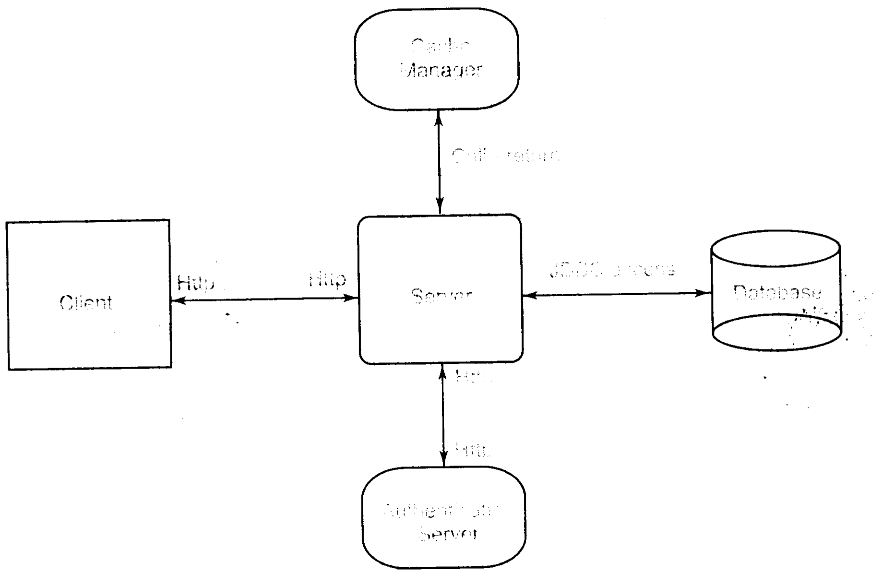

Figure 4.6: Architecture for the survey system with cache.

system). (Of course, detail level decisions like how a particular module is implemented also has implications on performance, but they are quite distinct and orthogonal to the architecture-level decisions.) We will later do a formal evaluation of these different architectures to see the impact of architectural decisions on some quality attributes.

4.4 Architecture Styles for C&C View

As mentioned above, an architecture view describes a structure of the system in terms of its elements and relationships among them. Clearly, different systems will have different structures, even for the same view. There are, however, some structures and related constraints that have been observed in many systems and that seem to represent general structures that are useful for architecture of a class of problems. These are called architectural styles. A style defines a family of architectures that satisfy the constraints of that style [35, 9, 135].

For example, for module views, some of the common styles are decomposition, uses, generalization, and layered. In decomposition style, a module is decomposed into submodules, and the system becomes a hierarchy of modules. In the uses style, modules are not parts of each other, but a module uses services of other modules (for example, a function call or a method invocation) to correctly do its own work. In the generalization style, modules are often classes, and a child class inherits the properties of the parent class and specializes it. This supports an is-a hierarchy, where a child module is also of the parent module type. In the layered style, the system is structured as a stack of layers, each layer representing some virtual machine that provides a clear set of services. In addition, a layer is allowed to use services only of its adjacent layers. Some of these will be discussed later in the design chapters.

In this section we discuss some common styles for the C&C view. Many styles have been proposed for C&C view, some for specific domains. Here we discuss only a few that are widely discussed in literature and which can be useful for a large set of problems [135, 35]. These styles can provide ideas for creating an architecture view for the problem at hand. Styles can be combined to form richer views. In fact, it is likely that for a problem, a combination of styles may provide the desired architecture. Many systems use multiple styles and different parts of a system may use different styles.

4.4.1 Pipe and Filter

Pipe and filter style of architecture is well suited for systems that primarily do data transformation some input data is received and the goal of the system is to produce some output data by suitably transforming the input data. A system using pipe-and-filter architecture achieves the desired transformation by applying a network of smaller transformations and composing them in a manner that together the overall desired transformation is achieved.

The pipe and filter style has only one component type called the filter. It also has only one connector type, called the pipe. A filter performs a data transformation, and sends the transformed data to other filters for further processing using the pipe connector. In other words, a filter receives the data it needs from some defined input pipes, performs the data transformation, and then sends the output data to other filters on the defined output pipes. A filter may have more than one inputs and more than one outputs. Filters can be independent and asynchronous entities, and as they are concerned only with the data arriving on the pipe, a filter need not know the identity of the filter that sent the input data or the identity of the filter that will consume the data they produce.

The pipe connector is a unidirectional channel which conveys streams of data received on one end to the other end. A pipe does not change the data in any manner but merely transports it to the filter on the receiver end in the order in which the data elements are received. As filters can be asynchronous and should work without the knowledge of the identity of the producer or the consumer, buffering and synchronization need to ensure smooth functioning of the producer-consumer relationship embodied in connecting two filters by a pipe is ensured by the pipe. The filters merely consume and produce data.

There are some constraints that this style imposes. First, as mentioned above, the filters should work without knowing the identity of the consumer or the producer; they should only require the data elements they need. Second, a pipe, which is a two-way connector, must connect an output port of a filter to an input port of another filter.

A pure pipe-and-filter structure will also generally have a constraint that a filter has independent thread of control which process the data as it comes. Implementing this will require suitable underlying infrastructure to support a pipe mechanism which buffers the data and does the synchronization needed (for example, blocking the producer when the buffer is full and blocking the consumer filter when the buffer is empty). For using this pipe, the filter builder must be fully aware of the properties of the pipe, particularly with regards to buffering and synchronization, input and output mechanisms, and the symbols for end of data.

However, there could be situations in which the constraint that a filter process the data as it comes may not be required. Without this constraint, pipe-and-filter style view may have filters that produce the data completely before passing it on, or which start their processing only after complete input is available. In such a system the filters cannot operate concurrently, and the system is like a batch-processing system. However, it can considerably simplify the pipes and easier mechanisms can be used for supporting them.

Lets consider an example of a system needed to count the frequency of different words in a file. An architecture using the pipes-and-filter style for a system to achieve this is given in Figure 4.7.

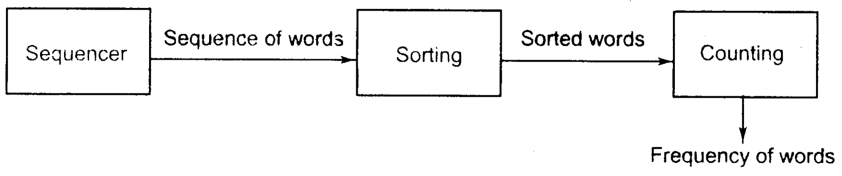

Figure 4.7: Pipe-and-Filter example.

This architecture proposes that the input data be first split into a sequence of words by a component Sequencer. This sequence of words is then sorted by the component Sorting, which passes the output of sorted words to another filter (Counting) that counts the number of occurrences of the different words. This structure of sorting the words first has been chosen as it will make the task of determining the frequency more efficient, even though it involves a sort operation. It should be clear that this proposed architecture can implement the desired functionality. Later in the chapter we will further discuss some implementation issues related to this architecture.

As can be seen from this example, pipe and filter architectural style is well suited for data processing and transformation. Consequently, it is useful in text processing applications. Signal processing applications also find it useful as such applications typically perform encoding, error correction, and other transformations on the data.

The pipe and filter style, due to the constraints, allows a system's overall transformation to be composed of smaller transformations. Or viewing it in another manner, it allows a desired transformation to be factored into smaller transformations, and then filters built for the smaller transformations. That is, it allows the techniques of functional composition and decomposition to be utilized something that is mathematically appealing.

4.4.2 Shared-Data Style

In this style, there are two types of components—data repositories and data accessors. Components of data repository type are where the system stores shared data—these could be file systems or databases. These components provide a reliable and permanent storage, take care of any synchronization needs for concurrent access, and provide data access support. Components of data accessor type access data from the repositories, perform computation on the data obtained, and if they want to share the results with other components, put the results back in the depository. In other words, the accessors are computational elements that receive their data from the repository and save their data in the repository as well. These components do not directly communicate with each other—the data repository components are the means of communication and data transfer between them.

There are two variations of this style possible. In the blackboard style, if some data is posted on the data repository, all the accessor components that need to know about it are informed. In other words, the shared data source is an active agent as well which either informs the components about the arrival of interesting data, or starts the execution of the components that need to act upon this new data. In databases, this form of style is often supported through triggers. The other is the repository style, in which the data repository is just a passive repository which provides permanent storage and related controls for data accessing. The components access the repository as and when they want.

As can be imagined, many database applications use this architectural style. Databases, though originally more like repositories, now act both as repositories as well as blackboards as they provide triggers and can act as efficient data storage as well. Many Web systems frequently follow this style at the back end—in response to user requests, different scripts (data accessors) access and update some shared data. Many programming environments are also organized this way the common representation of the program artifacts is stored in the repository and the different tools access it to perform the desired translations or to obtain the desired information. (Some years back there was a standard defined for the common repository to facilitate integration of tools.)

As an example of a system using this style of architecture, let us consider a student registration system in a University. The system clearly has a central repository which contains information about courses, students, pre-requisites, etc. It has an `Administrator` component that sets up the repository, rights to different people, etc.

The `Registration` component allows students to register and update the information for students and courses. The `Approvals` component is for granting approvals for those courses that require instructor's consent. The `Reports` component produces the report regarding the students registered in different courses at the end of the registration. The component `Course Feedback` is used for taking feedback from students at the end of the course. This architecture is shown in Figure 4.8.

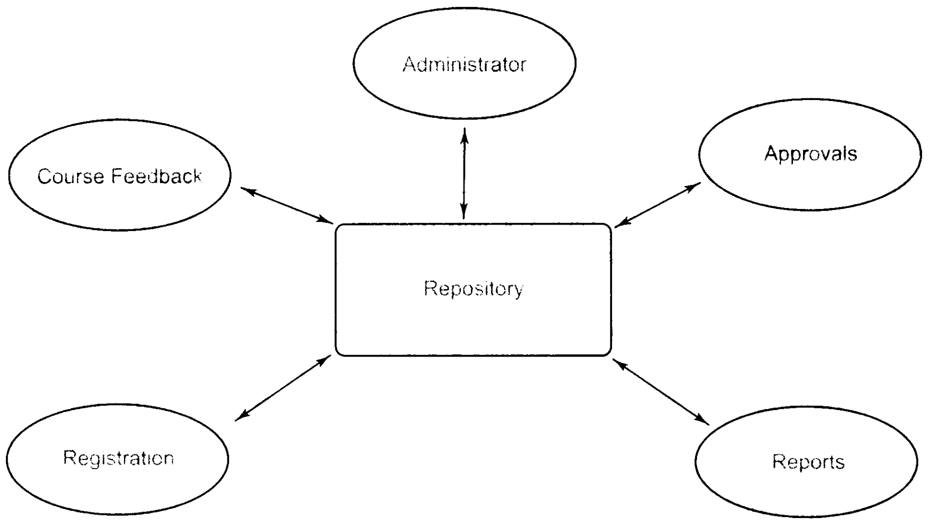

Figure 4.8: Shared data example.

