Why Is It So Hard To Define Software Architecture?

Jason Baragry and Karl Reed
School of Computer Science and Computer Engineering
La Trobe University
Bundoora, Vic 3083, Australia
Email: [baragry,kreed]@cs.latrobe.edu.au

Abstract

In recent years, software engineering researchers have elevated the study of software architecture to the level of a major area of study. A review of the published literature however, shows quite clearly that a unified view of software architecture has not been forth-coming. This paper contends that the existence of a "software architecture level of design" is based on the implicit assumption that the software development process is analogous to those "construction" disciplines in which the completed artefacts or systems exhibit a unique representational abstraction, fixed during the early stages of design, which we describe as "the architecture". We argue that our problems in obtaining an acceptable definition of software architecture are due to the assumption that software systems have an analogous, unique design abstraction, determinable at the early stages of the design. To determine the validity of this analogy, we contrast the nature and use of architecture in the traditional building process with software development to identify the differences, rather than the similarities that exist. These differences are explained using a theory of the software development process which highlights why these differences arise and, subsequently, why there has been trouble in developing a community-wide understanding of software architecture. Our conclusion is that due to the fundamental nature of the systems we construct, attempts to depict the large-scale structure of the system, in an analogous manner traditional building disciplines, results in many different architectures. These are fundamentally different representations and not merely different views of a single whole. Moreover, each of these is equally qualified to be labelled as the system architecture with respect to the general notion of what architecture is.

1 The Problem with the Definition of Software Architecture.

Software developers have been discussing the "architecture" of their systems since the late 60's (see Dijkstra [1] and Spooner [2]) however, it was Shaw's 1989 paper, "Larger Scale Systems Require Higher Level Abstractions" [3], and Perry and Wolf's subsequent paper dealing with the foundations of the new research area [4], which triggered the growth in the field we call "software architecture". Despite the volume of research since those papers were published, it is suprising that the software development community has failed to agree on exactly what we mean by the "software architecture level of design". Even the most popularly cited definition, provided by Garlan and Shaw [5], is not universally agreed upon. Descriptions of system architecture range from conceptual models of design to source code organisation and touch on more abstract notions such as frameworks, patterns, and styles [6]. These multiple definitions make it hard to compare and contrast different ideas in the field because they are based on slightly different notions of what software architecture should be and the purposes that it should serve. Consequently, architectural representations of implemented systems depict different concepts depending on which definition is used or the architectural biases which predominate.

Clements, in his overview of the field [7], suggested five reasons why the community has failed to reach on a consensus on what exactly is needed by software architecture.

1. Advocates bring their own methodological biases with them. While most definitions of the term agree at the core, they differ seriously at the fringes. Those differences are attributable to the motivation each researcher has for examining the structural issues in the first place.

2. The study is following practice, not leading it. Research still involves observing the design principles and actions used whilst developing real systems and abstracting the commonalities.

3. The field is still quite new.

4. The foundations have been imprecise. The field contains a remarkable number of undefined and ambiguous terms. In addition to the textual terms, diagrammatic representations of architectural structures also suffer from ambiguity in interpretation.

5. The term is over-utilised and its meaning as it relates to software engineering is becoming diluted.

We assert that the issues raised by Clements are manifestations of a deeper reason preventing us from achieving convergence. Our detailed study suggests that much of the current research in software architecture appears to be based on the assumption that the process of software development is analogous to that of other disciplines which produce built systems, yet this assumption is rarely justified or even questioned.. As a consequence, the research community appears to be subscribe to the following implicit syllogism:

"Traditional engineering disciplines design and build systems which exhibit a level of design abstraction known as the system architecture. Software developers build systems and can identify high level design abstractions. Therefore software systems have a system architecture".

This paper examines that validity of that implicit analogy to determine why it is so hard to develop a community-wide agreement on what we mean by the software architecture level of design. To achieve this end we examine traditional building architecture and identify aspects of the fundamental nature of the systems they build and the materials used to build them. This is then contrasted with the discipline of software development to identify the differences between the fields. Finally these differences are used to determine the validity of the syllogism presented above. Comparisons between software development and traditional engineering disciplines occur in many aspects of software engineering research, however, whilst most papers attempt to highlight similarities between the disciplines to validate the analogy, we attempt to highlight the differences and then determine whether or not the analogies remain valid.

