Introduction to Software Architecture and Other Confusing Topics

James Snyder

These readings provide an introduction to what software architecture is and is *not*. Additionally, important topics are presented to clarify how design and engineering impacts architecture.[1]

The purpose of this work is to provide an intellectual framework for a self-paced study in the topics of software architecture, frameworks, design patterns, and object-oriented software construction. This document is organized as a series of summary discussions on relevant topics with supplemental readings from selected book chapters and papers. Readings will be pointed out in side-bar comments adjoining summary discussions of the topic. It is important to note that the summaries assume completion of the readings.

Who Should Read This?

This course assumes that you have programmed in some language or have managed a group of developers. Even if you have not, it will provide clarification of many of the notions thrown around in software development circles today.

As a reader, you will get the maximum utilization of this material if you have programmed in some object-centered programming environment such as Scheme, CLOS, Smalltalk, Eiffel, Java, or even C++. Experience with Abstract Data Types is also helpful even if they were developed in older languages such as C, Pascal, Ada, or Modula-2.

Lastly, the format of the document was setup with large margins on the left to allow you to take copious notes as you read, think, and become stark-raving

1. This work is currently under revision for a content update.
For updates email jdsnyderii@alumni.cmu.edu

mad with the preposterous pontifications of perilous, persnickety prose. In other words, fasten your seat belt because we are going down a road that can be quite dangerous.

The Importance of Terminology

One of the reasons that so much confusion exists about the concepts covered in this course is that appropriate use of terminology has not been a priority. Several classic examples of muddled terminology are given below.

Software Engineering. Unfortunately, the term *software engineering* is not really about engineering in the strictest sense, it is about software development. Development consists of design (the formulation of a solution to a well-defined problem) and implementation (the combination of implementation technologies into something built). However, software engineering as a term was published in the literature and has since been a fixture much like Xerox for copiers and Kleenex for tissues.[1] A good characterization of engineering is given in [Shaw and Garlan 1996] page 6:

> Engineering relies on codifying scientific knowledge about a technical problem domain in a form that is directly useful to the practitioner, thereby providing answers for questions that commonly occur in practice. Engineers of ordinary talent can then apply this knowledge to solve problems rather than relying always on virtuoso problem solving.

Transparent Computing. Being from a computer science background, I have happily used the term *transparent* to describe varied characteristics of a system design. I have gratuitously and incorrectly used the following phrases on many occasions:

- I designed a *transparent* interface between classes
- X-Windows is a *transparent* windowing system
- My distributed object system is based on location *transparency*
- The communications layers are based on a *transparent* connection mechanism.

So, now that we can, by example, transparently see the definition of transparent, it should be clearly obvious that the meaning of transparent is to "clearly see the internal workings of an artifact." To make the point more clearly, let me pose the following question. Would you buy a *transparent* house? Unless you like people seeing you in your *underware[sic]*[2] in the morning, I think not. So, why would we propose to describe computing transparently?

1. The term was published by NATO in 1968 and has not left us since [Shaw and Garlan 1996], page 6.
2. No, I don't mean under*wear*.

Because we usually mean to say something like *opaque*, *independent*, or *seamless* instead of transparent—and from here forward I expect that you will. You can get a more complete description of this term in [Neumann 1996].

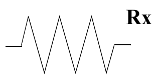
Is It A Class Or Object. If you consider yourself a software developer, can you clearly articulate the difference between a class and an object? Let me first state that a common *incorrect* answer is that there is no difference—there really is. More importantly, without being able to articulate the difference, confusion about design description or intent is bound to increase.

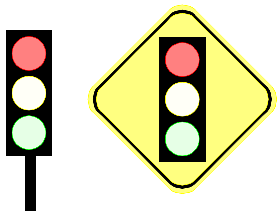
A second answer that is also incorrect is that classes exist at compile time, and objects exist at runtime. This description may capture some aspects of correctness for some programming languages, but as time progresses, this characterization continues to be less true (e.g. Java is reflexive). Still however, there exists a distinction between these terms that must be well-understood before people can properly communicate about software designs. This distinction also relates to questions about the suitability of pictorial notations to support design. A more precise definition of classes and the suitability of notations will be discussed in later sections.

Simulation-Based Design. In order to realize a simulation of a system, we need to have a pretty detailed model of how the system will behave. Effectively what we are doing is using the simulation results to analyze the solution captured in the simulation model that we hope solves problem.

Therefore, we have already designed the solution to such a degree as to be able to model it in the computer. Therefore, the simulation does not help us design, it helps us analyze specific models of design solutions. Simply put, the technique named *simulation-based design* is *engineering analysis* technique.

So What *Is* the Big Deal. Well, I guess that depends on what the definition of *is* — is[1]. In all seriousness, we need to bring rigorous definitions to many terms because we need to use them to communicate about and describe designs without ambiguity. For example, when a circuit designer draws the symbol for a resistor, it means something concrete and precise. When a building architect draws a line on the paper, it is in the context of a drawing scale and means something precise such as a wall edge or window. We find the same notions in everyday life such as traffic signals and signs.





1. For those of you who are not up on riveting politics, this phrase was used by President Clinton in response to a question posed during a deposition.

As a discipline, we need to be able to rely on such concrete definitions of terms to be able to create a broader understanding of software development. So, this course is the first place where you can practice and struggle to bring maturity to the field.

An Initial Definition for Software Architecture

Rather than wait for the grand *finale*, it is best to get the a working definition out into the open now. A software architecture is *the realization of the software artifacts in their operating environment*. More generally, architecture is *the well-intentioned combination and placement of design elements to achieve a specific set of functionality*. The implication of these definitions is that software architecture is a *destination* not a beginning.

So, this definition probably goes against every grain in your body from what you have been exposed to in the past. You might be saying to yourself, "hey, what about things like layered architectures?" These notions are in fact really *architectural styles or organizing principles* and will be covered in greater detail later on[1].

