HELLO2MORROW

The Value of Architecture

Alexander von Zitzewitz

hello2morrow Inc.

“If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.” - Gerald Weinberg

Overview

Software architecture has value in itself and is a critical factor determining the total cost, maintainability and success of a software development project. But in reality many software projects fail or never reach their true potential due to the erosion or lack of architecture.

After describing minimal requirements for designing and maintaining an architecture this paper will highlight the areas where architecture provides real value for a software project. It will also look at software architecture in the context of an agile project.

In the scope of this paper I define (software-) architecture as the decomposition of a software system into smaller manageable units (called architectural artifacts) and establishing rules defining allowed and forbidden dependencies between those artifacts. The artifacts on the highest level can be decomposed again into smaller lower-level artifacts and this process should go on recursively until the typical size of an artifact is small enough so that it can be easily maintained and understood by a single person. A good software architecture always tries to minimize the number of allowed dependencies between artifacts and never allows cyclic dependencies between artifacts. The architecture therefore describes the large-scale structure of a software system.

Designing an Architecture

The next logical step after gathering the initial requirements for a project is the design of an initial architecture. Like it is often impossible to gather all requirements at the beginning of a project, it is also not necessary to have a complete architecture that describes every detail and aspect of the system before coding begins. But a couple of important questions have to be answered upfront:

- What are the major components of my system and how do they depend on each other? (These are your architectural artifacts on the highest level)
- How do I organize my code?
- How do I build my system?
- What will be the artifacts created by the build?
- How do I organize cross-cutting or general functionality like persistence, logging, authentication etc. ?
- What is my general strategy for technical layering, e.g., where do I put

my application logic, how do model objects interact with each other, etc.?

- How and where do I validate data entering my system from the outside, how do I keep my system secure?
- If scalability is an issue, what are my scalability options?
- What are my deployment options?

The last two questions do not mean that you already have to answer them in detail, but you should have scalability and deployment in the back of your mind, because these aspects might very well have a big influence on the organization of your code.

Identifying the major components will probably help you to decide about the physical layout of your project and how you structure it in your IDE.

The code organization aspect is also very important because it determines the physical layout of your project in the developers workspace. It defines rules about naming and structuring packages and/or directories and how architectural artifacts are mapped to this physical layout.

There is nothing wrong with keeping it simple at the beginning, things will get more complex later anyway. As your project grows you will refine the initial large scale architecture by decomposing the highest level artifacts into smaller sub-artifacts.

If you are using Java, many of those questions are already answered for you, if you build your system on the base of the Spring Framework, which is highly recommended for Java enterprise applications. If you are using Ruby on Rails, most of those questions are also already answered by the design of the rails framework.

When designing your architecture, flexibility should always be an important goal. Since we usually do not know all the requirements at the beginning of the project and requirements change frequently anyway we have no other choice than trying to keep our architecture flexible so that it will be able to accommodate unforeseen features.

Maintaining an Architecture

After you have designed the initial architecture your biggest problem will be to make sure that it is actually reflected and respected by your code base. Many projects start with a solid initial architecture, but fail to ensure that the code is actually based on it. Architecture validation should be part of your build process. If you do not enforce architecture and validate it in an automated way it is very likely that your system will suffer from growing architectural erosion.

Architecture as an Enabler

If you invested the effort to design an initial architecture you should be able to reap the first benefits when coding starts. For every piece of code added it should be clear to which architectural artifact it belongs. Knowing that also defines on which other pieces of code you can depend upon and what other parts of your system should not be used by the

new code. Moreover you know exactly where to put the source code within the project structure.

If you stumble upon a new class (or other piece of code) where you don't know where it belongs in the architecture it means you will have to refine or change your architecture or think about a different way to add this specific functionality to your project.

While this approach requires you to think a bit more upfront and make certain decisions early in the process it also greatly simplifies the daily development work. That already provides value on its own, because it improves productivity and ensures that everybody in the team works on the basis of the same architectural rules and principles, which greatly increases the probability of a successful outcome.

More value is provided by the fact that the code is easier to read and understand, because it is well organized and structured. Adding new features usually requires fewer code changes, because a clear structure also helps to minimize coupling. Fewer code changes also mean fewer regression bugs and therefore a better quality.

Another important example for the value of architecture are security aspects. If your system has a good structure the interfaces where external data can enter your application should be pretty obvious. If you know that your system does not contain any unwanted dependencies (by validating your architectural rules in the build process) you can focus your scrutiny on these interfaces and make sure that all data coming from the outside is properly validated.

Without that certainty maintaining a high level of application security is much harder, because it requires you look at every potential data flow path in your whole system. It is easy to see that this requires a much bigger effort than maintaining a clean architecture and structure from the very beginning.

In other words, designing and maintaining a good and flexible architecture enables you to focus the energy of the development team on adding value to your application. It enables you to add new functionality with a reasonable effort. It enables you to keep the application productive for a long time while keeping the maintenance cost under control.

A broken architectural structure on the other hand works like sand in the gearbox. Since you avoided the effort to keep the architectural structure in good shape, you saved some time, but you also accumulated structural debt. Like debt in the real world structural debt requires interest payments in the form of increased effort for everything you do on your system. Moreover increasing structural debt makes your "credit rating" go down, so that the interest rate you have to pay increases to unsustainable levels. The point of bankruptcy is reached when the effort for code changes becomes prohibitively high so that your only choice is to throw the old system away and start from scratch. In that case you defaulted on your structural debt.

There are many real life case studies and examples out there that prove that accumulating structural debt can only give you short term benefits. Sometimes you have no other choice to meet an important deadline. But always keep in mind that you have to pay it back and the sooner you do it, the less interest you will have to pay.

Architecture and Agility

Agile development methods gain more and more traction in the software industry and there are many good reasons for the success of agile methods. Agile development is centered around “User Stories”, software usage scenarios that bring value to the end user. Moreover the process is based on short iteration cycles, where every iteration is supposed to produce tangible user value. Since software systems are built for real users it is certainly the right idea to focus the development process on the creation of tangible user value and early user feedback.

A problem can occur when the agile methodology is taken to the extreme where “User Stories” are considered as free floating independent items. “User Stories” are not floating in space, they all live in a context and usually have many obvious and subtle functional and technical dependencies between them.

A pure agile approach would pick the user story with the highest perceived user value as the first thing to be implemented in the new system. If you look at that “User Story” in isolation, the best way to implement it might be very different from the best way to implement it when taking other associated user stories and the context of the whole system into consideration. Therefore it is a smart idea to put the “User Story” into the context of the whole system and other “User Stories” that need to be implemented.

From a technical point of view the context is the architecture of the software system. With a solid architecture in place it is easy to implement the “User Stories” within the frame of the architecture. But of course the design of that architecture also requires that some key “User Stories” and the overall system context are known by the designer.

If you, on the other hand, decide to make architectural decisions on the fly like some people from the agile camp suggest, you might quickly come to a point, where your architectural structure becomes messy because you made architectural decisions without sufficient context information. Of course it is possible to refactor your code to improve your architecture, but with growing system size this option becomes less and less practical and realistic.

The trick is to find a balance between providing enough architecture upfront without loosing too much time with thinking about problems that can be solved further down the road.

The approach of combining just enough upfront architectural design with agility is called “Architectural Agility” and is presented nicely in [BRO], a paper from the Carnegie Mellon Software Engineering Institute.

References

[BRO] http://www.crosstalkonline.org/storage/issue-archives/2010/201011/201011-Brown.pdf