

SWARM: A Concluding Paper

By: Sander Triebert
TU Delft, July 2005

Software Architecture Vs. Building Architecture

Software architecture is a term, which doesn't have one single definition. Many people have different ideas over what exactly what the right way is to design a software system. Also explaining to other people what exactly a software architecture is or does, is therefore a bit hard. But often software architecture is compared to building architecture. This is done to give people a better, more reachable, idea of what a software architecture stands for. Let us look closer at this comparison.

The relation between the building architecture and the software architecture is based on the architecture part. This seems strange to mention, but it's the key in the whole story. It's about comparing the architectures and seeing the similarities, weaknesses and where the comparison ends.

The similarities, which both architectures have, form a basis for the comparison. One can think of the actors involved who want an end product and have different demands for it. In the building sector for example actors want to have a twenty-story building next to a river. The building architect has to design a building which is conform the demands of these actors, just like a software architect has to design a program to be conform the demands of his/hers customers. There are also external factors, which have to be taken in account when building a building. For example the materials used when building. The same holds for a software architecture. Here the outcome is depended on the current software engineering techniques, which are currently available.

A building architect starts at the bottom of the building and drawing it in stages, with different components, such as the wiring, drawn up in more specific plans. In the end there are multiple schemes or views of a plan for sub-contractors to build their piece of the total building. A software architect does the same kind of thing. He – should – divide the system he's building, in different kinds of structures and views, which handle different kinds of aspects, all related to their need.

There is also a weak point to be addressed. When designing a building, one should design it to last, a strong foundation, solid walls, good wiring etc. Not much things change to a buildings architectural plan when maintaining a building. Floors are kept clean on the inside, walls are painted on the outside, and furniture can be move freely inside the building to keep up with the needs of the people using it. A software architect doesn't only design software to last, but also to be maintained in a way that it can also be extended in the future. Buildings aren't designed to have these needs. One doesn't say, let's build another twenty stories on top of the building because the building ages or doesn't suffice anymore for what it was designed to do. Software architectures have to tackle this problem of aging of software. They should design their software in such a way, that it can be easily maintained, not only on the outside, but also on the inside. Adding a feature to a software package is a common thing nowadays, so the aging aspect of the comparison is a weak point when comparing software architectures and building architectures.

The comparison breaks down when one needs to use components, which are a default. In the building sector, there are many known and used standards, which a building architecture can rely on. Types of walls, isolation, glassing, frames, wiring, etc. are all kinds of standards, a building architect can use and keep in mind when designing a building. He knows that when he delivers his architectural plans to the developing party, they can – almost – make a one to one relation from plan to end product with the use of the available standards and products. Software architects don't have this luxury. There aren't many standards available for a software architect who wants to design software about which he knows, that it can be used as a firm foundation to build upon. Because every building block about which is known that it's a reliable implementation or standard when used, can improve the outcome. A building architect for example knows when he wants to design – add – wiring in a building, he just has to draw it in and because of the specs giving, the appropriate wire will be used, so the wiring component of the building is taken care of. A software architect unfortunately doesn't have these things to his disposal. Not having this knowledge will greatly change the rules of the game for a software architect, because more things have to be taken into account, then only the main architecture. This is also where the comparison ends between the building architect and software architect.

Brooks Conceptual Integrity

Conceptual integrity, “the underlying theme or vision that unifies the design of the system at all levels. The architecture should do similar things in similar ways”[1 4.7], that is how Brooks [2] view on systems is translated to the Software architecture. Meaning that what “Brook’s idea of conceptual integrity does for the user, architectural integrity does for the other stakeholders”[1 4.7]. But if it’s really holds up in systems of today is still the question.

Conceptual integrity does play a big part when designing and maintaining a system. It does sound as the “Holy Grail” solution, just make sure you have conceptual integrity in your system and your done. But this isn’t the case at all. It isn’t the key to making a system successful. Conceptual integrity is a nice guideline to follow, but it’s nothing more than that. This is because of the fact that it doesn’t guarantee anything about maintainability as a whole. Sure, it would be nice if everything is written in the same style and uses the same structures within source-code and has the same patterns apply throughout the design, but having this in place doesn’t ensure that everything used, has to be the right choice for a system. One could simply imagine that being forced to work a certain way, because of a choice that was made and has to be maintained because of “*conceptual integrity*” also isn’t the right way to work. Systems or sub-systems, that would have been better implemented due to different structures and design patterns used, are now forced to being implemented in an unnatural – wrong – way.

