Key Factors For A Successful Architecture

Sapir Shemer

Software architecture has evolved throughout the years to become a subject with a lot of breadth and depth. Many tools and conceptual frameworks have been developed to help architects do their work. Although there are a lot of tools for architecture, if you are like me, then your goto is probably pen and paper. And I mean **a lot of paper**.

Actually, I use so much paper that I got tired of my room being in such a mess from dozens of stacks of paper, so I bought myself a paper tablet for my work, which I do recommend if you are a pen&paper maniac like myself.

If you are already a software architect then you're aware that software architecture doesn't require more than pen and paper to get started (and even finish), as it is all conceptual and starts off messy until you organize everything in a concise, determined and understandable manner.

The key idea to making a good architecture is not only knowing the contemporary literature and definitions, but also having the experience to work through the architecture in the correct order, then to summarize it in a simple to understand format for developers and designers. As of now, most of the knowledge and literature about software architecture is more concerned with precise definitions instead of the process itself of architecting software.

Ironically, in contrast to what I just mentioned, we will start with the first concept of a good architecture, which is:

1. Understand the Definitions

> *"Would you tell me, please, which way I ought to go from here?"*
> *"That depends a good deal on where you want to get to," said the Cat.*
> *"I don't much care where–" said Alice.*
> *"Then it doesn't matter which way you go," said the Cat.*
> *"–so long as I get SOMEWHERE," Alice added as an explanation.*
> *"Oh, you're sure to do that," said the Cat, "if you only walk long enough."*
>
> *- Lewis Carroll, Alice's Adventures in Wonderland.*

Being void of definition may cause you to wander around making assumptions and bad design choices, in the end you will still achieve some description or abstraction of your intended software, but it will take more time, and will not be as optimal.

We are architecting software, so first and foremost we must understand what software is exactly? What purpose does it serve? Hardware is the physical ordering and structure of electronic devices, while software is the ordering and structure of hardware instructions (using readable code) to achieve the transmission and transformation of data within different physical devices.

Strictly speaking, *software is all about data*. Software engineers/architects are working to design and formulate a system that circulates, manipulates and processes data between different devices that interact with the environment. So when we architect software, we try to figure out how we can abstract our data as structures, transformations and transitions of data as interfaces and then implement them by the technologies and hardware available in a manner that achieves the desired requirements from our software.

I'm a big fan of academic literature as it strives to be very precise. Roy T. Fielding (The godfather of the REST architectural style) defines software architecture in his dissertation, that I recommend every aspiring architect to read, as:

> **Software Architecture** is an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture.

What I like about his dissertation is that he presents software architecture as an abstraction of a running system, rather than a formal description. This results with a slightly different outlook and use of architecture than the original definition of architecture by Perry & Wolfe in their famous paper "Foundations for the Study of Software architecture", which literally marked the foundations for the study of software architecture. In a strict sense, Roy's definition doesn't include rationale as part of a system's architecture.

As architects we define the elements of an architecture and their constraints that manifest their properties and relationships. Elements, as Perry & Wolfe define them, are components, connectors and data. Which have the following specifics:

- **Datum**: An element of information, may contain information in a structure that holds other data.
- **Component**: An abstraction of transformations or transmissions of data served as an interface, might contain an internal state.

There are two things to note here. First, notice that I've excluded a "connector" as a basic element of an architecture, as it is just a component that is used to transmit information for cooperation, communication and mediating between components. In other words, it is a subcategory of a component and not an individual decoupled definition. However, to not make all the world angry at me, here you go:

> **Connector**: An abstraction of transmissions of data between components for mediating coordination, cooperation and communication.

Secondly, the definition of a Datum here is different from the one proposed by R. T. Fielding, as he claims that a datum is only relevant to information that is passed between components via connectors, and doesn't include information that is permanently resident or hidden within a component. In contrast, resident/hidden information inside a component might have an effect on the reliance of a component on other components and vice versa. Thus, the definition here frees the architect from this constraint.

Whichever definitions you choose to follow is up to you. The differences between the definitions are negligible as far as I'm concerned, though more modern definitions are built upon past experiences so I do encourage you to stay up to date with the contemporary literature. Also, be skeptical about all the definitions that you find, people like to give stuff names and fancy acronyms, then vaguely define them or use them not as intended (e.g. RESTful APIs), so try to have your own opinion, even if you might be wrong - this is the only way to make advances. As long as you are equipped with a determined set of definitions for your work, you are set to go.

