Some Aspects of Software Quality Assurance for Small Projects

A. Spiteri Staines

Abstract— Agile and lightweight software based projects have requirements that are not always easily identified because they are not always well developed and documented in the same way large projects are documented. This work presents some aspects that can be included to improve these projects in a semi-structured approach.

Index Terms— Agile Methods, Best Practice Approaches, Quality Assurance, Quality Measurement, Lightweight Methods, Model Driven Engineering, Project Management, Software Engineering.

1 Introduction

In the past decades the aspects of software quality have been given considerable importance. Software quality is not something straightforward to quantify and assess compared to other measurements used in engineering.

Quality assurance is a bit of a paradoxical topic. If quality control is left to the later stages of development rather than having an ongoing quality process, serious unsolvable problems can arise [1]-[3].

Software is not a tangible product like hardware. It is not easily defined and represented. The definition of quality is quite abstract. Thus putting together software and quality creates a topic that is not so clear and straightforward. As a matter of fact software quality is very abstract and difficult to measure. The diverse types of systems and software applications and requirements engineering topics all contribute to increasing this complexity [4],[5]. In simple terms there is no absolute measure of software quality. Some fundamental questions can be asked about software quality [6]: i) when is the best time to influence software quality, ii) who should be responsible for software quality assurance, iii) which methods, metrics or approaches can we use or combine to measure the quality factors, iv) which stakeholders shall we involve. v) which quality measures can be involved in the process.

For small IT projects agile methods have certain advantages and important uses. Agile methods can be compared and combined with a direct approach that in a certain sense is more specific and straightforward. A fundamental argument on reasoning about the system implies that the system is an engineering solution or product that resolves a particular problem. The system is composed of software, and also the users, stakeholders, networking etc. The behavior of the system must be predictable and must satisfy the customer's needs. In fact the system has to add value to the customer's or user's process and meet a particular user's need or specific problem. Software quality assurance is not just limited to the software artifact but it is also part of the process. Software quality assurance from a wider perspective involves different entities and dimensions that are often overlooked [7],[8].

• *Anthony Spiteri Staines is lecturing in the Department Of Computer Infortation Systems at the University of Malta*

2 Background Information

2.1 Basic Quality Assurance Objectives

In this section, some very basic software quality assurance objectives are given [1],[2]. Software quality can be identified in i) the process and, ii) the product. Some of the main quality assurance objectives are: i) system/software development should meet the needs of all users, ii) the system is consistent with the needs of other users, iii) system goals are consistent with those of the customers and the organization, iv) the system should clearly meet certain security specification standards imposed by government, industry, competition or business domain [6], v) the software is developed at a suitable economic cost, vi) the errors in the development should be minimized, vii) there should be no need for rewriting program code or parts of the system, viii) the customer should be satisfied with the process and development taking place [3],[4]. It is obvious that these objectives are loosely defined and on their own do not give a clear indication of how they should be achieved in practice. Unfortunately different projects have different objectives, creating difficulties in selecting what is important.

2.2 Intangible Quality Assurance Objectives

Quality assurance intangible objectives relate to the performance of the system [1]-[3]. Some objectives are: i) economical ones that imply that the system cost is minimized, ii) effectiveness that imply that the system accomplishes the required tasks with the least effort, iii) maximizing throughput or use. The basic quality assurance objectives mentioned, can be considered to be intangible, because there is no single way of measuring them.

2.3 Quality Assurance in Traditional Software Development

Traditional system development had quality assurance plans that fell under the responsibilities of a quality assurance group [1]. Such a setup however is not particularly well suited to small projects. However this can be included as part of an organizational setup where a specific group of the organization is responsible for quality and standards assurance from a wide perspective. Many lessons learned from quality assurance in

traditional development like those related to product quality, timeliness, design, documentation, standards, etc. can be easily applied to smaller projects.

2.4 Agile Methods

Agile methods are based on creating systems and software artifacts that add value. The concept of service oriented architectures is based on these concepts [5]. Techniques and approaches like agile are directed towards solving problems in software quality. However the actual way of doing this is not always clear. The use of modern improved platforms, virtual machines, integrated environments, frameworks, collaboration software, version control mechanisms, configuration management systems, error tracking, traceability, etc. can dramatically improve the software process and product. Problem structuring and modelling at an architectural level could be useful.

