Why We Need A Different View of Software Architecture

Jason Baragry
Norwegian Computing Center.
P.O. Box 114 Blindern,
N-0314 Oslo.
Norway.
Email: Jason.Baragry@nr.no.

Karl Reed
Dept. of Computer Science.
La Trobe University.
Bundoora. Vic. 3083
Australia
Email: kreed@cs.latrobe.edu.au.

Abstract

The definition and understanding of software architectures and architecture views still shows considerable disagreement in the software engineering community. This paper argues that the problems we face exist because our understanding is based on specious analogies with traditionally engineered artefacts. A review of the history of ideas shows the evolution of this understanding. A detailed examination is then presented of the differences that exist between the nature of the systems, the content of their large-scale representations, and how they are used in practice in the respective disciplines. These differences seriously undermine the analogies used to develop our understanding and this is discussed in terms of software engineering as a whole.

1 Introduction

Software engineers have been discussing the architecture of their systems since at least the late 60s and software architecture research has been a separate field of study since the late 80s. However, while the ideas in this sub-discipline are still solidifying, confusion exists concerning the exact nature and meaning of software architecture and that confusion is restricting the progress of software architecture research and the adoption of its ideas in practice.

For example, Mobray [1] notes the importance of architecture research ideas but states they are hard to put into practice because of confused terminology, the lack of complete models, and disagreement about which views of the system are necessary. One reason for those differences is the lack of a universally agreed definition or even understanding of what software architecture is or should be. Similarly, Bennett [2] notes that the research community is almost unanimous in its conviction that software architecture describes something about the structure of a system and that it plays a vital role in determining the systems emergent properties. However, they are much less unanimous on the questions of which elements should be included in the architecture, how to co-ordinate different collections of those elements (views), and how to evaluate the architecture against the external requirements. The problem is not that there are no answers to these problems; rather, the difficulty arises from the fact that there have been so many different answers given.

Research efforts are attempting to solve these difficulties between software architecture theory and practice. However, despite the realisation that researchers need to do something to solve the discrepancies, we believe the problem is far more fundamental than currently understood. What is required is an examination of how we understand software systems, their development, and the large-scale structures used to represent them.

One thing that is obvious from the review of the literature is that the community's understanding of software architecture has evolved based on analogies with the large-scale structure of traditionally engineered systems. Take for example the philosophy of the self-proclaimed 'World-wide Institute of Software Architects':

> "There is a compelling analogy between building and software construction. It is not new, but it has never taken root and bloomed. The analogy is not just convenient or superficial. It is truly profound. It not only raises the right questions, it has the answer to what has been called 'The Software Crisis.'" [3]

Far from being a potential panacea, we believe this understanding is in fact the source of many of the problems in software architecture research. This paper argues that the analogy is indeed convenient, superficial and far from "truly profound". Moreover, the problems in software architecture exist because our understanding of the issues is based on these specious analogies with traditionally engineered systems and that, far from

0-7695-1360-3/01 $10.00 © 2001 IEEE

providing "an answer to what has been called 'The Software Crisis'", the differences between theory and practice will not be solved until the software engineering community develops a different view of software architecture.

The paper begins by presenting the current understanding of the terms 'architecture' and 'architecture views'. The historical development of the research community's understanding of these terms highlights their derivation from analogies with more traditional engineering disciplines. While those analogies served a useful purpose in facilitating our formulation of software development concepts, their failure to adequately consider the differences between software development and those other disciplines requires them to now be replaced. Those differences are detailed to show the limitations of our present understanding by using the specific area of 'architecture views' as the example. Those differences are grouped into three categories:

1. Differences between software and traditionally engineered systems.
2. Differences between the content of architecture views in the respective disciplines.
3. Differences between how those views are used in the development processes of the respective disciplines.

Finally, we make concluding comments about how this issue relates to the community's guiding assumptions about software engineering in general and the role of research in the progression of the discipline.

