Defining Software Architecture as an Emerging Discipline for Software Design

Jorge L. Ortega–Arjona
Departmento de Matemáticas, Facultad de Ciencias, UNAM. Ciudad Universitaria D.F., México.
Tel. +52 56 22 4858, E–mail jloa@fciencias.unam.mx

Abstract

This paper considers the term "Software Architecture" as an emerging discipline to the general topic of software design. Attempting to avoid using it as a buzzword, the term is examined to propose an adequate definition related to the representation and composition of software systems. This definition is justified by discussing its consonance with the current usage and its applicability to the class of software artefacts as presented in published works. In accordance with the definition proposed, the actual state on Software Architecture study and research is presented as progresses made in the main fields of software development. Next, Software Architecture is proposed as described or constituted by three areas that taken together, contribute to represent it. In conclusion, the relevance of Software Architecture is discussed, which depends on how the discipline can grow in maturity from its actual stage, so it may serve as a paradigm for a rational theory of software design.

Keywords: Software Architecture; Software System; Software Design, Theory, and Technology.

Resumen

Este artículo considera el término "Arquitectura de Software" como una disciplina emergente del tópico general de diseño de software. Tratando de evitar usarlo como "palabra de moda", el término se examina para proponer una definición adecuada, relacionada con la representación y composición de sistemas de software. Esta definición se justifica, discutiendo su consonancia con el uso común, y su aplicabilidad a la clase de artefactos de software como se presentan en los trabajos publicados. De acuerdo con la definición propuesta, el estado actual del estudio e investigación en Arquitectura de Software se presentan como progresos hechos en los principales campos de desarrollo de software. En seguida, la Arquitectura de Software se propone descrita o constituida por tres áreas que tomadas juntas, contribuyen a representarla. En conclusión, la relevancia de la Arquitectura de Software se discute, la cual depende de cómo la disciplina puede madurar a partir de su estado actual, a fin de servir como un paradigma para una teoría lógica de diseño de software.

Palabras Clave: Arquitectura de Software; Sistema de Software; Diseño, Teoría y Tecnología de Software.

1. Introduction

The world of computing, like many other fields of human activity, seems to be dominated to some degree by the dictates of fashion. Certain topics of study become fashionable and are widely pursued and applied, while others languish for decades in relative neglect. At times the reason for the modishness of a particular topic is quite apparent, and derives from practical importance. The application of telecommunications and Internet is enjoying an unparalleled vogue at the present time – no need to ponder why. In other cases, however, the fashion seems to be more a matter of words than of practice. A new word, perhaps a new name for a familiar activity, may aptly embody ideas in common currency. In a favourable case, it may generate a whole "mystique" by virtue of the very perfection with which it expresses a nascent idea.

A good example is the name S*ystem Engineering*, proposed by H. H. Goode and R. E. Machol (1957), which fostered the realisation that problems of control are one, capable of a unified treatment whether encountered in a biological or engineering context. Overnight, where there had been information theorists, servo–engineers, neurologists or behaviourists, a new "race" of system engineers emerged. Almost every facet of human experience, from quantum mechanics to business management, had to be re–evaluated from a system engineering standpoint. This phase of high fashion (and over selling) have lasted for many years, and it was been succeeded by a reaction in which the name fell from esteem, when overinflated expectations were supplanted by disillusion. Since then, and of course, the valuable work of responsible development has continued, and System Engineering has been established as a branch of knowledge which is neither worthless nor a panacea, but of real and growing value in the understanding of certain problems.

The discipline in current danger of a fashionable vogue is *Software Architecture*. It is not very clear from a study of literature who started the use of this name. Certain it is, however, that the remarkable book "The Mythical Man Month", by Fred Brooks Jr. (1975) made the expression "system architecture" meaningful to a large public. More specifically, within the software literature, the paper "Foundations for the Study of Software Architecture", by Perry and Wolf (1992), used of the term for describing the design of software systems at the organisation level. Since then, the "architectural approach", "the overall view of software", or simply, Software Architecture, has become something of a *sine qua non* of modernism in certain fields of software design. Many books have been published, concerning themselves with Software Architecture. Nevertheless, in most cases, their contents tend to represent conventional textbooks of software engineering, software programming, embedded applications, and so on.

The result is that, as it was the case of System Engineering, Software Architecture seems in danger of being over sold, and hence, of passing through a similar cycle of boom and slump. This is harmful because it will slow the acceptance of Software Architecture as a normal discipline, hindering its absorption into the general body of technology and science. Therefore, it seems that it might be a fruitful exercise to examine *what* Software Architecture is, and to determine (so far as may be) the limits of its competence. In this way, it seems not unreasonable to hope that its relevance to other aspects of the problem of software design may become apparent.

