Editor: Frank Buschmann ■ Siemens Corporate Technology
■ frank.buschmann@siemens.com

Learning from Failure, Part 2: Featuritis, Performitis, and Other Diseases

Frank Buschmann

In my last column, I began an analysis of common software architecture mistakes. I'd like to continue this discussion and explore three mistakes that—I'm sure—most architects know all too well. My architect colleague Klaus Marquardt and I name these mistakes as if they were diseases: featuritis, flexibilitis, and performitis.[1]

Featuritis

Featuritis is the tendency to trade functional coverage for quality—the more functions and the earlier they're delivered, the better. Reliability, performance, maintainability, and other qualities are postponed to "when the functionality is stabilized."

A software project's business stakeholders partly drive this thinking. But architects who blindly accept such thinking are, in my mind, not pragmatic. I explained at the Object-Oriented Programming 2009 conference that a key reason for software project failure is lack of operational quality, not missing functionality. If architects observe a preference for features, there's a strong demand for clarifying scope and requirement priorities, and it's their responsibility to initiate a discussion.

Have you ever wondered why some stakeholders prefer new functionality over quality? A product manager once said that if he didn't insist on large feature lists with tough realization deadlines, his team wouldn't deliver anything. This statement was a clear sign of mistrust. "They don't know what's important," the product manager said between the lines. "Therefore, I must require more than I actually need to get something meaningful back."

There's only one escape from such situations: architects must actively break the cycle of mutual misunderstanding and mistrust! They must initiate a discussion about the key features that will contribute to the system's success, including the quality they'll require.[2] Architects who don't raise their hands move into an awkward position: they implicitly commit to meet the system's requirements as stated. No wonder they're blamed if the project gets in trouble during development.

Another reason for featuritis is that architects prefer simple things over challenging and risky things. It's often easy to realize some fundamental functionality if you ignore the quality aspects. Progress is immediately visible with this strategy, but it also creates the illusion that an architecture is more mature than it actually is. At times this practice is appropriate, for example, for trade show prototypes. However, such demos are rarely of production quality. Either significant refactoring is necessary before the real development can begin or they must be thrown away. Not communicating the need for quality but enriching the scaffolding with further functionality will likely cause massive costs and schedule slips.

Flexibilitis

The same effect can occur if architects fall prey to flexibilitis and overload their architectures with extension, adaptation, and configuration facilities. Overly flexible systems are hard to configure, and when they're finally configured, they lack qualities like performance or security.

Flexibilitis is partly due to "polluting" systems with unneeded variability requirements, which suggests too broad of a scope.[2] Mostly, however, it's the architects who add more flexibility than needed, or even the wrong flexibility. When I asked an architect of an extremely flexible but completely unstable system why he created such a generic design he said, "I didn't want to constrain the architecture too early."

The root cause of this example is, however, of a more general nature: architects use flexibility as a cover for uncertainty. Rather than make difficult design decisions, they hide behind flexibility.[3]

0740-7459/10/$26.00 © 2010 IEEE

Another cause of too much flexibility is that some architects love to play. They never tire of adding variability "just in case," or "to be prepared," or "because it is so simple to add." However, too much flexibility isn't better; it adds complexity that creates an expensive burden. Such systems are hard to manage and likely lack operational qualities.

Performitis

Things can get worse if architects don't refocus on quality when they notice a project has a preference for features and flexibility. The closer such projects get to their delivery deadline, the more likely they are to become infected with another disease: performitis.

Marquardt characterizes performitis as:

> *Each part of the system is directly influenced by local performance tuning measures. There is no global performance strategy, or it ignores other qualities of the system such as testability and maintainability.*[1]

Does this sound familiar? My current project struggled with excessive performitis as a result of impressive featuritis and flexibilitis. It's a platform project, where the effect of all three ailments is often really expensive. In the beginning, there was a dominating product that dictated the platform's main features and the interfaces for their use. Since both were inappropriate for other products, the architects added extension hooks and additional interfaces to hook these products onto the platform. In the end, the platform was so flexible that core services suffered from unacceptable latency combined with excessive resource usage.

