**Chapter 1 - What Is Software Architecture**

**1.1 What SA is and what it isn’t**

There are 3 categories of architectural structure:

1. **Modules**. Modules are assigned specific computational responsibilities and are the basis of work assignments for programming teams. Modules are assigned specific computational responsibilities, and are the basis of work assignments for programming teams (Team A works on the database, Team B works on the business rules, etc.). The structure that captures that decomposition is a kind of module structure, the module decomposition structure in fact. Another kind of module structure emerges as an output of object-oriented analysis and design class diagrams. Module structures are static structures since they focus on the way the system’s functionality is divided up.
2. Other structures are dynamic, meaning that they focus on the way the elements interact with each other at runtime to carry out the system's functions. In this book we will call runtime structures **component-and-connector** (C&C) **structures**.
3. A third kind of structure describes the mapping from software structures to the system's organizational, developmental, installation, and execution environments. For example, modules are assigned to teams to develop, and assigned to places in a file structure for implementation, integration, and testing. Components are deployed onto hardware in order to execute. These mappings are called **allocation** **structures**.

Not all structures are architectural. A structure is architectural if it supports reasoning about the system and the system's properties. The reasoning should be about an attribute of the system that is important to some stakeholder.

**Architecture is an abstraction**Because architecture consists of structures and structures consist of elements and relations, it follows that an architecture comprises software elements and how the elements relate to each other – omitting information that isn’t useful. Thus, an architecture is an abstraction of a system that selects certain details and suppresses others.   
This abstraction is essential to taming the complexity of a system.

**System Architecture:**A system architecture is concerned with a total system, including hardware, software, and humans. It’s a representation of a system including a mapping of: functionality => hardware & software components, software architecture => hardware architecture, and a concern of the human interaction with these components

**Enterprise Architecture:**   
Enterprise architecture is a description of the structure and behaviour of an organization's processes, information flow, personnel, and organizational subunits, aligned with the organization's core goals and strategic direction

**1.2. Architectural Structures and Views**

A **view** is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. It consists of a representation of a set of elements and the relations among them. => a view is a representation of a structure

A **structure** is the set of elements itself, as they exist in software or hardware.

Three kinds of structures => three kinds of decisions that architectural design involves:

1. *Module structures* embody decisions as to how the system is to be structured as a set of code or data units that have to be constructed or procured. Module structures allow us to answer questions like:

* What’s the primary functional responsibility assigned to each module?
* What other software elements is a module allowed to use?
* What other software does it actually use and depend on?

=> It’s an excellent way to reason about a system’s modifiability

1. *Component-and-connector structures* embody decisions as to how the system is to be structured as a set of elements that have runtime behaviour(components) and interactions(connections).  
   These structures help us answer questions like:

* What are the major executing components and how do they interact at runtime?
* What are the major shared data stores?
* How does data progress through the system?

1. *Allocation structures* embody decisions as to how the system will relate t nonsoftware structures in its environment. These structures show the relationship between the software elements and elements in external environments in which the software is created/executed and helps us answer questions like:

* What processor does each software element execute on?
* What is the assignment of each software element to development teams?

**Useful module structures:**

*Decomposition structure*  
The units are modules that are related to each other by the *is-a submodule-of* relation, showing how modules are decomposed into smaller modules recursively until the modules are small enough to be easily understood.

*User structure*Here the units are also modules, perhaps classes. The units are related by the *uses* relation, a specialized form of dependency. A unit of software uses another if the correctness of the first requires the presence of a correctly functioning version of the second. The uses structure is used to engineer systems that can be extended to add functionality.

*Layer Structure*The modules in this structure are called layers. A layer is an abstract "virtual machine" that provides a cohesive set of services through a managed interface. (when you use abstract interfaces)

*Class Structure*The module units in this structure are called classes. The relation is *inherits* *from* or *is an instance* of. This view supports reasoning about collections of similar behaviour or capability and parameterized differences.

*Data Model*  
The data model describes the static information structure in terms of data entities and their relationships.

**Useful C&C Structures**

*Service Structure*The units here are services that interoperate with each other by service coordination mechanisms such as SOAP.

*Concurrency Structure*This structure allows the architect to determine opportunities for parallelism and the locations where resource contention may occur. The units are components and the connectors are their communication mechanisms.

**Useful Allocation Structures**

*Deployment Structure*The deployment structure shows how software is assigned to hardware processing and communication elements. The elements are software elements, hardware entities, and communication pathways. Relations are *allocated-to*, showing on which physical units the software elements reside, and migrates-to if the allocation is dynamic.

*Implementation Structure*This structure shows how software elements (usually modules) are mapped to the file structure(s) in the system's development, integration, or configuration control environments.

*Work Assignment Structure*This structure assigns responsibility for implementing and integrating the 1nodules to the teams who will carry it out

**1.4. What makes a “good” architecture**

There are rules of thumb that should be followed when designing most architectures, these can be divided in two: process recommendations and product recommendations.

Process recommendations:

1. The architecture should be the product of a single architect or a small group of architects with an identified technical leader.
2. The architect (or architecture team) should, on an ongoing basis, base the architecture on a prioritized list of well-specified quality attribute requirements.
3. The architecture should be documented using views. The views should address the concerns of the most important stakeholders in support of the project timeline.
4. The architecture should be evaluated for its ability to deliver the system's important quality attributes
5. The architecture should lend itself to incremental implementation, to avoid having to integrate everything at once (which almost never works) as well as to discover problems early.

Product recommendations:

1. The architecture should feature well-deed modules whose functional responsibilities are assigned on the principles of information hiding and separation of concerns. The information-hiding modules should encapsulate things likely to change, thus insulating the software from the effects of those changes.
2. Unless your requirements are unprecedented (unlikely), your quality attributes should be achieved using well-known architectural patterns and tactics specific to each attribute.
3. The architecture should never depend on a particular version of a commercial product or tool.
4. Modules that produce data should be separate from modules that consume data.
5. Don't expect a one-to-one correspondence between modules and components.
6. Every process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.
7. The architecture should feature a small number of ways for components to interact.

**Chapter 2 - Why is SA important?**

13 reasons..

**2.1. Inhibiting or Enabling 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.

* E.g. if your system requires high performance, then you need to pay attention to managing the time-based behaviour of elements, their use of shared resources, e

But an architecture alone cannot guarantee the functionality or quality required of a system. Poor downstream design or implementation decisions can always undermine an adequate architectural design.

A good architecture is necessary, but not sufficient, to ensure quality. Achieving quality attributes must be considered throughout design, implementation, and deployment

**2.2. Reasoning About and Managing Change**

Modifiability the ease with which changes can be made to a system is a quality attribute.   
The software development community is coming to grips with the fact that roughly 80 percent of a typical software system's total cost occurs after initial deployment.

Every architecture partitions possible changes into three categories: local, nonlocal, and architectural.

* Local: A local change can be accomplished by modifying a single element. For example, adding a new business rule to a pricing logic module.
* Nonlocal: A nonlocal change requires multiple element modifications but leaves the underlying architecture approach intact.
* Architectural: An architectural change affects the fundamental ways in which the elements interact with each other and will probably require changes all over the system.

=> Local changes are the most desirable, so an effective architecture is one in which the most common changes are local.

Deciding when changes are essential, determining which change paths have the least risk and assessing the consequences of proposed changes all require broad insight into relationships, performance, and behaviours of system software elements. These activities are in the job description for an architect.

**2.3. Predicting System Qualities**

It’s possible to make quality predictions about a system based solely on an evaluation of its architecture.

If we know that certain kinds of architectural decisions lead to certain quality attributes in a system, then we can make those decisions and rightly expect to be rewarded with the associated quality attributes.

Even if you don't do the quantitative analytic modelling sometimes necessary to ensure that an architecture will deliver its prescribed benefits, this principle of evaluating decisions based on their quality attribute implications is invaluable for at least spotting potential trouble spots early.

**2.4. Enhancing 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 creating mutual understanding, negotiating, forming consensus, and communicating with each other. 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

**2.5. Carrying Early Design Decisions**

Software architecture is a manifestation of the earliest design decisions about a system, and these early bindings carry enormous weight with respect to the system's remaining development, its deployment, and its maintenance life.

An architecture design can also be viewed as a set of decisions. The early design decisions constrain the decisions that follow, and changing these decisions has enormous ramifications. Changing these early decisions will cause a ripple effect, in terms of the additional decisions that must now be changed. Yes, sometimes the architecture must be refactored or redesigned, but this is not a task we undertake lightly (because the "ripple" might turn into a tsunami).

What are these early design decisions embodied by software architecture? Consider:

* Will the system run on one processor or be distributed across multiple processors?
* Will components communicate synchronously or asynchronously?
* Will the system depend on specific features of the operating system or hardware?
* What operating system will we use?

**2.6. Defining Constraints on an Implementation**

An implementation exhibits an architecture if it conforms to the design decisions prescribed by the architecture. This means that the implementation must be implemented as the set of prescribed elements, these elements must interact with each other in the prescribed fashion, and each element must fulfil its responsibility to the other elements as dictated by the architecture. Each of these prescriptions is a constraint on the implementer

**2.7. Influencing the Organizational Structure**

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).

The normal method for dividing up the labour in a large project is to assign different groups different portions of the system to construct. This is called the work-breakdown structure of a system. Because the architecture includes the broadest decomposition of the system, it is typically used as the basis for the work-breakdown structure

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.

Thus, once the architecture has been agreed on, it becomes very costly for managerial and business reasons to significantly modify it.

**2.8. Enabling Evolutionary Prototyping**

Once an architecture has been defined, it can be analysed and prototyped as a skeletal system. A skeletal system is one in which at least some of the infrastructure how the elements initialize, communicate, share data, access resources, report errors, log activity, and so forth is built before much of the system's functionality has been created.

For example, systems built as plug-in architectures are skeletal systems: the plug-ins provide the actual functionality. This approach aids the development process because the system is executable early in the product's life cycle. Among other things, this approach allows potential performance problems to be identified early in the product's life cycle.

These benefits reduce the potential risk in the project. Furthermore, 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.

**2.9. Improving Cost and Schedule Estimates**

One of the duties of an architect is to help the project manager create cost and schedule estimates early in the project life cycle.

Although top-down estimates are useful for setting goals and apportioning budgets, cost estimations that are based on a bottom-up understanding of the system's pieces are typically more accurate than those that are based purely on top-down system knowledge.

The organizational and work-breakdown structure of a project is almost always based on its architecture. Each team or individual responsible for a work item will be able to make more-accurate estimates for their piece than a project manager and will feel more ownership in making the estimates come true.

**2.10. Supplying a Transferable, Reusable Model**

The earlier in the life cycle that reuse is applied, the greater the benefit that can be achieved. While code reuse provides a benefit, reuse of architectures provides tremendous leverage for systems with similar requirements.

A software product line or family is a set of software systems that are all built using the same set of reusable assets. Chief among these assets is the architecture that was designed to handle the needs of the entire family. Product-line architects choose an architecture (or a family of closely related architectures) that will serve all envisioned members of the product line.

**2.11. Allowing Incorporation of Independently Developed Components**

Whereas earlier software paradigms have 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. This composition is possible because the architecture defines the elements that can be incorporated into the system.