2. Requirements Analysis

> *Anything that won't sell, I don't want to invent. Its sale is proof of utility, and utility is success.*
>
> *- Thomas A. Edison*

Architecture is made to fulfill a set of requirements. As architects we need to understand and elicit the requirements necessary, then make an architecture that fits the desired requirements. The problem is that for requirements there is a whole field of requirements engineering, and some even include the field of stakeholder theory to elicit and manage requirements correctly.

In other words, there is so much to read just about that, and I do recommend that you will take the time to get familiar with the literature, studies and content on those subject to broader you understanding and optimize the way in which you understand, fulfill and manage the system's requirements during the process of architecting a software. Since, in the end, the software is just there to satisfy the requirements.

For small projects, and even most projects, you don't really need to get too deep into the theory of making requirements, it is enough to understand a summary of requirements engineering, such as the road map of requirements paper. Large-medium enterprises usually have something called a requirements analyst which is the one responsible to be well versed in those topics and create a requirements document. For starters I find it essential to understand the following points about requirements:

1. **Always start with requirements**: If you don't have requirements to satisfy, then everything you will architect will be void of meaning.
2. **Functional requirements**: When eliciting requirements from stakeholders, you will always find them describing to you what the system "must do" - all of this is a functional requirement - how a system should function for the end user. Users view the system as a

blackbox that acts to perform a given task or complete an action - how it will do it internally and its structure is the work of the architect and system designer.

3. **Non-functional requirements**: This type of requirements is not immediately persummed, they basically describe what a system “shall be”. Such requirements are elaborated by the ISO/IEC 25010 standard. Those include security, performance, configurability, etc. You can elicit them by reviewing the functional requirements, and then figuring them out by presenting them to the stakeholders and asking for their opinions and needs.
4. **Break Big Requirements Into Small Projects**: According to The CHAOS Report by the Standish Group, most successful, on budget and on time projects are small scale projects, with a team of 6 or less professionals working on a project. A misconception people often say is “The size is the size, and the size is dictated by the requirements”, not only it’s not true, but believing it will lead you to architect and design big projects, thus already putting them into a category of lower success rates. Instead, break the requirements into smaller chunks that can be addressed as smaller projects with a different system design document for each.
5. **Ordering**: Separate Non functional requirements and functional requirements, give a number for each requirement so it can be referenced. Break requirements into smaller requirements and give them a sub-number. For example, if we have a requirement of “Users can buy items” which is numbered 1 it can have a sub requirement 1.1 “Users can choose to pay later” and 1.2 “Users can add new similar items in checkout”.
6. **Relate architectural decisions to requirements**: After you order the requirements of a system in a Software Requirements Specification document (SRS), you can then proceed to your architecture and low-level design by referencing the requirements by number in each decision and usually explain how it fulfills those requirements.

3. Software is about Data

Data is mere facts. Many of the architects and software engineers don’t look at data as such, rather they encapsulate it under abstractions of the real world as components or objects. Adhering to data from this point of view, as something that is part of an abstraction and not an individual fact, constrains the thought process and possibilities that may arise from data as an independent thing.

One of the things I’ve learned from the Data-Oriented Design book by Richard Fabian is that abstractions are easy for us, since we abstract things all the time, and this is true for real world scenarios, but a dog in the real world is not its representation in a computer. In a computer the data representing a dog for the end user is only a set of facts that make up the output that describes or visualizes a dog in a computer. When we abstract all the facts about dogs as a dog class, and all the information about it within this abstraction we block ourselves into a specific paradigm. We encapsulate data immediately as a component's hidden state or data that is passed before we think about it as an individual, and in doing so we might find ourselves passing more information than needed between components, or even have duplicate information residing in different components.

It’s not that I encourage everyone to use Data-Oriented Design, rather, it is not correct to use it for all case scenarios, it is most effective to use this framework when efficiency is highly necessary. However, the lesson from data-oriented design, that data is individual facts that are not entangled to any abstraction frees us from abstracting things right away, and allows the architect to view the data separately from everything, and only then think about how we can encapsulate or abstract away the needed parts while keeping some data as plain individual datums that can be either accessed by all components or stored as a different element.