Creating a lucid and coherent understanding of software architecture is challenging for both authors and readers of the subject. The introduction provided here will attempt to tackle the subject by providing a discrimination between the different aspects of software development — that is — how design and engineering *lead* to architecture.

Since the working definition is fairly severe compared to other definitions, it is important to justify it in concrete terms. The most appropriate way to justify the definition is to look outside the realm of software development to other design disciplines. The two examples provided here are from building architecture and computer architecture.

Architecture and Buildings

People have been working for thousands of years to understand and communicate the architecture of the built environment — in other words buildings. When we talk about a Gothic cathedral, we characterize the *architectural style* of the building. However, when we talk about the *Notre Dame* being Gothic and *good* architecture, it is the building itself in its environment that is the architectural product being evaluated against a plethora of criteria. The building is architecture while Gothic is style.

1. This is again an unfortunate lack of precision in terminology.

Architecture and CPUs

Another common definition of architecture is found in the field computer architecture. CPUs such as the Pentium or PowerPC *are* the architecture. We refer to the instruction set of the CPU when describing what the CPU can do. When we talk about the style of the CPU we conjure up terms like a Reduced Instruction Set CPU (RISC) or a Complex Instruction Set CPU (CISC). Similarly, we talk about pipelined architectural styles to characterize how instructions are processed and what processing optimizations are introduced to get speedup.

Architecture and Software

First, I need to point out that all the fuss about architecture is due to the fact that most problems require more than one person to realize. So, we need to exclude "academic" problems from the discussion[1]. Second, when it comes to software, the term architecture has been so overloaded that it is now marginally useful. If you have ever been involved in the development of a software system at the early phases of development, you have more than likely been exposed to people who demand to see your "architecture" (e.g. your UML class diagrams) so they can decide if it is "good". In reality, they should be asking questions like:[2]

READING ASSIGNMENT 1
Read Chapter 1 of [Cooper 1999] for in important lesson in user-centered thinking. "Good" architecture requires the architect to explicitly and intentionally address good user interactions as part of the design process.

- What are your organizing principles?
- Have you properly captured the requirements of the users?
- How do you expect the software solutions you develop to enhance or change things as they are done today?
- Do you understand the complexity and design rationale of the existing software systems you must interface with so as not to *break* them?
- How are you going to organize the design process participants so that they have the information they need, when they need it, and in a form that is complete for their needs?

An important software architecture reference is [Shaw and Garlan 1996]. While they implicitly or explicitly provide similar perspectives, they do not maintain as clear of a distinction between architectural style, design, and engineering. None the less, their definition of software architecture is important and is given below:

1. In fact most classroom problems are designed to graded on an individual basis and are not really representative of interesting problems.
2. To answer these questions, system architects need to understand the real-world systems they are designing to. However, understanding does not come in an instantaneous flash and therefore requires an incremental accumulation of knowledge to validate understanding. The process of formally capturing knowledge or real-world systems as they exist is called *Domain Modeling*. We would like computational support because domain modeling is a knowledge "chunking" problem and is hard to do without computer support much like complex document construction is hard to do without support.

READING ASSIGNMENT 2
Read Chapters 1 and 2 of [Shaw and Garlan 1996]. Note that the authors' definitions are not as crisp as we would like in this work. While it is probably the best description to date, a distinction between realized artifacts and design characteristics is not maintained.

The architecture of a software system defines that system in terms of computational components and interactions among those components. Components are such things as clients and servers, databases, filters, and layers in a hierarchical system. Interactions among components at this level of design can be simple and familiar, such as a procedure call and shared variable access. But they can also be complex and semantically rich, such as client-server protocols, database-accessing protocols, asynchronous event multicast, and piped streams.

In addition to specifying the structure and topology of the system, the architecture shows the correspondence between the system requirements and elements of the constructed system, thereby providing some rationale for the design decisions. At the architectural level, relevant system-level issues typically include properties such as capacity, throughput, consistency, and component compatibility.

More generally, architectural models clarify structural and semantic differences among components and interactions. These architectural models can often be composed to define larger systems. Ideally, individual elements of the architectural descriptions are defined independently, so that they can be reused in different contexts. The architecture establishes specifications for these individual elements, which may themselves be refined as architectural subsystems, or implemented in a conventional programming language.

SIDEBAR
Supplemental support for these characterizations can be found in [Monroe et al. 1997].

In more general terms, the term architecture is used here in reference to a physical, realized artifact. We need to use terms like organization, design, model, description, configuration, or style to describe notions about something not yet realized or to characterize aspects of an artifact.

The above definition provides two important notions. First, software architectures can be used recursively. Second, elements in the software architecture should be designed for reuse. To understand this point, we need to turn our attention to design.

A Characterization of Design

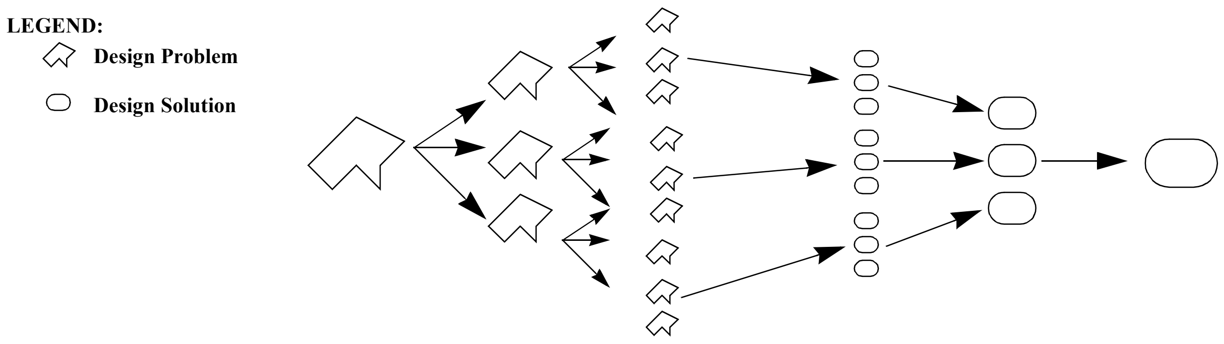
Having hopefully clarified the notions of style and architecture, we are still faced with a fundamental question. How do we build something so that in the end it is *good* architecture? We need a *design* to evaluate, and it is this term that many people mean when they say architecture. The design is used to communicate the way we believe the world *ought* to be *before* it is actually realized. It is also important to state that design is not engineering. A simple rule of thumb to discriminate between design and engineering is "if you are not measuring something, it's not engineering."