Note that the different computation components do not need to communicate with each other and do not even need to know about each others presence. For example, if later it is decided that the scheduling of courses can be automated based on data on registration (and other information about class rooms etc.), then another component called `Scheduling` can be simply added. No existing computation component needs to change or be informed about the new component being added. (This example is based on a system that is actually used in the author's University.)

There is really only one connector type in this style—read/write. Note, however, that this general connector style may take more precise form in particular architectures. For example, though a database can be viewed as supporting read and updates, for a program interacting with it, the database system may provide transaction services as well. Connectors using this transaction service allow complete transactions (which may involve multiple reads and writes and preserve atomicity) to be performed by an application.

Note also that as in many other cases, the connectors involve a considerable amount of underlying infrastructure. For example, read and writes to a file system involves a fair amount of file system software involving issues like directories, buffering, locking, and synchronization. Similarly, a considerable amount of software goes in databases to support the type of connections it provides for query, update, and transactions. We will see another use of this style later when we discuss the case studies.

4.4.3 Client-Server Style

Another very common style used to build systems today is the client server style. Client-server computing is one of the basic paradigms of distributed computing and this architecture style is built upon this paradigm.

In this style, there are two component types—clients and servers. A constraint of this style is that a client can only communicate with the server, and cannot communicate with other clients. The communication between a client component and a server component is initiated by the client the client sends a request for some service that the server supports. The server receives the request at its defined port, performs the service, and then returns the results of the computation to the client who requested the service.

There is one connector type in this style—the request/reply type. A connector connects a client to a server. This type of connector is asymmetric—the client end of the connector can only make requests (and receive the reply), while the server end can only send replies in response to the requests it gets through this connector. The communication is frequently synchronous—the client waits for the server to return the results before proceeding. That is, the client is blocked at the request, untill it gets the reply.

A general form of this style is a *n-tier* structure. In this style, a client sends a request to a server, but the server, in order to service the request, sends some request to another server. That is, the server also acts as a client for the next tier. This hierarchy can continue for some levels, providing a n-tier system. A common example of this is the 3-tier architecture. In this style, the clients that make requests and receive the final results reside in the client-tier. The middle tier, called the business-tier, contains the component that processes the data submitted by the clients and applies the necessary business rules. The third tier is the database tier in which the data resides. The business tier interacts with the database tier for all its data needs.

Most often, in a client server architecture, the client and the server component reside on different machines. Even if they reside on the same machine, they are designed in a manner that they can exist on different machines. Hence, the connector between the client and the server is expected to support the request/result type of connection across different machines. Consequently, these connectors are internally quite complex and involve a fair amount of networking to support. Many of the client-server systems today use TCP ports for their connectors. The Web uses the HTTP for supporting this connector.

Note that there is a distinction between a layered architecture and a tiered architecture. The tiered style is a component and connector architecture view in which each tier is a component, and these components communicate with the adjacent ones through a defined protocol. A layered architecture is a module view providing how modules are organized and used. In the layered organization, modules are organized in layers with modules in a layer allowed to invoke services only of the modules in the layer below. Hence, layered and tiered represent two different views. We can have a n-tiered architecture in which some tier(s) have a layered architecture. For example, in a client-server architecture, the server might have a layered architecture, that is, modules that compose the server are organized in the layered style.

4.4.4 Some Other Styles

Publish-Subscribe Style

In this style, there are two types of components. One type of component subscribes to a set of defined events. Other types of components generate or publish events. In response to these events, the components that have published their intent to process the event, are invoked. This type of style is most natural in user interface frameworks, where many events are defined (like mouse click) and components are assigned to these events. When that event occurs, the associated component is executed. As is the case with most connectors, it is the task of the runtime infrastructure to ensure that this type of connector (i.e., publish-subscribe) is supported. This style can be seen as a special case of the blackboard style, except that the repository aspect is not being used.

Peer-to-peer style, or object-oriented style

If we take a client server style, and generalize each component to be a client as well as a server, then we have this style. In this style, components are peers and any component can request a service from any other component. The object oriented computation model represents this style well. If we view components as objects, and connectors as method invocations, then we have this style. This model is the one that is primarily supported through middleware connectors like CORBA or .NET.

Communicating processes style

Perhaps the oldest model of distributed computing is that of communicating processes. This style tries to capture this model of computing. The components in this model are processes or threads, which communicate with each other either with message passing or through shared memory. This style is used in some form in many complex systems which use multiple threads or processes.

4.5 Discussion

Software architecture is an evolving area, and there are many issues that have not been fully resolved. In this section we discuss some issues in order to further clarify concepts relating to architecture and designing of architectures.

4.5.1 Architecture and Design

We have seen that while creating an architecture, the system may be partitioned in different ways, each providing a view that focuses on partitioning the system into parts to highlight some structure of the system. Views may highlight the runtime structure by showing the components that exist in the system and how the components interact, or the module structure which shows how the code modules are organized for the system, or the deployment structure which focuses on how the different units are assigned to resources.

As partitioning a system into smaller parts and composing the system from these parts is also a goal of design, a natural question is what is the difference between a design and architecture as both aim to achieve similar objectives and seem to fundamentally rely on the divide and conquer rule? First, it should be clear that architecture is a design in that it is in the solution domain and talks about the structure of the proposed system. Furthermore, an architecture view gives a high level view of the system, relying on abstraction to convey the meaning—something which design also does. So, architecture is design.

We can view architecture as a very high-level design, focusing only on main components, and the architecture activity as the first step in design. What we term as design is really about the modules that will eventually exist as code. That is, they are a more concrete representation of the implementation (though not yet an implementation). Consequently, during design lower level issues like the data structures, files, and sources of data, have to be addressed, while such issues are not generally significant at the architecture level. We also take the view that design can be considered as providing the module-view of the architecture of the system. As discussed before, we believe the third level of design (which we call the detailed design) is to design the logic of the various modules and their functions.

The boundaries between the first two levels of design—architecture and high-level design are not fully clear. The way the field has evolved, we can say that the line between architecture and design is really up to the designer or the architect. At the architecture level, one needs to show only those parts that are needed to perform the desired evaluation. The internal structure of these parts is not important. On the other hand, during design, designing the structure of the parts that can lead to constructing them is one of the key tasks. However, which parts of the structure should be examined and revealed during architecture and which parts during design is a matter of choice.

Generally speaking, details that are not needed to perform the types of analysis we wish to do at the architecture time are unnecessary and should be left for design to uncover.

It should, however, be pointed out that having an architecture imposes constraints on choices that can be made during the design, and while creating the design of the system these constraints should be honored. We discuss this issue a bit more later in this section. Design might also have an influence on architecture—during design some shortcomings in the architecture from the design perspective might be revealed. This may require the architecture to be modified. This situation, however, is the same with any stage—a later stage may require output of an earlier stage to be modified.

4.5.2 Preserving the Integrity of an Architecture

What is the role of the architecture as the project moves forward and design and development is done? Many novice designers treat architectures as just pictures which help in understanding the system but which have no role to play while code is being developed. A mistake made by novices and professionals alike is that after the architecture is discussed and agreed, when the teams go to build the system, they build it the way they want to without regards to the architectural decisions. One informal study at an organization revealed that the architecture extracted from the code of a product (the actual architecture of the software) had little resemblance to the architecture that was planned and was supposed to be implemented. There are many reasons for this which we will not go into, but lack of communication and enforcement are important contributors.

For an architecture design to be meaningful, it should guide the design and development of the system. And if the integrity of the architecture is fully preserved, the architecture of the final system should be the same as the architecture designed and agreed to during the early stages of the project. That is, the implementation must have the components shown in the C&C view, and the components should interact in the manner prescribed in the view.

It is important that the architecture integrity is preserved as otherwise the value of the architecture is minimal. Furthermore, one of the key reasons for designing an architecture (rather than just letting some architecture evolve as the system is built) is to be able to analyze the system properties well before the system is built and evolve optimal strategies for the final system. This means that accepting an architecture is far more than just agreeing that the architecture is correct (in that it can implement the system). The process of acceptance will generally involve formal or informal analysis of the architecture, and final acceptance also means that the system having the proposed architecture will have the properties desired for the system. By deviating from the architecture, we may go in a direction in which the system does not have the desired properties. Hence, it is extremely important that the integrity of the architecture is preserved.

Let us understand the issue of architecture integrity preservation through an example. Earlier, we discussed the architecture of a system to determine the frequency of different words in a text file. The architecture designed and agreed is shown in Figure 4.7. The architecture says that the system, in its execution, will have a component to split the file in words, another component to sort the words, and the third component to count the frequency of different words. The data is passed between the components using pipes.

Now consider the following implementation of the system. Each of the components is written as a complete C program. The first program creates a pipe and writes to it. The second program reads from this pipe and writes to another pipe. The third program reads from this new pipe. The overall system is a small script that executes these programs in parallel.

This implementation is clearly consistent with the architecture view that was designed earlier. Each of the three components have a runtime presence (they will be executed as separate processes by UNIX,) and they pass data using the UNIX pipe, which has the same semantics as the pipe connector of the pipe-and-filter style, in that it provides a buffered, streaming, data passing mechanism. Hence, it is easy to see that the C&C architecture view of this implementation is same as the architecture given earlier in Figure 4.7, and this implementation preservers the architectural integrity.

The above implementation clearly has huge performance overhead in that separate processes are created and the communication through UNIX pipes can be quite expensive. Now consider the following implementation, which avoids these performance problems by having the same functions in the system, but organizing them differently. The system, written in C, has three functions corresponding to the three components, and two functions for the two connectors. The connector function invokes one component function for getting the data, and then passes the data to the other component function. In the actual implementation, to further reduce the overhead, we let the main function perform the roles of both the connector functions. The structure is shown in Figure 4.9.