A good example of a system that does not have this property, but still was successful is Linux. Linux kernel had a good solid conceptual model [3], but the external libraries and drivers were left up to developer to implement how they seemed fit, just by the use of a common interface to communicate with the kernel. So not the whole system was designed with the same conceptual views in mind. But the Linux system still was successful, and is still gaining market share. This is thanks to the way Linux works. The most important part of the system, the kernel, was designed very carefully by a select group of designers and engineers. Thanks to this solid foundation, which was formed with the help of the common interfaces, third parties were able to develop (sub-) systems the way they seemed fit for their system. This of course lifts the weight of the shoulders of software architects, which else had to comply to a “*conceptual integrity*” which was setup for everything in Linux. They only needed to worry about how they implemented the common interface. So how their (sub-) system looked like from different levels, didn’t matter, as long as they addressed the common interface in the right way. This made Linux very popular amongst external designers and still made Linux to a success, and also maintainable, because the most important part of the system, the kernel, also has to comply with the common interface they setup and internally also has to comply with the conceptual model they designed for it. So the use of different conceptual models can be implemented correctly in a bigger system, so the need for conceptual integrity isn’t one that is a key to making systems successful, Linux has proven to be a good example of that.

This leaves the question, how can we measure to what extend conceptual integrity is applied throughout a system. Starting with the documentation is always the first step when analyzing systems or trying to find underlying architectures. When having read the documentation extensively, one would be able to see if everything is designed in such a way that it is conform the rules of conceptual integrity. So that everything is done in the same way, on whatever level we are looking at the system. That everything within that view/level is designed in the same way and obeys the same rules. But this is of course when having the luxury of good documentation, which of course is something that doesn’t happen all to often. So we have to resort to recovering the architecture used.

The easiest way to know if there is conceptual integrity when documentation is lacking, is to construct a certain model of the system of what is thought to be the model and make this comply with conceptual integrity. After you’ve done this, try to map the system on the model you made of this system, would it have been a model designed with conceptual integrity in mind. If you can make it fit, there could be a good chance that it has been designed with some conceptual integrity in mind, when it doesn’t fit or can’t make it fit even slightly, there is a big chance that it wasn’t. There is also they grey area in between, were it isn’t clear or not if the system is leaning over more to the integrity side or the non-integrity side, these cases are always very difficult. The main problem is, that some thing cannot always be conforming to a unified style and then the question is; how much does that odd factor weigh in respect to the total system. So measuring it will always be something what will be difficult, because there probably won’t be a definitive answer which can be given, because there will always be some divergence in one way or the other.

Attribute Driven Design for Reconstruction

Let us look at what an attribute driven approach exactly is. To quote [1 7.5], “*Attribute Driven Design (ADD) is a top-down design process based on using quality requirements to define an appropriate architectural pattern and on using functional requirements to instantiate the module types given by that pattern.*” This seems a very good method for defining an appropriate architectural pattern, because the most important factors of a system are handled as core

issues, namely the quality attributes. Especially when designing the system and choosing the appropriate pattern. It would be nice if this type of method could also be used to reconstruct an architecture.

If we look at how the method works, we see it's a quite simple method, which leads to good results. First one chooses the module to decompose; subsequently one is going to refine this module and finally these steps are repeated for every module that needs further decomposition. One can see that this is as a tree structure, with the root node as the whole system, which is decomposed in a number of child nodes until the leaf nodes don't need anymore decomposition. This can be very easily translated to also reconstruct architectures. The benefits of the approach are clear. It would lead to a quite fast method of reconstructing an architecture, because one starts looking at the most important part of the program and subsequently goes deeper into the architecture following the most important modules in the system down the tree. It's a structured and clean method to follow when reconstructing an architecture.

