UNIT 1:

1.0 Introduction:

Software Architecture:

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

2.0 The Architecture Business Cycle:

Definition: Architecture Business Cycle (ABC):

"Software architecture is a result of technical, business, and social influences. Its existence in turn affects the technical, business, and social environments that subsequently influence future architectures. We call this cycle of influences, from the environment to the architecture and back to the environment, the Architecture Business Cycle (ABC)."

1.The organization goals of **Architecture Business Cycle** are beget requirements, which beget an architecture, which begets a system. The architecture flows from the architect's experience and the technical environment of the day.

2.Three things required for ABC are as follows:

i. Case studies of successful architectures crafted to satisfy demanding requirements, so as to help set the technical playing field of the day.

ii. Methods to assess an architecture before any system is built from it, so as to mitigate the risks associated with launching unprecedented designs.

iii.Techniques for incremental architecture-based development, so as to uncover design flaws before it is too late to correct them.

2.1 How the ABC Works :

1. The architecture affects the structure of the developing organization. An architecture prescribes a structure for a system; as we will see, it particularly prescribes the units of software that must be implemented (or otherwise obtained)

and integrated to form the system. These units are the basis for the development project's structure. Teams are formed for individual software units; and the development, test, and integration activities all revolve around the units. Likewise, schedules and budgets allocate resources in chunks corresponding to the units. If a company becomes adept at building families of similar systems, it will tend to invest in each team by nurturing each area of expertise. Teams become embedded in the organization's structure. This is feedback from the architecture to the developing organization.

In the software product line case study, separate groups were given responsibility for building and maintaining individual portions of the organization's architecture for a family of products. In any design undertaken by the organization at large, these groups have a strong voice in the system's decomposition, pressuring for the continued existence of the portions they control.

2. The architecture can affect the goals of the developing organization. A successful system built from it can enable a company to establish a foothold in a particular market area. The architecture can provide opportunities for the efficient production and deployment of similar systems, and the organization may adjust its goals to take advantage of its newfound expertise to plumb the market. This is feedback from the system to the developing organization and the systems it builds.
3. The architecture can affect customer requirements for the next system by giving the customer the opportunity to receive a system (based on the same architecture) in a more reliable, timely, and economical manner than if the subsequent system were to be built from scratch. The customer may be willing to relax some requirements to gain these economies. Shrink-wrapped software has clearly affected people's requirements by providing solutions that are not tailored to their precise needs but are instead inexpensive and (in the best of all possible worlds) of high quality. Product lines have the same effect on customers who cannot be so flexible with their requirements. A Case Study in Product Line Development will show how a product line architecture caused customers to happily compromise their requirements because they could get high-quality software that fit their basic needs quickly, reliably, and at lower cost.
4. The process of system building will affect the architect's experience with subsequent systems by adding to the corporate experience base. A system that was successfully built around a tool bus or .NET or encapsulated finite-state machines will engender similar systems built the same way in the future. On the other hand, architectures that fail are less likely to be chosen for future projects.
5. A few systems will influence and actually change the software engineering culture, that is, the technical environment in which system builders operate and learn. The first relational databases, compiler generators, and table-driven operating systems had this effect in the 1960s and early 1970s; the first spreadsheets and windowing systems, in the 1980s. The World Wide Web is the example for the 1990s. J2EE may be the example for the first decade of the twenty-first century. When such pathfinder systems are constructed, subsequent

systems are affected by their legacy.

These and other feedback mechanisms form what we call the ABC, illustrated in Figure 1.4, which depicts the influences of the culture and business of the development organization on the software architecture. That architecture is, in turn, a primary determinant of the properties of the developed system or systems. But the ABC is also based on a recognition that shrewd organizations can take advantage of the organizational and experiential effects of developing an architecture and can use those effects to position their business strategically for future projects.

2.2 Where Do Architectures Come From?

1. An architecture is the result of a set of business and technical decisions.
2. There are many influences at work in its design, and the realization of these influences will change depending on the environment in which the architecture is required to perform.
3. An architect designing a system for which the real-time deadlines are believed to be tight will make one set of design choices; the same architect, designing a similar system in which the deadlines can be easily satisfied, will make different choices.
4. And the same architect, designing a non-real-time system, is likely to make quite different choices still.
5. Even with the same requirements, hardware, support software, and human resources available, an architect designing a system today is likely to design a different system than might have been designed five years ago.
6. In any development effort, the requirements make explicit some—but only some—of the desired properties of the final system.
7. Not all requirements are concerned directly with those properties; a development process or the use of a particular tool may be mandated by them.
8. But the requirements specification only begins to tell the story.
9. Failure to satisfy other constraints may render the system just as problematic as if it functioned poorly.

2.3 Building the ABC:

Building the ABC is done by identifying the influences to and from architectures as follows:

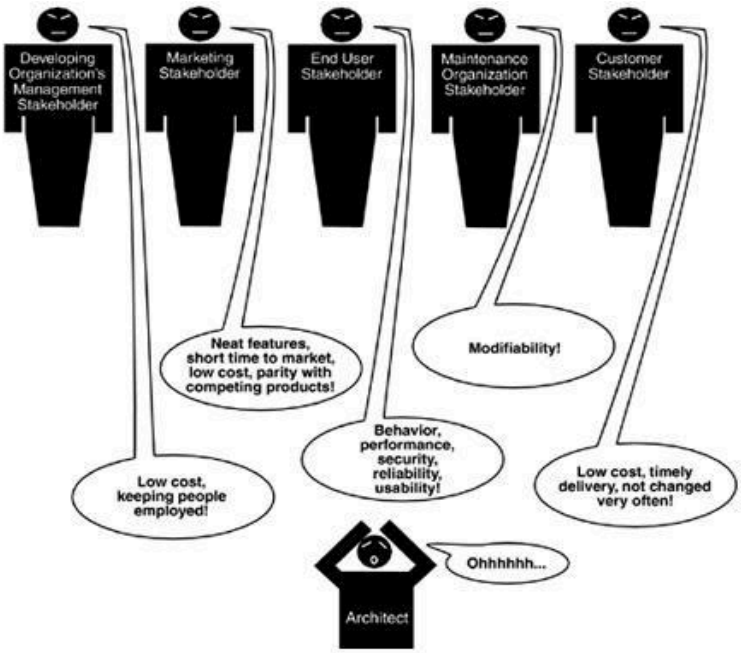
2.3.1 ARCHITECTURES ARE INFLUENCED BY SYSTEM STAKEHOLDERS:

1. Many people and organizations are interested in the construction of a software system.
2. **Stakeholders are**:
    - The customer,
    - the end users,
    - the developers,
    - the project manager,

- the maintainers, and
- even those who market the system.

3. **Stakeholders have different concerns that they wish the system to guarantee or optimize, including things as diverse as providing a certain behavior at runtime, performing well on a particular piece of hardware, being easy to customize, achieving short time to market or low cost of development, gainfully employing programmers who have a particular specialty, or providing a broad range of functions.**
4. Figure 1.2 shows the architect receiving helpful stakeholder "**suggestions.**"

Figure 1.2. Influence of stakeholders on the architect



5. Having an acceptable system involves properties such as performance, reliability, availability, platform compatibility, memory utilization, network usage, security, modifiability, usability, and interoperability with other systems as well as behavior.
6. Indeed, we will see that these properties determine the overall design of the architecture.
7. All of them, and others, affect how the delivered system is viewed by its eventual recipients, and so they find a voice in one or more of the system's stakeholders.
8. The underlying problem, of course, is that each stakeholder has different concerns and goals, some of which may be contradictory.
9. Properties can be listed and discussed, of course, in an artifact such as a requirements document.
10. But it is a rare requirements document that does a good job of capturing all of a system's quality requirements in testable detail.
11. The reality is that the architect often has to fill in the blanks and mediate the

conflicts.

2.3.2 ARCHITECTURES ARE INFLUENCED BY THE DEVELOPING ORGANIZATION:

1. In addition to the organizational goals expressed through requirements, an architecture is influenced by the structure or nature of the development organization.
2. For example, if the organization has an abundance of idle programmers skilled in client-server communications, then a client-server architecture might be the approach supported by management.
3. If not, it may well be rejected. Staff skills are one additional influence, but so are the development schedule and budget.

There are three classes of influence that come from the developing organization: **immediate business, long-term business, and organizational structure**.