Does this implementation preserve the architecture integrity? The answer to this is not totally unambiguous. The three functions implement the same functionality as the components in the architecture design. But can they be treated as separate components, as they are really just functions? The answer is not obvious. The components satisfy some properties of the filters in that they do not need to know the identity of the components from which they get the data or send the data. The connector in this implementation is really the parameter passing mechanism. Though the data is being passed as an organized series of data items, it is really not streaming or buffered in that complete data is passed at once. Furthermore, this structure does not allow the three components to execute in parallel; they are executed strictly in sequence.

Still, this implementation can be viewed as preserving the architectural integrity, particularly if no restrictions were explicitly placed on the components (in terms of

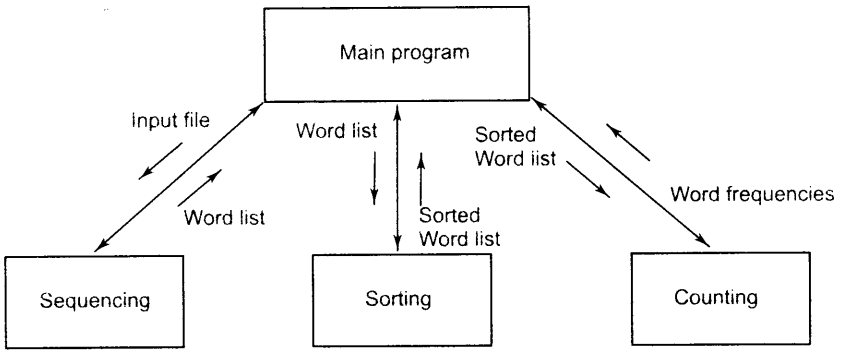

Figure 4.9: Another implementation of the system.

parallel execution) and the connectors (in terms of buffering and data passing). Note that these restrictions could have been placed explicitly in the architecture description or implicitly during its analysis—for example, the analysis may have assumed that the components are executing in parallel and may have taken data transfer speeds to be such that they are consistent with a separate pipe.

Now consider a third implementation. A designer and a coder look at the problem statement (and do not look at the architecture designed) and decide that an easy way to implement the problem is to get a word at a time and then simply increment the count for the word, if it already exists in the frequency list, or add the word if it does not. In this, we have a main C program which has a word extractor and a frequency counter. When the data is finished, the frequency output program shows the frequencies of the different words. The structure of this implementation is shown in figure 4.10.

This implementation clearly is not consistent with the architecture designed above. It does not have the same components and the processing approach is different from the one envisaged in the architecture. It should be pointed out that this implementation is also correct for the problem statement (the system does implement the requirements). Note that this mismatch of architecture is largely due to the approach and not due to the fact that all components are combined in one program. Even if the implementation was done using separate programs which pass the data through pipes (an implementation that can be easily done for this approach), the mismatch between the architecture of the system and the designed architecture will remain.

It is this type of mismatch that frequently occurs. If the implementers of the system start from the requirements and do not consider the architecture that has been created, they can create a correct system that may provide the functionality, but the architecture integrity may not be preserved. This can have other consequences. For example, the performance of the final system may not be as was envisaged when the architecture

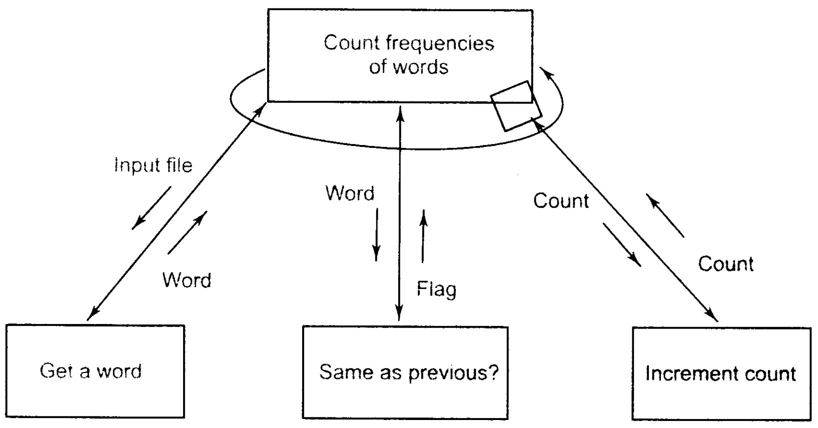

Figure 4.10: A third implementation.

was designed as the system may get engineered in a manner that it is unable to provide the performance. By preserving the architecture, if the architecture was designed and evaluated properly, the final engineered system is more likely to fulfill the other properties. Besides performance, an architecture may be evaluated for other properties like maintainability, suitability for use with the available resources, etc. By not preserving the integrity of the architecture, we risk not meeting requirements for these properties.

This example illustrates that it is easy to fall in the trap of ignoring the architecture and building the system from scratch. This is a pitfall that system designers and builders must avoid. The architecture design imposes some restrictions on how the system can be built and engineered, and these constraints and restrictions should be honored and the system should be built in a manner that the architecture of the final system is same as the architecture that was conceived (except, of course, the changes that were intentionally done after careful consideration and discussion).

4.5.3 Deployment View and Performance Analysis

As discussed earlier, there are different views through which software architectures can be represented and which views we use depends on the types of analysis we want to do at the architecture design time. We have focused on the C&C view as the primary view, and have discussed a few styles for this view as well.

If we want to analyze the performance of the architecture (to be exact, performance of the system that will have the proposed architecture,) then even though the C&C view represents the run-time structure, it is not enough. The reason is that though performance depends on the runtime structure of the software, it also depends on the hardware and other resources that will be used to execute the software. For example, the performance of a n-tier system will be very different if all the tiers reside on the same machine as compared to if they reside on different machines. In other words, the performance of a system whose architecture remains the same, can change depending on how the components of the architecture are allocated on the hardware. Hence, to do any meaningful performance analysis, we must specify the allocation as well. The same holds true if we want to do any reliability or availability analysis, as they also depend on the reliability of the hardware components involved in running the system.

To facilitate such an analysis, a deployment view needs to be provided. In a deployment view, the elements of a C&C style are allocated to execution resources like CPU and communication channels. Hence, the elements of this view are the software components and connectors from the C&C view, and the hardware elements like CPU, memory, and bandwidth. This view shows which software components are allocated to which hardware element. This allocation can be dynamic and this dynamism can also be represented.

Note that even the allocation view, which is necessary to do performance analysis, is not sufficient. To analyze the performance, besides the allocation, we will need to properly characterize the hardware elements in terms of their capacities, and the software elements in terms of their resource requirement and usage. Using the information, models can be built to determine bottlenecks, optimal allocation, etc. This is an active area of research and a survey of the area can be found in [4].

For doing any performance analysis, some models will have to be built, and these models will need information about the hardware and software components and how the software is allocated to software. The level of detail that can be obtained depends on the model used. At the basic level, some experience-based analysis can be done to see if there are any performance bottlenecks. The allocation of software can also be examined for optimality of performance, and if needed that allocation can be changed. For example, in a n-tier system, it may be found that the overhead of communication is too heavy between two tiers, and it may be better to allocate both of them to one machine. Or the analysis may reveal the reverse—allocating both the database and the business layer on the same machine might degrade the response time as concurrency will be lost, and it may be decided to add another machine to host the business layer and connect it to the machine hosting the database layer with a high speed connection.

Many such possibilities exist for performance analysis. Consequently, for C&C view of the architecture, it may be desirable to look at an allocation view at the time of creating the architecture. It may be added that not all C&C views render themselves easily or fruitfully for allocation view. The n-tier style (or client-server style), or the process view clearly render itself to an allocation view. However, it is not clear if the allocation view of a publish-subscribe view will be very useful. For giving an allocation view, it is best to chose a C&C view that renders itself naturally to the allocation view. If the views obtained so far do not render themselves to an allocation view, but an allocation view is essential for the desired analysis, then a view should be created that can be used for such an allocation and analysis.

4.5.4 Documenting Architecture Design

So far we have focused on representing views through diagrams. While designing, diagrams are indeed a good way to explore options and encourage discussion and brainstorming between the architects. But when the designing is over, the architecture has to be properly communicated to all stakeholders for negotiation and agreement. This requires that architecture be precisely documented with enough information to perform the types of analysis the different stakeholders wish to make to satisfy themselves that their concerns have been adequately addressed. Without a properly documented description of the architecture, it is not possible to have a clear common understanding. Hence, properly documenting an architecture is as important as creating one. In this section, we discuss what an architecture document should contain. Our discussion is based on the recommendations in [93, 35, 9].

Just like different projects require different views, different projects will need different level of detail in their architecture documentation. In general, however, a document describing the architecture should contain the following:

- System and architecture context
- Description of architecture views
- Across views documentation

We know that an architecture for a system is driven by the system objectives and the needs of the stakeholders. Hence, the first aspect that an architecture document should contain is identification of stakeholders and their concerns. This portion should give an overview of the system, the different stakeholders, and the system properties for which the architecture will be evaluated. A context diagram that establishes the scope of the system, its boundaries, the key actors in that interact with the system, and sources and sinks of data can also be very useful. A context diagram is frequently represented by showing the system in the center, and showing its connections with people and systems, including sources and sinks of data.

With the context defined, the document can proceed with describing the different structures or views. As stated before, multiple views of different types may be needed, and which views are chosen depends on the needs of the project and its stakeholders. The description of views in the architecture documentation will almost always contain a pictorial representation of the view, which is often the *primary presentation of the view.* As discussed earlier, in any view diagram it is desirable to have different symbols for different element types and provide a key for the different types, such that the type of the different components (represented using the symbols) is clear to a reader. It is, of course, highly desirable to keep the diagram simple and uncluttered. If necessary, to keep the complexity of the view manageable, a hierarchical approach can be followed to make the main view simple (and provide further details as structure of the elements).

However, a pictorial representation is not a complete description of the view. It gives an intuitive idea of the design, but is not sufficient for providing the details. For example, what is the purpose and functionality of a module or a component is indicated only by its name which is not sufficient. Hence, supporting documentation is needed for the view diagrams. This supporting documentation should have some or all of the following:

- *Element Catalog.* Provides more information about the elements shown in the primary representation. Besides describing the purpose of the element, it should also describe the elements' interfaces (remember that all elements have interfaces through which they interact with other elements). All the different interfaces provided by the elements should be specified. Interfaces should have unique identity, and the specification should give both syntactic and semantic information. Syntactic information is often in terms of signatures, which describe all the data items involved in the interface and their types. Semantic information must describe what the interface does. The description should also clearly state the error conditions that the interface can return.

