TOWARD PRINCIPLES FOR CALL SOFTWARE QUALITY IMPROVEMENT

Author FARMER, RODERICK A.
Author HUGHES, BADEN
Author GRUBA, PAUL

ABSTRACT

As signalled by the lack of discussion about software development methodologies in the literature, it may reliably be argued that the general approach to CALL software development suffers from the use of immature software engineering practices, the result being that most practitioners in the field build systems that are idiosyncratic, prone to error and difficult to maintain. To address these shortcomings, there is an urgent need in CALL to adopt and promote the use of established practices from the wider field of software engineering. The aim of this paper is to outline key principles of software engineering practices that are appropriate to CALL contexts. Accordingly, we provide an overview of central software engineering principles, set out characteristic shortcomings of non-quality driven projects, and focus on software quality improvement (SQI) practices. We then present five principles for improved software development, and conclude with suggestions for the research in the emerging area of CALL Software Engineering (CALL-SE).

INTRODUCTION

Despite the various research disciplines which fall under the umbrella of CALL, a common concern facing each domain is an effective development methodology for quality computer-based tools for supporting language learning and assessment. For integrative CALL, this point is underscored by the increased dependence upon complex software applications. As the complexity of integrative CALL projects increases, so too does the difficulty of attaining the objective of developing quality software.

Ironically however, in a field centrally concerned with computing, our combined experience suggests that CALL practitioners are poor at meeting this objective. To address this shortcoming, Chapelle (2001) and Mishan & Strunz (2003) advocate the use of specific programming languages and frameworks in order to improve the productivity of software developers and the corresponding quality of software. Although in certain contexts these benefits may be observed, there is little evidence to suggest that such approaches consistently and effectively improve the quality of software and lower the costs of development in the CALL domain.

We propose that for the state of CALL software development to mature, a paradigmatic shift is required, re-focusing on the innate difficulties of developing quality software for language learning environments. The field as a whole must acquire a greater appreciation of the complexities of software development, including organisational, managerial, developmental and user perspectives. It is proposed that increased maturity of CALL software will eventuate, reducing costs and improving the predictability of successful project outcomes. Thus, we propose that the domain of software engineering (SE) has much to offer CALL research in this regard. In support of this position, we describe the benefits of software engineering as a discipline and outline a series of key principles (and their benefits) that CALL practitioners can adopt as an initial step towards integrating software engineering principles and practices into their projects. We propose that research
in this area should be labelled CALL-Software Engineering (CALL-SE).

**THE BENEFITS OF SOFTWARE ENGINEERING TO CALL**

A general description of Software Engineering (SE) would portray it as “a discipline whose aim is the production of fault-free software, delivered on time and within budget, that satisfies the user’s needs” (Schach; 1999, p. 4). Moreover, software engineers are typically concerned with the application of this principle throughout the entire software development life-cycle, with the expectation of developing fit-for-purpose (or contextualised) quality project deliverables – whether that is documentation, project planning and estimation plans, or risk mitigation strategies.

Tuohy (2002, p. 49) provides an overview of factors characteristic of software developments projects who possess and reliable upon relatively immature processes and practices (Table 1). It is our opinion that these features are in fact characteristic of many CALL software development efforts at the current point in time.

<table>
<thead>
<tr>
<th>Table 1: Tuohy’s characteristics of immature software development practices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different groups with their own ways of working;</td>
</tr>
<tr>
<td>Disparate tools and procedures;</td>
</tr>
<tr>
<td>Little effort to effectively combine people of different training, experience or expertise;</td>
</tr>
<tr>
<td>Varying approaches to making use of existing company software or products;</td>
</tr>
<tr>
<td>No agreed system of handling problems;</td>
</tr>
<tr>
<td>Unsystematic approach to identifying and managing status of software and documentation;</td>
</tr>
<tr>
<td>Lack of effective traceability between project phases;</td>
</tr>
<tr>
<td>Discrepancies between source code and design or other documentation;</td>
</tr>
<tr>
<td>Tendency to proceed to implementation without firm requirements;</td>
</tr>
<tr>
<td>Prevalence of the view that “Main work is coding”, often resulting in build-and-fix software development;</td>
</tr>
<tr>
<td>Poor estimation of how much time and effort activities require;</td>
</tr>
<tr>
<td>Inability to assess impact of changes.</td>
</tr>
</tbody>
</table>

At the core of software engineering lies a series of key principles and practices that inform the planning, monitoring and control aspects of a project, as well as the verification and validation of results obtained through empirical measures. These key processes affect all phases of the software development life-cycle from the initial establishment of a project, through to design, implementation, and finally software release management and maintenance. Table 2 demonstrates the four fundamental quality attributes, according to Sommerville (1989, 4) all software systems should possess.