The payoff for this can be:

* Decreased time to market, it should be easier to use someone else’s ready solution than build your own.
* Increased reliability, widely used software should have its bugs ironed out already.
* Lower cost, the software supplier can amortize development cost across their customer base.
* Flexibility, if the component needed has a specialized purpose, it’s likely to be available from many sources.

**2.12. Restricting the Vocabulary of Design Alternatives**

As useful architectural patterns are collected, it becomes clear that although software elements 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 of elements and their interactions. By doing so we minimize the design complexity of the system we are building.

Architectural patterns guide the architect and focus the architect on the quality attributes of interest in large part by restricting the vocabulary of design alternatives to a relatively small number.

Properties of software design follow from the choice of an architectural pattern. Those 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, or by helping to surface inconsistencies in requirements

**2.13. Providing a Basis for Training**

The architecture, including a description of how the elements interact with each other to carry out the required behaviour, can serve as the first introduction to the system for new project members.

Module views are excellent for showing someone the structure of a project: Who does what, which teams are assigned to which parts of the system, and so forth. Component-and-connector views are excellent for explaining how the system is expected to work and accomplish its job.

**Chapter 4 – Understanding Quality Attributes**

A quality attribute (QA) is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders. You can think of a quality attribute as measuring the "goodness" of a product along some dimension of interest to a stakeholder.

**4.1 Architecture and Requirements**

Requirement categories:

1. **Functional requirement**. State what the system must do, and how it must behave or react to runtime stimuli
2. **Quality attribute requirement**. These requirements are qualifications of the functional requirements or of the overall product. A qualification of a functional requirement is e.g. how fast the function must be performed or how resilient it must be to erroneous input. A qualification of the overall product is e.g. the time to deploy the product or a limitation on operational cost.
3. **Constraints**. A constraint is a design decision that has already been made, e.g. to use a certain programming language.

The “response” of architecture to each of these types of requirements:

1. Functional requirements are satisfied by assigning an appropriate sequence of responsibilities throughout the design.
2. Quality attribute requirements are satisfied by the various structures designed into the architecture, and the behaviours and interactions of the elements that populate those structures.
3. Constraints are satisfied by accepting the design decisions and reconciling it with other affected design decisions.

**4.2 Functionality**

Functionality is the ability of the system to do the work for which it was intended.

Functionality has a very strange relationship with architecture. Functionality does not determine architecture, in fact if functionality was the only thing that mattered, you wouldn’t have to divide the system into pieces at all, just one big blob with no internal structure would do just fine.

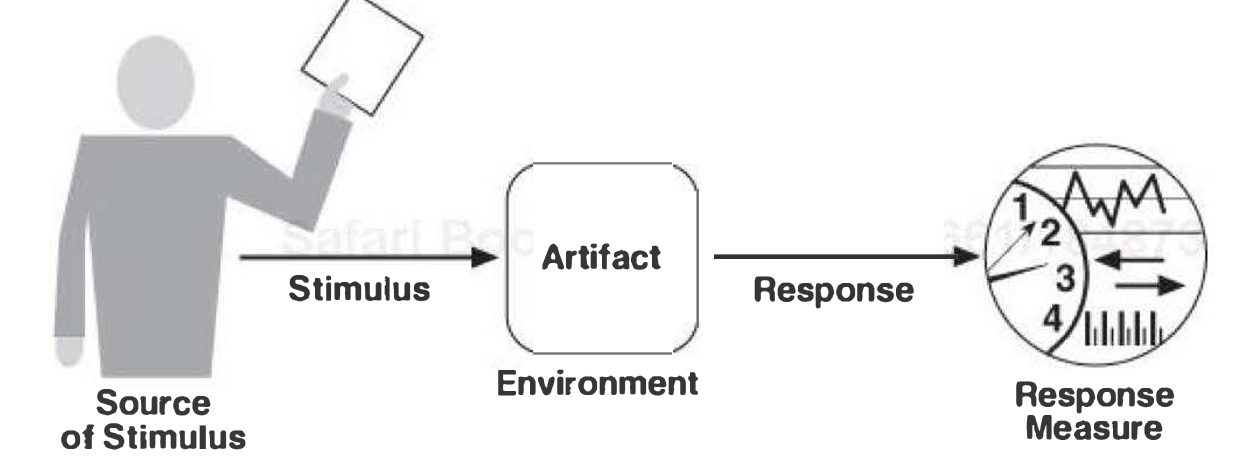
Instead, we design our systems as structured sets of cooperating architectural elements (modules, layers, classes, etc.) to make them understandable and to support a variety of other purposes.

**4.4. Specifying Quality Attribute Requirements**

A quality attribute requirement should be unambiguous and testable.

Quality attributes are commonly specified with a form that consists of 6 parts:

* **Source of Stimulus**. This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
* **Stimulus**. The stimulus is a condition that requires a response when it arrives at a system.
* **Environment**. The stimulus occurs under certain conditions. The system may be in an overload condition or in normal operation, or some other relevant state. For many systems, "normal" operation can refer to one of a number of modes. For these kinds of systems, the environment should specify in which mode the system is executing.
* **Artifact**. Some artifact is stimulated. This may be a collection of systems, the whole system, or some piece or pieces of it.
* **Response**. The response is the activity undertaken as the result of the arrival of the stimulus.
* **Response** **Measure**. When the response occurs, it should be measurable in some fashion so that the requirement can be tested.



General Quality Attribute Scenarios: those that are system independent and can potentially pertain to any system.

Concrete Quality Attribute Scenarios: those that are specific to the particular system under consideration.

**4.5. Achieving Quality Attributes through Tactics**

**Architectural Tactics** are techniques used to achieve the required quality attributes.  
A tactic is a design decision that influences the achievement of a quality attribute response tactics directly affect the system's response to some stimulus

The focus of a tactic is on a single quality attribute response. Within a tactic, there is no consideration of tradeoffs. Tradeoffs must be explicitly considered and controlled by the designer.

Why use tactics:

1. Design patterns are complex, they typically consist of a bundle of design decisions. But patters are often difficult to apply as is – architects need to modify and adapt them. By understanding the role of tactics, an architect can more easily assess the options for augmenting an existing pattern to achieve a quality attribute goal.
2. If no pattern exists to realize the architect's design goal, tactics allow the architect to construct a design fragment from "first principles.”

An example of a tactic:

*Schedule resources.* This is a common performance tactic but it obviously needs to be refined before applying, into a specific scheduling strategy, e.g. shortest-job-first.

**4.6 Guiding Quality Design Decisions**

Architecture is a result of applying a collection of design decisions.   
The following is a systematic categorization of these decisions so that an architect can focus attention on those design dimensions likely to be most troublesome:

1. Allocation of responsibilities
2. Coordination model
3. Data model
4. Management of resources
5. Mapping among architectural elements
6. Binding time decisions
7. Choice of technology

**Allocation of Responsibilities**

These decision include:

* Identifying the important responsibilities, including basic system functions, architectural infrastructure and satisfaction of quality attributes.
* Determining how these responsibilities are allocated to non-runtime and runtime elements (modules, components,..)

**Coordination Model**

Software works by having elements interact with each other through designed mechanisms. These mechanisms are collectively referred to as a coordination model.

These decisions include:

* Identifying the elements of the system that must coordinate or cannot coordinate.
* Determining the properties of the coordination, e.g. currency, completeness, correctness, consistency,..
* Choosing the communication mechanism that realize those properties

**Data Model**

These decisions include:

* Choosing the major data abstractions, their operations and their properties. This includes determining how the data items are created, initialized, accessed, persisted, manipulated, translated & destroyed.
* Compiling metadata needed for consisted interpretation of data.
* Organizing the data. E.g. determining whether the data is going to be kept in a relational database or a collection of objects or both.

**Management of Resources**

An architect may need to arbitrate the use of shared resources in the architecture.

These include hard resources like CPU, battery, memory, etc. and soft resources like system locks, software buffers, thread pools, etc.

These decisions include:

* Identifying the resources that must be managed and determining the limits for each.
* Determining which system elements manage each resource.
* Determining how resources are shared and the arbitration strategies employed when there is contention
* Determining the impact of saturation on different resources.

**Mapping among Architectural Elements**

An architecture must provide two types of mappings.

1. Mapping between elements in different types of architectural structures, e.g. mapping of modules to threads/processes.
2. Mapping between software elements and environment elements, e.g. from processes to the specific CPU where these processes will execute.

**Binding Time Decisions**

Binding time decisions introduce allowable ranges of variation. A binding time decision establishes the scope, the point in the life cycle, and the mechanism for achieving the variation.

When making binding time decisions, you should consider the costs to implement the decision and the costs to make a modification after you have implemented the decision.

For example, if you are considering changing platforms at some time after code time, you can insulate yourself from the effects caused by porting your system to another platform at some cost.

**Choice of Technology**

Every architecture decision must eventually be realized using a specific technology.

These decisions include:

* Deciding which technologies are available and realize the decisions made in the other categories.
* Determining whether the available tools to support this technology choice are adequate for development to proceed.
* Determining the extent of internal familiarity as well as the degree of external support available for the technology and deciding whether this is adequate to proceed.
* Determining the side effects of choosing a technology, such as a required coordination model or constrained resource management opportunities.
* Determining whether a new technology is compatible with the existing technology stack.

**Chapter 13 – Architectural Tactics and Patterns**

Architectural patterns and tactics are ways of capturing proven good design structures, so that they can be reused.

An architectural pattern:

* Is a package of design decisions that is found repeatedly in practice
* Has known properties that permit reuse
* Describe a class of architectures

Architectural tactics are simpler than architectural patterns. Tactics typically use just a single structure or computational mechanism, and they are meant to address a single architectural force.  
Tactics are the "building blocks" of design, from which architectural patterns are created.  
Most patterns consist of (are constructed from) several different tactics. For this reason we say that patterns package tactics.

**13.1 Architectural Patterns**

An architectural pattern establishes a relationship between:

* A *context.* A recurring, common situation in the world that gives rise to a problem
* A *problem.* The problem, appropriately generalized, that arises in the given context. The pattern description outlines the problem and its variants, and describes any complementary/opposing forces. It often includes quality attributes that must be met.
* A *solution*. A successful architectural resolution to the problem, appropriately abstracted. The solution describes the architectural structures that solve the problem, including how to balance the many forces at work. The solution for a pattern is determined and described by:
  + A set of element types (e.g. data repositories, processes and objects)
  + A set of interaction mechanisms (e.g. method calls, events or message bus)
  + A topological layout of the components
  + A set of semantic constraints covering topology, element behaviour and interaction mechanisms.

This *{context, problem, solution}* form constitutes a template for documenting a pattern.

Complex systems exhibit multiple patterns at once, which often have a layering internal structure.

**13.2 Overview of the Pattern Catalog**

Applying a pattern is not an all-or-nothing proposition. Pattern definitions given in catalogs are strict, but in practice architects may choose to violate them in small ways when there is a good design tradeoff to be had.