- *Architecture Rationale.* Though a view specifies the elements and and the relationship between them, it does not provide any insight into why the architect chose the particular structure. Architecture rationale gives the reasons for selecting the different elements and composing them in the way it was done. This section may also provide some discussion on the alternatives that were considered and why they were rejected. This discussion, besides explaining the choices, is also useful later when an analyst making a change wonders why the architecture should not be changed in some manner (that might make the change easy).

- *Behavior.* A view gives the structural information. It does not represent the actual behavior or execution. Consequently, in a structure, all possible interactions during an execution are shown. Sometimes, it is necessary to get some idea of the actual behavior of the system in some scenarios. Such a description is useful for arguing about properties like deadlock. Behavior description can be provided to help aid understanding of the system execution. Often diagrams like collaboration diagrams or sequence diagrams (we will discuss these further in the Chapter on OO design) are used.

- *Other Information.* This may include a description of all those decisions that have not been taken during architecture creation but have been deliberately left for future. For example, the choice of a server or protocol. If this is done, then it must be specified as fixing these will have impact on the architecture.

We know that the different views are related. In what we have discussed so far, the views have been described independently. The architecture document therefore, besides describing the views, should also describe the relationship between the different views. This is the primary purpose of the across view documentation. Essentially, this documentation describes the relationship between elements of the different views (for example, how modules in a module view relate to components in a component view, or how components in a C&C view relate to processes in a process view). This part of the document can also describe the rationale of the overall architecture, why the selected views were chosen, and any other information that cuts across views.

However, often the relationship between the different views is straightforward or very strong. In such situations, the different structures may look very similar and describing the views separately can lead to a repetition. In such situations, for practical reasons, it is better to combine different views into one. Besides eliminating the duplication, this approach can also help clearly show the strong relationship between the two views (and in the process also reduce the across view documentation). Combined views are also useful for some analysis which require multiple views, for example, performance analysis, which frequently requires both the C&C view as well as the allocation view. So, sometimes, it may be desirable to show some combined views.

Combining of views, however, should be done only if the relationship between the views is strong and straightforward. Otherwise, putting multiple views in one diagram will clutter the view and make it confusing. The objective of showing multiple views in one is not merely to reduce the number of views, but is to be done primarily to aid understanding and showing the relationships. An example of combining is when there are multiple modules in the module view that form the different layers in the layer view. In such a situation, it is probably more natural to show one view consisting of the layers, and overlaying the module structure on the layers. That is, showing the module structure within the layers. Many layered systems architectures actually use this approach. In such a situation, it is best to show them together, creating a hybrid style in which both a module view and a C&C view are captured. Overall, if the mapping can be shown easily and in a simple manner, then different views should be combined for the sake of simplicity and compactness. If, however, the relationship between the different views is complex (for example, a many-to-many relationship between elements of the different views), then it is best to keep them separate and specify the relationship separately.

The general structure discussed here can provide a guide for organizing the architecture document. However, the main purpose of the document is to clearly communicate the architecture to the stakeholders such that the desired analysis can be done. And if some of these sections are redundant for that purpose, they may not be included. Similarly, if more information needs to be provided, then it should be done.

Finally, a word on the language chosen for describing different parts of the architecture. Here the choice varies from the formal architecture description languages (ADLs) to informal notation. Many people now use UML to represent the architecture, which allows various possibilities to show the primary description of the view and also allows annotation capability for supporting document. We believe that any method can be used, as long as the objective is met. To allow flexibility, we suggest using a problem specific notation, but following the guidelines for good view representation, and using a combination of header definitions and text for the supporting documentation.

4.6 Evaluating Architectures

Architecture of a software system impacts some of the key nonfunctional quality attributes like modifiability, performance, reliability, portability, etc. The architecture has a much more significant impact on some of these properties than the design and coding choices. That is, even though choice of algorithms, data structures, etc., are important for many of these attributes, often they have less of an impact than the architectural choices. Clearly then evaluating a proposed architecture for these properties can have a beneficial impact on the project—any architectural changes that are required to meet the desired goals for these attributes can be done during the architecture design itself.

There are many nonfunctional quality attributes. Not all of them are affected by architecture significantly. Some of the attributes on which architecture has a significant impact are performance, reliability and availability, security (some aspects of it), modifiability, reusability, and portability. Attributes like usability are only mildly affected by architecture.

How should a proposed architecture be evaluated for these attributes? For some attributes like performance and reliability, it is possible to build formal models using techniques like queuing networks and use them for assessing the value of the attribute. However, these models require information beyond the architecture description, generally in forms of execution times, and reliability of each component.

Another approach is procedural—a sequence of steps is followed to subjectively evaluate the impact of the architecture on some of the attributes. One such informal analysis approach that is often used is as follows. First identify the attributes of interest for which an architecture should be evaluated. These attributes are usually determined from stakeholder's interests—the attributes the different stakeholders are most interested in. These attributes are then listed in a table. Then for each attribute, an experience-based, subjective analysis is done (though quantitative analysis can also be done), to assess the level supported by the architecture. The analysis might mention the level for each attribute (e.g., good, average, poor), or might simply mention whether it is satisfactory or not. Based on the outcome of this analysis, the architecture is either accepted or rejected. If rejected, it may be enhanced to improve the performance for the attribute for which the proposed architecture was unsatisfactory.

Many techniques have been proposed for evaluation, and a survey of them is given in [51]. Here we briefly discuss some aspects of a more elaborate and formal technique called architectural tradeoff analysis method.

4.6.1 The ATAM Analysis Method

The architectural tradeoff analysis method (ATAM) [105, 35], besides analyzing the architecture for a set of properties, also helps in identifying dependencies between competing properties and perform a tradeoff analysis. We will, however, mostly focus on the analysis. The basic ATAM analysis has the following steps, which can be repeated, if needed:

1. *Collect Scenarios.* Scenarios describe an interaction of the system. For architecture analysis, scenarios list the situations the system could be in and for which we would like to evaluate the architecture for different attributes. Besides normal scenarios, exceptional scenarios of interest should also be mentioned.

2. *Collect Requirements or Constraints.* These specify what are the requirements for the system. That is, what is expected from the system in these scenarios. The scenarios together with requirements form the basis for evaluation—we want to ensure that the software will satisfy these requirements in these scenarios. These requirements essentially specify the desired levels (hopefully quantitatively) for the quality attributes of interest. So, for example, instead of saying performance is of interest in this system, a constraint or a requirement will be like "the average response time should be less than 1 ms."

3. *Describe Architectural Views.* The views of different proposed architectures are collected here. These are the architectures that will be evaluated. What views are needed to describe a proposed architecture depends on what analysis needs to be performed, which is driven by the requirements or constraints. We will limit our attention to component and connector view only.

4. *Attribute-Specific Analysis.* Now we have the proposed architectures, the different quality attributes of interest for the system, and the different scenarios under which these attributes should be evaluated. In this step, each quality attribute is analyzed separately and individually. The analysis should result in what levels an architecture can support for the quality attribute. So, for example, an analysis can result in a statement like "the availability of this system is 0.95." Once the analysis for all the attributes is done, it can be seen to what degree the requirements identified earlier are met. The outcome of this analysis can become the basis for selecting one architecture over other. It can also form the basis of changing a proposed architecture in an attempt to meet the desired levels. If an architecture is changed, then the whole analysis needs to be repeated, making ATAM a spiral process.

5. *Indentify Sensitivities and Tradeoffs.* From the analysis, for each attribute, sensitivity of the different elements in the architecture view should be determined. That is, how much impact does an element have on the attribute value. From this we identify the ***sensitivity points***, which are the elements that have the most significant impact on the attribute value. Sensitivity points are these elements whose change will have the maximum impact on the quality attribute. ***Tradeoff points*** are those elements that are sensitivity points for multiple attributes. Changing these elements will have a significant impact on multiple attributes. These elements are where tradeoffs decisions will have to be taken, particularly if changing it favorably affects one attribute but negatively affects the other. For example, in a n-tier architecture, a server will be the tradeoff point as it significantly affects performance as well as availability. Replicating the server and performing updates on each server (to keep each current) can improve the availability, but can have a detrimental effect on update performance.

4.6.2 An Example

Earlier in the chapter, we gave an example of the student-survey system. We will take that example for evaluation and consider the second and the third architectures proposed (we do not consider the first one as it does not have the same functionality as others). For analysis, we add another architecture, in which the cache component is between the server and the database component. That is, in this architecture, each request is directly sent to the cache, which then decides whether to respond using data from the cache or from the database. This architecture is shown in Figure 4.11. We assume that the cache component in this architectures updates the database as well as its own data after every 5th survey. (For analysis we add a requirement on data currency to allow this. The requirement is given below.) We focus only on survey-taking and ignore the feature of just getting the survey result.

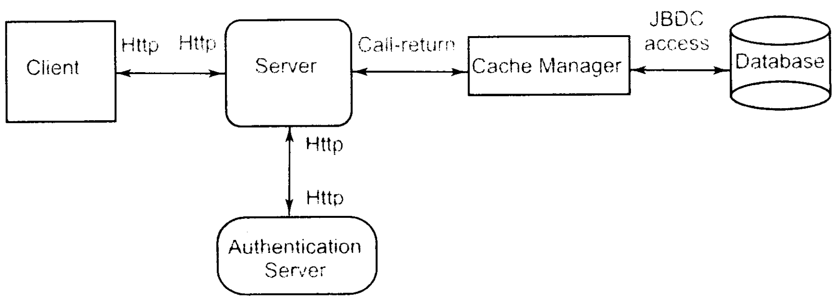

Figure 4.11: Another architecture for the student survey system.

Let us now analyze these architectures using ATAM. First we list the scenarios of interest. These are:

- *S1.* A student submits the survey form and gets current results of the survey (normal scenario: all servers are up, load normal.)
- *S2.* A student tries to take the survey many times.
- *S3.* The database server is temporarily down.
- *S4.* The network/system is highly loaded.

