Considerations and Proposals for Quality Engineering in the Context of SQuaRE*

Tsuyoshi Nakajima[1,*,†] and Kengo Morimoto[2,†]

[1] Shibaura Institute of Technology, 3-7-5 Toyosu, Koto-ku, Tokyo, Japan
[2] NEC Corporation, 5-7-1 Shiba, Minato-ku, Tokyo, Japan

Abstract

The SQuaRE family defines models, measurement, requirements definition, and evaluation of quality, gaining recognition in the field of system and software quality assurance. However, its utilization among software designers has not progressed. This is attributed to a tendency for designers to perceive quality in a narrow sense, neglecting a broader range of quality characteristics, and the requirement for advanced knowledge and techniques. This paper discusses the peculiarities of realizing quality requirements in design compared to functional requirements, highlights the importance of quality engineering, and proposes techniques from three perspectives: the deployment of quality requirements into design, selection of design alternatives and traceability, and the design of processes for realizing and validating quality.

Keywords

quality engineering, quality realization strategies, trade-off analysis, risk-based planning, SQuaRE

1. Introduction

The international standards related to system and software quality in the ISO/IEC 25000 series are referred to as the SQuaRE family, which has been gaining recognition in both industry and academia. The SQuaRE family defines what quality is (2501n), how to measure it (2502n), how to define requirements (2503n), and how to evaluate it (2504n).

The standards in the SQuaRE family are primarily utilized by quality assurance engineers and organizations. Although these standards hold significant value for designers as well, their current level of recognition and utilization among them remains low. This can be attributed to a tendency among software designers to focus on achieving functional requirements and solving technical challenges, often limiting their perception of quality to the absence of program crashes or unmet functional specifications, without considering the broader range of quality characteristics such as reliability, maintainability, and performance efficiency.

Furthermore, two interrelated aspects must be considered:

1. Realizing quality requirements through design and verification necessitates advanced knowledge and skills, which are currently insufficiently accumulated and shared.
2. Estimating the cost of non-quality is challenging, as it becomes diffused during system operation, with malfunctions, anomalies, or security vulnerabilities requiring ongoing maintenance.

Against this backdrop, quality engineering is emerging as a critical technical domain for deploying and validating quality requirements in system and software products [1]. A project for the international standardization of quality engineering (ISO/IEC /IEEE 25070) is currently underway within the SQuaRE family of standards.

This paper discusses the peculiarities of realizing quality requirements in design compared to functional requirements, highlights what is required from quality engineering, and outlines considerations for design and coding in terms of the deployment of quality requirements into design, selection of design alternatives and traceability, and the design of processes for realizing and validating quality, proposing core techniques for each aspect.

2. Quality engineering

2.1. Quality requirements and challenges in their implementation

Quality requirements have two distinct characteristics compared to functional requirements, making their implementation and verification in design considerably challenging:

1. **Quality has emergent properties.** Emergent properties refer to the phenomenon where system-wide characteristics or functions manifest in ways that cannot be predicted from individual components (modules or components) alone [2]. Due to this emergent nature, planning how to implement quality requirements and determining when and how to verify them is significantly more

IWESQ'24: International Workshop on Experience with SQuaRE Family and its Future Direction, December 03, 2024, Chongqing, CN
* Corresponding author, † These authors contributed equally.
tsnaka@shibaura-it.ac.jp (T. Nakajima);
kengo-morimoto@nec.com (K. Morimoto)
0000-0002-9721-4763 (T. Nakajima)

complex than for functional requirements. Additionally, establishing explicit traceability links to components or modules, which are the constituent elements of software, becomes difficult.

2. **Quality satisfaction is a matter of degree.** Although specific quality requirements can have set criteria that must be met, these criteria often cannot be rigid. Consequently, it is often necessary to carefully balance the satisfaction levels of different quality requirements, which may vary in priority, and to make design choices accordingly. This characteristic makes the process of implementing quality requirements more complex than functional requirements and complicates the traceability to design solutions, as simple associations are insufficient to represent the relationships.

