Continual Monitoring of Code Quality

Corporate Research and Technologies
Siemens Information Systems Ltd.
No. 84, Keonics Electronics City,
Bangalore, India
{chaitanya.kothapalli, ganesh.samarthyam, himanshu.singh, radhika.dv, rajaram.thiagarajan, k.ravikanth, shrinath.gupta, kiron.rao}@siemens.com

ABSTRACT
CQMM (Code Quality Monitoring Method) is a means for systematically monitoring and improving code level quality of a system during development. It employs goal directed monitoring using quality models and static code analysis tools. In this paper, we present the CQMM method, learnings gathered through pilot studies, and changes needed for its large scale adoption within our organization. This exercise was an important step towards evolving an organization wide common minimum baseline for code-centric quality. Initial results indicate that the process helps in exposing important code-centric issues, besides sensitizing developers to coding practices. We also demonstrate the usefulness of the approach by tracking code level issues on select open source projects.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement; D.2.8 [Software Engineering]: Metrics

General Terms
Measurement, Management

Keywords
Software quality assessment, static verification, continuous measurement, software metrics, bug detection

1. INTRODUCTION AND MOTIVATION

Managing software quality is a necessity as systems have become increasingly large, complex, and long-lived. ISO 9126 [1] defines ‘software quality’ as the totality of characteristics of a software product that bear on its ability to satisfy stated and implied needs. Software quality may also be classified as either external or internal. External software quality covers aspects directly perceived by end users (e.g., usability), while internal software quality refers to aspects only revealed by examining design and code (e.g., maintainability). In this paper, we focus on internal software quality, aka ‘code quality.’ Code quality is the capability of source code to satisfy the stated and implied needs for the current software project [2]. Also, a reasonable connection between internal and external software quality is empirically supported [23, 22, 17].

Lehman observes that the quality of a system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes [19]. Containing this problem, referred to as ‘code decay’ [9], calls for the adoption of a systematic monitoring for timely identification of important quality indicators.

As stated in [32], “static analysis tools are complementary to other fault-detection techniques for the economic production of a high-quality software product.” However, their use needs to be moderated for best results: not all quality issues they report may be relevant or equally important in a given context. For instance, when ‘portability’ is not a concern, issues affecting the same may be clearly ignored. Effectively harnessing the power of available tools for monitoring of key indicators requires a rational basis. The Code Quality Monitoring Method (CQMM) defines a systematic and goal directed approach to this end [28]. Its adoption facilitates the identification and planning of corrective actions, such as prioritizing bug fixes, refactoring, strengthening peer-review, etc. Since quality cannot be added as an after thought (Réel [29]), its adoption serves as proactive means for containing code-centric quality issues.

There is considerable literature on the use of static analysis tools. On the contrary, there is little work related to the systematic use of these tools for monitoring and controlling software quality especially in an organizational context. In what follows, we describe our experience in piloting and implementing CQMM within our organization. To illustrate its use we also present results collected from open source projects. The rest of the paper is organized as follows. Sec. 2 briefly covers related work; Secs. 3, 4, and 5, respectively present the CQMM process, results from pilot studies, and adoption and deployment at the organizational level. Sec. 6 presents trends of code quality issues on select open source projects.

2. RELATED WORK

Software metrics are widely used for evaluating software quality [11, 21, 16]. For example, QMOOD [5] is a hierarchical model for Object-Oriented design quality assessment; it uses OO design metrics (e.g., CAM - ‘Cohesion Among Methods of Class’) for evaluating high level design quality attributes (e.g., reusability). There is reasonable empirical evidence to demonstrate that many metrics serve as useful quality indicators (e.g., [6, 20]). Static code analysis
(whether manual or automatic) is another important means to investigate or evaluate quality of source code [10, 13].