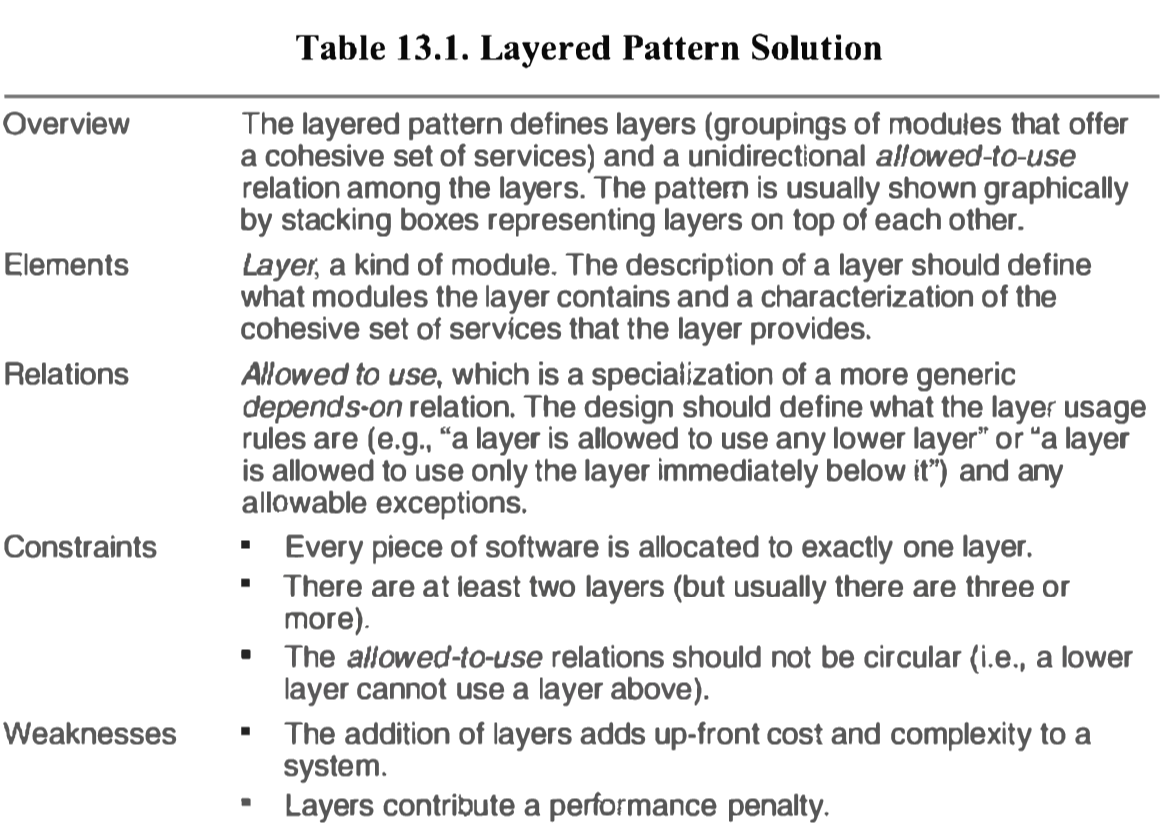
**Module Patterns**

**Layered Pattern**

*Context*: All complex systems experience the need to develop and evolve portions of the system independently. For this reason the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.

*Problem*: The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability and reuse.

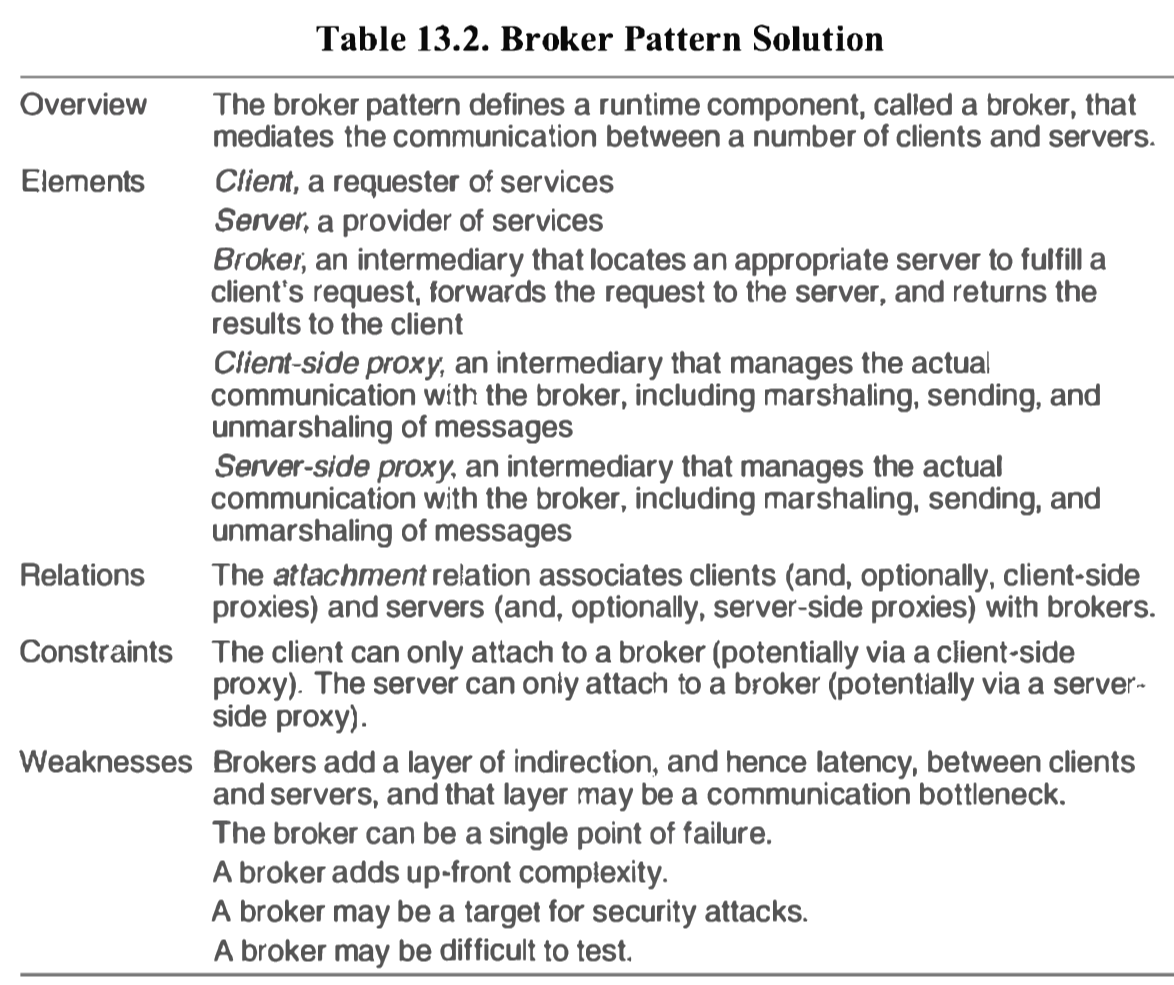
*Solution*:

 **Component-and-Connector Patterns**

**Broker Pattern**

*Context*: Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate how they will connect to each other and how they will exchange information.

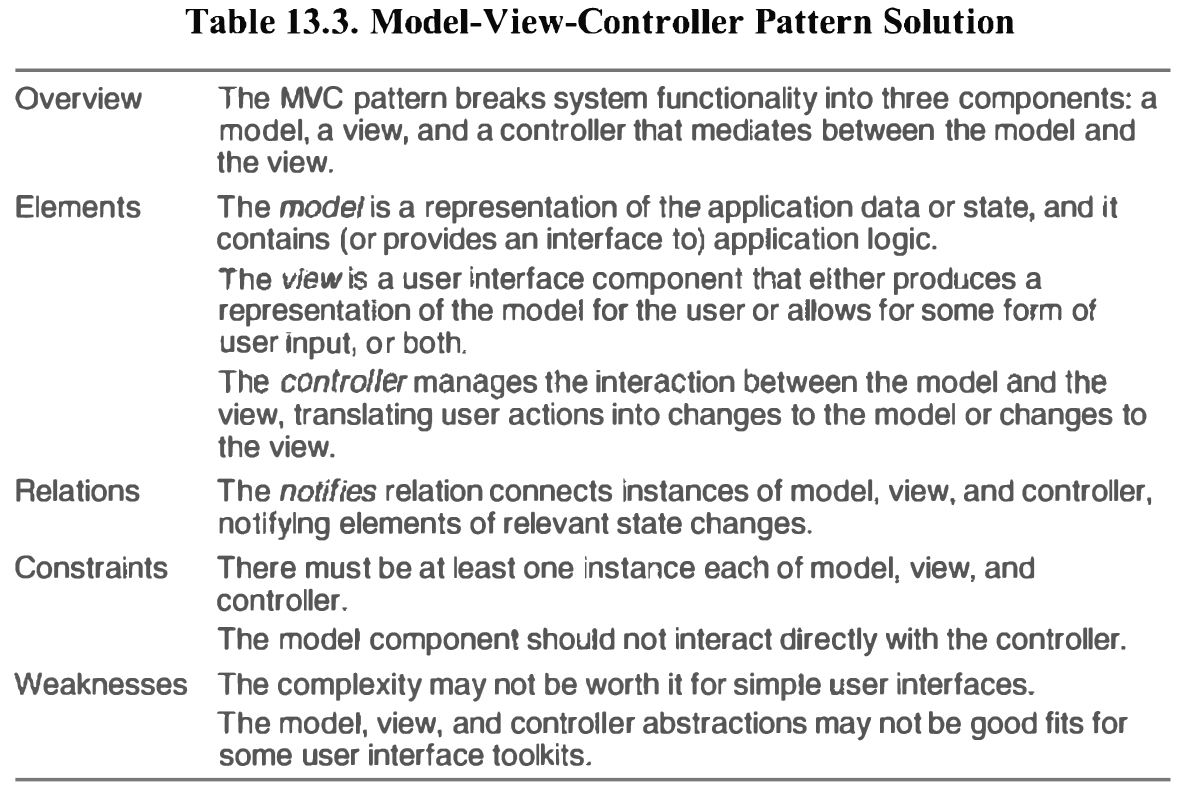
*Problem*: How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?

*Solution*:

**Model-View-Controller Pattern**

*Context*: User interface software is typically the most frequently modified portion of an interactive application. For this reason it is important to keep modifications to the user interface software separate from the rest of the system. Users often wish to look at data from different perspectives, such as a bar graph or a pie chart

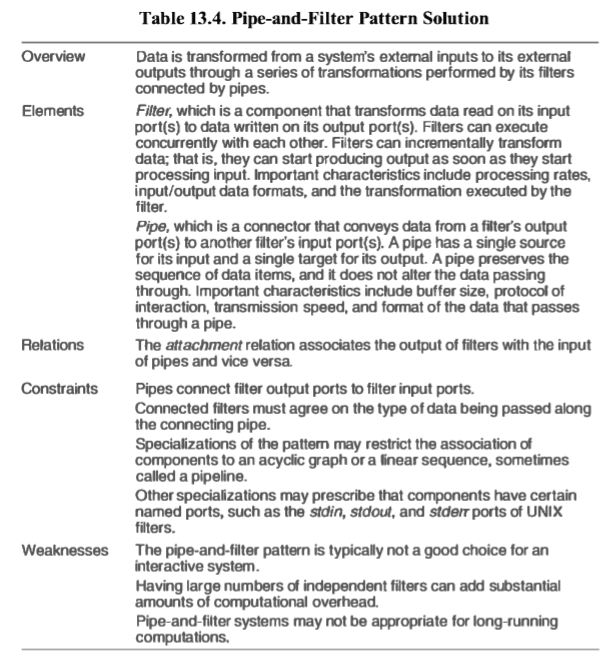
*Problem*: How can user interface functionality be kept separate from application functionality and yet still be responsive to user input, or to changes in the underlying application's data?