2.2. Requirements for quality engineering

Based on the unique characteristics of quality requirements outlined above, the following requirements are essential in system and software quality engineering:

- **R1**: The design solution shall comprehensively deploy the quality requirements.
- **R2**: Bidirectional traceability shall be established and maintained between the quality requirements and the design solution.
- **R3**: The design solution shall be verified to ensure that the quality requirements are realized.

3. Deployment of quality requirements into design

3.1. Considerations

There are four main approaches to achieving quality requirements (R1) in the design of a target entity:

1. **Adding functionality to the target entity:** this approach involves incorporating specific functions to meet quality requirements. Eckhardt et al. [3] reported that over 30% of quality requirements are met through functional implementations. Examples include adding confirmation dialogs for error prevention, logging functions for analyzability, and access control functions for confidentiality.
2. **Utilizing architectural structures of the target entity**: architectural structures can address quality requirements through various forms—physical placement, dynamic behavior, and static code structure. These structures are often assessed in terms of physical connectivity and isolation, behavioral qualities, and code quality. Architectural solutions for quality requirements have been developed as design knowledge, including architectural implementation strategies, architectural patterns, and design patterns.
3. **Assigning quality requirements directly to substructures of the target entity**: this is less common as many quality aspects lack emergent properties, making this approach suitable for specific cases. For example, in system reliability engineering, the overall system reliability can be calculated from the reliability of individual components if the arrangement (serial or parallel) is known. Similarly, the response time for a particular function (related to time efficiency) can be derived from the execution time of the involved components.
4. **Defining design rules and guidelines for detailed design and coding**: establishing guidelines for design and coding can help achieve quality requirements. For instance, setting guidelines for error messaging can enhance analyzability, creating unified user interface guidelines can improve learnability, and following secure coding rules can enhance security.

3.2. Proposal

The methods for implementing quality requirements in design have been crafted through the ingenuity and experience of skilled designers. Particularly, the second and fourth methods mentioned above involve a substantial amount of expertise and practical know-how. These methods should be organized and accumulated as design knowledge that can be readily accessible to designers.

We propose structuring this design knowledge according to each quality subcharacteristic in a product quality model and offering it in the form of a "Quality Realization Strategy Catalog," including the following items:

- **Strategy**: categories of implementation methods to achieve quality requirements architecturally.
- **Implementation method**: specific techniques and solution patterns for achieving quality requirements.
- **Description**: explanation of each implementation method.
- **Pros, cons, trade-offs, and prerequisites**: advantages that support the method's use, potential drawbacks, trade-offs between benefits and drawbacks, and prerequisites for application.

Table 1 presents an example of a quality realization strategy catalog focused on testability. This example organizes content described in [4] into the proposed catalog format.

By expanding the quality realization strategy catalog, as shown in Table 1, designers can effectively search for strategies that align with quality requirements from the catalog. They can consider the advantages and disadvantages of these strategies before applying them to achieve the required quality in the product. This approach eliminates the need to devise design solutions from scratch, reducing the dependency on the designers' skills and thereby lowering the probability of implementation failures.

4. Ensuring trade-offs and traceability of design solution

4.1. Considerations

Functional requirements can be realized through the decomposition of components and the assignment of responsibilities to them. Therefore, the process of translating these requirements into design and ensuring traceability between the components is relatively straightforward. In contrast, quality requirements are more complex and challenging due to their emergent nature.

Typically, there is a many-to-many relationship between quality requirements and design solutions. This

Table 1
Example of quality realization strategy catalog for testability requirements

Strategy	Implementation method	Description	Pros and cons, trade-offs, and preconditions for use
Manage input/output	Record/playback	Capture information passing through an interface and use it as input for the test harness. Information passing through the interface during normal operation is usually stored in a repository. This recorded information can be used as input for testing a component or for differential comparison during retesting.	**Pros:** - Makes it easier to test and verify interfaces. - Can be mechanically deployed. **Cons:** - May cause performance degradation. - Recorded data may increase. **Preconditions:** - Cannot be used for components requiring precise timing, as recording may alter processing timing.
	Separation of interface from implementation	Separate interfaces from implementations to allow swapping of implementations for various tests. If a component has defects, replace it with a stub to continue the remaining tests.	**Pros:** - Using object-oriented development languages or AOP, DI frameworks make this relatively easy as there are established methods. **Cons:** - Implementing this with pre-object-oriented languages (procedural languages) is costly.
	Specialized access routines/interfaces	Use special test interfaces to capture or specify component variables and specifications independently of normal execution through the test harness.	**Pros:** - In some development languages, debugger functionalities can implement this, making it easy to apply. **Cons:** - Differences in behavior when test interfaces are enabled versus disabled may require increasing test patterns. - Costs are high if development languages do not support test interfaces.
Internal monitoring	Built-in monitoring	Incorporate functionalities to monitor internal states to support testing.	**Pros:** - Can monitor difficult-to-test internal states, performance, and resources (memory, etc.). - Development languages like Java, MS .Net Framework provide monitoring tools. **Cons:** - May require testing with monitoring both on and off, increasing costs. - Costs are high if implementing through custom development.

