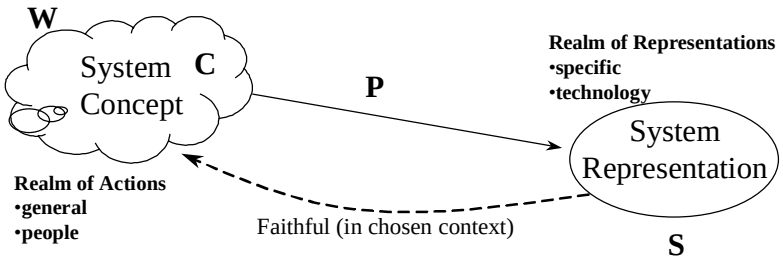
Unification of Components and Objects Through Abstractions

A position paper submitted to the Component Management Infrastructure Workshop
Dan Port, USC Center for Software Engineering

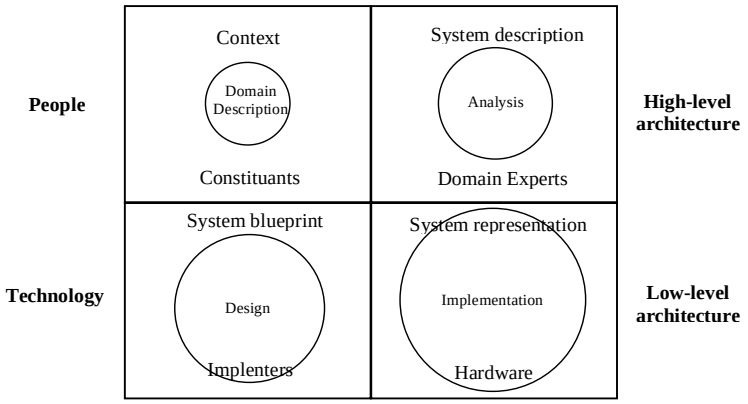
Introduction: Are component-based systems different from object-based systems? The position taken in here is that for sufficiently complex systems, both components and objects play vital roles in defining a systems fundamental architecture. Any perceived differences lie only in the level of abstraction the system is viewed. Rational for this will be offered through the consideration of *formal abstractions* as a generalization of components and objects, in addition to several other modeling elements. We will indicate a conceptual system development framework using formal abstractions in which both components and objects arise in a natural way. Such a framework clearly will affect the management infrastructure. Consequently, the framework provides insight into fundamental questions such as "Where do components come from, and how do they directly relate to the implementation?", "How do you find objects, and how do they directly relate to the architecture?", leading to the general question, "What is software architecture and how does it emerge as part of the development process?" The framework discussed herein has been applied extensively within several academic and industrial settings.

Background: Let us consider a conceptual system **C**, a "real world" domain **W**, and a software system **S**. We can now view the software development process as the construction of a mapping **P** that takes concepts and maps them to software elements. An essential quality of P is that it should be *faithful*, in that events in **S** map back in an analogous way to events in **C**.

The conceptual system **C** lives in the *realm of actions*, which provides two main qualities. First, the system **C** has *generality* in that it is defined at a high level of granularity and as such can live simultaneously in multiple contexts within **W**. Second, is defined within the context of *people*, or "real world" actions and events. The software system **S** lives in the *realm of representations*, and it too provides two main qualities. One is *specificity*, that is a software system must represent something specific, otherwise it would not be a well-defined representation. The other is that S must be defined within the context of *technology*, for that is the vehicle for software representations. This is summarized in the diagram below:
The required translation must take place through a process that gradually evolves **C** from general, people like



qualities, to specific, well defined technology oriented structures. One effective approach to this is to pass the system concepts through a series of *intermediate representations* (i.e. model layers such as analysis and design) each of which refine the concepts into structures appropriate to the overall qualities for a particular model layer. Although the characteristics of the realms of action and representations are not entirely orthogonal, they do provide a reasonable basis for intermediate representations and thus define model layers as indicated in the diagram below (which is not meant to imply any order in which the layers are traversed or layer size):



We find the entity, component, object, and data structure model elements through consideration of identifying the forms that exist within each model layer. For example within the Domain Description, the general-people kind of form is an entity of the organizations domain, and a specific-technology kind of form is a data structure within the implementation. It is natural to consider components in the analysis of a system and objects within the design. At a higher level of abstraction, closer to the conceptual system, Components capture the high-level architecture. Similarly, objects are closer to the implementation, and hence capture the lower-level architecture.