- An organization may have an immediate business investment in certain assets, such as existing architectures and the products based on them. The foundation of a development project may be that the proposed system is the next in a sequence of similar systems, and the cost estimates assume a high degree of asset re-use.
- An organization may wish to make a long-term business investment in an infrastructure to pursue strategic goals and may view the proposed system as one means of financing and extending that infrastructure.
- The organizational structure can shape the software architecture. In the case study in Chapter 8 (Flight Simulation: A Case Study in Architecture for Integrability), the development of some of the subsystems was subcontracted because the subcontractors provided specialized expertise. This was made possible by a division of functionality in the architecture that allowed isolation of the specialities.

2.3.3 ARCHITECTURES ARE INFLUENCED BY THE BACKGROUND AND EXPERIENCE OF THE ARCHITECTS:

1. If the architects for a system have had good results using a particular architectural approach, such as distributed objects or implicit invocation, chances are that they will try that same approach on a new development effort.
2. Conversely, if their prior experience with this approach was disastrous, the architects may be reluctant to try it again.
3. Architectural choices may also come from an architect's education and training, exposure to successful architectural patterns, or exposure to systems that have worked particularly poorly or particularly well.
4. The architects may also wish to experiment with an architectural pattern or technique learned from a book (such as this one) or a course.

2.3.4 ARCHITECTURES ARE INFLUENCED BY THE TECHNICAL ENVIRONMENT:

1. A special case of the architect's background and experience is reflected by the technical environment.

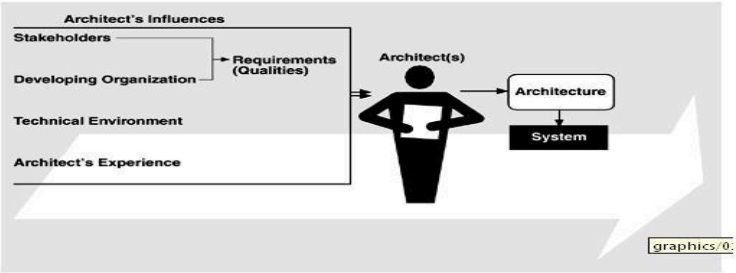
2. The environment that is current when an architecture is designed will influence that architecture.
3. It might include standard industry practices or software engineering techniques prevalent in the architect's professional community.
4. It is a brave architect who, in today's environment, does not at least consider a Web-based, object-oriented, middleware-supported design for an information system.

2.3.5 RAMIFICATIONS OF INFLUENCES ON AN ARCHITECTURE:

1. Influences on an architecture come from a wide variety of sources. Some are only implied, while others are explicitly in conflict.
2. Almost never are the properties required by the business and organizational goals consciously understood, let alone fully articulated.
3. Indeed, even customer requirements are seldom documented completely, which means that the inevitable conflict among different stakeholders' goals has not been resolved.
4. However, architects need to know and understand the nature, source, and priority of constraints on the project as early as possible.
5. Therefore, they must identify and actively engage the stakeholders to solicit their needs and expectations.
6. Without such engagement, the stakeholders will, at some point, demand that the architects explain why each proposed architecture is unacceptable, thus delaying the project and idling workers.
7. Early engagement of stakeholders allows the architects to understand the constraints of the task, manage expectations, negotiate priorities, and make tradeoffs.
8. Architecture reviews (covered in Part Three) and iterative prototyping are two means for achieving it.
9. It should be apparent that the architects need more than just technical skills.
10. Explanations to one stakeholder or another will be required regarding the chosen priorities of different properties and why particular stakeholders are not having all of their expectations satisfied.
11. For an effective architect, then, diplomacy, negotiation, and communication skills are essential.

The influences on the architect, and hence on the architecture, are shown in Figure 1.3. Architects are influenced by the requirements for the product as derived from its stakeholders, the structure and goals of the developing organization, the available technical environment, and their own background and experience.

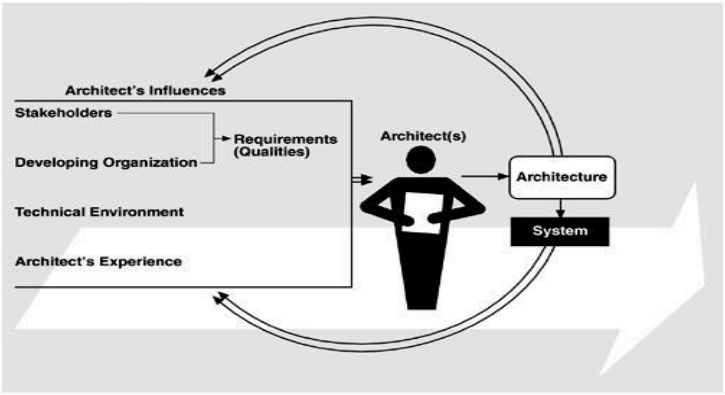
Figure 1.3. Influences on the architecture



2.3.6 THE ARCHITECTURES AFFECT THE FACTORS THAT INFLUENCE THEM:

1. The main message of this book is that the relationships among business goals, product requirements, architects' experience, architectures, and fielded systems form a cycle with feedback loops that a business can manage.
2. A business manages this cycle to handle growth, to expand its enterprise area, and to take advantage of previous investments in architecture and system building. Figure 1.4 shows the feedback loops.
3. Some of the feedback comes from the architecture itself, and some comes from the system built from it.

Figure 1.4. The Architecture Business Cycle



2.4 Software Processes and the Architecture Business Cycle:

Software process is the term given to the organization, ritualization, and management of software development activities. What activities are involved in creating a software architecture, using that architecture to realize a design, and then implementing or

managing the evolution of a target system or application? These activities include the following:

- Creating the business case for the system
- Understanding the requirements
- Creating or selecting the architecture
- Documenting and communicating the architecture
- Analyzing or evaluating the architecture
- Implementing the system based on the architecture
- Ensuring that the implementation conforms to the architecture

2.4.1 ARCHITECTURE ACTIVITIES:

As indicated in the structure of the ABC, architecture activities have comprehensive feedback relationships with each other.

2.4.1.1. Creating the Business Case for the System:

1. Creating a business case is broader than simply assessing the market need for a system. It is an important step in creating and constraining any future requirements like:

- How much should the product cost?
- What is its targeted market?
- What is its targeted time to market?
- Will it need to interface with other systems?
- Are there system limitations that it must work within?

2. These are all questions that must involve the system's architects. They cannot be decided solely by an architect, but if an architect is not consulted in the creation of the business case, it may be impossible to achieve the business goals.

2.4.1.2. Understanding the Requirements:

There are a variety of techniques for eliciting requirements from the stakeholders:

- Object-oriented analysis uses scenarios, or "use cases" to embody requirements.
- Safety-critical systems use more rigorous approaches, such as finite-state-machine models or formal specification languages.
- Collection of quality attribute scenarios that support the capture of quality requirements for a system.
- Another technique that helps us understand requirements is the creation of prototypes. Prototypes may help to model desired behavior, design the user interface, or analyze resource utilization. This helps to make the system "real" in the eyes of its stakeholders and can quickly catalyze decisions on the system's design and the design of its user interface.

2.4.1.3. Creating or Selecting the Architecture:

Conceptual integrity is the key to sound system design and that conceptual integrity can only be had by a small number of minds coming together to design the system's architecture.

2.4.1.4. Communicating the Architecture:

- For the architecture to be effective as the backbone of the project's design, it must be communicated **clearly and unambiguously to all of the stakeholders**.
- Developers must understand the work assignments it requires of them, testers must understand the task structure it imposes on them, management must understand the scheduling implications it suggests.
- The architecture's documentation should be informative, unambiguous, and readable by many people with varied backgrounds.

2.4.1.5. Analyzing or Evaluating the Architecture:

- In any design process there will be multiple candidate designs considered.
- Some will be rejected immediately. Others will contend for primacy.
- Choosing among these competing designs in a rational way is one of the architect's greatest challenges.
- Evaluating an architecture for the qualities that it supports is essential to ensuring that the system constructed from that architecture satisfies its stakeholders' needs. Becoming more widespread are analysis techniques to evaluate the quality attributes that an architecture imparts to a system.

2.4.1.6. Implementing Based on the Architecture:

- This activity is concerned with keeping the developers faithful to the structures and interaction protocols constrained by the architecture.
- Having an explicit and well-communicated architecture is the first step toward ensuring architectural conformance.
- Having an environment or infrastructure that actively assists developers in creating and maintaining the architecture (as opposed to just the code) is better.

2.4.1.7.Ensuring Conformance to an Architecture:

- Finally, when an architecture is created and used, it goes into a maintenance phase.
- Constant vigilance is required to ensure that the actual architecture and its representation remain faithful to each other during this phase.
- Although work in this area is comparatively immature, there has been intense activity in recent years.