*Solution*:

**Pipe-and-Filter Pattern**

*Context*: Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

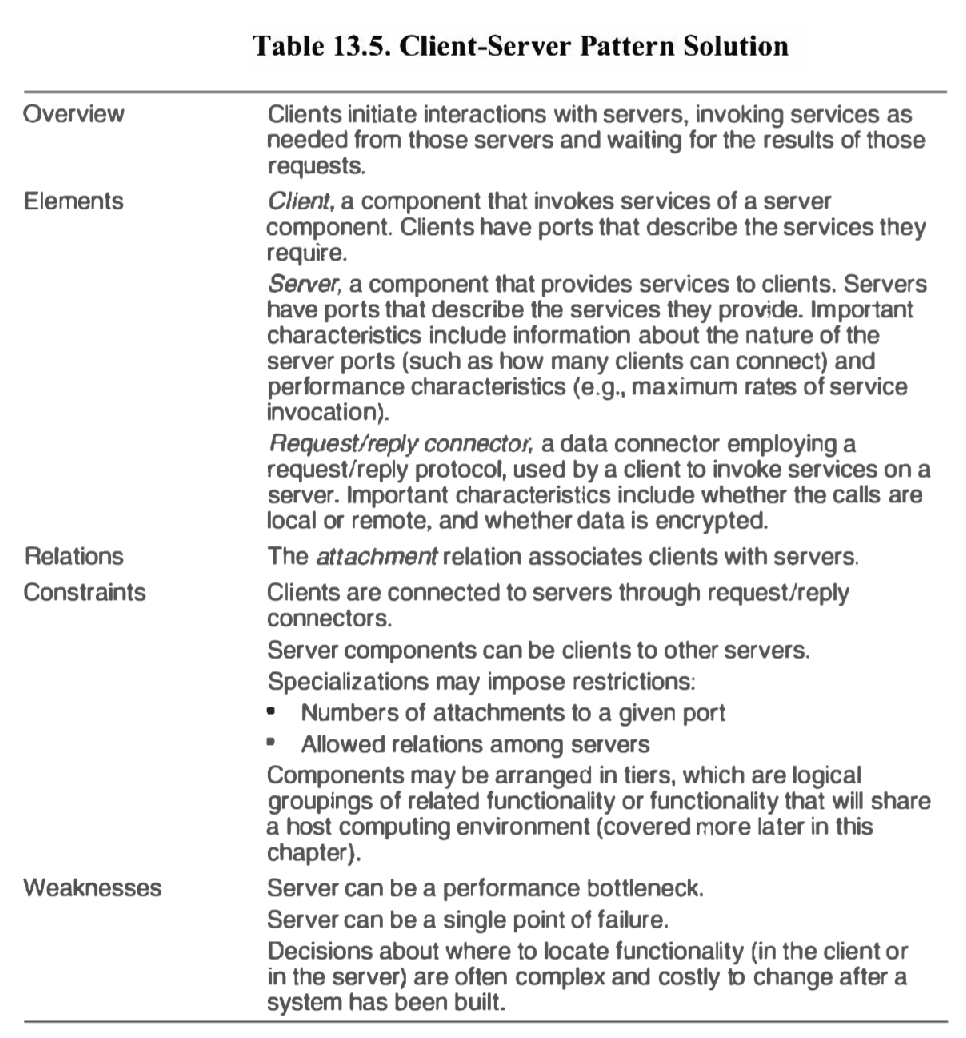
*Problem*: Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.

*Solution*:

**Client-Server Pattern**

*Context*: There are shared resources and services that large numbers of distributed clients wish to access, and for which we wish to control access or quality of service.

*Problem*: We want to improve scalability and availability by centralizing the control of these resources and services, while distributing the resources themselves across multiple physical servers.

*Solution*:

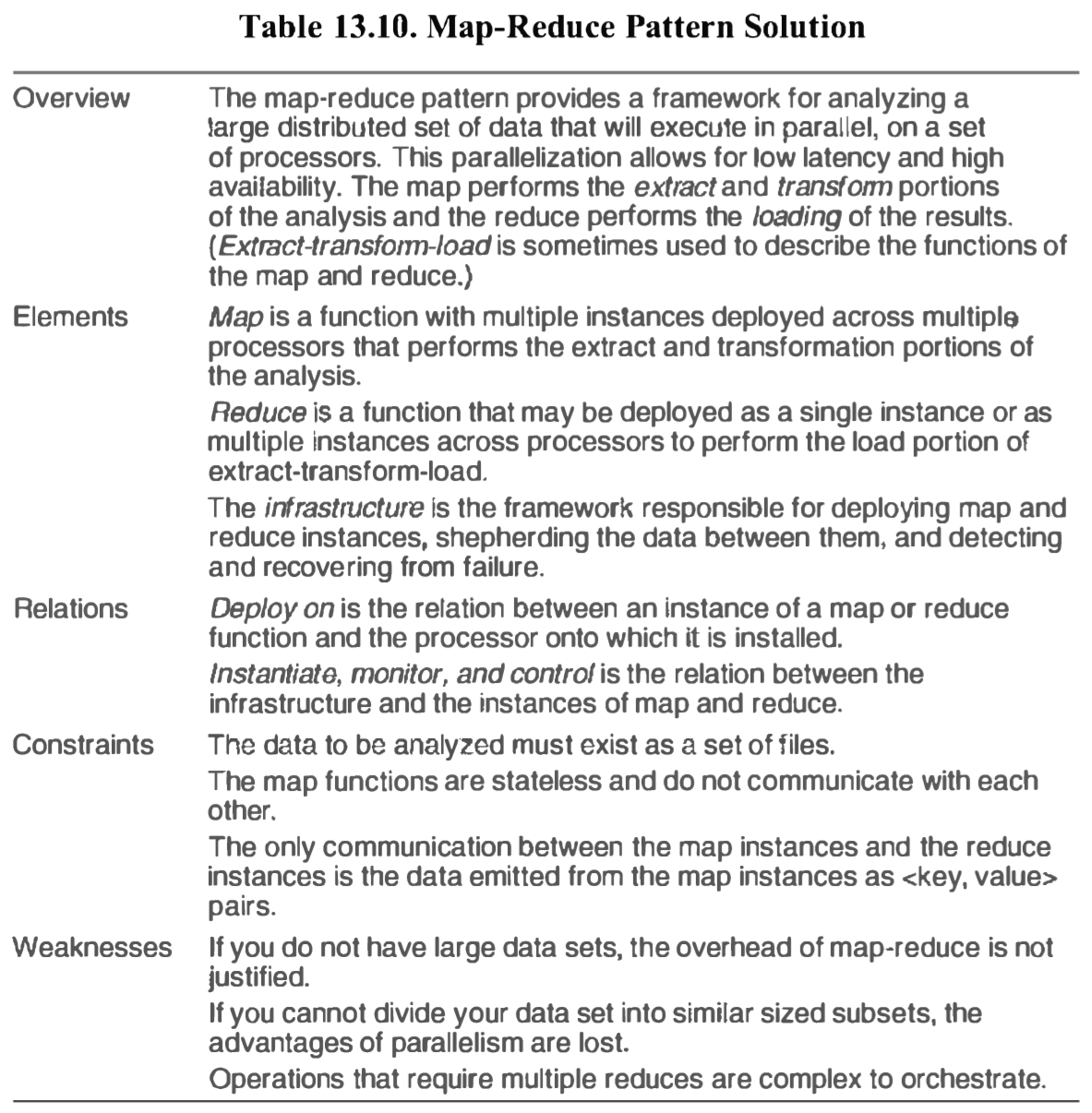
**Allocation Patterns**

**Map-Reduce Pattern**

*Context*: Businesses have a pressing need to quickly analyze enormous volumes of data they generate or access, at petabyte scale.

*Problem*: For many applications with ultra-large data sets, sorting the data and then analyzing the grouped data is sufficient. The problem the map-reduce pattern solves is to efficiently perform a distributed and parallel sort of a large data set and provide a simple means for the programmer to specify the analysis to be done.

*Solution*:



**13.3 Relationship between Tactics and Patterns**

**Patterns Comprise Tactics**Most patterns consist of (are constructed from) several different tactics, and although these tactics might all serve a common purpose such as promoting modifiability, for example they are often chosen to promote different quality attributes.  
Without any one of its tactics, the pattern might be ineffective. For example, if the *restrict dependencies* tactic is not employed, then any function in any layer can call any other function in any other layer, destroying the low coupling that makes the layering pattern effective.

**Using Tactics to Augment Patterns**

Using the broker pattern as an example, it’s main drawbacks were availability , performance, testability and security.

How can we use tactics to plug the gaps between the out-of-box broker pattern and a version of it that will let us meet the requirements of a demanding distributed system, here are some options:

* The *increase available resources* performance tactic would lead to multiple brokers, to help with performance and availability.
* The *maintain multiple copies* tactic would allow each of these brokers to share state, to ensure that they respond identically to client requests.
* *Load balancing* (an application of the scheduling resources tactic) would ensure that one broker is not overloaded while another one sits idle.
* Heartbeat, exception detection, or ping/echo would give the replicated brokers a way of notifying clients and notifying each other when one of them is out of service, as a means of detecting faults.

Of course, each of these tactics brings a tradeoff. Each complicates the design, which will now take longer to implement, be more costly to acquire, and be more costly to maintain.

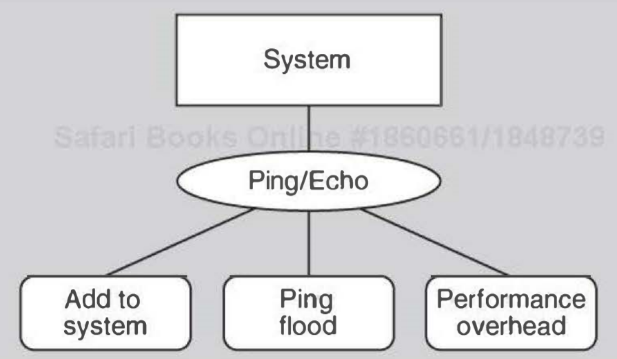
**13.4 Using Tactics Together**

Tactics are design primitives aimed at managing a single quality attribute response. Of course, this is almost never true in practice – every tactic has its main effect and its side effects – its tradeoffs.

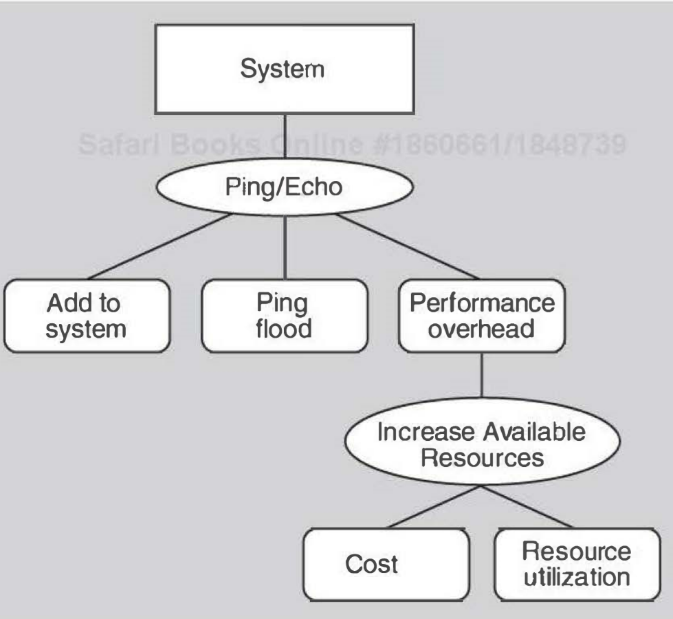
Whatever you do to improve one quality attribute endangers another. We are able to use tactics profitably because we can gauge the direct and side effects of a tactic, and when the tradeoff is acceptable, we employ the tactic.