Hopefully we have established that what we need to be discussing is how to go about designing software systems that lead to good software architecture. To understand this, we need to have a working definition of software design.

Design as Problem Solving

The definition of software architecture given in [Shaw and Garlan 1996] reflects the general, iterative nature of software design; that is, software design is the process of decomposing requirements into subproblems, finding solutions to those subproblems that can be implemented, and synthesizing these solutions into a comprehensive whole that meets all of the system's requirements as shown in Figure 1 below[1].

FIGURE 1. Problem Solving by Decomposition

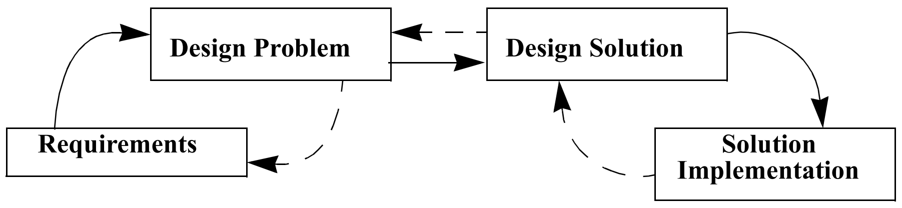


READING ASSIGNMENT 3
Read Chapter 5 of [Simon 1981] entitled "The Science of Design"

As subproblems become solvable, they lead to many design alternatives. Alternative solutions are combined and evaluated against many criteria for "goodness" until a whole solution is formed. Another perspective of this view of the world is shown in Figure 2 below. Essentially, requirements lead to a design problem that needs to be solved and can hopefully be clearly described. Given a problem, we then need to decompose that problem until solvable subproblems can be found, then their solutions are synthesized into a larger solutions until a complete solution is formed. As we discover things about the problem we are trying to solve, we may refine our requirements as indicated by the dashed arrows in Figure 2.

1. Note that a very important design concept is problem definition and not just about solutions — problem definition is actually the more difficult part.

FIGURE 2. Design From Requirements to Implementation



Iteration in Design Processes

In general, the activities shown in Figure 2 are done on a scale that is proportional to the problem. Ideally, we would like the requirements to solution implementation to occur inside one person's head thereby to reduce the need for communication between people. But, most interesting problems are bigger than one person can handle when constrained by either time or complexity. So we parcel the work out.

READING ASSIGNMENT 4
Read Chapter 2 of [Brooks 1975].

Many people assume that work and time are interchangeable thereby adopting the *Mongolian Hoard Theory of Management* — if one person takes 1,000 hours to do a job, 1,000 people can do the job in an hour. However, we all know that using this theory in practice creates all kinds of ambiguities as a result of information loss during communication. We have now created a *fundamental* paradox. Design lives somewhere between these two extremes.

As we decompose a problem into subproblems, we effectively create another design problem in which the process described in Figure 2 can be recursively applied[1]. As a result, we need to know when to stop decomposing a problem (i.e. when does recursion terminate?). The answer is really quite simple in practice. When the entire process can fit in one persons head — requirements to implementation — we can stop decomposing the problem. We can also describe the stages of decomposition by the phases a system design goes through as shown in Figure 3. Essentially, the notions of requirements to implementation keep getting broken down into smaller problem areas.

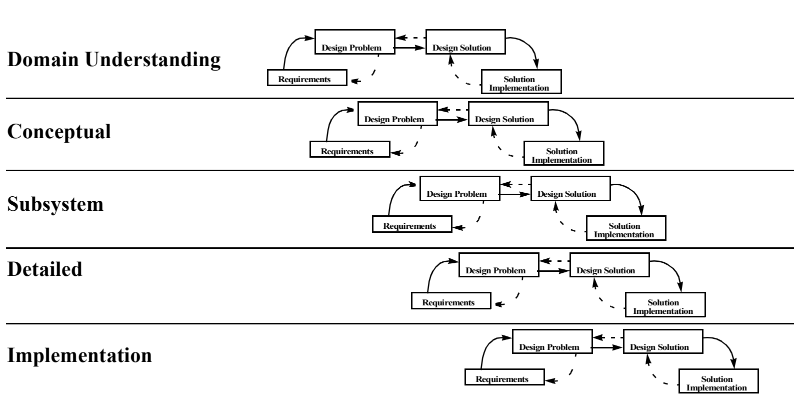
READING ASSIGNMENT 5
Read the papers by [Akin et al. 1995] and [Flemming and Chien. 1995]. Note that the term Architectural Programming *has nothing to do with software. These papers are from the* Building Design *literature to show well-formed ideas of design problems and design solutions.*

For example, if we have a design solution (e.g. a Factory Pattern) available to us, we will need to implement the solution. Notice that we again have requirements, design problems, solutions, and implementations even at this low level in the design process. However, the nature of the requirements, problems, and solutions change as we progress down the stages of design. The specificity of these problem/solution pairs also increases as we move down the stages of design.

1. Recall the recursive definition of [Shaw and Garlan 1996].

SIDEBAR
Note that there is no waterfall approach implied by the figure. Many complex relationships exist between all the phases in multidimensional ways and as such cannot be easily visualized.