We conclude that, due to the differences in the fundamental nature of the systems we build, and the materials we use to build them, software development does not have a unique architectural level of design. In fact, there exists three activities in the software development process in which high level representations are used and these activities all attract the term 'architecture'. They are:

1. The development of the conceptual model which defines the developers solution to the problem.
2. The arrangement of the static source code into the module and interconnection constructs provided by the programming language and operating environment.
3. The abstractions which depicts the systems conceptual operation and allows the developer to determine how well the implemented source code realises the solution depicted in the conceptual model.

These are not multiple views of the one architecture they are three independent representations, all of which can rightfully be labelled with the term 'architecture'.

2. Comparison of Traditional systems and software systems.

2.1. The Nature of Traditional Architecture.

The architecture of a built thing, in general parlance, refers to it's "unifying or coherent form or structure" [8]. This generic concept is easy to understand when dealing with our vast range of physical artefacts. Moreover, the generic term 'architecture' also appears to be appropriate when referring to the large-scale structure or form of software systems. In what follows, we examine the usage of the term "architecture" in an attempt to clarify its application to our field.

Interestingly enough, many reference books in the field of architecture itself fail to define the term (for example [9, 10]). Moreover, those which do, describe something quite ethereal which fails to assist in the application of the term to software. For example:

"The art of designing and building according to rules and proportions regulated by nature and taste, so that the resultant edifices arouse a response by virtue of their qualities of beauty, geometry, emotional power, picturesque, intellectual content, or sublime essence, is called *Architecture*, a term which suggests something far more significant, sophisticated, and intellectually complex than a mere building, although it must also involve sound construction, convenient planning, and durable materials. ... Architecture implies a sense of order, an organisation, a geometry, and an aesthetic experience of a far higher degree than that in a mere building." [11].

However, while the discipline of architecture itself has proceeded without a formal definition of the term - at least not in the sense that we seek, it does possess two aspects which our discipline lacks.

First, architects have been formally discussing the nature of their discipline from at least as early as Vitruvius' treatise, Ten Books On Architecture [12], in the last century BC. While it is generally accepted that Vitruvius was not the first person to systematically write about architecture, his works are the most ancient which have survived to this day. In addition, there exists a vast number of books detailing specific architectures, the history of the discipline, and theories explaining particular aspects of the discipline (for instance [13-15]). Whilst not all architects will agree on what the most appropriate solution may be for a particular problem's requirements or even on the best architectural design theory, the discipline itself has a common understanding of what it means to be an architect and what the goal of architectural design is - "That is what architects are, conceivers of buildings. What they do is to design, that is, supply concrete images for a new structure so that it can be put up. The primary task for the architect, then as now, is to communicate what proposed buildings should be and look like." [16]. This common understanding has coalesced over a long period of time through the publication of architectural theories and education of architects in apprenticeships, guilds, schools, and universities.

Second, there is another obvious aspect of building architecture which has facilitated the development of a common understanding. You can 'see' the results of architectural design - an architecture is a tangible thing. By looking at the architectural design of a building and the building itself, it is possible to 'see' the direct mapping between the two. Buildings, bridges, and even computer hardware have physical manifestations which can be visually observed. We note, however, that it is interesting to observe the extent to which the architectural 'visibility' of electronics systems blurs due to the increase in VLSI technology. Regardless, if we examine a building's architecture, we can identify the following characteristics:

1. At the lowest level there exists the materials from which the building is engineered. For example: bricks, steel, concrete, and wood.

2. Higher level concepts refer to structural arrangements of those materials. For example: column, window, room, staircase, house, courtyard, and town. These are not tangible in the same sense as our building materials, they are labels used to represent patterns of their structural juxtaposition. However, it is possible to visually depict the physical form of these large scale patterns. And it is possible to define them in terms of preagreed physical attributes of the building materials used to form the aggregation. These juxtapositions form both components and aggregations of components, with some of the resulting arrangements exhibiting recognisable 'styles': eg., Gothic, Vitruvius's Orders, etc.

