

Software Architecture vs. Software Engineering

Sten Sundblad, Tuesday, June 05, 2007

The content of this article might be changed without notice!

Executive Summary

Over the last few decades, successful business activities increasingly depend on the software used to improve the efficiency of those activities. There's a good side to that, because without access to efficiency-boosting software, no business can last very long in today's climate of fierce and global competition. The only exceptions would be the very small, local company or a company enjoying a monopoly.

Businesses today are often exposed to enormous pressure for change. To be able to change the way a set of business activities are performed, the software that supports these activities would typically also have to be changed. But if the software is too complex, and if its architecture is not well enough aligned to the business architecture, then the business must postpone its change until the software can be changed with it. Then market opportunities might get lost, competitive threats can't be met fast enough, and the business suffers. Sadly, according to many business managers, and to analysis companies such as the Gartner Group, this is the normal situation for many, perhaps most, enterprises.

Service Oriented Architecture (SOA) comes with a promise to increase the agility of businesses. This is because SOA, compared to older architectural styles, allows closer alignment between software architecture and the way business activities are organized. But what *is* software architecture, and what is the difference between it and software *engineering*, if any? These are important questions, because if we can't answer them properly, then the SOA promise won't be fulfilled.

So, in the context of building computer software, what is architecture and what is an architect? It's not as easy as it might sound to answer these questions in a useful way. Computer software is such an abstract product, and it's also a rather new phenomenon. Computer software has been created from the late 1940s only. In the first decades, both the computers and the ideas of what we could do with them were quite simple, so there was not much need for architecture in the first place.

Early Attempts to Structure Software

In the early days of our industry, software was designed while it was programmed. There was no conscious development process with formal steps to guide development work. A sales person might doodle a sketch on a piece of paper, or even a paper napkin, and get OK from the customer to build the system. Or, rather, develop the program, because computer software in these early days was more thought about as individual programs than as systems. So there was not much need for architecture in those days.

However, as the computer industry evolved, both computers and the requirements for software to run on them became increasingly complex. By time some smart individuals found that better results could be achieved if software was *designed* before it was programmed. Ed Yourdon, who created one of the first and most popular methodologies for structured analysis and design, was one of these individuals; David Parnas, who developed the concept of modular design, which became the foundation of object-oriented programming, was another; Niklaus Wirth who created some of the first structured programming languages – Pascal and Modula-2 – was yet another one of these creative individuals.

This started in the 1960s, and when these ideas made ground in the 1970s, the idea of structure and design of computer programs increasingly came to dominate the development of non-trivial software. The concept of *software engineering* was increasingly talked about, but nobody really talked about *architecting* software at that time.

Early Statements About Software Architecture

The first one, to my knowledge, that talked publicly about software architecture was Frederick P. Brooks, Jr. Brooks is best known as the *father of the IBM System/360*, and he managed the development of the Operating System/360 during its design phase. In 1999 he received the AM Turing Award “For landmark contributions to computer architecture, operating systems, and software engineering.”

In his book *The Mythical Man-Month*ⁱ Brooks talked about the importance of architecture for large and complex software systems such as the 360 operating system. To my knowledge, this is the first anyone has written about the concept of software architecture, and it was written in 1974 when the first edition of the book was published. The following is a direct quote from that book:

By the architecture of a system, I mean the complete and detailed specification of the user interface. For a computer this is the programming manual. For a compiler it is the language manual. For a control program it is the manuals for the language or languages used to invoke its functions. For the entire system it is the union of the manuals the user must consult to do his entire job.

Notice that this statement is not technical at all! It’s all about what can be observed from an external position, and nothing about the internal structure of the respective systems.

In the same book Brooks also quotes and paraphrases Gerrit Blaauw, another IBM employee who worked with Brooks on several projects. Blaauw had given an example of a clock, the architecture of which consists of the face, the hands, and the winding knob. When a child has learned this architecture he can tell time as easily from a wristwatch as from a church tower. The *implementation*, however, describes what goes on *inside* the case. Notice that the italics in this paraphrase is mine, not Brooks’ or Blaauw’s. It’s also my comment that the architecture of an analogue clock, as we all know, can be implemented in hundreds or even thousands of ways, while the architecture in principle stays the same.