3.0 What Is Software Architecture?

Architecture plays a pivotal role in allowing an organization to meet its business goals. Architecture commands a price (the cost of its careful development), but it pays for itself handsomely by enabling the organization to achieve its system goals and expand its software capabilities. Architecture is an asset that holds tangible value to the developing organization beyond the project for which it was created.

3.1 Requirements Of Software Architecture

1.Actually architecture is a set of components and connections among them and it should satisfy following requirements like:

- **What is the nature of the elements?**
- **What are the responsibilities of the elements?**
- **What is the significance of the connections?**
- **What is the significance of the layout?**

We must raise these questions because unless we know precisely what the elements are and how they cooperate to accomplish the purpose of the system.

1. Definition Of Software Architecture:

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

1. **Externally visible properties** are those assumptions other elements can make of an element, such as:
   - its provided services,
   - performance characteristics,
   - fault handling,
   - shared resource usage, and so on.

2. **Architecture defines software elements.**

- The architecture embodies information about how the elements relate to each other.
- This means that it specifically omits certain information about elements that does not pertain to their interaction.
- Thus, an architecture is foremost an abstraction of a system that suppresses details of elements that do not affect how they use, are used by, relate to, or interact with other elements.
- In nearly all modern systems, elements interact with each other by means of interfaces that partition details about an element into public and private parts. Architecture is concerned with the public side of this division; private details—those having to do solely with internal implementation—are not architectural.

3. The definition makes clear that systems can and do comprise more than one structure and that no one structure can irrefutably claim to be the architecture.

- For example, all nontrivial projects are partitioned into implementation units; these units are given specific responsibilities and are frequently the basis of work assignments for programming teams.
- This type of element comprises programs and data that software in other implementation units can call or access, and programs and data that are private.

- In large projects, these elements are almost certainly subdivided for assignment to subteams.
- This is one kind of structure often used to describe a system.
- It is very static in that it focuses on the way the system's functionality is divided up and assigned to implementation teams.
- Other structures are much more focused on the way the elements interact with each other at runtime to carry out the system's function.
- Suppose the system is to be built as a set of parallel processes.
- The processes that will exist at runtime, the programs in the various implementation units described previously that are strung together sequentially to form each process, and the synchronization relations among the processes form another kind of structure often used to describe a system.
- Are any of these structures alone the architecture?
- No, although they all convey architectural information.
- The architecture consists of these structures as well as many others.
- This example shows that since architecture can comprise more than one kind of structure, there is more than one kind of element (e.g., implementation unit and processes), more than one kind of interaction among elements (e.g., subdivision and synchronization), and even more than one context (e.g., development time versus runtime). By intention, the definition does not specify what the architectural elements and relationships are. Is a software element an object? A process? A library? A database? A commercial product? It can be any of these things and more.

4. The architecture defines relationship: Every computing system with software has a software architecture because every system can be shown to comprise elements and the relations among them.

- In the most trivial case, a system is itself a single element—uninteresting and probably nonuseful but an architecture nevertheless.
- Even though every system has an architecture, it does not necessarily follow that the architecture is known to anyone.
- Perhaps all of the people who designed the system are long gone, the documentation has vanished (or was never produced), the source code has been lost (or was never delivered), and all we have is the executing binary code.
- This reveals the difference between the architecture of a system and the representation of that architecture.
- Unfortunately, an architecture can exist independently of its description or specification, which raises the importance of architecture documentation

5.**The behavior of each element** is part of the architecture in so far as that behavior can be observed or discerned from the point of view of another element.

- Such behavior is what allows elements to interact with each other, which is clearly part of the architecture.

- This is another reason that the box-and-line drawings that are passed off as architectures are not architectures at all.
- They are simply box-and-line drawings—or, to be more charitable, they serve as cues to provide more information that explains what the elements shown actually do.
- When looking at the names of the boxes (database, graphical user interface, executive, etc.), a reader may well imagine the functionality and behavior of the corresponding elements.
- This mental image approaches an architecture, but it springs from the observer's mind and relies on information that is not present.
- We do not mean that the exact behavior and performance of every element must be documented in all circumstances; however, to the extent that an element's behavior influences how another element must be written to interact with it or influences the acceptability of the system as a whole, this behavior is part of the software architecture.

Finally, the definition is indifferent as to whether the architecture for a system is a good one or a bad one, meaning that it will allow or prevent the system from meeting its behavioral, performance, and life-cycle requirements. We do not accept trial and error as the best way to choose an architecture for a system—that is, picking an architecture at random, building the system from it, and hoping for the best—so this raises the importance of architecture evaluation and architecture design.

3.2.1 Various Definitions of Software Architecture:

- ***Architecture is high-level design***. This is true enough, in the sense that a horse is a mammal, but the two are not interchangeable. Other tasks associated with design are not architectural, such as deciding on important data structures that will be encapsulated. The interface to those data structures is decidedly an architectural concern, but their actual choice is not.
- ***Architecture is the overall structure of the system***. This common refrain implies (incorrectly) that systems have but one structure. We know this to be false, and, if someone takes this position, it is usually entertaining to ask which structure they mean. The point has more than pedagogic significance. As we will see later, the different structures provide the critical engineering leverage points to imbue a system with the quality attributes that will render it a success or failure. The multiplicity of structures in an architecture lies at the heart of the concept.
- ***Architecture is the structure of the components of a program or system, their interrelationships, and the principles and guidelines governing their design and evolution over time.*** This is one of a number of process-centered definitions that include ancillary information such as principles and guidelines. Many people claim that architecture includes a statement of stakeholder needs and a rationale for how those needs are met. We agree that gathering such information is essential and a matter of good professional practice. However, we do not consider them part of the architecture per se any more than an owner's manual for a car is part of

the car. Any system has an architecture that can be discovered and analyzed independently of any knowledge of the process by which the architecture was designed or evolved.

- ***Architecture is components and connectors***. Connectors imply a runtime mechanism for transferring control and data around a system. Thus, this definition concentrates on the runtime architectural structures. A UNIX pipe is a connector, for instance. This makes the non-runtime architectural structures (such as the static division into responsible units of implementation discussed earlier) second-class citizens. They aren't second class but are every bit as critical to the satisfaction of system goals. When we speak of "relationships" among elements, we intend to capture both runtime and non-runtime relationships.

4.0 Why Is Software Architecture Important?

There are fundamentally three reasons for software architecture's importance:

1. **Communication among stakeholders**. Software architecture represents a common abstraction of a system that most if not all of the system's stakeholders can use as a basis for mutual understanding, negotiation, consensus, and communication.
2. **Early design decisions**. Software architecture manifests the earliest design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its deployment, and its maintenance life. It is also the earliest point at which design decisions governing the system to be built can be analyzed.
3. **Transferable abstraction of a system**. Software architecture constitutes a relatively small, intellectually graspable model for how a system is structured and how its elements work together, and this model is transferable across systems. In particular, it can be applied to other systems exhibiting similar quality attribute and functional requirements and can promote large-scale re-use.

We will address each of these points in turn.

1. Communication among Stakeholders:

1. Each stakeholder of a software system—customer, user, project manager, coder, tester, and so on—is concerned with different system characteristics that are affected by the architecture.
2. For example, the user is concerned that the system is reliable and available when needed; the customer is concerned that the architecture can be implemented on schedule and to budget; the manager is worried (as well as about cost and schedule) that the architecture will allow teams to work largely independently, interacting in disciplined and controlled ways. The architect is worried about strategies to achieve all of those goals.

Architecture provides a common language in which different concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable even for large, complex systems (see the sidebar What Happens When I Push This Button?). Without such a language, it is difficult to understand large systems sufficiently to make the early

decisions that influence both quality and usefulness.

2. Early design decisions:

Software architecture represents a system's earliest set of design decisions. These early decisions are the most difficult to get correct and the hardest to change later in the development process, and they have the most far-reaching effects.

The Architecture Defines Constraints on Implementation

1. An implementation exhibits an architecture if it conforms to the structural design decisions described by the architecture.
2. This means that the implementation must be divided into the prescribed elements, the elements must interact with each other in the prescribed fashion, and each element must fulfill its responsibility to the others as dictated by the architecture.
3. Resource allocation decisions also constrain implementations.
4. These decisions may be invisible to implementors working on individual elements.
5. The constraints permit a separation of concerns that allows management decisions to make the best use of personnel and computational capacity.
6. Element builders must be fluent in the specification of their individual elements but not in architectural tradeoffs.
7. Conversely, architects need not be experts in all aspects of algorithm design or the intricacies of the programming language, but they are the ones responsible for the architectural tradeoffs.