All of this is combined by the means by which we map between the physical building materials from which larger structures are engineered and the concepts in our minds which are the abstractions we use to refer to particular structural arrangements. Different theories of epistemology explain how these mappings occur in different fashions. The goal of this paper is not to support or refute any particular theory of knowledge. We need merely to note the existence of the mapping between sensory experience of the world, and the larger granularity concepts/abstractions we use to refer to particular arrangements in that world.

2.2. The Difference between traditional Architecture and Software Systems.

Looking at software development in the same manner as we did with traditional architecture reveals the existence of similar aspects:

1. At the lowest level we also have the building materials which are used to construct our systems. These consist of the programming language constructs and the operating environment within which they execute. Software systems execute within a particular virtual environment and any attempt to analyse how the source code operates can only be made with reference to it. Here we are referring to the virtual machine which exists at the language representation level. For example, the execution of individual statements in a procedural programming language, such as C, has to be evaluated with respect to a 'single, sequential, thread of control' operating environment. Similarly, the evaluation of rule-based programming constructs, such as those found in Prolog, must be evaluated within the overall environment of a backward-chaining inference engine.

2. Above the level of building materials we also have larger abstract concepts which represent particular building components and their structural arrangements. These are often spoken about in descriptions of the system and used as labels on the boxes and lines which appear in graphical depictions of the system's design. Concepts such as queues, algorithm, message passing, and layers, are all terms used by software engineers in a similar fashion to concepts identified by architects and other engineers. This level of abstraction can be divided into larger scale components and also styles which relate to their structural arrangement.

Finally, there also exists the mapping process which is analogous to the one identified for architects. Software developers construct a conceptual model of the system to be implemented. This model consists of the structural arrangement of high-level, abstract concepts. The development process somehow realises those concepts and their relationships in terms of the software building materials. Conversely, the process of reverse-engineering involves the reverse mapping process where the larger level concepts, relevant to its operation, are somehow abstracted from the programming language implementation.

Comparing the two disciplines using a common framework reveals a number of differences not otherwise visible. The architecture of implemented software systems simply does not exist in the same manner as traditional built systems. The abstract concepts of building architecture can be 'seen' in the physical realisation of the system. It is possible to look at a building and identify the rooms, windows, archways, etc, eventhough each is merely an pattern of a particular arrangement of building materials.

At a larger level of granularity, it is possible to recognise building styles. Even somebody not formally trained in architecture could recognise Notre-Dame cathedral as an example of a building in the Gothic style. The concept is part of common parlance. In contrast, the only aspect of a software system which can be perceived through the senses is the source code and its arrangement into the program modules and interconnections, files, directories, libraries, etc of its virtual operating environment. The architecture of the software systems has to be generated from the collection of source code, and knowledge of how the code is executed by its virtual machine, through the process of abstraction. The ability to read the source code of our systems and attempt to understand its architecture is comparable to reading a description of a building rather than viewing the building itself, in an attempt to achieve the same end. This is a fundamental difference between our disciplines and cannot be ignored when making analogies between them.

We are working in a discipline that has a physical representation of the concepts we use to design which is fundamentally different to traditional engineering disciplines. Because we lack a direct perceptual mapping between high level concepts and physical implementations without resorting to abstraction, it is harder to ground our ideas and provide a means by which we could reach a common understanding. This, we postulate, makes the process for forming relationships between our ideal concepts and instantiated examples very difficult.

In summary, initial analysis suggests that the discipline of software development has two differences between it and other engineering disciplines.

1. Software systems do not have a physical representation which can be perceived in the same way as other engineered artefacts.

2. Software systems have a distinction between the form of the implemented system, the collection of source code, and the form of the executing system, the way the source code is executed by the environment to realise the required system. This distinction does not exist to the same extent in other engineering disciplines.

The existence of these differences serves to undermine our ability to draw valid analogies with traditional engineering disciplines. What is required is a better understanding of the nature of software development before determining whether those analogies and borrowed terminology remain valid. This will be presented in the next section. However, before that is done, we need to address the concept of software 'architectural views' and determine their relationship to the previous analysis.