Anyway, Blaauw’s example of the architecture of the clock is interesting, because it shows how important it is to *clearly separate architecture from implementation, architecture from engineering*,

architecture from technical design. If the architecture of the clock had contained details of its technical implementation design, then choosing a different implementation design would have been so much more difficult. In fact, to achieve a new implementation design, the original implementation design must first be removed from the architecture, separating the implementation design from the pure architecture of the thing, leaving it the way Blaauw described it in the first place.

Experienced From an External Position

To repeat myself, Brooks and Blaauw talks about architecture as something which can be experienced from an external position, while software engineering is concerned with structures which are internal to the software. Software engineering is also very much concerned with the procedures for creating these structures.

This is highly compatible with the borderline between building architecture and building engineering. The architecture is responsible for the functionality and the esthetics of the building, the engineer for its technical design.

The borderline is very clear between what is architecture and what is engineering in the process of creating tangible products such as buildings, bridges, airplanes and others. This line seems to be equally clear in the views of Brooks and Blaauw. Is it just as clear in the minds of most of today's software architects or software engineers? I firmly believe that it's not, and also that many things would be much better if it were.

With a sharp line between architecture and the implementation of architecture, it becomes possible to first create an architecture and then discuss different ways to implement it. It also becomes possible to *change* the implementation, while keeping the architecture unchanged, if the implementation turns out to be less effective than the engineers thought when they originally designed it. The implementation could also be changed when new options for the implementation become available.

Perhaps even more important, it becomes possible to discuss changes to the *architecture*, without making things unnecessarily complex, which they would be if details of the present implementation were involved in the discussions. Just as in the clock example.

I know that many would disagree with this, arguing that the architect should be responsible for high-level technical design as well. Then, the interesting question would *not* be where the borderline between the architect and the engineer is, but where it should be drawn between the architect and the *programmer*.

It's not difficult to understand such reasoning, especially if the architect is seen as a *person* rather than as a role. After all, most architects *do* double as software engineers, doing technical design of the implementation of a specific architecture created by themselves. For example, it's more likely than not that the clock architect in Blaauw's example was responsible for the implementation of the first clock as *well* as for its architecture.

However, I would argue that such a person is changing role, from architect to engineer, when he moves from external design to internal, technical implementation design. So there are two borderlines, one between what's architecture and what's implementation, and one between what's implementation design – engineering – and what is manufacturing or programming. And to my mind, it's the first of these borderlines that is the most important one.

The reason for this is that architecture should tell us about *what* we must build to respond to business and usage requirements, but not about *how* we should build it. Keeping the *how* away from all communication between the architect and the business people or users makes such communication so much easier. If we do mix in a lot of *how* in the conversation with buyers and users, then it's very easy to go astray and miss the real target. There's also a fat chance that you would lose your audience too.

Notice though, that having this sharp borderline between the *what* and the *how* does *not* make it impossible to introduce the *how* to those business persons who are interested.

Affecting Usage Behavior

I firmly believe that there is a specific place where we should draw the borderline between software architecture and software engineering. It's between what does and what does not affect usage behavior. If a certain design decision requires a different usage behavior than another, then that decision is architectural. If it does not require a different usage behavior, then it's an engineering decision. For example, this would make all the details of contract design very clearly architectural, while a decision to use a particular application block for, say, database access, would be just as clearly an engineering decision.

This becomes even more evident if we think in terms of *changing* an already taken and implemented design decision, and to consider such change for a product already in production.

For example, changing a request or a response schema would be architectural, because it would require changes to consumer and consumption behavior. For the same reason, changing requirements for how consumers must identify themselves would also be architectural.

On the other hand, changing from not using a specific internal design pattern to using it, or from using one application block to using another, would not affect usage behavior. The implementation might be better or more efficient, but that would not affect usage behavior, so it would not be architectural but engineering.

Again, some would disagree, arguing that such a change would possibly change performance and thus at least indirectly affect usage behavior. That might be true, but the real question would be whether the original design failed to meet the original performance requirements, or if the original performance requirements were changed. In such a case, the performance requirements are certainly architectural, but the way they are met are not. For the user/buyer, the fact that the requirement is met is what counts, not *how* it is met.