Many monitoring approaches integrate static analysis tool results, metrics, results etc. into the form of a dashboard, for example, Sonar\(^1\). Software project telemetry [15] provides an effective approach to automated metrics collection and emphasizes in-process control. A similar approach and tool (Altheia Core) is presented in [12]. ConQAT [7] integrates measurement tools (it also has support for creating new set of checks); it is designed for integration into the build process so that a set of project-specific quality criteria can be controlled on a continuous basis. Another system for source code-based software quality assessment and monitoring (for metrics, coding problems, bug numbers, test coverage information etc) is presented in [4]. Most available monitoring approaches and tools only collect and present data, but only a few are goal directed. Additionally, only a very few consider integration aspects into an organizational context. The CQMM approach is similar to the method presented by [18]; this process is supported by Software Analysis Toolkit (SAT). However, CQMM method is more flexible in that it can be adapted to the needs of a product, project, or component.

3. CQMM IN BRIEF

CQMM is an approach for systematically assessing and improving the code quality of a software product during its development life cycle. The process and associated tool support are presented in [28], and its detailed description is available for internal use\(^2\). This paper describes our experiences in adopting and applying CQMM to development projects within the organization.

CQMM is based on Evaluation Method for Internal Software Quality (EMISQ) [26]. EMISQ is an expert based method for evaluating code quality using static code analysis tools in conjunction with a suitable ‘quality model’ [8]. Fig. 1 provides the quality model used by CQMM for defining a mapping between rules supported by static analysis tools and the quality factors they affect. For example, the PC-Lint violation 1506: Call to virtual function ‘Symbol’ within a constructor or destructor affects both reliability and maintainability. Similarly, the FxCop violation CA1062: Validate arguments of public methods affects reliability. Nature, severity and number of violations are indicative of the extent of potential impact on quality\(^3\). As noted in [24], such “automated code inspection faults can be used as efficient predictors of field failures.”

The EMISQ process allows for the use of any suitable quality model for internal software quality. This method of assessing code allows for suitably tailoring quality factors for project specific requirements [25]. For example, an embedded C project can mandate the use of a technical quality model [27] selecting rules and advisories of MISRA C [3].

CQMM builds upon the essentials of EMISQ: goal directed measurement of quality indicators using static analysis tools, and application of quality models. CQMM is meant for continual monitoring for improvement whereas EMISQ is a need based, expert assessment. CQMM is integrated into the development process with the help of necessary tool support and, unlike EMISQ, is owned and managed by the development team.

3.1 CQMM process overview

The process comprises 11 activities grouped into three phases. The Setup and Tailor phase defines and integrates CQMM as per the requirements of a project. The Measure and Enhance phase, on a regular basis, collects and analyzes results of static analysis tools for monitoring and improvement. The Adjust and Control phase is carried out on a need basis to fine-tune the use of CQMM. The process clearly identifies the roles and responsibilities of the various stakeholders (project manager, quality assurance manager, etc.); we omit them here due to space constraints. Fig. 2 outlines the phases of CQMM, associated activities, and resulting output.

3.1.1 Setup and Tailor phase

Adapt CQMM into the development process of a project as per the following steps.

1. Analyze development environment to understand the static analysis tools used, quality assurance activities and reporting mechanisms already in place.

2. Define quality goals desired of the system (e.g., ease of maintenance, high robustness, etc.).

3. Define quality model in accordance with the identified goals and needs of stakeholders (e.g., EMISQ quality model in Fig. 1).

4. Define and tailor monitoring approach based on project needs (from one of trend, benchmark, and risk based approaches).

5. Tailor quality model and scope to project needs and validate the selected measures.

6. Integrate into the development process steps, such as, build process, release management, clearance procedures, etc.

This phase also establishes the baseline for the current quality, typically through an ‘EMISQ baseline review’.

3.1.2 Measure and Enhance phase

Monitor code quality regularly by repeating the following steps at predefined semantic points (e.g., specific releases) or time intervals (e.g., weekly).

7. Measure and classify identified metrics or indicators using static analysis tools.

8. Analyze results and aggregate them for presentation (e.g., as a trend).


3.1.3 Adjust and Control phase

Suitably modify the deployed process, if needed.

10. Adjust quality monitoring to bridge any perceived gaps in the CQMM implementation (e.g., adding or removing metrics, fine-tuning the monitoring approach, etc.).