Perry and Wolf [4] also compared the concepts of software architecture with traditional engineering disciplines, including the building industry. One of their observations was that "the software architect needs a number of views of *the* software architecture for the various uses and users" (emphasis added) and that "at present (1992) we make do with only one view: the implementation". They also make the observation that "it is very difficult to abstract the design and architecture of the system from all the details". We have made the point that the difference between traditional engineering and software development is that you do not need to "abstract the design and architecture" from buildings for example, because their is a direct mapping between the form of the building which is perceived by the senses (ie., seen), and the drawn representation of the building's architecture.

Following Perry & Wolf's observation, a number of researchers have presented a collection of views required to depict the software architecture of a system. For example, Kruchten [17] describes the architecture as a complex structure which can be represented by four different views.

- **Logical view:** A decomposition into a set of key abstractions, taken from the problem domain.
- **Process view:** How the main functional abstractions map onto executing processes and threads of control.
- **Physical view:** Reflects distributed aspect by showing how the software maps onto the hardware.
- **Development view:** The actual software module organisation in the development environment.

In addition, Soni [18], following a survey of many industrial software systems, identified four different large scale structural depictions used throughout the development process. And Kazman [19], while discussing the analysis of quality attributes of system architecture, has asserted that it can be described from (at least) three different perspectives.

Clearly these collections of views do capture many useful aspects of large scale software systems. Moreover, analysis has shown that they cover the same aspects of a software system's design using different categories. However, they assume there exists a single software architecture which exists in an analogous manner to traditional notions of architecture and that these views represent different subsets of the overall properties.

These software architecture views are compared to views of traditional architecture such as scale models, floor plans, elevation views, contextual drawings, and views to allow the viewer to determine the allocation of heating, plumbing, wiring, and natural lighting. These views of traditional architecture are all very important. However, there is only *one* building and *one* architecture for that building. It has a defined physical form, and these views depict a subset of the measurable properties of that form. For instance, the scale model depicts only enough information to show the relative sizes of the building components. Similarly, the view required to depict the effects of natural lighting represent enough properties of the proposed building to allow the designer to determine how the light illuminates the internals of the building as the sun crosses the sky. These 'views' of building architecture all depict a subset of the measurable properties of *the* single building architecture.

Traditional uses of 'architecture' utilise views to "provide an immense amount of detail about various explicit design considerations" [4]. Software developers have also identified a number of useful views which provide an immense amount of detail about various explicit design considerations. However, this does not prove the assumption that we have a single architecture which is analogous to traditional engineering systems since the differences between the disciplines have not been accounted for. To address these differences and determine the validity of the stated assumption we need to look more closely at what we, as software developers, actually do.

3. What Is Software Development?

Our purpose in this section is to show the concepts of software architecture within a single framework. This will allow us to determine why the identified differences between 'software architecture' and traditional architecture exist and whether or not the underlying assumption about the existence of a single software architecture, which is analogous to those of traditional engineering is valid.

We begin with the conjecture, developed by the first author [20] after compared software and hardware designs of automotive cruise control systems - that software developers construct executable models or theories which satisfy the solution requirements and implementation constraints of a particular problem. In contrast, traditional engineers construct tangible artefacts whose constituent materials exhibit physical properties which are combined to satisfy the solution requirements and implementation constraints to meet some human needs.

The software designs were those used by Shaw [21] in her comparison of different architectural styles, while the hardware designs were chosen from the traditional automotive engineering literature. This made an explicit comparison between the two approaches possible. Such comparisons are virtually non-existent in software engineering research.

The discipline of software development designs and implements systems for an incredibly diverse and ever expanding set of problem domains that. Therefore, it would be foolish to assume that such a sweeping characterisation as "engineers build artefacts and software developers build models" could possibly encompass all software development. The authors recognise that there are certainly instances in which software development produces artefacts in an analogous manner to engineering development. However the construction of models appears to be a general property of software development as it is currently practiced, and the source of much of the difficulties we encounter. This model-building theory has also been suggested by other researchers, for example [22-24].

3.1. The Conceptual Model.

The first step in the design process is the generation of a conceptual model or theory of how the solution should operate. This is often referred to as the conceptual or logical architecture of the system and outlines the structural concepts and relationships required in the implementation. The many stages of design serve to transform the concepts and relationships of the conceptual model into the source code of the implemented system. This code is subsequently executed by the machine. This process of executing the code statements of the implementation, results in the execution of the explanatory theory which was devised to solve the problem at hand.