The Architecture Dictates Organizational Structure

1. Not only does architecture prescribe the structure of the system being developed, but that structure becomes engraved in the structure of the development project (and sometimes, the structure of the entire organization).
2. The normal method for dividing up the labor in a large system is to assign different groups different portions of the system to construct.
3. This is called the work breakdown structure of a system.
4. Because the system architecture includes the highest-level decomposition of the system, it is typically used as the basis for the work breakdown structure, which in turn dictates units of planning, scheduling, and budget; interteam communication channels; configuration control and file system organization; integration and test plans and procedures; and even minutiae such as how the project intranet is organized and how many team picnics there are.
5. Teams communicate with each other in terms of the interface specifications to the major elements.
6. The maintenance activity, when launched, will also reflect the software structure, with teams formed to maintain specific structural elements.

A side effect of establishing the work breakdown structure is to freeze some aspects of the software architecture. A group that is responsible for one of the subsystems will resist having its responsibilities distributed across other groups. If these responsibilities have

been formalized in a contractual relationship, changing them can become expensive. Tracking progress on a collection of tasks being distributed also becomes much more difficult.

Once the architecture has been agreed on, then, it becomes almost impossible, for managerial and business reasons, to modify it. This is one argument (among many) for carrying out a comprehensive evaluation before freezing the software architecture for a large system.

The Architecture Inhibits or Enables a System's Quality Attributes

Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture. The relationship between architectures and quality is as follows:

- If your system requires high performance, you need to manage the time-based behavior of elements and the frequency and volume of inter-element communication.
- If modifiability is important, you need to assign responsibilities to elements such that changes to the system do not have far-reaching consequences.
- If your system must be highly secure, you need to manage and protect inter-element communication and which elements are allowed to access which information. You may also need to introduce specialized elements (such as a trusted kernel) into the architecture.
- If you believe scalability will be needed in your system, you have to carefully localize the use of resources to facilitate the introduction of higher-capacity replacements.
- If your project needs to deliver incremental subsets of the system, you must carefully manage inter-component usage.
- If you want the elements of your system to be re-usable in other systems, you need to restrict inter-element coupling so that when you extract an element it does not come out with too many attachments to its current environment to be useful.

The strategies for these and other quality attributes are supremely architectural. It is important to understand, however, that architecture alone cannot guarantee functionality or quality. Poor downstream design or implementation decisions can always undermine an adequate architectural design. Decisions at all stages of the life cycle—from high-level design to coding and implementation—affect system quality. Therefore, quality is not completely a function of architectural design. To ensure quality, a good architecture is necessary, but not sufficient.

Predicting System Qualities by Studying the Architecture

Is it possible to tell that the appropriate architectural decisions have been made (i.e., if the system will exhibit its required quality attributes) without waiting until the system is developed and deployed? If the answer were no, choosing an architecture would be a hopeless task—random selection would perform as well as any other method.

Fortunately, it is possible to make quality predictions about a system based solely on an evaluation of its architecture. Architecture evaluation techniques such as the Architecture Tradeoff Analysis Method support top-down insight into the attributes of software product quality that is made possible (and constrained) by software architectures.

The Architecture Makes It Easier to Reason about and Manage Change

1. The software development community is coming to grips with the fact that roughly 80 percent of a typical software system's cost occurs after initial deployment.
2. A corollary of this statistic is that most systems that people work on are in this phase.
3. Many if not most programmers and designers never work on new development—they work under the constraints of the existing body of code.
4. Software systems change over their lifetimes; they do so often and often with difficulty.
5. Every architecture partitions possible changes into three categories: local, nonlocal, and architectural.
6. A local change can be accomplished by modifying a single element.
7. A nonlocal change requires multiple element modifications but leaves the underlying architectural approach intact.
8. An architectural change affects the fundamental ways in which the elements interact with each other—the pattern of the architecture—and will probably require changes all over the system.
9. Obviously, local changes are the most desirable, and so an effective architecture is one in which the most likely changes are also the easiest to make.
10. Deciding when changes are essential, determining which change paths have the least risk, assessing the consequences of proposed changes, and arbitrating sequences and priorities for requested changes all require broad insight into relationships, performance, and behaviors of system software elements.
11. These are in the job description for an architect.
12. Reasoning about the architecture can provide the insight necessary to make decisions about proposed changes.

The Architecture Helps in Evolutionary Prototyping

Once an architecture has been defined, it can be analyzed and prototyped as a skeletal system. This aids the development process in two ways.

1. The system is executable early in the product's life cycle. Its fidelity increases as prototype parts are replaced by complete versions of the software. These prototype parts can be a lower-fidelity version of the final functionality, or they can be surrogates that consume and produce data at the appropriate rates.
2. A special case of having the system executable early is that potential performance

problems can be identified early in the product's life cycle.

Each of these benefits reduces the risk in the project. If the architecture is part of a family of related systems, the cost of creating a framework for prototyping can be distributed over the development of many systems.

The Architecture Enables More Accurate Cost and Schedule Estimates

1. Cost and schedule estimates are an important management tool to enable the manager to acquire the necessary resources and to understand whether a project is in trouble.
2. Cost estimations based on an understanding of the system pieces are, inherently, more accurate than those based on overall system knowledge.
3. As we have said, the organizational structure of a project is based on its architecture.
4. Each team will be able to make more accurate estimates for its piece than a project manager will and will feel more ownership in making the estimates come true.
5. Second, the initial definition of an architecture means that the requirements for a system have been reviewed and, in some sense, validated.
6. The more knowledge about the scope of a system, the more accurate the estimates.

3. ARCHITECTURE AS A TRANSFERABLE, RE-USABLE MODEL:

The earlier in the life cycle re-use is applied, the greater the benefit that can be achieved. While code re-use is beneficial, re-use at the architectural level provides tremendous leverage for systems with similar requirements. Not only code can be re-used but so can the requirements that led to the architecture in the first place, as well as the experience of building the re-used architecture. When architectural decisions can be re-used across multiple systems, all of the early decision consequences we just described are also transferred.

Software Product Lines Share a Common Architecture

1. A software product line or family is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.
2. Chief among these core assets is the architecture that was designed to handle the needs of the entire family.
3. Product line architects choose an architecture (or a family of closely related architectures) that will serve all envisioned members of the product line by making design decisions that apply across the family early and by making other decisions that apply only to individual members late.
4. The architecture defines what is fixed for all members of the product line and what is variable.

5. Software product lines represent a powerful approach to multi-system development that shows order-of-magnitude payoffs in time to market, cost, productivity, and product quality.
6. The power of architecture lies at the heart of the paradigm.
7. Similar to other capital investments, the architecture for a product line becomes a developing organization's core asset.

Systems Can Be Built Using Large, Externally Developed Elements

1. Whereas earlier software paradigms focused on programming as the prime activity, with progress measured in lines of code, architecture-based development often focuses on composing or assembling elements that are likely to have been developed separately, even independently, from each other.
2. This composition is possible because the architecture defines the elements that can be incorporated into the system.
3. It constrains possible replacements (or additions) according to how they interact with their environment, how they receive and relinquish control, what data they consume and produce, how they access data, and what protocols they use for communication and resource sharing.
4. One key aspect of architecture is its organization of element structure, interfaces, and operating concepts.
5. The most significant principle of this organization is interchangeability.
6. In 1793, Eli Whitney's mass production of muskets, based on the principle of interchangeable parts, signaled the dawn of the Industrial Age.
7. In the days before reliable physical measurements, this was a daunting notion. Today in software, until abstractions can be reliably delimited, the notion of structural interchangeability is just as daunting and just as significant.
8. Commercial off-the-shelf components, subsystems, and compatible communications interfaces all depend on the principle of interchangeability.
9. However, there is much about software development through composition that remains unresolved.
10. When the components that are candidates for importation and re-use are distinct subsystems that have been built with conflicting architectural assumptions, unanticipated complications can increase the effort required to integrate their functions. David Garlan and his colleagues coined the term architectural mismatch to describe this situation.

Less Is More: It Pays to Restrict the Vocabulary of Design Alternatives

1. As useful architectural patterns and design patterns are collected, it becomes clear that, although computer programs can be combined in more or less infinite ways, there is something to be gained by voluntarily restricting ourselves to a relatively small number of choices when it comes to program cooperation and interaction.
2. That is, we wish to minimize the design complexity of the system we are building.
3. Advantages to this approach include enhanced re-use, more regular and simpler designs that are more easily understood and communicated, more capable analysis, shorter selection time, and greater interoperability.
4. Properties of software design follow from the choice of architectural pattern.

