Software Architecture Assessment

François Vonk

Currently, software architectures are regarded as very domain or system specific. As a result, the majority of effort that is spent on developing an architecture focuses on system details. However, this leaves very little room or no room at all to deal with more fundamental quality criteria for architectures like layering, coupling, and cohesion. Also, documenting an architecture is often neglected, leading to many problems during the remainder of the project. This article describes why these quality criteria are of great importance to the architecture and the software development process.

Introduction

Architecture in general is an already ancient notion that most people associate with the art of constructing buildings. An important aspect of architecture in this respect is shape, because it is the first thing that people notice when they look at a building. However, architecture also plays an important role in many other disciplines and although the definition will not always be the same, shape will always play an important role.

Within the discipline of software engineering, shape indeed plays an important role. A crucial part of a software architecture is the decomposition of a large and complex system into a suitable number of smaller subsystems. This decomposition has to be described and motivated clearly and completely. The description of the decomposition depicts the shape of the software architecture via entities like subsystems, abstractions, and relations.

Why is an architecture so important? Systems that have to be realised in software nowadays can no longer be developed by a small number of people. Mostly, large software projects require an effort of about fifty to sixty man year. This means that a large team of people is required to finish the job within an acceptable amount of time. Such a team can only function when each person can work on a limited and well defined part of the entire project. This can only be realised by having an architecture that decomposes the complete system into subsystems that can be built by a small number of team members.

Another advantage of decomposition is a reduction of the complexity for the resulting subsystems. Each subsystem has to realise a smaller part of the entire problem and the decomposition itself will tackle a number of difficulties. This way each team member will be assigned a portion of work that can be realised and managed easier and more reliably. Also, the oversight and controllability of the overall complexity will improve as a result of the decomposition.

Once a system is in use, it will not stay unchanged. At forehand, users do not always know what they want or can expect, but also influences from outside the system may result in a need to change it. Again, the architecture plays an important role here, because it gives a global insight in the way of working of the system. By inspecting the architecture closely, the entry points to make changes to the system can be found relatively easy.

Apart from the advantages mentioned before, more reasons exist to have an architecture. One example is the reduction of the learning curve for people that are added to a project during its ramp-up. However, the most basic reason to have an architecture

is money! It is the goal of each company to make reliable software with the least possible effort. Reliable software because this reduces the maintenance cost and little effort because this reduces the development cost. Due to the size and complexity of the systems that have to be realised nowadays, this is only possible when a system is based on a sound and well documented architecture.

Quality of software architectures

Having an architecture is one thing, having a sound architecture another. The better the quality of the architecture, the more likely the system will be a success. However, how does one measure that quality? Or even more important, how does one create a sound initial architecture?

First, the criteria that determine the quality of an architecture have to be found. As discussed, minimal cost and effort while realising a reliable system is the main goal. This means that we have to be able to build and test a system rapidly, find and solve errors in the system quickly, and extend the system easily. This results in the following general quality criteria for an architecture: clarity, testability, maintainability, and extendibility.

Surprisingly, these criteria are not specific to a problem that must be solved by a system, while architectures in general are considered to be very problem specific. It is certainly true that a significant part of an architecture deals with problem specific issues, for example modelling system behaviour. However, the general quality criteria also require attention, which is often forgotten. Mostly, all time is spent on solving the problem specific issues while these only contribute partially to the quality of an architecture.

Measurable quality criteria for architectures

The four previously mentioned general quality criteria of an architecture have a somewhat global nature and they are hard to measure. Therefore, it is important to find other, measurable, quality criteria of an architecture that affect the four general ones. Five of these measurable criteria are: layering, coupling, cohesion, documentation, and views.

Layering

Layering is a suitable way to hide for example the hardware of a system from an end-user via various coherent levels of abstraction. Layering is hierarchical by nature, which means that layers are located above and below each other.

The concept of layering in software architectures is generally accepted and applied, see [POSA] and [SA]. The rules for using services from other layers often vary significantly and are subject to discussion many times. The strictest set of rules looks like this:

- It is not permitted to use services from layers located higher in the hierarchy.
- When services from a layer located lower in the hierarchy are used, this layer must be directly below the layer using the services.

