts using the method. This method will help with making accurate inferences because it includes details on collecting and aggregating data at a level that matches the development setting for GSD contexts, and involving practitioners at various phases of the investigation. Finally, the information that is produced from following the method can help practitioners make informed decisions when planning to develop software in comparable circumstances. Rigorous evaluation of the method in terms of its ease of use, usefulness and how well it helps with achieving the aforementioned goals, is highlighted as future work.

To summarize, this paper outlines a method for investigating quality, using defects, for software systems that are developed, as well as evolve, in GSD contexts. Thus, in the context of the method, quality investigation refers to the investigation of quality using defects. The objective of the method is as follows:

- To improve the consistency in the manner in which GSD studies are performed to make it easier to compare and synthesize findings across multiple studies.
- To reduce the likelihood of researchers making incorrect inferences in their studies. It does this by outlining a rigorous approach of understanding the GSD context and working in close collaboration with practitioners during the investigation.
- To increase the likelihood of researchers providing well-founded information and recommendations to practitioners for developing software in GSD contexts.
- To provide an instrument to practitioners to understand the implication of GSD contexts with regards to reported defects, as an indicator of quality, across multiple releases.

The remainder of the paper is organized as follows. The background is presented in Section 2. The research approach used in the development of the method, i.e., method engineering, is described in Section 3, and the latest version of the method is presented in Section 4. Lessons learned that led to the modification of the method, and from applying the latest version of the method are discussed in Section 5. Implications of the method for research and practice are discussed in Section 6. Conclusions and future work are presented in Section 7.

2. BACKGROUND

With the increasing popularity of GSD projects in industry there is an increasing number of studies in the area [1]. Often studies on quality perform their investigations using different types of defects. Notably, there are studies that use post-release defects, (e.g., [5, 12]), both pre- and post-release defects (e.g., [6]), and other studies use those reported during integration (e.g., [7, 20]). However, team setup in GSD projects can differ across distributed projects. For example, team members can be dispersed and working jointly but in different locations or individual teams can be in different locations working independently [4]. Therefore, it is unclear which defects are appropriate for better understanding the implications of development settings in GSD.

The difference in the development settings and the data collection in distributed development studies in GSD is a possible source of inconsistent findings reported across the studies. Cataldo and Herbsleb, for example, report a link between distributed development and quality [7], whilst Bird et al. report contradicting findings [21]. Bird et al. used post-release defects and the development setting consisted of developers within close proximity (in the same building). Cataldo and Herbsleb,

DOI: 10.1002/smr

in comparison, used pre-integration defects from a development setting that consisted of developers spread across national boundaries working in dispersed feature teams. This makes it difficult to compare the findings.

Generally, there are few studies on quality across GSD studies that use quantitative data [22], and those that are available differ in terms of the data used and the development settings studied. This makes it difficult to compare results. Inconsistencies in study design and execution have been noted as a cause for concern in empirical software engineering research [23, 24]. Runeson et al. [25] noted a similar issue in their study on fault-prone components, which prompted them to propose a classification scheme for studies within that area. Similarly, the method documented in this paper provides an initial approach to address the inconsistency in empirical studies in GSD when investigating quality (using defects) for evolving object-oriented software systems.

The method presented in this paper was formulated for a specific goal, and then reformulated multiple times using lessons learning from experience of applying the method. This goal-oriented approach is also emphasized by the goal-question-metric (GQM) [26]. The goal-oriented approach from GQM together with the process-solving iterative cycle of the plan-do-check-act cycle (PDCA), which emphasizes learning from experience, is also the basis of the software process improvement paradigms, such as the quality improvement paradigm (QIP) [27, 28]. Thus, the key aspects of the method proposed in this paper, which includes taking into account the development context, are not particularly unique. However, the novelty of the method is in the configuration of the method and the insights obtained from following the method, specifically for software that evolves in GSD contexts. The method provides a structure for investigating software quality using defects as a proxy for quality. Therefore, the purpose of the method, the focus on GSD, and the insights obtained from following the method differentiates it from general software process improvement paradigms.

3. RESEARCH APPROACH

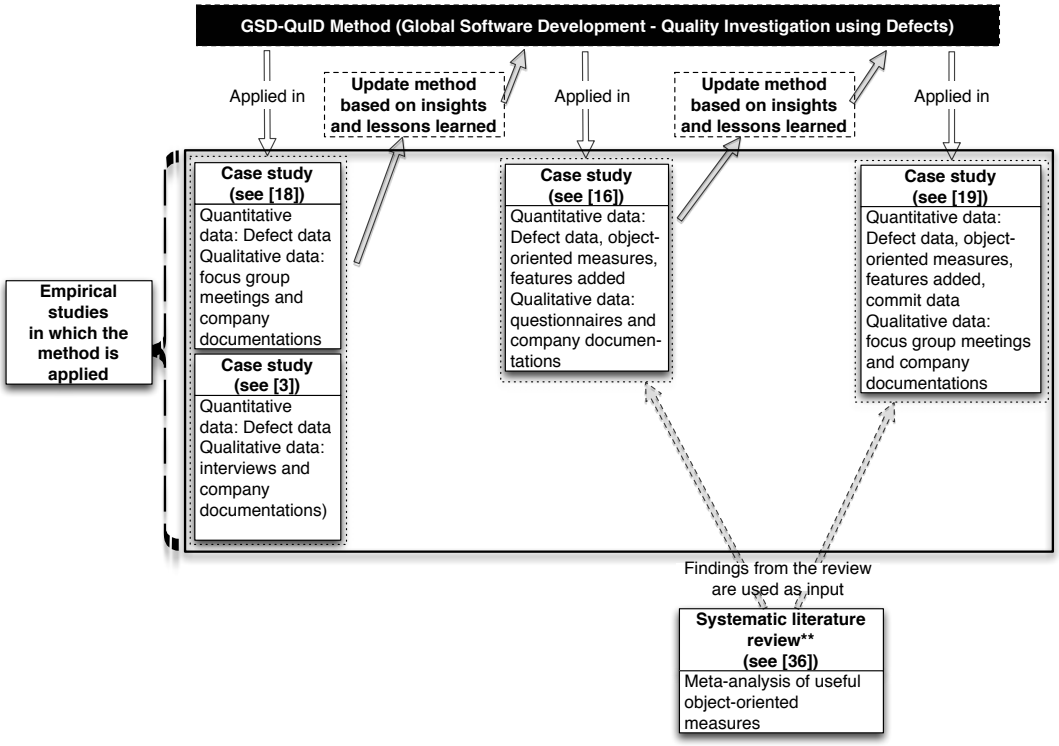
The method presented in this paper is a result of numerous refinements and improvements using lessons learned from applying the method in multiple GSD contexts in industry. This approach of developing a method is referred to as method engineering. Method engineering is characterized as the development of methods or processes through systematic adaptation, tailoring, and improvement [29]. An example of the steps that were followed in the development of the method, and the method fragments, can be found in the report by Mayer et al. [17].

Figure 2 shows the link between the modifications of the method and the empirical studies in which the method was applied. It captures the steps involved in the iteration and refinements that were done to the method. Each of the refinement is based on lessons learned from applying the method in separate GSD empirical studies.