FIGURE 3. Phases in System Design



Communicating Software Designs

Given the sheer complexity of the above notional design process, we need to understand how to turn these potentially random, stochastic, and chaotic processes into something orderly. As we will see below, the key to creating order is *human communication* that in turn will preserve the integrity of the design throughout the design process.

Achieving Conceptual Integrity in Software Design[1]

In Chapter 4 of [Brooks 1975], the author asserts that good architecture results from preserving the *conceptual integrity* of the design throughout the development process. Additionally, he asserts that preserving integrity can only occur if clear communication between the participants occurs. He also states that organizations form around lines of communication (*not the other way around*), therefore, communication structures need to be designed so that they form the appropriate organizations. Furthermore, he asserts that good design cannot result from a democratic approach—it requires an aristocracy. Simply put, I agree, and in my opinion, this book is one of the most essential readings for any software developer or manager of software developers.

READING ASSIGNMENT 6
Read Chapters 4 and 5 of [Brooks 1975].

So if we have an aristocracy, who gets the title of *Lord of the Dance*? Clearly the answer is the *System Architect*. Now matter how big or small the project is, the role must be filled. The sole responsibility of the system architect is to ensure that the conceptual integrity of the design is maintained.

1. In the context of C4ISR in the DoD, they have established extremely verbose notions of what the design deliverables are and they call it the C4ISR Architecture Framework V2.0. Essentially, they have created a set of documentation standards and deliverables that are required by all C4ISR systems. They call out three architecture views: the Operational View, the Technical View, and the System View.

READING ASSIGNMENT 7
Read Chapter 13 of [Cooper 1999].

So you may ask *why is design so important in software*? I will never be able to answer the question better than Alan Cooper [Cooper 1999]. A managed process is required to effectively execute design that is, in turn, required to preserve integrity that, in turn, results in architecture that lasts. Specific kinds of team organizations will be addressed later in the readings.

A Perspective Outside Software

Drawing on our building analogy, the blue prints of a building describe an abstraction of parts of the design of a building, however, the blue prints are not complete in themselves. First, blue prints come in plan and elevation with different levels of detail (e.g. 1/4″ and 1/8″ scale). They are intended to communicate information at different levels of abstraction.[1]

Additionally, blue prints are supplemented with construction specifications that laboriously detail every item that will go into the final product. For example, the kinds of carpeting, masonry, concrete strength and color must all be enumerated both in the construction specifications as well as in the drawings. The items in the specifications and the drawings must also match exactly. So, if the architect draws something in the blue prints but does not specify it in the specifications, they must absorb the cost, and *vice versa*.

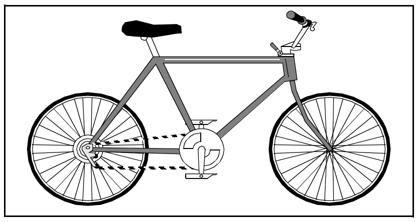
Describing Software Designs

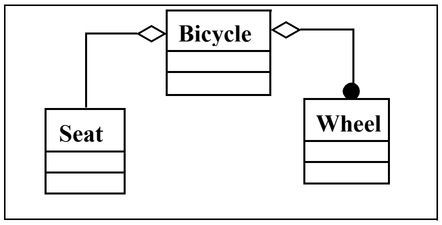
So, where can we find a similar notational mechanism in software design? I hope that class diagramming *notations*, as found in UML, jump out at you. However, it is important to understand that these notations are not architecture, rather they are descriptions and communicate only a specific part of the design just as blue prints only communicate specific parts of the design. Both kinds of notations must be supplemented with other descriptions. Often times only natural language will suffice.

An excellent example of this phenomenon can be found on the book cover of [Larman 1998] where it very clearly shows that we use notation as abstractions to help communicate about things but these communications are not the things themselves. Figure 4 shows a picture that is similar to [Larman 1998]. This reference is an excellent source for understanding how notation can be a helpful abstraction mechanism within the context of everyday software development.

1. Blue prints have evolved over thousands of years of experimentation and are intended to communicate a precise and common understanding to skilled practitioners.

FIGURE 4. Notation as Abstraction

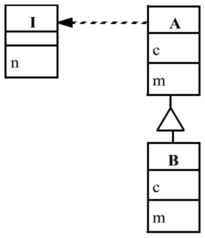

This is not a bicycle



This is not a bicycle

Current notational systems, including the popular UML, suffer from some important problems. Unlike other notational systems, software diagramming systems suffer from too much ambiguity. To make this case, let's look at a simple example. Figure 5 show two classes *A* and *B* as a subclass of *A*. Additionally, *A* is shown to implement the interface *I*. Given this example, we can now start asking questions about what the diagrams describe by asking the following questions:

FIGURE 5. Ambiguity in Software Notation



- What happens to the definition of the method *m*? Does *B* replace *m* or are they both available?
- What happens to the definition of the attribute *c*? Does *B* override *c* or are both *c*s available?
- How does the method *n* from interface *I* affect classes *A* and *B*? Is *n* inherited or must *A* and *B* conform to *I*?

The answers to the above questions are simple — it depends on the language they are intended to represent. As an implication, these class diagrams are abstractions of a *type system* in some programming language. Furthermore, these class abstractions can only have unambiguous meaning if the notions they represent have well-defined implementations and exist *before* code generation occurs. However, this requires additional information that is not captured in the diagrams.

An additional implication of these types of notational systems is that they represent the static aspect of class structure and therefore by any definition *cannot* be an architecture. It should be painfully obvious now that class diagrams communicate a small aspect of design and ignore, rightly or wrongly, many other important aspects of design communication — particularly the behavioral aspects of classes, for example using UML stereotypes.