5. Patterns that are more desirable for a particular problem should improve the implementation of the resulting design solution, perhaps by making it easier to arbitrate conflicting design constraints, by increasing insight into poorly understood design contexts, and/or by helping to surface inconsistencies in requirements specifications.

An Architecture Permits Template-Based Development

An architecture embodies design decisions about how elements interact that, while reflected in each element's implementation, can be localized and written just once. Templates can be used to capture in one place the inter-element interaction mechanisms. For instance, a template can encode the declarations for an element's public area where results will be left, or can encode the protocols that the element uses to engage with the system executive.

An Architecture Can Be the Basis for Training

The architecture, including a description of how elements interact to carry out the required behavior, can serve as the introduction to the system for new project members. This reinforces our point that one of the important uses of software architecture is to support and encourage communication among the various stakeholders. The architecture is a common reference point.

5.0 System Architecture versus Software Architecture:

1. In creating a software architecture, system considerations are seldom absent.
2. For example, if you want an architecture to be high performance, you need to have some idea of the physical characteristics of the hardware platforms that it will run on (CPU speed, amount of memory, disk access speed) and the characteristics of any devices that the system interfaces with (traditional I/O devices, sensors, actuators), and you will also typically be concerned with the characteristics of the network (primarily bandwidth).
3. If you want an architecture that is highly reliable, again you will be concerned with the hardware, in this case with its failure rates and the availability of redundant processing or network devices. On it goes. Considerations of hardware are seldom far from the mind of the architect.
4. So, when you design a software architecture, you will probably need to think about the entire system—the hardware as well as the software. To do otherwise would be foolhardy.
5. No engineer can be expected to make predictions about the characteristics of a system when only part of that system is specified.
6. But still we persist in speaking about software architecture primarily, and not system architecture.
7. Why is this? Because most of the architect's freedom is in the software choices, not in the hardware choices.
8. It is not that there are no hardware choices to be made, but these may be out of the architect's control (for example, when creating a system that needs to work on

arbitrary client machines on the Internet) or specified by others (for reasons of economics, legal issues, or compliance with standards); or they will likely change over time.

9. For this reason, we feel justified in focusing on the software portion of architecture, for this is where the most fundamental decisions are made, where the greatest freedoms reside, and where there are the greatest opportunities for success (or disaster!).

1. Documenting Software Architecture:

Documenting the architecture is the crowning step to crafting it. Even a perfect architecture is useless if no one understands it or (perhaps worse) if key stakeholders misunderstand it. If you go to the trouble of creating a strong architecture, you must describe it in sufficent detail, without ambiguity, and organized in such a way that others can quickly find needed information. Otherwise, your effort will have been wasted because the architecture will be unusable.

The architecture for a system depends on the requirements levied on it, so too does the documentation for an architecture depend on the requirements levied on it—that is, how we expect it will be used. Documentation is decidedly not a case of "one size fits all." It should be sufficiently abstract to be quickly understood by new employees but sufficiently detailed to serve as a blueprint for analysis. The architectural documentation for, say, security analysis may well be different from the architectural documentation we would hand to an implementor. And both of these will be different from what we put in a new hire's familiarization reading list.

Architecture documentation is both prescriptive and descriptive. That is, for some audiences it prescribes what should be true by placing constraints on decisions to be made. For other audiences it describes what is true by recounting decisions already made about a system's design.

All of this tells us that different stakeholders for the documentation have different needs—different kinds of information, different levels of information, and different treatments of information.

This might mean producing different documents for different stakeholders. More likely, it means producing a single documentation suite with a roadmap that will help different stakeholders navigate through it.

One of the most fundamental rules for technical documentation and software architecture documentation in particular is, to write from the point of view of the reader. Documentation that was easy to write but is not easy to read will not be used, and

"easy to read" is in the eye of the beholder—or in this case, the stakeholder.

Understanding who the stakeholders are and how they will want to use the documentation will help us organize it and make it accessible to and usable for them.

Each stakeholder come in two varieties: seasoned and new.

Perhaps one of the most avid consumers of architectural documentation is none other than the architect at some time in the project's future.

Views:

Perhaps the most important concept associated with software architecture documentation is the **view**. Software architecture for a system is "the structure or structures of the system, which comprise elements, the externally visible properties of those elements, and the relationships among them." And we said that a view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. A structure is the set of elements itself, as they exist in software or hardware.

A view simply represents a set of system elements and relationships among them, so whatever elements and relationships you deem useful to a segment of the stakeholder community constitute a valid view.

Software architecture is a complex entity that cannot be described in a simple one-dimensional fashion.

Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view.

This principle is useful because it breaks the problem of architecture documentation into more tractable parts, which provide the structure for the remainder of this chapter:

- Choosing the relevant views
- Documenting a view
- Documenting information that applies to more than one view

1] Choosing the Relevant Views:

The many purposes that architecture can serve—as a mission statement for implementors, as the starting point for system understanding and asset recovery, as the blueprint for project planning, and so forth—are each represented by a stakeholder wanting and expecting to use the documentation to serve that purpose.

Similarly, the quality attributes of most concern to you and the other stakeholders in the system's development will affect the choice of what views to document. For instance, a layered view will tell you about your system's portability. A deployment view will let you reason about your system's performance and reliability. And so on. These quality attributes are "spoken for" in the documentation by analysts (perhaps even the architect) who need to examine the architecture to make sure the quality attributes are provided.

In short, different views support different goals and uses. This is fundamentally why we

do not advocate a particular view or a collection of views. The views you should document depend on the uses you expect to make of the documentation. Different views will highlight different system elements and/or relationships.

Table 9.2 shows a representative population of stakeholders and the kind of views they tend to find useful. You should use it to help you think about who your stakeholders are and what views might serve them well. Which views are available from which to choose? **Views are divided into three groups: module, component-and-connector (C&C), and allocation.** This three-way categorization reflects the fact that architects need to think about their software in at least three ways at once:

1. How it is structured as a set of implementation units
2. How it is structured as a set of elements that have runtime behavior and interactions
3. How it relates to non-software structures in its environment

Table 9.2. Stakeholders and the Architecture Documentation They Might Find Most Useful

	Module Views				**C&C Views**	**Allocation Views**	
Stakeholder	**Decomposition**	**Uses**	**Class**	**Layer**	**Various**	**Deployment**	**Implementation**
Project Manager	s	s		s		d	
Member of Development Team	d	d	d	d	d	s	s
Testers and Integrators		d	d		s	s	s
Maintainers	d	d	d	d	d	s	s
Product Line Application Builder		d	s	o	s	s	s
Customer					s	o	

End User					s	s	
Analyst	d	d	s	d	s	d	
Infrastructure Support	s	s		s		s	d
New Stakeholder	x	x	x	x	x	x	x
Current and Future Architect	d	d	d	d	d	d	s
Key: d = detailed information, s = some details, o = overview information, x = anything							

This is a three-step procedure for choosing the views for your project:

1. **Produce a candidate view list:**

   Begin by building a stakeholder/view table, like Table 9.2, for your project. Your stakeholder list is likely to be different from the one in the table, but be as comprehensive as you can. For the columns, enumerate the views that apply to your system. Some views (such as decomposition or uses) apply to every system, while others (the layered view, most component-and-connector views such as client-server or shared data) only apply to systems designed that way. Once you have the rows and columns defined, fill in each cell to describe how much information the stakeholder requires from the view: none, overview only, moderate detail, or high detail.

2. **Combine views:**

   The candidate view list from step 1 is likely to yield an impractically large number of views. To reduce the list to a manageable size, first look for views in the table that require only overview depth or that serve very few stakeholders. See if the stakeholders could be equally well served by another view having a stronger constituency. Next, look for views that are good candidates to be combined—that is, a view that gives information from two or more views at once. For small and medium projects, the implementation view is often easily overlaid with the module decomposition view. The module decomposition view also pairs well with uses or layered views. Finally, the deployment view usually combines well with whatever component-and-connector view shows the components that are allocated to hardware

elements—the process view, for example.

3. **Prioritize:**

After step 2 you should have an appropriate set of views to serve your stakeholder community. At this point you need to decide what to do first. How you decide depends on the details specific to your project, but remember that you don't have to complete one view before starting another. People can make progress with overview-level information, so a breadth-first approach is often the best. Also, some stakeholders' interests supersede others. A project manager or the management of a company with which yours is partnering demands attention and information early and often.