The number of variations on these rules is not only in theory large, but also in practice. This is not necessarily bad, but it is important that relaxing the rules is not done too many times and only for a purpose, for example performance. Furthermore, all exceptions to the rules should be well documented.

The most suitable number of layers depends on the size of the system that has to be realised. The larger the system, the more layers are required in the architecture. However, it is important to limit the maximum number of layers to seven or eight, because more layers introduce too much implementation and performance overhead. An example is the OSI model, that is sometimes described as unmanageable with its seven layers. To obtain the optimal number of layers, superfluous layers have to be removed and missing layers have to be added. Superfluous layers are characterised by a large number of services that do no more than pass on information. Such services and the layers containing them will often be skipped in design and implementation, thus violating the rules for layering. Missing layers have to be added when the purpose of a service and the layer containing it differ. Also, when some layers contain significantly more services than the rest of the layers, they have to be split.

Coupling

Decomposing a system is an important role of an architecture and layering can be a first step in the decomposition process. The next logical step is to decompose the layers found and so on. The final result has to be a number of services, also called modules, that have to be implemented. The goal is to find modules, that can be realised by a small number of team members.

A module will not be an autonomous part in an architecture, but will have relations to other modules. The number of relations and the type of relation between two modules determine how much the modules depend on each other. This dependency is called coupling and plays an important role when trying to determine the quality of a software architecture. The weaker the coupling between two modules, the more beneficial this is to the architecture, because the impact of changes to weakly coupled modules is smaller than to strongly coupled ones. Furthermore, modules with few dependencies are easier to test and re-use. The total amount of coupling is determined by its type and size. Weak coupling is achieved when the type is weak, see below, and the size is small. The dependency between two modules increases when the amount of coupling increases.

Three types of coupling can be distinguished, see [SD]:

- data coupling: a module does not influence the behaviour of a module it is related to, but merely passes on data.
- control coupling: a module influences the behaviour of a dependent module by sending a signal; no data is communicated.
- hybrid coupling: the integration of data and control coupling, i.e., one module uses data to control the behaviour of another module.

The weakest type is data coupling, because a module sends information to another one without being concerned how this data is manipulated. The strongest type is hybrid coupling, because it is misleading. A module sends data to another one and is concerned how the data is manipulated. However, the other module is not aware of this, because it receives data and not a control signal. Control coupling is stronger than data coupling because the influencing nature, see Figure 1 and Figure 2. This does not mean that control coupling is inherently bad, but it has to be applied only when necessary. For example triggers to a device driver can best be modelled via control coupling. However, when control coupling is chosen, the reason for using it has to be valid and well documented.

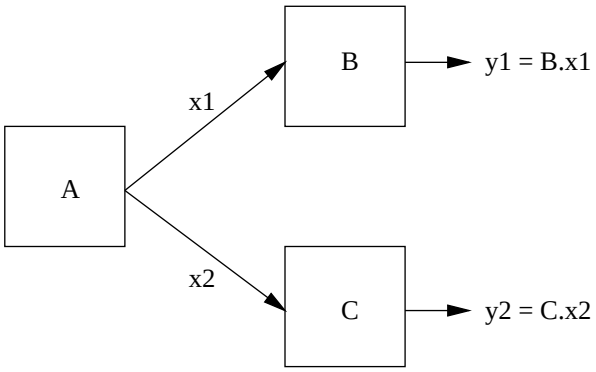

Figure 1: Data coupling

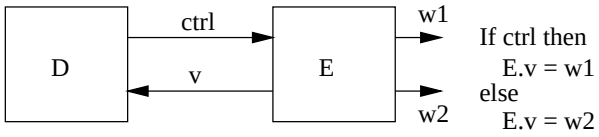

Figure 2: Control coupling

When the result of B is y1 while y2 is expected, the cause of the problem will probably be found in A. When E results in w1 instead of w2, the cause of the problem can be in either D or E. D can send the wrong control signal, but it is also possible that E contains an error in the if-statement. An even worse scenario is that both D and E contain the error described. This will result in a system that functions correctly, but contains two errors. Only when D and E are used separately, the errors will surface.

