Searching Basic Notions for Software Architecture Design

Jorge L. Ortega-Arjona and Graham Roberts
Department of Computer Science,
University College London
Gower Street, London WC1E 6BT, U.K.
February 1999.

Abstract

Software Architecture is founded on the belief that software design can make use of practices and techniques of design in general, expected to generate software quality products. However, to achieve this, Software Architecture design should be based on a set of notions for design, which allows software architects to think about, understand and communicate their design knowledge. In this paper we search in the fields of design and problem-solving to find an initial set of basic notions for Software Architecture design.

1. Introduction

In order to obtain effective practices, techniques, formalisms and tools to support software design, Software Architecture has to be founded on a set of notions that provides a description of the software design universe. This description should be able to inform about designers' efforts that find their expression in the form of computer programs. Further, the description does not have to be correct, it does not need to refer solely to indisputable facts, but it must be informative in describing conditions that are relevant to an evolving software system, failing or succeeding.

It is possible to think of software design as a particular kind of design activity which may or may not employ well-defined methods, or as descriptions of the products of software design activity. For example, Object-Oriented design attempts to categorize software design according to classes of objects, identifying objects that have known bodies of overt knowledge associated with them. This knowledge might refer to conditions that have to be satisfied during the course of designing an object, and functions that the object has to perform which may partially define the object. Such knowledge can form the basis of well-defined methods which then are incorporated as parts during the software design process. However, this attempt focuses its attention on what we already know how to do: to devise methods that consist of overt stepwise operations and to equate such methods with the whole of a software design process. The weakness of this approach to software design is that it may be too dependent on anticipations of designed objects, their particular properties and functions. When the anticipation of an object does not correspond to its actual requirements, its methods have to be redesigned and so do the computer implementations of the methods.

2. Design in general

Software Architecture is founded on the belief that software design can make use of design practices and techniques expected to generate products that exhibit quality attributes of interest, similarly to Building Architecture [4]. Software Architecture design, like Building Architecture design, can be then conceived based on a description of a design in general. Describing this design in general, then, could be a good start point to search for notions for Software Architecture design.

We can briefly describe design in general as a process of synthesis that attempts to connect a desired function to a solution form [2,8,9]. Design has to achieve a fusion between parts to create new parts, so that the products are recognized as having a right and proper place. The word *parts* refer to anything – physical components, abstract ideas, representations, etc. These parts occur in some high-level context or environment from which parts are extracted, designed upon and results replaced. In all cases, results are judged making reference to that context. It is characteristic of design that both the process and the product are not subject to explicit and complete criteria [3].

This description of design contrasts sharply with the orthodox scientific and technological understanding, which rely predominantly on a process of analysis [2,7]. In this case, the approach attempts to decompose a problem into parts until individual parts are recognized as being amenable to known operations and results are reassembled into a solution. This process has a marginal role in design when evaluating selected aspects of tentative proposals, but the absence of well-defined and widely recognized criteria for design excludes it from the main stream of analytical developments.

From the previous description of design as a process of synthesis, it is possible to identify three key elements of design in general [7]:

- *Design objects.* Design objects are the primary product of design, subject to diversity of expression. As they represent different perceptions of concepts and things, in general lack of agreed abstract definitions.
- *Design processes.* Design processes are not problem solving in the orthodox form of problem statements that reveal solution paths. Usually, they are affected by a context and conflicting criteria for validating results from many solutions.
- *Design knowledge.* Design knowledge is found in general as a no formal, complete and shared knowledge base. It relies on integration of overt and intuitive knowledge, which necessarily manifests in idiosyncratic design practices.

Let us consider in particular design knowledge. Knowledge of designers, what each knows, can be only achieved in their drawings and words, as external manifestations of their thoughts. Intelligent behaviour is perceived essentially as the ability to externalize thought, to give expression to both overt and intuitive knowledge. Good design is the product of other people's perception and interpretation of values and aspirations of those other people. Thus, this understanding of design places responsibility within people, and focuses research attention on formal expressions that pass between people [7].

The description of design previously presented recognizes that design products are inherently not predictable by overt procedures alone, nor by problem-solving methods. Design can be more precisely described as an activity of event exploration and differentiation [3,8,9].

This point of view about design is not intended to support the idea that design is somehow mystical. Instead, it simply recognizes a distinction between abilities that can be represented as overt knowledge, and abilities that remain within individuals as intuition. Overt knowledge can be represented in some formal expression outside individuals, such as written words, so the external representation contains an explanation of the knowledge. Intuition refers to that knowledge acquired directly by individual experience, inwardly learned, that cannot be explained overtly. In any activity in which people are answerable to other dissimilar people, it can be found overt knowledge trying to inform intuition. Intuitive knowledge, then, appears to be decisive in determining people's actions [7,8].

