Quality Issues

Jean Charles Salvin
jesi04@student.bth.se

Sebastian Stein
sest04@student.bth.se

Abstract

The term quality has many faces. In different product types different quality attributes might be considered to be important. The same problem might occur while changing the perspective between market-driven and bespoken software development. This report analyses this dependency. In this context it will be discussed if software is more like goods or services. It will be shown as well how to apply quality tools in software development, which are normally used for goods and services. Finally, the supply process in software development will be discussed taking quality issues into account.

1. Introduction

In a video game performance is one of the main quality requirement. For a controller in a car break system, reliability and safety are much more important than performance. This little example shows the importance of quality attributes depends on the system to be developed. Before discussing this subject, first a list with common software quality attributes is needed. It is presented in the next section as well as a discussion how the identified quality attributes differ from quality attributes of goods and services. In the following section quality attributes are called quality dimensions.

2. Quality Dimensions

Bergman et al. [1, p. 31ff] describe several quality dimensions for goods and services. They do not show how those quality dimensions relate to software.

By combining [7, p. 121ff] and [2, p. 89ff] the authors created a list with common quality dimensions used in context of software:

- Usability
- Performance
- Reliability
- Portability
- Interoperability
- Safety
- Security
- Maintainability

However, this list is not complete, someone can always add other software quality dimensions. The authors believe the list above reflects the most important software quality dimensions according to standard software engineering literature[1]. In the following sub-sections, the authors will discuss where the quality dimensions of software, goods and services differ and their similarities. In a final sub-section, a discussion is presented if software is more like goods or services in a quality context and what can be learned from more mature industries.

2.1. Comparison Software Quality and Quality of Goods

Some similarities can be found between software quality and quality of goods:

Usability: In software as goods, the complexity level shown to the user is essential. If it is too difficult, nobody will be able to use it. Quality in software resides in the fact, that it is easy to use or not.

Taking the example of a text editor software and a washing machine, similarities in context of usability can be found. In a text editor software, if there are too many options, buttons or if someone has to follow too many steps to get something done, the usability is considered to be poor. For the washing machine, if there are too many options or buttons, sometimes someone might also get lost. In both cases automatisms are needed, so that the user do not have to guide every action.

[1] e. g. [7]

Maintainability: For a software it is crucial that it can be maintained during the life-cycle. The same is true for a good. For a good it might be the case, that it must be changed during the life-time to adapt to changing laws like safety at work laws. In case of software it might also be needed to change the software according to altered tax laws.

Performance: Performance is important for goods as well as software. Often the user judges the overall quality of a software by his performance experiences with the software. A more performant software is considered to have higher quality. The same is true in case of goods. For example a copier copying 20 copies per minute will be considered to have a higher quality then a copier with just 5 copies per minute.

Besides similarities, there are also some differences:

Appearance: For a good the appearance is an important factor, since some customers will base their buying decision on the appearance of the product. In software appearance is normally not considered. Of course, the graphical user interface has to follow certain rules, but this is more an issue of usability, since all applications should follow the same usability guidelines.

Environmental impact: There is a big difference of impact between software and a good on the environment. Normally a software or the production of a software has no impact at the environment at all. In contrast, the effects to environment by goods and their production processes must be considered.

Flawlessness: Since goods are often not as complex as software, it is possible to release them without any flaws. There are well known techniques available to check a product for quality problems. The same is not true for software. It is nearly impossible to release a software without bugs, even in simple applications. The authors experienced this several times by themselves.

2.2. Comparison Software Quality and Quality of Services

Similarities can be found between software quality and quality of services:

Tangibility: When comparing tangibility of software and services, it can be stated that both are not tangible at all. It might be possible to touch a consultant, but the consultant is not the service, because the service is the task performed by the consultant. In software, this is similar as it is impossible to take software in hands. It is possible to touch the computer the software is running on, but not the software.

Communication: One has to communicate with a software as well as with a representative of a service. In software user interfaces are used. User and interface must use the same language, for example it is impossible for an user to understand binary code. In services high quality communication is a key success factor.

Reliability: Software has to be reliable. It has to be available (no crashes), the performance should not decrease during runtime and the same input should always lead to the same output[2]. A service has to be also reliable, the service must be done continuously and as defined in the contract.

Besides similarities, there are also some differences:

Access: A service must be easily and every time accessible. Unfortunately this might not be possible, because a consultant might be on holiday or unavailable. In contrast a software is always accessible. It can be launched on user request.

Credibility: A service is provided by human beings and companies. Someone can trust a human. Also a human being can take responsibility. A software can not, because a software can not decide how to perform an action. The execution is determined by the source code. Therefore it is impossible for a software to act for credibility.

Courtesy: As the previous statement, there is a difference between a supplier courtesy and a software courtesy. Nobody expects a software to be polite or not. This is not an attribute of a software.

2.3. Summary

In the previous sub-sections, the authors compared software quality with quality of goods and services. There are similarities, but also major differences. A discussion if software is in context of quality more like goods or services is meaningless. Someone has to accept software is something different. But that does not mean to not take best practices for quality of goods and services into account. For example the 7 quality tools [1, see p. 216ff] can be applied in software development as shown in section 6 on page 4.

[2] e. g. $1 + 1 = 2$ for a calculator

3. The 4 Quality Movement Phases in Context of Software Development

Bergman et al. [1, p. 91ff] explain the quality movement with 4 phases. The phases take place at different stages in the development process (at the end, during, before, . . .). It is possible to find a similar mapping for software engineering as well. Nevertheless, it must be clearly stated, that this mapping is based on a subjective interpretation.

To be able to do this mapping, a software process must be used for referring. The authors have chosen the classical waterfall approach [7, p. 66ff] for this section. This process consists of the phases analysis, design, implementation (production), and test.

One of the 4 quality movement phases is quality inspection. Quality inspection is done at the end of the production step. In software development, the code tests and integration tests are usually done after the implementation step in the test phase.

Another quality movement phase is quality control. This phase is done during production to identify first signs of possible defects and fix them immediately. To prevent coding bugs in software development, one can use coding standards and structured or object-oriented programming techniques.

Quality assurance tries to prevent problems and quality flaws even before production starts. This can be found in software development as well. For example one can apply test-driven development. Here the unit tests are written first before the implementation is done. Another possibility to ensure high quality even before implementation is to use requirements engineering [5, see e. g.] to elicit the user's needs and to keep track of those requirements and possible changes during the rest of the development project.

The final phase of quality movement is to have a continuous process. This phase is called quality management. In software development quality models like CMM and ISO 9000 try to ensure steady improvements throughout the whole software process.

4. Quality in Mass-market and Bespoke Products

Bergman et al. focus in their book [1] mainly on quality in mass-market production. Nevertheless, quality issues must be also considered in bespoken projects. Here quality is as important as in mass-market products.

There are some differences dealing with quality issues in both product types. For example it might be easier to exactly define the required quality level for the different attributes in a bespoke project. Indeed a close interaction with the customer is possible. In a mass-market production, the customer can mostly not be consulted directly. Thus, the software vendor has to make assumptions about the needed quality level. Using techniques like market analysis and customer surveys can improve this problem.

Releasing a low quality product to the mass-market may have bigger drawbacks then releasing a low quality product to a single customer in a bespoke project. Indeed, the information about the low quality of the product might spread quickly through the market, whereas a single customer may not be able to communicate the quality problems to many other customers. In our days, quality problems of market products are reported in media like internet, television or radio. If for example a car producer has to call back cars, this is often reported in daily news. Webpages present the opinions of several people about certain products. For a potential customer, it is easy to check what other people think about the product in general and in context of quality.

The required levels for the different quality attributes might differ if the same product is developed for mass-market and for a single customer. In mass-market, it is important to have certain basic level for all quality requirements. So the software will be experienced as high quality by many customers. In a bespoken project, the software can be optimised to have a highly level of quality in some quality attributes like performance or reliability. The development can focus on the quality attributes mostly demanded by the customer. This means that in a bespoken project, trade-offs between different quality attributes can be done depending on the customer's agreement. In a mass-market project, this is much more complicated, because different users will have different needs on a single product. For example, the authors see security as a crucial qualitative requirement for operating systems, but other users might pay more attention to performance or usability.

5. Quality in different Software Systems

The importance of quality attributes is different depending of the developed system. In the following sub-sections, 3 different examples will be presented. The examples are used to discuss the differing importance of quality attributes.

5.1. Example 1: End-user Desktop Application

In case of an end-user desktop application like a CD-ROM burning application, an encyclopedia or an Internet browser, the following quality attributes are most important in the authors' opinion:

- Usability
- Interoperability

- Security

This type of software is used by many different users, who have different knowledge and skills. However, everybody should be able to use the software. This implies that the usability of the software must be very good.

Interoperability seems also highly important, so that the outputs produced by one application can be used in another one[3]. Typically, a desktop environment consists of several applications. The combination of those applications give the user a powerful tool to solve his daily work.

As many inexperienced people are using this type of software, it is important that the software provides a certain level of security. Indeed, end-users do often not have the necessary knowledge to deal with security issues. People are often managing confidential data[4] with such applications. It is in the very interest of the user, that this data is secured.

5.2. Example 2: Video Game

Even if video games are mostly used by the same user group as described in the section above, the most important quality attributes are other ones:

- Performance
- Maintainability
- Usability

Video games should be fun to use and this means that they have to be very fast. Also it is important that the game also runs on slightly[5] outdated hardware.

After a video game gets released, often updates must be deployed to fix bugs. Besides fixing bugs, the software vendor might also like to add new functionality[6], if a game turns out to be very successful. Therefore it is crucial that the game can be easily maintained.

Usability remains also a key point in the authors' opinion, since it should be possible to use a video game without reading the manual. This also means the game must be easy to install and maybe some tutorial missions must be provided.

5.3. Example 3: Embedded Control System

Embedded control systems are software integrated to machines. Common examples are software integrated into cell phones, washing machines and process controllers in a (chemical) factory. The authors think the following quality attributes are most important:

- Reliability
- Safety
- Security

This type of software is very much related to common life. The infrastructure of a society relies on this type of software, e. g. electricity, water or gas. There is normally no direct end-user for this software. Nobody is aware of the software as long it is working properly.

Reliability and safety quality seem obvious as they affect humans' common life. Humans rely on this kind of software for their safety. For example, if while driving a car the breaks are not functioning because of a software bug, the driver has a serious problem.

In recent times security became another important quality attribute for this kind of software. Societies rely so much on such software, that such systems are a valuable target for terror attacks. Of course everything must be done to prevent such attacks.

6. The 7 Quality Tools in Software Projects

The 7 quality tools are described in Bergman et al.'s book [1, p. 216ff] chapter 10. In this section the authors show how the presented tools can be used in software projects. The authors never saw those tools applied in their work experience, but they think it might be valuable to use the presented tools.

6.1. Data Collection

The 7 tools presented by Bergman et al. are statistical methods. Before a statistical method can be used, data is needed. Therefore the first tool [1, p. 217ff] collects the necessary data and provides it for further investigations.

Data can be collected during all phases of software development. It is possible, for example, to collect the following data:

- number of new and changed requirements
- value of fulfilled/implemented requirements
- number of classes, components, functions, etc.
- number of identified and fixed bugs (bug reports)
- number of failed test cases
- number of produced and changed lines of code
- numbers to measure project progress (see [6, p. 340ff])

[3] e. g. copy-paste of text between different application
[4] e. g. online banking
[5] hardware not older then 2 years
[6] e. g. expansion packs, new scenarios, new campaigns

It is important to only select the necessary data. Here it is a great advantage, if the data can be exported from tools already used during software development, because then not much extra effort is needed for data collection. The following tools might provide data:

- requirements engineering databases
- CASE tools or UML tools
- bug tracking database
- test framework
- version control system
- project management tool

6.2. Histograms

Histograms [1, see p. 220ff] are used to group data into categories. This is useful, if the analysis of all data points is not possible, because of the amount of the data. In such cases it is interesting to group the data, e. g.:

- number of change requests per week
- number of fixed bugs per week

It is also possible to create groups for different parts of the developed system:

- number of identified defects per component
- number of changes to certain classes (an often changed class might be too huge or may have a bad design [4, idea is presented here])

6.3. Pareto Charts

Pareto charts [1, p. 222ff] are a tool to visualise the current biggest problem in a project. This can be done by first listing all problems and their estimated costs/impacts in a pareto chart. Project management should try to first first solve the problem with the biggest associated costs.

It may be also possible to use pareto charts to decide which requirement should be implemented first. To do this, someone has to put the requirements with their relative profits in a pareto chart. The relative profits can be calculated using the following formula:

$$relative\ profit = earnings\ for\ requirement/costs$$

If possible first the requirement with the highest relative profit should be selected for development, if no other constraints have an influence on the decision. A problem might occur, if the requirements are not on the same level, e. g. if one requirement would cause costs of 10.000 SEK and another 100.000 SEK. Then a comparison as described above is not meaningful.

6.4. Cause-and-effect Diagrams

It is the task of a software developer to find solutions for given problems. Therefore, software developers have already knowledge to identify the problems' cause. Nevertheless, the use of structured techniques like cause-and-effect diagrams [1, p. 224ff] might be powerful tools in some cases.

Assuming there is a connection problem between a client and a server, which are not located on the same machine. There can be several causes for that, e. g. a broken physical network connection, server not listening to the interface the client is connected to or the client might not have enough access rights to connect to the server.

Those possible causes can be drawn in a cause-and-effect diagram. Furthermore, it is possible to decompose the causes in sub-causes. Someone working to fix this problem can use the cause-and-effect diagram to identify the actual cause in a structured way. It is also possible to provide cause-and-effect diagrams as a general work instruction for service personal. A service worker can check the most common reasons for a problem by having a quick view at the cause-and-effect diagram.