2] Documenting a View:

There is no industry-standard template for documenting a view, but the seven-part standard organization that we suggest in this section has worked well in practice. First of all, whatever sections you choose to include, make sure to have a standard organization. Allocating specific information to specific sections will help the documentation writer attack the task and recognize completion, and it will help the documentation reader quickly find information of interest at the moment and skip everything else.

The seven-part standard organization for documenting view is as follows:

1. ***Primary presentation*** shows the elements and the relationships among them that populate the view. The primary presentation should contain the information you wish to convey about the system (in the vocabulary of that view) first. It should certainly include the primary elements and relations of the view, but under some circumstances it might not include all of them. For example, you may wish to show the elements and relations that come into play during normal operation, but relegate error handling or exceptional processing to the supporting documentation.

   The primary presentation is usually graphical. In fact, most graphical notations make their contributions in the form of the primary presentation. If the primary presentation is graphical, it must be accompanied by a key that explains, or that points to an explanation of, the notation or symbology used.

   Sometimes the primary presentation can be tabular; tables are often a superb way to convey a large amount of information compactly.

2. ***Element catalog*** details at least those elements and relations depicted in the primary presentation. Producing the primary presentation is often what architects concentrate on, but without backup information that explains the picture, it is of little value. For instance, if a diagram shows elements A, B, and C, there had better be documentation that explains in sufficient detail what A, B, and C are, and their purposes or the roles they play, rendered in the vocabulary of the view. For example, a module decomposition view has elements that are modules, relations that are a form of "is part of," and properties that define the

responsibilities of each module. A process view has elements that are processes, relations that define synchronization or other process-related interaction, and properties that include timing parameters.

In addition, if there are elements or relations relevant to the view that were omitted from the primary presentation, the catalog is where those are introduced and explained.

The behavior and interfaces of elements are two other aspects of an element catalog; these will be discussed shortly.

3. ***Context diagram*** shows how the system depicted in the view relates to its environment in the vocabulary of the view. For example, in a component-and-connector view you show which component and connectors interact with external components and connectors, via which interfaces and protocols.
4. ***Variability guide*** shows how to exercise any variation points that are a part of the architecture shown in this view. In some architectures, decisions are left unbound until a later stage of the development process, and yet the architecture must still be documented. A variability guide should include documentation about each point of variation in the architecture, including:

- The options among which a choice is to be made. In a module view, the options are the various versions or parameterizations of modules. In a component-and-connector view, they might include constraints on replication, scheduling, or choice of protocol. In an allocation view, they might include the conditions under which a software element would be allocated to a particular processor.
- The binding time of the option. Some choices are made at design time, some at build time, and others at runtime.

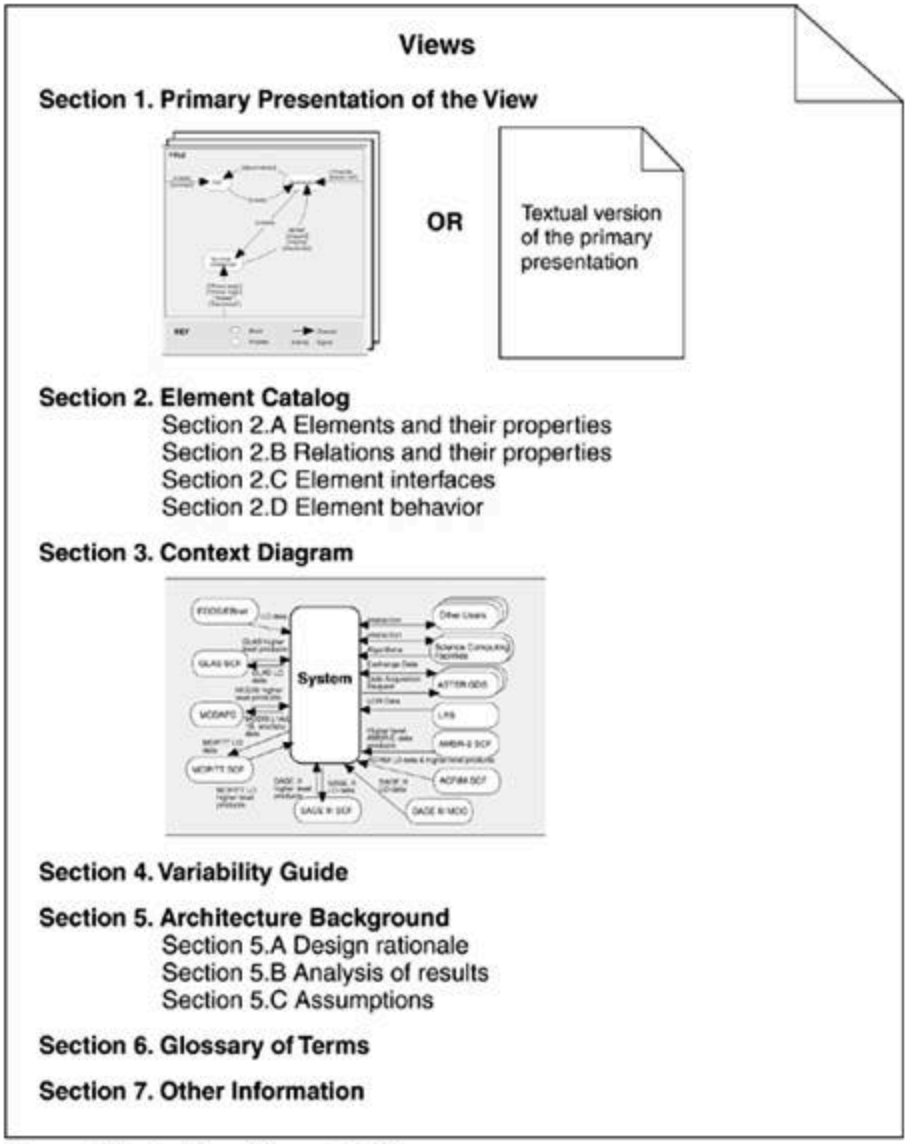
5. ***Architecture background*** explains why the design reflected in the view came to be. The goal of this section is to explain to someone why the design is as it is and to provide a convincing argument that it is sound. An architecture background includes:

- Rationale, explaining why the decisions reflected in the view were made and why alternatives were rejected.
- Analysis results, which justify the design or explain what would have to change in the face of a modification.
- Assumptions reflected in the design.

6. ***Glossary of terms*** used in the views, with a brief description of each.
7. ***Other information***. The precise contents of this section will vary according to the standard practices of your organization. They might include management information such as authorship, configuration control data, and change histories. Or the architect might record references to specific sections of a requirements document to establish traceability. Strictly speaking, information such as this is not architectural. Nevertheless, it is convenient to record it alongside the architecture, and this section is provided for that purpose. In any case, the first part of this section must detail its specific contents.

Figure 9.1 summarizes the parts of the documentation just described.

Figure 9.1. The seven parts of a documented view



Source: Adapted from [Clements 03].

- **DOCUMENTING BEHAVIOR:**

Views present structural information about the system. However, structural information is not sufficient to allow reasoning about some system properties. Reasoning about deadlock, for example, depends on understanding the sequence of interactions among the elements, and structural information alone does not present this sequencing information. Behavior descriptions add information that reveals the ordering of interactions among the elements, opportunities for concurrency, and time dependencies of interactions (at a specific time or after a period of time).

Behavior can be documented either about an element or about an ensemble of elements working in concert. Exactly what to model will depend on the type of system being

designed. For example, if it is a real-time embedded system, you will need to say a lot about timing properties and the time of events. In a banking system, the sequence of events (e.g., atomic transactions and rollback procedures) is more important than the actual time of events being considered. Different modeling techniques and notations are used depending on the type of analysis to be performed. In UML, sequence diagrams and statecharts are examples of behavioral descriptions. These notations are widely used.

Statecharts are a formalism developed in the 1980s for describing reactive systems. They add a number of useful extensions to traditional state diagrams such as nesting of state and "and" states, which provide the expressive power to model abstraction and concurrency. Statecharts allow reasoning about the totality of the system. All of the states are assumed to be represented and the analysis techniques are general with respect to the system. That is, it is possible to answer a question such as Will the response time to this stimulus always be less than 0.5 seconds?

A sequence diagram documents a sequence of stimuli exchanges. It presents a collaboration in terms of component instances and their interactions and shows the interaction arranged in time sequence. The vertical dimension represents time and the horizontal dimension represents different components. Sequence diagrams allow reasoning based on a particular usage scenario. They show how the system reacts to a particular stimulus and represent a choice of paths through the system. They make it possible to answer a question such as What parallel activities occur when the system is responding to these specific stimuli under these specific conditions?