Applying tactics to a pattern can produce negative effects in one area, but how adding other tactics can bring relief and put you back in an acceptable design space.

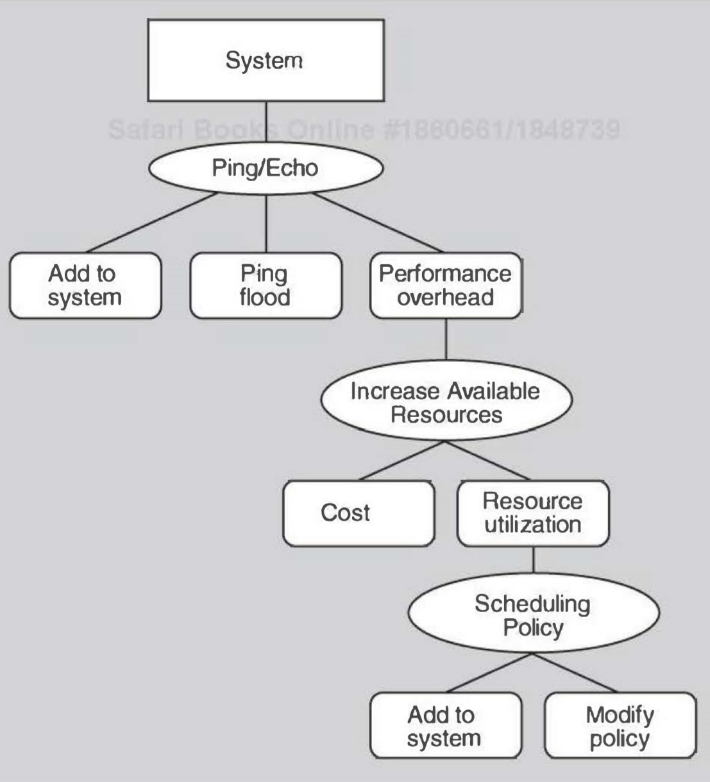
Consider a system that needs to detect faults in its components. A common tactic for detecting faults is *ping/echo*. Common considerations associated with ping/echo are these:

* *Security*. How to prevent a ping flood attach
* *Performance*. How to ensure that the performance overhead of ping(echo is small
* *Modifiability*. How to add ping/echo to the existing architecture.

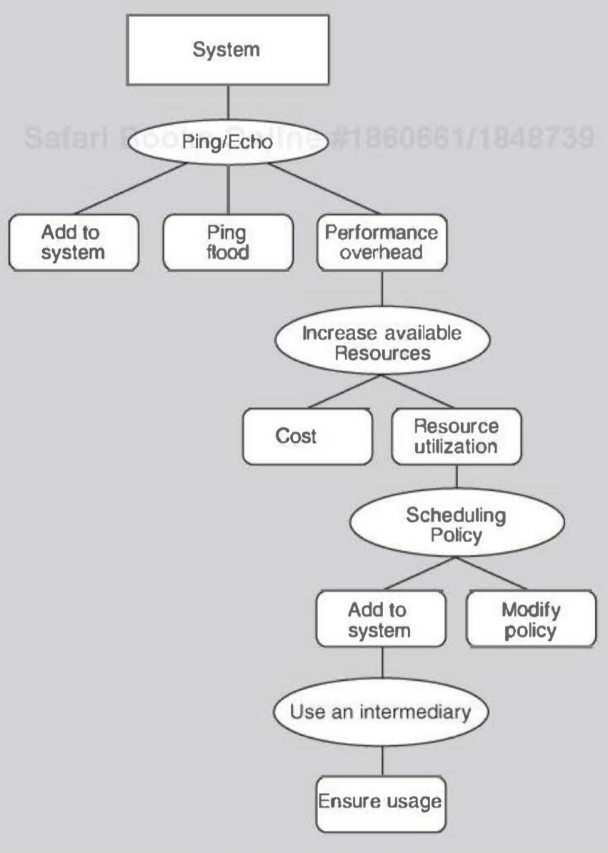
Suppose that the architect determines that the performance tradeoff is the most severe.  
A tactic to address the performance side effect is *increase available resources*. Considerations associated with that tactic are:

* *Cost*. Increased resources cost more.
* *Performance*. How to utilize the increased resources efficiently

Now the architect chooses to deal with the resource utilization consequence of employing increase available resources. These resources must be used efficiently or else they are simply adding cost and complexity to the system.  
A tactic that can address the efficient use of resources is the employment of *a scheduling policy*. Considerations associated with the scheduling policy tactic are these:

* *Modifiability*. How to add the scheduling policy to the existing architecture?
* *Modifiability*. How to change the scheduling policy in the future?

Next the architect chooses to deal with the modifiability consequence of employing a scheduling policy tactic.   
A tactic to address the addition of the scheduler to the system is to use an intermediary, which will insulate the choice of scheduling policy from the rest of the system. One consideration associated with use an intermediary is this:

* *Modifiability*. How to ensure that all communication passes through the intermediary?

A tactic to address the concern that all communication passes through the intermediary is restrict dependencies. One consideration associated with the restrict dependencies tactic is this:

* *Performance*. How to ensure that the performance overhead of the intermediary is not excessive?

This design problem has now become recursive!   
At this point (or in fact, at any point in the tree of design decisions that we have described) the architect might determine that the performance overhead of the intermediary is small enough that no further design decisions need to be made.

**Chapter 18 – Documenting Software Architectures**

Documentation speaks for the architect. It speaks for the architect today, when the architect should be doing other things besides answering a hundred questions about the architecture. And it speaks for the architect tomorrow, when he or she has left the project and now someone else is in charge of its evolution and maintenance.

**18.1. Uses and Audiences for Architecture Documentation**

Architecture documentation should:

* be sufficiently transparent and accessible to be quickly understood by new employees.
* be sufficiently concrete to serve as a blueprint for construction.
* have enough information to serve as a basis for analysis.

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

Fundamentally, architecture documentation has three uses:

1. *Architecture documentation serves as a means of education*, i.e. for introducing people to the system.
2. *Architecture documentation serves as a primary vehicle for communication among stakeholders.*
3. *Architecture documentation serves as the basis for system analysis and construction.* I.e. the architecture tells implementers what to implement, and determines with what other teams the development team for the module must communicate.

The document can also be used to

* Register and communicate issues that come up and must be resolved during development
* Hold information necessary to evaluate the variety of quality attributes.
* Provide guidance to those who wish to understand the behaviour of certain modules.

**18.2. Notation for Architecture Documentation**

Roughly, there are three main categories of notation for documenting views:

* *Informal Notation*. Views are depicted (sýna) using general-purpose diagramming and editing tools and visual conventions chosen for the system at hand. The most common tool for informal notations is PowerPoint.
* *Semiformal Notations*. Views are expressed in a standardized notation that prescribes graphical elements and rules of construction, but it doesn’t provide a semantic treatment of the meaning of those elements. UML is a semiformal notation.
* *Formal Notations*. Views are described in a notation that has a precise semantics. Formal analysis of both syntax and semantics is possible. Formal notations are often called architecture description languages (ADLs), they typically provide both graphical vocabulary and an underlying semantics for architecture representation.

Typically, more formal notations take more time and effort to create and understand, but they repay this effort in reduced ambiguity and more opportunities for analysis. Conversely, more informal notations are easier to create, but they provide fewer guarantees.

**18.3 Views**

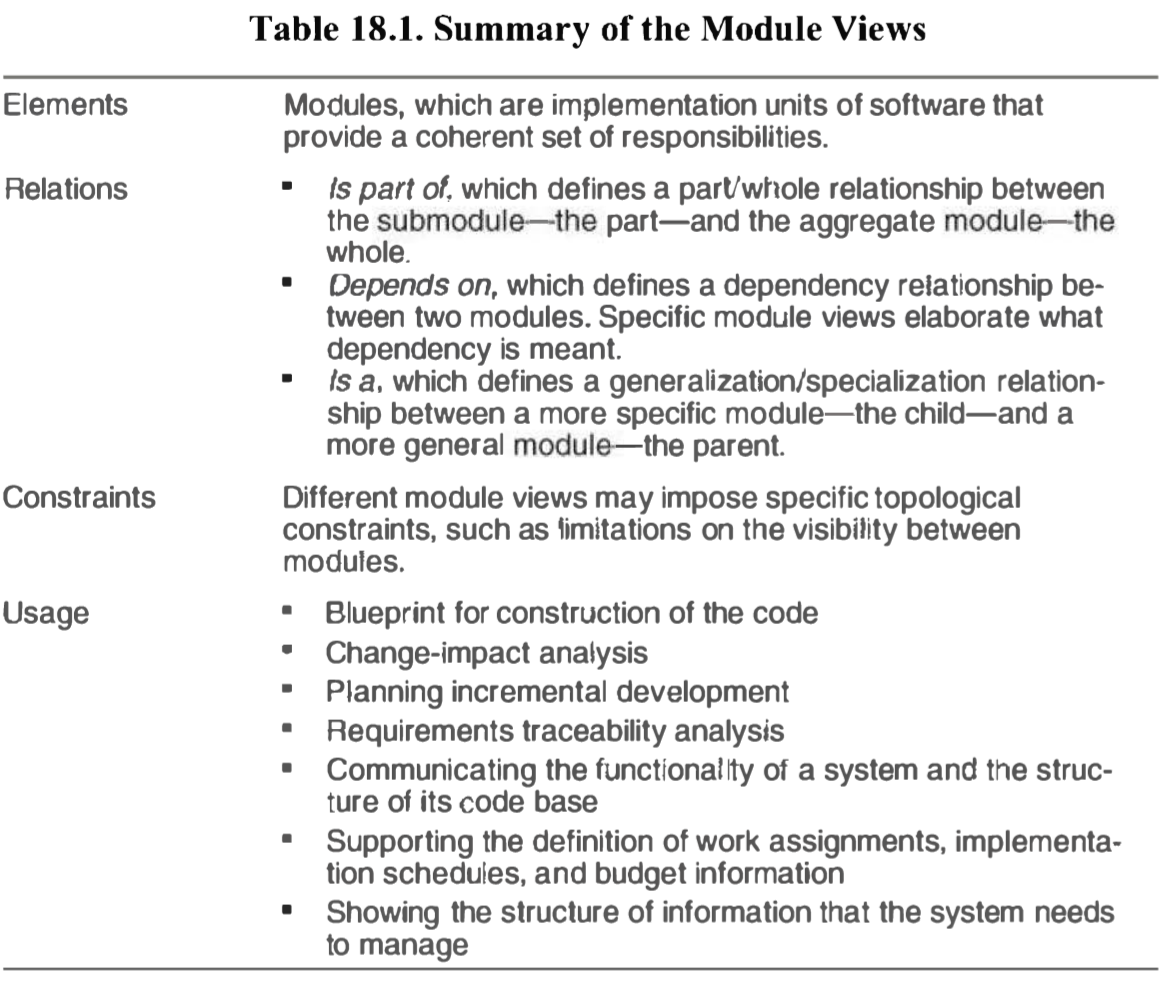
A software architecture is a complex entity that cannot be described in a simple one-dimensional fashion. A *view* is a representation of a set of system elements and relations among them, not all but those of a particular type.

