Software Quality and Life Cycles

Hannu Jaakkola[1] and Bernhard Thalheim[2]

[1] Tampere University of Technology, Pori
P.O.Box 300, FIN-28101 Pori, Finland
hannu.jaakkola@tut.fi
[2] Kiel University, Computer Science and Applied Mathematics Institute,
Olshausenstrasse 40, 24098 Kiel, Germany
thalheim@is.informatik.uni-kiel.de

Abstract. Quality of software has growing role of the modern software engineering work. Typical current *trends in the development process* are the dominating role of quality systems, the use of "productivity tools" in the development, fast time to market and the adoption of the practices of the software development industrialization. Correspondingly the *software product trends* point out the growth of software size, longer life time, the critical role of software in services and products, increasing amount of client oriented variations of the same base product, and standardized platforms under the product solutions itself. As a consequence the new paradigms are guiding the software development: e.g. object oriented development culture is spreading fast and the role of reuse in its different forms (component level, patterns, designs) is becoming more important. Even the standard platforms, like Symbian, are giving guidelines to the developers restricting the freedom of solutions. The question on the software quality – in principle a clear and simple concept – is becoming more complex to specify and especially to teach for becoming software specialists. The paper includes the discussion on software quality issues from different points of view. The approach adopted classifies the quality based on software life cycles pointing out that the importance of different factors is changing along the development cycle.

1 Introduction

The goal of every software developer is to produce software having good high quality – both in the personal and in the organizational level. The competitiveness of the software company is also fully dependent on its working practices. *Software Process Improvement (SPI)* of a company can be based on different models having very similar goals: to produce software that has good quality according to the plans and agreements agreed between the client and producer. Widely adopted models for SPI are ISO 9000-2000, SPICE (Software Process Improvement and Capability dEtermination) and CMMI (Capability Maturity Model Integration). Common to all these models is the aim to encourage the developer for *continuous improvement*. A common approach is as well the main idea, that the well working and specified process is the compulsory condition to be able to produce quality software (experiences in organiz-

ing SPI activities are reported by Jaakkola [10]). However, or unfortunately, this may not be enough. In addition we have to keep in mind also some properties of the *good product*, that are not only process based but can be derived from the product itself.

The motivation of this paper comes from the academic world. In teaching both software development (design and implementation) and engineering (the integrated process) the question of the good software properties rises up. The starting point of this discussion can be derived from the trends of modern software properties and development practices. In the *development process* e.g. following trends can be noticed:

- dominating role of quality systems,
- use of "productivity tools" in the development,
- fast time to market and
- adoption of the practices implementing the idea software industrialization in its many forms.

Correspondingly the *software product trends* point out e.g.

- growth of software size,
- long life time of products,
- critical role of software in services and products,
- increasing amount of client oriented variations of the same base product and
- standardized platforms under the product solutions itself.

The trends from one point of view increase the complexity of software development but from another point of view are able to provide clear guidelines to follow at least partially to solve the problem.

In the Software Engineering (SE) literature, conferences and journals the topic is widely discussed from many different viewpoints. In spite of the publishing date twenty years ago the article "*There is No Silver Bullet*" written by Fred Brooks (1987) gives a good frame to analyze the question "What is software quality?" The message of the article is that producing software is a difficult and complex task; the complexity comes from the *essence* of the software and cannot be avoided at all. He classifies the problems of software engineering into two groups: essential and accidental. Essential problems are complexity, invisibility, adaptability, uniqueness, scalability and discontinuity. We can take into account these problems, but not to avoid and totally solve. The techniques and methods, that often are seen as a solution, focus on and are able to avoid the consequences of *accidental* problems (caused by the human behavior) only. Better tools, a high abstraction level in the programming tools etc are means to this.

Generally, the solutions implemented as a part of modern software development culture, are object oriented development, reuse in its different forms (component level, patterns, designs); see e.g. [9], the use of standard platforms, like Symbian, giving guidelines and tools to the developers but restricting the freedom of an individual organization and developer, and in general the availability of software and process standards spreading and providing documented good practices widely available.