As shown in Figure 2, the method was first applied in two separate empirical studies, i.e., [3] and [18]. The method was then modified based on the experience and lessons from applying it in the two studies. This modification included using input from a systematic literature review that was performed to identify empirically evaluated source code measures for object-oriented software systems. Overall, this modification resulted in an updated method, which was then applied in a another empirical study that is presented in [16]. Further refinements were then performed based on insights and lessons learned from the application of the method in that study. This resulted in the latest version of the method that is presented in this paper, which was also applied in another empirical case study, i.e., [19].

Table I shows the information about the studies and the GSD contexts in which the method was applied. The main events related to GSD that occurred during evolution of the products that were used in each of the studies are also provided in the table.

DOI: 10.1002/smr



*** The systematic literature review was performed by following the steps proposed by: Kitchenham BA and Charters S, "Guidelines for performing systematic literature reviews in software engineering," Technical Report EBSE, July 2007.*

Figure 2. Development of the Method

4. METHOD FRAGMENTS FOR GSD-QUID METHOD (GLOBAL SOFTWARE DEVELOPMENT - QUALITY INVESTIGATION USING DEFECTS)

The latest version of the method is shown in Figure 3. The method consists of four phases. The dotted boxes within each phase are referred to as method fragments [30]. Definitions or descriptions of key terms used in the context of the method are provided in Table II.

Phase 1 involves gathering contextual information, e.g., the specific GSD context. Phase 2 involves preparation of quantitative data, i.e., actual defect data and measures linked to defects. Data analysis is carried out in Phase 3. In Phase 4, the results of the analysis are evaluated taking into consideration the context.

DOI: 10.1002/smr

Table I. Application of the Method: Empirical Studies and the GSD Contexts

	Empirical Studies and GSD Context	**Main Events**
First Insights to the Method	The method was first applied in two separate studies (see [3] and [18]). The case company was Ericsson, a multinational telecommunication corporation. In [3] data from one product from the company was used, and in [18] data from two products was used. The products used in the studies were large software systems that were part of a compound system. The sites involved in the development of the products were located in Sweden and India. This type of a project is characterized as offshore insourcing [8].	Offshoring as a transfer (the sending site was in Sweden and the receiving site was in India).
Second Insights to the Method	Based on insights from previous studies in [3] and [18] the method was refined and applied in the study that is presented in [16]. The company, which requested to remain anonymous, provided gauging solutions to companies that stored and transported gas and chemical substances like oil. Two medium-sized products from the company were used in the study. The products were each developed in a distributed project with dispersed teams from two of the companys sites that were located in Sweden and Russia. This is also characterized as offshore insourcing [8].	Offshoring of project management responsibilities (from the site in Sweden to the site in Russia).
Latest Version of the Method	The method was further refined based in insights from the study presented in [16]. The method was then applied in a study that was conducted at a company that develops online gaming systems (see [19]). The company requested to remain anonymous. The system used in the study was a medium-sized product that was sold with several games. It was developed in a project with distributed teams between the case company and two offshore companies. The site for the case company that was involved in product development was located in Sweden, whilst for the two vendors one was located in Ukraine and the other in India. Whilst the product was evolving there were also changes in the number of vendors involved in product development. This type of a project is characterized as offshore outsourcing [8].	Change in characteristics of a setting (i.e., multisite environment that had distributed teams, which incurred changes in sites involved, and eventually resulted in single-site execution).

The numbering of the phases suggests a logical order of how to follow the method. However, in certain cases it may be necessary to iterate some phases. For example, if there are data anomalies or errors uncovered during Phase 3 or Phase 4, it would be important to redo Phase 2. It may also be necessary to improve the visualizations if there are difficulties with understanding them, in which case Phase 3 would need to be performed again. Phase 1 should be redone whenever there are any aspects about the context that are unclear. This can be, for example, uncertainties about the period during which certain events occurred and the exact releases that were developed during those events.

The suggestion is to involve experts that have extensive knowledge about the evolution of the product during all four phases of the method, so that they can help with verifying the accuracy of the data and information used in the investigation. More details about each phase in the method is provided in Sections 4.1-4.4.

DOI: 10.1002/smr

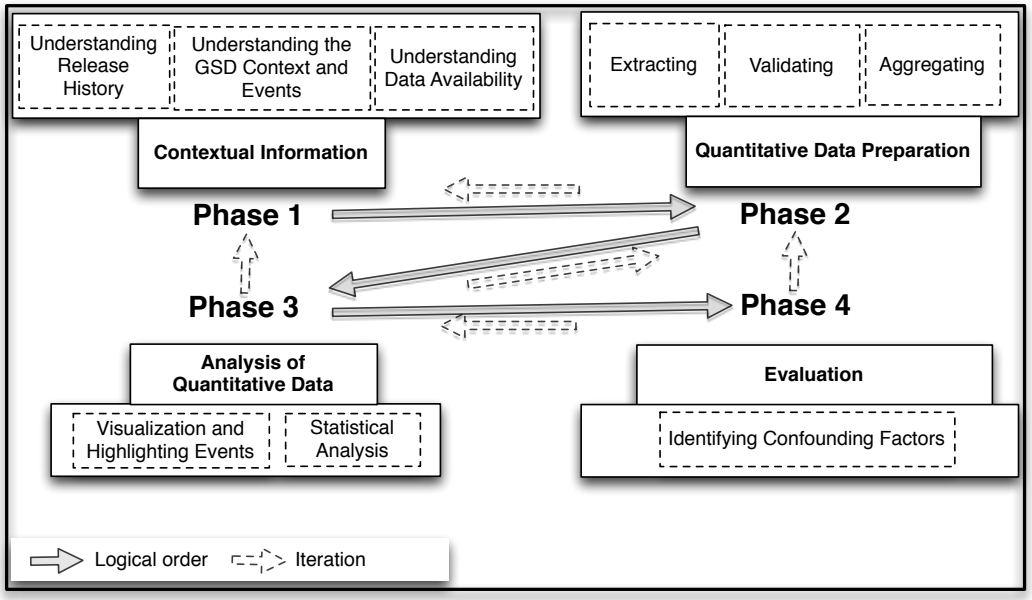


Figure 3. GSD-QuID (**G**lobal **S**oftware **D**evelopment – **Qu**ality **I**nvestigation using **D**efects)

4.1. Phase 1: Contextual Information

4.1.1. Understanding Release History. The period of investigation should be well defined, and this is influenced by release history and the availability of data for the releases. The two main types of releases that are common for evolving systems are evolutionary and maintenance releases (See Table II for descriptions). In the context of the method, release history refers to the time period that is of interest in the investigation, number of product releases, and the dates that each release was made available to customers within that time frame. This information is important for constructing an evolutionary timeline of the product.

4.1.2. Understanding the GSD Context and Events. GSD contexts can consist of collaboration between developers in separate locations, with small to large geographical or temporal distance, as well as from the same or different companies [8]. These varying characteristics are illustrated by Šmite et al. [8] and are summarized in Table III.

In terms of software development activities, typically the development settings in GSD can be distinguished by the project arrangements. Typically projects can have teams that are distributed or team members that are dispersed. A distributed project with dispersed terms is an environment in which team members are dispersed across different locations and engage in collaborative development with the intention of producing a software artifact, e.g., a component [4]. A distributed project with distributed teams is an environment in which teams are in separate locations, and each have collocated team members, and perform their development activities independently with the goal of integrating their work into one larger software artifact release [4].