2 A History of Our Understanding of Software Architecture.

The first papers to describe the large-scale structure of software systems appeared in the mid to late 1960s. For example, in 1968 Dijkstra detailed the large-scale structure of the 'THE-Multiprogramming System' [4] where he discussed the advantages of partitioning the operating system into layers like 'onion-rings'. Another example exists in the transcripts of the 1969 NATO conference on Software Engineering where Sharp discussed the importance of software architecture and the differences between design at that level of detail and other software engineering. [5] (p. 150). Later, Spooner developed his "Software Architecture for the 1970s" [6], contrasting it with Dijkstra's large-scale system structure. As the 70s progressed, practitioners began detailing the advantages of theorising about those system-level structures and the consequences of decisions made at those higher levels of design (e.g., [7]). In addition, Brooks wrote his essays on software engineering [8] in which chapter four, *Aristocracy, Democracy, and System Design*, stressed the importance of the conceptual design phase and how it affects subsequent development. These examples show software developers were able to identify and reason about high-level structures of their software systems and recognised the importance of decisions made at that level of design. Moreover, it shows that the term 'architecture' was well established as the word for designating those structures.

Brooks, who was the originator of many software architecture ideas, also published articles on the architecture of computer hardware [9]. Given this, and the extent to which Brooks draws on analogies with hardware development paradigms in *The Mythical Man-Month* [8], it could be argued that many of the concepts Brooks used for understanding the large-scale partitioning of software systems evolved from his understanding of the concepts involved in computer architecture. This influence was considered to some extent. In the early 1960s, Brooks and Weinberg discussed the appropriateness of the term 'architecture' for describing structural design issues in computer systems. Brooks was worried about the appropriateness of the analogy, however as their discussion progressed it seemed to hold [10]. At that time, their discussion considered computer systems as both hardware and software, in contrast to the more software-centric analogies used in recent times [11]. In addition, their concept of software architecture included the interface with the computer operator as well as the large-scale system structure [11]. That aspect is also evident in Brooks' later comments on the integrity of the system architecture.

> "By architecture of a system, I mean the complete and detailed specification of the user interface." [8]

Coplien notes therefore, that as early as 1965 the discipline of software development was already enough on its feet to consider the influence of design theories in other artefact construction disciplines [12].

Despite these, and many other examples of software developers reasoning about the large-scale structures of their systems, it was Mary Shaw's 1989 paper, *Larger Scale Systems Require Higher Level Abstractions* [13] that was significant in the emergence of the area of research that is today referred to as 'software architecture'. In that paper, Shaw recognised the existence of high-level system representations that are used during the development process and which could be recorded and passed onto other designers. Shaw had been working on abstraction techniques previously [14] and noted the use of those abstractions in the development process could result in a "software architecture level of design." Shaw's work identified and labelled a number of different styles of architecture that are still used as

examples today. For example, 'layered' and 'pipe & filter'. While Shaw's paper discussed the importance of higher-level system abstractions, it merely identified the concepts that others began to theorise about.

Perry and Wolf's paper [15], as its title suggests, laid the foundations for many architecture research ideas. It also contained the first attempt to define architecture, or at least, the important concepts of software architecture. They stated that a model of architecture consists of three components: elements, form, and rationale. The elements are either processing, data, or connecting elements; form is defined in terms of properties and relationships among the elements (the constraints); and rationale provides the underlying basis for the architecture in terms of system constraints. Much of the understanding in that paper was derived through analogies with other disciplines that highlighted useful similarities and differences. For example, computer hardware, network architecture, and traditional building architecture. One of those analogies compared the different representations of a software system with the multiple views of a traditional building design that are used by the various stakeholders in the development process. That specific analogy is discussed in detail in a later section.

From those research foundations, many definitions of software architecture have emerged. Of the early definitions, the one by Garlan and Shaw [16] was often cited. However, neither this, nor any other definition, has become an accepted standard. The Software Engineering Institute web site houses many of the definitions that have been published in software architecture literature [17].

The most recent definitions differ from the earlier ones by catering for issues that emerged out of published experience reports – the existence of multiple views of software architecture. A number of software architecture case studies and theories based on practical experience were published suggesting the need for multiple large-scale representations to capture the architecture of a software system. For example, Soni [18], as a result of surveying many software systems used in industrial applications, identified four different large-scale structural depictions used throughout the development process. Kazman [19], while discussing the analysis of quality attributes of system architecture, asserted that the architecture could be described from (at least) three different perspectives. Finally, Kruchten presented his collection of system representations that had been successfully used to capture the architecture information in several large projects [20]:

- Logical view: Where the required system is decomposed into a set of key abstractions, taken (mostly) from the problem domain.
- Process view: Depicts how the main, functional abstractions map onto executing processes and threads of control.
- Physical view: Reflects distributed aspects by showing how the software maps onto the hardware.
- Development view: Focuses on the actual software module organisation in the development environment.