The next step in any architecture after the requirements, is understanding the data and its relationships. Data is everywhere, and the software only transforms it and transmits it to the right places. I look at data in the following order:

1. **Viewable Data**: Data that is presented to the end user (via some UI) or that is the result of the system (Output). This data is almost immediately deducible from the functional requirements.
2. **Received Data**: Data that represents an end-user action or an external system. This kind of data is received by the system during runtime and it dictates how the system will act. Also deducible from the functional requirements.
3. **Stored Data**: Data that is stored in a DB or filesystem. Its structure and necessary information are relevant to understand which components and how they shall be structured to handle such data. The stored data is based on the viewable & received data, as it helps and stores the information necessary to allow the viewable data to manifest by the system.
4. **High-level Transformations**: Between stored data and the next type of data we will discuss, I first map the transformations that produce the viewable data or c omplete different required user actions from received data. I might even do this while thinking about the stored data, because those transformations might require different data to be stored.
5. **Runnable Data**: Data that is used in runtime, session or caching systems by the software. This type of data exists “between” viewable data and stored data, and is used by the system to transform, edit and transmit the stored data for outputting the viewable data, thus it is the last data to be thought about.

From Viewable Data and Received Data, we can deduce the MockUp and end-user components and their structure, including interfaces. From Stored Data we can understand the most effective way to architect our databases and their technologies. From Runnable Data we understand the running components and transformations that help us complete the functionalities and qualities required. With time you will also notice that working through the architecture in this order will sometimes make you go back some steps and make changes because you might find that some relationships are irrelevant and easily elicited in runtime, or can be more specifically defined.

4. Architect with different Viewpoints

The introduction for the famous paper introducing the “4+1” view model of software architecture sums it up quite neatly:

> We all have seen many books and articles where one diagram attempts to capture the gist of the architecture of a system. But looking carefully at the set of boxes and arrows shown on these diagrams, it becomes clear that their authors have struggled hard to represent more on one blueprint than it can actually express. Are the boxes representing running programs? Or chunks of source code? Or physical computers? Or merely logical groupings of functionality? Are the arrows representing compilation dependencies? Or control flows? Or data flows? Usually it is a bit of everything.

Just like architecting a building, we need to view an application from various perspectives, each of which is exposing a different aspect of the system, and they work together to manifest the requirements that the system needs to fulfill. A software system and its components can have the following views:

1. **Data**: The plain data, data types and their relations. Usually the first step of the architecture as elaborated in 3. Software is about Data.
2. **Data Placements**: Where is the data placed? Is it in memory? Is it part of a component? Is it cached? Is it in session? Is it in a database? Is it in a file? etc…
3. **Processes**: What is the process that the system runs? What is their flow? How can they be distributed or separated into different threads/processes on the hardware? Which processes depend on which processes? Which components are responsible for which processes? What processes are looping and calling other processes and which are passive? etc…
4. **Interfaces**: What general interfaces are used throughout the components of the system? What does an interface of a network node look like and its standards? What is the general interface of run-time classes or components that represent an interface to data? etc…
5. **Components/Systems**: What are the runtime components of the system? Which component communicates with which component? Which component is a system that breaks into subsystems (other components)? etc…
6. **Physical**: What is the hardware used by the system? How is the hardware connected and with what configurations? Are there different connections and devices in development, test, stage and production environments? This view can help you know how to simulate tests.
7. **Servers/Cloud**: What is the structure of servers? How components will be distributed amongst the servers or cloud providers? Which cloud services should you use for each component/system? Will you use load balancing? Container clusters? etc…
8. **Files/Folders**: What is the structure of the project in files folders? What is the goal/responsibility of each folder? How will the project be broken into different repositories if needed? etc…

9. **Combining Views**: Which data is used by which process? How data and processes are mapped into components? Which components/systems implement which interfaces? How files/folders are mapped as components/systems? How does the cloud and physical architecture hold components and the data?
10. **Scenarios**: Views can be expressed in terms of scenarios of cases and user actions in the system, each scenario is like combining the views and describing how the views manage to fulfill and stick up to a given scenario, and manifest a functional/nonfunctional requirement.