DOI: 10.1002/smr

Table II. Key Terms (In the context of the method)

Terms	Descriptions
Commits	Detailed information about changes made to the source code artifacts for enhancements and corrective purposes.
Defect data	Any issues that deviate from the expected external behavior of the system that is traced to an issue in the source code.
Distributed project with dispersed teams	This is when team members are dispersed across different locations. This forms a virtual team [4] where the dispersed team members are collaborating in the development of a software artifact, e.g., component or software release.
Distributed project with distributed teams	This is when there are separate teams in different locations, with collocated team members, and the teams execute development tasks independently. This forms loosely coupled teams [4, 31]. Eventually software development work from all teams is integrated into one larger software artifact.
Evolutionary releases	Releases that mainly consist of new features that are not implemented in any of the previous releases.
External quality measure	Defect data.
Features added	This pertains to new features as defined in [32] that is not present in previous releases.
Maintenance releases	They are intermediate releases that mainly consist of defect-fixes.
Object-oriented measures	Measures that quantify source code properties of object-oriented software systems, .e.g, complexity, inheritance, coupling, etc.
Offshoring	"Leveraging resources from a different country" [8].
Product modification measures	Features added data and commit data.
Release history	Releases made available to customers in a given time period.
Single-site development	This is when all development activities are performed at a single development location.
Source code measures	Source code measures and object-oriented measures are used interchangeably in the context of the method. Please see description of object-oriented measures.

Table III. Characteristics of GSD Contexts

Characteristics of GSD projects	Explanation
Geographical distance	Distance between, or proximity of, the development locations
Legal entity	Insourcing ("leveraging company-internal human resources" [8]) or outsourcing ("leveraging external third-party resources" [8])
Location	In relation to the case company, onshore or offshore
Temporal distance	Difference in time zone between development locations

Given the different characteristics (see Table III), and the different ways that development locations can collaborate, there are many changes that can alter the development setting in GSD. Table IV contains some common examples of events, but it should be noted that it is not an exhaustive list.

DOI: 10.1002/smr

Table IV. GSD-related Events that Change the Development Setting in GSD

Example of events	Explanation
Change in characteristics of a setting	Pertains to changes in any of the characteristics in Table III, e.g., change in the number of development locations involved, or change from an offshore company to an onshore company, etc.
Change of project arrangement	Changes between an environment with distributed teams to one with dispersed team members or vice versa
Offshoring as a transfer	Relocation of development work, as observed in software product transfers [18, 33], which may result in loosely coupled distributed teams, dispersed teams, or single-site development in an offshore location.

4.1.3. Understanding Data Availability. The following would be helpful for investigating defect-related quality of software intensive systems: defect data, source code artifacts (for extracting object-oriented measures), commit data and features added across releases. In the context of the method, defects are used as an external quality measure. An external measure reveals information pertaining to the external behavior of the system [15]. Object-oriented measures quantify the intrinsic and static source code properties of object-oriented software [15]. Example of properties that are quantified by object-oriented measures are size, inheritance, coupling and complex properties [34, 35]. Whereas external measures use a "black box" approach to uncover information about the quality of the software system, internal measures take a "white box" approach. Data about the commit activities and features added across releases capture the modification and evolution of the software system. Commit data contains details about the people involved in changing the source code. Features added details the enhancements and improvements implemented.

The information gained in this phase is mostly qualitative in nature. This can be captured in the form of documents or reports, and they can also be recorded in the form of audio from discussions with company representatives during, e.g., focus group meetings or interviews.

Figure 4 provides a depiction of the relation between GSD contexts and quality.

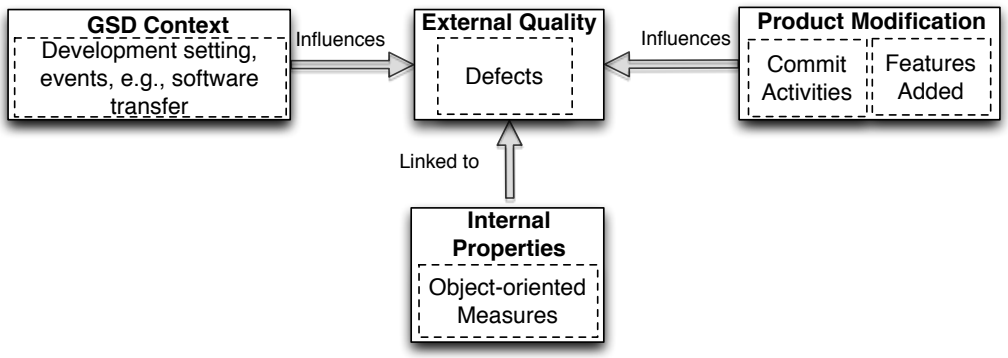


Figure 4. Relation between GSD and Quality

DOI: 10.1002/smr

4.2. *Phase 2: Quantitative Data Preparation (Defects, product modification and source code measures)*

Phase 2 involves extracting, cleaning, and aggregating defect data, which is the quality indicator, as well as product modification and source code measures, which are linked to defects.

4.2.1. Extracting. Tools and identification of useful measures for extraction are important in this method fragment.

Tools: The extraction of defects, features added and commits is dependent on the tools used to store and maintain the data. Some examples are JIRA[†] for defect data and features added, and Subversion[‡] for data about the commit activities. Tools for extracting object-oriented measures are dependent on the programming language. Given that nowadays it is common to find systems that are developed with more than one language, tools that can operate on multiple languages would be probably more sought-after. Examples of such tools are the Understand Tool[§] and Source Monitor[¶]. However, the discussion on tools is a broad area that is beyond the scope of this paper.

Measures: There is a plethora of object-oriented measures, and it is impractical to extract and collect all measures. Ideally, a few appropriate measures that quantify different properties should be selected. The systematic literature review on empirically evaluated measures that was performed by Jabangwe et al. [36] can help with identifying useful measures to extract. For example, results in the study suggest that measures that quantify complexity, coupling and size properties are good indicators of quality related issues, and an example of these measures is listed in Table V. However, it is important to note that the relationship between the measures and quality does not imply cause-effect relationship.

Table V. Example of Empirically Evaluated Object-oriented Measures

Measure	Definition	Source code property quantified
CBO	Coupling between objects [34]	Coupling
LOC	Lines of source code [36]	Size
RFC	Response for a class [34]	Complexity
WMC	Weighted method per class [34]	Complexity

It should be noted that the measures presented in Table V are only a subset of good measures to include when investigating quality. There are other measures that can also be considered. In their meta-analysis of useful source code measures for object-oriented software, Jabangwe et al. [36] also present other measures, such as code churn measures, McCabes cyclomatic complexity measure, etc.

Post-release defects can help to understand the change in overall product quality when the product is on the market. However, given the unique distributed nature of development locations, defects reported during integration or pre-release are better for determining the effect of the GSD context and the GSD-related events. This is because problems can occur during integration [7], but this is not evident from the post-release defects as they may already have been fixed. Therefore, to better understand changes in quality in GSD contexts, defects need to be from a level that is representative of the development setting. The proposal is that if it is a distributed project with dispersed teams such that the development locations are tightly coupled and are working in a collaborative manner on, for example, components, then pre-release defects that are traceable to the specific components should be collected. If the development locations are working independently in a distributed project

[†]More information on JIRA can be found on: https://www.atlassian.com/software/jira
[‡]More information on Subversion can be found on: https://subversion.apache.org/
[§]More information about Understand Tool can be found on the website: http://www.scitools.com/
[¶]More information about Source Monitor can be found on the website: http://www.campwoodsw.com/sourcemonitor.html

DOI: 10.1002/smr

with distributed teams, then a separate analysis of defects found during testing at each location and those reported during integration would provide a better view of how quality changed over time.