Matching Notation and Abstraction Levels

Given that a significant number of abstraction levels exist during design, it becomes increasingly important that we understand how notation communicates the appropriate meaning at its particular level in the abstraction spectrum. However, we currently do not have such notations available to use. We need to either impose convention on existing tools and methods (e.g. tailored Rational-Rose and UML), or we need create notations sometimes on-the-fly.

If we do not have appropriate notations, we cannot communicate design adequately. What is important about notation is that it needs to have a well-defined place in the development life-cycle of software. To that end, we need to turn our attention to development activities.

A Notional Software Development Process for Design and Implementation

To put the material covered so far into more accessible terms, we need to shift the focus of the course to more tangible topics. This section discusses the notions of classes, design patterns, and frameworks and how the construction of these design elements can be integrated into the software design process.

Synthesis in Software Design

Assuming that we have sufficiently decomposed a problem into the relevant parts, we can then synthesis software systems by combining classes into patterns, combining patterns into frameworks, and combining frameworks into an architecture. A simplified notion of the approach is shown in Figure 6.

FIGURE 6. Synthesis of Software Systems

CLASSES ⇒ DESIGN PATTERNS ⇒ FRAMEWORKS ⇒ ARCHITECTURE

READING ASSIGNMENT 8
Read Chapter 4 of [Meyer 1988].

While most people believe they know what a class is, it is very likely that they will use the definition of a type from a programming language rather than an *abstract data type* (ADT) realized as a type in a programming language — *usually* object-oriented languages. In other words, classes are ADT *realizations* and *objects* are realizations (i.e. instances) of a class. What does that mean to you and me? The key distinction is that when you specify the ADT you need to know and specify what the *contractual obligations* are. In other words, we need to support *programming-by-contract* which is eloquently covered by [Meyer 1988].

READING ASSIGNMENT 9
Read Chapter 1 of [Gamma et al. 1995].

Given that we can now nicely define and build classes in programming languages, we can constrain the construction of classes using a clear understanding of a design problem and design solution. The seminal work describing this notion is of course the Design Patterns reference [Gamma et al. 1995].

Given that we can define and implement design patterns, we need be able to combine them in interesting and useful ways. A significant body of work has been done describing and implementing such combinations, and the term that has emerged is *frameworks* or domain-specific *application frameworks*. This definition was stated in the [Gamma et al. 1995] reading and has been generally accepted as a working definition.

READING ASSIGNMENT 10
Read the paper by [Fayad and Schmidt 1997] and the paper by [Bäumer et al. 1997].

However, to better understand the implications of combining design patterns into frameworks, a better understanding of frameworks is needed. Additionally, using multiple frameworks together in a single software system can be quite challenging and requires a clear notion (i.e. conceptual integrity) of what the overall design principles are.

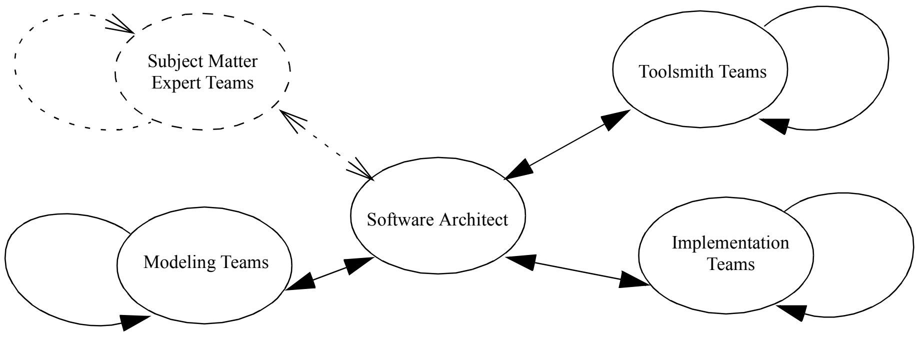
An Organizing Principle for People

Because software development is a human activity, we need to understand how people can work effectively together to produce software. It is important to point out that there is no easy answer to this question, but it is clear that as the complexity of the problem increases, the organizational structure needs to change. Good system designers assess the complexity of the problem at hand and formulate an appropriate organizational structure.

READING ASSIGNMENT 11
Read Chapters 3, 7, and 11 of [Brooks 1975].

A fundamental shift in how we organize software developers and managers is needed and ironically has been pointed out almost 30 years ago. The notion of a *surgical team* was pointed out by [Brooks 1975], and has both been universally accepted as one of the best approaches and ignored by the community. The lack of acceptance is probably related to two causes the first of which is university neglect — they do not tell people about it. The second cause is probably due to the fact that an explanation of how to construct a surgical team in-the-large has not been developed. So, a first cut at this is shown in Figure 7. Note that the center of the universe revolves around a software architect. Also note that the dashed arrows represent where *domain modeling* occurs as part of the design process.