Those four views are depicted with a fifth view that illustrates them with a few use-cases or scenarios. Indeed these views are considered analogous to the depiction of software architectures in the increasingly popular Unified Software Development Process [21] (p. 62).

From those experience reports, the use of multiple views to represent the system architecture has become accepted in the discipline and has become part of more recent definitions of software architecture. For example, Bass et al state:

> "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the external visible properties of those components, and the relationships among them." [22]

The intent of the definition is that "a software architecture must abstract away some information from the system ... and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction.". The authors also note that "the definition makes clear that systems can comprise more than one structure, and that no one structure holds the irrefutable claim to being the system architecture."

It is now clear that when developing software systems many large-scale system depictions are used. The prevailing consensus in software architecture research is that these representations are different views of the system architecture, where each view provides a different abstraction of the underlying implementation detail. Therefore, each view is a subset of the detail that exists in the implementation. This way of understanding the nature of software architecture views can be traced back to the 'foundations' paper by Perry and Wolf [15]. From their analogies with traditional building architecture they noted:

> "... a building architect works with the customer by means of a number of different views in which some particular aspect of the building is emphasized. ... For the builder, the architect provides the ... floor plans plus additional structural views that provide an immense amount of detail about various explicit design considerations such as electrical wiring, plumbing, heating, and air-conditioning. ... Analogously, the software architect needs a

number of different views of the software architecture for the various uses and users." [15]

The same analogy was used by Bass et al to explain their definition of architecture. They claim the multiple representations are analogous to the different building representations used by the architect, the interior decorator, the landscaper, and the electrician. They summarise the most useful representations or views used by software developers as: module structure, conceptual or logical structure, process structure or co-ordination structure, physical structure, uses structure, calls structure, data flow, control flow, and class structure. [22]

Despite the many definitions, confusion still exists concerning the exact nature of the representations, why they are necessary, and which ones should or should not be included in the description of the system architecture. Other researchers have offered explanations for this.

Clements, in his overview of the field [23], suggests five reasons why the community has failed to reach a consensus on what exactly we mean 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.
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 commonalties.
3. The field is still quite new.
4. The foundations have been imprecise. The field contains a remarkable number of undefined and ambiguous terms.
5. The term is over-utilised and its meaning as it relates to software engineering is becoming diluted.

That confusion concerning the meaning of software architecture is also observed by Bass et al [22]. However, they suggest the lack of a well-accepted definition is not as troubling as it appears because the concept of software architecture can still be successfully used while a discipline-wide consensus evolves. [22]

To summarise the current understanding of software architecture:

- Software developers have been able to identify and theorise about the large-scale structures of software systems since early in the discipline.
- Those large-scale structures are considered the 'architecture' of the software system. That understanding is based on analogies with traditional engineering disciplines whose built systems exhibit large-scale structures that are termed the 'architecture'.
- Research has successfully sought to improve the development process at the software architecture level of design.
- Experience suggests many system representations are required to depict the architecture of a software system.
- Those representations are considered analogous to the multiple representations of traditionally built artefacts.
- Confusion still exists about the exact nature of software architecture and the views used to represent it.

3 Issues that Undermine the Current Understanding of Software Architecture.

The logical progression from the recognition of large-scale structures in software systems; to Shaw's call for an architecture level of design; through to Perry and Wolf's foundations for the discipline; and finally to the explanation of the multiple, high-level representations required to depict a software system as different views of the implementation detail appears valid. However, a more thorough comparison of the systems built by the respective disciplines shows it is quite specious. It 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'. The problem of obtaining an acceptable definition of software architecture or a set of common architecture views is due to the assumption that software systems have an analogous, unique design abstraction, determinable at the early stages of the design. That understanding of architecture and the use of architecture views follows from Perry and Wolf's statement,

"... there are a number of interesting architectural points in building architecture that are suggestive for software architecture."

However it ignores the statement that began that sentence,

"While the subject matter of the two is quite different ..." [15].

The subject matter of the two is quite different and any attempt to use analogies between the disciplines can only be done by ensuring that conjectures extrapolated from those analogies are not invalidated by those differences. This section examines those differences and finds 3 categories where the analogy fails to hold. They are:

1. Differences between software and traditionally engineered systems.

2. Differences between the content of architecture 'views' in the respective disciplines.
3. Differences between how those views are used in the development processes of the respective disciplines.