The good thing about the method is that when applied, one can almost follow the guidelines for constructing an architecture. The whole system is what one has to start with and this is also the case when reconstructing. Then instead of trying to come up with a matching pattern for the quality and functional requirements of the module, we have to look at these same quality and functional requirements but from the point of view of how they are organized. Looking at the source code and documentation many things can be seen already. One could find for example a tree structure with directories in the source code. After having found such an initial structure present, the next step is to look at the same functional and quality requirements within these modules and try to match them to an architectural pattern. The matching of modules to architectural patterns would be the hardest part in the whole process of applying the ADD method. If in whatever design pattern the original designers also had the quality and functional at the highest priority, and chose their pattern to match – fit – these requirements, one can more easily come to the same conclusion of the pattern used. This is also the weak point of the ADD method when reconstructing an architecture. When the original designers didn't have the most important functional and quality requirements at heart, finding a pattern based on this assumption would be difficult, which is crucial for the ADD method, because it follows the most important parts of the system down the composition tree. To continue, when a pattern has been identified within a module, one continues to decompose the module if necessary into sub-modules. So when reconstructing an architecture, one further looks at the information they have about the source code and/or documentation and decide whether the current selected module was decomposed into smaller parts. These steps have to be repeated for every step in the reconstruction process.

The biggest problem that could exist when trying to reconstruct an architecture, is that there has to be information about the constraints, functional and quality requirements. Since ADD in the construction phase of an architecture has this as a requirement, this also counts for the reconstruction phase. Especially finding the constraints and quality requirements is something, which is hard to do when no proper documentation is present. On the upside, if this information is present, the method mentioned above for reconstruction is a good guideline for reconstructing an architecture, which finds all important modules and tries to map these to the appropriate patterns.

BlackBoard's Quality Attributes

Nowadays we see a growing need for e-learning environments. Many students and teachers rely on these systems to support the courses they follow and teach. Thus one of the most important things is that such a system has a high availability, because the information on it needs to be shared and available, because many people depend on it. So during design, aside from general availability scenarios, also concrete availability scenarios are needed, which are "*those that are specific to the particular system under consideration*" [1 4.3]. Let us look closer at a well known e-learning system, namely BlackBoard.

The BlackBoard is a perfect example of an e-learning system, which requires a high availability, because of the many users connecting to it. To give an idea of what the concrete availability scenarios are for a system as the BlackBoard see *Table 1*. We are using the six parts, which a quality attribute scenario consists of; see [1 4.3].

These kinds of very crucial systems, like the BlackBoard, have to be designed with high availability in mind. There are numerous availability tactics [1 5.2] that can be applied when designing such a system. We'll discuss a few of these availability tactics, which could prove useful to such a system.

	Source of Stimulus	Stimulus	Environment	Artifact	Response	Response Measure
Scenario 1	User (teacher, student, ...)	Login request	Normal Operation	Process	Logging in user	No downtime
Scenario 2	User(teacher, student, ...)	Page request	Overloaded system	Process	Alert administrator of system overload, inform user of unavailable system	Half a day
Scenario 3	User(teacher, student, ...)	Not existing page request	Normal Operation	Process	Inform user of not existing page, continue to operate	No downtime

Tabel 1

A heartbeat type of fault detection can be a good way of detecting if an error has occurred. Since systems have many different modules, some less crucial than others, to keep the communication bandwidth low, such a tactic would have modules send a periodically signal to another component, which listens to these heartbeats. In this way easy monitoring of the system and fault detection can be implemented. Also implementing a design, which throws up exceptions is a useful design and can even be used in conjunction with the heartbeat system. This is because the exception type of detection, can also inform the user of faults that occur within the system, but don't have any impact on availability at once. Not all faults lead to failures. So these types of tactics would ensure a system, which is capable of detection faults. But handling faults also has to be dealt with.