Some of the requirements or constraints for the system are:

- *Security.* A student should be allowed to take the survey at most once.
- *Response Time.* A student should get a response time of less than 2 sec on an average, 80% of the time.
- *Availability.* The system should at least have availability of 0.85 (that is, when a student comes, there is a 85% chance that he can successfully take the survey).
- *Data Currency.* The survey result given to a student should be reasonably current and should not be older than 5 submissions before.

Now let us evaluate the three architecture proposals for these attributes. For analysis, we will look at each of the attributes and then study the three architectures under the scenarios relevant for that attribute. For security and data currency, we will analyze based on our understanding of the architecture. For availability and response time, formal models are possible. We will use simple probability-based approach here.

For the availability analysis, we assume that the cache is on the same machine as the server, the database is on a different machine, and that availability of each of the machines is 0.9. We also assume that when the database goes down, during its repair time, on an average, 10 student survey requests come. For response time, we assume the following response times:

Component	Normal conditions	Heavily loaded
Server + security	300ms	600ms
Database	800ms	1600ms
Cache	50ms	50ms

We also assume that a timeout of about 2 seconds is used when the server tries to access the database, and that the network is heavily loaded 1% of the time.

We do not show all the computations here, but illustrate a few. The availability for the first architecture is the probability that both the server and database are up, as that is the only case in which a request can be serviced. This is 0.9 * 0.9 = 0.81. For the second and the third case, it is slightly more complex. When the database is down (on an average for 10 requests), up to 5 requests can still be serviced. Hence, even when the database is down, the system is up for half the time. This gives us an additional availability of 0.5 * 0.9 * 0.1 = 0.045 (half the probability that server is up and database is down). So, the availability of these architectures is 0.81 + 0.045 = 0.855.

Determining the average response time is slightly more involved, as we have to consider both the heavy load and normal load situations. In normal load, for first architecture, the average response time will be 300 + 800 = 1100 ms. When the database is down, then this architecture cannot service a request. For the second architecture, the response time is 300 + 800 + 50 = 1150 ms in the normal scenario. When the database is down, some requests can be serviced (probability computed above) by the cache but cache is used only after the database times out. That is, the response time in this scenario is 300 + 2000 + 50 = 2350 ms. For the third architecture, in the normal scenario, the average response time is 350 * 0.8 (for those requests serviced by cache) plus 1350 * 0.2 (for those that go to the database). That is, the average response time is 550 ms. When the database is down, the response time for the requests that can be serviced remains the same as they are serviced from the cache in a normal manner. Similar analysis can be done when the network is congested. For simplicity, we will double these times (as the response time when the system is congested is double that when it is not).

With these, we build a table for the different attributes for the scenarios of interest. This is given in Table 4.1.

Property (Scenario)	Architecture 1	Architecture 2	Architecture 3
Security (S1)	Yes	Yes	Yes
Security (S2)	Yes	Yes	Yes
Response time (S1)	1100	1150	550
Response time (S3)	N/A	2350	550
Response time (S4)	2200	2300	1100
Availability (S3)	0.81	0.855	0.855
Data Currency (S1)	Yes	Yes	Yes
Data Currency (S2)	N/A	Yes	Yes

Table 4.1: Analysis of the architecture options.

From this table we can clearly see that security and data currency requirements are satisfied by all three architecture options. As the probability of normal scenario is over 0.8 for all architectures, and response time in normal scenario is less than the required for all, the response time requirement is also met by all the three architectures. However, the availability requirement is met by only the second and the third architectures. Of these two architectures, the third should be preferred as it provides a better response time.

4.7 Summary

Architecture of a software system is a design of the system that provides a very high level view of the system in terms of parts of the system and how they are related to form the whole system. Depending on how the system is partitioned, we get a different architectural view of the system. Consequently, the architecture of a software system is defined as the structures of the system which comprise software elements, their externally visible properties, and relationships among them. The macro level view that the architecture facilitates development of a high quality system. It also allows analysis of many of the system properties like performance that depend mostly on architecture to be done early in the software life cycle.

There are three main architectural views of a system—module, component and connector, and allocation. In a module view, the system is viewed as a structure of programming modules like packages, classes, functions, etc., which have to be later constructed. In a component and connector (C&C) view, the system is a collection of runtime entities called components. During execution, these components interact with each other through the connectors. An allocation view describes how the different software units are allocated to hardware resources in the system. These different views are related as they are of the same system, but this relationship may or may not be straightforward. If different views are created, then their relationship must be clearly identified and stated. In this chapter, we focus mostly on C&C view.

There are some common styles for a C&C view which have been found useful for creating this architecture view for a system. We have discussed pipe and filter, shared data, client server, publish-subscribe, peer to peer, and communicating processes styles. Each of these styles describe the types of components and connectors that exist and the constraints on how they are used. For example, the pipe and filter has one type of component (filter) and one type of connector (pipe) and components can be connected through the pipe. The client-server style has two types of components (client and server) and there is one connector (request/reply). A client can only communicate with the server, and an interaction is initiated by a client. In shared data style the two component types are repository and data accessors. Data accessors read/write the repository and share information among themselves through the repository.

Designing an architecture is the first step towards building a solution for the problem described in the SRS. The architecture forms the foundation for the system and rest of the design and development activities. Consequently, it should be properly documented. A proper architecture document should describe the context in which the architecture was designed, the different architectural views that were created, and how the different views relate to each other. In the context, it is important to identify the different stakeholders and their objectives, as that drives the architecture choices. Generally, one view is taken as the primary view and the architecture description revolves around that. Different views can be combined with the primary view, if this combination does not complicate the architecture diagram or description. The architecture description of a view, which often revolves around an architecture diagram, should clearly specify the different types of elements and their external behavior. The architecture rationale, or why the architecture choices were made, should also be documented.

As considerable analysis and thought can go into designing the architecture to ensure that the final selected architecture can support the desired system properties, it is essential that the architecture be preserved during the rest of the development. That is, the architecture, once selected and specified, constrains the design and development. It is essential that these constraints be respected and the design be consistent with the architecture. We have illustrated how these constraints can be violated while building a system that still provides the desired functionality. It is essential that such violations do not occur.

A considerable analysis of the system attributes like performance, security, reliability, and modifiability is possible when the architecture views are ready. There are many approaches for performing this analysis. We have briefly discussed the ATAM approach in which first the key scenarios are enumerated, along with the constraints for the attributes of interest for the system. Then the proposed architectures are evaluated to see how well they satisfy the constraints under the different scenarios. The result of this analysis can be used to compare different architecture choices. The analysis can also be extended to perform sensitivity and tradeoff analysis.

Exercises

1. Explain why architecture is not just one structure consisting of different parts and their relationship.
2. What do you think is the relationship between the component and connector view and the module view. In the situations where this relationship is simple, how will you express it in one diagram?
3. In the student-survey example, extend the architecture diagram to also show the module view. (If you want to look at the code for the architecture, it is available from the Web site.)

4. In the analysis done in Section 4 of the student-survey example, some allocation of software elements to hardware was chosen to perform the analysis. Draw an allocation view for this allocation.

5. **For the example analyzed in Section 4 of the student-survey system, let us make the following changes: (1) Suppose the student can also ask for just the survey result, and the response time requirement for this is different. (2) Assume that the availability of server is much higher (say 0.99) than the database server (0.9). Now do the analysis.**

6. A closed-loop control system generally works as follows. A centralized controller gets values from the various sensors, does the computation to determine the various settings (often by solving some partial differential equations), and then issues necessary commands to the actuators in the system so that the system remains balanced and gives optimal performance. What would you chose as the primary view to describe the architecture of this control system—give reasons for your selection. Design and draw (the primary view) at least two architectures for this system, clearly defining the different elements in it.

7. **For the above example, create some scenarios (including some for failures) and some requirements, and then analyze your architectures to see which one is the best.**

8. Consider an interactive Web site which provides many different features to perform various tasks. Show that the architecture for this can be represented as a shared-data style as well as client-server style. Which one will you prefer and why?

9. Consider the instant messaging system that you use. Which of the views is best suited to describe its C&C view. Give a diagram to describe the architecture (C&C view) of such a system.

Case Studies

Here we briefly discuss the architecture for the two case studies. The complete architecture document is available from the Web site.

Case Study 1—Course Scheduling

This is a batch processing type system, where data is taken by the system in the start of the processing from the two input files, and the output is produced at the end. No data updates are done (that is, there is no need for a repository). For such a system, the pipe-and-filter style is eminently suited. Hence, we use this style for the architecture of this system. The proposed architecture is shown in Figure 4.12.

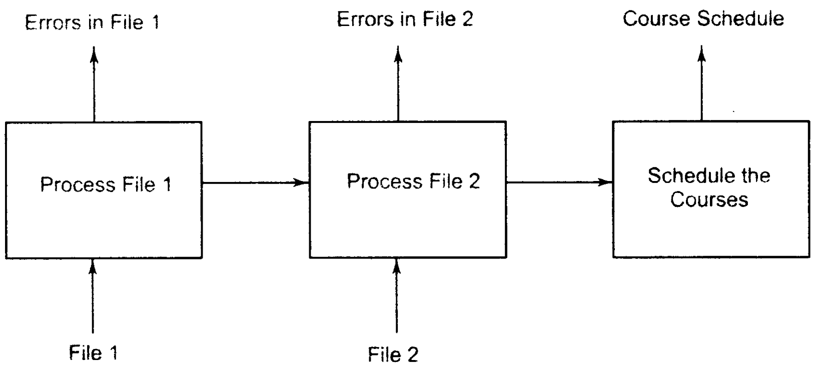

Figure 4.12: Architecture for course scheduling case study.

A bit more discussion and evaluation of the architecture is given in the architecture document for this case study, which is available from the Web site. It might be added that this case study was originally built without any architecture design in the process (architectures were not well established when this case study was done). However, this architecture, even though done retrospectively, is representative of the system and the actual system architecture closely resembles this architecture, as will be clear when we discuss the design of this case study in later chapters.

Case Study 2—PIMS

The main stakeholders for the PIMS system are the individual users who might use the system and the system designer/builder who will build PIMS. The main concerns of the two stakeholders are:

- **For Users:** The usability of the system and providing rate or returns and the net-worth info. Reasonable response time is also a concern.