A simple measure from commit data is the total number of commits. The number of commits should be computed per development location. If the data permits it can be computed by number of developers per development location. Other commit related measures proposed and empirically evaluated on their link with quality can be found in [10, 37, 38, 39]. Similar to defects, the measures extracted from commit data should be at a level that is indicative of the GSD development setting. For example, if it is a distributed project with dispersed teams then the measures from the commit data should be traced to the component level. In a distributed project with distributed teams the commit data can be traced to the development locations or sites.

4.2.2. Validating. After extracting the quality and product modification measures, it is important to verify the accuracy and plausibility of the data. The foundation of the method is to enable the analysis of quality from the perspective of the source code. Thus all measures need to be traceable to the source code artifacts, and all other data should be excluded. This means that defects should be linked to a problem in the source code, and each commit should be traceable to changes made to the source code. Given that the method is intended for evolving software systems, data should be traceable to specific evolutionary or maintenance releases.

4.2.3. Aggregating. Often companies assign severity levels to defects to help with prioritizing maintenance activities. As an example, this severity level may be an indicator of the effect of the defect on system performance and security. This categorization can be used to further split the data into, for example, high, medium and low severity. The dates for which each defect was reported can then be used to count the number of defects reported for each severity per day, week or month. If possible, defect data can further be split into customer reported versus internal reported defects.

Companies may also assign criticality levels to the features added across releases. The criticality may be, for example, based on the customer demand for certain features. This can also be used to categorize the data. Grouping of commit data can be done per development team, site, or location, etc. After the measures are aggregated they are then ready for further analysis.

4.3. Phase 3: Analysis of Quantitative Data

Phase 3 entails analyzing the quality and product modification measures in relation to the events in the development setting across releases. Two possible approaches for the analysis are visualization and statistical analysis.

4.3.1. Visualization and Highlighting Events. There are many visualization approaches that can be used to represent the quality of evolving software systems [40, 41], for example, kiviat (spider) diagrams [41, 42] and memory graphs and tree maps [43]. However, from experience, the less complex and simple two-dimensional visualization approaches, e.g., bar charts and line graphs, are efficient and effective for analyzing quality changes. We also surveyed papers on evolution and quality, and also reviewed papers from the systematic literature review on object-oriented measures and quality (from [36]), and found that bar charts and line graphs were the most common methods of visualizing changes in measures.

Data about the features added and commits can be visualized across releases using bar charts. The releases are then numbered R_1 to R_n on the x-axis, and the total number of features added and commits for each release on the y-axis. The bar charts will help visualize changes in the data across releases.

Defect data and object-oriented measures for both evolutionary and maintenance releases can be a large amount of data. The important aspect to get from the data is the trend over time. Hence, a line graph is ideal to visualize large amount of data from object-oriented measures and the defect data. For both types of data the total values is placed on the y-axis, and the labels for the specific releases are then placed on the x-axis for the graph with object-oriented measures. For the graph with defect data, the time period is shown on the x-axis. As an example, if the defect data is between 2001 and

DOI: 10.1002/smr

2010 and the defects shown in the figure are those reported per day, it would be difficult to state every date on the x-axis; indicating the years is sufficient in this example. Another visualization approach is a cumulative flow diagram. It can be used to show the rate at which defects are reported and fixed thus showing the defect-resolution cycles across releases. Examples of how to use cumulative flow diagrams can be found in the studies by Petersen and Wohlin [44] and Jabangwe et al. [18].

Visualizing a Combination of Measures: Combining measures in one visualization method helps to identify any similarities or differences in patterns of the measures, across releases. A heat map, which uses color intensity to highlight variation of high to low values, can be used to visualize how measures change from one release to another as exemplified in [16]. Another approach is to normalize each quality measure and visualize them in the same bar or line graph. Moving-range chart, a visualization approach often used in Six Sigma related studies [45], can be used in this context to investigate shifts in measures between releases, at different periods during evolution. The charts can also be useful for investigating the link in patterns between quality measures and product modification measures as well as periods with unusual shifts. A similar approach is used in [16].

Highlighting Events of Interest in the Visualizations: The events such as the ones indicated in Table IV can be highlighted in the visualizations. The choice of the event to highlight is most likely driven by the purpose of the investigation. For example, if the purpose is to understand how an organizational change influences defect patterns, then the period before, during and after the change would be highlighted in the visualizations. This helps with investigating the relation between the changes in the measures and the events in the development setting.

Interpreting Visualizations: Generally, the higher the number of defects reported the lower the quality. In addition, an increase in complex features added, coupled with an increase in the number of people making commits can have a negative impact on quality. The interpretation of object-oriented measures relies on the type of source code property quantified, how they quantify the property, and their relationship with external quality attributes. Meta-analysis results that show the direction of relationship of the object-oriented measures and quality attributes, as presented in [36], can be used to interpret the change in the measures. This interpretation of object-oriented measures can also be combined with the software evolution laws presented in [46].

Example of a Visualization: Figure 5 uses hypothetical data to show how the visualization can be done[‖]. This is just one possible illustration of how all the data can be visualized to help with the analysis. In Figure 5, the commit data is aggregated at the development team level, and it shows the organizational change that occurs between each development setting. In this example, the characterization of defects from high to low relate to the effect the defects have on the systems reliability, i.e., high defects cause system failure whilst low defects are trivial issues. For features the characterization from high to low pertain to the importance of the feature to customers.

From the example in Figure 5, it can be observed that despite no visible difference in features added across releases, during "Development Setting 2" there is a higher number of defects reported and there is an increase in complexity. This suggests a decrease in quality. The commit data indicates a link in the decrease in quality with the number of development teams involved, given that there are three teams involved during "Development Setting 2" in comparison to only one during the other development settings. However, there may be other confounding factors, such as, change in defect reporting processes. Therefore, further inquiries should be done with practitioners to better understand how the development setting influenced the changes in quality measures, and to identify any confounding factors. This is the purpose of Phase 4.

[‖] A similar visualization approach is used in the article by Jabangwe et al. [19]