The size of the coupling between two modules is determined by the number of relations between them and the sizes of these relations. The size of a relation is determined by the amount of information that is exchanged via that relation. Keeping the size of the coupling small is beneficial to the quality of an architecture.

Since coupling is often depicted, the number of relations can be determined but not the size of the relations. Therefore, the interface description for each relation also has to be part of the architecture

in order to determine the size of the coupling. This description contains a detailed explanation of all information that is communicated via a relation. Via the number of relations between two modules and the sizes of these relations, the total size of the coupling can be determined.

Decreasing the amount of coupling can be realised in the following ways:

- try avoiding hybrid coupling,
- use control coupling only when needed,
- do not send information from module A to module B via module C when this information can also be sent directly from A to B and a dependency already exists between them,
- do not exchange information that can also be calculated from data that is already received (unless this is necessary for example to increase the performance),
- merge small modules containing functionality that is strongly related, see next section.

Cohesion

Cohesion is a notion that applies to individual modules within an architecture. Each module offers services that are in some way related to each other. The type of this relation determines how coherent the services are within a module. Services cohere strongly when the relation between them has a functional nature, this is called functional cohesion. Functionally coherent services will often have a lot of interaction, so distributing them over multiple modules will increase the coupling within an architecture. Apart from functional cohesion, other types exist, see [SD]. A type that is often used is logical cohesion, because it depicts the way people think. An example of logical cohesion is doing all initialisation in one module. Using a type of cohesion different from functional is not necessarily bad, but the reasons for its introduction have to be valid and well documented.

The following example shows why functional cohesion is mostly preferable over for example logical cohesion:

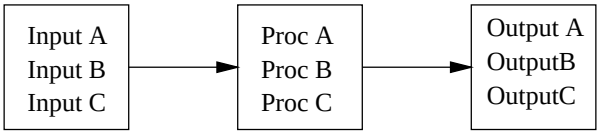

Figure 3: Logical cohesion

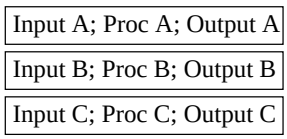

Figure 4: Functional cohesion

Suppose function A represents addition, function B multiplication, and function C division. When an error is found in the outcome of the division, the solution of Figure 3 has three eligible modules that can contain the error. Possibly, three modules have to be inspected, changed, and reviewed as a result of the error. In the solution of Figure 4, the error is located in the lower module, which is the only one that has to be inspected, changed, and reviewed. Furthermore, the modules in Figure 3 have coupling, which is not the case for the ones in Figuer 4. Therefore, the solution in Figure 4 is easier to test and its modules are more suitable for re-use.

Documentation

Not only creating an architecture is important, also describing its shape is crucial and often gets too little attention. The description of an architecture should be part of that architecture. The initial architecture is commonly developed by a group of system architects after which it has to be deployed to the rest of the project team. This is only feasible when the architecture is well documented, otherwise the team members cannot realise the software in the way the architecture intended.

The following issues are important when describing an architecture:

- use clear and unambiguous sentences or use mathematical expressions when possible,
- distinguish various views, see also the next section,
- use suitable and known modelling techniques.

Measuring the quality of documentation is perhaps one of the most difficult exercises. No documenta-

tion is obviously a sign of bad quality, but an abundance is also unwanted. When too much documentation is generated, it will be read badly or not at all. Furthermore, the review time will increase exponentially when the amount of documentation grows. Having good documentation means that all planned documents are written and that they concisely describe all relevant issues using the appropriate and agreed diagramming techniques.

Finally, it is important to realise that the description of an architecture is primarily made for the team members that have to develop the system. They have to be able to understand the architecture by reading its description. Therefore, it is important to pay attention to this when reviewing the documentation, possibly by involving a number of engineers in the reviews of the architecture.