- **For designer/builder:** The system is easy to modify, particularly to handle future extensions mentioned in the SRS (that is, the system may become a multi-user system, which may require use of databases instead of files for keeping data). It should be easily portable.

Hence, the key property which the architecture should aim to satisfy (besides the functionality) is modifiability or extensibility of the system. Portability and response time are other factors which should drive the architecture decisions.

Due to the data-oriented nature of the system, it should be clear that a shared-data style would be the best. As the system is rather small, the first architecture proposal was to have two main components—a repository to keep the data on investments, and a processing component to do all the processing. And have another component to get the latest value of the shares from the Web into the repository.

We then realized that there are two independent aspects of processing, one dealing with data entry and edit, and the other which does the computation of rates of return and net worth. These two do not need to communicate with each other. Separating the two will make the system more modular and making modifications will be easier. For example, user interface changes will affect only the edit component, while any change in reporting or computations will affect the other only. The simplified view of these two architectures is shown in Figure 4.13.

Even though the second architecture is better than the first one as it has better modularity, the processing components are still very tightly coupled with the data repository. If the repository changes from one database to another, then besides making the changes to the repository, the code of both the computation components will have to be changed.

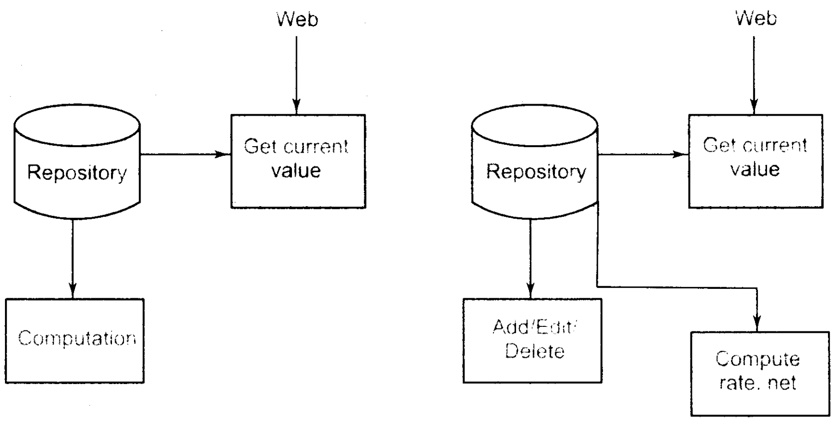

Figure 4.13: Initial architectures for PIMS.

To separate the data access from the computation, a layer was added between the computation components and the repository. (By adding a layer, we are mixing module view and C&C view, but as it is clear and simple, we continue with it.) The complete description of this architecture, which includes the user interface component also, is given in the architecture document for this case study. The document also gives the complete picture of the previous architecture also, for which only a simplified view is given here.

The architecture document also gives a simple analysis of the different architecture, leading to the selection of one architecture as the proposed architecture for PIMS.

Chapter 5

Planning a Software Project

For a successful project, both good project management and good engineering are essential. Lack of either one can cause a project to fail. We have seen that project management activities can be viewed as having three major phases: project planning, project monitoring and control, and project termination. Broadly speaking, planning entails all activities that must be performed before starting the development work. Once the project is started, project control begins. In other words, during planning all the activities that management needs to perform are planned, while during project control the plan is executed and updated.

Planning may be the most important management activity. Without a proper plan, no real monitoring or controlling of the project is possible. Often projects are rushed toward implementation with not enough time and effort spent on planning. No amount of technical effort later can compensate for lack of careful planning. Lack of proper planing is a sure ticket to failure for a large software project. For this reason, we treat project planning as an independent chapter.

The basic goal of planning is to look into the future, identify the activities that need to be done to complete the project successfully, and plan the scheduling and resources. The inputs to the planning activity are the requirements specification and the architecture description. A very detailed requirements document is not essential for planning, but for a good plan all the important requirements must be known, and it is highly desirable that architecture decisions have been taken. The major issues project planning addresses are:

Process planning
Effort estimation
Schedule and Resource Estimation
Quality plans
Configuration management plans
Risk management
Project monitoring plans

In the rest of this chapter we will discuss each of these issues and some techniques for handling them.

5.1 Process Planning

We have already discussed in detail the development process and the various process models. For a project, during planning, a key activity is to plan and specify the process that should be used in the project. This means specifying the various stages in the process, the entry criteria for each stage, the exit criteria, and the verification activities that will be done at the end of the stage. As discussed, some established process model may be used as a standard process and tailored to suit the needs of the project. In an organization, often standard processes are defined and a project can use any of these standard processes and tailor it to suit the specific needs of the project. Hence the process planning activity mostly entails selecting a standard process and tailoring it for the project.

Tailoring is an advanced topic which we will not discuss in any detail. Generally, however, based on the size, complexity and nature of the project, as well as the characteristics of the team, like the experience of the team members with the problem domain as well as the technology being used, the standard process is tailored. The common tailoring actions are modify a step, omit a step, add a step, or change the formality with which a step is done. After tailoring, the process specification for the project is available. This process guides rest of the planning, particularly detailed scheduling where detailed tasks to be done in the project are defined and assigned to people to execute them.

5.2 Effort Estimation

For a given set of requirements it is desirable to know how much it will cost to develop the software, and how much time the development will take. These estimates are needed *before* development is initiated. The primary reason for cost and schedule estimation is cost-benefit analysis, and project monitoring and control. A more practical use of these estimates is in bidding for software projects, where cost estimates must be given to a potential client for the development contract.

The bulk of the cost of software development is due to the human resources needed, and therefore most cost estimation procedures focus on estimating effort in terms of person-months (PM). By properly including the "overheads" (i.e., the cost of hardware, software, office space, etc.) in the cost of a person-month, effort estimates can be converted into cost.

For a software development project, effort and schedule estimates are essential prerequisites for managing the project. Otherwise, even simple questions like "is the project late?" "are there cost overruns?" and "when is the project likely to complete?" cannot be answered. Effort and schedule estimates are also required to determine the staffing level for a project during different phases.

Estimates can be based on subjective opinion of some person or determined through the use of models. Though there are approaches to structure the opinions of persons for achieving a consensus on the effort estimate (e.g., the Delphi approach [20]), it is generally accepted that it is important to have a more scientific approach to estimation through the use of models. In this section we discuss only the model-based approach for effort estimation. Before we discuss the models, let us first understand the limitations of any effort estimation procedure.

5.2.1 Uncertainties in Effort Estimation

One can perform effort estimation at any point in the software life cycle. As the effort of the project depends on the nature and characteristics of the project, at any point, the accuracy of the estimate will depend on the amount of reliable information we have about the final product. Clearly, when the product is delivered, the effort can be accurately determined, as all the data about the project and the resources spent can be fully known by then. This is effort estimation with complete knowledge about the project. On the other extreme is the point when the project is being initiated or during the feasibility study. At this time, we have only some idea of the classes of data the system will get and produce and the major functionality of the system. There is a great deal of uncertainty about the actual specifications of the system. Specifications with uncertainty represent a range of possible final products, not one precisely defined product. Hence, the effort estimation based on this type of information cannot be accurate. Estimates at this phase of the project can be off by as much as a factor of four from the actual final effort.

As we specify the system more fully and accurately, the uncertainties are reduced and more accurate effort estimates can be made. For example, once the requirements are completely specified, more accurate effort estimates can be made compared to the estimates after the feasibility study. Once the design is complete, the estimates can be made still more accurately. The obtainable accuracy of the estimates as it varies with the different phases is shown in Figure 5.1 [20, 21].

Note that this figure is simply specifying the limitations of effort estimating strategies—the best accuracy a effort estimating strategy can hope to achieve. It does not say anything about the existence of strategies that can provide the estimate with that accuracy. For actual effort estimation, estimation models or procedures have to be developed. The accuracy of the estimates will depend on the effectiveness and accuracy of the estimation procedures or models employed and the process (i.e., how predictable it is).

Despite the limitations, estimation models have matured considerably and generally give fairly accurate estimates. For example, when the COCOMO model (discussed later)

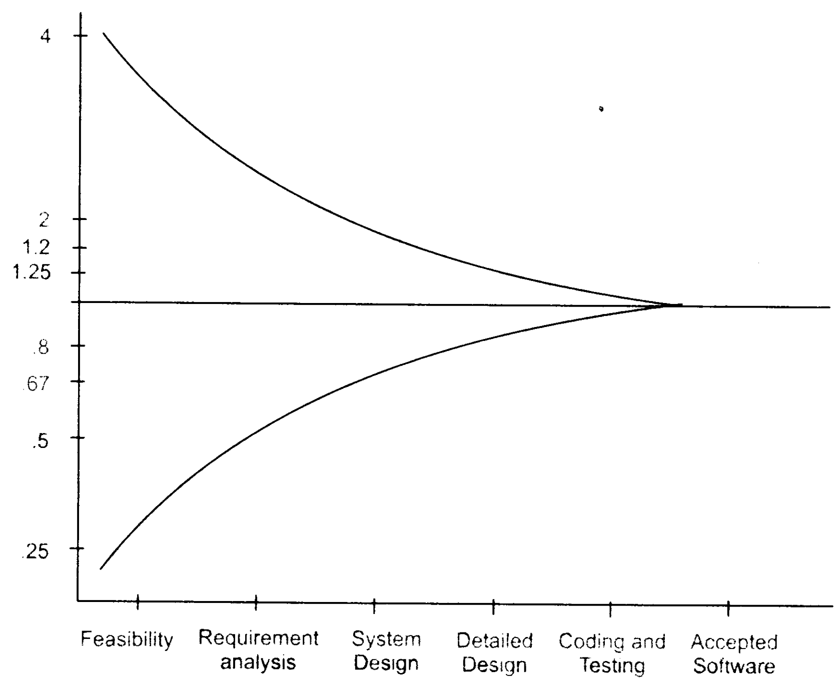

Figure 5.1: Accuracy of effort estimation.

was checked with data from some projects, it was found that the estimates were within 20% of the actual effort 68% of the time.

It should also be mentioned that achieving an estimate after the requirements have been specified within 20% is actually quite good. With such an estimate, there need not even be any cost and schedule overruns, as there is generally enough slack or free time available (recall the study mentioned earlier that found a programmer spends more than 30% of his time in personal or miscellaneous tasks) that can be used to meet the targets set for the project based on the estimates. In other words, if the estimate is within 20% of the actual, the effect of this inaccuracy will not even be reflected in the final cost and schedule. Highly precise estimates are generally not needed. Reasonable estimates in a software project tend to become a self-fulfilling prophecy—people work to meet the schedules (which are derived from effort estimates).