* E.g. a layered view of a system would show elements of type layer, a pure layered view would not show e.g. the system’s services, clients/servers, etc.

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

What views are relevant just depends on your goal for the documentation. A layered view will let you reason about your system's portability, a deployment view will let you reason about your system's performance and reliability, etc.

**Module Views**

A module is an implementation unit that provides a coherent set of responsibilities. A module might take the form of a class, a collection of classes, a layer, etc

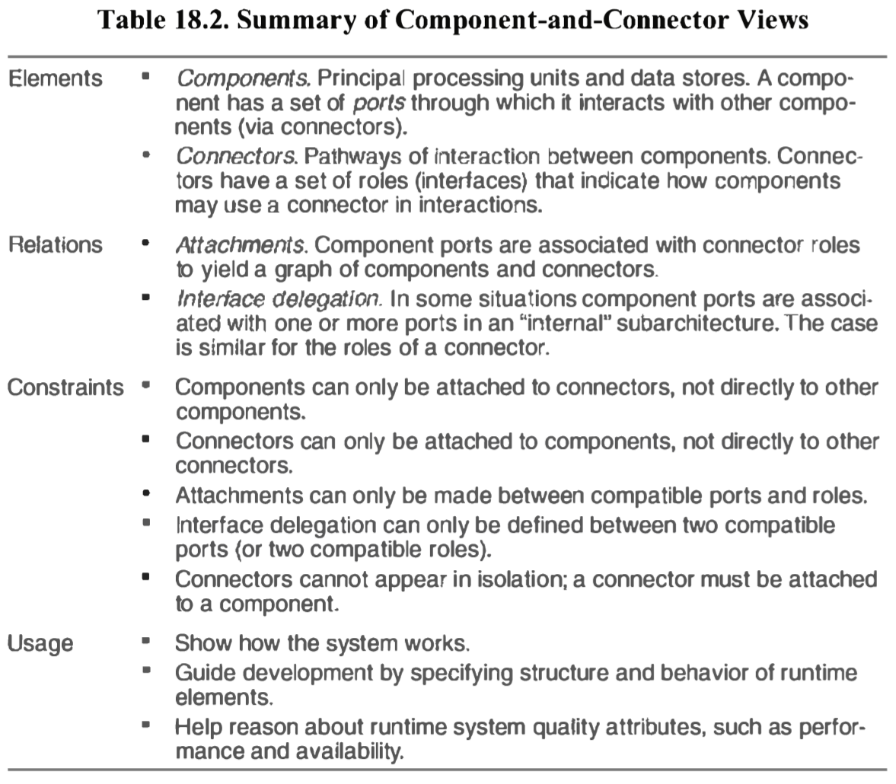
Properties of modules that help to guide implementation or are input to analysis should be recorded as part of the supporting documentation for a module view. The list of properties is likely to include the following:

* *Name*. The modules name, it often suggests something about its role in the system.
* *Responsibility*. A way to identify its role in the overall system.
* *Visibility of interfaces(s)*. When a module has submodules they can be private/public
* *Implementation Information*. Some information recoded about the implementation. Might include:
  + *Mapping to source code units*. The files that constitute the implementation of a module.
  + *Test Information*. The test plan, test cases, test scaffolding, and test data.
  + *Management Information*. E.g. the module's predicted schedule and budget.
  + *Implementation Constraints*.
  + *Revision history*.

Module views that show dependencies among modules or layers provide a good basis for change-impact analysis.  
A module view can be used to explain the system's functionality to someone not familiar with it.

**Component-and-Connector Views**

Component-and-connector views show elements that have some runtime presence, such as processes, objects, clients, servers, and data stores (*Components*). And the pathways of interaction, such as communication links and protocols, information flows, and access to shared storage (*Connectors*).

*Port*: Components have interfaces called ports. A port defines a point of potential interaction of a component with its environment. A port typically has an explicit type, which defines the kind of behaviour that can take place at that point of interaction.

*Roles*: Connectors have roles, which are its interfaces, defining the ways in which the connector may be used by components to carry out interaction.

*Attachments*: indicate which connectors are attached to which components, thereby defining a graph of connectors and components.

Typical properties:

* Reliability. What is the likelihood of failure for a given component or connector?
* *Performance*. What kinds of response time will the component provide under what loads.
* *Resource* *requirements*. What are the processing and storage needs of a component or a connector.
* *Functionality*. What functions does an element perform.
* *Security*. Does a component or a connector enforce or provide security features.
* *Concurrency*. Does this component execute as a separate process or thread.
* *Modifiability*. Does the messaging structure support a structure to cater for evolving data exchanges
* *Tier*. For a tiered topology, what tier does the component reside in.

C&C views are commonly used to show to developers and other stakeholders how the system works - one can trace through a C&C view, showing an end-to-end thread of activity.

**Notations for C&C views**

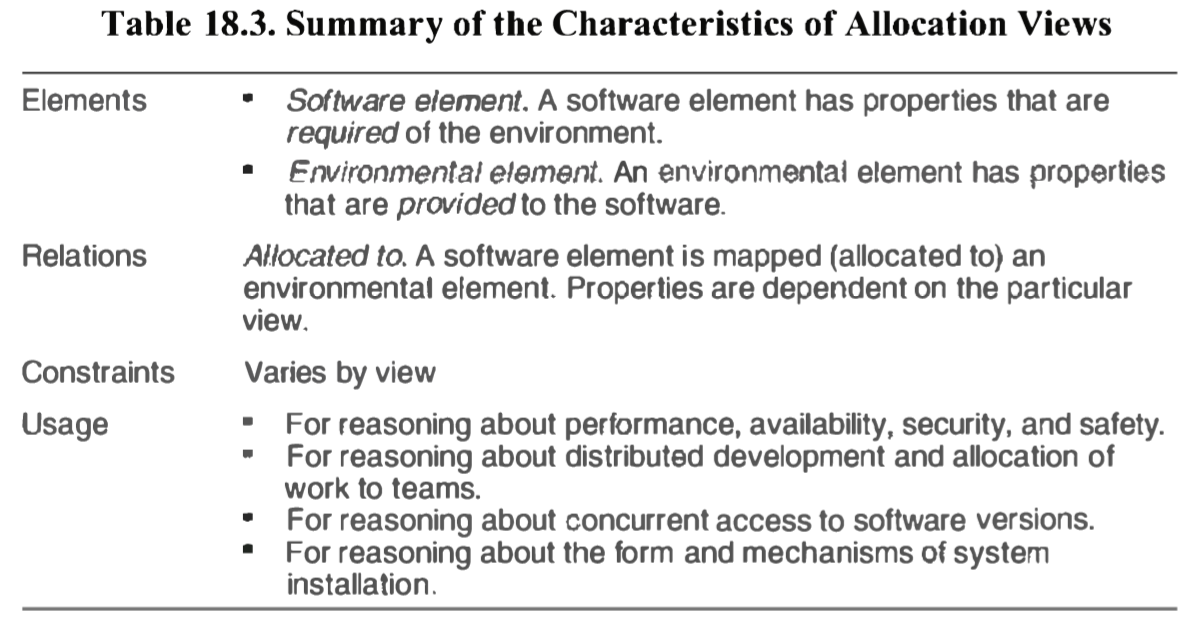
The primary guideline is simple: assign each component type and each connector type a separate visual form (symbol), and list each of the types in a key.

UML components are a good semantic match to C&C components, the same is not true of UML connectors. UML connectors cannot have substructure, attributes, or behavioural descriptions.

**Allocation Views**

Allocation views describe the mapping of software units to elements of an environment in which the software is developed or in which it executes.

The environment might be the hardware, the operating environment in which the software is executed, the file systems supporting development or deployment, or the development organization(s).

Allocation views can depict static or dynamic views. A static view depicts a fixed allocation of resources in an environment. A dynamic view depicts the conditions and the triggers for which allocation of resources changes according to loading.

**Quality Views**

Module, C&C, and allocation views are all structural views: They primarily show the structures that the architect has engineered into the architecture to satisfy functional and quality attribute requirements.

Another kind of view, which we call a quality view, can be tailored for specific stakeholders or to address specific concerns. These quality views are formed by extracting the relevant pieces of structural views and packaging them together. Example: A *reliability* *view* would be one in which reliability mechanisms such as replication and switchover are modeled. It would also depict timing issues and transaction integrity.

**18.4 Choosing the Views**

Documenting decisions during the design process produces views, which are the heart of an architecture document. It is most likely that these views are rough sketches more than finished products ready for public release.

You can determine which views are required, when to create them, and how much detail to include if you know the following:

* What people, and with what skills, are available
* Which standards you have to comply with
* What budget is on hand
* What the schedule is
* What the information needs of the important stakeholders are
* What the driving quality attribute requirements are
* What the size of the system is

Three-step method for choosing the views:

**Step 1. Build a stakeholder/view table**Enumerate the stakeholders for your project’s software architecture documentation down the rows, enumerate the views that apply to your system to the columns. 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.

**Step 2: Combine Views**Look for marginal views in the table: those that require only an overview, or that serve very few stakeholders. Combine each marginal view with another view that has a stronger constituency.

**Step 3: Prioritize and stage**At this point you need to decide what to do first. You should consider:

* The decomposition view (one of the module views) is a particularly helpful view to release early.
* Be aware that you don’t have to satisfy all the information needs of all the stakeholders.
* You don’t have to complete one view before starting another, a breadth first approach is often better.

**18.5 Combining Views**

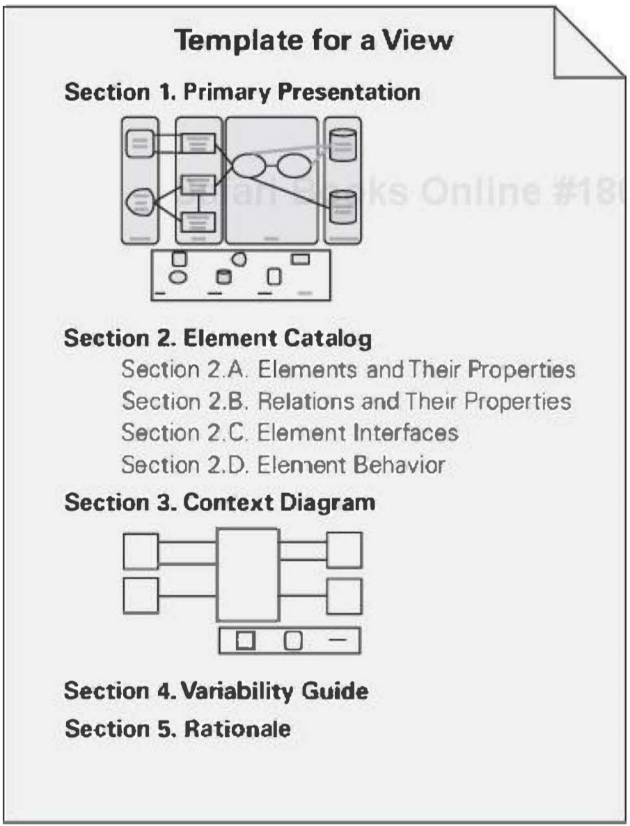
The basic principle of documenting an architecture as a set of separate views brings a divide-and conquer advantage to the task of documentation, but if the views were irrevocably different, with no association with one another, nobody would be able to understand the system as a whole.