3.1 Differences between Systems.

System Form. A comparison of the disciplines shows that two important differences exist between the artefacts produce by software developers and those produced by the more established engineering disciplines. The first is the concept of form and the other is the concept of system execution. Those differences between the fundamental natures of the respective systems have a significant impact on the way we use the notions of architecture and architecture views in the development process.

Systems produced by traditional engineering disciplines are corporeal. They have a physical form, a tangibility that allows the viewer to perceive its large-scale structure – its architecture. That architecture can be viewed in the original design documents, traced throughout the design process and viewed in the physical realisation of the system. Obviously you cannot see all of the details of the architectural design by looking at the physical system. For instance, the precise nature of the materials used, the exact physical dimensions of components, and hidden areas such as ventilation shafts may all be indeterminable. However, the large-scale structure of the system is evident in the design and in the finished artefact.

While not all architects agree on the most appropriate solution for a particular problem's requirements or even on the best architectural design theory, the discipline does have 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." [24].

Architects represent the geometric properties of the building materials and/or components. The physical magnitudes and relations of those components and how they are juxtaposed in space. That is the case in traditional architecture, civil engineering, and mechanical engineering. Those architectures depict the physical form of the system or the components that comprise the system. System 'functionality' is then inferred from those components[1].

Australia's most famous piece of architecture, the Sydney Opera House, provides a good example. Figure 1 depicts the large-scale system design developed by the architect. It also depicts a picture of its physical appearance [25]. Put simply, you can see the architecture in the design and in the realisation.

The analogous concept of form does not exist for software systems. In general parlance, the architecture of a physical artefact describes its "unifying or coherent form or structure" [26]. That generic concept is easy to understand when dealing with our vast range of physical artefacts. People without specific training in the respective fields can perceive building architecture, computer architecture, naval architecture, etc. However, difficulties arise when you apply the same concept to elicit the architecture of a system whose only tangible manifestation of the construction is the source code implementation [2].

You cannot see the architecture of a software system by looking at the thousands of lines of source code. It simply does not exist in the same fashion. The difference is so obvious it can easily be missed. Others have claimed the user interface can be thought of as a tangible aspect of a software system. The UI is certainly a tangible aspect of the system, however you still cannot determine the large-scale structure of a software system by looking at its UI in just the same way as you can't determine the large-scale structure of a car's engine by looking at its dashboard. There is a fundamental difference between the forms of the systems produced by the respective disciplines.

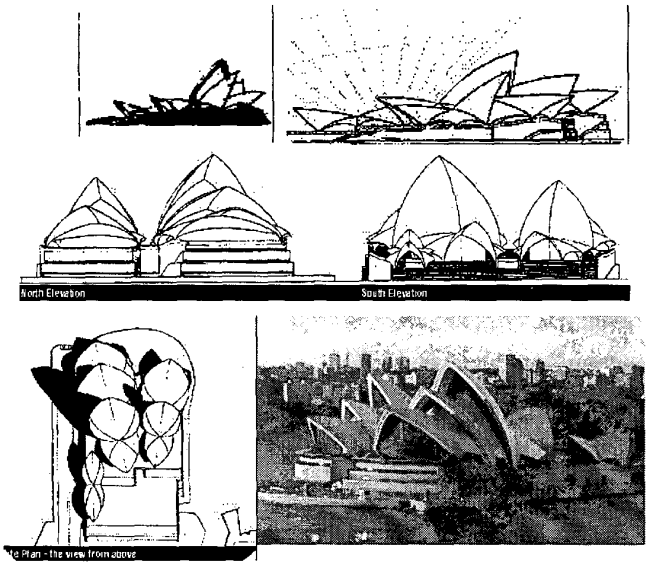

Figure 1: Architecture Diagrams and Physical Representation of the Sydney Opera House

[1] We recognise that electronic engineering generally does not have this property.

Software systems have no analogous physical form. They are not tangible systems and therefore their high-level, abstract, design representations must be different to those produced by the peer level of design in other engineering disciplines. Empirical research has shown that software developers produce multiple, high-level abstractions to represent their systems and the evolution of research ideas has assumed that they can be devised and used in an analogous manner to those architecture views of other disciplines. Indeed, it may be possible. However, the current understanding of software architecture views is based on an assumption that, while employed for a long time, has never been validated. During software development, large-scale design representations are created in the conceptual design phase, the implementation stage, the maintenance stage, and all other stages in between. Do they have any relation to each other? Is it possible to derive them all from the source code? Are they immutable in the same sense as traditionally built architectures? Software engineering researchers answer "Of course!" to these questions and use further analogies with other engineering disciplines as justification. Those justifications however, fail to consider the differences between the disciplines and the lack of tangibility of software is one difference that makes the use of those analogies hard to justify. To determine whether those multiple representations of software architecture are views in an analogous sense to other disciplines the following question needs to be answered. What is it about the nature of our discipline, rather than other disciplines, which makes it so?