In general, it is question on the culture, where every individual developer and in the inter-organizational projects even the organizations, are forced to follow tightly the rules of predefined process and product architecture solutions. There is also critique against this the approach, where an individual (engineer or organization) is tied to the apron strings. The "Agile Software Development" methods can be seen as a manifes-

tation of this new trend (see e.g. [1]). Agile Software Development emphasizes the role of team work and loosening the bindings to the predefined processes and specifications.
The balance in the software quality discussion is heavily weighted to the process side, which undoubtedly has earned its position. However, the discussion on the topic "what are the components of the software product quality?" is interested especially in education and training. This discussion is able to give input to the process view, too.
Two important technologies supporting software developers of object oriented software are available: the UML language (see e.g. [3]; the original standard is [13]) and the development process applying it (Rational Unified Process – RUP; see e.g. [8,12]). In spite of the fact that UML is still under development and improved versions of it are available, and that RUP itself is not very widely adopted in business, the guidelines provided by them are very valuable. UPEdu, specified by Robillard and Kruchten (2003), is a simplified application of RUP. It gives guidelines how to apply RUP principles to a software project; in this application an iterative development principle is highlighted.
This paper concentrates on the question of the software. The paper includes the discussion on software quality issues from different points of view: overall, specification and design quality (chapters 3-5). As the basis a short introduction to the UPEdu development process is given (Chapter 2). The purpose is to open discussion on the problem caused by the different quality goals in different life cycle phases. As a conclusion (Chapter 6), some basic principles to avoid conflicts between these conflicting goals are given.

2 Object oriented software development cycles

Object oriented software development process is two dimensional, iterative process, in which time dimension (development phases) and development processes (disciplines; iterations) are combined (Fig. 2-1; [14], p. 39) to provide and useful practical approach. The development phases are same as the traditional life cycle phases and are based on the natural path in problem solving. The *inception* includes requirement elicitation and analysis, the *elaboration* phase includes requirements analysis and design, the *construction* phase concentrates on implementation and testing, and the *transition* phase on system testing and adoption of the system developed. The vertical dimension lists the character of the work done as described above. The message of the figure is that development work is iterative (adding details to the solution phase by phase) and cyclic (we have to return to same disciplines (processes) phase by phase.
In addition the model includes some management disciplines (processes): Configuration and Change Management, Project Management. From process category point of view UPEdu is much simplier than the SPICE Model, which specifies tens of processes (disciplines).