DOI: 10.1002/smr

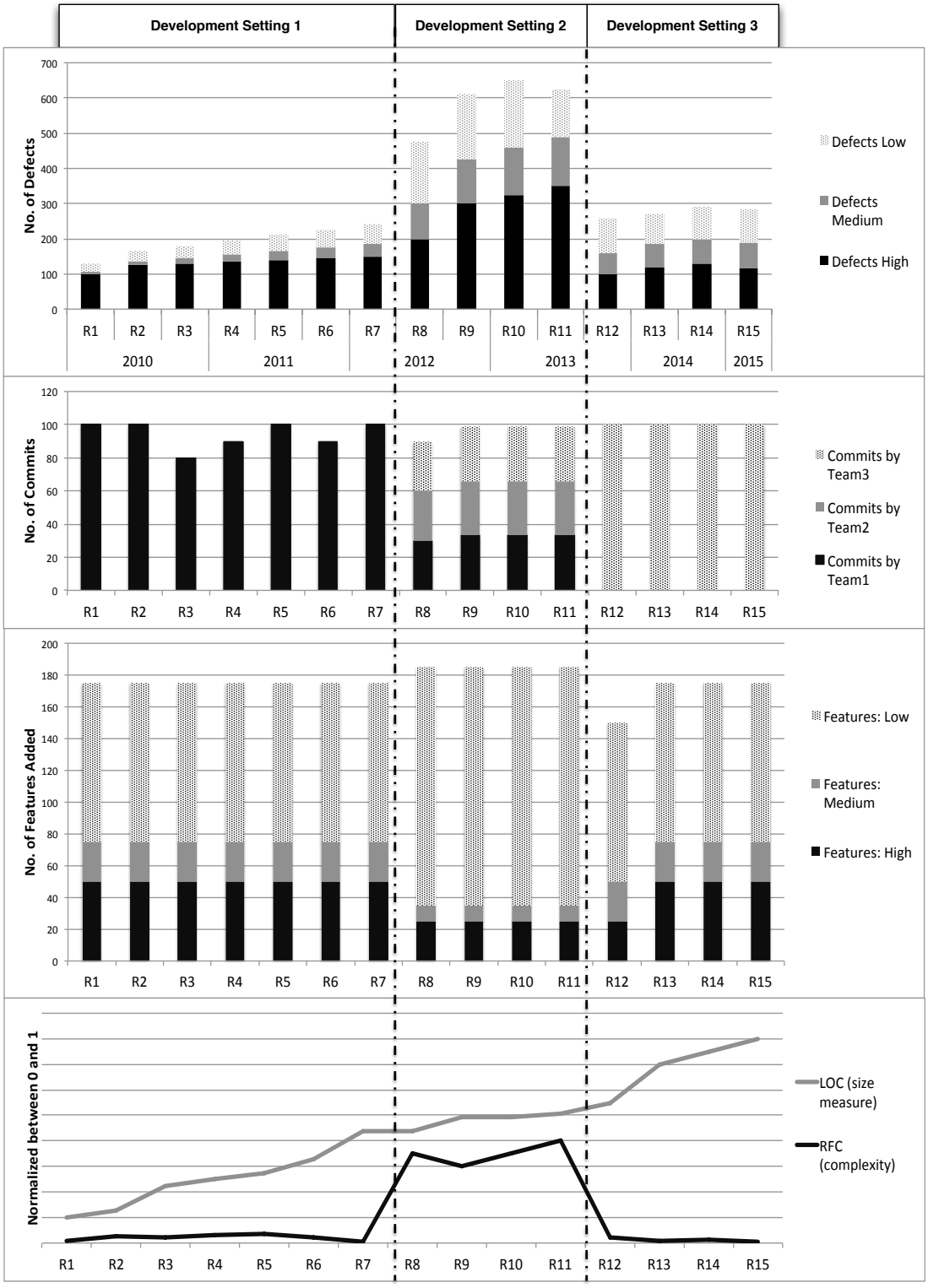


Figure 5. An Example of How to Visualize Data

Prepared using smrauth.cls
DOI: 10.1002/smr

4.3.2. Statistical Analysis: Statistical methods can be used to analyze the quality and product modification measures together with the development setting. Examples are correlation analysis, univariate and multivariate analysis. This analysis approach can be compared to combining the measures in a visualization approaches. This is because both approaches can be used to assess the existence of a relationship between either the measures themselves (e.g., quality measures and product modification measures), or the measures and the events in the development setting.

4.4. Phase 4: Evaluation (of Quantitative Data Analysis Results Given the Context)

The purpose of Phase 4 is to get deeper insights and explanations about the changes in defects, and measures that influence or is linked to defects. During this phase the GSD context and events in the development setting are taken into consideration to identify any confounding factors.

Identifying Confounding Factors. Company representatives that are directly and indirectly involved during the evolution with the product can provide information on factors that contributed to the patterns in the quality and product modification measures. This information can be elicited from them through, for example, focus group meetings during which the patterns in the data are presented and discussed.

There are existing studies that identify factors that can influence quality, e.g., architectural restructuring [18] and refactoring activities [47]. Factors reported in similar studies can be evaluated on their relevance as a confounding factor in the investigation. This can be verified by experts at the company as well as reviewing product documentations.

This method has been applied in three published empirical studies i.e., [3, 16] and [18]. Each of the studies lists critical factors that can influence quality for software developed in GSD contexts. Thus, the results in the studies can be used as a source of additional factors to consider during the evaluation phase. Moreover, studies that reflect on software engineering contexts (e.g., [24] and [48]) can also help with identifying confounding factors.

5. LESSONS LEARNED

Lessons learned from applying the GSD-QuID method, and which subsequently led to the modification and evolution of the method, are presented in Section 5.1. The lessons learned from applying the latest update of the method, which is presented in this present paper, are discussed in Section 5.2.

5.1. Evolution of the Method and Link to Empirical Studies

There were two major modifications to the GSD-QuID method, and they were done based on lessons learned from applying the method in separate empirical studies. The modifications and evolution of the method are as follows.

First Insights to the Method: The method originated from an empirical investigation initiated by one of our industry partners. The company had an interest in understanding the impact on quality of offshoring software development activities for their evolving software systems. The initial version of the method involved the use of defect data as the only quantitative data (see [3] and [18]).

When discussing the increase of defect inflow during offshoring activities, company representatives in the empirical studies in [3] and [18] reflected on the following: the varying complexity and size of the product across releases; and the difference in the total number of features, as well as their complexity, across releases. Based on their reflections, a decision was made to extract object-oriented measures across releases that capture complexity and size properties of the source code, and to also collect and analyze features that were added across releases in order to understand the changes that led to shifts in complexity and size between releases. Since there are many object-oriented measures in literature, e.g., [34, 49] and [35], a systematic literature review was performed to identify the most useful measures. The systematic literature review is presented in [36], and it is used as a basis to decide which measures to include in the method.

DOI: 10.1002/smr

Second Insights to the Method: The improved version of the method included analysis of defect data, and object-oriented measures, as quality measures, and features added as a product modification measure. This version of the method was then applied in a different GSD study that involved distributed development and a handover of project management responsibilities to an offshore location [16].

With the visualization of both the object-oriented measures that were extracted from releases and features that were added across releases, company representatives in the study presented in [16] were able to picture the evolution of their product based on defects as well as data that is linked to changes in defects, i.e., features added and object-oriented measures. This provided a better picture of quality than just using defect data, which was the case in the previous studies in [3] and [18]. Company representatives in the study presented in [16] were able to better retrospectively reflect on trends and shifts across different GSD development settings in defect data, source code complexity and size, across releases. As a result we were able to better investigate how the trends and shifts related to patterns in defects inflow and GSD events across releases.

A post-mortem analysis of the study presented in [16] revealed a need to verify reflections from company representatives regarding the nature of collaboration between developers, development sites or development teams, across releases. This information is tracked by version control systems in the form of commit data, and contains information regarding how, as well as what, developers contributed to during the development and maintenance activities across releases. This information is vital given that changes in the number of contributors (or number of those making commits) across releases can have implications on how quality varies [10].

Object-oriented measures were only extracted from evolutionary releases (see [36]), i.e., releases that introduce new features. Measures from maintenance releases, i.e., releases that contain mainly defect fixes, needed to be included to have a better evolutionary view for the products. This also ensured that object-oriented measures were consistent with defect data, which were collected for all releases for a given time period. In addition, this also helped to better understand how internal properties vary between main and intermediate releases in relation to the defects.