System Execution. The other important difference between software systems and traditionally engineered artefacts concerns the concept of system execution. Software has a distinction between the implemented system, the collection of source code, and the executing system, that is, the way the source code is executed by the implementation environment to realise the system. This distinction does not exist in any other discipline. A software system is nothing more than a collection of source code statements until it is compiled and executed, statement by statement, by the 'virtual machine' implied by the semantics of the programming language. It is not until this stage that the system realises the desired result – a fact that is taught to all computer science students and perhaps forgotten not long after.

Some researchers contest the uniqueness of the distinction between system implementation and system execution. Counter arguments make analogies with other disciplines such as, "What about the flow of movement through a building?" or "What about the execution of a motor vehicle or electronic device?" To refute those claims, a distinction is made between the operation and the execution of a system. This distinction is critical to realising the differences between software systems and traditionally built artefacts and, therefore, warrants a few examples. Users can operate a software system through its user interface but that operation cannot occur until the system is being realised through its execution by the computer. Motor vehicles and electronic devices certainly operate but they are not executed in the same manner. The construction of a motor vehicle results in the existence of a constant mechanical linkage between the physical components. As the driver is operating the vehicle, the gross structure of its dynamic operation is exactly the same as the gross structure that was the result of its construction. Similarly, computer architecture remains the same whether the machine is being used or not. A user can operate mechanical and electronic devises but they have no need of an external system to provide its execution. They may require power through electricity or combustible fuel for the components of the system to operate and exhibit the required properties. However, once supplied that power they continue to execute independently and have no need of concepts such as a 'threads of control'.

3.2 Differences Between the Content of Architecture Views.

The difference between the concepts of system form and system execution in the respective disciplines affects the content of the architecture and architecture views used in the respective development processes. This does not simply refer to the obvious differences between corporeal systems and software systems but rather to the content of each view and its relationship to the system as a whole.

Traditional building disciplines produce many different representations of their system architecture. Those views are constructed by removing some of the implementation detail and leaving a subset of the devised form. Each view may correspond to particular viewpoint of one of the actors in the development process and each view is understood in the context of the global structure using the understanding of the physical form or features of the entire system. For example, how the wiring moves throughout the spatial arrangement of the automotive vehicle, or how the plumbing system is laid out within the spatial arrangement of the building. Those high-level representations can be developed both before the system is realised and as documentation after the system is completed. They depict a view of what some aspect the physical system is or will be. Not how the system will operate, but how that aspect of the system will exist as a corporeal artefact.

The content of architecture views as viewpoint-oriented subsets of the global design or implementation is not repeated in software architecture views. Earlier in this paper we presented the different collections of architecture views identified by Soni, Kazman, Kruchten, and Bass et al. The specific views in each of those collections can be grouped into the following three categories:

- Static Implementation Architectures: The representations that depict the source code modules and the relationships between them. Examples from the identified taxonomies include – source code, module interconnection, structural, development, physical, call-structure, object-structure, etc views.
- Dynamic Operation/Execution Architectures: The architectures that depict how the system executes in terms of functional abstractions of the implemented system and execution abstractions of the computing environment (e.g., processes, distributed machines, threads of control information). Examples from the identified taxonomies include – execution, allocation, process-structure, coordination, etc views.
- Conceptual/Logical Architectures: The representations used during the conceptual design phase of development that depict what the designer believes should be implemented. Examples from the identified taxonomies include – conceptual, domain level, logical, etc views.

If the many large-scale system representations of software systems are in fact analogous to the different views of traditionally engineered artefacts then these categories should all be obtainable from the underlying software implementation. However this is not the case. The architectures used to represent the only 'tangible' part of the system that exists, the source code implementation, are fundamentally different to those used to represent the executing system. Representations of the source code implementation depict how the system is implemented using the building blocks provided by the implementation language(s). These building blocks include files, procedures, functions, rules, object definitions, etc. That is the only system representation that can be directly perceived by us, yet it does not contain all the implementation detail necessary to understand what the system does or how the system executes to realise the requirements[2]. It is missing services provided by the operating system; services provided by other software systems, both those provided at compile time by linking in additional libraries and those provided at run-time by communicating processes; and it is missing information that affects the operation of the system because it is hidden in data values rather than being explicit in procedural invocation.

