Software Architecture

February 21, 2022

Brae Webb & Richard Thomas

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Software Architecture

February 21, 2022

Brae Webb & Richard Thomas

1 Introduction

An introduction to Software Architecture would be incomplete without the requisite exploration into the term 'software architecture'. The term is often overloaded to describe a number of completely detached concepts. The overloaded nature of the term makes an introduction quite challenging. Martin Fowler wrestles with this difficultly in his talk on "Making Architecture Matter"[1]. In the talk Fowler settles on the slightly vague definition from Ralph Johnson [1]:

Definition 1. Software Architecture

The important stuff; whatever that is.

In this course, we will try to narrow the scope slightly. We need a definition which encompasses the numerous practical strategies which you need to survive and thrive in industry life. This definition should not be attributed to the term 'Software Architecture'; that term is much too broad to define succinctly. This is purely the definition used to provide an appropriate scope for the course.

Definition 2. Software Architecture: The Course

The set of tools, processes, and design patterns which enable me to deliver high quality software.

2 High Quality Software

We assume that as software engineers you wish to deliver high quality software systems. What makes life interesting[2] is that *quality*, like *beauty*, is in the eye of the beholder. As a diligent and enthusiastic software engineer, you may think that *high quality* means well designed software with few defects. On the other hand, your users may think that *high quality* means an engaging user experience, with no defects. While your project sponsor, who is funding the project, may think that *high quality* means the software includes all the features they requested and was delivered on-time and on-budget. Rarely is there enough time and money available to deliver everything to the highest standard. The development team has to balance competing expectations and priorities to develop a software system that is a good compromise and meets its key goals.

From the perspective of designing a software architecture, competing expectations provides what are sometimes called *architectural drivers*.

2.1 Functional Requirements

A seemingly obvious driver is the functional requirements for the software system, i.e. what the software should do. If you do not know what the software is meant to do, how can you design it? You do not need an extensive and in-depth description of every small feature of the software, but you do need to know

[1] https://www.youtube.com/watch?v=DngAZyWMGR0

[2] As in the apocryphal Chinese curse "May you live in interesting times."

what problem the software is meant to solve, who are the key users, and what are the key features of the software. Without this information, you do not know what style of architecture is going to be appropriate for the software system.

For example, consider an application that allows users to write and save notes with embedded images and videos. Say the decision was made to implement it as a simple mobile app that saves notes locally. If it is then decided that web and desktop applications are needed, allowing users to sync their notes across applications and share notes with others, the application will need to be redesigned from scratch. Knowing up-front that the software needed to support multiple platforms, and that syncing and sharing notes was important, would have allowed the developers to design a software architecture that would support this from the start.[3]

2.2 Constraints

Constraints are external factors that are imposed on the development project. Commonly, these are imposed by the organisation for whom you are building the software system. The most obvious constraint is time and budget. A sophisticated software architecture will take more time, and consume more of the total budget, than a simple but less flexible architecture.

Other common constraints are technology, people, and the organisation's environment. Technology constraints are one of the most common set of constraints that affect the design of the architecture. Even if it is a "greenfields" project[4] there will usually be restrictions on choices of technology that may or may not be used. For example, if all of the organisation's existing applications are running on the Google cloud platform, there will probably be a restriction requiring all new applications to be built on the same platform to avoid the overheads of working with different cloud providers.

People constraints takes into consideration the skills that are available within the organisation and the availability of developers for the project. Designing an architecture that requires skills that are not available, will add an overhead to the project's development cost. If contractors can be hired, or training is available, these may reduce the overhead but the decision needs to be made based on the risks and benefits.

The organisation's environment may influence other constraints, or add new constraints. An organisation that encourages innovation may be flexible in allowing some technology constraints to be broken. If the project is of strategic value to the business, there may be options to negotiate for a larger budget or to adopt a new technology[5], if they could lead to a more robust solution with a longer lifespan. Politics can introduce constraints on architectural choices. If there is an influential group who promote a particular architectural style, it may be difficult or impossible to make different choices.

2.3 Principles

Principles are self-imposed approaches to designing the software. Typically these are standards that all developers are expected to follow to try to ensure the overall software design is consistent. From a programming perspective, coding standards and test coverage goals are examples of principles that all developers are expected to follow. Architectural principles typically relate to how the software should be structured and how design decisions should be made to work well with the software architecture. Consequently, principles usually do not influence the architecture[6], rather the architecture will influence which principles should be prioritised during software design.

As an example, if the software architecture is designed to be scalable to accommodate peaks in load, then an architectural principle might be that software components should be stateless to allow them to

[3] This is different to building a quick-and-dirty prototype to explore options. That is a valid design process, and in that case minimal effort will be put into creating a well-designed app.

[4] This refers to the idea that it is a new project and is not limited by needing to conform to existing system's or expectations.

[5] I have consulted with organsiations who have adopted new, and risky at the time, technologies to potentially gain business benefits like easier extensibility.

[6] The exception to this is if some principles are constraints enforced by the organisation.

be easily replicated to share the load. Another principle might be that an architecture that relies on an underlying object model, will adopt the SOLID design principles [2].

2.4 Quality Attributes

While the functional requirements specify what the software should do, non-functional requirements specify properties required for the project to be successful. These non-functional requirements are also termed *quality attributes*.