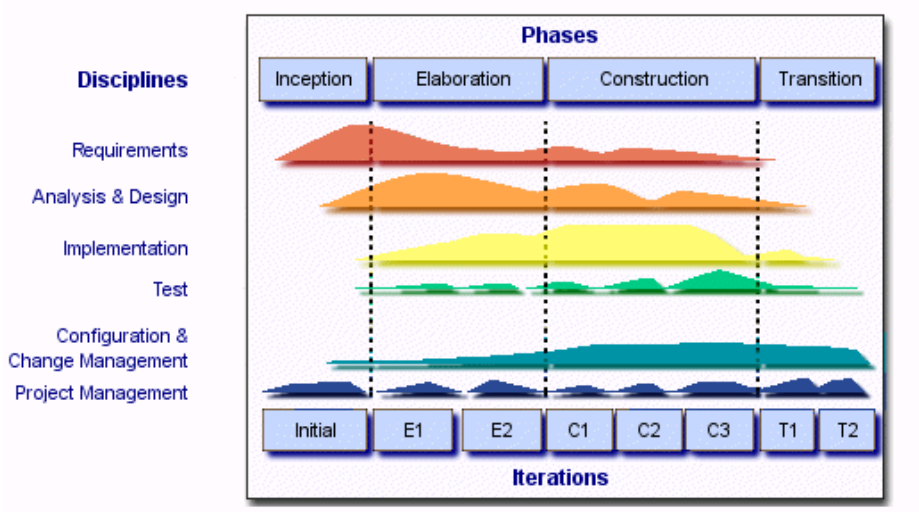


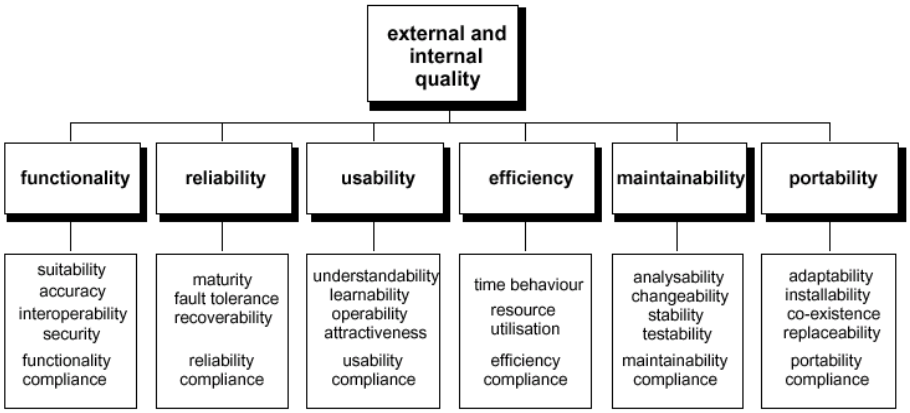
Fig. 2-1. UPEdu development cycles and disciplines (processes). [16]

Every Development phase has its own goals:
Inception phase is client oriented and the final result of it (requirements specification) provides on the one hand a "*natural" description* to the client (real world aspects) and on the other hand an exact (enough) specification to the designers continuing the work towards the implemented solution.
Elaboration phase is designer oriented and the final result includes detailed requirements specification and architecture of the system under development; the result combines the *real world* and the *design world* aspects in the result model.
Construction phase implements the plan and includes "*implementation world*" to the system solution.
Transition phase starts the adoption process of the system implemented and returns back to the concepts of the "real world" first in the form of system and user tests and finally in the form of the usage.
The three "worlds", real, design and implementation world, have different goals to the deliverables. Theses partially conflicting and ambiguous goals are the source of software quality factor discussion of this paper.

3 Overall quality

The quality factors of software are widely discussed in the literature. There is no special reason to concentrate on this detail more widely than by giving one view to the topic more or less as an example. Generally, independently on the source, all discussion agrees the fact that the general quality goals given to the software are conflicting and the final properties are more or less based on the *compromise* between

the conflicting goals. The stakeholders of the system under development give the boundary values.
One of the commonly accepted source specifying the properties of a good software is the standard specified by ISO [4]. The standard classifies the quality attributes in four classes: process quality, internal and external quality, and quality in use. The values of these attributes influence each other. In addition, there must be *metrics* providing bases for giving goal and controlling the level approached. From software quality point of view the most important classes are internal and external quality attributes and quality in use attributes (Fig. 3-1 a and b).



a. Internal and external quality

b. Quality in use

Fig. 3-1. Overall software quality ([4], p. 7 and 12)

Without going to the details, it I easy to notice that the goals specified above are in some cases supporting each other but in some cases may be conflicting. In spite of that the list (including detailed specification in the standard) are giving useful guidelines for software developers.
The ISO series of product quality standards has been extended by three new parts (Technical Reports – TR) covering the metrics for external [5], internal [6] and quality in use [7] attributes. These standards are not in the focus of this paper.

4 Specification quality

The goal of the requirements analysis and specification (in UPEdu model inception + elaboration) is to provide analysed requirements elicitated from the clients (and other stakeholders) to the designers to establish a basis for the design and further for the implementation of the system. The final result of the requirements specification analysis is the abstraction of the real world in the form of diagrams describing the data contents and operations manipulating the data. In object oriented development the most meaningful elements of the document are

- *use cases* as the documentation of the requirements (and providing cases for the system tests),
- *analysis class* diagram including preliminary information about the attributes and methods of the classes,
- potential *communication diagrams* describing the communication between objects,
- potential *state diagrams* describing the internal behavior of the classes, and
- potential *other processing descriptions* providing information about the execution of the methods.

