

Implementing System Quality Attributes

Build high-quality software, leverage industry practices, and plan to build quality into your solution but be sure to prioritize carefully.

Gabriel Morgan

Implementing System Quality Attributes

When I was a teenager, my friend's father gave him the spare family station wagon as his first car. It was an old, decrepit Ford LTD station wagon that wasn't exactly the type of car anyone at our age would flaunt to attract the type of attention we were after. But it was the only car either of us had, so it was just fine. Of course, once we could fix it up a bit that is. We decided to turn it into a something with the aesthetics of a roadster by raising the rear wheels, replacing the carburetor with a set of after-market dual-carburetors, replacing the aged seats with five-point harness racing bucket seats, and giving it a new coat of paint.

During that experience, I remember leafing through loads of catalogs containing descriptions of parts from various after-market manufacturers that all matched the published Ford LTD station wagon specifications for the 454 Cleveland V8 engine, interior seat mounts, and rear suspension brackets needed for the car's conversion. We chose what we could afford and installed them ourselves following simple installation instructions. We didn't need to rebuild anything, nor force-fit any of the parts, nor fabricate any custom mounts to reuse that old station wagon for our new purposes. At the time, this event meant nothing to me other than a bit of fun over a couple of weeks during our summer break.

Now, let me tell you a very different story surrounding a very similar event. I once delivered a software solution for a customer who was a market leader in the online retail industry primarily focused on selling computer goods. The solution consisted of a set of systems which automated order acceptance through fulfillment business processes. The customer was thrilled the day the solution went into production. To him it meant that he could reduce the costs by automating much of the manual activities for these processes, and direct his focus on empowering his marketing group.

Wahoo, drinks for everyone...until a couple of years later.

The customer's industry became increasingly more competitive and in order to survive. It was clear he needed to diversify into several lines of retail goods, outsource billing services to lower costs, lower his prices by comparing multiple suppliers' price offerings, and improve customer satisfaction by committing to delivery dates by partnering with multiple shipping partners. He hoped to win new customers and to improve customer satisfaction. But the systems I had developed were not built to easily support this type of change in the business. Nor were the systems designed to be compatible with the flurry of IT packages that hit the market providing specialized components for system integration, customer care management, faster supplier enablement, marketing campaign management, and so on. I ultimately had to redesign parts of the architecture, which took considerably more time than it would have taken had I initially

designed for change in the first place. In the end, the systems required a major overhaul to meet the needs of the business.

Let's go back to my story of converting my friend's station wagon (and repurposing it for the needs of two teenagers) versus my experience converting my customer's online retail systems (to support multi-store channels, multi-supplier relationships, and provide for more sophisticated functionality). The two experiences were conceptually similar scenarios but far different in execution. A few years ago I began to think about this and ask myself a few questions. Would it have been possible for the original online retail system to be more resilient to changes in business processes, as well as changes in IT technologies? Is it possible to design systems to become more resilient to change and improve the chances for not having to perform major overhauls? I believe it is.

I'm not suggesting that we (software architects) are the sole contributors responsible for reaching the level of efficiency to which the automobile industry has reached today; as this takes experience, industrialization, collaboration, healthy competitive market of part suppliers, and adherence to automobile industry standards and quality control (which we in the software industry salivate for). What I am suggesting is that we should consider learning from the design concepts that make the automobile industry unbelievably successful and see how we go.

I propose that Solution Architects, in addition to delivering working systems to customers, are responsible for ensuring system quality with the expressed intention of building systems to be resilient to change. In this article, I will describe an approach for building systems designed to sustain changes in business processes, as well as changes in technology platforms. Hopefully, Solution Architects can leverage this discussion and reap the long-term benefits of improving their customer's return on investment.

Plan for Implementing System Quality

When implementing system quality, it's best to start with a plan. Software quality, by definition, is the degree to which software possesses a desired combination of attributes (IEEE 1992). Therefore, in order to improve system quality we need to focus our attention on software quality attributes. Ultimately, there are only a few system quality attribute primitives that all system qualities can map to. In my experience, I've grown quite fond of the system quality attributes listed below which include Agility, Flexibility, Reusability, Performance, Security, and so on. Note that these are by no means the definitive set of system quality attributes, nor are the definitions the only acceptable ones to consider. I've used the definitions below and have been relatively successful. Be sure to come up with your own set so that you can clearly communicate to stakeholders how system quality will impact of your architecture decisions.