The source code is the lowest level of system granularity, the detail from which larger-scale abstractions are generated. However, it is missing the detail necessary for understanding how the system will execute. That additional detail is available only at run-time after the source code has been compiled and is being executed. The missing information is depicted in the abstract concepts evident in the architecture representations of the dynamic operation of the system. Those representations detail the operating system processes, the inter-process communication abstractions, the distributed nature of the system and the other services that become part of the system at runtime. The representations we have to depict the static implementation of the system and those that represent the dynamic operation of that system are different. One is not merely a subset or more abstract 'view' of the other. They are different, and the reason they are different is because of the differences that exist between the discipline of software development and those from which we draw the concepts of architecture and architecture views. Our systems have no tangible form and our systems have a distinction between system implementation and system execution.

The difference between system implementation and system execution also highlights the fact that no software system representation, from lowest level of detail, through to most abstract architecture contains the information that explains how the system is executed. It is not immediately obvious because few, if any, other disciplines require it in their system representations. In other disciplines you look at the architecture of a system and infer how it works. That is because those systems are not executed by another machine. Software systems are executed and knowledge of the operation of that execution engine, the virtual machine implied by the language, is necessary to understand how the system is executed.

The majority of systems are implemented in procedural or object-oriented languages. Developers can conceptualise the operation of those by implicitly following the procedural invocations as the imagined thread of control moves through the system components. Object-oriented terms like 'message passing' are still, at the code level, procedure invocations. Designers viewing system representations automatically apply that knowledge of how that model of abstraction operates to solve a problem, often without explicitly realising it. It becomes evident however, when attempting to

[2] Again, some may argue that the user interface constitutes a tangible aspect of the system. That debate is not considered here because it does not alter the subsequent conjectures.

understand a system representation that has been implemented in a language that utilises its own virtual machine rather than traditional procedural invocation. For example, understanding how a system implemented in Prolog operates must be done with the knowledge of how a backward-chaining inference engine works. The dynamic execution architectures of a realised system are not generated by abstracting away detail from the large and complex implementation because those details do not exist in the implementation. Again, we have an architecture representation that is not a subset or abstraction of some other, more complex, representation. It is different to the implementation because of the fundamental nature of software systems.

Like the static implementation and dynamic operational views of a software system, it is impossible to consider the conceptual views as a subset of the implementation detail. The concepts represented in the logical or conceptual level depictions of software architecture contain abstract, domain level concepts. They are mentally conceived entities that have no tangible manifestation. They may attempt to model or mimic tangible things, but they themselves have no form. The realisation process of a software system as an executing computer program occurs by implementing those mentally conceived, domain level concepts using the constructs provided by the programming language and operating system, and subsequently executing them in a machine. Those mentally conceived notions might be similar to implementation level concepts, however they do not have to be. Indeed the essence of software development is the process of implementing those domain level concepts of our minds using the constructs provided by whatever implementation environment is at our disposal. This is not generally the case in any other engineering discipline [27]. High-level software design representations consist of abstract concepts that depict domain level functionality and/or behaviour. In contrast, large-scale representations of the implementation consist of concepts provided by the implementation medium. For instance, language constructs (e.g., functions, rules), virtual machines, files, operating system processes, etc. They are different collections of concepts.

The difference between the two can be explained through a better understanding of a word that is often used in software architecture research – 'abstraction'. The existence of different architectures for a software system has been explained as different abstractions of the complex implementation detail. The definition of the word abstraction is often quoted from Shaw's work as a simplified description of a system that emphasises some of the system's details or properties while suppressing others [14]. That definition matches the one in a standard English dictionary. It also matches how views are assumed to be generated in traditional built architecture, where each view is a subset of the system as a whole. However, that is not the situation with software architectures. They match a definition of abstraction discussed in philosophy and psychological – see for example [28]. In those fields, abstraction is the technique by which higher order concepts are used to further intellectual reasoning by representing distinct, yet similar, particular instances. For example, apples and bananas can be represented by a single concept, fruit. That is how abstraction is used in software architecture. The collection of particular implementation concepts, such as objects, message queues, etc are represented by a different concept such as a blackboard. A blackboard does not exist in the software system. What 'exists' is a collection of programming objects or procedures, in conjunction with operating system message queues. We simply choose to refer to that collection by the single concept 'blackboard'. Similarly, there is no particular instance of 'fruit'. There are apples, bananas, oranges, etc. We simply choose to refer to them collectively as 'fruit'.