After having noticed faults in a system like the BlackBoard, a good recovery tactic would be to implement an active redundancy design. By having redundant systems, which perform the core activities of such a system in parallel, one can easily switch over to a redundant system. One can think of a user-database, file-repository and such, with information, which isn't retrieved from other sources that easily. Thanks to the fact that these redundant systems run in parallel with the active one, a switch over to a redundant system can be made in seconds, which saves on the downtime and both systems are up-to-date. A second tactic, which could prove valuable, is one that also includes rollback features. Making frequent checkpoints of the system in a consistent state will lower the time needed to restore a system into a consistent state when it has entered a state of inconsistency. Also for a system as the BlackBoard this could come in handy, because of the many users and applications, which make use of it, the system may fail in unusual manners and revert into a inconsistent state.

When designing such tactics in favor of high availability, trade-offs will have to be made, especially the modifiability will have to suffer. When wanting to ensure high-availability, systems will have to be designed in such a way that there is a kind of monitoring in the system, which is capable of monitoring all activities and listening for faults. If one wants to modify something in the system, it has to be taken in to account that every modification made, will have to be checked and double check, whether to see if it could throw the system in any inconsistent state or have it fail. If so, also the appropriate measures have to be taken into account and whether these are already in place or also have to be added or modified is the question. So the total response measure of the quality modifiability attribute will rise.

It also has an impact on the performance attribute. Adding extra functionality to the systems, which monitors the whole system and has to respond to it has an impact on the performance. So for example trying to comply with certain performance goals, can be harder to comply with due to the fact of the raising number of requests and communication internally which have to be dealt with, due to the fault detection and recovery. This raises the response measure of the performance attribute.

In terms of security, the high availability attribute and the security attribute go hand in hand in systems like the BlackBoard. Security is one of the most important things together with availability, since the availability depends in certain aspects on the security attribute. Un-allowed data injection into systems, the changing of data and such are things the security attribute has to deal with, while trying to still deliver its services to the legitimate users. Delivering these services and high availability are sort of on the same line. They both want to ensure the availability of data and services to users, so here the trade-off would be minimal, since they work towards the same goal.

To conclude our evaluation of the BlackBoard system, we will look at the usability tactics, which were or seem to have been applied. As discussed in [1 5.7], we can translate this to some functionality that the BlackBoard has implemented. It supports both "*user initiative*" and "*system initiative*". The user can cancel, undo and intervene in all processes on the BlackBoard. All these things are incorporated into the system. Also "*system initiative*" is implemented into the system. We will discuss them one by one.

- “*Maintain a model of the task*” – For this kind of behavior, JavaScript is needed, cause the BlackBoard is a web-application. JavaScript has not been integrated in BlackBoard in such a way that it real-time monitors the task at hand and interacts directly into the user space. Internally it seems to only handle request per page and doesn’t have a model of the task at hand.
- “*Maintain a model of the user*” – This is done by BlackBoard. With the help of personal preferences the user can set a profile with settings and features that it wants to see enabled/disabled in the BlackBoard system.
- “*Maintain a model of the system*” – The BlackBoard system does have a model of what situation it’s in and can give for example prognoses about the duration of the current activity. For example the progress of a survey, the number of users online.

So as we can see, the BlackBoard does use usability tactics, but don’t have all of them incorporated into the system. This is also mainly, because it’s a static web-interface thru which the users access the system, which limits the options of having various indicators or tools, which help end assist the user at their end. Another reason for not incorporating all these models is because they use up resources. Generating the pages statically and not having continuous communication between the client and server side of the BlackBoard, the system can use it resources to also serve other users. This also is important when thinking of a high-availability. The less resources are used per user, the more users can access the system simultaneously and the less chance of the system collapsing when dealing with large amounts of users at the same time.

The BlackBoard Case

Let us continue with the BlackBoard case. What if instead of reconstructing or designing the architecture of BlackBoard, your goal is to get acquainted with BlackBoard. Let us discuss the steps that should be taken to familiarize yourself with an existing system, in this case, BlackBoard.