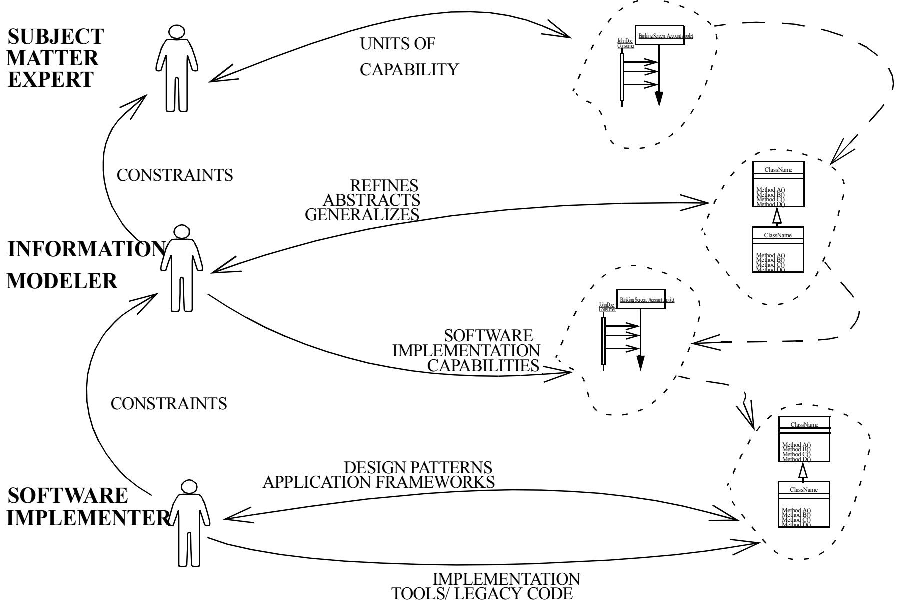
FIGURE 7. Interactions Between Teams



Teams and Software Development

Now that we know we need teams (as if we didn't already), we need some notional description of how these teams communicate and what they produce as a result of the work they do. Such a process is shown in Figure 8. *Subject matter experts* help software architects formulate requirements that are suitable to develop into well-defined design problems. *Information modelers*, in conjunction with the software architect, take these *design problems* and realize *alternative design solutions*. Then, *software implementation teams* take the selected design solutions and realize implementations. As the development effort progresses, all these activities occur simultaneously at varying levels of intensity. Therefore, any labels such as *waterfall*, *spiral*, or *iterative* really do not capture the essence of the process.

FIGURE 8. Software Design Processes


SUBJECT MATTER EXPERT
UNITS OF CAPABILITY
CONSTRAINTS
REFINES ABSTRACTS GENERALIZES
INFORMATION MODELER
SOFTWARE IMPLEMENTATION CAPABILITIES
CONSTRAINTS
DESIGN PATTERNS APPLICATION FRAMEWORKS
SOFTWARE IMPLEMENTER
IMPLEMENTATION TOOLS/ LEGACY CODE

Given this picture, we can characterize Structure Analysis/Structured Design (SASD) as starting the process by laying out requirements (the top left corner) and then using implementation technologies (the lower right corner) to build up design elements into a solution. The problem with this approach is that as requirements change many of the existing design solutions must be thrown away because they no longer satisfy the design problem at hand (i.e. they are *function-centric*). In other words, this approach works well when the requirements, design problems, and design solutions are well-known, but is not flexible for problems that are less well-understood and undergo active problem discovery.

On the other hand, object oriented techniques, when properly used, allow software developers to more easily preserve the conceptual integrity of design by 1) focusing on the information (i.e. *data-centric*) and 2) preserving familiar user abstractions from requirements to implementation. Another way of stating the basic difference between object-based approaches is:

Ask not what a system *does*, ask what it does it *to*.

READING ASSIGNMENT 12
Read Chapters 7 and 8 of [Cooper 1999].

Now, many techniques that espouse to be object-oriented are in fact a wolf in sheep's clothing. The most ravenous example unfortunately is CORBA and the CORBAservices. Using CORBA or CORBAservices often forces a process that results in an object-oriented encapsulation of SASD. Why is this so? Because it asserts the notion that encapsulation can be done without knowing the semantics or behavior of system elements, that is, just create an interface, and we will worry about the implementation later[1]. We have just defined functional decomposition in OO clothes. The fact that most people accept this as an OO technique is rooted in the culture of software and is nice described by [Cooper 1999].

Reuse Abuse

READING ASSIGNMENT 13
Read the paper by [Kruger 1992].

One of the most widely touted benefits of object oriented approaches is the ability to reuse *things*. However, it is unfair to talk about software reuse without really understanding what reuse is. To this end, a comprehensive discussion of reuse is in order and is given by [Kruger 1992] — enough said.

Accidental Complexity

An interesting phenomenon occurs when people try to combine existing software elements together — unexpected things happen. Many times we do not even know that these things happen until long after a system is deployed. The term for this phenomenon is called *accidental complexity* and was coined by [Schmidt and Fayad 1997]; this is the technical term for the problems associated with Structured Analysis/Design. They provide an excellent summary of how this complexity is introduced and how it might be dealt with.

READING ASSIGNMENT 14
Read the papers [Schmidt and Fayad 1997], [Snyder and Peck 1998].

Optional readings are [Wallace and Wallnau 1996] and [Kleindienst et al. 1996] and can be found in the Section "Supplemental Readings"

An example of how this complexity can be explicitly addressed by design is covered by [Snyder and Peck 1998]. The problem described in this paper discusses a fairly common integration problem, namely, how to construct a three-tier information system that uses both an ODBMS and CORBA as middleware. Several key integration problems are described with solutions.

Architectural Mismatch

Now that we have seen that significant difficulties can occur by reusing software elements, we need to better understand the reasons these difficulties exist. The paper by [Garlan et al. 1995] gives a good characterization of these associated issues and provides the appropriate term *architectural mismatch*.

READING ASSIGNMENT 15
Read the paper by [Garlan et al. 1995].

It is important to notice here that mismatch is discussed in the context of reusing existing elements, that is, software elements that actually exist. It is important to relate this perspective back to the working definition of software architecture in this course — something that exists in the world.

1. Specific, in-depth examples are needed to completely understand the issues and can be found in the optional READING ASSIGNMENT 14.

Reuse and Abstraction

Reuse occurs differently at different levels of abstraction in both design and implementation. Therefore, we need very different design and engineering techniques. For example, reusing a class is much different than reusing a design pattern as it is with patterns.

Reusing a design is much harder as the complexity of the reuse elements increases because the sheer number of element interactions to consider is larger. It should also be clear that architectures are *only* reusable if whole systems become a subsystem in something else, that is, a working system is integrated into a larger whole.