Concrete definitions of the expression *Software Architecture* by recognised authorities are but few, and those few tend to be periphrastic. It will therefore be convenient to put forward an argument as to the nature and scope of Software Architecture in didactic form, and then to consider how well this exposition accords with notions of the subject implicit in published work in the field. A natural starting point seems to be to consider what software designers mean when they speak of a software system.

2. What is a Software System?

Many products may be regarded as an integration or assembly of parts. The parts are independent, following the cohesion heuristic, that is to say that their properties can be defined without reference to others (Parnas, 1972). A software product made up in this way of a collection of separable parts is called a software system. The parts of which it is built are usually called software components (Rechtin & Maier, 1997).

The components of the software system are the simplest parts that are to be considered as distinct units. Just how simple these parts are is a quite arbitrary choice. Often the software components, so called, are themselves made up of still simpler sub–components. However, in a given analysis or description of a software system, it is usually convenient to limit the resolution to a certain level of complexity, and the parts so revealed are simply thought of as components. This does not, of course, preclude the possibility of sometimes referring to sub–systems, that is, an assembly of software components forming less than the complete software system.

The simpler the components are chosen to be, the simpler will be their design and implementation (in general), but, of course, the more there will be of them. Thus, the choice of the components is often an attempt to strike a balance between analytical problems arising from over–complex descriptions of individual components, and others arising from the sheer number of components to be handled.

However, there may be other bases of choice. Quite often, the software designer chooses software components from a list of *available* units (the meaning of "available" here will be considered below), in which case what is to be treated as a component is, in effect, fixed. Thus, for example, it has for some years been quite possible to design a software system or application using ready–made parts. The design of the software system is based on the properties of the available components and how these are assembled together, in many cases without the need for any non–standard part. Thus a complete software system may be constructed as a simple assembly–job of software components, following a pre–defined relationship, configuration or pattern, using only the simplest tools, and perhaps without coding at all.

The economic advantages of this kind of software design in which the required software system is an assembly of pre–implemented, standardised parts, is self–evident. By permitting production runs on several types of software components, the reduction of full production costs can be realised by reducing possibilities. Another important advantage which should not be overlooked is the investment on special tools for the assembly, and on training assembly operatives. Tools and training are more likely to transfer to a new design when a software system is changed, if the old and new designs are based on a common family of standardised software components.

There are also other less obvious benefits to be gained. Use of large volumes of types of standardised software components makes it economic to carry out large scale life testing, and can thus lead to a more reliable software system. When software systems are built from standard parts, feasibility checking is facilitated. The list of advantages could continue, including such things as simplification of component inventories, ease of repair by substitution and so on. The main disadvantage is a certain limitation on the software designer's freedom.

Software components are described here as *available*, but this term merits some further consideration. What is meant by *available* is either that the software component can be had at once, say from a commercial provider, or that there is a reasonably high probability that it can be implemented by modification or development of other existing components. In the latter case, software products are being considered within the category of things known to be feasible and makeable, either by virtue of being within the range of current technology, or so little beyond it that the prospect of successful development is good. Thus, available software component in this sense can be thought as having a high probability of being available in pre–implemented form at the time when the software system is assembled. From this point of view, whether they are actually in existence at the time when software design is carried out or whether it is merely known that they could be produced is of secondary importance.

Whenever the number of software systems of fairly similar kind to be produced (or the number of types of components in a given system) is large enough to permit component production on an extensive scale, the balance of advantage tends to swing strongly in the direction of software system construction as an assembly of pre–implemented components. Thus, the design problem resolves itself into selecting the right components from those available and inter–connecting or assembling them in the right relationship, configuration or pattern.

Therefore, Software Architecture is defined here simply as *the discipline or study of designing software systems, as the result of an "assembly–job" of software components*. Although the emphasis is commonly on existing standard components, the use of available components specially made for the software system is not excluded.

The essential basis of Software Architecture, so understood, is a concern with the assembly of components, whose properties are largely given. Hence, a *software architect* (who acts as a translator between the client's problem domain concepts and the programmer's solution domain concepts) is concerned with the problems of deriving the overall software system properties from those of the components, and with questions of compatibility between components with the software system as a whole. The software architect is not, however, concerned with the detailed design of elements, but is satisfied to treat them as having certain established specifiable properties. To sum up, the software architect's attitude to the software system may be either analytical or synthetic, but regarding all its components under a purely phenomenological light.