This description is fully based on the view of the real world having no elements coming from design and implementation solutions. Good *software quality* in this phase is a description having *clear and close conceptual connection to the corresponding real world concepts,* describing the target system in such a level, that all the stakeholders have an opportunity to understand "what is the system" under construction. Two typical components of the system – data and handling – may be conflicting from the software quality point of view: that what is a good database specification is maybe not a good process specification. This conflict becomes more relevant in the design work, however, and is discussed more detailed in chapter 5.

5 Design Quality

The purpose of the design work is to develop the result of the requirements work first towards the architecture and after this to the level of module designs. In this phase, the work is enriched by the agreed rules of the design paradigm and design decisions. These are e.g.

- the rules coming from the object oriented approach,
- the attitude to reuse,
- the selected architecture style (if any), and
- the attitude to the product strategy.

Object oriented development culture itself gives a good base to create conceptually clear and structurally good designs. The basic rules like

- simple solutions,
- local solutions,
- modularity,
- restricted service interfaces and

- abstract data types

are in favour with the overall quality factors. However, the good object structure is not in all cases in favour with the good database structure: which of the classes are "database components" and which have some other purposes. The main question is that a good database schema may not be a good solution from the point of view of the good class structure – at least not natural in all cases. To be short – database quality and software quality may be conflicting.

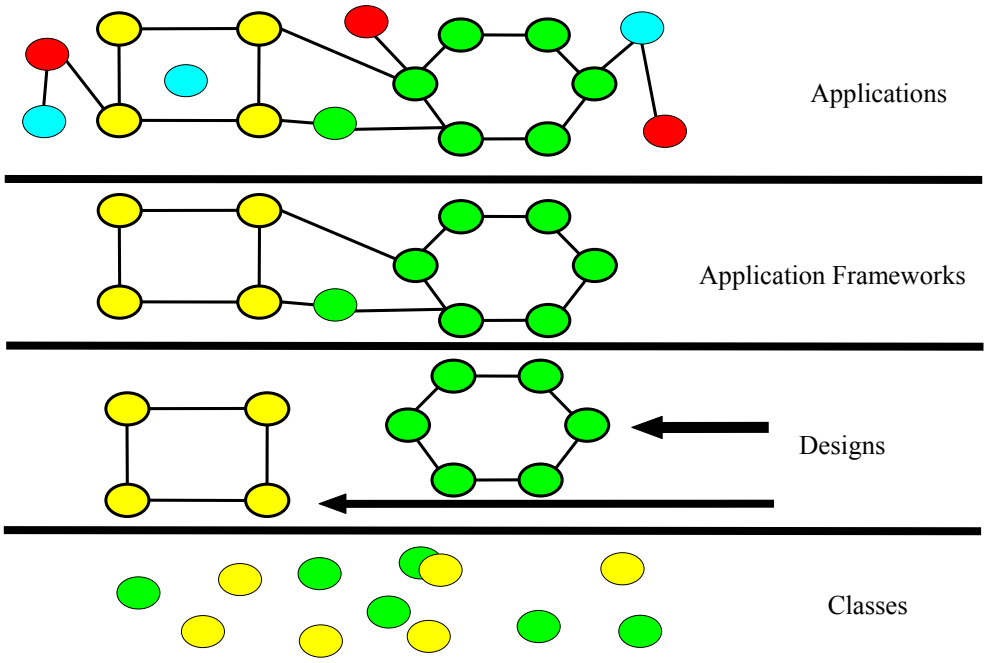


Fig. 5-1. Different abstraction levels of reusable assets.

Reuse is one solution to grow the productivity of software companies. Usually the term reuse is connected to the reuse of code modules. Fig. 5-1 gives a wider approach to reuse: in addition to the code reuse there are good examples on the reuse higher level abstractions: designs and application frameworks

Reuse has two directions, which must be applied according to the company's reuse strategy (more detailed discussion in [9]). The effect of the reuse can be seen in the structure of the software: instead of natural solutions and structures the needs of the reuse are dominating the solutions. In the same time as a side effect, software structure becomes more standardized – i.e. will be based on predefined standardized solutions filling the expectations of some stakeholders (designers, implementers, testers). From client and end-user point of view the clarity and other good software properties are lost in the same time.