<table>
<thead>
<tr>
<th>Table 2: Sommerville’s four fundamental quality attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maintainability</strong></td>
</tr>
<tr>
<td><strong>Reliability</strong></td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
</tr>
<tr>
<td><strong>Fit-for-purpose</strong></td>
</tr>
</tbody>
</table>

**CALL Conference 2004 - Proceedings**
To build robust, reliable, maintainable software, a commitment to software quality is essential. Nonetheless, the term ‘quality’ itself is ambiguous. Deming (1986), for example, asserts that quality means there is a predictable degree of consistency and reliability at low-cost. Eriksson & Torn (1991) submit that quality is a context-specific construct and is project-driven. Herbsleb et al (1995) define quality as the state of software when it is delivered to the customers and (according to their data) is innately linked to the number of post-release defects. Furthermore, Krishnan (1993) implies that usability, reliability and capability are inherent to quality. As a synthesis of these statements, we propose a definition of quality appropriate to CALL contexts:

Software quality refers to the degree to which all system/product deliverables correctly address the operational and functional requirements of each identified project stakeholder, and is delivered on time, on budget, and is maintainable over an extended period.

This definition provides a means of distinguishing between the various quality needs of the project stakeholders, including the appropriate distribution of resources. Preceding this definition is the necessity to establish organisational, project and developer specific procedures which are quality driven and goal oriented. Furthermore, commitment to this holistic quality improvement objective must be longitudinal and in the light of a continual evaluation of its applicability. Whilst there are several well-established quality-improvement methodologies and process maturity models which could be explored to further elaborate our position, we present an aggregate of these approaches as a set of core software quality improvement (SQI) principles would be more instructive to CALL practitioners.

**QUALITY-ORIENTED PRINCIPLES FOR SOFTWARE ENGINEERING IN CALL**

Quality improvement paradigms in Software Engineering (SE) are similarly structured to any other software development process; their methodologies/processes are either driven top-down, bottom-up, or integrative (top-down and bottom-up). As such, quality improvement strategies differ markedly from one to the next based on this core assumption. Moreover, it should be recognised that these strategies are not “one size fits all”, being both resource and domain dependent.

Introducing the concept of software quality to CALL practitioners must take into account the unique environment in which CALL exists. This does not suggest that highly successful universal software quality frameworks should be abandoned in favour of more localised CALL specific methodologies, but rather that a contextual analysis of the applicability of these SQI practices should precede any wide-spread adoption. The following five quality improvement principles are presented as a starting point for CALL practitioners in engaging in critical assessment of the quality of software quality. These principles adopt an integrative strategy to SQI, in as much as they highlight the need for organisational supports (top-down) and emphasise the utility of correct software engineering practices during the software development life-cycle (bottom-up).

**PRINCIPLE 1: UNDERSTAND THE SOFTWARE ENGINEERING ECOSYSTEM**

Without a doubt, the most critical principle in software engineering is the importance of understanding the environment in which software development occurs, and how the nature of this environment in turn impacts software quality. We refer to this environment as the software engineering ecosystem. As the name implies, the software engineering
ecosystem is grounded in a total life-time view of the software development process.

A standard reference model for the software development cycle is provided by the seminal Waterfall model introduced by Royce (1970). Whilst the Waterfall model has been criticised for its failure to address the real nature and complexity of software development (in particular, incremental and iterative development, client interaction, and release management) it does elucidate the important phases of the software development life-cycle (SDLC).