Mapping Owner and Usage Requirements

Software architecture is about mapping! It's about mapping owner and usage requirements to a technological solution. Or rather the other way around, to define a technological solution that maps to owner and usage requirements.

This is the job of a generalist more than that of a super technologist. The software architect must have hisⁱⁱ feet firmly placed on both sides of the borderline. He must be able to understand business problems, business processes, business strategies, business goals, and business jargon well enough to be able to map business requirements and software artifacts to each other. He must be able to understand technological opportunities, shortcomings, risks and costs well enough to come up with a software architecture which solves the business problem, and which is implementable at a reasonable cost for the owner. But he is *not* required to be able to *implement* that design, or even to design the implementation.

How Much Details Are Involved?

During our architectural classes we have often heard that some work is at a level too detailed for the architect to bother about. The underlying assumption seems to be that the borderline between architecture and engineering should divide less detailed from more detailed specifications. For example, the architect shouldn't bother with the detailed specification of a WSDL file. That should be the responsibility of the software engineer.

We don't believe this is the best way to separate architecture from implementation. The architect *should* be concerned with all the details necessary to achieve a proper mapping, from business and usage requirements to a rigid specification of the externals of services. This should *include* detailed specifications of the data that must flow between consumers and services, no matter how detailed these specifications are.

These specifications, together with equally detailed specifications of the *behavior* expected of a service and its consumers, form the *contract* that must be obeyed by both consumer and service. Specifying this contract is the architect's responsibility, and it is part of the mapping of technological artifacts to business requirements. The resulting map then acts as requirements for the *engineer* of the service, and it also represents *restrictions* the engineer must relate to. The engineer is free to decide how to *implement* the contract, but he is not free to *change* it at will, no matter how strong his technical reasons would be. Together with the owner of the service, it's the software architect who owns the contract and is responsible for it.

Commonly Drawn Borderline is Fuzzy

We at 2xSundblad fully understand that most of the now practicing senior developers that are thought of as architects don't share our view about the need for a sharp borderline between architecture and engineering, or about the position of such a line. It's much more common to include a fair deal of technical design in architecture. In fact, it's very common not to distinguish software architecture from software engineering at all.

This is very similar to the situation with traditional architecture during the Renaissance. Architects such as Michelangelo and Leonardo da Vinci, and many others, played all the three roles of architect, artist, and engineer for the projects they were involved in. They could, even though they were generalists, because the technologies and materials available to them weren't overly complex. To quote Wikipedia on architecture, "it was still possible for an artist to design a bridge as the level of structural calculations involved was within the scope of the generalist".

As new technologies and new materials became available, architects found it increasingly difficult to manage the technical aspects of building. It became necessary to separate the two roles. The architect would set function and esthetic values for the user in focus, striving to design a product which would solve the problem it was supposed to solve, and give pleasure to users while using – and even looking at – the product. The architect would also be responsible for the feasibility of building the product at a reasonable cost. But he would not be responsible for its internal and technical design. That responsibility would fall on the engineer.

Software architecture is now largely in a situation which reminds one of the situation for traditional architects during the Renaissance. Up till now, most modern software has been built as end-to-end applications, composed by tightly connected binary components. The major way of consuming such a software application is by way of a user interface, which means that the surface exposed for consumption is both small and simple, while the internals might be huge and complex. For such an application, there's not much room for architecture in the sense we use the concept in this article, but there is much need for great technical design.

SOA Changes the Ball Game

Service-Oriented Architecture (SOA) changes this quite a bit. In a SOA environment, business software is no longer built as end-to-end applications. Instead, it is built as systems of loosely coupled user applications and services, which communicate by sending language- and platform-independent messages to each other.

Typically, in such an environment, services tend to be parts of multiple systems. For example, consider a Products service! It might start its life as part of a sales system. Later it might be involved in a purchasing system, a product development system, a marketing system, a warehousing system, and perhaps in several other systems too. This process may take years, and it really never ends.

The service is the same, but its responsibilities and its external exposure are increased with each system it's enrolled in. Such a service must be able to adapt itself to new requirements, and to new users, over a long period of time. To make this possible, the service must be architected in total independence of the service's internal structure. The architect must be able to see the service as a black, impenetrable box to which he can add or modify service interfaces as needed by consumers of the service.