Often quality attributes are specified by phrases ending in -ility. Medical software needs *reliability*. Social media needs *availability*. Census software needs *scalability*.

Below is a collection of non-exhaustive quality attributes to help give you an idea of what we will be looking at in this course.

Modularity Components of the software are separated into discrete modules.

Availability The software is available to access by end users, either at any time or on any platform, or both.

Scalability The software is simultaneously usable by a large amount of end users.

Extensibility Features or extensions can be easily added to the base software.

Testibility The software is designed so that automated tests can be easily deployed.

Quality attributes are one of the main focuses of a software architect. Quality attributes are achieved through architecture designed to support them. Likewise, software architecture quality and consistency is achieved by principles put in place by a software architect and the development team.

Architects are responsible for identifying the important attributes for their project and implementing architecture and principles which satisfies the desired attributes.

2.5 Documentation

The importance of these architectural drivers means they need to be documented[7]. The extent and format of the documentation will depend on the size of the project, team organisation, and the software engineering process being followed. The larger the team, or if the team is distributed between different locations, the more important it is that the documentation is well-defined and easily accessible. The documentation may be stored as notes in a wiki or as a collection of user stories and other notes. Or, it could be an extensive set of requirements and design specifications following a standard like ISO/IEC/IEEE 15289[8].

3 Attributes in Tension

One of the defining characteristics of quality attributes is that they are often in conflict with each other. It is a valiant yet wholly impractical pursuit to construct software which meets all quality attributes.

The role of a software architect is to identify which quality attributes are crucial to the success of their project, and to design an architecture and implement principles which ensure the quality attributes are achieved.

[7] Documentation is the castor oil of programming, managers know it must be good because programmers hate it so much [3].

[8] https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:15289

The first law of software architecture, as defined by Richards [4], reflects the difficultly in supporting multiple quality attributes.

Definition 3. The First Law of Software Architecture

Everything in software architecture is a trade-off.

Galster and Angelov [5] define this as 'wicked architecture'. They identify the 'wicked' nature of architecture as the main difficultly in teaching the subject.

Definition 4. Wicked Architecture

There are often no clear problem descriptions, no clear solutions, good or bad solutions, no clear rules when to "stop" architecting and mostly team rather than individual work.

They stress that "In contrast, other software engineering topics such as programming lead to solutions that can be checked and measured for correctness. Architecture may be the worst-case scenario when it comes to fuzziness in software engineering".

Despite this difficultly, in this course we intend to expose you to a number of case studies, architectures, and tools which aim to give you experience in tackling the trade-offs involved in software architecture.

4 The World Today

Software architecture today is more important than ever. The importance of architecture can be considered a result of *expectations* and *infrastructure*. Today we expect our software to be available 24/7. To exemplify this point, in October last year Facebook went offline for 6-7 hours out of the 8760 hours of the year. Those 6-7 hours of downtime resulted in mass media coverage, $60 million loss of revenue, and a 5% drop in company shares which caused Zuckerberg's wealth alone to drop $6 billion. Interestingly, the outage also caused other sites such as Gmail, TikTok, and Snapchat to slowdown.

This is a massive change in public expectations for software availability. As recently as 2017, human resources at UQ would monopolise the university's computing resources for an evening each fortnight to calculate the payroll. Nowadays that lack of availability from even the university's software systems would be completely unacceptable. The change in expectations has forced developers to adapt by designing architectures capable of supporting this heightened up-time.

In addition to shifting expectations, developers now have access to a range of Infrastructure as a Service (IaaS) platforms. IaaS empowers developers to quickly and programmatically create and manage computing, networking, and storage resources. In part, this enables individual developers to support up-times comparable to tech giants. Of course, to be able to support these up-times software has increased in overall complexity. A website is now commonly spread over multiple servers, marking a change from centralised systems to distributed systems.

5 Conclusion

You should now have some understanding of what software architecture is and that it is, at least in our view, important. Let's return to our definition for the course.

Definition 2. Software Architecture: The Course

The set of tools, processes, and design patterns which enable me to deliver high quality software.

In practical terms, what does this mean you should expect from the course? First, you will learn how to communicate your visions of software architecture through *architectural views*. From there, you will use

quality attributes such as extensibility, scalability, etc. to motivate the introduction of common architectural patterns, processes, and tooling which support those attributes.

For example, you can expect extensibility to motivate an introduction to plugin-based architectures. You can expect scalability to motivate our introduction to load balancers. And testability to motivate a look into A/B testing practices.

You can view the planned outline for the course on the course website[9]. All the course material can be found on GitHub[10]. If you think you can explain a concept more succinctly, or find a typo, you are encouraged to submit a pull request to help improve the course. We really hope that you enjoy the course and, perhaps more importantly, benefit from it in your careers as software development professionals!

References

[1] M. Fowler, "Software architecture guide." https://martinfowler.com/architecture/, August 2019.

[2] R. C. Martin, "Design principles and design patterns." https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf, 2000. Accessed: 2022-01-10.

[3] G. M. Weinberg, *The Psychology of Computer Programming*. USA: John Wiley & Sons, Inc., 1985.

[4] M. Richards and N. Ford, *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, Inc., January 2020.

[5] M. Galster and S. Angelov, "What makes teaching software architecture difficult?," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pp. 356–359, Association for Computing Machinery, 2016.

[9] https://csse6400.uqcloud.net/

[10] https://github.com/CSSE6400/software-architecture