3 Problem Definition

The problem is how to have real quality assurance in small projects. This is nothing short of a simple task. Quality measures imply specific tradeoffs between different attributes. Measurement is not so straightforward and accurate but just a guideline. The main key concepts can be summarized into three main points. These are i) trust, ii) dependability and iii) cost minimization.

Trust implies that system users exhibit a certain amount of familiarity and comfort with the application that they use or are entrusted to use.

The concept of trust does not necessarily imply that a system is reliable or efficient. Trust is a mental process that cannot be measured using normal means because for the user, this value lies in his mind and is very vague or difficult to define. Trust is an intangible measurement and many of the measurements used for software quality try to measure some basic aspects of trust. Some indirect indicators or pointers of trust in the system can be obtained by the amount of use of the system and the number of entities that use it.

Trust can be seen in the process of systems development where the stakeholders influence the whole process of developing the artifact. This type of trust again is quite complex and can be the result of many issues like the notations used for the design, the tools or frameworks being used for development, the level of confidence in the programmers and many other facts. E.g. if a new system is being developed by a new company this can cause a particular reaction. Obviously a good level of trust will instill a certain amount of confidence in the end result.

The second quality is dependability. This is based on trust, however this can be measured to a better extent. Dependability is not determined by the level of trust and vice-versa. Dependability implies that a system does exactly what it is supposed to do. I.e. it meets the users or stakeholders criteria perfectly. In the case of the software artifact, this has to take place without failures and problems. Dependability is measureable by the robustness or reliability of the system. But dependability is more abstract and difficult to measure than reliability. Quality assurance plans seem to be more oriented towards dependability rather than trust. This happens because many metrics are defined for measuring performance. Performance depends on the smaller measures of effectiveness and efficiency. These are easier to measure because they can be based on numeric data that is logged off from the system. The issue of dependability affects both the process and the product of small projects. Software development has to be a dependable process. Dependability relies upon quality culture and a best practice approach. A certain level of dependability is achieved after a number of years. This can be seen from the principles of the capability maturity model (CMM).

Cost minimization implies that the process and the artifact take place with proper cost control. Resources cannot be underutilized. Cost minimization is an economic principle. It results from proper quality control mainly in the software development process. This implies that the artifact produced does not require an excessive amount of rework or maintenance. The cost required for developing quality beyond a certain level becomes prohibitive. A required level of confidence has to be agreed upon prior to the development stage. Having requirements that are properly specified does not automatically imply that the best tradeoff of cost vs good requirements is achieved.

Some typical problems for small projects are: i) certain goals might only be achievable in the future. ii) it is difficult to predict defect rates.

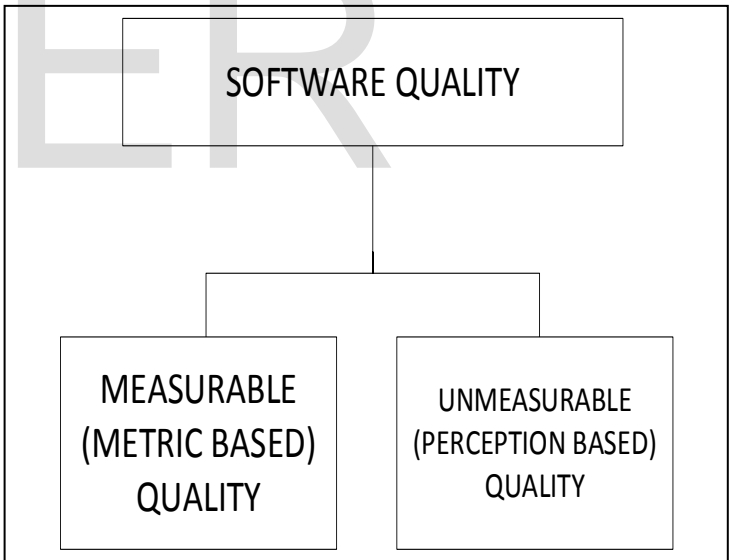

Fig. 1. Measurable vs Unmeasurable Software Quality. This fig. briefy states that software quality can be i) measured or ii) perceived in the user's mind.