means that a single quality requirement can be fulfilled by multiple design solutions, and a single design solution can potentially address multiple quality requirements. As discussed in Section 3, the realization of quality requirements through architectural structures involves multiple design solutions that satisfy both functional requirements and (at a minimum) quality requirements. This necessitates a complex selection process among the multiple design solutions.

In the selection of design solutions, various quality requirements, often with differing priorities, come into play. The selected design solution must achieve a reasonable balance of satisfaction across these quality requirements while also demonstrating superiority over other design solutions.

The process of selecting design solutions—specifically, which quality requirements are relevant (even if some are left unaddressed), how their satisfaction levels are evaluated, and how decisions (compromises) are made—is a crucial basis for design rationale. However, it is common for the design artifacts to retain only the description of the selected design solution, while such design rationale is often not documented. This rationale is important information for maintaining the product and should ideally be traceable.

4.2. Proposal

The realization of quality requirements is characterized by a many-to-many relationship. Therefore, for each quality requirement, it is necessary to consider design solutions that fulfill that requirement, evaluate their cost-effectiveness, and make decisions regarding their adoption. Conversely, significant design solutions that pertain to the fulfillment of multiple quality requirements should undergo trade-off analysis based on evaluations of their satisfaction levels.

Table 2
Example of quality trade-off analysis

				Design Solution		
				Package software	**Package software with customization**	**Cloud service**
Quality Requirements						
ID	**Quality sub-characteristic**	**Description**	**Importance**			
QR-01	Functional completeness	Covers all existing functions	**Critical**	**FM**	**FM**	**LM**
QR-02	Functional correctness	No errors in displayed data such as position, quantity	**Major**	**LM**	**FM**	**FM**
QR-03	Time behavior	All displays appear within 3 seconds	**Major**	**LM**	**LM**	**LM**
QR-04	Interaction capability	Screen operation and content do not cause operator mistakes	**Minor**	**PM**	**LM**	**FM**
Constraints						
ID	**Description**					
C-01	The total development cost shall not exceed [specified amount].			**FM**	**DM**	**FM**
C-02	Software licensing and third-party service costs must not exceed [specified amount] annually.			**FM**	**FM**	**FM**
Comprehensive Evaluation				**X**		

Key
The degree to which each design solution satisfies the quality requirements and constraints is:
FM (Fully Meet) - The design solution fully satisfies the quality requirement or constraint.
LM (Largely Meet) - The design solution largely satisfies the requirement, with minor gaps.
PM (Partially Meet) - The design solution meets the requirement to some extent but has significant gaps.
DM (Doesn't Meet) - The design solution does not satisfy the requirement or constraint.
As a result of the comprehensive evaluation, an "X" is assigned to the selected design solution.

Furthermore, these analyses should be documented as part of the traceability of quality requirements within the design artifacts (R2).

Table 2 presents a proposed quality trade-off analysis table. This table compares three different design approaches for system construction (Package Software, Customized Package Software, and Cloud Services) across multiple quality requirements.

The rows of the table consist of three sections: Quality Requirements, Constraints, and Comprehensive Evaluation.

The Quality Requirements section comprises the following:

Table 3
Example of risk-based quality engineering plan

Risk identification				Risk analysis		Risk mitigation plan				
Target	**Quality sub-characteristic**	**Quality requirement**	**Risks**	**Sever-ity**	**Impact on rework**	**Basic strategy**	**Risk treatment**	**Cost**	**Effec t**	**Deci-sion**
Delivery status monitoring function	Functional completeness	Covers all existing functions	Users may not be able to use expected functions, potentially leading to complaints and possible reputation damage.	H	-	Improve test coverage	Ensure traceability to existing functional specifications and thorough review	M	H	Accept
		Can operate with current data	Operational instability may occur, potentially causing major operational stoppages.	H	-	Improve test coverage	Conduct connection tests	M	H	Accept
	Functional correctness	No errors in displayed data such as position, quantity	Incorrect information may lead customers to make wrong decisions, potentially resulting in a loss of customer trust.	H	-	Early V&V	Improve test coverage	M	H	Accept
							Introduce viewpoint reviews	L	M	Accept
							Use coding standards	L	M	Accept
	Interaction capability	Screen operation and content do not cause operator mistakes	Frequent operational errors may lower work efficiency, and operational mistakes may lead to losses.	M	H	Early V&V	Develop prototypes for specification adjustment	H	H	Accept
						Improve test coverage	System testing: verification by experienced testers	M	H	Accept
	Time behavior	All displays appear within 3 seconds	Delays in system response may reduce operator work efficiency, potentially hindering operations and lowering customer satisfaction.	M	H	Improve design quality	Use timing charts	L	M	Accept
							Use coding standards	L	M	Accept
						Improve test coverage	Introduce viewpoint reviews	L	M	Accept
							Include internal and external expert reviews	H	M	Reject
						Early V&V	Iterative development and performance verification	H	H	Accept
System management subsystem	Functional correctness	Correct and timely (within 5 seconds) system state awareness	System malfunctions or issues may not be detected in time, potentially leading to significant trouble.	H	H	Early V&V	Construct a simulated operational environment	M	H	Accept
						Improve test coverage	Introduce viewpoint reviews	M	H	Accept
							Improve coverage in complex conditions	M	H	Accept
	Fault tolerance	24-hour operation, availability above 99.94%	Frequent system downtime or stoppages may severely impact operations, potentially resulting in a significant loss of customer trust.	H	H	Early V&V	Construct a simulated operational environment	H	L	Reject
						Improve design quality	Employ dual server systems and availability design	M	H	Accept

Key High (H), Medium (M) and Low (L)

- **ID**: a unique identifier assigned to each quality requirement, facilitating easy reference and maintaining consistency.
- **Quality subcharacteristics**: the specific quality subcharacteristics targeted by the quality requirements.
- **Description**: a concise and clear description of the content of the quality requirement.
- **Importance**: the level of importance assigned to each quality requirement. The general levels are as follows:
    - **Critical**: essential for the success of the design solution, requiring full compliance.
    - **Major**: important, but flexibility is allowed, and it should generally be satisfied.
    - **Minor**: desirable but not mandatory; partial satisfaction is acceptable.

The Constraints section consists of:

- **ID**: a unique identifier assigned to each constraint, making it easier to refer to specific constraints in analysis and documentation.
- **Description**: a concise explanation of each constraint.

The design solutions are arranged in columns, indicating the extent to which each design solution satisfies the quality requirements and constraints.

The Comprehensive Evaluation represents the final assessment based on the overall performance of each design solution concerning all quality requirements and constraints. This row is used to identify the most balanced design solution, ensuring that design solutions that do not meet the constraints are not selected.

By creating a quality trade-off analysis table like Table 2, it becomes possible to choose the design solution that meets the constraints while achieving the best balance of quality characteristics. Documenting the decision-making process and its rationale ensures that trade-offs among various quality characteristics are recorded, making it easier to reconsider or adjust designs when quality requirements or system constraints change. Establishing traceability between design solutions and quality requirements in this way enhances the adaptability of the system and reduces the risk of unintended impacts during modifications.

5. Process design for achieving quality requirements

5.1. Considerations

In development, achieving product quality is the top priority. To ensure this achievement, it is necessary to design a process that focuses on quality engineering planning. Quality engineering planning involves planning the implementation of appropriate quality engineering strategies and methods within the right development process to realize the key quality requirements for the product.

The realization of quality cannot be guaranteed solely by the availability of design knowledge presented in Section 3. In quality engineering, activities are required not only to create quality but also to verify that the development process meets the quality requirements. Verification activities include prototyping to confirm the validity of the requirements and to ensure that the implementation can realize those requirements, reviewing or performing static analysis on the static structure of deliverables, and conducting functional and non-functional testing.

5.2. Proposal

The realization of quality requirements and the reduction of development costs and timelines are significant trade-off considerations in development environments. It is essential to focus on the risks associated with failing to meet the required quality demands of a product when selecting the best strategy. Table 3 illustrates an example of a quality engineering (QE) plan aimed at fulfilling quality requirements.

Therefore, a risk-based approach to quality engineering planning is proposed.

Table 3 illustrates an example of a quality engineering (QE) plan aimed at fulfilling quality requirements. Assessing both the severity of impacts on users and society if quality requirements are not met, and the effects of potential rework if risks materialize, is essential. This clarifies the priority of addressing specific risks.

a) **Identification of QE goals for the target entities**: in this example, the QE goal is to meet the quality requirements.
b) **Definition of QE requirements**: first, subjective points that should be verified as inspection targets are listed, including scenarios in which the system will be used, main functions, and subsystems that make up the system. Then, the critical quality characteristics for inspection targets are listed, with defined metrics and target values for each characteristic. These serve as the quality requirements in this QE example.
c) **Identification and analysis of QE risks**: risks are evaluated from two perspectives: the severity of potential issues in the operational environment if quality requirements are unmet and the scope of rework required.
d) **Mitigation of risks**: based on these assessments, the following core quality engineering strategies and measures are selected to address risks:

- **Improving design quality**: by thoroughly executing the design process, quality can be enhanced and variability reduced. Examples include using design modelling, and patterns, as well as involving external experts.
- **Increasing test coverage**: compared to other inspection aspects, focus on increasing review density and test coverage (in terms of perspectives, elements, levels, and combinations). Examples include implementing perspective reviews, using static analysis, rigorous boundary testing, and utilizing test combination techniques.
- **Early V&V (Verification and Validation)**: perform verification and validation as early as possible. Examples include using prototype models, introducing scenario reviews, and expediting tests with real data.

If the impact severity is high, measures should be taken to both prevent the introduction of defects related to the quality in question and detect introduced defects, addressing both design quality improvement and test coverage enhancement. Additionally, if the impact of rework is substantial, early V&V is implemented to confirm achievement of the relevant quality early on.

6. Conclusion

This paper addressed the unique challenges in implementing quality requirements in design, as compared to functional requirements. Specifically, it highlighted the emergent nature of quality and the degree-based assessment of satisfaction levels. Considering these characteristics, we identified issues within system and software quality engineering from three perspectives: the deployment of quality requirements into design, selection of design alternatives and traceability, and the design of processes for realizing and validating quality.

As concrete solutions, we propose quality realization strategy catalogs, quality trade-off tables for design solution selection, and risk-based quality engineering plans.

Thus, the proposed methodology allows for the anticipation of problems due to software and system non-quality, because the correction costs caused by poor quality are higher if identified downstream in the commissioning process.

Moving forward, we aim to advance discussions within ISO/IEC/IEEE 25070 Quality Engineering and develop international standards to provide designers with more practical information.

Acknowledgements

Heartfelt appreciation is extended to Dr. Alessandro Simonetta of University of Rome Tor Vergata for his invaluable insights and constructive comments on the initial draft, which have enriched this work. Other factors, such as business context, and time and cost constraints, also need to be considered.

References

[1] Nakajima, T.. "Study group report on SQuaRE future direction." *CEUR of 2019, volume 2545, pages 1-5, link https://ceur-ws.org/Vol-2545/.*

[2] H. Washizaki, eds., Guide to the Software Engineering Body of Knowledge V4.0, IEEE Computer Society, 2024.

[3] Eckhardt, J., et al. Are "non-functional" requirements really non-functional? an investigation of non-functional requirements in practice." Proceedings of the 38th international conference on software engineering. 2016.

[4] Bass, L., Clements, P., and Kazman, R., Software Architecture in Practice (Second Edition), CMU/SEI, 2011.