Software Quality Attributes: Following All the Steps

Clarrus Consulting Group Inc. 7770 Elford Avenue Burnaby, BC Canada V3N 4B7 **T** +1 (604) 540-6718 www.clarrus.com

Software Quality Attributes: Following All the Steps

Software quality attributes are likely the most neglected category of overall project scope on software projects. This is due to a perfect storm of influences:

- We naturally think of requirements in terms of the functional capabilities of our system,
- The discipline of extracting and refining these quality attributes is a process requiring the collaboration of all the project stakeholders, and
- There is little guidance in the popular literature on how to break down this difficult problem.

When we discuss the scope of a product, we almost invariably discuss in terms of capabilities, features, or how this system needs to do the same things as that system that we are replacing. Whether this is done as a list of features on the back of a dinner napkin, a more detailed collection of use cases or other analysis artifacts, or the elusive detailed set of functional requirements, this is the traditional way of looking at things. This perspective is critical, to be sure, but focusing solely on this view is a key reason that many software projects end up disappointing the client in the end.

Understanding the functionality of the system allows us to discern between a piece of tax software and the latest first-person shooter game, but we need to go beyond this to understand many of the distinctions between one tax application and another. The two systems may have roughly the same features (based on the current taxation business rules), but very often, one package will be far more attractive to the end user. This difference can and should be expressed in advance, so that we can actually build these distinctions proactively, rather than hoping for the best or trying to retrofit them in after we discover their absence in our first attempts to integrate the system. Many of these distinctions are covered in the characteristics of software quality discussed here.

Correctness as a Quality Attribute

It is interesting to note that functionality, which many teams consider the sole focus of requirements issues, is merely one element in a broad landscape of considerations for overall product quality. This sentiment echoes the original taxonomy that I was exposed to for overall quality, from the Rome Air Development Center (RADC), when I was involved in avionics-based embedded systems development. Their overall taxonomy consisted of thirteen items, correctness being one of them.

While correctness or functionality appears in a number of different taxonomies, it is reasonable to leave this out of a practical taxonomy for quality attributes, as it is generally addressed adequately through other means on most projects.

Specifying the quality of our system is a challenge if we attempt to leap directly to testable statements of different aspects of quality. This is analogous to attempting to craft testable detailed functional requirements statements without having applied the appropriate analysis techniques to better understand our system first. This article provides a series of steps, each of which is focused and straightforward. In combination, though, we gain a procedural approach for specifying our software quality over a comprehensive set of characteristics.

1. Start with a Broad Taxonomy

Quality covers a broad range of characteristics, which is one of the first challenges to overcome. Usability, performance and security are common examples of capabilities that are difficult to express as functionality, and there are many more areas to consider. We need a taxonomy that we can expect to cover the breadth of quality issues we may run into.

There are a number of taxonomies available. In Managing Software Requirements [1], the authors suggest usability, reliability, performance and supportability (then include an 'other' catch-all category to pick up the pieces). Indeed, as published in Roger Pressman[2], some groups at HP have settled on an acronym, FURPS, that captures their perspective: functionality, usability, reliability, performance and supportability. Note that in recent years, FURPS has evolved into FLURPS+ for some teams at HP, adding localization and another catchall category (the +).

McCall, Richards and Walters suggest 11 different elements[3], Karl Wiegers suggested 12[4], RADC had 13 [5]. Based on experience, Wiegers has added 2 more to his list, and currently uses 14 (the two additional elements are installability and safety). This number of elements seems to be roughly the amount required to gain reasonable coverage for overall quality. A comparison of these lists in Table 1 shows many terms are common to all of them, an indication that there is a consensus in the industry for describing the breadth of the quality landscape.

Table 1: Comparison between several broad Quality Taxonomies

McCall et. al.	RADC	Wiegers
Correctness	Correctness	
Reliability	Reliability	Reliability
Usability	Usability	Usability
Integrity	Integrity	Integrity
Efficiency	Efficiency	Efficiency
Portability	Portability	*Portability*
Reusability	Reusability	*Reusability*
Interoperability	Interoperability	Interoperability
Maintainability	Maintainability	*Maintainability*
Flexibility	Flexibility	Flexibility
Testability	Verifiability	*Testability*
	Survivability	Robustness
	Expandability	
		Installability
		Safety
		Availability

[1] Dean Leffingwell, Don Widrig, Managing Software Requirements: A Use Case Approach (2nd Edition), Addison Wesley, 2003. ISBN 03211224X

[2] Roger Pressman, Software Engineering: A Practitioner's Approach (5th Edition), McGraw Hill, 2001. ISBN 0073655783

[3] Jim McCall, Paul Richards, Gene Walters, Factors in Software Quality, NTIS, 1977

[4] Karl Wiegers, Software Requirements (2nd Edition), Microsoft Press, 2003. ISBN 0735618798

[5] The Rome Air Development Center is now the Rome Laboratory, as of 1991.