There are many issues involved in the generation of the conceptual model from the initial problem requirements, its transformation to source code implementation, and its operation as an explanatory theory for the original problem. This paper cannot possibly elucidate all of those aspects, however we will present enough detail to meet the original goal, that is, to show why it is so hard to define software architecture.

Theories and models are collections of concepts connected by relationships that allow causal interaction to occur. In the fields of science and natural philosophy these models are used to explain observable phenomena. In software development the concepts and their interactions provide the abstractions (the theory or model) we believe explains the phenomena we wish to realise as a means of solving a particular problem (the system). Consequently, the discipline of software development can gain insights into development approaches by analysing what cognitive psychologists have learnt about conceptual development and what philosophers, especially philosophers of science have learnt about the nature and limitations of theory development.

Let us explain this briefly. Observations of the world lead to the development of concepts, categories, and explanatory theories, all of which have implications for how we perform the first step of software design - the development of the conceptual model. The software developer does not start with a clean piece of paper and unconstrained imagination when developing the conceptual model. Philosophy and cognitive psychology describe the influences that culture, languages, psychological biases have on perception, and expertise in particular domains has on the development of our concepts and theories. Despite this, we still have a degree of creativity with which to select the concepts we need to form our models. However, the concepts we choose to use form a particular model of reality and must be logically consistent with each other. Moreover, there may be more than one logically consistent collection of concepts which constitute a useful model of the same reality. Therefore, not only can we entertain alterative models and theories depending on our different aims and purposes, but we can also entertain multiple (and not fully consistent) models and theories in the same context [25]. One set of concepts is no closer to a 'true' reality than any other, it is only more useful than another in so far as it yields conceptual interpretations which better suit the needs of the situation.

Examples of the generation of multiple conceptual models representing the same problem can be found in software development. Mary Shaw's paper [21], compared eleven previously published software architecture designs for an automobile cruise control system, categorising them into the following different types: object orientated, control/state based, process control feedback loops, and traditional structured design (with appropriate real-time modelling extensions). Moreover, even the designs which employed the same architectural style, eg., object-oriented, consisted of collections of different concepts.

McAuley research in conceptual development [25] points out that this general situation results in two types of issues which need to be considered. Firstly, we need to consider the internal relationships. Those which exist between the constituent concepts of our cognitive models. The second variety, external relationships, also have two varieties. The first deals with the relative fit with one another of two or more cognitive models for the same problem. Such models inevitably conflict somewhere. The second sort of fit is between our cognitive model and the world. Does it allow us to explain the 'reality' under observation? We have no guarantee that our "idealised cognitive models and theories cut the world at its joints" [25]. However our various cognitive models offer alternative descriptions of the world and it is possible to recognise from time to time that certain descriptions are not only less helpful than others, but also that some are, for all intents and purposes, false. Determining the effectiveness of proposed models is a central issue in the philosophy of science. For example, Popper's schemata [26] of theory development.

Other issues which affect our ability to generate useful conceptual models or initial architectures include our ability to generate more abstract representations of our theories to cover a wider range of applicability [27]. This issue affects what we currently call styles and software patterns yet it has a much longer history of research in philosophy than it does in books by Christopher Alexander. In addition, the means by which we develop our hierarchy of categories [28, 29] and the affect domain expertise has on what concepts we depict in our conceptual models are also the subject of research in other disciplines.

Following the development of the conceptual model, the succeeding stages of development involve a series of steps whose ultimate goal is the implementation of that conceptual model using the chosen implementation medium.

3.2. Implementing The Conceptual Model.

The implemented software system has two attributes of interest to us here. First, there is the implementation which is created using source code and, second, there is the way in which that source code operates to realise the different aspects of the concepts and relations of the conceptual model.

The implementation is constructed using the programming constructs of the chosen languages and calls to support environments such as GUI packages, databases, other run-time libraries, and operating system services. As part of this realisation process, the developer always remains aware of, and is influenced by, the way in which that source code will operate. Run-time constructs can combine with the source code in many ways to provide the implementation. These include aspects such as separate and communicating processes, threads of control, distributed machines, communication networks, and virtual machines. Whilst a distinction has been made here between the implementation constructs of the source code and the run-time constructs of the source code's environment, the delineation is quite blurred and many aspects can reside in either of these arbitrarily designated categories.