The effects of the selected *architecture style* have in a way a similar effect than reuse: it dominates structural solutions instead of the natural structure. As an example of the architecture style is MVC++ ([11] pp. 55-60). According to MVC++, three types of objects are separated: model, view and controller. The *model layer (M)* corresponds to a real world and "static" problem domain. The *view layer (V)* is the outer software layer visible to the end user. Typically there is one view class for each dialog box and window of the user interface. The *controller layer (C)* controls the interaction between the model and the view. The model layer objects usually appear in the analysis class diagram and the view components are derived from the user interface specification. Controller classes are needed to connect the "dynamic" view part of the system to the "static" model part. According to the software life cycle model *OMT++* the analysis class diagram (model layer) is produced in the *analysis phase*. The analysis object model is the basis of the design object model including classes closer to the implementation level. In this phase the class diagram may be restructured, where view components and respective controller objects are added to the model. Controller objects can be seen as adapters that integrate the model and view objects in an application specific way.

One of the ideas of using MVC++ architecture is to separate reusable classes of the application from the classes that implement application-specific functionality or that provide interfaces to the real world. The features of object technology – inheritance, dynamic binding, association and aggregation – are used to implement reusability. As in the case of reuse, the natural structure will be replaced by the rules supporting good design culture.

Product strategy is one important way to affect the policy adopted for the product releases, product variations and maintainability. A modern approach in this meaning is based on the products that inherit similar features of the product line and application platform (Fig. 5-2).

The common parts of the applications are organized into one independent high-level subsystem of its own. This subsystem is called an *application platform*. Application products depend on it (but not vice versa). Whereas application products provide applications to users, the application platform provides reusable components, frameworks and design guidelines to software designers. In addition, to manage reusable assets, the *application platform* also enables a *product line approach* to help system development in the future. The leading principle is to release a line of closely related products and product variants cost effectively over time (short time to market). The products are built on a common application platform that holds common software assets. The motivation to collect reusable software assets in an application platform is to make future variation easy and economical by using the results of projects in the past when creating new products. The difference from the application framework approach is that the new products are also created using the assets of the application platform. Like in the case of the reuse and architecture style, instead of natural structure the design factors are dominating the solutions.

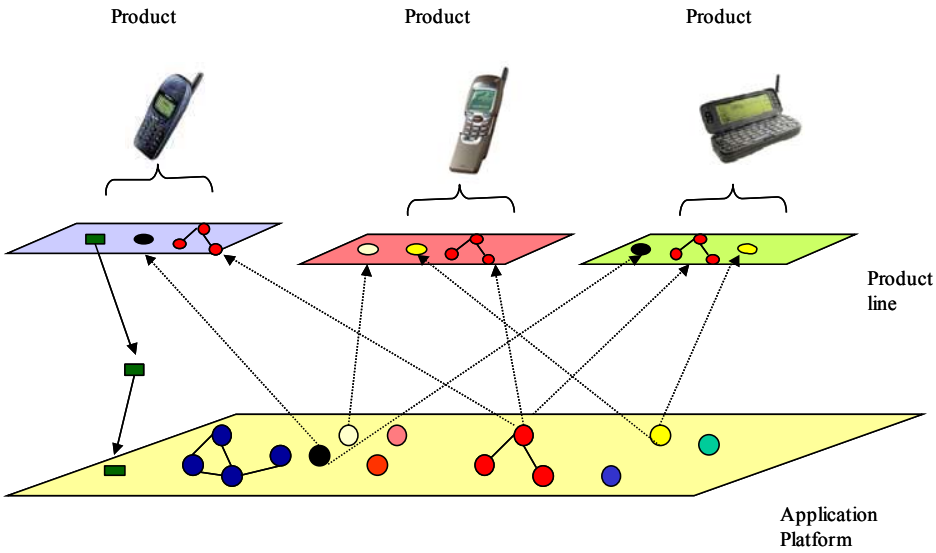


Fig. 5-2. Product line approach to the product development

As a consequence of the discussion above, the good design quality if the software can be specified to include such components that are supporting standard approaches to the solutions and support effective software development and maintenance practices. The solutions may be natural to the designers and help them to move from a project to another, because the design culture has established a common view for all designers. In addition, the implementation of the software may be supported by the good design solutions providing basis for the automation. Especially maintenance phase will get support from good designs.