Lets start with the documentation. When starting to delve into a new system documentation of the existing system is the primary source of information about the system. The first and foremost thing one should do is to know what the system exactly does or stands for. Things like a product description or the standard manual will suffice. This will give you a very global idea of the capabilities and also the asked requirements of the system. After you’ve got a picture in your head about what the system exactly does and is, one can delve into the more in-depth information. One should start looking at the structures and views of the system. Depending on your assignment within the BlackBoard design group, obtain the appropriate documentation. If for example you’re a head architect, ask for the *work assignment* structures at first. You’ll then have a clear picture of that the system has been divide into, in respect to the given architecture. But if you’re a member of a team developing a certain module/part of the system, obtain the documents specific for this module. Keeping the “4+1” [4] approach in mind, try whatever function you have within the company, to get a clear picture of the (sub-)system. This can be done by starting at the top and zooming in more on the details. Following the “4+1” approach concerning the views, find out what the *development view* is, so ask for matching documentation about how the system is organized. Followed by the documents of a more *logical* nature, which feature the functional requirements and layout. When analyzing these documents, the *process documents* come into play, which adds non-functional requirements to the (sub-)systems. These documents might explain choice made for certain structures used in other views due to non-functional requirements. Also let these be accompanied by scenarios of usage and internal workings of the system. These scenarios are based upon the initial parameters, constraints, functional and quality requirements setup for this system. After having zoomed in from top level, the whole system, to a much lower level, one ends up looking at the documentation about the source-code and the source-code itself. But one has to keep in mind, that not all systems have named or categorized their documentation in a way that you can simply see the difference between certain views or have documentation of the system at all. The key is to still try and start at the most highest level of documentation available and narrow in – on perhaps your assigned (sub-)system – to find the information needed to keep a good knowledge of were every piece of the system fits in.

When evaluating the architecture, it’s important to also have stakeholders present. One could think of the original architects and requirement engineers (who also represent the customers [1 9.1]); they can inform you about choices made for certain patterns and such. Also important are the implementers, to give you insight about how exactly the patterns are translated to source-code. Maintainers of the system also play a big part in the evaluation, because they give feedback about the state of the system and the maintainability. And of course the end users and testers of the system are a vital stakeholder, because these people actually work with the implemented system and can test and experience if the software lives up to its promises.

Looking at the security side of the system, one needs the documentation about the *physical environment* of the system and the *developing requirements* to see where the security is need and linked. When evaluating the code one should stay on alert for a few things. All modules and pieces of code that take input in text form, from the user should be under careful consideration for code injection leaks. Also connections made via sockets or via modules from remote clients should be also closely watched. So things like login modules have to be inspected thoroughly for security leaks. Another thing that may trigger your suspicion concerning security are al read and write actions that can take place in a system, these actions always play vital roles in systems and can grant, if wrongly implemented, users access to unauthorized information.

After having evaluated the documentation, one could come to the conclusion that the system was layered. To find evidence of this one could look for extensive usage of the *uses* relation in the source-code[1 2.5]. Depending on the programming language, the *uses* relation can be defined in different ways. Often the source-code is also divided into services of functionality. Within the source-code no direct relations are found between the functionality of the layers that differ more then one layer. So a layer n can only use the services of a layer n-1 [1 2.5]. Since the layers where presumably found – because of the conclusion about the layered system – this can be checked quickly in the source-code. Whenever there are relations that go across multiple layers and for example extend or implement various subsystems, which aren't associated with the current layer or layer above, one could simply refute the hypothesis.

Another conclusion that could be drawn would be that the system has a shared repository style. In such a system, many connections throughout the whole source-code are made to one or a few shared repositories. These can be identified in the source-code by multiple links and requests to one or a few certain components in the system. The dataflow can be clearly identified to and from those repositories. These components and connectors are created to store, create and access data. Whenever such a structure is identified or hypothesized, the source code connector elements have to resemble those in the hypothesis. So the elements which serve as connectors and components have to be presents in the source-code. This could be as specific functions, which represent these elements or as simple as the source file structure which doesn't match the hypothesized repository style.

