JJ's Reference Architecture

Author: Jan-Joost van Zon
Date: December 2014 – July 2017
[Under Construction]

Introduction

Contents

Definition of Software Architecture

Software architecture has thousands of definitions. If you believe them all, then software architecture is about everthing that has to do with software development.

Mostly it is about actually building the software.

It is about modular design of software components, which can be done by splitting up code into functionalities, like a shopping basket, or financial reports, or splitting up code into technical aspects, like validation, security and persistence. You can also split up by both technical aspects and functional aspects at the same time, giving you a more fine-grained (2-dimensional!) separation of concerns.

Software architecture also has a strong focus on maintainability and being prepared the future as a system grows.

The programming side of software architecture is about making frameworks, coding the functionalities, combining different technologies and using best practices. It also involves technical design, which by the way you can do in your head as well as on paper.

Inter-Disciplinary Aspects

Software architecture also involves technical details outside of software programming, such as the basic outlining of hardware infrastructure, collaborating with infrastructure technicians, server administrators, hosting providers, software vendors.

Software architecture also includes soft-skills that do not have much to do with technology. Planning the development of software both in rough outlines as well as task details, guarding that work, prioritizing, organizing and replanning, making concessions, work preparation,

managing software lifecycle, going from design to implementation to test to production and after care, having proper source control in place, managing the team that codes, the team that tests, discussing functionalities, goals and planning with management, stakeholders, staff and end-users. Basically talking to anyone even slightly involved in the development of the software. Coaching developers, expanding the teams knowledge, making the team work optimally together and efficiently, and give people room to focus, so a lot of work gets done well. It can involve managing budgets for hardware and software and also functional designing.

Fortunately this does not need to come down to one person. Even though a software architect can overview the whole process, lots of tasks can be delegated to other team members, so you can make software architecture work as a team.

Possible Choices over Definite Choices

This is kind of a personal note on where this document stands right now.

Originally I described a fixed way of working here, that generally works well if you want to build large dependable systems with a lot of flexibility. I applied these methods of working in a team under my lead. It worked, but required a lot of discipline of team members to do things the way the boss wants.

I want to move away from this a little bit, and see the methods described here more like a suggestion box of different ways to do things. I will try to describe different alternatives next to the one I prefer and highlight the pros and cons, so you can perhaps see why I came to the conclusion that one method is better than the other.

Much of the document is still described in definites, rather than suggestions. As I find the time to work on this documentation, I will be changing the tone.

What you will also find is that I describe a lot of things you could do wrong. The suggestion is often that there is a better way to do it. I will try to reformulate things so they start with a positive approach rather than starting with the negative.

Currently (2017-06-28) it is full of TODO's that indicate texts I still want to write or rough texts to polish up. So please be forgiving of those.

But, now: back to business.

Technical over Functional

This document mostly goes into detail about technical aspects of software architecture: those aspect of software development, that go beyond the individual application: techniques and best practices that can be applied to the development of *any* application.

You could call it '*functionality-independent software architecture*'.

You could also call this architecture a '*pattern stack*', because it takes a technical-first approach, rather than functional-first in that a layering is described where fixed design patterns are used from one layer to the next.

The idea behind this is that even though technology changes fast, functionality changes faster. So a technical-first order will be less likely to change, and this results in a more stable subdivision into parts upon which the functionality builds.

To also accommodate for quickly changing technology, we use abstractions of these technologies to be able to replace them and not have to reprogram the whole application if we make a switch.

This gives us a subdivision into parts into which everything fits, even when not everything is put in yet.

Application Architecture vs Service Architecture

There are two parts of this software architecture:

- Application architecture
- Service architecture

The *application architecture* is the main part. It is about business domains and everything you could show on a screen, including a framework of reusable parts. The *service architecture* is explained separately and is mostly about linking different systems together.

The way of working described here is just a suggestion. It describes *a* way of working, not *the* way of working. The described principles and practices can be used at will.

Fundamental Principles

The main principles of this software architecture are:

- Maintainability
- Code scalability
- Platform and protocol independence

Code scalability does not refer to hardware scalability, but rather that the code base can grow and grow, while keeping it maintainable. 50 apps should be as maintainable as 5. This means that quality demands are high. As a companies' amount of software products grows, software architecture is necessary to create an economically viable amount of software maintenance, or a company might run into problems.

Another way of putting this is: The next software change should not be more difficult than the previous one, regardless of how large the system has become.

Platform and protocol independence is something given extra attention in this software architecture. A lot of split up into parts is, due to the fact that not every technology is supported on every platform. This allows us to take our pick from technologies more easily.

This software architecture also puts a lot of focus on fixed patterns of working. These patterns are proven to work well, and if we all work the same way and understand the system of organization, we can more easily navigate the code, regardless of who wrote it.

Top 12 Improvements

This document goes into detail about a lot of best practices. But to keep focus on what goes wrong most of the time, and would offer the greatest improvement of code, here is a list of practices, that if done right, may greatly increase the quality of your software.

1. Avoid code duplication
2. Avoid error hiding
3. Use clear names
4. Use correct indentation
5. Separation of concerns
6. Proper encapsulation

And second in line:

7. Reject, don't correct
8. Clear interfaces
9. 'No handy extras' / 'Ya ain't gonna need it'
10. Solve a problem at its core, instead of making work-arounds.
11. To solve a bug, first reproduce it
12. Proper use of encoding