4 Proposed Solutions

Improving software quality cannot be done by using predefined quality assurance methods. This is because each project has differences in size and requirements. Each problem is unique and will require a specific attested form of problem solution to it.

A two fold solution can be proposed. i) Keeping sufficient

factual data and information about software projects, ii) Identifying appropriate strategies.
The first part of the solution to the problem lies in keeping sufficient historic records, factual data and information. In other fields of study like business and organization management statistics and management science can be used to identify possibilities and results. The same concepts can be applied to software development. Normally for small projects it is highly improbable that data is kept in the same way as it is done for larger projects. If software developers, analysts and teams keep proper records, it could be possible to identify causes of project failure. This can help to improve software design and development strategies and address shortcomings in software quality.

The second part of the solution implies identifying and devising proper strategies to solve the problem. Unfortunately this is a difficult task and there is no fixed solution that will work in every case. It is impossible to guarantee a complete success. The aspect of doing things in the right way has to be embedded in the organization's culture.

5 Some Adhoc Quality Assurance Success Factors

The following are considerations that seem to have contributed to improving software quality in some aspect or another in modern software development environments. These are shown in fig. 2.

5.1 Use Principles of Agile Modeling

This can sound a bit contradictory because agile or lightweight methods imply simplifying the work. However if Agile methods are properly examined, there are quality principles that have to be applied for the successful implementation. Agile modelling is not part of Agile methods. These are independent notations which exists to support Agile methods.

Agile modelling is a process that can be used along with agile and lightweight solutions. Agile modelling implies that agile can be improved by applying certain core principles and diagrammatic notations to model a particular problem. The focus is to simplify various problems.

Agile modelling is a process for modelling and documenting systems. It is based on these core principles:

i) Assure simplicity and travel light
ii) Embrace change
iii) Use incremental small changes
iv) Use exclusive models fit for purpose
v) Maximize stakeholder values and investment
vi) Create models in parallel
vii) Use simple tools and solutions

These methods can be further developed using the following criteria:

i) Carry out verification using questionnaires based on a scoring method

ii) Use expert judgment from external auditors and senior experts looks for present/ presence of specific criteria.

iii) QA measures are used to define quality charac

iv) Use a missing criteria method. Here the QA analyst looks for present/ presence of specific criteria.

5.2 Use Architecure Centric

Software architectures serve as a blueprint for the system and project developing it. In simple terms the architecture serves to deploy the problem in order for obtaining a feasible solution [9],[10]. The architecture is like glue that holds the complete structure together. Thus it is responsible for the vision that unifies the different stages of the system development. Modern systems are composed of a set of components that interface together and allocate specific functionality fulfilling certain rules. Decomposition can benefit from patterns used for a given solution. The rules, decompositions and patterns that are included in the architecture contribute to quality because they enforce compliance and structure. The concepts underlying MDD (model-driven development), PIMs (platform independent models) and CIMs (computation independent models), if properly understood are useful for quality at a high level of abstraction [11]-[13].

5.3 Use Model Driven Approach Concepts

The following principles from model driven approaches are imperative for quality improvement [8], [9], [12],[13]:

i) Efficiency : implies the elimination of manual work and errors, improves implementing UML models.

ii) Agility: Approach makes it possible to work in a given way. It implies the quick development and testing of the system from the requirements documentation.

iii) Flexibility in hardware platforms. This implies testing the code even if the hardware is not yet available

5.4 Use Reliability Measures

Several different ways exist to measure reliability. Reliability is not something that is normally measured for small projects because in some occasions this has a limited importance. However very simple things like an accuracy checklist or error tolerance checklist can contribute to having proper measures in place.

5.5 Use Correctness Measures

Correctness is not a simple measure. If it is given more importance than necessary this can become a problem to the project. Correctness depends on different stakeholder views. For measuring simple forms of correctness, simple checklists and scoring methods can be used.

Model completeness, user viewpoint checks, consistency and user checklist and operability checklists are useful for improving quality in software. The concept of a completeness checklist can be extended to other parts of the software project at different levels. These measures are not direct measures of software quality, however they can contribute to indirect improvements in quality.