The Architecture of Web Applications

When looking specifically at Web applications nowadays, we see an often returning architecture, namely the one with producers and consumers of data, better known as the client-server structure. We see this is often used in web applications because of the advantages this kind of structures has, namely [1 2.5]:

- *Separation of concerns*
- *Physical distribution*
- *Load balancing*

From an architectural point of view, the separation of concerns is welcome feature. This is because it is actually supporting the modifiability tactic. The separation of the concerns of the client and the server, let the architectural designers implement, test and deploy changes more easily, so it takes less time and costs. So this is related to the modifiability tactics. To list the specific tactics that could be used from the modifiability tactics we find[1 5.3]:

- *Maintain semantic coherence*
- *Generalize the module*
- *Hide Information*
- *Maintain Existing Interface*
- *Restrict Communication Paths*

Also the load balancing can be related to the performance tactics. One can more easily design conform to performance constraints, when they have the ability to balance the load on the system. Specific performance tactics used for achieving a client-server relation are[1 5.4]:

- *Increase computational efficiency*
- *Reduce computational overhead*
- *Introduce concurrency*
- *Maintain multiple copies of either data or computations*
- *Increase available resources*

To conclude we also have the physical distribution, which is possible with a client-server type of architecture. The location of the client(s) and/or the servers doesn't have any relation between them. This of course is a pro when taking into account the physical distribution of the system in terms of end-product placement.

When one already has a distributed web application we talked about earlier, which was possible with a client-server structure, how one does derive a model view of such an application when some parts of the application are being executed miles from each other remains the question.

The first thing to look for is the source-code and documentation of such a system. Whether the distribution of such a system isn't localized, doesn't necessarily mean that the source-code and documentation isn't available locally. Even if the system is written in various types of languages, one thing known for sure is that a specific – part of the – module, whatever the function may be, it is written in one language. So if one looks at the different types of sources and executables, which come from these source-code files and documentation, one can easily determine the functionality and thus assign them to modules and construct a module view of the system based upon the gathered source-code and documentation.

When no source-code is available, it will get difficult. The main problem then is if the locality of the various computers is a bigger problem than not having the source-code. Having no access to the various computers is only a problem if there is no source-code. One way of trying to find out what the functionalities are of the distributed systems is cutting them of one by one and see what errors come up. And judging by the errors try to fill in the functionality of a computer in the distributed network. Ending up with also a mapping of functionality of the distributed computers, which will together form the module view. But it would seem strange if distance of a computer nowadays would form an obstacle when determining the processes and programs it has running. Determining the functionality of a program can be easily derived from the process-name, executable-name or even the available source-code on the remote computer (as mentioned earlier).

Determining what the component and connector view or the allocation view is, should be a bit easier. The connectors and components of the local computer can be easily determined by the connections it makes internally. For the connections and components that reside outside the local system it's more difficult to map them in a view. One could only follow up on the made connectors and assume that the remote computer is the component, which the connector expects and based on that assumption, build your view. This of course with the assumption that the remote computer is a black-box and not accessible/reachable. When this is not the case, this would be like reconstruction any other system.

The allocation view is one, which can be derived the easiest by simply looking at the distributed system, and mapping the various computers in the external environment. Showing the relations between itself and the local computer, to be named accordingly.

This concludes the recovery of the different views of a distributed web application. What clearly comes out of the analyses is that nowadays location only plays a role when systems are black-boxed, no documentation/source-code is available and no remote connections can be made to the hosting computer. In all other cases deriving and view would be possible, as described.

References:

- [1 X.Y] **Software Architecture in Practice – Second Edition**
Len Bass, Paul Clements and Rick Kazman
Addison-Wesley Longman, Reading, MA, 2003.
Chapter X, Subsection Y.

- [2] **No Silver Bullet: Essence and Accidents of Software Engineering**
Frederick P. Brooks, Jr.
Computer vol 20, no.4: p.10-19. April 1987

- [3] **Conceptual Architecture of the Linux Kernel**
Ivan Bowman
January 1998
<http://www.grad.math.uwaterloo.ca/~itbowman/CS746G/a1/>

- [4] **Architectural Blueprints—The “4+1” View Model of Software Architecture**
Philippe Kruchten
Paper published in IEEE Software 12 (6)
November 1995, pp. 42-50