11. Audit quality monitoring by an external quality assessor to review its deployment and practice within the project.

---

\(^{1}\)http://www.sonarsource.org/

\(^{2}\)The detailed CQMM handbook is around 100 pages.

\(^{3}\)Since static analysis tool violations are used as the basis for evaluating quality, the term ‘metrics’ is used to refer to static analysis tool violations in EMISQ method and in this paper.
3.2 CQMM tool support overview

Tool support for CQMM, as presented in [28], are Software Project Quality Reporter (SPQR) [14] and Software Quality Monitor (SQualM). SPQR (an Eclipse plug-in) was originally developed to support EMISQ reviews. It helps create and manage a mapping between quality models and rules supported by various static analysis tools. Further, it enables one to tailor a quality model, rate violations, and generate a code quality report. SQualM provides the additional support needed for monitoring, namely periodic collection of metrics and visualization of quality indicators (either as a trend or as a comparison against some benchmark).

4. CQMM INITIAL PILOTING

The pilot studies were intended to understand the feasibility of CQMM (as presented in [28]) and the changes needed for adopting it within the organization. To this end, we selected three projects, in early to mid stages of development, of which two were C# based and one used both C++ and C#. The code size considered for baseline EMISQ review was between 50 and 100 KLOC for each project. Table 1 gives details of the tools used and the selection of quality attributes (and their importance) for the pilots. All steps towards establishing CQMM, mentioned in Sec. 3.1.1, were first carried out. Subsequently, activities outlined in Sec. 3.1.2 were executed by the respective teams, on a regular basis as a part of their development process. Activities referred to in Sec. 3.1.3 were not attempted as the need for the same did not arise.

It is worthwhile to note that the CQMM process was also independently applied on two Java based projects as documented in [28]. The two independent studies conformed with CQMM process but for the tool support employed. We used only SPQR since SQualM [28] was still under development. Besides, SQualM could not be employed for C++ and C# owing to the absence of benchmark data.

Important observations from our studies are:

1. The process is suitable for use, but is heavyweight for a wide deployment within a limited time.
2. The EMISQ baseline review consumed most of the effort needed to setup CQMM within a project requiring about six person weeks of effort on about 50 KLOC.
3. The process is generic enough to be applied to different kinds of projects (cross-language and cross-domain), and the pilots corroborate this.
4. The project in its early phase benefited the most.
5. Tool support to facilitate continual monitoring was inadequate. Although SPQR supported charting, it was designed for interactive use for EMISQ reviews.

5. CQMM LARGE-SCALE DEPLOYMENT

From the experience gained through the pilots it was important to introduce certain changes to the process (as originally defined) for wider adoption of CQMM within the organization. Integrating CQMM as an integral step within development process is substantially different from applying it in isolated projects. This is because projects vary in their complexity based on factors like size of code base and development team, end quality demands, technologies used, delivery requirements, etc. To this end, the following requirements were found to be key. The CQMM process:

4 Being an expert based evaluation mechanism guided by static analysis tools, EMISQ requires reasonable manual effort in analyzing reported findings and compiling a detailed technical report.
5.1 CQMM process adaption

Adopting CQMM for a large-scale use needed certain simplifications to render it both lightweight and readily usable for most of the common quality concerns. Also, it was important that the simplification preserved the essence of CQMM. Changes were only needed for Setup and Tailor (Sec. 3.1.2), since Measure and Enhance (Sec. 3.1.1) is meant for practice by the development teams, and Adjust and Control (Sec. 3.1.3) is invoked on need basis.

5.1.1 Baseline quality model

The objective of conducting an EMISQ review in CQMM is to establish a quality baseline for the project. This option was close to impractical for large scale deployment given the effort needed for a review and the number of projects to be supported (close to 50). We therefore harnessed information collected from EMISQ assessments conducted in the past (over 30), to evolve a baseline of quality related issues and requirements common to most projects. Since the reviews spanned different kinds of real world projects (domains, sizes, languages etc.), the information extracted provided a reasonable foundation for evolving a suitable baseline. This information was further augmented with inputs drawn from important and widely recommended coding practices.