Software architecture views are not developed by merely removing the unwanted detail. They involve the generation of higher level, abstract concepts to represent the underlying detail. Moreover, many higher level concepts can be used to represent the same particular instances. That is why many architectures can be used to describe the high level structure of a software system. That is, a conceptual architecture can be realised by many implementation architectures and an implementation architecture can be represented by many conceptual architectures.

It is true that some representations, for example high-level object diagrams, have a smaller cognitive distance between the design level concepts and the implementation level concepts. Similarly, when modifying an existing system or building upon some previously implemented system the conceptual architecture may consist of components that have direct analogues in the implemented system. However, it is not true of all high-level software architectures developed early in the design process. They are different from the architectures developed during the same stage of other disciplines and are not different views of the implementation complexity.

3.3 Differences between How Views are Used in Practice.

The final difference to be noted concerns how these large-scale structures are used in practice. Shaw's original architecture paper noted the existence of large-

scale software representations ("abstractions") and proposed these could result in an "architectural level of design" that is analogous to the one that is presumed to exist in traditional engineering disciplines [13]. Traditional building disciplines develop the architecture, the gross structural form of the system, during the initial design stages of the development process. The form is specified in large-scale representations and a process of refinements specifies precisely how that form will be realised in terms of physical materials. The gross-structure of the form remains throughout the process.

This is not the case in software development. The creation of large-scale, conceptual representations is also noted during software system design. However the process of moving from the conceptual representations to the dynamic operation and static implementation 'views' is not an analogous process of refinements and specifications (regardless of how it is popularly described). This is due to the nature of the elements that are contained in those representations. They are not representations of corporeal components in an analogous manner to traditional system architectures. As we have discussed previously, the concepts represented in the design level depictions of software architecture contain abstract domain level concepts, which must be realised using the constructs provided by the programming language, operating system, and other existing components, and then subsequently executed by the machine. Progress in software design research is concerned with reducing the cognitive distance between the concepts that exist in our minds and those that are realisable in the implementation medium of our discipline. Programming language improvements, such as object-oriented languages and FGLs, have attempted to bring the implementation level closer to the mentally conceived components. Alternatively, design methods and patterns attempt to provide techniques that help to develop mental level components, and their interactions, that are more easily, and predictably, realisable in our implementation medium(s). Regardless of these advances, the cognitive distance exists and must be traversed during all software design activities.

Because the nature of our systems are different to those of traditional engineering disciplines and the nature of the content of our large-scale representations for them are different, they way they are used will also be different. Therefore it is impossible to consider "an architectural level of design" for software development that is analogous to those other disciplines. It is important to note that we are not saying analysis at this level of design is neither possible nor useful. Advances in areas such as product-line architecture are obviously benefiting the community. However, in order to reason why they are so useful and in order to perform research to establish improved practices, it is necessary to develop a view of software architecture based on the nature of software systems and not traditionally engineering artefacts.

4 Conclusion.

This paper has argued that the problems that exist between software architecture theory and practice exist because our understanding of the issues is based on specious analogies with traditionally engineered artefacts. A review of the history of the field shows how our understanding has evolved and how it appears plausible. Nevertheless, a closer investigation reveals significant differences between our discipline and those with which we made those analogies used to derive that understanding. Certainly software developers utilise many large-scale representations of their systems during and after the development process. Traditional engineering disciplines also utilise many large-scale representations of their systems during and after their development process. However, differences exist between the types of systems developed in the respective disciplines; the relationship between the content of the different representations and those implemented systems; and differences between how those representations are utilised in the development processes of the respective disciplines. Those differences seriously question the theories extrapolated from our present understanding.

We are not suggesting that all research in software architecture is pointless and should be abandoned. The discipline is undoubtedly producing results that benefit the community. Research in psychology shows that disciplines often form the basis of their understanding of new phenomena on something that is already well understood (see for example [29]). However, as the discipline progresses it is often necessary to reject that initial understanding and develop something more appropriate. Research in the philosophy of science has considerable literature in this area. It is beyond the scope of this paper to go into those details but it is something already investigated by the authors. We are suggesting that in order to improve research in software architecture and to reduce the difference between theory and practice, a different way of understanding the nature of our systems and how they can be engineered is required.

