The Concept of Software Structure and its Relations with Software Architecture and Software Patterns

Jorge L. Ortega Arjona and Graham Roberts
Department of Computer Science
University College London
Gower Street
London WC1E 6BT, U.K.
May, 1998.

Abstract

In this paper, we introduce and analyse the definition of software structure as a fundamental concept for software design and construction. This has lead us to consider a more precise definition of software architecture, which contrasts the difference between software architecture and software structure after reviewing other definitions of software architecture provided by several authors. The goal is to define and clarify the concept of software structure, and to discuss or at least promote discussion about its relation with software architecture and software patterns for the design and construction of software programs.

1. Introduction

In order to define the concept of software structure, it is necessary to have a clear concept of software architecture, by analysing previous definitions and approaches. From these approaches, the more significant seems to be "The Foundation for the Study of Software Architecture" [4], in which Perry and Wolf propose *"a philosophical foundation for the model of software architecture, developing an intuition about software architecture through analogies to existing disciplines"*[4]. In this paper, it is considered that the field of building architecture *"provides some of the more interesting insights for software architecture"* [4]. Probably building architecture has been chosen because it is the field of construction that may provide us with the necessary experience and insight about how to structure and construct systems. Therefore, when we need to understand or explain some concept more clearly, we will point to an analogy between building architecture and software architecture to try to explain the concept.

2. Software Architecture

Criticism has arisen from the comparison between a building and a software program. Probably this is because the comparisons have tried to consider a software program in exactly the same terms as a building; that is, as an entity of space. However, it can be considered that a software program is not an entity of space, but more precisely an entity of time. A software program can be conceived as the organization of tasks through time, in contrast with the building, which is the organization of activities though space. Building architecture is the making of space, involving the organization of three dimensional space by the creation of a building to perform certain activities. In an analogous way, software architecture is proposed as the making of time, in which the organization of tasks through time is carried out by the creation of a software program that performs such tasks. This particular approach obeys intuitively philosophical and practical experiences with programming software programs.

The temporal nature of software has being expressed over and over in most of the software architecture bibliography. An example of this is found in [4], when Perry and Wolf address the process and data interdependence:

- *"there are some properties that distinguish one state of data from another; and"*
- "*those properties are the result of some transformation produced by a processing element"*.

Considering both statements, intuitively it can be seen that the temporal nature of software programs is due to them being conceived as a set of transformations through time. Just as the work of a building architect is organize the space, in which a building represents an organization of tridimensional space, then the work of a software architect should be to organize time. A software program represents the organization of the activity of processing elements through time.

2.1. Another definition

The concept of software architecture is difficult to deal with. It seems to be slippery. A large number of definitions have been provided by a similar number of authors [1]. After reading and analysing them, we propose a new one, based on the previously mentioned analogy.

Following the analogy that building architecture is defined as *"the art of planning, designing and constructing buildings"*[2], we simply state:

Software architecture is the art of planning, designing and constructing software programs.

This definition may seem shallow, but this will be clarified when we introduce the concept of software structure.

2.2. The role of software architecture - *firmitas*, *utilitas* and *venustas*

In order to introduce the definition and role of software structure, we should first define the role of software architecture. Again, we support our approach with an analogy to building architecture, specifically on a classical definition provided by Vitruvius, who when asked about the role of the architect replied: *"the architect is a technically skilled and sensitive man, who is capable of imbuing a building with firmitas (firmness), utilitas (commodity), and venustas (delight)"* [3]. Perhaps a more useful modern translation that is possible to apply as well to software program construction would be: structure, appropriateness and (why not?) aesthetics.

- **Software structure.** A software program first of all has to be structurally sound, keep out the elements and keep its occupants comfortable. Software structure is precisely the object of our analysis. We will expand this point further in a following section, introducing an intuitive but more precise definition of software structure.

- **Software appropriateness.** A software program must be fit for the use for which it is intended. Software appropriateness is represented in the form of a software program that covers a set of functional and non-functional properties.

  Functional properties denote aspects of the functionality of a software program, commonly related to a specified functional requirement. A functional property is visible to users of the software program by means of a particular functionality or aspects of its implementation. Functional properties expose *what* the software program is capable and not capable of doing [5].