6.5. Stratification

A histogram[7] might show that during the last week, several new bugs were reported. It might be interesting to find out, if most of the bugs were identified in a certain module, component or sub-system, because this can mean a bad design of this part of the system. Therefore the corresponding histograms of the different modules, components or sub-systems must be compared. Bergman et al. [1] call this change of the abstraction level *stratification*. The authors have seen similar techniques in decision support systems. There the action of changing the abstraction level is called drill-down.

6.6. Scatter Plots

The use of scatter plots [1, p. 230f] could be questioned in context of software development, because it is only useful if the value of a variable depends on just one single factor. Software development is a complex task and in the authors' opinion it is not possible to build such simple relationships. However, in some special cases or on a highly abstraction level, it might be possible to use the technique. For example one can plot the dependencies between found defects and number of inspections. More inspections should identify more defects, but at some point an asymptote might be reached and more inspections do not reveal more defects.

[7] see section 6.2 on page 5

6.7. Control Charts

As shown in section 6.1 on page 4, it is possible to collect a lot of different data during software development and to create quality indicators based on the data. By adding an upper and lower control limit, one creates control charts [1, p. 231ff]. Control charts are useful as an indicator if something is going wrong. A control chart only shows that there might be a problem, but it does not show where the problem is. Other techniques and tools must be used to investigate the problem further. Control charts are a very useful tool for project managers to monitor the overall status of a project. However, since the control charts can only analyse measure-able data, the project manager has to also monitor the soft-factors of a project like state of the team spirit, customer satisfaction, commitment, etc.

7. Supply Process in Software Development

Supply Chain Management is usually defined as a network of facilities to provide the following functions: materials procurement, transformation of those input materials during production into products and distribution of the products to the customers [3, see for more details].

Analysing this definition, the following 3 parts of the supply chain can be identified: procurement, production, and distribution.

Bergman et al. [1, p. 283] focus in their description only on the procurement part of the supply chain. They describe how to design the relationship with the supplier and what techniques should be used to check the quality of supplied components.

It is not clear to the authors, if it is meaningful to define a supply chain for software development. For example it must be asked what a software vendor can procure before software development starts. There is also no explicit production step in software development. The distribution of software can be easily done by using common sale channels and general marketing methods.

For procurement, the authors think that it is today common to acquire specifications. The authors experienced that in bespoke product development, a consulting company formulates the specification of a future software product. This specification is used by the software vendor to extract the requirements and to develop the software accordingly. The consulting company often has in such cases only a relationship to the customer and not to the software vendor. Of course it would be beneficial, if the software vendor can design a long-term relationship with the consulting company, so that the software vendor is often chosen to realise a specification.

Using external resource might be possible during the implementation phase of software development. For example, the implementation can be completely handed over to a company in a low salary country. This activity is called outsourcing. To do successful outsourcing, it is important to have a good relationship with the contract company. The communication between both companies must be ensured. It might also be beneficial, if software vendor and contract company use similar design methods, documentation standards and software processes. Outsourcing can create a big risk, if the outsourcer is not doing high quality work. In this case the software vendor has to pay for the quality problems produced by the outsourcer.

Another possibility during the implementation step is to use commercial off-the shelf (COTS) components [7, p. 429ff]. Those components can be used instead of developing the functionality by oneself. Typical widely used COTS are database servers, application servers, and class libraries.

Nevertheless, using COTS can be problematic in context of quality [7, p. 431]. The software vendor can not control the development process of the COTS. He must rely on quality checks like tests after the COTS were released.

It might be possible to solve this problem with having a close relationship to the COTS vendor, but it might not be possible in all cases to build this relationship, since COTS are often mass-market products.

8. Conclusions

Software is different from goods and services. Nevertheless, there are some similarities. The needed software quality attributes differ for bespoke and market-driven development. The same is true for different kinds of systems. It is possible to apply statistical quality tools during software development and the supply chain concept can be used in software development.

Literatur

[1] B. Bergman and B. Klefsjö. *Quality: from Customer Needs to Customer Satisfaction*. Studentlitteratur, Lund, 2003.

[2] J. Bosch. *Design & Use of Software Architectures: Adopting and evolving a product-line approach*. Addison-Wesley, London, 2000.

[3] S. Chopra and P. Meindl. *Supply Chain Managment: Strategy, Planning and Operation*. Prentice Hall, 2001.

[4] S. Diehl. Softwarevisualisierung. *Informatik Spektrum*, 26(4):257–260, 2003.

[5] G. Kotonya and I. Sommerville. *Requirements Engineering: Processes and Techniques*. Wiley, Chichester, 2004.

[6] J. M. Nicholas. *Project Management for Business and Engineering*. Elsevier Inc., Burlington, MA, 2004.

[7] I. Sommerville. *Software Engineering*. Pearson, Boston, 7th edition, 2004.