3. Some questions that need answers

From the previous description of design as an activity of event exploration and differentiation, we can state that a set of notions for Software Architecture design should then tell us how software designers perceive things and how they operate on their perceptions, manifest in the overt expression of their thoughts. This knowledge should inform our efforts in devising new techniques, formalisms and tools that can assist the work of software designers. The notions for design need to be general, abstracted from instances of designers' behaviour and not dependent on particular prescriptions for future behaviour. The need to accommodate anything a software designer might think or will have the effect of linking efforts on Software Architecture to equivalent efforts in other fields, an obvious example being Building Architecture.

To search an initial set of basic notions for Software Architecture design, and thinking about the previous description of design, we formulated some questions around software design, considering the universe of things a software designer might think and do to produce a software program. We might model this universe, for example, in an Object-Oriented fashion, in terms of classes, relationships and operations. These constituents of the model imply meanings: classes might correspond to physical or abstract objects, described in terms of their attributes; relationships describe the composition of objects into larger composite objects; operations enable the transformation of objects and composite objects. These meanings imply a certain understanding, a set of basic notions. What we want to know is whether the notions implicit in the model correspond with a software designer's perception, or what different notions are required that will generate new constituents for a new model.

By now, let us consider the following questions around Software Architecture design: Do software designers think in components as discrete elements that are complete in themselves, or as composition of components, made up of separate objects, for example, that are just placed together? Is the description of one component affected as a consequence of changes in the relationships of its internal components? When two components are joined, does each lose its identity as a separate component? Does the notion of components with boundaries have any relevance to designer's perceptions during the course of design? These questions can be extended, exposing yet more questions at more fundamental levels of understanding, and it will be difficult to provide answers. Yet any action, any system that is implemented, perhaps unknowingly implies answers. In the context of evolution of practical applications, these implicit answers are found in retrospect to be wrong when a software program or system fails.

4. Searching Basic Notions for Software Architecture design – Alternative concepts from problem-solving

Design was previously described as not problem-solving. However, we can explore this proposition as a useful way to answer our questions and expose some basic notions for design, by questioning some concepts that we have imported from problem-solving fields. For this, let us identify a typical problem-solving approach by the following necessary constituents [2,8]:

- A *problem*, expressed as a known state of being, within a single well-defined domain.
- A *group of knowledge procedures*, available within the domain, by which a given state may be changed.
- A *goal*, expressed in terms that specify some new state, including the conditions that have to be met by a solution, and the boundaries to the selection of procedures for changing the existing state.

Notice that this definition excludes design if it is considered that a problem-solving approach relies solely on overt knowledge. When it is considered so, many other activities not usually associated with design are also excluded. This observation may suggest that advances in general design theory are likely to prove significant well beyond particular fields of design application.

Furthermore, this view of problem-solving approach rests on some concepts that are important to the internal functions of problem-solving systems [2]. These concepts can be identified as necessary conditions for any problem-solving process. First, any instance of problem has to have a start and finish, and is wholly contained between those boundaries. Typically, problem specifications and solutions can be undone by moving these boundaries. Second, and as a consequence, the wholeness of a problem gets composed into discrete parts, or subproblems, until parts present the conditions that are required by the available change procedures. Third, to ease the task of matching part instances with procedures, emphasis is placed on prior typing parts. Sequences of change procedures provide solution paths. Solutions are found by aggregating results, and problems have to have single (or very few) solutions. Finally, a match between a solution and a goal can then be recognized as a correct result.

These concepts have become widely and firmly established in many fields. However, to apply these to design means a selective decomposition of ill-defined design practices into well defined subtasks that are amenable to a problem-solving approach. Just then we can propose techniques, formalisms and tools that perform analytic functions to evaluate, for instance, performance requirements for a proposed design software. Unfortunately, in the present situation, these techniques, formalisms and tools, by themselves, contribute nothing or very little to our understanding of design synthesis, where synthesis has to reconcile disparate interests in a design product [1,4,5].

The key question we have to consider, then, is whether these concepts that support problem solving are valid as notions for software design and, if not, can we formulate valid alternative notions?

4.1. Wholeness of design

In general, the start and finish of a design appears to be circumstantial [2,8]. The context in which such decision about design occurs is people, and they decide when it is to start and it is finished. There a few overt criteria for recognizing when a design is complete, and in the case of software design, there are no precise definitions of those boundaries.

The boundaries to any instance of design are ill-defined. In the case of software these considerations come from an unbounded domain of any arbitrary subset of all people. We do not know how to draw a boundary around the design interests in a software so that we know we have the whole software [5].