Table 1—System Quality Attributes

| Quality Attribute | Definition |
|--------------------------|---|
| Agility | Agility is the ability of a system to be both flexible and undergo change rapidly. (MIT ESD 2001) |

| | |
|-------------------------|--|
| Flexibility | Flexibility is the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed. (Barbacci 1995) |
| Interoperability | Interoperability is the ability of two or more systems or components to exchange information and to use the information that has been exchanged. (IEEE 1990) |
| Maintainability | Maintainability is: <ul style="list-style-type: none"> • The aptitude of a system to undergo repair and evolution. (Barbacci 2003) • The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. (2) The ease with which a hardware system or component can be retained in, or restored to, a state in which it can perform its required functions. (IEEE Std. 610.12) |
| Reliability | Reliability is the ability of the system to keep operating over time. Reliability is usually measured by mean time to failure. (Bass 1998) |
| Reusability | Reusability is the degree to which a software module or other work product can be used in more than one computing program or software system. (IEEE 1990). This is typically in the form reusing software that is an encapsulated unit of functionality. |
| Supportability | Supportability is the ease with which a software system can be operationally maintained. |
| Performance | Performance is the responsiveness of the system—the time required to respond to stimuli (events) or the number of events processed in some interval of time. Performance qualities are often expressed by the number of transactions per unit time, or by the amount of time it takes to complete a transaction with the system. (Bass 1998) |
| Security | Security is a measure of the system’s ability to resist unauthorized attempts at usage and denial of service while still providing its services to legitimate users. Security is categorized in terms of the types of threats that might be made to the system. (Bass 1998) |
| Scalability | Scalability is the ability to maintain or improve performance while system demand increases. |
| Testability | Testability is the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met (IEEE 1990). |

| | |
|------------------|---|
| Usability | Usability is: <ul style="list-style-type: none"> • The measure of a user’s ability to utilize a system effectively. (Clements 2002) • The ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component. (IEEE Std. 610.12) • A measure of how well users can take advantage of some system functionality. Usability is different from utility, a measure of whether that functionality does what is needed. (Barbacci 2003) |
|------------------|---|

Back to my story regarding the station wagon conversion—I was pleased with the amount of effort it took to modify and extend the Ford LTD station wagon. The designers of the Ford LTD station wagon predicted that there would be a need for owners wanting to replace parts with after-market components. So the car was designed to be maintained, in my case upgraded, and to be compatible with parts from other car part vendors.

There are a few system qualities that contributed to this positive experience, such as Maintainability and Testability, but there is one system quality that stands out: system Flexibility. The Ford LTD station wagon designers specifically implemented the Ford LTD design for Flexibility. Maybe not in these exact terms, but I do think they thought about making the designing the automobile to be compatible with parts from other automobile parts manufacturers. Of course, the primary driver for optimizing Flexibility was probably not for benefit of the after-market car part manufacturers, but rather to give Ford the benefit of choosing from several bidding car part manufacturers. Thus allowing them to choose the right balance of quality and cost for their production model. As a consequence, car part manufacturers benefited from after-market sales by providing upgraded replacement parts to the same flexible design specification.

Someone smart once stated, "You can't manage what you can't measure," therefore you need to think about the measures of system quality to make sure you can plan how to go about monitoring them throughout the software lifecycle. You will need to plot out just what you will expect the system designers to build so that it can be optimized for system quality. When planning how to implement system quality, a good approach you can leverage is the IEEE Software Quality Metrics Methodology (IEEE 1992). Basically, this methodology suggests following these steps to define and monitor system quality metrics:

1. Establish software quality requirements.
2. Identify software quality metrics.
3. Implement software quality metrics.
4. Analyze results of these metrics, and finally.
5. Validate the metrics.

Prioritizing System Quality Attributes

The IEEE Software Quality Metrics Methodology (IEEE 1992) is more of a framework than a strict methodology. You need to add the details yourself to make it work.