5. Use Ontology Graphs instead of UML

> *Everything should be made as simple as possible, but not simpler.*
>
> *- Albert Einstein*

Many use UML diagrams to describe objects/classes and their relationships, relational databases and other views between components and data in an architecture. UML poses a set of disadvantages:

1. **No rigorous formulation**: It is not grounded on a mathematical model, and is purely conceptual. Thus, it is hard to understand alternative structures and models stemming from a specific one.
2. **Hard to cross-compare**: Very hard, if even impossible, to compare to other models of UML, because it is not rigorous enough.
3. **Not reusable**: Hard to reuse one model of UML for another application, should almost always be made from the beginning.
4. **Not user friendly**: Creating and modifying a UML diagram is difficult when you have 3 types of relations between elements, and data/functions mapped on the same diagram in each component.
5. **Same viewport, too much information**: People use UML for mapping too much information in a single diagram that can be decoupled to more readable and easily configurable models of representation.

Strictly speaking, it is not simple, it contains too much information on a single diagram, and it involves no rigorous definition based on a mathematical field. Those disadvantages have been present in the study of knowledge representation for years, until David I. Spivak proposed in his paper a new framework for knowledge representation called Ologs, or Ontology logs. Ontology is the study of what something “is”, and Ologs provide a nice representation for it, while keeping things very simple, and is also rigorously defined using the mathematical field of category theory.

D. I. Spivak presents the advantages of Ologs in his paper as:

- an olog gives a precise formulation of a conceptual world-view,
- an olog can be formulaically converted into a database schema,
- an olog can be extended as new information is obtained,

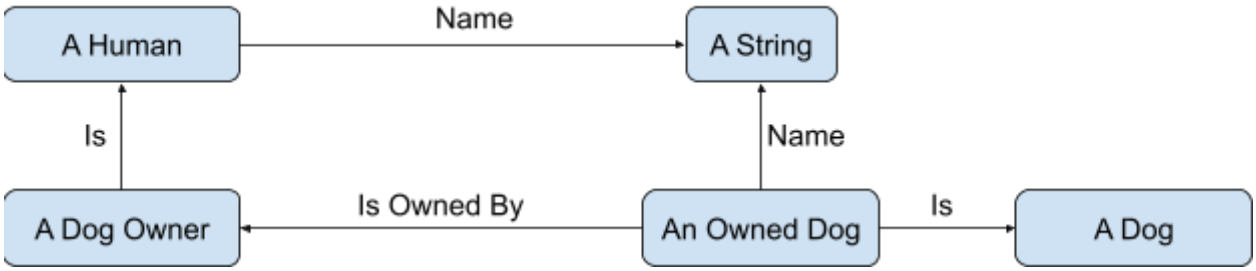
- an olog written by one author can be easily and precisely referenced by others,
- an olog can be input into a computer and “meaningfully stored”, and different ologs can be compared by functors, which in turn generate automatic terminology translation systems.

The main disadvantage that David mentions in his paper is that a good Olog demands a clarity of thought that regular written text or conversion can more easily elide. However, it is easily solved by combining the two together. I use Ologs to convey the elements and their relationships of an architectural view or database schema, and then I describe their constraints and more complex configurations and properties using written notes along with the olog, this way I get the best of both worlds.

I recommend you to read D. I. Spivak’s paper to fully understand what are Ologs and their inference rules, but I will still provide a simple introduction with a real world example.

An Olog consists of only 2 types of objects: Categories and Functors. A **category** is a collection of objects (i.e. a class or database schema), and a **functor** is a unary function from one category to another that “transforms” or “translates” an object in a category to an object in the other category.

An Olog is drawn as a directed graph, where the nodes are interpreted as categories and the edges as functors. Each category and functor have their own name or symbol. For example, we have the category of all Dog owners, which are part of the category of all people, and we have the category of all dogs that have an owner, each of them has a name which is a functor from itself to a category of all strings, visually we will represent it as such:



The attentive reader will immediately notice how easy it is to formulate this diagram into a database schema. It is obvious that the categories will become database tables, and the functors will become fields, some of which will be foreign keys (Owner, Is) and others will have some primitive data type (Name).