The development organization created a performance task force. But instead of eliminating the problem's root causes, the team addressed its symptoms by working around the programming platform and sacrificing a clear design for bit-level optimizations. The services' performance improved to the limits in the given design, but at the cost of their maintainability, reliability, and architectural clarity. Since these measures were still insufficient to meet the required performance, the project finally rewrote several core platform services to ensure their operational quality with a "just enough" flexibility to meet the products' needs.

Walking Skeletons

When pragmatic architects begin design work, they first create a *walking skeleton*, a baseline architecture that supports the functional requirements with the highest contribution to the system's business case and the most challenging quality attributes. Often the two categories overlap, which makes the requirements extremely architecturally relevant. In product line engineering, a commonality/variability analysis helps identify architecturally significant requirements: those that are business-relevant, with challenging quality attributes, and common across the product line's scope. [4,5]

Pragmatic architects also drive the baseline's design with the system's application domain[6] to ensure the architecture is partitioned into meaningful components with meaningful interfaces, relationships, and interactions. Just meeting the system's requirements in some arbitrary structure isn't sustainable over time; features must execute in the right environment. And this is only possible if the system's architecture portrays its application domain properly.

Use cases and scenarios help guide the concrete design of a baseline architecture by defining the path through the domain model that's actually realized. We need the essential end-to-end slices through the system; small but expressive, challenging, and representative. Too much breadth in a single aspect distracts from a system's architectural whole. The quality requirements associated with the selected scenarios ensure that the baseline architecture also includes appropriate technical infrastructures for these qualities; and that the relevant domain components are designed to use these infrastructures appropriately.

I limit my initial design work to the main success and failure scenarios of the architecturally significant requirements, leaving their extensions and variations aside for a while. Why? If I'm unable to address the main success and failure scenarios properly, no extensions or variations of these scenarios will help. Once this architectural backbone is stable and working, I open it up to support the required variability. Feature models[7] and concrete extension and variation scenarios help guide this work. Feature models' core strength is that they capture variability at a systemic level, in contrast to specific scenarios, and they support a defined variability management. In reality, developers can't choose system variations of their own accord. Rather they come in certain packages—for instance, because they support a specific system version or are dependent on one another. Feature models portray these clusters of related variable features clearly from both a technical and a marketing perspective. Consequently, opening a given design to support variability should also happen in a defined manner—by supporting the extension and variation scenarios of the top-priority variation packages defined in the feature model. Again focusing on essence, not breadth, is the key.

These practices help define a baseline architecture that's as lean as possible while addressing all aspects necessary for architectural success. If realized in an agile manner, the baseline architecture can even be executable, hence the term walking skeleton. And it's a perfect prescription for avoiding featuritis, flexibilitis, and performitis. Grady Booch noted that "architecture represents the significant design decisions that shape a system, where significant is measured by cost of change."[8] Walking skeletons help minimize the possibility of costly architectural changes.

References

1. K. Marquardt, "Performitis," *Proc. 8th European Conf. Pattern Languages of Programs*, Universitätsverlag Konstanz, 2003, pp. 48–49.
2. F. Buschmann, "Learning from Failure, Part 1: Scoping and Requirements Woes," *IEEE Software*, Nov./Dec. 2009, pp. 68–69.
3. K. Henney, "Use Uncertainty as a Driver," *97 Things Every Software Architect Should Know*, R. Monson-Haefel, ed., O'Reilly, 2009, pp. 321–361.
4. K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*, Springer, 2005.
5. D.M. Weiss and C.T.R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison-Wesley 1999.
6. E. Evans, *Domain Driven Design*, Addison-Wesley, 2004.
7. K. Czarnecki and U. Eisenecker, *Generative Programming, Methods, Tools and Applications*, Addison-Wesley, 2000.
8. G. Booch, "On Design," *Handbook of Software Architecture*, blog, 2 March 2006, www.handbookofsoftwarearchitecture.com/index.jsp?page=Blog&part=2006.

Frank Buschmann is a principal engineer at Siemens Corporate Technology, where he's a principal software architect in the Software and Engineering Division's Architecture Department. Contact him at frank.buschmann@siemens.com.