5.2.2 Building Effort Estimation Models

An estimation model can be viewed as a "function" that outputs the effort estimate. clearly this estimation function will need inputs about the project, from which it can produce the estimate. The basic idea of having a model or procedure for estimation is that it reduces the problem of estimation to estimating or determining the value of the "key parameters" that characterize the project, based on which the effort can be estimated.

Note that an estimation model does not, and cannot, work in a vacuum; it needs inputs to produce the effort estimate as output. At the start of a project, when the details of the software itself are not known, the hope is that the estimation model will require values of parameters that can be measured at that stage.

Although the effort for a project is a function of many parameters, it is generally agreed that the primary factor that controls the effort is the size of the project, that is, the larger the project, the greater the effort requirement. One common approach therefore for estimating effort is to make it a function of *project size*, and the equation of effort is considered as

$$EFFORT = a * SIZE^b,$$

where a and b are constants [5], and project size is generally in KLOC or function points. Values for these constants for a particular process are determined through regression analysis, which is applied to data about the projects that has been performed in the past. For example, Watson and Felix [142] analyzed the data of more than 60 projects done at IBM Federal Systems Division, ranging from 4,000 to 467,000 lines of delivered source code, and found that if the SIZE estimate is in thousands of delivered lines of code (KLOC), the total effort, E, in person-months (PM) can be given by the equation $E = 5.2(SIZE)^{.91}$.

Often, however, simple productivity may be used to determine the overall estimate from the size. That is, if productivity is P KLOC/PM, then effort estimate for the project may be SIZE/P person-months. This approach will work if the size and type of the project are similar to the set of projects from which the productivity P was obtained.

This approach of determining total effort from the total size is what we refer to as the *top-down approach*, as overall effort is first determined and then from this the effort for different parts are obtained.

In a *top-down estimation* model by using size as the main input to the model, we have replaced the problem of effort estimation by size estimation. One may then ask, why not directly do effort estimation rather than size estimation? The answer is that size estimation is often easier than direct effort estimation. For estimating size, the system is generally partitioned into components it is likely to have. Once the components of the system are known, as estimating something about a small unit is generally much easier than estimating it for a larger system, sizes of components can be generally estimated quite accurately. Once size estimates for components are available, to get the overall size estimate for the system, the estimates of all the components can be added up. Similar property does not hold for effort estimation, as effort for developing a system is *not* the sum of effort for developing the components (as additional effort is needed for integration and other such activities when building a system from developed components). This key feature, that the system property is the sum of the properties of its parts, holds for size but not for effort, and is the main reason that size estimation is considered easier than effort estimation.

With top-down models, if the size estimate is inaccurate, the effort estimate produced by the models will also be inaccurate. Hence, it is important that good estimates for the size of the software be obtained. There is no known "simple" method for estimating the size accurately. When estimating software size, the best way may be to get as much detail as possible about the software to be developed and to be aware of our biases when estimating the size of the various components. By obtaining details and using them for size estimation, the estimates are likely to be closer to the actual size of the final software. In general, there is often a tendency by people to underestimate the size of software [20].

A somewhat different approach for effort estimation is the *bottom-up approach.* In this approach, the project is first divided into tasks and then estimates for the different tasks of the project are first obtained. From the estimates of the different tasks, the overall estimate is determined. That is, the overall estimate of the project is derived from the estimates of its parts. This type of approach is also called activity-based estimation. Essentially, in this approach the size and complexity of the project is captured in the set of tasks the project has to perform.

The bottom-up approach lends itself to direct estimation of effort; once the project is partitioned into smaller tasks, it is possible to directly estimate the effort required for them, specially if tasks are relatively small. A risk of bottom-up methods is that one may omit some important activities in the list of tasks. Also, directly estimating the effort for some overhead tasks, such as project management, that span the project can be difficult.

Both the top-down and the bottom-up approaches require information about the project: size (for top-down approaches) or a list of tasks (for bottom-up approaches). In many ways, these approaches are complementary, and often it may be desirable to determine the effort using both the approaches and then using these estimates to obtain the final estimate.

5.2.3 A Bottom-Up Estimation Approach

If architecture of the system to be built has been developed and if past information about how effort is distributed over different phases is known, then the bottom-up approach need not completely list all the tasks, and a less tedious approach is possible. Here we describe one such approach used in a commercial organization [97].

In this approach, the major programs (or units or modules) in the software being built are first determined. Each program unit is then classified as simple, medium, or complex based on certain criteria. For each classification unit, an average effort for coding (and unit testing) is decided. This standard coding effort can be based on past data from a similar project, from some guidelines, or some combination of these.

Once the number of units in the three categories of complexity is known and the estimated coding effort for each program is selected, the total coding effort for the project is known. From the coding effort, the effort required for the other phases and activities is determined as a percentage of coding effort. From information about the past performance of the process, the likely distribution of effort in different phases of this project is decided, and then used to determine the effort for other phases and activities. From these estimates, the total effort for the project is obtained.

This approach lends itself to a judicious mixture of experience and data. If suitable past data are not available (for example, if launching a new type of project), one can estimate the coding effort using experience once the nature of the different types of units is specified. With this estimate, we can obtain the estimate for other activities by working with some reasonable or standard effort distribution. This strategy can easily account for activities that are sometimes difficult to enumerate early but do consume effort by budgeting effort for "other" or "miscellaneous" category.

The procedure for estimation can be summarized as the following sequence of steps:

1. Identify modules in the system and classify them as simple, medium, or complex.

2. Determine the average coding effort for simple/medium/complex modules.

3. Get the total coding effort using the coding effort of different types of modules and the counts for them.

4. Using the effort distribution for similar projects, estimate the effort for other tasks and the total effort.

5. Refine the estimates based on project-specific factors.

This procedure uses a judicious mixture of past data (in the form of distribution of effort) and experience of the programmers. This approach is also simple and similar to how many of us plan any project. For this reason, for small projects, many people find this approach natural and comfortable.

Note that this method of classifying programs into a few categories and using an average coding effort for each category is used only for effort estimation. In detailed scheduling, when a project manager assigns each unit to a member of the team for coding and budgets time for the activity, characteristics of the unit are taken into account to give more or less time than the average.

5.2.4 COCOMO Model

A top-down model can depend on many different factors, instead of depending only on one variable, giving rise to multivariable models. One approach for building multivariable models is to start with an initial estimate determined by using the static single-variable model equations, which depend on size, and then adjusting the estimates based on other variables. This approach implies that size is the primary factor for cost; other factors have a lesser effect. Here we will discuss one such model called the COnstructive COst MOdel (COCOMO) developed by Boehm [20, 21]. This model also estimates the total effort in terms of person-months. The basic steps in this model are:

1. Obtain an initial estimate of the development effort from the estimate of thousands of delivered lines of source code (KLOC).

2. Determine a set of 15 multiplying factors from different attributes of the project.

3. Adjust the effort estimate by multiplying the initial estimate with all the multiplying factors.

The initial estimate (also called *nominal estimate*) is determined by an equation of the form used in the static single-variable models, using KLOC as the measure of size. To determine the initial effort E_i in person-months the equation used is of the type $E_i = a * (KLOC)^b$. The value of the constants a and b depend on the project type. In COCOMO, projects are categorized into three types—organic, semidetached, and embedded. These categories roughly characterize the complexity of the project with organic projects being those that are relatively straightforward and developed by a small team, and embedded are those that are ambitious and novel, with stringent constraints from the environment and high requirements for such aspects as interfacing and reliability. The constants a and b for different systems are:

System	a	b
Organic	3.2	1.05
Semidetached	3.0	1.12
Embedded	2.8	1.20

The value of the constants for a cost model depend on the process and have to be determined from past data. COCOMO has instead provided "global" constant values. These values should be considered as values to start with until data for some projects is available. With project data, the value of the constants can be determined through regression analysis.

There are 15 different attributes, called *cost driver attributes*, that determine the multiplying factors. These factors depend on product, computer, personnel, and technology attributes (called *project attributes*). Examples of the attributes are required software reliability (RELY), product complexity (CPLX), analyst capability (ACAP), application experience (AEXP), use of modern tools (TOOL), and required development schedule (SCHD). Each cost driver has a rating scale, and for each rating, a multiplying factor is provided. For example, for the product attribute RELY, the rating scale is very low, low, nominal, high, and very high (and in some cases extra high). The multiplying factors for these ratings are .75, .88, 1.00, 1.15, and 1.40, respectively. So, if the reliability requirement for the project is judged to be low then the multiplying factor is .75, while if it is judged to be very high the factor is 1.40. The attributes and their multiplying factors for different ratings are shown in Table 5.1 [20, 21]. The COCOMO approach also provides guidelines for assessing the rating for the different attributes [20].

	Rating				
Cost Drivers	Very Low	Low	Nom-inal	High	Very High
Product Attributes					
RELY, required reliability	.75	.88	1.00	1.15	1.40
DATA, database size		.94	1.00	1.08	1.16
CPLX, product complexity	.70	.85	1.00	1.15	1.30
Computer Attributes					
TIME, execution time constraint			1.00	1.11	1.30
STOR, main storage constraint			1.00	1.06	1.21
VITR, virtual machine volatility		.87	1.00	1.15	1.30
TURN, computer turnaround time		.87	1.00	1.07	1.15
Personnel Attributes					
ACAP, analyst capability	1.46	1.19	1.00	.86	.71
AEXP, application exp.	1.29	1.13	1.00	.91	.82
PCAP, programmer capability	1.42	1.17	1.00	.86	.70
VEXP, virtual machine exp.	1.21	1.10	1.00	.90	
LEXP, prog. language exp.	1.14	1.07	1.00	.95	
Project Attributes					
MODP, modern prog. practices	1.24	1.10	1.00	.91	.82
TOOL, use of SW tools	1.24	1.10	1.00	.91	.83
SCHED, development schedule	1.23	1.08	1.00	1.04	1.10

Table 5.1: Effort multipliers for different cost drivers.

The multiplying factors for all 15 cost drivers are multiplied to get the effort adjustment factor (EAF). The final effort estimate, E, is obtained by multiplying the initial estimate by the EAF. That is, $E = EAF * E_i$..