You don’t need to make things such as “1 to 1”, “1 to n” or “n to n” constraints on this diagram, everything is basically “1 to n” in the opposite direction of the functor (for example, the same name can be used by multiple dogs), and all the other constraints in this view are irrelevant until you mention them separately (or can even manifest from the diagram), this way we separate the

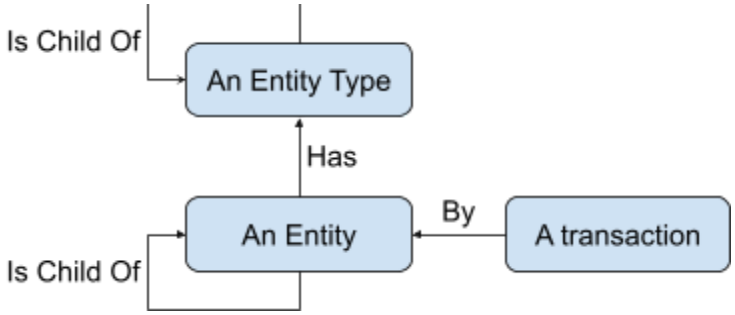
constraints on the relationships and the relationships between elements in the architecture, thus creating easier to read and modify views of the architecture.

Another note on simplicity, is how easy it is to read this diagram. For example, if we read the functors “Is owned by” in its corresponding direction we get: “an owned dog is owned by a dog owner”. In the case of the “is” functors we get: “A dog owner is a human” and “An owned dog is a dog”.

In most cases I will not mention the name functor, and will leave all primitive data based functors in a written list to separate the relationships and primitives, thus creating a cleaner separation, and a better view and way to work on the relationships themselves without involving both data and functions in the same diagram.

At one time, I was trying to understand the requirements of a client we have, they needed a system to map transactions that pass through what they call a “channel” which is owned by a “merchant”, up until this point everything was pretty clear. Then, they started to argue with one another which “entities” should be above in the hierarchy of a “merchant”, they wanted an entity that will contain a set of merchants and another entity that will contain a set of the previous entity, and probably so on, they were not sure how they should call each entity and how long the hierarchy should be, so I decided to solve their problems with simple inference rules from Ologs, I’ve mapped the following diagram:

By a simple look at the diagram, and how the stakeholders were arguing how to name each entity and which order to give it I’ve decided on the following view:



The “An entity is child of an entity” stems from the previous diagram, and the “entity type” stems from their discussion on namings and hierarchy, with this architecture they can choose which entity types they want to use, their names and hierarchy, and then the specific entities with their own unique names are mapped into the corresponding entity type - thus when they will want to agree on new entity types or new namings, they will be able to do so dynamically without involving further development (satisfying configurability, so that we wouldn’t need to put up with their changes and modifications later).

The only constraints applied to the diagram is for any object of an entity, its parent should have an entity type which is the parent of the entity type of its child. This constraint is then accomplished programmatically.

This type of deduction is almost immediate using the rules of inference from D. I. Spivak's paper, while with UML it is harder to deduce, messier to work around with, and less intuitive.

Conclusion

1. **Understand the definitions**: Know the contemporary literature about architecture and design, have your own opinion on the definitions and discussions around the subject.
2. **Requirements Analysis**: First elicit the needed requirements. Break big requirements into smaller ones and break them down into smaller projects, each with his own SRS so that it will be easier to implement.
3. **Software is data**: After requirements, map all the data necessary to make the system work. Map the data that makes up the output on the system (viewable data), the data that is stored long term, and by understanding all the actions and transformations that the system needs to make on this data, map also the runtime data that the system needs to hold in memory. Only then, after you have all your data mapped, find how you can encapsulate it in elements and components.
4. **Use different viewpoints**: Don't make one diagram to rule them all, a single diagram cannot fully grasp the architecture and different aspects of a system. Instead, break the architecture into multiple diagrams, each of which has a different view on the architecture - one that maps the data, another that maps the relationship between components, another that maps some processes of the system and their distributed properties, etc.
5. **Use Ologs instead of UML**: Make your architecture ideas simpler, easier to read, reusable and rigorous using Ologs, UML adds too much information on a single view, and has no grounded rigorous inference rules like ologs.