5.6 Use Usability Measures

The usability factor is strongly dependent on the application being developed. Usability is not straightforward to measure and training or different user groups can have a different perceptions of this. The concept of operability or usability measures the simplicity of the system operations.
User friendliness and the level of skills required to operate a system measure usability. Simplicity is important for having good quality.

Interoperability explains how the system can be operated across different groups and platforms. All projects require a reasonable amount of effort for their success. Systems need to integrate with existing technologies and architectures whilst satisfying the requirements of different stakeholders. Thus quality must be included from the start and planning phase of particular projects providing for better results whilst offering something that is economically viable and sound. Usability is related to interoperability, connectivity, user satisfaction maximization and maximizing the stakeholders value of the system. The concepts of value added can be applied to usability and the usefulness of the artifact. This can be extended to the process for developing the artifact.

5.7 Create the Right Culture and Environment

The right culture is imperative for quality in agile methods and small projects. Unfortunately creating the right culture is not something that can be done overnight. The only way this can be allowed to happen is through a sustained effort over a long period of time.

It is important that the appropriate culture exists even before starting any project. Without this culture it is impossible to have proper quality in software projects.

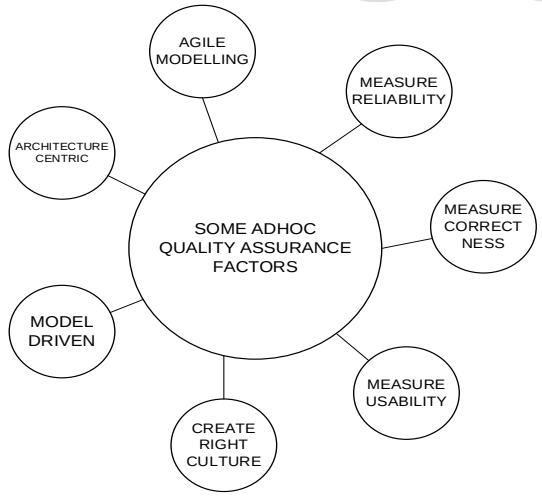

Fig. 2. Several Quality Assurance (QA) Factors. This fig. briefly shows the main factors that can affect QA for small projects. Each factor is very vast and detailed.

6 Results and Findings

The solutions, suggested in the problem solution do not imply that there is a single right mix. These are generic solutions that can be applied to any small project as required and are based on common sense and sound principles. Considering a traditional approach, key concepts used here can be applied to modern projects. But modern projects have more complexities and intricacies because of different platforms, levels and components that spawn across different levels. Thus architectural features have to be considered.

In a best practice approach these principles can be found to have valid use. Modern lightweight development principles focus on agile solutions, so agile modelling concepts like: travel light, use models fit for purpose, use simple tools and solutions, etc. are definitely important and useful. On their own agile modelling can help to improve quality but other concepts of measurement like score sheets and ranking metrics can prove to be useful to give more tangible measurement to the agile modeling.

Modern software development is based on PIMs and MDE (model driven engineering), hence focusing on an architecture centric approach at the top level can help to improve the quality of the design and the actual system [14].
If an architecture centric approach is not used directly it would still benefit to apply the concepts mentioned in this work. These are adaptable to finding a better and more integrated overall solution.

Reliability measures can be used to overcome the performance and correctness problems in the system. This is a vast topic. This paper has just skimmed the surface.
Usability is another important attribute. It is possible to have really good applications and development that have usability problems. Usability on its own right can be very important because it determines the success or the failure of the system. It is the actual user who guarantees if a system is useful or not. Creating the right culture follows from the other factors that have been mentioned.

The usefulness of these key principles is observed in various case studies. However these are not normally grouped together. The success of projects is clearly determined by having a systematic and structured approach. This maximizes the stakeholder value.

For this work a basic questionnaire was given to a group of software engineering students who had been familiarized with agile and its notations. Some of the questions dealt with the following issues: i) too much models create confusion to maintain, ii) extreme programming does not require any models, iii) no single modelling approach can work for different problem domains, iv) good models should have strong visual expression etc.

From the results it is obvious that Agile or lightweight software development definitely requires models. On the other hand too many models create confusion.
If a similar questionnaire would be presented to industry where small software projects are done it will probably generate similar results.