Views

To keep an architectural description clear, it is useful to describe the different issues treated by the architecture separately in so-called views, for example the views as used in the SONI model. This way, every issue can be treated in the most suitable way making it easier to understand, for example modelling an object-oriented module interconnect architecture via object diagrams. Furthermore, views allow the architecture to be reviewed in parts, so the attention of a review can focus on the essentials of that particular view.

The number of views that have to be distinguished in the description of an architecture strongly depends on the system that has to be realised. The more complex a system, for example real-time embedded or parallel applications, the more views are necessary in the architectural description. Normally, the initial number of views will never be too big, because people tend to describe too little instead of too much. Missing views are caused by not describing certain issues of the architecture or by describing multiple issues in one view. Finding missing views can be done by looking at architectures of comparable systems and by checking that all parts of a view's description serve that view's purpose. When the latter is not the case, the part that does not fit the purpose either has to be located in another existing view or in a completely new one.

When using multiple views to describe an architecture, there is bound to be duplication of information, for example the names of modules and relations that are relevant in more than one view. Duplication of information can lead to inconsistencies within the architectural description. To avoid duplication as much as possible, the maximum number of views has to be limited to four or five. Furthermore, inconsistencies can be avoided by reviewing and using tools.

From measurable to general quality criteria

The effects of the measurable quality criteria (layering, coupling, cohesion, documentation, and views) are directly proportional and strongly related to each other. A layer is in fact a special kind of module, due to its hierarchical nature. Furthermore, using functional cohesion within modules will mostly reduce the coupling between modules. Documentation and views are related to all other criteria, because they are used to describe them and provide a means to deploy the architecture. When an optimal number of layers, weak coupling between modules, functional cohesion within modules, sufficient views, and good documentation can be achieved, the general quality criteria (clarity, testability, maintainability, and extendibility) of an architecture will benefit.

Testability is a criterion that deserves some more attention. Not only because it is often neglected, but also because weak coupling, functional cohesion, and good documentation alone are not sufficient to increase it. In order to test efficiently and effectively, it is necessary to have facilities for automatic testing. Incorporating these facilities in a system after its architecture has been deployed mostly creates problems, because serious changes have to be made to the architecture. Therefore, facilities for automatic testing have to be built into the initial architecture.

Re-usability is a general quality criterion that was mentioned but not elaborated on. Creating re-usable modules or layers in an architecture is not directly

beneficial to the current project, but it may be to the company. Re-usable modules can decrease the development effort in future projects that require these module, thus decreasing the cost for those projects. Fortunately, the measurable quality criteria have the same effect on re-usability as they have on the other general quality criteria.

Conclusion

Despite the fact that an architecture is partially problem specific, it is possible to construct general guidelines to assess the quality of an architecture. This article proposes a number of such guidelines, but a more detailed set of rules and domain knowledge is required for an actual assessment.

In essence a sound architecture is one that has functional coherent modules, has weak coupling between modules, and is well documented. This makes an architecture clear, testable, maintainable, and extendible.

The measurable quality criteria mentioned in this article are not solely suitable for assessing an existing architecture, but can also be used to create an initial architecture. Many times, an initial architecture is constructed, reviewed, and deployed. However, why not assess it before it is deployed? Such an assessment can be made part of the review phase. By assessing an initial architecture in a number of iterations before deploying it, the quality can be improved significantly.

References

[POSA] *Pattern Oriented Software Architecture* F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. ISBN 0-471-95869-7

[SA] Software Architecture *Perspectives on an Emerging Discipline* M. Shaw, D. Garlan. ISBN 0-13-182957-2

[SD] *Structured Design* E. Yourdon, L. Constantine. ISBN 0-138-54471-9

About the author

François Vonk is currently working for Alert Automation Services b.v. as a Senior Technical Designer in the field of embedded software. On the side, he is involved in Software Process Improvement by giving a course in Personal Software Process. This article is the result of a study into the discipline of software architectures that was started by Alert Automation Services. François holds an M. Sc. in Computing Science and a Master of Technological Design in Software Technology, both from the Eindhoven University of Technology.