The Potential for Automation

It is possible to use software systems to help us formalize our approach to software development. The real question is where can automation help and on what kinds of problems. As it turns out, a particular kind of architectural style is incredibly common and relatively well understood namely the Shared Information System.[1]

The earliest example of a well-defined shared information system is the classical relational database client-server software systems. To help automate the retrieval of information in this environment, we can embed queries into programming language source code using the Embedded SQL facilities of the supplied database system. We can then use a preprocessor to turn the Embedded SQL into procedural calls of the native language and compile the source code into an executable. So, we have automated some aspect of the client's information pull from the database.

Shared Information System Evolution

As systems in the traditional client-server environment become more complex, *accidental complexity* was introduced because certain computations we put in the clients and were not available in the server[2]. A good example of this issue is the calculation of overtime pay for a payroll system. The computation of the overtime must be placed in all client applications rather than made available via the server even though we clearly want a universal notion of overtime pay to be maintained. If we change our calculation of overtime, we would like to change it in only in one place and make the new result available to all existing applications without requiring client modification.

1. Chapter 4 of [Shaw and Garlan 1996] cover this style in detail.
2. In fact, a term "impedance mismatch" has been coined to describe the accidental complexity.

In response to the the above problems, the next logical evolution of the client-server approach is to insert a level of indirection between the client and server to allow for each to have different rates of modification. Two important architectural styles have emerged from this perspective — N-tier architectures and mediated architectures [Wiederhold 1995]. Essentially, we create a "middle tier" that allows us to present functionality greater than the server itself can provide thereby allowing the overtime pay calculation to reside in a single place. However, we do not have full access to the server environment and cannot leverage the availability of the server's information to our advantage unless we explicitly build it in. So, we have a more maintainable system, but it is not as good as we might expect. While this approach significantly improves the robustness of the resulting system, accidental complexity is still introduced with this approach.

A Different Approach

While current technologies and approaches such as database systems and CORBA provide partial solutions for constructing shared information systems, a comprehensive and reliable approach is not currently available because many important details are not formally captured and are left as implementation exercises. Essentially, these solutions limit their scope to structural or syntactic descriptions. What is lacking is the focus on the semantic and behavioral content of information. To compensate for these deficiencies, many differing technologies are combined using either informal or *ad hoc* methods that naturally evolve into monolithic systems that are difficult to maintain or extend.

Current academic research has shown that to make the next "leap" in technology for building shared information systems, an approach based on more formal methods is needed because automation requires formalization. This research has resulted in a new technology area called *conceptual modeling environments* (CME) where, the basic approach is to provide an environment in which a *formal description* of a shared information model can be developed independently from underlying implementation technologies and software architectures (recall the software process in Figure 8). Once a model is developed, information servers can be automatically constructed that are *provably correct* and *targeted* towards specific software architectures and implementation technologies. Additionally, given such a model, sharing information with (or integrating) existing or new applications into the shared information base can also be automated using *formal integration specification languages* that complement the shared models.

READING ASSIGNMENT 16
Optionally read [Snyder and Muckelbauer 1999]. This paper provides an example of how this approach can be used in a practical application.

Because this topic is rather lengthy, the details of this approach are not described here. However, two important references provide more depth. First, [Snyder and Muckelbauer 1999] provides a description of how CMEs can be used in a specific kind of problem thereby providing some concreteness to the concepts. For a comprehensive description of this approach, see [Snyder 1998].

Architectural Style and Description Languages

To characterize systems "in the large", we need to bring a higher degree of formalism to the table. However, we are still rather primitive as a discipline in this respect. In the past few years, important progress has been made to help characterize overall system designs.

READING ASSIGNMENT 17
Optionally read Chapters 6 and 7 of [Shaw and Garlan 1996].

Before reading literature from this area, it is important to understand that two kinds of formalisms exist 1) ways of describing systems *as they are* (i.e. the natural world) and 2) describing systems as they *ought to be* (i.e. the artificial world — recall "The Science of Design" [Simon 1981]). As such, two classes of notations have emerged to address the above difference. Notations for *architectural style* capture the natural world while *architecture description languages* (ADLs) are intended to capture the artificial world. These concepts are covered in Chapters 6 and 7 of [Shaw and Garlan 1996].

The definitions provided in the current literature are still subject to ongoing research and development, and as such, do not have the clarity we would like to have as in other areas. For example, many of the accounts of "desirable" ADL features have a significant overlap with CMEs. But, CMEs are *not* ADLs. We do not have the space to prove that assertion here, but the careful practitioner will discover this difference by careful thought and reflection.

Concluding Thoughts

At this point you should be able to understand the difference between architecture, design, and engineering and why they are all important in realizing software systems. Additionally, you should be able to clearly articulate why using engineering techniques to solve a design problem is bad, and *vice versa*.

The pursuit of new software technologies and development techniques is essential to provide more design solution alternatives over time that are more cost-effective, reliable, and scalable. Additionally, these new technologies need to be motivated from a clear understanding of an overall real-world problem and how existing technologies fail to provide adequate solutions especially as the assumptions of existing solutions fail to hold true. Lastly, we as practitioners need to understand the complexities of the interaction between the physical world and the artificial world — never forgetting that the ultimate aim is to provide human-centric solutions to system design.

Annotated References

[Akin et al. 1995]

Akin, Ö., Sen, R., Donia, M., and Y. Zhang (1995). "SEED-Pro: Computer-Assisted Architectural Programming" *Journal of Architectural Engineering*. **1**(4). pp. 153-161.

This paper describes how building designers attempt to formalize some user requirements (called an architectural program) into specifications that can lead to a design problem.

[Bäumer et al. 1997]

Bäumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D., and H. Züllighoven (1997). "Framework Development for Large Systems" *Communications of the ACM*, **40**(10) pp. 52-59.

This paper provides an example of how to use abstraction and frameworks to provide flexible system development.

[Brooks 1975]

Brooks, F. (1975). *The Mythical Man-Month: Essays on Software Engineering*. Reading: Addison-Wesley.