6 Conclusions

6.1 Different quality views – a life cycle approach

Software quality is not a clear term: first of all the good properties are not always in favour and secondly the concept "good" has different meanings in different phases of the work. The paper has concentrated in three quality categories

- overall quality,
- specification quality
- design quality

The situation is clarified in Fig. 6-1.

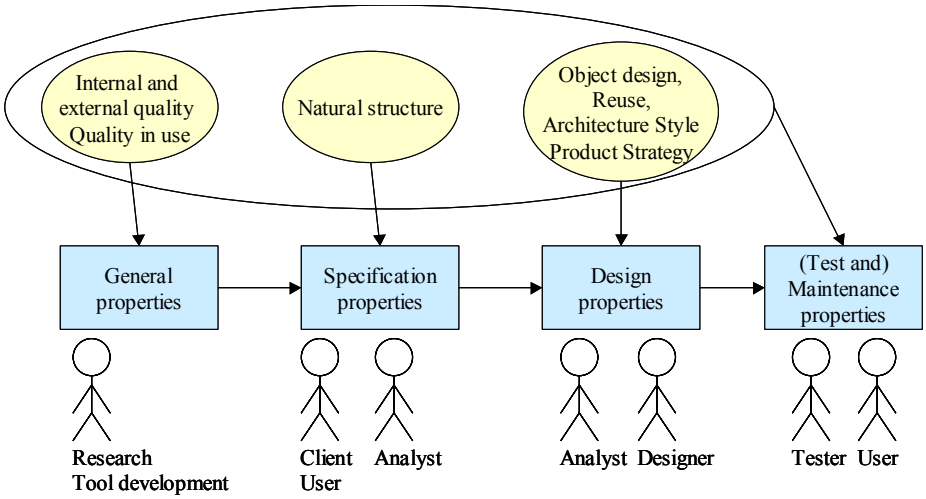


Fig. 6-1. Software quality aspects in different life cycle phases

In spite of having conflicting goals regarding the software quality in different phases, the well specified software process, including exact rules for documentation, requirements management, change management etc. have primary importance in beneficing on the good properties of the software. The maintenance phase represents 2/3 of the software life cycle costs. Because of that a good strategy is to support especially all the activities included in the maintenance. In thinking more closely the three different approaches to the concept "good software quality" it is easy to notice, that actually, it is not question of the conflicting factors but different focus in the same stock of properties. Making these different approaches to support each other is the key to the better success. Traceability in the documentation provides practical means to it. In the flow of different docus to the same design it is easy to notice, that the central stakeholder groups in different phases are varying. That what is important for the client is not important for the designer.

6.2 Further work – towards abstraction level maping

We outlined already that quality criteria are applicable on certain abstraction levels. On others they are not applicable. The abstraction layer model for database design has been introduced in [15]. This model consists of six layers.
Motivation layer: The stakeholder specification is the outcome of this layer. It coincides with the Requirements elicitation phase in the SPICE framework. Within this