This paper does not attempt an explanatory model of the process which transforms the concepts and relations contained in our conceptual models to the concepts and relation types which are provided by our implementation and run-time constructs - assuming such a complex, cognitive process is, in fact, explainable. However, a number of issues are worth highlighting and have been examined by the authors. They are: the utilisation of design methods, the affect of the implementation medium on our ability to generate conceptual models, the nature of modelling itself, and the use of multiple design models. Readers will note that we are specifically focussing on the philosophical and cognitive aspects rather than the more traditional process-model centred argument.

In summary, the software development process begins with a conceptual model of the solution, which the developer believes will solve the problem. This model, its creation, development and evaluation, is subject to the processes and activities which have been detailed by researchers in traditional fields of theory development. The concepts and relationships chosen to comprise this conceptual model can be completely arbitrary. The designer may be influenced by issues such as education, known design methods and knowledge of programming languages, however, as long as the concepts form a coherent and logically consistent system, and the model serves as a useful theory for solving the problem, the types of concepts and their relationships can be of any type. However, to implement the model, the designer must utilise the module and interconnection mechanisms provided by the operating environments or virtual machine(s). This implementation stage is not a mere decomposition, it requires a cognitive leap from one collection of concepts to the other. As a result, there is no simple mapping between the domain of concepts used by the designer to represent the conceptual model and the domain of concepts provided by the implementation environment to realise the system. This is why there can be no single architecture which is analogous to traditional engineering systems. Obviously it would be beneficial to reduce the distance of this 'cognitive leap' by providing an implementation domain which consisted of concepts which were closely matched the domain of concepts used of to think about the world. In fact, this has been the claim of object-oriented design languages and methods. However, while these approaches may have reduced the gap somewhat, there is still some degree of cognitive distance between the domain of the conceptual model and that of the implementation environment. The only way we can cross these gaps is through mental processes, such as abstraction, which are a part of the mental faculties of all of us. The problem is, however, that they are subjective to the person performing the transformation which results in different results for different developers.

4. Conclusion: What does this mean for Software Architecture?

The use of the term architecture to deal with the higher level abstractions used in software development is based on the implicit assumption that we are sufficiently analogous to other "design and construct" disciplines (usually engineering based) that the semantics associated with the terminology will hold across disciplines. Objective comparisons with other disciplines, in particular those engineering disciplines to which we aspire, is essential. Focussing on the differences reveals that our discipline has a number of characteristics which makes it difficult to apply the term architecture as it is used in common parlance.

1. Software systems have no tangible representation which allows us to directly perceive the realisation of the large scale abstractions which were used in the design. This makes it difficult to identify them unambiguously and communicate them to others.

2. The implementation of the software system consists of two distinct parts. The static source code implementation and the dynamic system execution. Most traditional engineering disciplines have no such distinction between the form and execution of their systems.

Convergence of a common view of software architecture requires the creation of a view of software development in general which incorporates these characteristics. In the process we have highlighted differences between the software concept of architecture and that of other disciplines. This theory is based on the conjecture that software developers are currently, when contrasted with builders of traditional systems, model or theory builders rather than the creators of traditional engineering artefacts.

From this it follows that there are valid reasons for not having a single 'architecture'. There exists three areas within this model building process where high-level representations of the system are considered in an analogous manner to traditional architecture design. However the representations depicted in these three areas are not merely views of the one architecture. Rather, they are fundamentally different, yet all can be labelled as the system architecture with equal validity. This was exemplified in a recent case-study performed by the authors on one of their own systems [30]. The case-study followed the evolution of the HyperEdit project over a number of years from system conception, through initial implementation, and finally, major maintenance. It is not possible in the space available to detail all the design reasoning behind these architectural representations, however it is possible to show the actual architectures which were used by developers and maintainers at succeeding stages in the systems development and the differences between them.