Latest Version of the Method: Finally the method prescribes gathering defect data and object-oriented measures from maintenance and evolutionary releases. In addition, feature added data and commit data were included in the method as product modification measures. The rationale for including commit data in the data collection when conducting the study in [19] was to use the data to better understand the GSD context in terms of development activities. This becomes essential when the product has a long history with many evolutionary changes. and there is a need to better understand shifts in development activities and possible links to defects trends. A good example is the products in the study that is presented in [16], which was conducted prior to the study in [19]. Lack of commit data in the study in [16] meant that we were unable to corroborate opinions of employees at the case company in relation to how shifts in development activities could or could not have been linked to defect trends or change in source code complexity and size.

In the study presented in [19], the commit data revealed how the employees or different locations in the GSD project contributed to the evolution of the software product. We were able to trace these changes to specific releases. Using the commit data we were able to identify and clarify inconsistencies between information in the data and the retrospective reflections from company representatives on how development activities were performed. Presenting and discussing the data with company representatives also helped with capturing a more accurate picture of how and why employees in the GSD project collaborated in a particular manner, and how this changed across software releases. In turn, we were able to better understand the link in the changes to defect trends or change in source code complexity and size. This helped to better understand the GSD context as well as quality in that context.

Table VI provides a summary of the data captured apart from defect data, and the aims of each study that were achieved with the help of the method, GSD-QuID. The studies are presented in the table in the order in which they were performed.

Overall, the latest update of the method was applied in an offshore outsourcing context, which is a study presented in [19]. After the study, the method was presented to, and discussed

DOI: 10.1002/smr

Table VI. Data Collected and Aims Using the GSD-QuID Method

Studies (in chronological order)	**Data captured (apart from defect data)**	**Aims achieved Using GSD-QuID**
[3] and [18]	Opinions from company representatives on change in defect inflow across releases using interviews [3]; Insights from company representatives on reasons (e.g., events) linked to change in defect inflow and defect resolution cycles across releases using focus group meetings [18].	Exploring the changes in quality across releases before, during and after offshoring activities [3]. Exploring defect-resolution cycles across releases before, during and after offshoring activities [18].
[16]	Object-oriented measures that capture size and complexity properties; Features added across releases; Opinions from company representatives of the observations made in the analysis of the defect data and the object-oriented measures (using questionnaires).	Performing an explanatory study on the implication on quality across releases of handover of managerial responsibilities in an environment with dispersed teams.
[19]	Object-oriented measures that capture size and complexity properties; Features added across releases; Commit data; Reflections from company representatives regarding events and other factors that were linked to the change patterns in the quantitative data across releases using focus group meetings.	Exploring the implication on quality of distributed development and changes in number of companies involved in development activities, across releases.

with, practitioners that were involved in testing, development, architectural design, and project management activities. The method received favorable comments and no additional improvements were suggested.

It is important to point out that the lessons learned from each iteration of the method served as input for the upcoming studies. This sequence in how the studies were performed is also captured in Table I. For example, lessons from the studies in [3] and [18], which are studies done at Ericsson, helped in the empirical investigation that is presented in [16], which is a different company. Consequently, after the method is refined using insights from [3] and [18] and [16], the method is then applied in a different case company that is presented in [19]. Therefore, the usefulness of the data collected is not based on the viewpoints of the companies that suggested changes in the data collection. This is a potential drawback of the approach followed in developing the method. However, the insights used to refine the method we gathered from experts in industry that have extensive experience with managing, participating in, or analyzing data from, GSD projects.

5.2. Lessons From Applying the Latest Version of the Method

The following are the lessons learned from applying the latest update of the method in the most recent study presented in [19]. The lessons for each phase are indicated by the text in italics.

Phase 1: *A clear characterization of the GSD context provides critical support for the subsequent phases.* The collection and analysis of data need to be appropriate for the studied GSD context to

DOI: 10.1002/smr

make correct inferences. Hence, Phase 1 is critical for the method as it sets the course for applying the method, and has implications on the next phases.

Using multiple sources of data increases accuracy of information gathered about the GSD context. Based on our experience, conducting interviews, formal and informal meetings, and reviewing company and product documents helps in understanding the context of the study. Using data from more than one source, e.g., meetings and documents, enabled data triangulation.

Phase 2: *Some object-oriented measures that quantify complexity and size properties may capture redundant information.* Similar observations have been made in other studies, e.g., [50, 51]. Thus an effort should be made to ensure that the measures used in the investigation are capturing unique information about the source code. This can be done by performing a principal component analysis as done by Briand et al. [50] or by checking if the trend across releases differs between the measures.

There is a potential link between patterns in the object-oriented measure RFC (response for a class) [34] and refactoring activities across releases. Changes across releases in the RFC measure were attributed to refactoring activities in the latest empirical case study in which the method was applied. Thus the measure may be useful for identifying releases that need refactoring activities, or to retrospectively investigate refactoring activities. However, this needs further investigation.

Involvement of experts at the company at which the method is applied is important for ensuring data accuracy, especially for commit data. Commit data can have user-names that are not traceable to specific developers or teams. In such cases they may belong to automated activities. Another issue is that the developers who make commits, and subsequently appear in the commit log, may not necessarily be the actual developers who implemented a change in the source code. Such issues can be verified by practitioners. However, this emphasizes the need to clean the data, and get assistance from company experts who are aware of the product's evolution to ensure that the data is accurate and representative of the development setting.

Using data from both maintenance and evolutionary releases facilitates the investigation of quality of evolving systems. It is challenging to get access to source code for many releases in industry due to intellectual property and other related concerns. However, a large pool of data enables the construction of a longitudinal view that shows changes in defects and source code measures over a long period of time. This helps to better understand the link between changes in various source code measures and changes in defects across multiple releases. Thus, the suggestion is to gather measures for both evolutionary and maintenance releases if possible.

Phase 3: *Bar and line charts are efficient means of visualizing changes across releases in data linked to quality.* For the visualizations in Phase 3, we have used heat maps, bar and line charts, and also experimented with Kiviat diagrams. Our experience shows that the simple bar and line charts were intuitively understood by practitioners and, most importantly, required the least time and effort to interpret.

It is more convenient to have diagrams placed together, and normalized by the releases (as in Figure 5). We have used individual diagrams, but they were difficult to follow, and also see the trends in relation to the development settings. The visualization example shown in Figure 5 was the most helpful in our experience.

Phase 4: *Focus group meetings were effective means of discussing and getting insights from practitioners about changes across releases in data linked to quality.* Focus group meetings fostered open and deep discussions with practitioners (see also [52]). In comparison with interviews, focus group meetings helped getting richer information about (a) the changes in measures linked to quality, and (b) the link between the changes in the measures and the corresponding GSD-related events. Focus group meetings also created a platform for practitioners to help each other recollect and discuss confounding factors from past releases.

Overall, during each phase, an effort should be made to involve experts who are knowledgeable about the evolution of the product. They can provide explanations and/or identify appropriate approaches for rectifying issues with erroneous, missing, or conflicting data. There can also be uncertainties and inconsistencies in practitioners' reflections of past events or development settings.

DOI: 10.1002/smr

Therefore, an effort should also be made to use other sources of information (e.g., documentation) to perform data triangulation and thus increase accuracy and validity of findings [53].