First, so as to speed up this process, prioritize the system quality attributes before spending time identifying software quality metrics. That is, what's the point of spending time identifying software quality metrics for low priority system quality attributes you're not going to monitor?

Second, don't strictly follow the serial process regarding steps 4 and 5 above, because it is good practice to monitor for the existence of well-engineered system quality requirements as part of the overall solution requirements. As Karl Wieggers notes "software quality attributes or quality factors, are part of the system's nonfunctional (also called non-behavioral) requirements." (Wieggers 2003) The assumption here is that including system quality requirements into the solution's project requirements improves the chances for a high-quality solution. For this reason, don't assume that you have to complete each step before moving to the next, as you may find yourself missing the boat on an opportunity to improve system quality. For example, you may find yourself well into the design phase of your project by the time you complete steps 1 through 3. As you monitor and analyze the metrics regarding system quality requirements, you may recognize that you're too late to go back and inject missing system quality requirements for the project, which leaves you at greater risk of developing a poor-quality solution.

At this point, you should have a set of defined system quality attributes to draw upon, so the next step is to prioritize them for your particular solution. You should also think about how you will go about monitoring for each attribute throughout the software development lifecycle with the aim of determining if system quality is being implemented.

Ideally, you would want to optimize for all quality attributes, but the fact is that this is nearly impossible because any given system has Tradeoff Points (Clements 2002) which prevent this. A Tradeoff Point is a property that affects one or more attributes. Essentially, changing one quality attribute often forces a change in another quality attribute—either positively or negatively. This is important because knowing the prioritized system quality attributes and their Tradeoff Points aids the decision-making process during design activities.

The graph below is taken from the book *Software Requirements Second Edition* (Wieggers 2003) and describes an example set of system quality attributes and their Tradeoff Points.

Table 2-System quality tradeoff points

| | Availability | Efficiency | Flexibility | Integrity | Interoperability | Maintainability | Portability | Reliability | Reusability | Robustness | Testability | Usability |
|------------------|--------------|------------|-------------|-----------|------------------|-----------------|-------------|-------------|-------------|------------|-------------|-----------|
| Availability | | | | | | | | + | | + | | |
| Efficiency | | | - | | - | - | - | - | | - | - | - |
| Flexibility | | - | | - | | + | + | + | | + | | |
| Integrity | | - | | | - | | | | - | | - | - |
| Interoperability | | - | + | - | | | + | | | | | |
| Maintainability | + | - | + | | | | | + | | | + | |
| Portability | | - | + | | + | - | | | + | | + | - |
| Reliability | + | - | + | | | + | | | | + | + | + |
| Reusability | | - | + | - | | | | - | | | + | |
| Robustness | + | - | | | | | | + | | | | + |
| Testability | + | - | + | | | + | | + | | | | + |
| Usability | | - | | | | | | | | + | - | |

Here is a quick example to illustrate Tradeoff Points. Let’s say that there is a need for a system service which aims to provide a means for processing banking transactions. The business has noted in the requirements that it must be fast. Assuming that the system designer has not read this article, she immediately begins to optimize the service for Performance. In this hypothetical case, the system designer might opt to build a high-performance application by building a system which:

- Captures system requests directly from UDP communications port.
- Uses proprietary message communication semantics.
- Accepts system requests and processes application logic for system requests in a single process space.
- Embeds application logic in procedural code.
- Persists data in local memory space for quick put/get operations.
- Sends responses to system calls in real-time to a high-performance receiving application preferably as close to the hardware level as possible using a proprietary binary communication transport.

A system such as this would have a number of Tradeoff Points. The three that stick out for me are:

- *Interoperability.* It will be difficult to interoperate due to the unstructured, unreliable UDP communication protocol and the unsupported message formats which are not based on industry standards.
- *Flexibility.* The application's decomposition doesn't lend itself to change or reuse for other purposes due to the proprietary protocols, coupled messaging, application logic, and propriety memory store. The application also lacks application decomposition with encapsulated boundaries preventing other components from being plugged into the system.
- *Reliability.* UDP communication packets are unreliable as packets could be lost. In-memory data is not persisted to disk and could be lost if the memory is released via a process fault, reboot, etc.

It may be that these sacrifices are intentional and accepted in order to achieve a high level of performance...but maybe not. The business and IT owners may actually want the system to be flexible enough to withstand changes to the business processes it supports or technology changes that are inevitable. The point is that tradeoffs are made when optimizing system quality attributes and that if systems are designed with this understanding upfront, there will be fewer surprises down the road.

There are several processes that describe how to prioritize system quality attributes to derive system architecture such as Software Engineering Institutes (SEI) Attribute-Driven Design (Kazman 2000). In addition, there are some fairly well documented Tradeoff Points from SEI's Architecture Tradeoff Analysis Method (ATAM). ATAM is a very sophisticated method for determining Tradeoff Points through attribute characterizations which use the Stimuli-Architectural Decision-Responses construct (Kazman 2000).

ATAM is focused on the evaluation of software and included in the method are techniques for identifying and prioritizing system quality attributes. There are several other techniques/methods for identifying system quality Tradeoff Points in the industry. From these industry practices, the decision of which to use and how much to use it depends on factors such as budget, resource competency, time to deliver, solution size, and complexity.

I often tend to simply adopt key concepts from these, infer a set of likely important system quality attributes from the solution requirements, and collaborate with the key stakeholders to decide on three high-priority system quality attributes on which to focus. So, my advice to you is to understand the available approaches and weigh them for each solution to determine what's best to include in your plan.

Monitor System Quality Attributes

Up to this point, we have defined a plan and are now ready to define the metrics of system quality and to monitor and track them throughout the software development lifecycle. The

purpose of using metrics is to reduce subjectivity during monitoring activities and to provide quantitative data for analysis.

So, what metrics should a Solution Architect inject into the solution to better address system quality? Any good system architecture has quality requirements, so this is a good place to start. To help explain this, consider Flexibility; a specific system quality attribute that I personally consider very important in delivering Agility to the business and IT stakeholders. Here are some Flexibility metrics to consider as an example of what to monitor for:

- *Quality of Service (QoS) system requirements.* The Solution Architect is responsible for the integrity of a solution and poorly-defined system requirements can lead to confusion downstream and result in a poor quality solution. So, to mitigate this, ensure that system requirements include QoS requirements specifically for those which are correlated to the prioritized system quality attributes. Also, accompanying QoS requirements with Use Cases and Quality Scenarios (Bass Kazmann 1999) will further improve communicating the requirement to the project team and mitigate misinterpretation. For the purposes of the Flexibility example, here are two QoS examples:
 - “With no more than one hour of labor, a business user who has at least six months of experience in the field shall be able to modify core business processes automated by the system without requiring a change in the system’s source code which, when done, will place the changes into a queue to be tested.” This requirement demonstrates the ability for the system to withstand changes in the business process.
 - “A software engineer can change the Find Customer function in the solution to point from the current SAP CRM module to the Microsoft CRM provided Customer Search service interface (including the effort required to research, design, code, unit test, and document) and release the code changes to the testing environment within one day.” This requirement demonstrates the ability for the system to withstand changes IT systems.
- *System patterns that improve quality.* The Solution Architect should also identify system patterns which should be used to optimize for the prioritized set of system quality attributes. In my example of system Flexibility, patterns that improve system Flexibility include Façade (Gamma 1995), Adapter (Gamma 1995), Service Layer (Fowler 2003), and so on.
- *System anti-patterns that degrade quality.* Solution Architects should provide additional guidance on what system designers are to avoid and one method to do this is to identify the anti-patterns which negatively affect system quality. In my example of system Flexibility, anti-patterns such as Shared Database (Hohpe Woolf 2004) and Data Replication (Fuller Morgan 2006) degrade system Flexibility.

The point is that it’s up to the Solution Architect to identify metrics which are used to measure system quality. Unfortunately, no one has fully defined all measures based on scientific mathematics so this can be a challenge. Aim to define as much as you can, leverage existing metric definitions from industry practices as a starting point, and build on it from your experience.

Once the system quality metrics are defined, embed them into the solution artifacts and monitor for their adoption throughout the software lifecycle. During the build and testing phases, review the system and find deviations from what you defined. Any deviations should be considered for correction.

By implementing software quality into a solution as described in this article, you will be better positioned to give the business owner a quality solution; one that is more likely to withstand changes in the business and technology, therefore maximizing the return on investment.

Review

Let's go through a quick summary of the three key points in this article. First, build high-quality software. Secondly, leverage the industry practices that guide Solution Architects to build high-quality software systems. And thirdly, build a plan for implementing system quality into your solution and avoid optimizing for all quality attributes (as this is nearly impossible to do). Instead, prioritize the quality attributes and focus your attention on the top three. Ideally, if you succeed, you will have improved the chances that your software will last, and may also reap the long-term benefits of improving your customer's return on investment.

Critical Thinking Questions

- What are the most significant system quality attributes that contribute to Agility and how would you weight them?
- Besides Quality of Service requirements, design patterns, and anti-patterns, what other measurable metrics could you monitor to track system quality?
- What other methods, beyond giving guidance to system designers, could you employ to improve system quality? That is, are there methods that other team members could adopt as their responsibility to improve system quality?

Sources

- [Bachmann 2000] Bachmann, F.; Bass, L.; Chastek, G.; Donohoe, P. & Peruzzi, F. *The Architecture Based Design Method* (CMU/SEI-2000-TR-001 ADA375851). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. Available WWW: <URL: <http://www.sei.cmu.edu/publications/documents/00.reports/00tr001.html>>.
- [Barbacci 1995] Barbacci, M.; Klien, M.; Longstaff, T; Weinstock, C. *Quality Attributes - Technical Report CMU/SEI-95-TR-021 ESC-TR-95-021*. Carnegie Mellon Software Engineering Institute, Pittsburgh, PA.
- [Barbacci 2003] Barbacci, M. *Software Quality Attributes and Architecture Tradeoffs*. Software Engineering Institute, Carnegie Mellon University. Pittsburgh, PA.
- [Bass 1998] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*. Reading, MA; Addison-Wesley.
- [Bass Kazmann 1999] Bass, L.; Clements, P.; & Kazman, R. *Architecture-Based Development*.

- [Fowler 2003] Martin Fowler. *Patterns of Enterprise Application Architecture*, Boston, MA. Addison-Wesley.
- [Fuller Morgan 2006] Tom Fuller, Shawn Morgan. *Data Replication as an Enterprise SOA Antipattern*. The Architecture Journal Issue 8. Microsoft Corporation. <URL: http://architecturejournal.net/2006/issue8/F4_Data/default.aspx>
- [Gamma 1995] Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley. Carnegie Mellon Software Engineering Institute
- [Hohpe Woolf 2004] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns*. Boston, MA. Addison-Wesley.
- [IEEE 1990] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York, NY.
- [IEEE 1992] IEEE Std 1061-1992: *IEEE Standard for a Software Quality Metrics Methodology*. Los Alamitos, CA: IEEE Computer Society Press.
- [Kazman 2000] Kazman, R.; Klein, M. & Clements, P. *ATAM: Method for Architecture Evaluation* CMU/SEI-2000-TR-004 ADA382629. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. Available WWW: <URL: <http://www.sei.cmu.edu/publications/documents/00.reports/00tr004.html>>
- [MIT ESD 2001] Tom Allen, Don McGowan, Joel Moses, Chris Magee, Dan Hastings, Fred Moavenzadeh, Seth Lloyd, Debbie Nightingale, John Little, Dan Roos, Dan Whitney. *ESD Terms and Definitions (Version 12)*; Massachusetts Institute of Technology. Engineering Systems Division. ESD-WP-2002-01, October.
- [Wieggers 2003] Karl Wieggers. *Software Requirements Second Edition*; One Microsoft Way, Redmond, WA; Microsoft Press.

About the Author

Gabriel Morgan has over 10 years developing software and is currently an Enterprise Application Architect on Microsoft's IT Enterprise Architecture team. He has extensive experience in the design, development, and implementation of distributed software systems and experience solving enterprise application problems such as flexibility, performance, and scalability. He has a patent-pending process and tool for performing system quality reviews called the Microsoft System Review. His current interests lie in deriving system architecture from business strategy.