Much more than when architecting end-to-end applications, the service-oriented software architect must also consider *enterprise* architecture when architecting a software solution to a specific business

problem. The enterprise architect has, or should have, a vision of a future, electronic, serviced enterprise. Every new or modified software solution, and every *part* of such a solution, should be another piece in the enterprise architecture puzzle, making that vision come just a little bit closer. This way, the software architect is not only responsible for the software system at hand, but also for the way it will fit into its present and future environment.

It is rightfully claimed by many that SOA has a potential to increase business agility. In other words, businesses building up a SOA environment should be able to change themselves more quickly and with less pain than other businesses. For this to come true, however, software architecture must be well aligned with business architecture. This does not happen by itself; the software architect must understand business architecture, and the need to align software architecture to it, to make that happen. This requires new insights and new competencies and makes the game a little bit different from before.

So, service-oriented software architecture is different from, and in several ways more complicated than, end-to-end application architecture. What's more, it's not nearly as technical, which is especially apparent when you consider the architectural requirement of being able to see each service as an impenetrable black box to be able to freely add features to its external exposure as needed by consumers.

Implementing a service is also more challenging than implementing an application. This is partly due to the expectation of one and the same service to play different roles in different systems. It's also due to the distributed character of service-oriented systems, which requires consideration of platform- and language independency, security, message reliability, complex management of transactions, idempotency, and much, much more.

So, moving away from a culture of end-to-end application development to a service-oriented culture, with all its new challenges, stimulates the idea of separating architecture from implementation, and the architect profession from the engineering profession. It's not very different from what happened in traditional architecture and engineering some time after the Renaissance, when traditional architects moved away from engineering tasks.

No Grand Master Plan – Organic growth

Many senior developers and software engineers warn businesses and developers about SOA. More often than not, the reason seems to be their assumptions that SOA is almost married to the waterfall type of development process, and that it totally requires a so-called "Big Up Front Design" or is based on a Grand Master Plan. Such a plan would be developed by some theoretical person sitting in an ivory tower. Luckily, both of these assumptions are false.

A while ago we were involved in a discussion with a colleague (C). C told us about a customer who had decided to go for SOA in a big way, and to stop all software development in order to develop a business architecture, reengineering many or most of their business processes. With that new business

architecture in place, software development could be started again. The reason for this was the company's fast growth, which probably (my conclusion) was inhibited by the structure of their software.

C was very critical to that decision, because he felt that the customer needed new software now, especially because their business was growing so fast. He recommended the customer *not* to do a business architecture but to continue having software developed the same way as before.

All this tells us that both C and his customer saw this connection between SOA and a Grand Master Plan as unavoidable. C lost his customer on this, and the customer went on to seemingly build that Grand Master Plan on which to found new systems about three years later.

It's so tragic, because SOA doesn't require a Grand Master Plan or a Big Up Front Design at all! In fact, if you *do* try to set up a Grand Master Plan on which to base future system development, as C's customer at least *planned* to do, then you're almost certain to fail. If C had understood this, he could have convinced the customer that his plan to establish a business architecture on which to base service-oriented development was great, but also that his plan to postpone all software development until the business architecture were done was a grand mistake!

It's probably true that you need an enterprise architecture in order to succeed with SOA. The enterprise architect should set up a team of business managers, business architects, and software architects to establish a vision of the future electronic serviced enterprise. It should not have to take more than about a month to establish a first and workable version of such a vision, and it should not involve more than one or two people of each of the categories mentioned. The vision would then act as a guiding framework, helping the enterprise, business, and software architects to work together, getting every piece of non-trivial business software well aligned with the business and its architecture.

Working from such a vision, services will grow organically, as needed by real people in real jobs, and the entire business system portfolio will grow organically from its parts. We say this often, and we firmly believe in it. There's not much room in this article to elaborate any further about these ideas of "No Grand Master Plan" and "Organic Growth", but you can learn much more about them in an upcoming course which we plan to publish in an online format in the summer of 2007. You don't even have to buy the course for this information; part of it is in a free preview, already published well in advance of the real course. Look for information on this preview at <http://www.2xsundblad.com/archtraining/archsystems/preview>.

As a matter of fact, the course contains more information about how to establish that vision of a future electronic serviced enterprise too, but that's not in the preview, just in the complete course.