Sometimes the most convenient way to show a strong association between two views is to collapse them into a single combined view (step 2). A combined view is a view that contains elements and relations from two or more views.

The easiest way to merge views is to create an overlay that combines the information that would otherwise have been in two separate views.

**18.6 Building the Documentation Package**

The principle of architecture documentation: Our task is to document the relevant views and to document the information that applies to more than one view.

Template that applies to all types of views:

**Primary Presentation**Shows the elements and relations of the view. The primary presentation should contain the information you wish to convey about the system.  
The primary presentation is most often graphical. It can be an informal, semiformal or formal graph. Occasionally it can be textual, such as a table or list.   
It’s role is to present a terse summary of the most important information in the view.

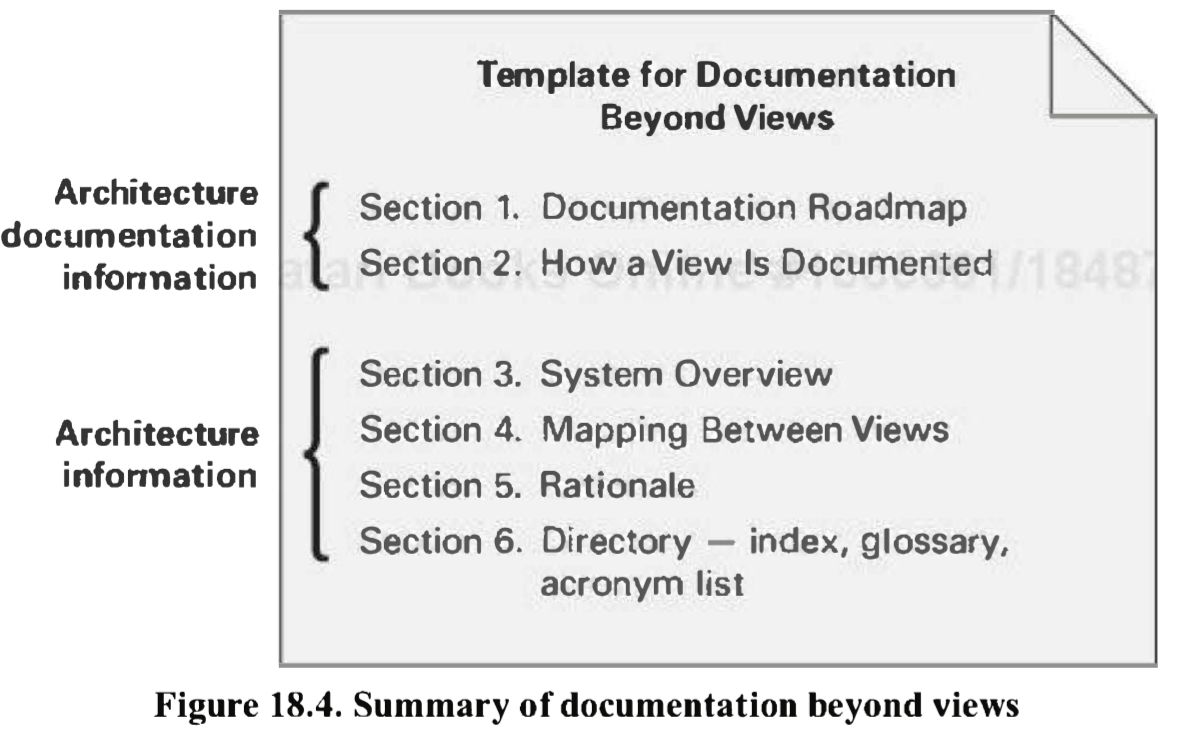
**Element Catalog**The element catalog details at least those elements depicted in the primary presentation, e.g. if a diagram shows elements A, B, and C, then the element catalog needs to explain what A, B, and C are.  
Specific parts of the catalog include:

* Elements and their properties.
* Relations and their properties.
* Element interfaces
* Element behaviour

**Context Diagram**Shows how the system or portion of the system in this view relates to its environment.   
The purpose of a context diagram is to depict the scope of a view.

**Variability Guide**Shows how to exercise any variation points that are a part of the architecture shown in this view.

**Rationale**Explains why the design reflected in the view came to be.  
The goal is to explain why the design is as it is and to provide a convincing argument that it is sound.  
The choice of a pattern in this view should be justified here by describing the architectural probletn that the chosen pattern solves and the rationale for choosing it over another.

**Documenting Information Beyond Views**

Documentation beyond views can be divided into two parts.

1. *Overview of the architecture documentation.* How is the documentation organized – makes it easier for stakeholders of the architecture to find the information they are looking for.
2. *Information about the architecture*. 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 and the way the views are related to one another.

**Section 1 – Documentation Roadmap**  
The documentation roadmap tells the reader what information is in the documentation and where to find it.   
It consists of:

* *Scope and summary*. The purpose of the documentation, what is covered/not covered.
* *How the documentation is organized*. For each section in the documentation, give a short synopsis of the information that can be found there.
* *View overview*. List of view names and their patterns.
* *How stakeholders can use the documentation*. The map follows with a section describing which stakeholders and concerns are addressed by each view (a table).

**Section 2 – How a View is Documented**This is where you explain the standard organization you're using to document views either the one described in this chapter or one of your own. It tells your readers how to find information in a view.

**Section 3 – System Overview**This is a short prose description of the system's function, its users, and any important background or constraints. This section provides your readers with a consistent mental model of the system and its purpose.

**Section 4 – Mapping Between Views**To help the reader understand the associations between views.

**Section 5 – Rationale**Documents the architectural decisions that apply to more than one view. The decisions about which fundamental architecture patterns to use are often described here.

**Section 6 – Directory**The directory is a set of reference material that helps readers find more information quickly. It includes an index of terms, a glossary, and an acronym list.

**18.7 Documenting Behaviour**

Documenting an architecture requires behaviour documentation that complements structural views by describing how architecture elements interact with each other.

There are two kinds of notations available for documenting behaviour. The first kind of notation is called trace-oriented languages; the second is called comprehensive languages.

**Traces**  
Traces are sequences of activities or interactions that describe the system’s response to a specific stimulus when the system is in a specific state. A trace describes a sequence of activities or interactions between structural elements of the system.

Four notations for documenting traces (there are more):

* *Use cases*. Describe how actors can use a system to accomplish their goals. Use cases are frequently used to capture the functional requirements for a system. The use case description is textual but a UML diagram is often used alongside as a graphical notation.
* *UML Sequence Diagram*. Shows a sequence of interactions among instances of elements pulled from the structural documentation.
* *UML Communication Diagram*. Shows a graph of interacting elements and annotates each interaction with a number denoting order. Communication diagrams are useful when the task is to verify that an architecture can fulfil the functional requirements.
* *UML Activity Diagram*. Shows a business process as a sequence of steps and include notation to express conditional branching and concurrency as well as to show sending/receiving events.

In contrast to trace notations, comprehensive models show the complete behaviour of structural elements. Given this type of documentation, it is possible to infer all possible paths from initial state to final state.  
UML state machine diagram allows you to trace the behaviour of your system, given specific inputs.

**18.8 Architecture Documentation and Quality Attributes**

If architecture is largely about the achievement of quality attributes and if one of the main uses of architecture documentation is to serve as a basis for analysis, where do quality attributes show up in the documentation? There are 5 major ways:

1. Any major design approach (e.g. pattern) will have quality attribute properties associated with it. Client-server is good for scalability, layering is good for portability, and so forth. Explaining the choice of approach is likely to include a discussion about the satisfaction of quality attribute requirements and tradeoffs incurred. In our approach, we call this discussion the *rationale*.
2. Individual architectural elements that provide a service often have quality attribute bounds assigned to them. Consumers of the services need to know how fast, secure, or reliable those services are. These quality attribute bounds are often defined in the interface documentation for the elements, or they may simply be recorded as *properties* that the elements exhibit.
3. Quality attributes often impart a "language" of things that you would look for. Security involves security levels, authenticated users, availability conjures up mean time between failure, fail over mechanism, etc. Someone fluent in the "language" of a quality attribute can search for the kinds of architectural elements.
4. The documentation often contains a *mapping to requirements* that shows how requirements (including quality attribute requirements) are satisfied. If your requirements document establishes a requirement for e.g. availability, then you should be able to look it up by name or reference in your architecture document to see the places where that requirement is satisfied.
5. Every quality attribute requirement will have a constituency of stakeholders who want to know that it is going to be satisfied. For these stakeholders, the architect should provide a special place in the documentation's introduction that either provides what the stakeholder is looking for, or tells the stakeholder where in the document to find it -> documentation roadmap.

**18.9 Documenting Architectures That Change Faster Than You Can Document Them**

Here's what you can do if you're an architect in a highly dynamic environment.

* *Document what is true about all versions of your system.*
* *Document the ways the architecture is allowed to change.*

**18.10 Documenting Architecture in an Agile Development Project**

The Views and Beyond and Agile philosophies agree strongly on a central point: If information isn't needed, don't document it

When producing Views and Beyond-based architecture documentation using Agile principles, keep the following in mind:

* Adopt a template or standard organization to capture your design decisions.
* Plan to document a view if (but only if) it has a strongly identified stakeholder.
* Fill in the template for a view when the information becomes available.
* Produce just enough design information to allow you to move on to code.
* Don't feel obliged to fill up all sections of the template, and certainly not all at once.
* Agile teams sometimes make models in brief discussions by the whiteboard. When documenting a view, the primary presentation may consist of a digital picture of the white board.

**Chapter 19 – Architecture, Implementation and Testing**

**19.1 Architecture and Implementation**

Architecture is intended to serve as the blueprint for implementation.  
It is very easy for code and its intended architecture to drift apart; this is sometimes called "architecture erosion." This section talks about four techniques to help keep the code and the architecture consistent.

**Embedding the Design in the Code**A key task for implementers is to faithfully execute the prescriptions of the architecture.  
Throughout the code, implementers can document the architectural concept or guidance that they're reifying. That is, they can "embed" the architecture in their implementations. They can also try to localize the implementation of each architectural element, as opposed to scattering it across different implementation entities.

**Frameworks**Frameworks that are large and sophisticated often encode architectural interaction mechanisms, by encoding how the classes (and the objects derived from them) communicate and synchronize with each other.  
A framework amounts to a substantial (in some cases, enormous) piece of reusable software, and it brings with it all of the advantages of reuse: saving time and cost, avoiding a costly design task, encoding domain knowledge, and decreasing the chance of errors from individual implementers coding the same thing differently and erroneously. On the other hand, frameworks are difficult to design and get correct.

**Code Templates**Using a template has architectural implications: it makes it simple to add new applications to the system with a minimum of concern for the actual workings of the fault-tolerant mechanisms designed into the approach. Coders and maintainers of applications do not need to know about message-handling mechanisms except abstractly, and they do not need to ensure that their applications are fault tolerant that has been handled architecturally.  
Templates represent a true common ground where the architecture and the implementation come together in a consistent and useful fashion.