layer, only some quality criteria such as suitability, learnabilty, attractiveness, and understandability are of interest.
The business process layer is used for specification of the processes as they should be seen by the owners of the enterprize. This layer contains the Requirements analysis layer in the SPICE framework. The main document is the system specification in which all properties of the software system are specified. The documents of this layer are verified against the requirements collected in the motivation layer. The layer is also used for the development of the systems architecture design and the distribution and modularization within the distributed information system.
The business user layer has been introduced for explicit representation of the application in the viewpoint of users in the operating departments. Within this layer, the data views are displayed in a form which is used in reality.
The conceptual layer mainly coincides with the system design layer in the SPICE framework. For information system applications The layer is used to describe the database schema, the database functionality, the view generation and maintenance mechanism and finally the story space with all possible discourses for utilizing the information system within the broad variety of application scenarios.
The implementation layer is used for the description of the logical and physical meta-information on the information system. It contains the engineering activities software construction, software and system integration, software and system testing, and software installation of the SPICE 2.0 framework.
The maintenance, evolution and utilization layer is not part of the conceptual design framework. This layer can be, however, supported by development steps of the co-design framework at the conceptual layer.
The first five layers have been discussed in detail in [15] where the co-design framework is entirely characterized.

We observe now that the other quality criteria can be classified similar to the motivation layer. Quality criteria such as suitability are mapped to either development obligations for the information systems development process or to quality criteria of the succeeding abstraction layers. Learnability is one of the very fuzzy criteria. It can be mapped, however, to appropriateness at the business user layer. Appropriateness directly leads to a number of implementation obligations that must be fulfilled during corresponding development steps at the implementation layer in the co-design framework. This approach has already been intentionally and implicitly used for the development of websites.

Learnability means in this case comprehensibility, i.e. easy to use, to remember, to capture and to forecast (In the Cottbus website development team this requirement set is characterized as "grand-mother simplicity".). It incorporates clarity of the visual representation, predictability, directness and intuitiveness. These properties allow the user to concentrate on the task. The work-flows and the discourse structure correspond to the expectations of the users and do not lead to surprising situations. They can be based on metaphors and motives taken from the application domain.