The Waterfall model pre-supposes the establishment of a development structure that supports the project environment. This structure should support the verification of processes, practices and work products at, and between, each phase of development. This is essential for building quality software. This also implies a high degree of documentation and organisation, without which, the development process can be said to be ad hoc and unpredictable (Pauk et al., 1995). While institutional CALL software development remains relatively small-scale, the Waterfall model\(^1\) will continue to provide an acceptable frame of reference for future software development.

**Principle 2: Adopt a Sustainable Development Methodology**

Arguably one of the most important yet difficult aspects of developing and managing quality-driven software systems is establishing a clear set of processes and procedures that lead to consistent, predictable and reliable results. Understandably, achieving a particular level of quality is of questionable benefit if there is little confidence surrounding its sustainability (Pauk et al., 1995). Therefore, adopting a sustainable method of development – processes, practices, standards – is vital to the success of a quality-driven project (Herbsleb et al., 1994). This overarching software quality principle is inherent to all SQI paradigms.

However, there are a number of considerations which must precede the adoption of a sustainable SQI program at both the organisational and project levels, notably

1. The degree of commitment to SQI available at an organisational/managerial level;
2. The degree of software engineering expertise within the organisation/project;
3. The complexity of the project;
4. The size of the project environment;
5. Available resources, including tools, budget, time and domain experts;
6. Impact upon mission critical services

Unquestionably there are a range of other considerations involved in evaluating an appropriate SQI framework (Fuggetta et al., 1998; Herbsleb, 1994; 1997; Persse, 2001; Pauk et al., 1995; O’Connor & Coleman, 2002; Kirshnam, 1993; Hollenbach et al., 1997), including a strong commitment to thorough documentation. However our experience shows that these six considerations provide sufficient support for differentiating the various competing paradigms. Table 3 identifies five SQI paradigms and their applicability according to these considerations.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Description</th>
<th>Best-Use Practice</th>
</tr>
</thead>
</table>

\(^1\)Including variants such as Rapid Prototyping, Saw-tooth and V-model SDLC’s.
<table>
<thead>
<tr>
<th>Approach</th>
<th>Description</th>
<th>Best-Use Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMM</td>
<td>The Capability Maturity Model (CMM) describes the management processes of an organisation and how they affect software quality, especially maintainability. Seeks to improve generalisability, predictability and reliability of software quality outcomes.</td>
<td>Requires strong commitment to quality from senior management. Top-down process driven. Greatest quality improvements when complexity and size of project is substantial. Cost is initially high but returns on investment justify cost. Essential for mission critical development.</td>
</tr>
<tr>
<td>GQM</td>
<td>The Goals/Quality/Metrics (GQM) approach addresses the need for goal-driven measurement. It focuses on quality goal expectations, development of metrics and data collection. Can be applied within the CMM model.</td>
<td>In the absence of strong management, GQM would be suitable for small to medium sized (hours, complexity) projects. Little domain expertise required, results are less predictable, tools required for data collection and metrical analysis.</td>
</tr>
<tr>
<td>SPICE / ISO 15504</td>
<td>Unlike the CMM, SPICE is assessment based and non-conformist (no rigid guidelines), and assessment is component based (such as requirements analysis) and not KPA-based.</td>
<td>SO typically does not infer quality practices, just conformance to cited procedures. Quality is derived from experts, management and software engineering expertise. Many management tools exist.</td>
</tr>
<tr>
<td>PSP/PIPSI</td>
<td>Personal Software Process (PSP) methodologies attempt to improve software quality by improving developer practices. This requires measuring, assessing and updating personal quality performance criteria.</td>
<td>PSP methodologies reportedly best suited for small to medium sized project. Suited to lack of management initiatives, suitable for complex projects. Requires substantial training and software engineering expertise. Fewer tools required.</td>
</tr>
<tr>
<td>IEEE Standards</td>
<td>The IEEE standards are guidelines for developing software according to Software Engineering principles. These guidelines are supported by leading industry and academic experts.</td>
<td>Presents fundamental software engineering principles. Provides developers with recommendations on how to produce bottom-up quality driven software. This is a minimum requirement of any software project.</td>
</tr>
</tbody>
</table>

These SQI frameworks should be adopted at the start of a project if their full benefits are to be observed (Dromey, 2003; Slaughter et al., 1998, Krishnan, 1993; Eriksson & Torn, 1991). These benefits include reduced variability of outcomes, increased predictability of SQI initiatives and project success, reduced development cost and increased coherence of team member behaviour (Herbsleb, 1994; 1997; Pauk et al., 1995; Tuohy, 2003).