6. IMPLICATION TO RESEARCH AND PRACTICE

Common quality models and standards, such as ISO 20510 [15], are quite general. They do not take into consideration the development settings of GSD projects, which are increasingly common in industry. The proposed method takes into account the specific characteristics of GSD projects that might affect software quality and evolution.

The method is for investigating changes in quality using defects across releases. This investigation can be done using data collected at a high level of the source code, i.e., the release level or at a lower level, e.g., a particular component. However, inferences that are made on one level should not be transferred to the other levels [54]. For example, inferences that are made based on data that is aggregated at the release level should not be directly transferred to the component level. Each level requires a separate investigation that is specific to that level.

The implications of the proposed method for research are threefold. Studies on quality, using defects, for software developed in GSD contexts need to take into account the peculiarities of the context. Measures that capture the evolution of the product and have a potential influence on defects should be included in the analysis of evolving systems. Data collection and analysis need to be adapted to the specific configuration of the studied GSD context. Lastly, effort should be made to involve practitioners during various phases of the investigation to verify the accuracy of the data used.

Taking into account the development setting, and clearly characterizing the GSD context, makes it easier to compare findings between studies in GSD. It also helps with understanding the extent of generalizability of aggregated findings across studies [23, 24], which would be useful information for practitioners.

It is important to note that the source code measures identified by Jabangwe et al. [36] are those that show some consistency with their relationship with defects across multiple empirical studies. However, the correlation that the measures may have with defects does not imply cause-effect. Therefore, too much reliance should not be placed on the measures. Hence, in the method we highlight the importance of collecting and triangulating multiple sources of data, and then visualizing each data in a separate figure, and then placing the figures side by side (e.g., Figure 5). By "triangulation" we mean using different sources of data and not necessarily collecting more of the same data. The aim is to analyze a problem from different angles using more than one type of data as encouraged in the guidelines for conducting empirical studies, e.g., case studies [53], with the goal of strengthening validity of the overall findings. This notion is adopted in our method. Furthermore, it is important to do a deeper analysis of the actual code to decide where, for example, refactoring seems to be an appropriate action. However, this is beyond the scope of the method proposed. The purpose with the proposed use of multiple sources of data as well as the visualization approach is to discourage relying on one dataset. This then mitigates the possibility of making incorrect inferences.

From the practitioners' point of view, their primary interest is on the results of the method. This is with regards to the link between software quality and GSD contexts, as well as the confounding factors. The confounding factors are particularly valuable information for practitioners. This information can be used to make informed decisions when planning or running projects. For example, the empirical findings by Jabangwe et al. [18] can be used to identify confounding factors that can have a negative implication on quality in certain GSD contexts. This information can then be used to plan mitigation strategies when engaging in similar circumstances.

DOI: 10.1002/smr

7. CONCLUSION

The method, GSD-QuID, proposed in this paper is intending to help with investigating quality, using defects, for object-oriented software systems that evolve in GSD contexts. The method was improved and refined incrementally using lessons learned from applying it in multiple GSD empirical studies. The method can help to ensure consistency in the manner in which GSD studies that focus on software quality are designed and performed. This will make it easier to compare and aggregate GSD studies, and help building a stronger knowledge base.

The method takes into account GSD contexts in terms of the project arrangements and changes in development setting as a result of GSD events, such as offshoring. This is important because the context can have varying implications on quality.

The method includes analyzing object-oriented measures that can be linked to defects, as well as data that captures product modification, features added and commit activities, which influence defects across releases. The method also involves practitioners during key phases of data collection and analysis. Their knowledge about the evolution of the product helps to ensure data accuracy and to better understand confounding factors for software quality in the studied context, and thereby increasing the validity of the findings.

A possible direction for future work is identifying areas of improvement in the method. This can be done by applying the method in GSD contexts with characteristics that differ from the cases in which it has been applied thus far.

The following were the overarching goals when developing the GSD-QuID method. The aim was to formulate a method for investigating how quality changes across releases for systems developed in GSD contexts. The goal with the method was to provide a useful tool for investigating the implications on quality of GSD events that occur whilst a software product evolves. The current version of the method was constructed by following a method engineering approach using insights from a systematic literature review presented in [36] and four GSD empirical studies in [3, 16, 18, 19]. Thus, the method is an integration and synthesis of lessons learned. However, the method is yet to go through a rigorous evaluation. Therefore, as future work, the method needs to be evaluated against its ease of use, usefulness and how well it helps with achieving the aforementioned goals. In order to reduce bias, at least one of the evaluations should be performed by one or more individuals that were not involved in the development of the method.

From conception until the current version, the GSD-QuID method has only been applied in GSD contexts. Thus, the method is based on a specific approach that we followed when conducting empirical studies for software developed in GSD contexts. It can be used for projects with multiple vendors or multiple sites, or it can be applied in a case were there are large-scale projects with multiple teams, were the focus is on comparing the results from different teams. However, it is possible that the method may be useful in other contexts. The method can be tailored and modified. But because the method is yet to be applied in non-GSD contexts, we refrained from making an assertion that the method can be used in contexts other than those found in GSD projects. Therefore, as part of future work, the usefulness of the method in non-GSD contexts warrants further investigation.

ACKNOWLEDGMENTS

This research is partly funded by the Swedish Knowledge Foundation under the grant 20120200 (2013-2016) and Ericsson Software Research.

REFERENCES

1. J. Verner, O. Brereton, B. Kitchenham, M. Turner, and M. Niazi, “Systematic literature reviews in global software development: A tertiary study,” in *Proceedings of the 16th International Conference on Evaluation Assessment in Software Engineering*, pp. 2–11, 2012.