This is by far the most insightful text about how software development gets really messed up and why. Additionally, the author explains rational approaches to getting people to work together productively in a software development effort. There is a second edition to this book that has an additional four chapters on more modern issues.

[Cooper 1999]

Cooper, A. (1999). *The Inmates Are Running the Asylum: Why High-Tech Products Drive Us Crazy and How to Restore the Sanity*. Indianapolis: Sams

This book is probably the only book that explains the business case for user-centered system development including specific citations on using design to preserve the conceptual integrity of a system. This book is critical for understanding how not think about software.

[Fayad and Schmidt 1997]

Fayad, M. and D. Schmidt (1997). "Object Oriented Application Frameworks" *Communications of the ACM*, **40**(10). pp. 85-87.

An excellent summary of the pitfalls and benefits encountered in try to build systems for reuse using object-oriented approaches.

[Flemming and Chien. 1995] Flemming, U. and S. Chien (1995). "Schematic Layout Design in SEED Environment" *Journal of Architectural Engineering*. **1**(4). pp 162-169.

This paper shows how that a building design can use a formal model of a schematic layout problem to generate a large number of design solutions (called layouts) that would not be possible if done by hand.

[Gamma et al. 1995] Gamma, E., Helm, R., Johnson, R. and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Reading: Addison-Wesley.

This book is the standard reference for 1) showing how design problems and design solutions are realized at certain levels of detail in the design process, and 2) how to catalog problem descriptions and solutions into an encyclopedic form.

[Garlan et al. 1995] Garlan, D., Allen, R., and J. Ockerbloom (1995). "Architectural Mismatch or Why It's Hard to Build Systems Out of Existing Parts" in *Proceedings of the 17th International conference on Software Engineering*, April 24 - 28, 1995, Seattle, WA USA. pp. 179-185.

This is an earlier version of a similar paper in IEEE Software, **12**(6), November 1995, called "Architectural Mismatch: Why Reuse is So Hard".

[Kruger 1992] Krueger, C. (1992). "Software Reuse" *ACM Computing Surveys*. **24**(2).

This is the definitive survey on software reuse to date.

[Larman 1998] Larman, C. (1998). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*.

This text is one of best texts I have seen that show how to use design patterns in the context of a UML environment to realize software solutions to many interesting problems. It is full of example patterns that are usable in everyday system development.

[Meyer 1988] Meyer, B. (1988). *Object Oriented Software Construction*. Prentice-Hall, Englewood-Cliffs, NJ.

If you are looking for a text on understanding what object-oriented programming and software construction is about, this is *the* book. The two most important concepts communicated in this book are *programming by contract* and how to use a language to support design/implementation under *programming by contract* approach.

[Neumann 1996] Neumann, P. G. (1996). "Linguistic Risks" *Communications of the ACM*, **39**(5), p. 154.

This short interlude on the use of language in computer science is quite neat.

[Schmidt and Fayad 1997] Schmidt, D. and M. Fayad (1997). “Lessons Learned Building Reusable OO Frameworks for Distributed Software” *Communications of the ACM*, **40**(10). pp. 85-87.

An excellent summary of the pitfalls and benefits encountered in try to build systems for reuse using object-oriented approaches.

[Shaw and Garlan 1996] Shaw, M. and D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall.

This text was developed as a result of a software architecture course at the Software Engineering Institute at Carnegie Mellon University. It is intended to be used in such a course, but stops short of how the covered concepts are used in everyday development.

[Simon 1981] Simon, H. A. (1981). *The Sciences of the Artificial (Second Edition)*. Cambridge: The MIT Press

This is an exceptional book that explains the difference between many different fields of study and how things that people make are treated very much differently from trying to describe the physical world as it is.

[Snyder and Peck 1998] Snyder, J. and C. Peck (1998). *PicoDDB*. ATL Internal Technical Report. Delivered under program contract for the DARPA-sponsored DDB program. December 24, 1998.

This document describes the design rationale for building a three-tier information system using CORBA as middleware for an OODMBS.

Supplemental Readings

[Clements and Northrop 1996] Clements, P. C. and L. M. Northrop (1996). Software Architecture: An Executive Summary. Technical Report CMU/SEI-96-TR-003. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. 15213.

[Kleindienst et al. 1996] Kleindienst, J., F. Plasil, and P. Tuma (1996). “Lessons Learned from Implementing the CORBA Persistent Object Service” *Proceedings OOPSLA ‘96. ACM SIGPLAN Notives*, **31**(10), pp 150-167. ACM Press, Reading, MA.

[Monroe et al. 1997] Monroe, R., Kompanek, A., Melton, R., and D. Garlan (1997). “Architectural Styles, Design Patterns, and Object-Oriented Programming” *IEEE Software*, **14**(1), January - February 1997.

[Meyer 1997]	Meyer, B. (1997). *Object Oriented Software Construction: Second Edition*. Prentice-Hall, Englewood-Cliffs, NJ.
[Snyder 1998]	Snyder, J. (1998). *Conceptual Modeling and Application Integration in CAD: The Essential Elements*. Ph. D. Dissertation in Computational Design. Department of Architecture and Engineering Design Research Center. Carnegie Mellon University. Pittsburgh, PA.
[Snyder and Muckelbauer 1999]	Snyder, J. and A. Muckelbauer (1999). *Dynamic Data Distribution in Model-Based Battle Command*. ATIRP Factor 3a Research Plan. LM-ATL Technical Report.
[Wallace and Wallnau 1996]	Wallace, E. and K. Wallnau (1996). “A Situated Evaluation of the Object Management Group’s (OMG) Object Management Architecture (OMA)”. *Proceedings OOPSLA ‘96. ACM SIGPLAN Notives*, **31**(10), pp 168-178. ACM Press, Reading, MA
[Wiederhold 1995]	Wiederhold, G. (1995). “Mediation in Information Systems” *ACM Computing Surveys*, **27**(2), New York: ACM Press, pp. 265-267.