Within the CQMM framework, absence of quality is defined in terms of the extent to which violations appear. In practice, this translates to the volume and criticality of the reported violations which affect the selected quality factors. The baseline quality model evolved captures important coding violations, their impact on quality attributes, and the severity of the impact. This may be formally represented as tuples of the form:

\[
\langle \text{quality attribute}, \text{quality sub-attribute}, \text{tool identifier}, \text{rule identifier}, \text{criticality rating} \rangle
\]

Table 2 illustrates this mapping. The criticality rating is from Severity 1 to 5, with 1 being the highest. Note that one rule can affect multiple quality attributes.

Our data from past EMISQ projects shows that the quality attributes efficiency, maintainability and reliability were found to be of importance to most projects. The baseline quality model, therefore, was tailored to accommodate these quality attributes. This quality model could be suitably adjusted for projects with other quality requirements of importance. This step was left as a responsibility for the project team adopting CQMM. Unlike EMISQ reviews where several tools are used even for one language, we started with a minimal set of tools that could be easily adopted by a large set of projects. As of this writing, Extended FxCop (C#), PC-Lint (C++), FindBugs and PMD (Java) have been integrated into the baseline quality model.

5.1.2 Simplifying process steps

The creation of baseline quality model subsequently allowed for alterations that rendered the process suitable for a large-scale adoption. These modified process steps of Setup and Tailor are presented below.

1. Analyze the development environment was considerably simplified owing to the existing organization-wide, uniform processes.
2. Define quality goals was retained as defined originally.
3. Define quality model was not required as a separate process step as we went along with the EMISQ quality model (Fig. 1). Also, adopting the baseline quality model was well suited for addressing common quality needs of projects.
4. Define and tailor monitoring approach was also dropped as only trend-based monitoring was feasible given that it is the easiest to adopt and understand.
5. Tailor quality model and scope too benefited from the use of the baseline quality model. This formed the starting point for suitable customization for specific projects, based on quality goals identified in step 2, above. In our experience, we found that many projects have a set of custom rules (for example, custom FxCop rules in C#), which needed to be integrated into the CQMM framework. This involved mapping them into the EMISQ quality model and providing a default rating. This also required additional tool support (see Sec. 5.2). We provided support for integrating such rules into the CQMM toolkit and tracking them as part of the CQMM process.

<table>
<thead>
<tr>
<th>Project</th>
<th>Language</th>
<th>Tools</th>
<th>Quality attributes (and their importance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project 1</td>
<td>C#</td>
<td>Extended FxCop*, ClockSharp, Gendarme, Checkstyle</td>
<td>Maintainability High, Reliability High, Efficiency High, Portability Medium, Security Medium</td>
</tr>
<tr>
<td>Project 2</td>
<td>C#</td>
<td>Extended FxCop, ClockSharp, Gendarme, Checkstyle</td>
<td>Maintainability High, Reliability Medium, Efficiency Medium, Portability Medium, Security Medium</td>
</tr>
<tr>
<td>Project 2</td>
<td>C++</td>
<td>PC Lint, PMD, Understand</td>
<td>Maintainability Low, Reliability Medium, Efficiency Medium, Portability Medium, Security Low</td>
</tr>
<tr>
<td>Project 3</td>
<td>C#</td>
<td>Extended FxCop, ClockSharp, Gendarme, QBench</td>
<td>Maintainability High, Reliability Medium, Efficiency Medium, Portability - Medium, Security -</td>
</tr>
</tbody>
</table>

*This comprised rules provided by FxCop augmented with around 100 custom rules. The custom rules were added based on the issues not detected by FxCop but found manually during EMISQ reviews.

*Checkstyle, PMD and QBench were used for detecting code duplicates.
6. **Integrate into the development process**, in the context of a large-scale roll out, primarily comprised two steps. The first was to integrate CQMM with the build environment for seamless collection of results (Sec. 5.2.1). The second was to identify certain important release points at which clearance procedures or quality gates, based on CQMM results, were mandated. Tool support needed some tailoring for this.