These broad taxonomies can be broken into several schemes. Wiegers breaks the list into items that are of interest to the user, and items that are of interest to the developer (italicized in Table 1). McCall's list is broken down into the product's operational characteristics, its ability to undergo change (italicized), and its adaptability to new environments (bolded). While this really doesn't help in specifying software requirements for our projects, it helps us understand the rationale for the breakdown, and to appreciate that no single stakeholder can adequately speak to all factors of quality for a product. Like all types of requirements, it is dangerous to assume that the designated analysts can generate reasonable quality requirements on their own.

Given the similarity between these lists, it could almost be a matter of drawing straws to determine which one to use for our analysis. I have found the list of 14 from Wiegers to be the most commonly used, but appreciate that this list has come from the contributions of others in the past. While it can be appealing to start with a shorter list, such as recommended by Leffingwell or used at HP, there is danger that we will miss a critical area for our project. As we will see, it is quite straightforward to start with a broad list and then remove those elements that do not apply for a given project. To start with a short list and expect to recall other areas as required, however, is a riskier approach.

Remember, these taxonomies have been developed to provide reasonable coverage of all of the aspects of product quality. Few of these terms are defined in a manner that makes them easily quantifiable (as good requirements statements should be). We will deal with this at a later stage, but there is some prioritization to be done first.

2. Reduce the List, Prioritize the Remainder

From project to project, the elements of quality that are important will vary tremendously. Embedded systems may have critical efficiency requirements given limitations of space and power, but usability considerations may be completely irrelevant. Conversely, while handheld applications may see similar efficiency requirements to those of traditional embedded systems, usability will be an important consideration for adoption and differentiation from the competition.

An effective way of prioritizing the range of quality attributes is to tackle the problem in two stages. This makes the process efficient, but still protects us from missing any important elements (as may occur if we start out with an abbreviated list).

For these prioritization steps, it is critical that all stakeholder communities are involved in the discussion. A common mistake in requirements analysis is to decide on requirements by acting for a stakeholder that we don't adequately represent, and it is very rare for an analyst to be able to authoritatively make decisions for the broad range of categories we are considering here.

The first stage is to consider each of the attributes in turn, and to determine if there is any current knowledge that would make that attribute definitively in or out of scope for this project. As an example, a handheld product to be built for a specific operating environment, with no expectation of changes to this environment, may not need to consider portability issues. This same product, though, because of limitations of memory or processing capacity, may need to definitively deal with efficiency issues at this stage. Not all of these attributes can be simultaneously optimized, and indeed, some of the areas of quality are at odds with one another (for example, a system that is build to accommodate high efficiency will inherently be less maintainable – Wiegers provides an excellent matrix showing these tradeoffs).

Generally, what we find is that at least half of all the attributes are definitively in or out of scope, but the others will require some discussion. It is these discussions, with appropriate stakeholder representation, that are critical. While it may be appealing to skip this step by using a shorter list (based on the products you generally build), this results in a very small time savings compared to the increased risk of overlooking critical requirements.

As training examples of the value of discussion, we often work through a Cafeteria Ordering System, an online system that will allow employees to order food and have it delivered, to save time. While the criteria of safety may seem irrelevant to most that participate in the discussion, allergies can be a cause for concern, and drive requirements such as the need to post ingredients for all meals. Similarly, interoperability would appear to be easily handled through straightforward deployment on existing servers, but the growth of consumer electronics could generate requirements to handle a wide range of different devices. No attribute should be considered definitively out before a stakeholder discussion takes place.