Abstractions: We have set up the framework that integrates various model layers that can bridges the gap from **C** to **S** in an evolutionary way (although we did not specify a particular evolutionary process). There is one major issue we have neglected up this point, and it is vital. For all this to be meaningful, it must actually operate on something meaningful. This implies the need for a structure that we can operate on in a general way that preserves important information and can be easily refined as the structure transition throughout the layers. For this, we introduce the idea of *abstraction.* An abstraction will serve as the basic "stuff" from which we create models and in a general and flexible way can represent the concepts that arise within our development process. Formally, an abstraction is a recursive set of qualities as indicated in the following:

Abstraction **A**
Quality of Abstraction **A**(1)
    Quality of Quality **A**(1,1)
    Quality of Quality **A**(1,2)
        Quality of quality of quality **A**(1,2,1)
        Quality of quality of quality **A**(1,2,2)
        ...
    Quality of Quality **A**(1,3)
    Quality of Quality of quality **A**(1,3,1)
    ...
Quality of Abstraction **A**(2)
…

Where a quality is a fundamental element of **W** that represents a value. Qualities may themselves be abstractions which contain other qualities. A qualities values can only be resolved through constraints (even if the constraints are implicit, obvious, or unstated). Adding qualities to an abstraction (or to existing qualities) constitutes a refinement, whence it becomes more specialized. In this way abstractions unify the various model elements. An abstraction may start as a system responsibility that involves various entities. These entities may then be specialized for use in a particular system as components. In turn, components may be represented in software as collections of objects. Finally, each object will be implemented through some manner of data structure.

Although there are a great many interesting issues regarding abstractions, such as elegance and engineering, they can not all be addressed here.

The conceptual system **C** is literally a collection of concepts. A concept is a difficult thing to work with directly, and thus each concept is represented with abstractions. In our case, let **C** = ($\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{a}_3$, ...) be the set of abstractions that represent the conceptual system. Note that the only assumption were are really making is that there is a conceptual system. it's not critical that this representation be complete, consistent, or even well defined. Such issues can be handled through the lifecycle that constructs the map **P**.

Abstraction evolution: We now come to the culmination of our approach to model unification. Since our ultimate goal is to produce **S** from a faithful mapping **P**, specificity requires the break down of concepts contained in **C** into more refined abstractions that have the qualities of software. This can be accomplished by applying the model layers $\mathbf{P}_{DD,}\mathbf{P}_A\mathbf{P}_{D,}\mathbf{P}_I$ to the abstractions ($\mathbf{a}_{1,}\mathbf{a}_2\ \mathbf{a}_3$ ...). Abstractions start out general and due to pressures applied at each layer (e.g. domain description, analysis, design) evolve, and the abstraction becomes more specialized through refinement. It is common to break an abstraction up into more specialized abstractions, such as a component decomposing into several objects when moving from analysis to design. This process is what we call *abstraction evolution*. Treating all elements in the domain as abstractions allows for a unified approach to modeling. We can employ common techniques to describe and operate on abstractions and the model can be adjusted by changing the type of abstraction used to represent a particular construct. For example, an element may start out as an attribute, then become a relationship, then an object, but the essential information is preserved throughout the transitions.

Implicit in the evolutionary process is the notion that only the most useful and fittest abstractions will survive. The names change to reflect the degree of specialization an abstraction has attained and the new qualities it has acquired. An abstraction may stop evolving at any point, but may start again, or influence other abstractions or become extinct. Eventually the set of abstractions ($\mathbf{a}_{1,}\mathbf{a}_{2,}\mathbf{a}_{3,}$ ...) will evolve into a set of software abstractions ($\mathbf{s}_{1,}\mathbf{s}_{3,}\mathbf{s}_3$, ...) preserving their essential concepts from **C**, thus encouraging faithfulness, but now having qualities of a software system. The diagram below depicts the various specialization's that an abstraction can undergo through the various model layers.