Being generic, CQMM identifies documentation with almost each activity (see Fig. 2). The effort needed for some of this documentation was found to be considerable during pilot studies. This also meant that CQMM would not scale well if directly applied over a large number of projects. Our approach based on employing baseline quality model helped simplify or drop some of the steps and associated documentation (see items in gray in Fig. 2). Also, additional templates were created for certain activities.

**5.2 CQMM tool support**

As observed earlier, tool support was not adequate at the start of this exercise. We first explored the possibility of modifying SPQR to make it a command-line application. This course had to be abandoned as the tool:

- was primarily designed for assisting EMISQ based code reviews (e.g., manage quality models, rate reported findings, etc.)
- being an Eclipse plug-in, its functionality could not be easily exposed for use through the command-line
- lacked the interfaces and functionality needed for a monitoring tool: e.g., features for project specific configuration, integration with build systems, etc.

Incompatibilities of existing tool support (SPQR and SQualM) for large-scale deployment warranted the development of appropriate tool assistance. It was important that this new toolkit preserved data uniformity with SPQR. Additionally, it also needed to account for variations across development teams.

**5.2.1 CQMM toolkit**

The toolkit was designed to configure and integrate the tailored process within the development life cycle and environment of current and new projects. It uses the baseline quality model evolved for addressing large-scale deployment and supports:

- tailoring quality model, data collection, and reporting
- translating violations to a common format
- presenting trends and violation reports
- identifying newly introduced violations

Fig. 3 provides an overview of how the toolkit integrates into the development environment to regularly monitor code quality. Reporting in CQMM highlights issues based on their relevance to the quality concerns of a project, thus enabling effective management of code quality. Reports are in the form of trend charts and detailed violation reports, for use by managers and developers, respectively.

**5.2.2 Defining thresholds for violations**

The primary goal of continual monitoring is to minimize code quality issues through early detection. Although applying trend based monitoring allows for observing variations in quality issues, this alone is insufficient for determining whether code quality issues are within ‘acceptable limits.’

Use of a fixed threshold value for violations is neither correct nor practical, as violations reported have a direct bearing on the size of the code base. Also, static analysis tools are prone to varying degree of false positives. To reduce this impact, the baseline quality model attempts to provide suitable ratings to violations.

To establish a suitable upper limit on the permissible number of violations for each new version, we define a quantifiable acceptability criteria based on volume of issues and code size: change in violation count must be limited by the factor by which size of the code base changes vis-a-vis the baseline version. To make this measure meaningful the reference or baseline version selected must be one deemed to be of “good” or “acceptable” quality. When a satisfactory baseline version is not available, the a project may use a suitable stable version. This allows for a project to derive a benchmark giving due consideration to its current context and desired goals.

Below, we describe how the threshold values is calculated.

Let $L_i$ and $V_i$ ($i > 0$) represent the code size and observed violation counts for the $i^{th}$ version. The following inequality identifies the condition under which reported violations may be considered to be under control:

$$V_0/L_0 \geq (V_i/L_i)$$

Although this condition is somewhat loose, it helps establish an upper bound or threshold for each new version. Below, $T_0$ and $T_i$, define the thresholds for the baseline and $i^{th}$ measurement point:

<table>
<thead>
<tr>
<th>Quality attribute</th>
<th>Quality sub-attribute</th>
<th>Tool identifier</th>
<th>Rule identifier</th>
<th>Criticality</th>
<th>Short rating</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliability</td>
<td>Soundness</td>
<td>PC-Lint</td>
<td>571</td>
<td>Severity 1</td>
<td>Suspicious cast</td>
<td></td>
</tr>
<tr>
<td>Maintainability</td>
<td>Readability</td>
<td>PMD</td>
<td>APFF</td>
<td>Severity 2</td>
<td>Avoid protected field in final class</td>
<td></td>
</tr>
<tr>
<td>Efficiency</td>
<td>Resource utilization</td>
<td>PC-Lint</td>
<td>672</td>
<td>Severity 2</td>
<td>Possible memory leak in assignment to pointer ‘Symbol’</td>
<td></td>
</tr>
<tr>
<td>Efficiency</td>
<td>Resource utilization</td>
<td>FxCop</td>
<td>CA1804</td>
<td>Severity 3</td>
<td>Remove unused locals</td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td>Soundness</td>
<td>FindBugs</td>
<td>SMET</td>
<td>Severity 3</td>
<td>Certain swing methods should only be invoked from the Swing event thread</td>
<td></td>
</tr>
</tbody>
</table>

11 Change in code size may be effective LOC, number of IL instructions etc., but must be uniformly applied across versions for a project.
The above formulation is sufficiently general for use by all projects. A specific project can choose to tighten or loosen the acceptable limit (e.g., by a factor of 10%). However, under certain circumstances – for example, when code is refactored – a new baseline may need to be established for the threshold.

Lastly, projects with historical data are better positioned to define more meaningful thresholds. A means to define absolute thresholds is also supported by the toolkit for such projects.

6. RESULTS AND DISCUSSION

As of date, we have deployed CQMM into around 25 projects. It is somewhat early to provide a comprehensive analysis: although the roll out was initiated around eight months ago, the time frame for its integration was left to the convenience of individual projects. Based on feedback, the following observations regarding its adoption are noteworthy:

- regular monitoring is helping raise the awareness of code quality issues among developers
- integration into the development process is helping expose code quality issues
- the use of static code analysis tools as part of the development process is streamlined

In what follows, we present case studies which illustrate how CQMM helps in exposing issues related to code quality as software evolves.

6.1 CQMM on enterprise projects

Projects adopting CQMM in the early stages of development benefit most. Here we illustrate the results for one such development project. Fig. 4 provides a trend of violations corresponding to three quality attributes which are being regularly monitored. For these versions, the code size consistently increased from approximately 26 to 33 eKLOC as new features were being added.

6.2 CQMM on open source projects

We selected four open source projects differing in code base size, language used, and maturity level (see Table 3) in order to understand if violation trends reported by the tailored CQMM process (Sec. 5.1) can serve as indicators of changes in quality. We collected violations for each project across multiple versions, as outlined in Sec. 3.1.2 (Measure and Enhance phase) using the baseline quality model (Sec. 5.1.1). Only issues rated at Severity I and II, affecting the attributes reliability, efficiency, and maintainability were considered, and the results are presented in Fig. 5. For thresholding we used the first measurement point to be the baseline. Since the study simulates continual monitoring, but in retrospect, it is difficult to conclusively show the benefits of adopting this process. It is, however, possible to draw correlations between violation trends and indicators affecting code quality.
Figure 5: CQMM based trends observed on select open source projects

<table>
<thead>
<tr>
<th>Project</th>
<th>Short description</th>
<th>Language</th>
<th>Analysis tools used</th>
<th>No. of versions tracked</th>
<th>Size unit</th>
<th>Size (initial - final)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache HTTPD</td>
<td>Apache HTTP server</td>
<td>C</td>
<td>PC-Lint</td>
<td>9 (2.0.36 to 2.0.55)</td>
<td>Effective LOC</td>
<td>87109 - 96909</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>Charting library</td>
<td>Java</td>
<td>PMD</td>
<td>11 (0.5.6 to 1.0.6)</td>
<td>Effective LOC</td>
<td>14830 - 217338</td>
</tr>
<tr>
<td>Gendarme</td>
<td>Static code analyzer</td>
<td>C#</td>
<td>Extended FxCop</td>
<td>8 (2.0 to 2.6)</td>
<td>IL Count</td>
<td>30113 - 55879</td>
</tr>
<tr>
<td>BugNET</td>
<td>Issue tracker</td>
<td>C#</td>
<td>Extended FxCop</td>
<td>11 (0.6.4 to 0.8.193)</td>
<td>Source LOC</td>
<td>3543 - 9990</td>
</tr>
</tbody>
</table>
• Apache HTTPD\textsuperscript{12}, the C-based web server, is known to be a mature and stable system. Violations were collected over a window of 3.5 years. During this period, both code size and violation count grew by around 10\%, in lock-step.