**Keeping Code and Architecture Consistent**Code can drift away from architecture in a depressingly large number of ways.   
*First*, there may be no constraints imposed on the coders to follow the architecture.  
*Second*, often when problems are encountered, the architecture is abandoned.   
*Third*, after the system has been fielded, changes to it are accomplished with code changes only, but not recorded in the architecture

One simple method to remedy the lack of updating the architecture is to not treat the published architecture as an all-or-nothing affair, parts may become out of date, but simply marking those sections keeps the documentation alive.

**19.2. Architecture and Testing**

**Levels of Testing and How Architecture Plays a Role in Each**

*Unit Testing*Unit testing refers to tests run on specific pieces of software. Unit testing is usually a part of the job of implementing those pieces.   
A unit corresponds to an architectural element in one of the architecture's module views. In object-oriented software, a unit might correspond to a class. In a layered system, a unit might correspond to a layer, or a part of a layer.  
Architecture plays a strong role in unit testing. First, it defines the units: they are architectural elements in one or more of the module views. Second, it defines the responsibilities and requirements assigned to each unit.

*Integration Testing*Integration testing tests what happens when separate software units start to work together. At the end of integration testing, the project has confidence that the pieces of software work together correctly and provide at least some correct system-wide functionality.  
Once again, architecture cannot help but play a strong role in integration testing. First, the increments that will be subject to integration testing must be planned, and this plan will be based on the architecture. The uses view is particularly helpful for this, as it shows what elements must be present for a particular piece of functionality to be fielded.

*Acceptance Testing*Acceptance testing is a kind of system testing that is performed by users, often in the setting in which the system will run.   
Architecture plays less of a role in acceptance testing than at the other levels, but still an important one. Acceptance testing involves stressing the system's quality attribute behaviour by running it at extremely heavy loads, subjecting it to security attacks, depriving it of resources at critical times, and so forth.

**Black-Box and White-Box Testing**Black-box testing treats the software as an opaque "black box," not using any knowledge about the internal design, structure, or implementation. The tester's only source of information about the software is its requirements.  
Architecture plays a role in black-box te-sting, because it is often the architecture document where the requirements for a piece of the system are described.

White-box testing makes full use of the internal structures, algorithms, and control and data flows of a unit of software. Tests that exercise all control paths of a unit of software are a primary example of white-box testing.

Gray-box testing lies, as you would expect, between black and white. Testers get to avail themselves of some, but not all, of the internal structure of a system. For example, they can test the interactions between components but not employ tests based on knowledge of a component's internal data structures.

**Risk-based Testing**Risk -based testing concentrates effort on areas where risk is perceived to be the highest, perhaps because of immature technologies, requirements uncertainty, developer experience gaps, and so forth.

Architecture can inform risk-based testing by contributing categories of risks to be considered. Architects can identify areas where architectural decisions (if wrong) would have a widespread impact, where architectural requirements are uncertain, quality attributes are demanding on the architecture, technology selections risky, or third-party software sources unreliable.

**Test Activities**Testing, depending on the project, can consume from 30 to 90 percent of a development's schedule and budget. Here are some of the activities associated with testing:

* *Test Planning*. Test activities have to be planned so that appropriate resources can be allocated.
* *Test Development*. This is an activity in which the test procedures are written, test cases are chosen, test datasets are created, and test suites are scripted.
* *Test Execution*. Testers apply the tests to the software and capture and record errors.
* *Test Reporting and Defect Analysis*. Testers report the results of specific tests to developers, and they report overall metrics about the test results to the project's technical management.
* *Test Harness Creation*.

**The Architect's Role**Here are some of the things an architect can do to facilitate quality testing. First and foremost, the architect can design the system so that it is highly testable. The architect can work with the test team to establish what is needed, and together they can come up with a definition of the testability requirements using scenarios.

In addition to designing for testability, the architect can do the following to help:

* Insure that testers have access to the source code, design documents, and the change records
* Give testers the ability to control and reset the entire dataset that a program stores in a persistent database
* Give testers the ability to install multiple versions of a software product on a single machine

**Chapter 20 – Architecture Reconstruction and Conformance**

Architecture Reconstruction is the process of reverse engineering where you build, maintain and understand a representation of an existing architecture. Two main purposes of architecture reconstruction:

1. To document an architecture where the documentation never existed or where it has been come out of date
2. To ensure conformance between the as-built architecture and the as-designed architecture

In architecture reconstruction, the "as-built" architecture of an implemented system is reverseengineered from existing system artifacts.

**20.1 Architecture Reconstruction Process**

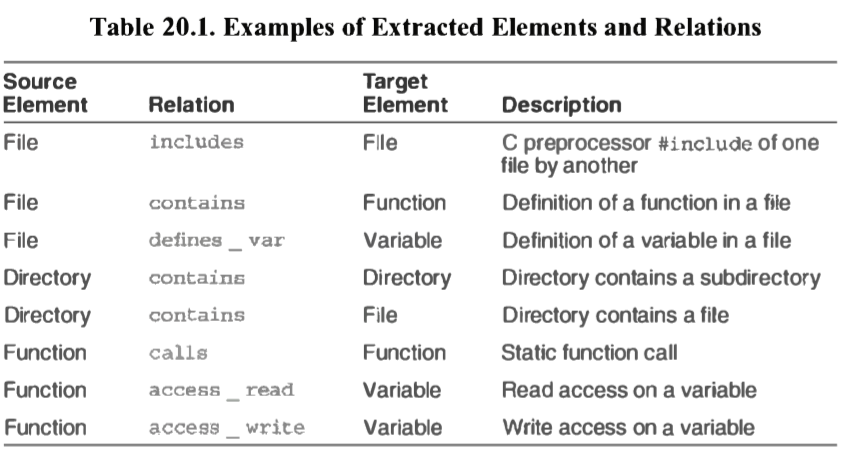
The process comprises the following phases.

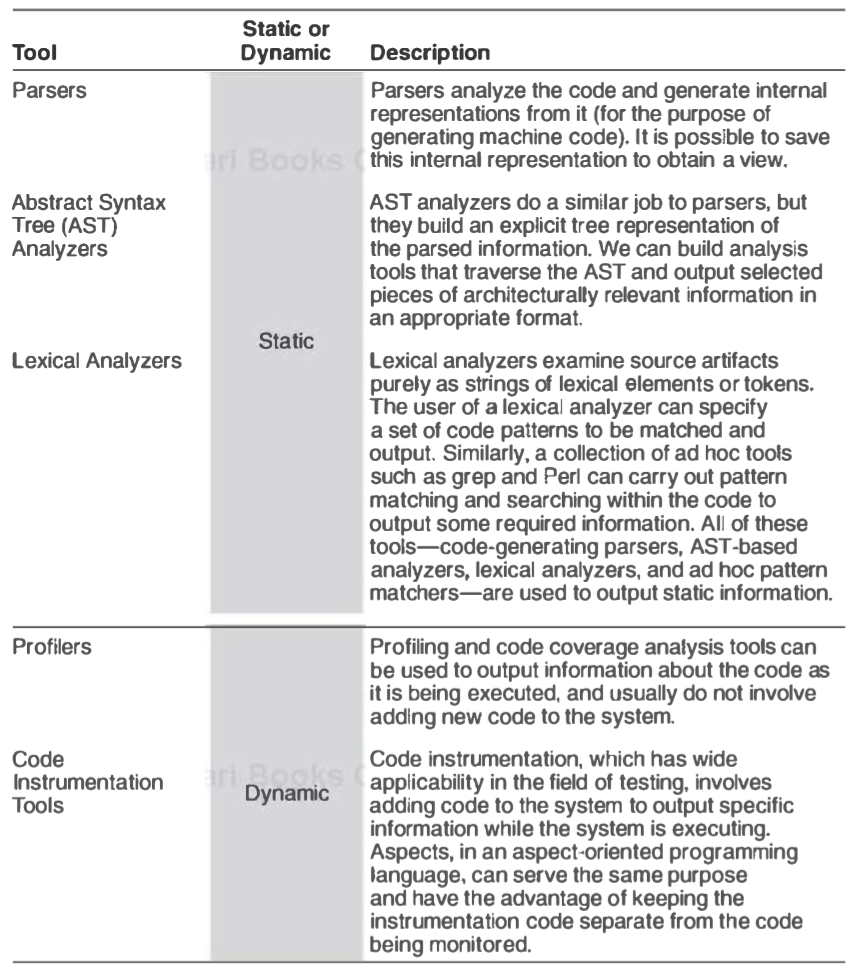
1. *Raw view extraction*. In the raw view extraction phase, raw information about the architecture is obtained from various sources, primarily source code, execution traces, and build scripts. Each of these sets of raw information is called a view.
2. *Database construction*. The database construction phase involves converting the raw extracted information into a standard form, the standard form is then used to populate a reconstruction database.
3. *View fusion and manipulation*. The view fusion phase combines the various views of the information stored in the database. Individual views may not contain complete or fully accurate information. View fusion can improve the overall accuracy. Furthermore, view creation and fusion is typically associated with some expert interpretation and manipulation. For example, an expert might decide that a group of elements should be aggregated together to form a layer.
4. *Architecture analysis*. View fusion will result in a set of hypotheses about the architecture. These hypotheses need to be tested to see if they are correct, and that is the function of the analysis step.

The four phases of architecture reconstruction are iterative

\* A view: a representation of a set of system elements and relations among them

**20.2 Raw View Extraction**

From the source artifacts (code, header files, build files, and so on) and other artifacts (e.g., execution traces), you can identify and capture the elements of interest within the system (e. g., files, functions, variables) and their relationships to obtain several base system views.

Common tools for populating views:

**20.3 Database Construction**

Some of the information extracted from the raw view extraction phase, while necessary for the process of reconstruction, may be too specific to aid in architectural understanding.  
We need to manipulate such raw views, to collapse information (for example, hiding methods inside class definitions), and to show abstractions (for example, showing all of the connections between business objects and user interface objects, or identifying distinct layers)  
There exists good tools for this, e.g. Lattix and SonarJ, they fully encapsulate the database so the user of the tool don’t need to be concerned with is operation.

**20.4 View Fusion**

In this phase, the extracted views are manipulated to create fused views. Fused views combine information from one or more extracted views, each of which may contain specialized information.

The process of creating a fused view is the process of creating a hypothesis about the architecture and a visualization of it to aid in analysis. These hypotheses result in new aggregations that show various abstractions or clusterings of the elements. By interpreting these fused views and analyzing them, it is possible to produce hypothesized architectural views of the system. These views can be interpreted, further refine There are no universal completion criteria for this process; it is complete when the architectural representation is sufficient to support the analysis needs of its stakeholders.

**20.5 Architecture Analysis: Finding Violations**

Consider the following situation: You have designed an architecture but you have suspicions that the developers are not faithfully implementing what you developed.  
Whatever the root cause, this divergence of the architecture and the implementation spells problems for you, the architect. So how do you test and ensure conformance to the design? 2 Major possibilities:

* *Conformance by Construction.* Automatically generating a substantial part of the system based on an architectural specification. For most systems, this is not possible.
* *Conformance by Analysis.* This technique aims to ensure conformance by analyzing system information to flag nonconforming elements, so that they can be fixed – brought into conformance.