It is difficult to find the right mix or balance how to achieve quality issues. These can considerably vary from one project to another. Stakeholder groups can suggest different opinions as to what type of quality is really important.
Thus the following observations have been reached:

i) Continued support for sustaining & improving quality in small software engineering projects is needed

ii) Results might not be seen immediately

iii) Online stores or knowledge repositories where valuable information and lessons learned can be stored for future reference should be created

iv) A best practice approach culture should be created.

The fact that quality in software is intangible and abstract makes it something very difficult to implement and complex to measure. Unlike other engineering approaches where quality can be measured from the success or failure of a product subject to a set of physical and functional examinations or tests at the physical level, software cannot be examined in the same way. Measurement principles mentioned in the solution are applicable to a wide variety of small projects. The difficulty with measurement in small projects is that, metrics are seldomly applied in this case.

6 CONCLUSIONS

It is possible to disagree to what quality actually is in small projects. From the viewpoint of IT strategy, planning and quality in software projects is imperative if the projects are to succeed. If the concept of 'value added' is embraced quality is a must.

As time passes in an organization, more experience and ground is gained in what quality implies and means for the organization. From the perspective of agile and lightweight methods, quality has to become part of the culture that pervades every level of the software process. This cannot happen overnight. Having more complex choices and outcomes and different forms of leadership and planning means that there is the need for simplifying and unifying principles that will allow the correct implementation of quality factors.

REFERENCES

[1] W.E. Perry, "*Quality Assurance for Information Systems: Methods, Tools and Techniques*", Wiley & Sons.: NY ,pp. 361-395 1991.

[2] P. Clements, L. Bass, R. Kazman and G. Abowd, "Predicting Software Quality by Architecture-Level Evaluation",*5th Int. conf. on Software Quality*, vol 5, no 0, Austin, TX, pp. 485-497, 1995. (QICID: 11205, ASQC)

[3] L. Bass, P. Clements and R. Kazman, "*Software Architectures in Practice*", Addison Wesley, 2006.

[4] M. Aleksy, "Coverage of Design for Service Principles in Software Engineering", *6th Int. Conf. on Complex, Intelligent, and Software Intensive Systems (CISIS)*, Palermo, Italy, pp. 100-105, 2012.

[5] G. Booch, "The Defenestration of Superfluous Architectural Accoutrements", *IEEE Software Domain Specific Languages and Modelling*, vol l 26, no 4, pp. 7-8, 2009.

[6] L. Cao, B. Ramesh and M. Rossi, "Are Domain-Specific Models Easier to Maintain than UML Models?",*IEEE Software Domain Specific Languages and Modelling*, vo.l 26., no. 4., pp. 19-21, 2009.

[7] R. Collier, G. O'Hare andC. Rooney. "A UML-based Software Engineering Methodology for Agent Factory", *Int. Conf. on Software Eng. And Knowledge Eng. (SEKE),* pp.25-30, 2004.

[8] M.A. Jeusfeld, M. Jarke, and J. Mylopoulos, "*Metamodelling for Method Engineering*", MIT press 1st ed., 2009.

[9] M.A. Jeusfeld, M. Jarke, H.W. Nissen and M. Staudt, "*ConceptBase Managing Conceptual Models about Information Systems, Handbook on Architectures of Information Systems*", Springer, ch. 12, pp. 265-285, 1998.

[10] D.L. Moody, "The Physics of Notations: Towards a Scientific Basis for Constructing Visuals in Software Engineering", *IEEE trans. On Software Eng.*, vol 35, no 6, pp. 756-779, 2009.

[11] J.A. Stone, E. Madigan, "Inconsistencies and Disconnects", *Communications of the ACM*, vol.5, no 4, pp. 76-79, 2007.

[12] OMG, MDA - The Architecture of Choice for a Changing World (2013), OMG Documentation Website: http://www.omg.org/mda

[13] A. Mattsson, B. Lundell, B. Lings, B. Fitzgerald, "Linking Model Driven Development and Software Architecture: A Case Study", *IEEE Transactions on Software Engineering* ,Vol 35 , Issue 1, pp.83-93, 2009.

[14] A. Van Lamsweerde, "*Requirements Engineering: From System Goals to UML Models to Software Specifications*", Wiley, 2009.