The large number of violations reported against Apache HTTPD are primarily due to the PC-Lint rule 534 (\textit{Ignoring return value of function \text{"String\"}}) - 1668 out of 2521 and 1916 out of 2809, respectively, in the first and the last measured versions. Although this issue may be common with legacy C code, the study [31] predicts up to 30\% of such missing checks in Apache to be likely bugs. On brief analysis, we found two instances of this issue (file \texttt{rotatelogs.c}) in version 2.0.36 (the first version in our study):

\begin{verbatim}
apr_file trunc (sLogFD, 0); // PC-Lint 534 !
if (apr_file write (sLogFD, errbuf, knWrite) != APR_SUCCESS) {
    fprintf(stderr, "Error writing to the file %s\n", buf2);
    exit(2);
}
\end{verbatim}

This violation was reported as an issue (bug id 45084) only recently, in version 2.2.8.

More true positives may be identified by correlating violations with change history records. For instance, we found three true positives of PC-Lint 534 involving calls to the function \texttt{ap_run_pre_connection(\ldots)} in \texttt{mod_proxy_ftp} and \texttt{mod_proxy_http} which were fixed in version 2.0.45.

• JFreeChart\textsuperscript{13}, the Java-based charting application, was measured for violations between 2001 and 2010. During this period its code base grew considerably, from 15 KLOC (version 0.5) to 217 KLOC (version 1.0.13). Although early releases show a controlled but steady rise in issues, intermediate ones record a lower growth in issues, even as the code base increased by an order of magnitude. Following the initial versions, violations remain well below the threshold curve, but a brief analysis reveals several true positives.

For example, the code snippet below (ColumnArrange-ment.java), from the very last version, is one of the seven instances of the PMD violation EIFS: EmptyIfStmt, signaling incomplete code.

\begin{verbatim}
if (this.hori zontalAlignment != HorizontalAlignment.LEFT) {
    for (int i = 0; i < blocks.size(); i++) {
        // Block b = (Block) blocks.get(i);
        if (this.hori zontalAlignment == HorizontalAlignment.CENTER) {
            //TODO: shift block right by half
        } else if (this.hori zontalAlignment == HorizontalAlignment.RIGHT) {
            //TODO: shift block over to right
        }
    }
}
\end{verbatim}

Also, in the last version, there are 197 instances of the rule NPAC: NPathComplexity indicating a high number of acyclic paths in methods compared to a single instance in the first version. Many new violations were introduced over time: for example, there are 14 instances of empty catch blocks in the last version, but none in the first.

The trend for JFreeChart, in Fig. 5, was generated using only PMD. We also collected results for JFreeChart using both PMD and FindBugs, and the results are presented in Fig. 6. There is no change in threshold since FindBugs did not reveal any issues (being tracked by the baseline quality model) for the first version. However, many new violations are exposed in the versions 0.9.12 and 0.9.20 by FindBugs. Analysis, shows that most of the new issues reported by FindBugs are false positives due to the rules:

\begin{itemize}
  \item \texttt{HE.EQUALS_USE_HASHCODE} : Class defines equals() and uses Object.hashCode().
  \item \texttt{SE_NO_SERIALVERSIONID} : Class is Serializable, but doesn\'t define serialVersionUID.
\end{itemize}

Such false positives can be re-rated depending on project specific context and experience. The deployed process (Sec. 5.1) allows for such changes.

• Gendarme\textsuperscript{14}, the C#-based static analysis tool, was measured for violations from version 2.0 to 2.6 comprising both beta and release editions of the software.\textsuperscript{15} The violation trend shows considerable swings with the peaks occurring at the beta-releases.

Several rules present higher counts in the beta versions, and could be attributed to changes and enhancements which the software underwent. For instance, the beta version of 2.6 contained around 418 instances for the rule \textit{Non constant static fields should not be visible}, compared to 40 in the release version.

Notwithstanding the false positives, several issues of value be found. For example, the number of instances of the issue CA2201: Do not raise reserved exception types grew from two to seven between the release versions 2.4 and 2.6. A code snippet containing this violation, which was later fixed in revision 155451, appears below:

\begin{verbatim}
// FIX: Exception changed to ArgumentException
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6.png}
\caption{JFreeChart trend - PMD and FindBugs}
\end{figure}

\textsuperscript{12}http://httpd.apache.org/
\textsuperscript{13}http://www.jfree.org/jfreechart/
\textsuperscript{14}http://www.mono-project.com/Gendarme
\textsuperscript{15}We could only access the binaries (but not the sources) for these versions, we used the number of IL instructions to calculate code size.
BugNET\textsuperscript{16}, the C#-based defect tracking tool, was tracked starting with one of its early releases, and over this period its code base grew threefold. Although the violation count has grown over time, the trend chart shows that it is more or less under the threshold curve. Analysis shows violations to be varying in nature with most being true positives. For example, the count of the FxCop violation CA1031: Do not catch general exception types, were 13 and 62 in the first and the last versions measured. A sample code snippet (ManageUsers.ascx.cs) containing this violation is given below:

```csharp
try {
    objUser .ChangePassword
    (objUser .GetPassword() , NewPassword .Text);
    lblMessage .Visible = true;
    lblMessage .Text = "Password changed successfully" ;
}
catch (Exception ex) {
    // TODO : Log this error
    lblMessage .Visible = true;
    lblMessage .Text = "The password could not be changed , please verify that the password is 7 characters or more" ;
}
```

This catch block would need to be corrected by catching a custom exception instead.

Our brief study on open source projects attempts to corroborate that monitoring internal code quality indeed helps expose possible signs of degradation. In fact, analysis of the source code, which was driven by rise in violations, was capable of identifying possible (and real) bugs, as also potentially harmful coding practices. Monitoring was regulated by our baseline quality model to render it suitable to common concerns of software projects, even when they appear to share little. We used the initial state of the projects to derive dynamic thresholds due to the absence of project specific information. Nevertheless, this proved useful in arriving at “reasonable control limits” against which violation trends could be compared.

7. CONCLUSIONS AND FUTURE WORK

CQMM is envisioned as a generic process to monitor and control code quality issues. In this paper, we have presented our experience in applying this method within our organization. Our pilot studies revealed that although projects could benefit from its use, effort to set it up (within a project) was considerable. Since our goal was to roll out CQMM for use within a large number of development projects (as an integral process step), it was necessary to render the process lightweight without sacrificing the original vision. Also, adequate tool support was needed to facilitate its integration into the development life cycle.

To simplify the adoption of CQMM we developed a baseline quality model to capture important code quality issues common to most projects. This baseline model was mainly derived from the data gathered and experience gained from around 30 EMISQ assessments conducted earlier by us. It is important to note that the baseline quality model captures the “common quality denominator” for projects. This model is intended to form a starting point which, over time, needs to be suitably adapted to better address project specific needs and characteristics. Our approach helps amortize the effort for tailoring during the set-up phase over the measure and enhance phase. This preserves the original vision while simplifying its adoption. Internalizing the process, however, requires both time and effort of the development team.

CQMM is a means for realizing improved control on code quality issues. To substantiate this, we presented results from one of the projects adopting it. Currently, around 25 projects are in the process of integrating CQMM into their development process. It is somewhat early to present results of this experiment as adoption and internalization of a new process requires time. Therefore, to demonstrate its usefulness, we simulated measurement via CQMM on four open source projects of varying nature and maturity. Our study shows that by monitoring trends it was possible to identify important issues affecting internal quality; a brief analysis was sufficient to reveal several avoidable coding practices and potential as well as real (reported) bugs.

Use of quality models to harness the power of static code analyzers is central to CQMM. Since the responsibility for identifying violations lies with the static code analyzers used, we foresee extending tool support to reduce both false positives and false negatives to be vital. Going forward, we also plan to enhance the process by incorporating learnings drawn from its practice. Here, we anticipate improvement opportunities by way of fine tuning the baseline quality model based on field data, and using this information to improve confidence measures for findings.

8. REFERENCES

\begin{enumerate}
\end{enumerate}