The conceptual model of the system is shown in Figure 1. It represents the high-level abstractions and their interconnections (the model) which the developers conceived to solve the problem. Once the design was implemented in code, the system architecture was represented in terms of a partitioning of modules provided by the programming environment. This was as a layered architecture which depicted clear interfaces between the HyperEdit engine routines, the communication layer, and the database server routines. Moreover, it was possible to identify these 'layers' in the source code interfaces. In addition to the layered representation, the developers also used a call-graph representation between specific procedures in the code. Finally, to facilitate the maintenance process, a representation was required which highlighted the dynamic operation of the system. The system was implemented in the X/Motif environment which provides its own event-based control mechanism. This event-based mechanism is not explicitly evident in the source code calls, it is part of the virtual machine of the programming language which executes the code.

Consequently, to understand how the system operates, this global control mechanism, which is not evident in the layered architecture, needed to be made explicit in an another architectural representation (Figure 2).

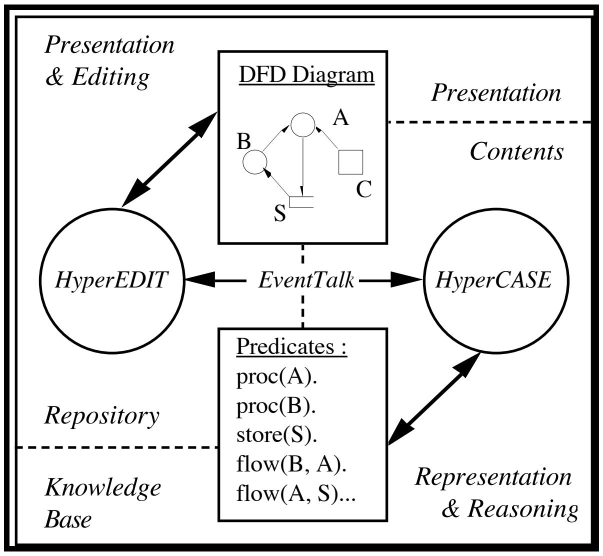

Figure 1: Conceptual Architecture

In conclusion, we believe that much of the current research in software architecture relies on the implicit assumption that software development is analogous to traditional building disciplines which exhibit a single, identifiable level of abstraction in their design process which is called the architectural level of design. We have examined this analogy and compared software development with these traditional disciplines in an attempt to highlight differences between them rather than just the similarities. A number of fundamental differences between the disciplines were identified which were subsequently explained by representing the process of software development as a model building discipline. This theory highlights why we have so much difficulty in developing a common understanding of what the proposed "software architecture level of design" is or should be. The theory suggests there can be no single software architecture which is analogous to those found in traditional building disciplines. Rather, as the development building process proceeds, three types of high-level representation of the design/system are required. Each of these consists of high-level abstractions and represent a view of the system. However they are not different views of a single architecture. The are fundamentally different and all can claim to be *the architecture* based on our current generic definitions. They are:

1. The Conceptual Model Representation: Produced as the initial step of the design process and represents the model which is to be implemented as a solution to the particular problem. It consists of the concepts and relations which constitute the designer's conceptual model. Those constructs may be similar to those provided by the implementation medium.

2. The Static Implementation Representation: Depicts the source code implementation of the system and their dependencies. It represents the structural form of the implemented system but does not contain enough explicit information to depict the control flow through the executing system.

3. The Dynamic Operational Representation(s): These depict how the system operates and are a hybrid of the previous two implementations. It contains information which details the operation of the statically implemented system, yet is at a higher level of abstraction than the code.

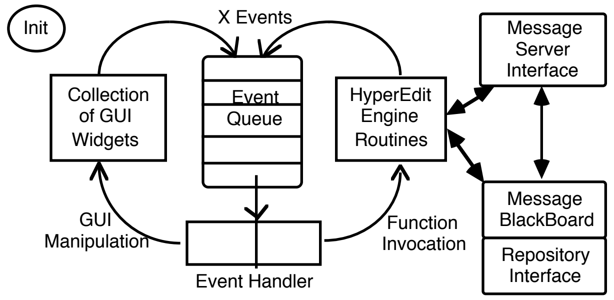

Figure 2: Event-Based Operational Architecture

5. References.

1. Dijkstra, E.W., *The Structure of the "THE" - Multiprogramming System*. Communications of the ACM, 1968. **11**(5): p. 341-346.

2. Spooner, C.R., *A Software Architecture for the 70's: Part I - The General Approach*. Software - Practice and Experience, 1971. **1**(Jan-March): p. 5-37.