Once we have narrowed the list of attributes to consider for this specific project, we can take the prioritization a step further. An approach that I have found effective is to build a matrix of the remaining attributes, and to perform a comparison of each pair in turn. For each pair, the group selects one as being more important than the other. Table 2 shows a sample prioritization of the list. To use this approach, we take each pair in turn and make a Boolean decision (in the table, we have selected usability over availability, and indicated so with an up-arrow). When we have done all these comparisons, we have an ordered ranking, which is best managed with a spreadsheet that can tally the totals as shown in Table 2. In this example, we have selected Usability as most important overall.

Table 2: Sample Prioritization of Selected Quality Attributes (partial)

Attribute	Score	Availability	Usability	Maintainability	Reusability	Portability
Availability	**0**		^	^	^	^
Usability	**4**			<	<	<
Maintainability	**3**				<	<
Reusability	**2**					<
Portability	**1**					

Had we not performed the first culling of the list, this would become quite a tedious exercise. By chopping the list in half (which is typical), we reduce the number of comparisons by a factor of four. We find that many comparisons are quite straightforward, and some will generate significant discussion. Again, we cannot know this in advance, or without reasonable representation from a range of stakeholders.

When we are done this, we have narrowed the large list we started with, using two approaches, and are left with a subset of quality attributes that are ranked in overall importance for our specific project. We have done nothing to actually produce testable requirements at this point, but we are now ready to do so, and our narrowed focus will ensure that we build the appropriate requirements. This culling and prioritization takes little time in a focused session.

3. Translate Into Quantifiable Criteria

Unfortunately, the terms or attributes we have used to this point are not easily framed in a quantifiable fashion. To specify “the system shall be user friendly” brings us no closer to testable requirements, and a reasonable approach to getting to more appropriate terms is missing from many of today’s prominent sources of information on requirements (check any of the requirements sources referenced).

This critical step is actually one that has been around for decades, and is more often found in books related to software quality or quality engineering than to books focused on software requirements. What we need to do here is perform a translation, from the landscape of quality that has ensured we are properly covered, to a corresponding set of criteria that will allow us to more easily specify our desired results in quantified, testable terms.

There are many of these testable criteria that we can choose from. Table 3 shows an abbreviated list of criteria that map to quality attributes. A more thorough list of almost 50 criteria is captured in a spreadsheet that maps these relationships

(in the list available for download, there are currently 12 criteria that address testability, for example). The spreadsheet supports all the other steps identified in this article as well, and is available to download[6] and use on your projects.

It is likely you have used many of these as specific measures for projects in the past. They can range from specified GUI standards and response times that can support usability, Mean Time to Between Failures and Mean Time to Repair that together quantify availability requirements, and many others. From your experience, you will likely be able to add others to the list as well.

This table has grown from a number of different sources. When reviewing current requirements-based books for quality requirements (Wiegers, Leffingwell, Lauesen[7]), we get a few examples of terms that are used for specific quality attributes (such as response times for usability), but nowhere near a list that covers all criteria, and no discussion of the need to perform the mapping from attributes to criteria identified here. Indeed, in one popular text, the author simply laments "the fuzzy notion of usability" and provides no further advice on how to derive measurable quality requirements. Instead, we need to dive into Quality based references such as Deutsch[8] and Galin[9] for a more complete description of this approach. Pressman shows the relationship between the attributes and criteria, but does not provide a clear path from the broad attributes to concrete measurable requirements statements.

Table 3: Partial Mapping of Attributes (across the top) to Criteria (down left side): there are significantly more Criteria available than listed here, and all attributes are represented in the supplied spreadsheet.

	Reliability	Robustness	Availability	Flexibility	Usability	Safety	Testability	Portability
Error handling		x			x	x		
Hazard Analysis		x				x		
Inline Code Use							x	
Modularity	x			x			x	x
MTBF	x		x					
MTTR			x					
Simplicity	x			x		x	x	x
Training				x				

There are a couple of key points here. Generally, we need to identify more than one of these criteria to provide adequate coverage of any one aspect of quality (otherwise, we would simply specify our quality requirements in the original terms we had previously identified). Secondly, most of the criteria we can use will at least partially satisfy more than one of these aspects of quality at the same time (specified error handling mechanisms can support robustness, usability and safety, for example). If a criterion at least partially satisfies two or more of the attributes we deem as important (such as Modularity in Table 3), it is a more attractive and efficient criterion to use.