DOI: 10.1002/smr

2. D. Šmite, C. Wohlin, T. Gorschek, and R. Feldt, "Empirical evidence in global software engineering: A systematic review," *Empirical Software Engineering*, vol. 15, no. 1, pp. 91–118, 2010.
3. R. Jabangwe and D. Šmite, "An exploratory study of software evolution and quality: Before, during and after a transfer," in *Proceedings of the 7th IEEE International Conference on Global Software Engineering*, pp. 41–50, 2012.
4. D. Šmite, "Distributed project management," in *Software project management in a changing world* (G. Ruhe and C. Wohlin, eds.), pp. 301–320, Springer Berlin Heidelberg, 2014.
5. E. Kocaguneli, T. Zimmermann, C. Bird, N. Nagappan, and T. Menzies, "Distributed development considered harmful?," in *Proceedings of the 35th International Conference on Software Engineering*, pp. 882–890, 2013.
6. C. Bird and N. Nagappan, "Who? where? what? examining distributed development in two large open source projects," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pp. 237–246, 2012.
7. M. Cataldo and J. D. Herbsleb, "Factors leading to integration failures in global feature-oriented development: An empirical analysis," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 161–170, 2011.
8. D. Šmite, C. Wohlin, Z. Galviņa, and R. Prikladnicki, "An empirically based terminology and taxonomy for global software engineering," *Empirical Software Engineering*, vol. 19, no. 1, pp. 105–153, 2014.
9. L. Yu and A. Mishra, "An empirical study of lehmans law on software quality evolution," *International Journal of Software and Informatics*, vol. 7, no. 3, pp. 469–481, 2013.
10. N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 521–530, 2008.
11. F. Seth, E. Mustonen-Ollila, and O. Taipale, "The influence of management on software product quality: An empirical study in software developing companies," in *Systems, Software and Services Process Improvement* (B. Barafort, R. OConnor, A. Poth, and R. Messnarz, eds.), vol. 425 of *Communications in Computer and Information Science*, pp. 147–158, Springer Berlin Heidelberg, 2014.
12. N. Ramasubbu, M. Cataldo, R. K. Balan, and J. D. Herbsleb, "Configuring global software teams: A multi-company analysis of project productivity, quality, and profits," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 261–270, 2011.
13. J. A. McCall, P. K. Richards, and G. F. Walters, "Factors in software quality. volume I. concepts and definitions of software quality.," tech. rep., General Electric Co Sunnyvale, Nov 1977.
14. B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod, and M. J. Merrit, *Characteristics of software quality*, vol. 1. Amsterdam, The Netherlands: North-Holland publishing company, 1978.
15. ISO/IEC-25010, "Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models." International organization for standardization, 2010.
16. R. Jabangwe, J. Börstler, and K. Petersen, "Handover of managerial responsibilities in global software development: a case study of source code evolution and quality," *Software Quality Journal*, pp. 1–28, Online in August 2014.
17. R. J. Mayer, J. W. Crump, R. Fernandes, A. Keen, and M. K. Painter, "Information integration for concurrent engineering (iice) compendium of methods report," tech. rep., DTIC Document, 1995.
18. R. Jabangwe, K. Petersen, and D. Šmite, "Visualization of defect inflow and resolution cycles: Before, during and after transfer," in *Proceedings of the 20th Asia-Pacific Software Engineering Conference*, vol. 1, pp. 289–298, 2013.
19. R. Jabangwe, D. Šmite, and E. Hessbo, "Distributed software development in an offshore outsourcing project: A case study of source code evolution and quality," *Information and Software Technology*, vol. 72, pp. 125–136, April 2016.
20. M. Cataldo and S. Nambiar, "On the relationship between process maturity and geographic distribution: an empirical analysis of their impact on software quality," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pp. 101–110, 2009.
21. C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Does distributed development affect software quality? An empirical case study of Windows Vista," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 85–93, 2009.
22. F. da Silva, C. Costa, A. Frana, and R. Prikladinicki, "Challenges and solutions in distributed software development project management: A systematic literature review," in *Proceedings of the 5th International Conference on Global Software Engineering*, pp. 87–96, 2010.
23. C. Wohlin, "Writing for synthesis of evidence in empirical software engineering," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 1–4, 2014.
24. K. Petersen and C. Wohlin, "Context in industrial software engineering research," in *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 401–404, 2009.
25. P. Runeson, M. C. Ohlsson, and C. Wohlin, "A classification scheme for studies on fault-prone components," in *Product Focused Software Process Improvement* (F. Bomarius and S. Komi-Sirviö, eds.), vol. 2188 of *Lecture notes in computer science*, pp. 341–355, Springer Berlin Heidelberg, 2001.
26. V. R. Basili, "Software modeling and measurement: The goal/question/metric paradigm," tech. rep., College Park, MD, USA, 1992.
27. J. Münch, O. Armbrust, M. Kowalczyk, and M. Soto, "Process improvement," in *Software Process Definition and Management*, The Fraunhofer IESE Series on Software and Systems Engineering, pp. 139–176, Springer Berlin Heidelberg, 2012.
28. V. R. Basili, "The experience factory and its relationship to other improvement paradigms," in *Proceedings of the 4th European Software Engineering Conference*, pp. 68–83, Springer-Verlag, 1993.
29. M. Kuhrmann, D. Méndez Fernández, and M. Tiessler, "A mapping study on the feasibility of method engineering," *Journal of Software: Evolution and Process*, 2014.
30. A. H. ter Hofstede and T. Verhoef, "On the feasibility of situational method engineering," *Information Systems*, vol. 22, no. 6-7, pp. 401–422, 1997.

DOI: 10.1002/smr

31. E. Ó. Conchúir, P. J. Ågerfalk, H. H. Olsson, and B. Fitzgerald, "Global software development: where are the benefits?," *Communications of the ACM*, vol. 52, no. 8, pp. 127–131, 2009.
32. ISO/IEC/IEEE-24765, "Systems and software engineering – Vocabulary." International organization for standardization, 2010.
33. D. Šmite and C. Wohlin, "Lessons learned from transferring software products to india," *Journal of Software: Evolution and Process*, vol. 24, pp. 605–623, October 2012.
34. S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476–493, June 1994.
35. J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, pp. 4–17, January 2002.
36. R. Jabangwe, J. Börstler, D. Šmite, and C. Wohlin, "Empirical evidence on the link between object-oriented measures and external quality attributes: A systematic literature review," *Empirical Software Engineering*, pp. 1–54, Online in March 2014.
37. A. Meneely and L. Williams, "Secure open source collaboration: an empirical study of linus' law," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pp. 453–462, 2009.
38. C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 4–14, 2011.
39. D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov, "Dual ecological measures of focus in software development," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 452–461, 2013.
40. S. Bassil and R. K. Keller, "Software visualization tools: Survey and analysis," in *Proceedings of the 9th International Workshop on Program Comprehension*, pp. 7–17, 2001.
41. S. Diehl, *Software visualization – Visualizing the structure, behaviour, and evolution of software*. New York, USA: Springer-Verlag Berlin Heidelberg, 2007.
42. T. Mens and S. Demeyer, *Software evolution*. New York, USA: Springer-Verlag Berlin Heidelberg, 2008.
43. B. Shneiderman, "Tree visualization with tree-maps: 2-d space-filling approach," *ACM Transactions on Graphics*, vol. 11, no. 1, pp. 92–99, 1992.
44. K. Petersen and C. Wohlin, "Measuring the flow in lean software development," *Software: Practice and Experience*, vol. 41, pp. 975–996, August 2011.
45. L. Cagnazzo and P. Taticchi, "Six sigma: A literature review analysis," in *Proceedings of the International Conference on E-activities and Information Security and Privacy*, pp. 29–34, 2009.
46. M. M. Lehman and L. A. Belady, *Program evolution: Processes of software change*. CA, USA: Academic Press Professional, Inc., 1985.
47. R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A case study on the impact of refactoring on quality and productivity in an agile team," in *Balancing Agility and Formalism in Software Engineering* (B. Meyer, J. Nawrocki, and B. Walter, eds.), vol. 5082 of *Lecture notes in computer science*, pp. 252–266, Springer Berlin Heidelberg, 2008.
48. T. Dybå, D. I. Sjøberg, and D. S. Cruzes, "What works for whom, where, when, and why?: On the role of context in empirical software engineering," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, (New York, NY, USA), pp. 19–28, ACM, 2012.
49. M. Lorenz and J. Kidd, *Object-oriented software metrics: A practical guide*. New Jersey, USA: Prentice-Hall, 1994.
50. L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationship between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, pp. 245–273, May 2000.
51. G. Pai and J. Bechta Dugan, "Empirical analysis of software fault content and fault proneness using bayesian methods," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 675–686, 2007.
52. C. Robson, *Real world research*. West Sussex, UK: John Wiley & Sons, 2nd ed., 2011.
53. P. Runeson, M. Höst, A. Rainer, and B. Regnell, *Case study research in software engineering*. New Jersey, USA: John Wiley & Sons, 2012.
54. D. Posnett, V. Filkov, and P. Devanbu, "Ecological inference in empirical software engineering," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 362–371, 2011.

DOI: 10.1002/smr