What about User Interaction and User Interfaces?

As so often before, I can't find the room needed to talk about this important subject, at least not in this article. I'll have to postpone it to a possible future article about that subject alone. However, it's not possible for me to leave this subject totally out of an article about the responsibilities of a software architect.

As I have already said in this article, I firmly believe that that software architecture is about mapping business and usage requirements to technological artifacts, and that it's about designing the externally visible parts of the user applications and services involved in the resulting solution. Knowing that, it should be crystal clear that I see user interaction and user interfaces as extremely important parts of software architecture.

The software architect can't afford to ignore this increasingly important interface between users of business software and the software itself. If he does, he turns the responsibility for the ultimate usage experience over to the engineers, and to the developers.

You might ask why that would be so bad! Well, in my mind, if there's anything the architect should be responsible for, it's usage experience. In a service-oriented environment, there's a lot of usage experience. There's the software developer's experience of consuming a service interface – or a service endpoint if you prefer that term.

If the operations of that endpoint are well defined, described, and documented, if they are properly named so their business reasons-to-be are clear, and if the operations collected in this endpoint belong together for business reasons, then chances are that the consuming developer's experience of using this endpoint will be awesome. On the other hand, if they are badly collected, badly named, badly documented and so on, then the usage experience will probably be rather bad.

But there's also the end user's experience of using the system to think about. Today, there are so many options for the design of user interaction to choose from. There's the ordinary wide-reaching web application, and there's the less wide-reaching but richer Windows application. In both cases, there are also different form factors to choose from.

Another option is to make an application available through a portal, or, recently, to use new options such as the so-called Web 2.0 and its offspring. I believe it's fair to say, for example, that Microsoft's new technology called Silverlight could be thought of as an offspring to Web 2.0.

Each of these options is different in terms of the kinds of usage experience they can offer. You can't say that one is inherently better than the other – it all depends on what is important to achieve. With so many so different options to choose from, the choice made for a certain software system can make or break that system. This is one reason the software architect can't afford to ignore the end user experience. After all, the software architect should always have usage experience as a guiding-star, so how could he defend ignoring those users for which the system most likely is being built in the first place?

There's so much more to say about this, but, as I said, I'll have to save that to another and later opportunity.

Some Final Observations

Also notice that business requirements aren't just about the functionality needed. They are also about things such as security, performance, operations, cost of development and operation, and much, much more. Therefore, the architectural specification must consider all of these kinds of non-functional requirements. Typically, this would require close cooperation between the architect and the engineer, both before and during implementation design. However, this doesn't change the general picture of what should be architecture and what should be implementation design.

By the way, if the person playing the architect role and the person playing the engineering role is one and the same, such cooperation should be very smooth. Seriously, I do believe that many capable senior developers with a business interest should be able to play both these roles. Still, for such a person, it would be extremely important, even if difficult, to separate the architectural work from the engineering work he does.

Finally on this key, notice that architecture could be used not only as a blueprint for implementation, but also as a blueprint for a buying decision. If a product exists in the open market and can be bought, it's great to have an architecture which represents *your* business's requirements to compare to the architecture of the product you're thinking of buying. Such use of an architectural blueprint should reinforce the value of keeping implementation design out of it.

There's also the question of contracting the implementation of different parts of an architecture to different organizations, perhaps even in different countries. The more your architecture consists of specialized applications and services, such as user applications, business process and use case services, entity services, and infrastructure services, the more interesting will it be for you to hire companies specializing in implementing a certain kind of services. This happens today, and we have customers who works as subcontractors in this sense as well as customers who subcontracts implementation tasks in this way.

Having said all this about the difference between software architecture and software engineering, we must once again say that the software architect or architects must work very closely together with the software engineer or engineers involved, be they the same person, different persons in the same organization, or different persons in different organizations. They must work together, both to ensure the best possible architecture, given all circumstances, and to ensure the best possible implementation of that architecture. They belong to the same team, and they should do everything in their power to achieve the best end result that team is capable of.

ⁱ Frederick P. Brooks Jr., *The Mythical Man-Month*. Addison-Wesley. ISBN 0-201-835953.

ⁱⁱ The words he, him, and his, as used in this article, should be interpreted as "he or she", "him or her", and "his or her".