There are some very interesting candidate criteria in this table, many of which help underline the notion that we should consider quality for the system, rather than simply for the software component of that system. Imagine, for example, a

---

[6] The speadsheet is available for download at "http://www.clarrus.com/documents/Quality Attributes.xls"

[7] Soren Lauesen, Software Requirements: Styles and Techniques, Addison Wesley, 2002. ISBN 0201745704

[8] Michael Deutsch, Ronald Willis, Software Quality Engineering: A Total Technical and Management Approach, Prentice Hall, 1988. ISBN 0138232040

[9] Daniel Galin, Software Quality Assurance: From Theory to Implementation, Pearson/Addison-Wesley, 2004. ISBN 0201709457

system requirement to support maintainability and reusability that specifies key documents that are maintained and readily available in a specified form and location, for a specified time. While not testable by running code, this is a strong requirement for a system that will be fielded and maintained for decades.

Thus far, we have identified and prioritized the attributes of quality for our project. With this table, we need to then identify a set of measurable criteria that will give us reasonable coverage of these attributes. These will provide the terms we use to identify our measurable quality requirements, and as with the functional requirements, we need to continue to specify these elements until we have reached a point where we have adequately addressed the quality needs of our system. At some point we will reach diminishing returns (commonly called analysis paralysis), where the value of continuing to specify additional criteria is less than that obtained by forging ahead toward the implementation. This approach allows us to focus our precious analysis time in selecting the most important criteria for our needs.

4. Specify as Well-structured Requirements

Now that we have the appropriate terms that we can use to specify quality, we start to get into more familiar requirements territory.

Finally, we look at the characteristics of good requirements, provide some tips on how these apply to the approach to quality requirements we are expressing here, and tie the whole process together.

Wiegers suggests 10 characteristics that we should consider when developing our requirements: complete, consistent, correct, feasible, modifiable, necessary, prioritized, traceable, unambiguous, and verifiable. While we will never have a perfect specification, I have found that using this list while reviewing requirements can dramatically improve the results.

To some degree, we deal with several characteristics simply by following the process outlined in this series. Our initial range of quality attributes and disciplined breakdown supports completeness, and the series of steps we take give us a natural traceability that we can retain for future review. If we retain the rationale as part of this process, we have improved the modifiability of the requirements: we will be able to refer back to understand why we made the choices we did, and have better confidence in our modifications as a result.

Prioritization is addressed, but it makes sense to further prioritize our quality requirements in the context of all the requirements to ensure we are focusing on the most important areas of our system. Similarly, the approach supports consistency and necessity by deriving criteria from the attributes, but again these should be considered in the broader context as well.

The final four characteristics (correctness, feasibility, unambiguity, verifiability) all relate to the specific statements we generate from the criteria we have selected. Having translated to measurable criteria helps us for verifiability, but these four characteristics are closely related to one another. There are a number of points to consider when expressing the requirements:

- Ensure the statements are broken down so that each is uniquely testable.
- Carefully express the conditions surrounding the requirements, and be sure the statements are written in terms that can be controlled in a test environment.
- Express valid ranges and boundary conditions where necessary.
- Specify both expected behavior as well as how the system should behave in exceptional conditions (indeed, this is a key part of expressing quality requirements).

- Avoid ambiguous terms (such as several, if possible…) or expressing requirements that simply are not feasible (such as instantaneous response).

While there may be other points to consider beyond these, the critical issue is that once we have followed the first few steps of this approach, we are left with a set of terms that allow us to express the quality of our system in a manner that is identical to the way we express the functionality. We can then proceed to validate the expressed quality requirements in the same way we deal with out functional requirements. The first stages of this approach have served as a proxy for the traditional analysis techniques that we use for determining our functional requirements.

It is important to follow all of the steps in this series - skip any one and we open ourselves up for all kinds of grief:

- If we fail to start with a broad taxonomy for overall software quality, there is potential for missing an entire area of quality that may be critical. Often, neglected areas of quality are of interest to internal stakeholders: there are no quality requirements to ensure that the system is developed to be testable or maintainable, for example. Left to chance, systems are rarely built to satisfy these critical areas.

- If we fail to cull the irrelevant attributes and prioritize the remainder, we are making the entire effort of developing quality requirements more difficult than it has to be. Given a finite amount of time to develop requirements, we will be less likely to adequately cover the areas that are most important to us, diminishing the value of the effort overall.