- **DOCUMENTING INTERFACES:**

An interface is a boundary across which two independent entities meet and interact or communicate with each other. Elements' interfaces—carriers of the properties externally visible to other elements—are architectural. Since you cannot perform analyses or system building without them, documenting interfaces is an important part of documenting architecture.

Documenting an interface consists of naming and identifying it and documenting its syntactic and semantic information. The first two parts constitute an interface's "signature." When an interface's resources are invokable programs, the signature names the programs and defines their parameters. Parameters are defined by their order, data type, and (sometimes) whether or not their value is changed by the program. A signature is the information that you would find about the program, for instance, in an element's C or C++ header file or in a Java interface.

Signatures are useful (for example, they can enable automatic build checking), but are only part of the story. Signature matching will guarantee that a system will compile and/or link successfully. However, it guarantees nothing about whether the system will operate successfully, which is after all the ultimate goal. That information is bound up in the semantics to the interface, or what happens when resources are brought into play.

An interface is documented with an interface specification, which is a statement of element properties the architect chooses to make known. The architect should expose only what is needed to interact with the interface. Put another way, the architect chooses

what information is permissible and appropriate for people to assume about the element, and what is unlikely to change. Documenting an interface is a matter of striking a balance between disclosing too little information and disclosing too much. Too little information will prevent developers from successfully interacting with the element. Too much will make future changes to the system more difficult and widespread and make the interface too complicated for people to understand. A rule of thumb is to focus on how elements interact with their operational environments, not on how they are implemented. Restrict the documentation to phenomena that are externally visible.

Elements that occur as modules often correspond directly to one or more elements in a component-and-connector view. The module and component-and-connector elements are likely to have similar, if not identical, interfaces and documenting them in both places would produce needless duplication. To avoid that, the interface specification in the component-and-connector view can point to the interface specification in the module view, and only contain the information specific to its view. Similarly, a module may appear in more than one module view—such as the module decomposition or uses view. Again, choose one view to hold the interface specification and refer to it in the others.

A Template for Documenting Interfaces

Here is a suggested standard organization for interface documentation. You may wish to modify it to remove items not relevant to your situation, or add items unique to it. More important than which standard organization you use is the practice of using one. Use what you need to present an accurate picture of the element's externally visible interactions for the interfaces in your project.

1. ***Interface identity***. When an element has multiple interfaces, identify the individual interfaces to distinguish them. This usually means naming them. You may also need to provide a version number.
2. Resources ***provided***. The heart of an interface document is the resources that the element provides. Define them by giving their syntax, their semantics (what happens when they are used), and any restrictions on their usage. Several notations exist for documenting an interface's syntax. One is the OMG's Interface Definition Language (IDL), used in the CORBA community. It provides language constructs to describe data types, operations, attributes, and exceptions. The only language support for semantic information is a comment mechanism. Most programming languages have built-in ways to specify the signature of an element. C header (`.h`) files and Ada package specifications are two examples. Finally, using the `<<interface>>` stereotype in UML (as shown in Figure 9.4) provides the means for conveying syntactic information about an interface. At a minimum, the interface is named; the architect can also specify signature information.
    - Resource syntax. This is the resource's signature. The signature includes any information another program will need to write a syntactically correct program that uses the resource. The signature includes the resource name, names and logical data types of arguments (if any), and so forth.
    - Resource semantics. This describes the result of invoking the resource. It might include

- assignment of values to data that the actor invoking the resource can access. It might be as simple as setting the value of a return argument or as far-reaching as updating a central database.

- events that will be signaled or messages that will be sent as a result of using the resource.

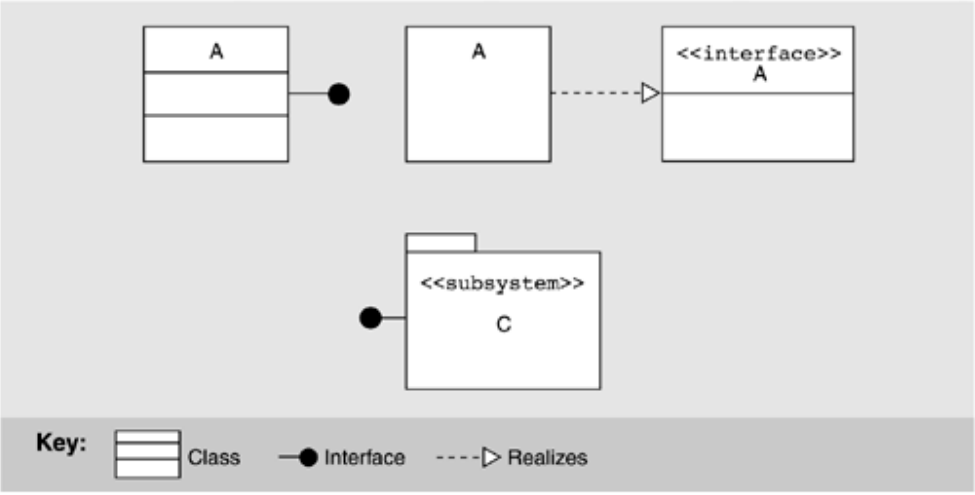
- how other resources will behave in the future as the result of using this resource. For example, if you ask a resource to destroy an object, trying to access that object in the future through other resources will produce quite a different outcome (an error).

- humanly observable results. These are prevalent in embedded systems; for example, calling a program that turns on a display in a cockpit has a very observable effect: The display comes on.

In addition, the statement of semantics should make it clear whether the resource execution will be atomic or may be suspended or interrupted. The most widespread notation for conveying semantic information is natural language. Boolean algebra is often used to write down preconditions and postconditions, which provide a relatively simple and effective method for expressing semantics. Traces are also used to convey semantic information by writing down sequences of activities or interactions that describe the element's response to a specific use.

- Resource usage restrictions. Under what circumstances may this resource be used? Perhaps data must be initialized before it can be read, or a particular method cannot be invoked unless another is invoked first. Perhaps there is a limit on the number of actors that can interact via this resource at any instant. Perhaps only one actor can have ownership and be able to modify the element whereas others have only read access. Perhaps only certain resources or interfaces are accessible to certain actors to support a multi-level security scheme. If the resource requires that other resources be present, or makes other assumptions about its environment, these should be documented.

Figure 9.4. Interfaces in UML



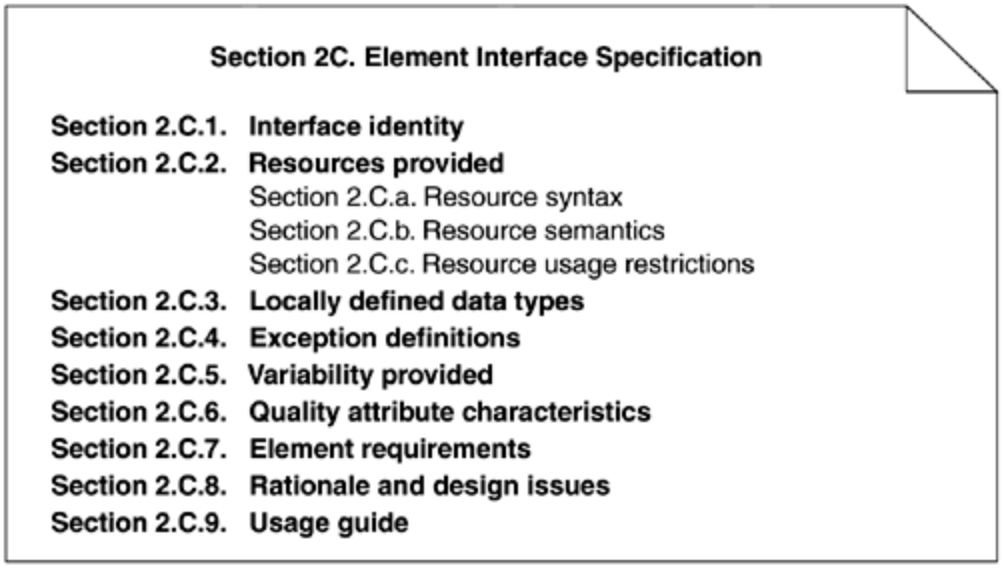
3. ***Data type definitions***. If any interface resources employ a data type other than one provided by the underlying programming language, the architect needs to communicate the definition of that data type. If it is defined by another element, then a reference to the definition in that element's documentation is sufficient. In any case, programmers writing elements using such a resource need to know (a) how to declare variables and constants of the data type; (b) how to write literal values in the data type; (c) what operations and comparisons may be performed on members of the data type; and (d) how to convert values of the data type into other data types, where appropriate.
4. ***Exception definitions***. These describe exceptions that can be raised by the resources on the interface. Since the same exception might be raised by more than one resource, it is often convenient to simply list each resource's exceptions but define them in a dictionary collected separately. This section is that dictionary. Common exception-handling behavior can also be defined here.
5. ***Variability provided by the interface***. Does the interface allow the element to be configured in some way? These configuration parameters and how they affect the semantics of the interface must be documented. Examples of variability include the capacities of visible data structures and the performance characteristics of underlying algorithms. Name and provide a range of values for each configuration parameter and specify the time when its actual value is bound.
6. ***Quality attribute characteristics of the interface***. The architect needs to document what quality attribute characteristics (such as performance or reliability) the interface makes known to the element's users. This information may be in the form of constraints on implementations of elements that will realize the interface. Which qualities you choose to concentrate on and make promises about will depend on context.
7. ***Element requirements***. What the element requires may be specific, named resources provided by other elements. The documentation obligation is the same

as for resources provided: syntax, semantics, and any usage restrictions. Often it is convenient to document information like this as a set of assumptions that the element's designer has made about the system. In this form, they can be reviewed by experts who can confirm or repudiate the assumptions before design has progressed too far.