**Principle 3: Clearly elicit requirements from stakeholders**

The ability to create quality software is directly linked to the elicitation and correct specification of the software requirements as they pertain to each of the project stakeholders. The requirements elicitation and specification process has a direct impact upon all subsequent aspects of the software development process. Incorrect specification

---

3 Schach (1999, 54-55).
4 O’Connor & Coleman (2002).
5 Schmidt (2000).
of requirements (ambiguity, lack of clarity, inconsistency, incompleteness etc...) can greatly increase the costs associated with project development and maintenance. Whilst it is beyond the scope of this paper to describe the myriad of requirements elicitation and specification techniques, a number of important characteristics can be presented.

1. Requirements should be clearly documented in a way which is accessible to all members of the development effort and relevant stakeholders. This documentation should present the requirements in an order, unambiguous and consistent manner.

2. Requirements should be sought for both functional (computational activity) and non-functional (external constraints such as hardware, environmental limitations such as operating conditions) aspects of the system.

3. The software requirements should clearly indicate the scope of the project. This implies that there should be a set of core software requirements that must be met in order to achieve the agreed upon user/client acceptance criteria. Furthermore, requirements should be prioritised in order to meet these objectives.

4. Requirements must be described in a way that affords traceability. As each subsequent stage of development needs to verify that the system meets the fit-for-purposefulness of the system (functional and non-functional requirements), it must be possible to trace these requirements as they progress through design, implementation, testing and maintenance.

5. The specification of requirements should not indicate how the requirements are to be implemented, rather they should simple focus on what is required in terms of desirable behaviour. Including implementation in requirements specification leads to inflexible software development.

As indicated in our first principle, the requirements must address the needs of all of the project stakeholders. Project stakeholders are those people who either influence the nature of the project organisationally, or who ultimately help to determine the success of the project. Eriksson and Torn (1991, p. 154) identify three principal stakeholders typically present in a project development – acquirers, users and developers. Typically, acquirers are interested in commercial aspects and refer to quality as a cost/benefit scenario. In contrast, users are most frequently interested in capability and reliability of the system when making their quality assessments (Krishnan, 1993). Developers are essentially interested in those software quality attributes which are related to software design, implementation and testing practices. As CALL practitioners are mostly concerned with developer correct and stable software for acquirers and users, it follows that requirements elicitation and specification techniques should be implemented that maximise the relatedness of the development effort to these individuals.

**Principle 4: Establish a clear set of quality objectives**

Software quality is a context-sensitive activity, therefore apart from a few universal characterisations of software quality it is necessary to establish a clear set of quality goals or objectives that the project will commit to. Central to establishing these goals is an evaluation of current practices and expectations, as well as how these quality goals can be objectively measured, quantified, and subsequently result in greater maturity. In Table 4 we derive from quality-goal methodology presented in GQM (Basili et al, 1995) a series of steps in establishing, monitoring and assessing quality objectives.

<table>
<thead>
<tr>
<th>Quality Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
</tr>
</tbody>
</table>

6

CALL Conference 2004 - Proceedings
<table>
<thead>
<tr>
<th>Quality Goals</th>
<th>Purpose</th>
<th>Examples include control, improvement, tracking.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality Focus</td>
<td>What quality attributes (functional/non-functional) are related to this quality goal?</td>
<td>Examples include robustness, reliability, correctness, reuse, portability, scalability.</td>
</tr>
<tr>
<td>View Point</td>
<td>The quality goal concerns which stakeholders?</td>
<td>Examples include program manager, end-user, research associates, department.</td>
</tr>
<tr>
<td>Environment</td>
<td>What is the context of this quality goal?</td>
<td>Examples include organisation-wide improvement, current project, current program.</td>
</tr>
<tr>
<td>Baseline</td>
<td>What is the current state of quality in the system/organisation</td>
<td>Examples can be relative terms such as high-low, or as metric (if available). Must remain consistent across project.</td>
</tr>
<tr>
<td>Hypothesis</td>
<td>What gains/losses are hypothesised through the implementation of this quality goal?</td>
<td>Examples include decreased maintenance costs, CASE tool investment.</td>
</tr>
<tr>
<td>Goal Priority</td>
<td>What is the relative priority of this goal?</td>
<td>Values can be relative (high-low) or ordinal (such as 1, 5, 10). Priority should be consistent with the nature of the cited hypothesis.</td>
</tr>
</tbody>
</table>