As a typical example, the design of a proxy interface [6] sits in the context of an original object, a group of classes, in a software program, in a computer, in a network, etc. Equally, the proxy interface sits in the context of access and security, affecting the ease of using a particular service by a particular client, etc. Furthermore, the proxy has to be coded for available computer technology, marketed and installed. We can say more about the proxy interface but we do not know where to draw the boundary around it to define a complete and discrete design interest.

4.2. Whole and parts

Decomposition of design into parts has the effect that parts are differentiated according to different domains, with no known relationships between domains that allow results to be aggregated into design solutions [2,3]. In the case of software we do not know how to add the result of a performance evaluation to a result of a reusability appraisal.

Design has to reconcile different arbitrary and contradictory interests in the form of a design product, and this reconciliation cannot be achieved by overt process alone. Parts, as components, are defined by the perceptions of different people, and their synthesis with respect to any perception of the whole is dependent on idiosyncratic contributions of those individuals [8].

4.3. Discreteness of parts

If the existence of parts can be recognized, usually it is not known how to define them as discrete parts [2]. In the case of software, a component tends to be defined by its relations with other components, and changes to these relations change the component. Obviously, it is not practical to work with discrete parts where changes to one part are likely to propagate unforeseen changes to many others [5].

4.4. Prior typing of parts

Part instances occur as unforeseen events that do not permit confidence in prior typing. Attempts at typing either have to be undone as new instances make unforeseen demands on type specifications or they compel conformity of instances, which adds extraneous constraints on a design product [2].

This difficulty becomes still more instructive if we also question the usual distinction between parts and software units, which may be represented by data and/or functions which can be performed by or on such software units [4,5]. It is not clear whether this is a clear differentiation for theoretical understanding of software design; Object-Oriented, for example, considers functions as part of the descriptions of objects.

4.5. Correctness of results

Lastly, consider the notion of correctness of results of design products. Here we have to recognize that design goals generally do not include explicit specifications that allow us to match products so that we can know that we have correct results [2,8]. In the case of software it is interesting to note that there is no single abstract formal definition or representation of a software program. Nor do we have rigorous classifications of software that express the similarities and differences between, say, compilers and word processors among all other instances of software programs [5]. In the absence of such knowledge, goals cannot be explicit, and we have to accept uncertainty in our solutions.

5. Conclusion – Searching for other notions for Software Architecture design

Here, we present an initial set of basic notions for Software Architecture design, obtained from the description of design as a process of synthesis and its contrast with problem-solving approaches. However, what other notions can we think of that will be more appropriate to Software Architecture design and, incidentally, to many other human activities that exhibit properties similar to those of design? The answers could have far reaching implications, but we are at present ill equipped to provide them. Our thinking is too much conditioned by established concepts associated with successful problem-oriented systems. We need to dig deep into theory, to bring to the surface concepts that exist in other theoretical fields, such as architecture, linguistics and, why not, philosophy, and reshape them into a theory that describes software design.

Meanwhile, we can identify other promising questions around Software Architecture design, which probably may guide to validate the basic notions presented, or perhaps add new ones: Can we think of components as unbounded groups of components and independent of types, just components of components of components? Can we conceive of a notion of software process that uniformly embraces different programming paradigm approaches, as well as their activities and events? Can we represent relationships between components that will enable us to bridge across arbitrarily different paradigms and domains to work with divergent interests in software design projects? Can we create an overt systematic environment that will be able to represent any unrehearsed thing in the head of any software designer? Referring to experience of established software systems, the orthodox answer to all these questions would be no, but then we would not be able to design.

6. References

[1] Gregory Abowd, Len Bass, Paul Clements, Rick Kazman, Linda Northrop and Amy Zaremski. *Recommended Best Industrial Practice for Software Architecture Evaluation.* Technical Report CMU/SEI-96-TR-025 ESC-TR-96-025. January 1997.

[2] Christopher Alexander. *Notes on the Synthesis of Form.* Harvard University Press, Cambridge, 1964.

[3] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, New York, 1979.

[4] Len Bass, Paul Clements and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley, 1998.

[5] Douglas Bennett. *Designing Hard Software. The Essential Tasks.* Manning Publications Co. Greenwich, CT, USA, 1997.

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

[7] Francis Duffy and Les Hutton. *Architectural Knowledge. The Idea of a Profession.* E&FN Spon, an imprint of Routledge. London, 1998.

[8] Bill Hillier. *Space is the Machine.* Cambridge University Press, Cambridge. 1996.

[9] Bill Hillier. *A note on the intuiting of form: three issues in the theory of design.* Environmental and Planning Buildings: Planning and Design Anniversary Issue, 1998.