Earlier versions of this material have elicited comments suggesting we are merely poking holes in the current understanding of software architecture without providing a legitimate alternative, and that is certainly one valid assessment. However, we believe this issue is so fundamentally important that it is necessary to make people aware of the problems so that a community-wide

discussion can begin. Moreover, fitting a thorough treatment of the problems and possible solutions into a single paper is extremely difficult in this philosophical area. What we hope to achieve is a commitment to the development of a better understanding of the fundamental nature of software systems and their development. Answers are needed to the questions that are often posed in commentary-style journal articles (e.g., [30]) and in informal conference discussions and keynote addresses (e.g., [31, 32]). "What do we build and how do we build them?" "What does software engineering really mean?" These are not easy questions to answer. They will not present quantitative results that are easily testable or easily publishable. What is required is work on the philosophical foundation of the discipline. We have already working towards solutions, see for example [33], however we believe more literature and conference-based discussion is required. Without a good understanding of the nature of our own discipline we will continue to grasp at analogies and attempt fit the square-pegs of other disciplines into the round-holes of our own problems.

5 References.

1. Mobray, T.J., *Will the Real Architecture Please Sit Down?* Component Strategies, 1998(December).

2. Bennett, D., *Desiging Hard Software: the essential tasks*. 1997: Manning Publications.

3. WWISA, *Philosophy*. 1999, Worldwide Institute of Software Architects. http://www.wwisa.org/

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

5. NATO, Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy Oct 27-31, 1969, in Software Engineering Concepts and Techniques: Proceedings of the NATO confereces, J.N. Bruxton and B. Randall, Editors. 1976, Petrochelli/Charter.

6. 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.

7. Parnas, D.L., On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, 1972(December).

8. Brooks, F.P., The Mythical Man-Month: Essays in Software Engineering. 1975: Addison-Wesley Publishing.

9. Brooks, F.P., *Architectural Philosophy*, in *Planning a Computer System - Project Stretch*, W. Buchholz, Editor. 1962, McGraw-Hill. p. 5-16.

10. Coplien, J., *Architecture as Metaphor*,, http://www.bell-labs.com/~cope/ArchitectureAsMetaphor.html, March 2000.

11. Weinberg, J., *Architecture as Metaphor*. Personal Communication. March 2000.

12. Coplien, J.O., Reevaluating the Architectural Metaphor: Toward Piecemeal Growth. IEEE Software, 1999(Sept/Oct).

13. 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.

14. Shaw, M., Abstraction Techniques in Modern Programming Languages. IEEE Software, 1984(Oct): p. 10-26.

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

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

17. SEI, *Software Architecture Definitions*,, http://www.sei.cmu.edu/architecture/definitions.html, September 1998.

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

21. Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Developent Process*. 1998: Addison Wesley Longman.

22. Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*. SEI Series in Software Engineering. 1998: Addison-Wesley.

23. Clements, P.C., *Software Architecture: An Executive Overview*. 1996, Software Engineering Institute. CMU/SEI-96-TR-003.

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

25. *Sydney Opera House*,, http://www.soh.nsw.gov.au, April 1999.

26. Miriam-Webster Dictionary: http://www.m-w.com/netdict.htm. 1997.

27. Baragry, J. and K. Reed. Why Is It So Hard To Define Software Architecture? in Asia Pacific Software Engineering Conference. 1998. Tapei, Taiwan.

28. Corsini, R.e., *Encyclopedia of Psychology*. Vol. 1. 1984, New York: NY Wiley.

29. Holyoak, K.J. and P. Thagard, *Mental Leaps: Analogy in Creative Thought*. 1995: MIT Press.

30. Gilb, T., *Level 6: Why We Can't Get There From Here.* IEEE Software, 1996(January).

31. Reed, K. Commercial Software Engineering, The Way Forward. (keynote address). in Australian Software Engineering Conference. 1987. Canberra. ACT. Australia.

32. Xia, F. (Panel Session) How Can We Conduct Research In Software Engineering. in Asia Pacific Software Engineering Conference. 1998. Taipei, Taiwan.

33. Baragry, J., Understanding Software Engineering: from analogies with other disciplines to a philosophical foundation., PhD thesis in Dept of Computer Science and Computer Engineering. 2000, La Trobe University.: Australia. p. 350. (Available from the author).