Table 4: A GQM-derived process for establishing quality goals

Without the establishment of quality goals and the ability to effective monitor and measure related activities, it is impossible for projects/organisations to describe their current and future levels of quality and process maturity. Following from our third principle – requirements engineering – it should be clear that quality goals can be established at the start of a project, when the requirements, or needs, of the various stakeholders are elicited.

**Principle 5: Use a standard evaluation framework**

An important principle for the assessment of software quality in a continuously evaluative environment is to adopt a single formal framework for quality evaluation. While a number of such evaluation frameworks exist at various levels of formalism (ranging from the informal “does it work?” to ISO-style quality assurance frameworks), we advocate that the IEEE Software Engineering (Schmidt, 2000) are in fact a useful point of reference. The IEEE Software engineering standards are a universally employed set of recommendations and guidelines for engineering-in-quality into software systems, and evaluating resultant software.

The IEEE Standards on Software Engineering have been instrumental in progressing the tools and techniques applied in software engineering. The standards act as guidelines to assist software engineers in developing quality products and as a baseline for evaluation. Although the IEEE Standards are only recommendations, they are highly detailed, and adherence requires application of a strict set of practices. As such, the standards have been shown to reduce variability in software development efforts, increase predictability of project outcomes, and increase overall quality of project deliverables (Schach, 1999, 281; Tuohey, 2003) during each phase of the software development lifecycle.

An example of how aspects of the IEEE standards can be applied to CALL is seen in recent
work by Grützner, Weibelzahl, and Waterson (2004) on improving the quality of educational courseware and learning material. Additionally, the validity of the work conducted by Grützner et al (2004) can be assessed using the IEEE standards, highlighting the various strengths and weaknesses in their approach. Unlike the CMM, IEEE standards are principally bottom-up or practice driven. It is assumed that the principled approach advocated by the standards are valid in as much as they are preferred by leading experts from industry and academia alike.

**CONCLUSION**

To provide CALL practitioners with a foundation in software quality improvement, we have elucidated five fundamental principles for building quality-enhanced software, namely: understand the software engineering ecosystem; adopt a sustainable development methodology; clearly elicit requirements from stakeholders; establish a clear set of quality objectives and use a standard evaluation framework.

Through this discussion we have made reference to leading software quality improvement paradigms as well as internationally recognised standards on software engineering software systems. Moreover, we have described how CALL practitioners can evaluate these processes from a top-down bottom-up perspective, assisting them to make an informed decision as to the most appropriate SQI for their context and project infrastructure.

In this paper we have argued that the current state of CALL software development is immature, relying upon ad hoc processes and practices, and failing to consistently produce quality software project deliverables. We have argued that the introduction of core software engineering principles, including software quality improvement practices, can greatly increase the quality and usefulness of CALL software whilst reducing the lifetime costs of the development effort.

**REFERENCES**


Biodata

Rod Farmer is currently a PhD Candidate in the Department of Computer Science and Software Engineering at The University of Melbourne.

Baden Hughes is a Senior Research Associate in the Language Technology Research Group within Department of Computer Science and Software Engineering at The University of Melbourne.

Paul Gruba is a Senior Lecturer at the Horwood Language Centre within the School of Languages at the University of Melbourne.

Contact

Rod Farmer
Department of Computer Science and Software Engineering
The University of Melbourne
Parkville, Victoria 3010
Australia
Email: raf@cs.mu.oz.au

Baden Hughes
Department of Computer Science and Software Engineering
The University of Melbourne
Parkville, Victoria 3010
Australia

Email: badenh@cs.mu.oz.au

Paul Gruba
Horwood Language Centre
The University of Melbourne
Parkville, Victoria 3010
Australia

Email: paulag@unimelb.edu.au