- If we fail to make the translation from the broad quality attributes to a set of criteria, we will have a tough time expressing the quality of our system in terms that can be objectively verifiable. In addition, failure to make this translation as a conscious part of the overall process will make it more difficult for us to identify criteria that support several of our key attributes at the same time.

- If we fail to specify these criteria with due consideration to the characteristics of good requirements, we will have gone through this effort and still not developed a solid specification. Indeed, while the first few steps of this process are the equivalent of using reasonable analysis models to derive the functional requirements, this last step of ensuring the statements adhere to these characteristics is common across all requirements.

Real-world Results

This approach has been used with a number of clients over the past five years. We have found that much of the effort for identifying quality attributes for a project can be done in a single collaborative session with appropriate stakeholder representatives present.

Certainly, the narrowing of the broad list and prioritization of those remaining attributes is achievable in an hour's moderated session. From there, an initial translation may also take place in this group session, or different representatives can be tasked with actions to clarify specific criteria that will support the important quality needs for this project. At this point, simply follow the review and refinement process used for the traditional functional requirements (you do have one, don't you?).

As an example, one group we worked with had previously focused on lists of high level features and exhaustive use-case analysis as the sole approaches to determining the requirements for their system. In walking through a quality analysis using this approach, it quickly became clear that interoperability and usability were the most important areas of quality for their project, based on concerns of ease of use for operators and interaction with a third-party system for critical data.

This drove the definition of specific requirements for error handling and adherence to GUI standards (that complemented their use-case analysis), as well as the conscious negotiation of an agreement with that third-party vendor to ensure a

collaborative approach to managing that interface. While the product in question had been in development for several years, these elements had not previously been considered, and indeed had been the source of significant issues in earlier versions of the product.

Overall, this approach to identifying and capturing quality requirements takes what is generally considered to be a difficult issue, and breaks it down into a series of steps, each of which is easily managed. This makes the approach attractive for a collaborative setting involving stakeholders that may otherwise be intimidated by the analysis process, and provides a deterministic path toward achieving our goals in the limited time available on most projects. While it may not be the most detailed or sophisticated treatment of software quality attributes available, its simplicity and stepwise approach provides an opportunity for analysis in an area that is largely neglected on most software projects. If it draws the stakeholders together to discuss these quality issues on projects where they would otherwise think only in terms of functional requirements, it is an effective approach, and our experience is that this is the case.

This approach also allows us to easily retain all of our decisions along the way. Download and use the spreadsheet provided, follow this four step process, and expressing software quality will no longer be the difficult process you once thought it was.

About Jim Brosseau

Jim has been in the software industry since 1980, in a variety of roles and responsibilities. He has worked in the QA role, and has acted as team lead, project manager, and director. Jim has wide experience project management, software estimation, quality assurance and testing, peer reviews, and requirements management. He has provided training and mentoring in all these areas, both publicly and onsite, with clients on three continents.

He has published numerous technical articles, and has presented at major conferences and local professional associations. His first book, Software Teamwork: Taking Ownership for Success, was published in 2007 by Addison-Wesley Professional. Jim lives with his wife and two children in Vancouver.

About Clarrus

It is often the same class of challenges that are the root cause of most business inefficiencies. A key aspect of these challenges is that they manifest themselves differently in every organization. That's where we come in.

Clarrus is a down-to-earth, value-driven company with a focus on improving project teams and the projects they work on, across all industries.

Clarrus is unique in that it understands the interplay of human dynamics and project mechanics. Our consulting services and workshops bring a practical and proven approach to increasing the effectiveness and satisfaction of your project teams and delivering results.

Clarrus believes in the principle of teaching people to fish. Our approach nets its richest results in complex environments where multi-disciplinary teams bring diverse perspectives to the table. But, we also help small teams thrive, too. In a nutshell:

- We harness the best of best project practices and apply only what's needed for tangible short- and long-term results.
- We create an environment of trust so our sometimes tough questions can open the doors to peak performance.
- We provide inventive and practical ways for your teams to approach challenges.
- We secure the engagement of your project teams by focusing on issues where the impact is greatest.

Contact us today.

- On the web at http://www.clarrus.com,
- By e-mail at info@clarrus.com, or
- By phone at +1 (604) 540-6718.

November 2, 2010 - Version 1.1