8. ***Rationale and design issues.*** As with rationale for the architecture (or architectural views) at large, the architect should record the reasons for an element's interface design. The rationale should explain the motivation behind the design, constraints and compromises, what alternative designs were considered and rejected (and why), and any insight the architect has about how to change the interface in the future.
9. ***Usage guide***. Item 2 and item 7 document an element's semantic information on a per resource basis. This sometimes falls short of what is needed. In some cases semantics need to be reasoned about in terms of how a broad number of individual interactions interrelate. Essentially, a protocol is involved that is documented by considering a sequence of interactions. Protocols can represent the complete behavior of the interaction or patterns of usage that the element designer expects to come up repeatedly. If interacting with the element via its interface is complex, the interface documentation should include a static behavioral model such as a statechart, or examples of carrying out specific interactions in the form of sequence diagrams. This is similar to the view-level behaviors presented in the previous section, but focused on a single element.

Figure 9.2 summarizes this template which is an expansion of section 2.C from Figure 9.1.

Figure 9.2. The nine parts of interface documentation



Source: Adapted from [Clements 03].

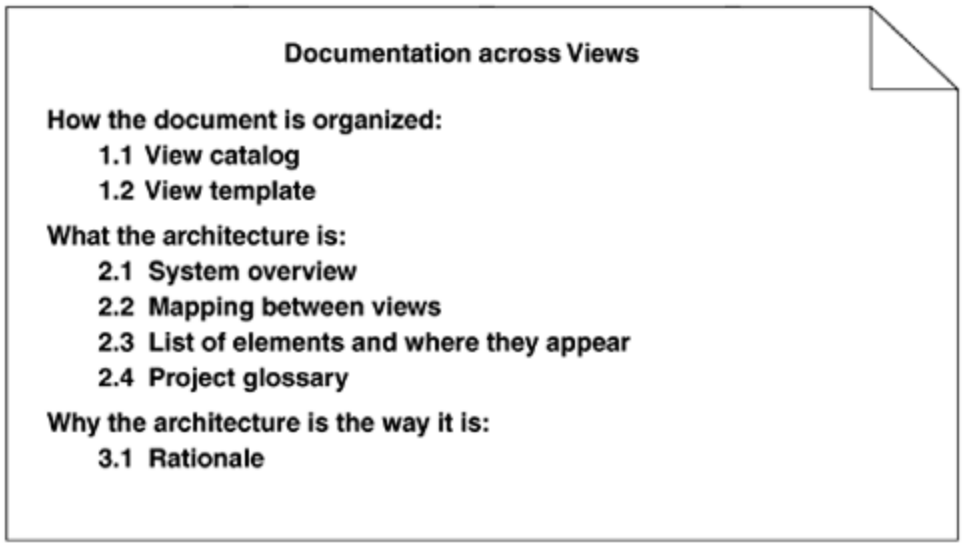
3] Documenting information that applies to more than one view (Documentation across Views – Cross-View documentation):

We now turn to the complement of view documentation, which is capturing the information that applies to more than one view or to the documentation package as a whole. Cross-view documentation consists of just three major aspects, which we can summarize as how-what-why:

1. ***How the documentation is laid out and organized*** so that a stakeholder of the architecture can find the information he or she needs efficiently and reliably. This part consists of a view catalog and a view template.
2. ***What the architecture is***. Here, the information that remains to be captured beyond the views themselves is a short system overview to ground any reader as to the purpose of the system; the way the views are related to each other; a list of elements and where they appear; and a glossary that applies to the entire architecture.
3. ***Why the architecture is the way it is***: the context for the system, external constraints that have been imposed to shape the architecture in certain ways, and the rationale for coarse-grained large-scale decisions.

Figure 9.3 summarizes these points.

Figure 9.3. Summary of cross-view documentation



Source: Adapted from [Clements 03].

1. HOW THE DOCUMENTATION IS ORGANIZED TO SERVE A STAKEHOLDER:

Every suite of architectural documentation needs an introductory piece to explain its

organization to a novice stakeholder and to help that stakeholder access the information he or she she is most interested in. There are two kinds of "how" information:

- A view catalog
- A view template

View Catalog:

A view catalog is the reader's introduction to the views that the architect has chosen to include in the suite of documentation.

When using the documentation suite as a basis for communication, it is necessary for a new reader to determine where particular information can be found. A catalog contains this information. When using the documentation suite as a basis for analysis, it is necessary to know which views contain the information necessary for a particular analysis. In a performance analysis, for example, resource consumption is an important piece of information, A catalog enables the analyst to determine which views contain properties relevant to resource consumption.

There is one entry in the view catalog for each view given in the documentation suite. Each entry should give the following:

1. The name of the view and what style it instantiates
2. A description of the view's element types, relation types, and properties
3. A description of what the view is for
4. Management information about the view document, such as the latest version, the location of the view document, and the owner of the view document

The view catalog is intended to describe the documentation suite, not the system being documented. Specifics of the system belong in the individual views, not in the view catalog. For instance, the actual elements contained in a view are listed in the view's element catalog.

View Template:

A view template is the standard organization for a view. Figure 9.1 and the material surrounding it provide a basis for a view template by defining the standard parts of a view document and the contents and rules for each. The purpose of a view template is that of any standard organization: It helps a reader navigate quickly to a section of interest, and it helps a writer organize the information and establish criteria for knowing how much work is left to do.

2. WHAT THE ARCHITECTURE IS:

This section provides information about the system whose architecure is being documented, the relation of the views to each other, and an index of architectural elements.

System Overview:

This is a short prose description of what the system's function is, who its users are, and any important background or constraints. The intent is to provide readers with a

consistent mental model of the system and its purpose. Sometimes the project at large will have a system overview, in which case this section of the architectural documentation simply points to that.

Mapping between Views:

Since all of the views of an architecture describe the same system, it stands to reason that any two views will have much in common. Helping a reader of the documentation understand the relationships among views will give him a powerful insight into how the architecture works as a unified conceptual whole. Being clear about the relationship by providing mappings between views is the key to increased understanding and decreased confusion.

For instance, each module may map to multiple runtime elements, such as when classes map to objects. Complications arise when the mappings are not one to one, or when runtime elements of the system do not exist as code elements at all, such as when they are imported at runtime or incorporated at build or load time. These are relatively simple one- (or none-) to-many mappings. In general, though, parts of elements in one view can map to parts of elements in another view.

It is not necessary to provide mappings between every pair of views. Choose the ones that provide the most insight.

Element List:

The element list is simply an index of all of the elements that appear in any of the views, along with a pointer to where each one is defined. This will help stakeholders look up items of interest quickly.

Project Glossary:

The glossary lists and defines terms unique to the system that have special meaning. A list of acronyms, and the meaning of each, will also be appreciated by stakeholders. If an appropriate glossary already exists, a pointer to it will suffice here.

3. WHY THE ARCHITECTURE IS THE WAY IT IS: RATIONALE:

Similar in purpose to the rationale for a view or the rationale for an interface design, cross-view rationale explains how the overall architecture is in fact a solution to its requirements. One might use the rationale to explain

- the implications of system-wide design choices on meeting the requirements or satisfying constraints.
- the effect on the architecture when adding a foreseen new requirement or changing an existing one.
- the constraints on the developer in implementing a solution.
- decision alternatives that were rejected.

In general, the rationale explains why a decision was made and what the implications are in changing it.