  Non-functional properties deal with features of a software program that are not covered by its functional description. Typically, a non-functional property addresses aspects related to the reliability, compatibility, development effort, ease of use, maintenance, etc. of a software program. In general, non-functional properties reflect *how* the software program performs. These attributes are qualified, but difficult to quantify [5].

- **Software aesthetics.** A software program must be pleasing to the senses, otherwise it is not a result of software architecture. However, aesthetics in terms of software is a difficult issue. By definition, beauty is in the eye of the beholder. Software appearance is not something that can be judged impartially. Often, criticism of software is directed by how it looks, instead of how it works. What seems to be happening in the realm of aesthetics in general is that there are as many ideas of what beauty is as there are critics.

A final remark that concerns the three characteristics is about the important matter of software evolution. Software architecture should provide with elements to guide the evolution of a software program through time. Software structure should be defined in a form that allows and supports software evolution. Software appropriateness and software aesthetics represent and reflect the changing properties that a software program experiments during software evolution. Dewayne Perry addressed this dependence when stated *"Architectural Evolution is the fundamental challenge"* [7].

3. Software Structure

3.1. Definition of Software Structure - the difference between Software Architecture and Software Structure

Typically, the term architecture is used, overused, and some times even abused when trying to define more precisely the structure of a software program. Even though *"architecture of something is its structure"*[2], and many authors follow this definition, we consider that the use of such definition may lead to unfortunate misunderstanding.

A software program has a structure. Software architecture helps to define a software structure for a software program, but a software structure does not define either a complete software program, or a software architecture. Many of the definitions found in [1] do not consider this subtle difference.

Probably, the previous statement could be clarified by appealing to the analogy. Consider a building as the result of the building architecture. It posses a defined structure as some elements that serve to hold it up, while other elements clad, decorate, subdivide, or enhance its use. In the same sense, a software program should posses a structure in the form of some elements that serve wholly or in part to hold up the program, while other nonstructural elements clad, decorate, subdivide, or in some other way, enhance its use.

Simply defined, *"the structure is something that consists of parts connected together in an ordered way"* [2]. Comparing this definition with those definitions of software architecture, it can be observed that in general, what has been defined as software architecture is more precisely software structure. The software structure, then, is more than only an enumeration of components and connectors. A rationale about how they are ordered and organized is an important element of it. From this perspective, almost any definition contained in [1] can be taken to define software structure. Anyway, we propose a particular definition, according to our approach:

> *Software structure is the set of elements that serve wholly or in part to give form and hold up a software program, and provides support for the logical reasoning about the order in which data is operated on by operations.*

3.2. Software Structure properties

As mentioned before, our basic consideration is that a software program is an entity of temporal nature. From this perspective, software structure should share such a nature, that is, software structure nature should also be related to time. Understanding and considering the properties in time that a software structure should accomplish is an important step for software design and construction. We are still studying these characteristics, but up to now, we had noticed some principle properties that we expect that a software structure should present: stability, composability and geometry.

3.2.1. Stability

A good conception about software structure is its reactive nature. Software structure should put together and provide stability to its software program, just in the same way a building structure keeps a building stable. However, this should be a *time* stability, that is, the software structure of a program is related with those parts of the program that determine the main actions in the program and tend to remain stable during execution and during the program's life time. This does not mean that the software structure is not expected to evolve and change. Software programs are not an static entities. They are supposed to change as the result of the activities performed by designers and programmers. However, different parts of a software program change at different rates. Specifically, it would be proper that software structure evolves slowly, supporting all other elements that tend to present a faster change. By understanding this, the design, construction, maintenance and modification of software can be performed in a safer way. However, a first difficulty seems to arise in how to recognize what parts of a software program represent structural and non-structural members. In a software program, elements that compose a structure are not defined as clearly as in a building. A possible solution may come from software patterns.

3.2.2. Composability

Another interesting property that we consider software structure should accomplish has been addressed during the last few months by Coplien [6] when explaining the topic of *sketch* as a way to describe structure. In this article, Coplien denotes that a sketch can be used to portray the basic structure of software programs, and contrasts its use to design diagrams using standard notations:

> *"How are Alexander's sketches different from OMT, UML, or Booch notation? My colleague Joe Davidson has an interesting theory: He notes that OMT diagrams don't easily compose, whereas sketches do. (Try composing OMT diagrams for two patterns you use together)."*[6]

Describing software structure in the way that it is possible to compose it from its basic elements as a sketch seems to be a good option. The proposal is then a sketch that depicts the temporal relations between the software structure components. Keeping key relations stable, and a structure in time seems to arise. Furthermore, composability is an important property for piecemeal growth [6].

3.2.3. Geometry

The development of building architecture in the way we know it nowadays is mainly due to an important relation between the development of geometry expressed in mathematical terms and its applications to building structures. Precisely, this relation is based on the space order of physical components in a building's structure, and a mathematical geometric representation that allow calculation and prediction of the behaviour of the structure.

Considering this point of view, we think that it is important to develop a kind of geometry applied to software programs in the time domain. This geometry should depict the time order of components and their interaction through time, allowing perhaps in the same fashion to calculate and predict its behaviour. In summary, the relation between an abstract, mathematical geometric model and a software program should be structural in time terms. Geometry explains a software program in terms of order, facilitating to think about it and providing with a sense of software aesthetics. Again, we are still trying to find and define such kind of description and geometric modelling.

4. Software Patterns and Software Structure

Software Patterns have been proposed as an important approach to support the development, construction and evolution of high-quality software programs, complementing exiting techniques, methods and processes of software architecture [5]. In this sense, they can help to identify, design and construct the structural and non-structural elements needed to compose a software program. The software patterns approach proposes a flexible design and construction order of elements of a software program. This order can be observed as the answer to the following question:

> *"... at what point of development should patterns be used: during analysis, high- or low-level design, or even during implementation? There is no single correct answer, but a rule of thumb is that you should use the high-level architectural patterns earlier than medium-level design patterns, which are themselves used before idioms"* [5].

This statement expresses an order relation among pattern categories (architectural patterns, design patterns and idioms) that are related with the structural and non-structural elements during different phases of software design. Architectural patterns defined from the order of provided data and required operations may help to design and construct the fundamental structure of the software program. On the other hand, non-structural elements are developed with the support of design patterns to design each subsystem in the program. At the final stage, idioms are used to support the detailed implementation of different elements, transforming the design into code.

Software patterns may also provide support for software evolution. The stability of change in a building is based on an structure that change slowly over time. Software structure has the same objective: to provide stability of change in a software program. Experience shows that difficult problems arise when attempting to modify slow parts at a faster rate than it is allowed. Software patterns can be used to capture information about this change rate of different elements, providing stability through the lifetime of a software program.

5. Summary and comments.

This paper portrays some of the topics of an ongoing research, proposing an initial discussion about the concept of software structure, and its objective during design and construction os a software program. Software structure is proposed as an element of software architecture that can be defined in general using software patterns, and in particular architectural patterns defined from the characteristics of order of data and operations. Software structure is an integral part of a software program, and should be designed aiming to keep properties such as stability, composability and geometry.

Also, it should be recognised that software structure is an important but not the only element of software architecture. Other not less important elements are required to compose a complete picture of how to design, construct and maintain software programs.

As in any ongoing research, each one of the previous statements about software architecture and software structure are open to be further analysed, improved and discussed. In fact, this is the primary goal of this paper.

As a final remark, a statement found in building architecture: *"Understand structures and building systems, and you understand the science of architecture"*[3]. The same statement can be applied to software architecture.

6. References

[1] Software Engineering Institute, Carnegie Mellon University. *Software Architecture definitions.* URL: http://www.sei.cmu.edu/architecture/definitions.html. Last modified: 25 april 1998.

[2] *Collins Cobuild English Dictionary*. HarperCollins Publishers. 1997 edition.

[3] Douglas E. Gordon and Stephanie Stubbs. *How Architecture Works.* Van Nostrand Reinhold, New York, 1991.

[4] Dewayne E. Perry and Alexander L. Wolf. *Foundations for the Study of Software Architecture*. ACM SIGSOFT, Software Engineering Notes, Vol. 17 No. 4. October 1992.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal. Pattern-Oriented Software Architecture. John Wiley & Sons, Ltd., 1996.

[6] James O. Coplien. *Worth a Thousand Words.* Column without a name, C++ Report. May, 1998.

[7] Dewayne Perry and Takuya Katayama. *Critical Issues in Software Evolution*. Panel in the 20th International Conference on Software Architecture, ICSE'98. Kyoto, Japan. April, 1998.