3. Justifying the Definition

A definition of Software Architecture concerned with software systems which can be resolved into an assembly of software sub–systems demands a further justification. A definition for a technical term can be justified if it is consonant with current usage, if it is operationally functional in the sense of making decisions about its applicability possible in almost all likely cases, and if it defines a class not adequately represented by any other term in use.

So far as coincidence with current practice of Software Architecture is concerned (SWEBOK, 2001), it will be convenient to refer the books published by Buschmann *et al.* (1996), Shaw and Garland (1996), and Bass, Clements and Kazman (1998). Whilst these books may be more representative of American and European practice, they have commanded wide readership.

Perhaps Bass, Clements and Kazman (1998) seem to propose the clearest approach to Software Architecture as expressed here, offering a formal definition by making the following points:

(1) they are dealing with software systems, which have some features in common,
(2) all software components of such software systems have "integrity", in the sense that they contribute to a single aim, and produce a required set of outputs from given inputs,
(3) such software systems are large in terms of number of software components, cost, and other cognate criteria, and
(4) they are artefacts created from a business perspective, in the sense that they are wholly or largely developed as the result of a business cycle, which includes human influence.

Shaw and Garland (1996) are also concerned with describing and formalising the Software Architecture of complex software systems, mainly by analysing case studies encountered and developed in the software systems literature.

Buschmann *et al.* (1996) also define the term Software Architecture in the very general sense of a set of interrelated sub–systems and software components. However, the books aims more properly to expose how Software Architecture is developed as the result of a software design activity, based on Software Patterns. In fact, such Software Patterns constitute a terminology (a language) for the software system synthesis, in which each pattern defines and describes a set of interacting software components. Incrementally, Software Patterns characterise structural and functional components, in the sense in which the term Software Architecture has been defined here.

All these authors consider software design from an assembly job standpoint. In the book by Buschmann *et al.* (1996) this is explicit. They consider Software Patterns as topological system models which contain elements which are considered available in the sense considered here. So far as Bass, Clements and Kazman (1998) and Shaw and Garland (1996) are concerned, software system synthesis is considered less explicitly, but again the construction of a software system from an assembly of available parts seem to be of the essence of their argument.

Given that the definition proposed is not too ill according with the current usage, is it functionally valid? Does it permit the easy distinction of a class of software artefacts to which it applies? The weakest point in this respect is the concept of "availability" discussed above. It is desired to restrict Software Architecture as a name only for those situations in which software component properties are in some sense given, and yet not to insist that the components are in actual existence at the time at which the software system design is conceived. This is a weakness, since it makes the decision whether or not a software component is available, in this sense, a purely probabilistic matter: however, the so–called crucial components, in which availability is questionable, are few in number in typical software system designs. This must be so if a reasonable completion probability is to exist for the whole software system.

4. The Actual State on Software Architecture Study and Research

Given that the term Software Architecture has the meaning proposed here, what have been its proper concerns up to date? To date, the primary interest in published work has been in the problem of synthesis. This partly arises from the pragmatic importance of the software industry. Also, this is partly because of the assumption that a software system is designed as an assembly of software components, which are drawn from a "catalogue" of available parts, and making a systematic design procedure at least feasible. In this situation, the design theory has been reduced to the creation of a "rule book", giving the rules of assembly. Some progress has been made in problems of synthesis, and it may be of interest to summarise briefly the main fields of study and research work, which are related with the given definition of Software Architecture.

(a) Specification studies. The starting point for software design is a specification. Broadly this is a statement of the purpose that the software system is to serve, the factors available for achieving this, the situation or environment in which the software system is required to operate, and means by which a good software system can be distinguished from a bad one, that is, a measure of the value of the proposed software system. Almost invariably the specification is presented to the software designer in incomplete form. So, the first task is to complete it. This aspect of Software Architecture has been a major preoccupation in the published work in the field (Bass, Clements and Kazman, 1998; Shaw and Garland, 1996) .

(b) Studies of software design situation. Another topic of interest is the interaction between the clients and sponsors (who want and finance the software system), the software designer, and the component designers. This constitutes the design situation in which the design process is carried on. There would be scope for much research of this situation from a sociological, economic, psychological and communication theoretic standpoint.

(c) Architecting heuristics. A major concern of system synthesis is, of course, the heuristic problem: how a software design can be created in practice from a specification. In a chapter devoted to Software Architecture, Rechtin and Maier (1997) retrieve and report the most representative work in this area.

(d) Change and Evolution. The change in properties of a software system due to alteration of some of its components can be regarded as a species of change and evolution study. There is much room for further work in design for change and evolution of software systems.