3. Shaw, M., *Large Scale Systems Require Higher-Level Abstraction*. Proceedings of Fifth International Workshop on Software Specification and Design, IEEE Computer Society., 1989. : p. 143-146.

4. Perry, D.E. and A.L. Wolfe, *Foundations for the Study of Software Architecture*. ACM SigSoft, 1992. **17**(4).

5. Garlan, D. and M. Shaw, *An Introduction to Software Architecture,* in *Advances in Software Engineering and Knowledge Engineering,* V. Ambriola, Editor. 1993, World Scientific:

6. SEI. *Software Architecture Definitions*. <http://www.sei.cmu.edu/architecture/definitions.html>. (Accessed September 1998).

7. Clements, P.C., *Software Architecture: An Executive Overview*. 1996, Software Engineering Institute:

8. *Miriam-Webster Dictionary: http://www.m-w.com/netdict.htm*. 1997, .

9. Pevsner, N., J. Fleming, and H. Honour, *A Dictionary of Architecture*. 1975, .

10. Standen, D., *Terms in Practice: a dictionary for Australian arthitects*. 1981, Royal Austrlian Institute of Architects.

11. Curl, J.S., *Encyclopedia of Architectural Terms*. 1993, .

12. Vitruvius, P., *Vitruvius, On Architecture*. 1931, .

13. Gelernter, M., *Sources of Architectural Form: a critical history of Western design theory*. 1995, Manchester University Press.

14. Kruft, H.-W., *A History of Archiectural Theory: from Vitruvius to the present*. 1994, Zwemmer.

15. Watson, D., *Rule-Generated Architecture*. Course Notes of the Advanced Design Processes Course offered by the School of Architecture at Deakin university, 1990, Geelong, Australia: Deakin University Press.

16. Kostof, S., *The Architect: chapters in the history of the profession*. 1986, Oxford University Press.

17. Kruchten, P., *Architectural Blueprints - The "4+1" View Model of Software Architecture*. IEEE Software, 1995. (November).

18. Soni, D., R.L. Nord, and C. Hofmeister. *Software Architecture in Industrial Applications*. in *ICSE '95*. 1995. Seattle, Washington.:

19. Kazman, R., *et al. SAAM: A Method for Analyzing the Properties of Software Architectures*. in *ICSE*. 1994. Sorrento, Italy: IEEE Computer Society Press.

20. Baragry, J. *An Initial Comparison of Software and Engineering Designs of Automotive Cruise Control Systems*. in *Australian Software Engineering Conference*. 1996. Melbourne, Australia: IEEE Computer Society Press.

21. Shaw, M., *Making Choices: A Comparison of Styles for Software Architecture*. IEEE Software: Special Issue on Software Architecure, 1995. **12**(6).

22. Blum, B.I., *Beyond Programming: To A New Era Of Design*. 1996, Oxford University Press. 423.

23. Lehman, M.M., *Programs, Life Cycles, and Laws of Software Evolution*. Proceedings of the IEEE, 1980. **68**(9): p. 1060 - 1076.

24. Naur, P., *Programming as Theory Building*. Microprocessing and Microprogramming, 1985. **15**(5 (May)): p. 253-261.

25. McAuley, R.N., *The role of theories in a theory of concepts,* in *Concepts and Conceptual Development,* U. Neisser, Editor. 1987, Cambridge University Press: p. 288-309.

26. Popper, K.R., *Of Clouds and Clocks,* in *Objective Knowledge: an evolutionary approach*. 1979, Oxford University Press:

27. Lee, H.N., *Percepts, Concepts and Theoretical Knowledge*. 1973, Memphis State University Press.

28. Neisser, U., *The ecological and itellectual bases of categorization,* in *Concepts and Conceptual Development,* U. Neisser, Editor. 1987, Cambridge University Press: p. 1-11.

29. Neisser, U., *From Direct Perception to Conceptual Structure,* in *Concepts and Conceptual Development,* U. Neisser, Editor. 1987, Cambridge University Press: p. 11-23.

30. Baragry, J. and K. Reed, *HyperEdit: A Case Study in Software Architecture*. 1998, La Trobe University: