Don’t Fire the Architect! Where Were the Requirements?

Jane Cleland-Huang

RECENTLY, MANY of us have been dismayed by the problems that accompanied the launch of the US Healthcare.gov website. The website, which was designed to help insurance seekers learn about the healthcare reform law and find and compare private and public insurance options, failed dramatically during its initial launch. Blame for its failure has been alternately laid at the feet of politicians, project managers, architects, development platforms, and the go-live timeline. Unfortunately, fiascos of this depth and magnitude often have long-term repercussions on people’s lives, services provided, and trust in the IT community as a whole. We therefore need to seek deeper understanding of their root causes so that we can learn from our mistakes and do a much better job the next time around.

Where It Begins
In most major debacles of this nature, and certainly in the case of Healthcare.gov, the failure was caused by a complex interplay of problems. Any single-point remedy is therefore clearly insufficient for “solving” such problems or for preventing them from occurring again. However, anyone following the news would likely agree that the crux of Healthcare.gov’s failure occurred at the architectural level. Consumer Reports noted that of the 9.47 million people who attempted to register at HealthCare.gov during the first week of its launch, only 271,000 succeeded (http://rt.com/usa/consumer-reports-obamacare-site-557). The system simply was not designed to handle the massive influx of initial users. Given government access to demographics, we could imagine that it would have been possible to more accurately predict usage patterns, and this suggests that there was simply insufficient effort made to build a system that would meet the performance and availability needs of its stakeholders. Furthermore, Morgan Wright of Crowd Sourced Investigations reported in a hearing to the US House Committee on Science, Space, and Technology that Healthcare.gov created massive opportunities for “fraud, scams, deceptive trade practices, identity theft, and more” (http://science.house.gov/hearing/full-committee-hearing-my-data-healthcaregov-secure). Security solutions were seemingly slapped together in a shoddy manner instead of through the kind of systematic approach that is expected in a high-quality software solution. Architectural failures were pervasive across the entire system, and the system underperformed in numerous ways.

Within a few days of the failed launch there were calls to “fire the architect!” Although this demonstrated a certain understanding of the software development process, it also suggested that architects design solutions in a vacuum. This is not the case. Architects can only consistently design and deliver correct solutions if the requirements are known. In fact,
it turns out that the “requirements” for Healthcare.gov came in the form of many thousands of pages of legal healthcare regulations. Although such business rules can be translated quite seamlessly into functional requirements, they are typically devoid of quality concerns which could ultimately translate into architecturally significant concerns describing qualities such as performance, security, availability, and usability.

This is a fairly common mistake. Too often, the focus is primarily on functionality, while important design constraints related to required system-level qualities are assumed rather than explicitly stated. Over-specifying design constraints in early stages of the requirements process can lead to premature and possibly inferior solutions, which are unlikely to satisfy the real needs of stakeholders; however, completely ignoring such constraints will almost definitely produce disastrous consequences. It is important to elicit, analyze, and specify quality concerns as well as functional ones.

In practice, many software requirements specifications contain few, if any, quality-related requirements.1 For whatever reason, people seem to assume that everyone implicitly understands and agrees on the required system qualities. However, questions such as “how secure?” “how fast?”, or “how reliable?” often remain unanswered. The Healthcare.gov project serves as a chilling reminder of what happens when such questions are not adequately addressed or not fully implemented in the delivered system. While specifying quality concerns in no way guarantees that the system will satisfy them, it is clearly an important first step.

Where It Could Change
There are many different approaches we might take for eliciting and then documenting such requirements. We could specify them informally as user stories, adopt structured techniques such as Tom Gilb’s Planguage,2 or utilize template approaches such as Volere.3 Here, I use the Volere approach to illustrate four interrelated requirements that could apply to the Healthcare.gov problem (see Figure 1). The Volere template, called a shell, allows the requirement to be described informally, supported by a

---

**FIGURE 1.** An example of how a Volere shell could be applied to Healthcare.gov requirements.
rationale, and then specified in a non-ambiguous clearly testable manner as a “fit-criterion.” A more detailed explanation of the Volere approach is provided in Suzanne Robertson and James Robertson’s book *Mastering the Requirements Process.*

The first example specifies a basic performance requirement about the servicing of quotes. The second specifies a usability requirement to keep the user informed of the status of his or her query. The third places constraints on the downtime of service providers. The fourth specifies what needs to happen if a service provider is unavailable at the time of the request.

In addition to these four examples, there would obviously be many more requirements, some of which would contextualize the four examples. For instance, we would expect to see availability and performance requirements contextualized according to anticipated usage patterns, as well as requirements specifying privacy and integrity constraints. Only once a fairly complete set of concerns has been elicited is it possible to address the task of designing an architecture capable of meeting the requirements. Instead of having an oblique, ill-defined set of goals, we design against a clear understanding of the required behavior of the system.

According to Jeffrey Palermo, CEO of Clear Measure Inc. in Austin, Texas, the architectural design of Healthcare.gov was flawed from the start. The basic architecture was designed and built around the notion that the system would forward requests for quotes from insurance seekers to external vendors in real-time; however, this massive interconnectivity and the subsequent burden on the government servers caused the system to collapse. Palermo suggests, on the basis of experiences with other successful large-scale, high-performance systems, that it would have been far better to cache information from individual vendors on the server and to generate quotes from the cached data.

The real question here is whether architectural design mistakes could have been avoided if requirements had been more clearly defined and candidate architectural solutions evaluated against them. Although we can’t categorically answer that question, understanding architecturally significant requirements early in the process would have facilitated a feasibility analysis of the planned approach and likely unearthed its inadequacies before tens of millions of dollars had been wasted on an ill-conceived design. The Software Engineering Institute has developed several different architectural assessment techniques for exactly this purpose. For example, in the Architectural Tradeoff Assessment Method, requirements are specified as quality attribute scenarios and the architecture is evaluated systematically against its ability to satisfy them.

Clearly, quality concerns, such as the ones I’ve illustrated using the Volere shell, drive architectural design; however, it can never be a one-way street. Sometimes alternate architectures will satisfy different requirements to different degrees, and therefore tradeoffs will be necessary and requirements must be adjusted to reflect this. Equally as clear, writing quality-related requirements is no panacea to the problem of failed architectures. Failure can occur for numerous reasons, of which designing and building a system against an ill-defined set of requirements is just one.

I’ll end with a comment about the degree of formality we need to specify quality concerns. Martin Glinz explored this issue in an earlier *IEEE Software* article and identified various contexts that required more, or less, formality in the way requirements are specified. However, the sheer scope and challenges involved in the Healthcare.gov project should have served as a trigger for more formality in the process. While stressful time-to-market deadlines often lead to corner cutting, smarter development teams will realize that such projects have little time for missteps and that investing time in understanding requirements early in the process will pay dividends over the long run.

**Acknowledgments**

I thank James Robertson from the Atlantic Systems Guild for his insightful comments on this article and particularly for his feedback on the sample requirements.

**References**


**JANE CLELAND-HUANG** is a professor at DePaul University. Contact her at jhuang@cs.depaul.edu.