(e) Software design realisation. An important stage in the design of a software system is the connection of a functional description in problem domain with a software representation in the solution domain. This software representation possesses the right relationships of data to results, represented by a software program. Such an activity involves viewing and linking each functional description with some, perhaps coded, software representation. The best examples of software design realisation are provided as Software Patterns by Gamma e*t al.* (1995), Buschmann *et al.* (1996) , and many others.

(f) Performance studies. Performance is an attribute of a software system, which is generally neglected during software design and development. Nevertheless, it is important to consider performance as integral to software design, while minimising the amount of work which do not ultimately issue efficient software systems. This has led to the development of the technique of performance engineering: a method of forming a running estimate of the probability that the software system being studied will prove not only for a successful operation and capable of construction, but also considering performance goals at an economic cost. Smith and Williams have published several papers about software performance studies – see, for example, Smith and Williams (1993), and Williams and Smith (1998), which represent a good example of this approach.

(g) Software environments. The environment in which the software system operates may be neutral, co–operative, or competitive so far as the operation of the software system is concerned. Competitive environments are the most usual. This is particularly apparent in the case of particular software applications, but is also true of any software system operating in a commercial environment, or in the presence of other software system designed to achieve equivalent ends.

(h) Failure design. All software systems are subject to failure, and usually have many different ways in which they can fail. The relative probability of different modes of failure can be varied by choice of the particular realisation adopted for the required software system. Since different cost penalties are associated with different modes of failure, each realisation will thus have a unique expected failure cost. It is also

possible to reduce failure costs by improving the software reliability, either by improvement of components or by the introduction of suitably connected redundant components, but however at a calculable first–cost penalty. Failure design aims at optimisation of the software design so as to minimise the sum of first and subsequent costs from this cause.

(i) *Distributed software design.* The distribution of a software system is related with the physical disposition of its components, executing on hardware resources in space. Generally, the restrictions on relative disposition of components in software systems executing on a single computer are fairly weak. However, this tends to have a strong impact when considering a distributed software design. This has resulted in the development of a variety of simplified standardised distribution methods for software systems, which tend to minimise either design effort or software production costs, or both.

In addition to these aspects of the problem of software system synthesis, the Software Architecture literature also has concerned about notation techniques that aim for software system analysis. It is recognisable the Unified Modeling Language (UML) as the most widely accepted notation technique originated in the Object–Oriented literature. Proposed by Booch, Rumbaugh and Jacobson (1997), UML has been accepted and used as the standard notation since its introduction for describing software systems. Another notation technique by Bennett (1996) provide diagrams for describing software systems in a more tangible form, and aiming for introducing explicit measures about the "goodness" of design. The list of notation techniques could continue. To some extent it is true to say that notation techniques develop as the need for them becomes apparent from difficulties arising in system synthesis.

4. A Brief Descrption of Software Architecture

Considering the definition provided for Software Architecture, the intention now is to propose a description of the areas that taken together contribute to represent Software Architecture as a new discipline for software design. The structure and content of a Software Architecture should be enough to allow the communication and transference of knowledge of real software architectural work. Considering this, Software Architecture can be divided generally into three broad areas: design, theory, and technology. The majority of the activities in software development practice deal explicitly with one of these areas, although there are overlaps. Design necessarily considers theory and technology. Theory is the study of different design solutions and style approaches during the history of a software system. Technology represents how to apply theories and methods for carrying out architectural design concepts. This interrelation of areas of speciality can be considered the keystone of the discipline of Software Architecture, itself as an amalgamation of disciplines.

4.1 Software Design

Software design can be considered as the unifying element, the primary activity that brings together all the diverse contributing elements in Software Architecture. In general, software design really refers to the representation or descriptions of software systems (graphical or textual). There are several different kinds of informal and formal representations, which are particular or correspond to different phases of software design. Thus, the study of software design is the study of fundamental principles and techniques of software representation and composition.

4.2 Software Theory

Software theory is the second broad area of study within Software Architecture. By studying theory, the software architect learns the theoretical background on which software systems are based. The theory considers the developments of precedent software systems. This may be approached by time periods or by specific topic of interest (focusing, for example, on paradigms, technologies, approaches, designer and programmers, philosophy, or software system types, etc.). The study of precedent software systems and their associated theoretical basis introduces to a legacy of previous programmers and software designers, and suggests how that legacy might be applicable to the present or future systems. Looking at theory precedents discloses ideas, beliefs, and principles relevant to what software developers do today, and perhaps, will do tomorrow.

4.3 Software Technology

Software technology categorises the third element of Software Architecture. It encompasses subjects dealing with how software designs are implemented, and with principles and methods of software implementation and control. Ideally, technology should not be considered as separate and different from software design. Instead, it is a label for activities that are more closely associated with science, engineering, and management, as to both content and method. Within the sphere of technology, some sub–technologies can be considered by now: programming languages and algorithms, interface technology, and management. Some of each can be found in all software projects, although in different doses and with widely ranging levels of application.

4. Conclusion

All the forgoing may be considered as providing the necessary ground work for defining the term Software Architecture as an emerging discipline of the more general topic of software design. If the definition of Software Architecture proposed here is accepted, it will be seen as a particular case of the general software design problem, as described by Kapor (1991). Indeed, Software Architecture attempts to embody even the actual process of designing a real software system.

Nevertheless, many considerations in Software Architecture made by now are not exactly paralleled in reality. For instance, the design situation of a sponsor or client, a software designer, and component designers, as separate individuals activated by rational motives, in many cases is not quite so. These entities might probably be groups, and their motivations might (at least in part) be determined by interpersonal relationships within and without the group. Similarly, only rarely will a software system be assembled entirely from available software components. More commonly, only some crucial components will have marginal availability status. Perhaps much specification data will be unavailable, software users will not behave rationally, the software environment will change unpredictably as a result of uncontrolled developments, and so on. A more sophisticated second generation software design theory will take some of these factors into account, while practice will continue contributing with uncertainty factors to design software. However, as it was the case of many others disciplines, a naive formalised theory has proved a valuable starting point.

Just as Software Architecture can grow in maturity by progressively complicating its model of the software design process, so it seems not unreasonable to hope that the whole discipline of Software Architecture may serve as a paradigm for a rational theory of software design. By a rather serious restriction on the range of competence, concerning itself primarily with assembly–job of software systems in a design situation, it seems possible to develop useful synthesis techniques. Some (such as architecting heuristics and failure design) can be taken over directly into a more general context; others (such as performance studies) may need fairly a radical modification; yet others (change and evolution) may have little application in fields outside Software Architecture.

As software system synthesis and analysis techniques grow in potential, it becomes profitable to try where possible to re–express other kinds of software design problems in architectural terms. Nevertheless, if the term Software Architecture is allowed merely to become a fashionable tag for the study of any fairly complex software product, it will fall into disrepute. However, if it is applied to a particular kind of design activity, which, as described here, seems to be more and more characteristic of the maturing technologies, it can continue to be useful.

5. References

Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice*. Addison–Wesley.

Bennett, D. (1996). *Designing Hard Software – The Essential Tasks.* Manning Publications Co.

Booch, G., Rumbaugh, J., Jacobson, I. (1997) *UML notation guide, version 1.1* Rational Software Corporation, Santa Clara, CA, September 1997.

Brooks, F. (1975). *The Mythical Man–Month.* Addison–Wesley, Reading, MA.

Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., and Stal, M. (1996). *Pattern–Oriented Software Architecture. A System of Patterns.* John Wiley & Sons, Chichester, England.

Goode, H.H. and Machol R.E. (1957). *System Engineering.* McGraw–Hill, New York.

Kapor, M. (1991). *A Software Design Manifesto: Time for a Change.* In Bringing Design to Software, T. Winograd (ed.), ACM Press 1996, Addison–Wesley. Originally published in Dr. Dobb's Journal 172.

Gamma, E., Helm, R., Johnson, R. Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object–Oriented Software.* Addison–Wesley.

Parnas, D.L. (1972) *On the Criteria to Be Used in Decomposing Systems into Modules.* Communications of the ACM, Vol. 15, No. 12, December 1972.

Perry, D. E., and Wolf, A. L. (1992) *Foundations for the Study of Software Architecture*. ACM SIGSOFT, Software Engineering Notes, Vol. 17 No. 4.

Rechtin, E. and Maier, M. W. (1997). *The Art of Systems Architecting.* CRC Press.

Shaw, M. and Garlan, D. (1996*). Software Architecture: Perspectives of an Emerging Discipline*. Prentice–Hall, Englewood Cliffs, NJ.

Smith, C. U., and Williams, L. G. (1993) *Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives.* IEEE Transactions on Software Engineering, Vol. 19, No. 7.

SWEBOK (2001) *Guide to the Software Engineering Body of Knowledge, IEEE Trial version 0.95*. A. Abran, J. W. Moore, P. Bourque, and R. Dupuis (eds.). Computer Society Press, May 2001. `http://www.swebok.org/`

Williams, L. G., and Smith, C. U. (1998) *Software Performance Evaluation of Software Architectures.* Proceedings of the 1st International Workshop on Software and Performance, Santa Fe, New Mexico, United States, 1998.