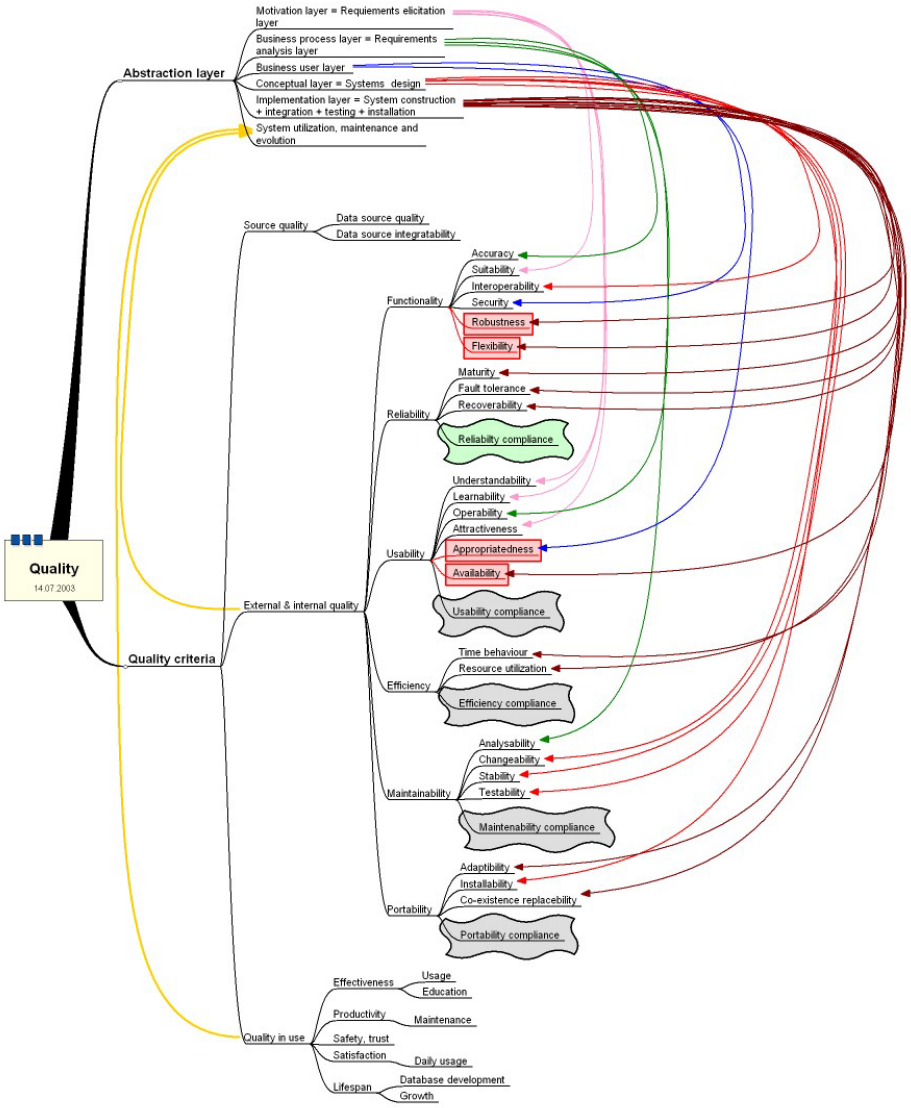


Fig. 6-2. Quality criteria abstraction levels

In the same way other quality criteria can be mapped to quality criteria or development obligations. Finally, the quality criteria are either entirely satisfied and the development obligations are fulfilled. This feedback cycle is also displayed in the mind map in Fig. 6-2. The reverse mappings direct directly to either the implementation or to the maintenance, evolution and utilization layers.

We are currently working out the mappings of quality criteria to other quality criteria and development obligations. These obligations rule the termination of development steps in the co-design framework. A step is considered to be successfully completed if all obligations for this step are worked off and the necessary documents have been completed.

References

1. Abrahamson, P., Salo, O., Ronkainen, J., Warsta, J.: Agile software development methods. Review and analysis (2002) http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf
2. Brooks, F.: There is No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer 20,4 (1987) 10-19
3. Fowler, M.: UML Distilled. 3rd Edition. Addison Wesley Publishing (2004)
4. ISO/IEC. IS 9126-1 Software Engineering - Product quality - Part 1: Quality model. ISO (2001)
5. ISO/IEC. TR 9126-2:2003, Product quality -- Part 2: External metrics. ISO (2003)
6. ISO/IEC. TR 9126-3:2003, Software engineering -- Product quality -- Part 3: Internal metrics. ISO (2003)
7. ISO/IEC. TR 9126-4:2004, Software engineering -- Product quality -- Part 4: Quality in use metrics. ISO (2004)
8. Jacobson, I.: Booch Grady, Rumbaugh James, The Unified Software Development Process, Addison Wesley Publishing (1999)
9. Jaakkola, H., Kukkonen, J., Varkoi, T.: Best Practices as Reuse Infrastructure. In Koloumdjian J., Mayr H., Erkollar A (editors), Proceedings of the ReTIS'2001 - Data and Document Re-engineering for the Web. Osterreichische Computer Gesellschaft, Vienna (2001) 9-31
10. Jaakkola, H. et al.: Experiences in Software Process Improvement with Small Organizations. In Hamza M.H. (ed.), Proc. of the International Symposium on Software Engineering, Databases and Applications. Innsbruck. IASTED (2002) 13-17
11. Jaaksi, A., Aalto, J-M., Aalto, A., Vättö, K.: Tried & True Object Development. Industry Proven Approach with UML. Cambridge University Press (1999)
12. Kruchten, P.: The Rational Unified Process: An Introduction (2nd Edition). Addison Wesley Publishing (2001)
13. Object Management Group (OMG), The Unified Modelling language Specification, V 2.0 (2005) http://www.uml.org/#UML2.0
14. Robillard, P. N., Kruchten P.: Software Engineering Processes: With the UPEDU. Addison Wesley Publishing (2003) http://www.upedu.org/uped
15. Thalheim, B.: Entity-Relationship Modeling - Fundamentals of Database Technology. Springer, Berlin (2000).
16. Upedu. Unified Process for Education. Overview. École Polytechnique de Montréal (2005) http://www.upedu.org/upedu/index.asp?TruY=861184239958042