By this method, the overall cost of the project can be estimated. For planning and monitoring purposes, estimates of the effort required for the different phases is also desirable. In COCOMO, effort for a phase is a defined percentage of the overall effort.

The percentage of total effort spent in a phase varies with the type and size of the project. The percentages for an organic software project are given in Table 5.2.

Phase	Size			
	Small 2 KLOC	Intermediate 8 KLOC	Medium 32 KLOC	Large 128 KLOC
Product design	16	16	16	16
Detailed design	26	25	24	23
Code and unit test	42	40	38	36
Integration and test	16	19	22	25

Table 5.2: Phase-wise distribution of effort.

Using this table, the estimate of the effort required for each phase can be determined from the total effort estimate. For example, if the total effort estimate for an organic software system is 20 PM and the size estimate is 20KLOC, then the percentage effort for the coding and unit testing phase will be 40 + (38 - 40)/(32 - 8) * 20 = 39%. The estimate for the effort needed for this phase is 7.8 PM. This table does not list the cost of requirements as a percentage of the total cost estimate because the project plan (and cost estimation) is being done after the requirements are complete. In COCOMO the detailed design and code and unit testing are sometimes combined into one phase called the *programming phase.*

As an example, suppose a system for office automation has to be designed. From the requirements, it is clear that there will be four major modules in the system: data entry, data update, query, and report generator. It is also clear from the requirements that this project will fall in the organic category. The sizes for the different modules and the overall system are estimated to be:

Data Entry	0.6 KLOC
Data Update	0.6 KLOC
Query	0.8 KLOC
Reports	1.0 KLOC
TOTAL	3.0 KLOC

From the requirements, the ratings of the different cost driver attributes are assessed. These ratings, along with their multiplying factors, are:

Complexity	High	1.15
Storage	High	1.06
Experience	Low	1.13
Programmer Capability	Low	1.17

All other factors had a nominal rating. From these, the effort adjustment factor (EAF) is

$$EAF = 1.15*1.06*1.13*1.17 = 1.61.$$

The initial effort estimate for the project is obtained from the relevant equations. We have

$$E_i = 3.2 * 3^{1.05} = 10.14PM.$$

Using the EAF, the adjusted effort estimate is

$$E = 1.61 * 10.14 = 16.3PM.$$

Using the preceding table, we obtain the percentage of the total effort consumed in different phases. The office automation system's size estimate is 3 KLOC, so we will have to use interpolation to get the appropriate percentage (the two end values for interpolation will be the percentages for 2 KLOC and 8 KLOC). The percentages for the different phases are: design—16%, detailed design—25.83%, code and unit test—41.66%, and integration and testing—16.5%. With these, the effort estimates for the different phases are:

System Design	.16 * 16.3 = 2.6 PM
Detailed Design	.258 * 16.3 = 4.2 PM
Code and Unit Test	.4166 * 16.3 = 6.8 PM
Integration	.165 * 16.3 = 2.7 PM.

5.3 Project Scheduling and Staffing

Once the effort is estimated, various schedules (or project duration) are possible, depending on the number of resources (people) put on the project. For example, for a project whose effort estimate is 56 person-months, a total schedule of 8 months is possible with 7 people. A schedule of 7 months with 8 people is also possible, as is a schedule of approximately 9 months with 6 people.

As is well known, however, manpower and months are not fully interchangeable in a software project. A schedule cannot be simply obtained from the overall effort estimate by deciding on average staff size and then determining the total time requirement by dividing the total effort by the average staff size. Brooks has pointed out that person and months (time) are not interchangeable. According to Brooks [25], "... man and months are interchangeable only for activities that require no communication among men, like sowing wheat or reaping cotton. This is not even approximately true of software...."

For instance, in the example here, a schedule of 1 month with 56 people is not possible even though the effort matches the requirement. Similarly, no one would execute the project in 28 months with 2 people. In other words, once the effort is fixed, there is some flexibility in setting the schedule by appropriately staffing the project, but this flexibility is not unlimited. Empirical data also suggests that no simple equation between effort and schedule fits well [127].

In a project, the scheduling activity can be broken into two subactivities: determining the overall schedule (the project duration) with major milestones, and developing the detailed schedule of the various tasks.

5.3.1 Overall Scheduling

One method to determine the normal (or nominal) overall schedule is to determine it as a function of effort. Any such function has to be determined from data from completed projects using statistical techniques like fitting a regression curve through the scatter plot obtained by plotting the effort and schedule of past projects. This curve is generally nonlinear because the schedule does not grow linearly with effort. Many models follow this approach [5, 20]. The IBM Federal Systems Division found that the total duration, M, in calendar months can be estimated by

$$M = 4.1E^{.36}.$$

In COCOMO, the equation for schedule for an organic type of software is

$$M = 2.5E^{.38}.$$

It should be clear that schedule is not a function solely of effort. Hence, the schedule determined in this manner is not really fixed. However, it can be used as a guideline or check of the schedules reasonableness, which might be decided based on other factors.

One rule of thumb, called the *square root check*, is sometimes used to check the schedule of medium-sized projects [97]. This check suggests that the proposed schedule can be around the square root of the total effort in person-months. This schedule can be met if suitable resources are assigned to the project. For example, if the effort estimate is 50 person-months, a schedule of about 7 to 8 months will be suitable.

From this macro estimate of schedule, we have to determine the schedule for the major milestones in the project. To determine the milestones, we must first understand the manpower ramp-up that usually takes place in a project. The number of people in a software project tends to follow the Rayleigh curve [126, 127]. That is, in the beginning and the end, few people work on the project; the peak team size (PTS) is reached somewhere near the middle of the project. This behavior occurs because only a few people are needed in the initial phases of requirements analysis and design. The human resources requirement peaks during coding and unit testing. Again, during system testing and integration, fewer people are required.

Often, the staffing level is not changed continuously in a project and approximations of the Rayleigh curve are used: assigning a few people at the start, having the peak team during the coding phase, and then leaving a few people for integration and system testing. If we consider design and analysis, build, and test as three major phases, the manpower ramp-up in projects typically resembles the function shown in Figure 5.2 [97].

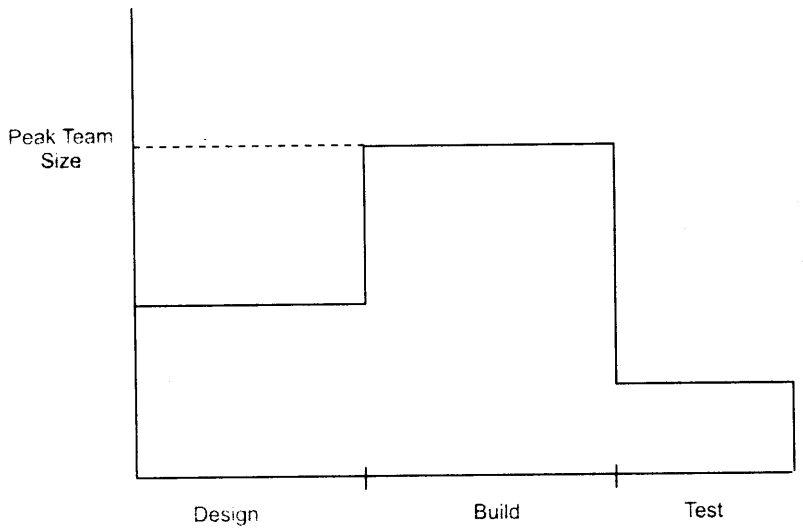

Figure 5.2: Manpower ramp-up in a typical project.

For ease of scheduling, particularly for smaller projects, often the required people are assigned together around the start of the project. This approach can lead to some people being unoccupied at the start and toward the end. This slack time is often used for supporting project activities like training and documentation.

Given the effort estimate for a phase, we can determine the duration of the phase if we know the manpower ramp-up. For these three major phases, the percentage of the schedule consumed in the build phase is smaller than the percentage of the effort consumed because this phase involves more people. Similarly, the percentage of the schedule consumed in the design and testing phases exceeds their effort percentages. The exact schedule depends on the planned manpower ramp-up, and how many resources can be used effectively in a phase on that project. Generally speaking, design requires about a quarter of the schedule, build consumes about half, and integration and system testing consume the remaining quarter. COCOMO gives 19% for design, 62% for programming, and 18% for integration.

5.3.2 Detailed Scheduling

Once the milestones and the resources are fixed, it is time to set the detailed scheduling. For detailed schedules, the major tasks fixed while planning the milestones are broken into small schedulable activities in a hierarchical manner. For example, the detailed design phase can be broken into tasks for developing the detailed design for each module, review of each detailed design, fixing of defects found, and so on. For each detailed task, the project manager estimates the time required to complete it and assigns a suitable resource so that the overall schedule is met.

At each level of refinement, the project manager determines the effort for the overall task from the detailed schedule and checks it against the effort estimates. If this detailed schedule is not consistent with the overall schedule and effort estimates, the detailed schedule must be changed. If it is found that the best detailed schedule cannot match the milestone effort and schedule, then the earlier estimates must be revised. Thus, scheduling is an iterative process.

Generally, the project manager refines the tasks to a level so that the lowest-level activity can be scheduled to occupy no more than a few days from a single resource. Activities related to tasks such as project management, coordination, database management, and configuration management may also be listed in the schedule, even though these activities have less direct effect on determining the schedule because they are ongoing tasks rather than schedulable activities. Nevertheless, they consume resources and hence are often included in the project schedule.

Rarely will a project manager complete the detailed schedule of the entire project all at once. Once the overall schedule is fixed, detailing for a phase may only be done at the start of that phase.

For detailed scheduling, tools like Microsoft Project or a spreadsheet can be very useful. For each lowest-level activity, the project manager specifies the effort, duration, start date, end date, and resources. Dependencies between activities, due either to an inherent dependency (for example, you can conduct a unit test plan for a program only after it has been coded) or to a resource-related dependency (the same resource is assigned two tasks) may also be specified. From these tools the overall effort and schedule of higher level tasks can be determined.

A detailed project schedule is never static. Changes may be needed because the actual progress in the project may be different from what was planned, because newer tasks are added in response to change requests, or because of other unforeseen situations. Changes are done as and when the need arises.

The final schedule, frequently maintained using some suitable tool, is often the most "live" project plan document. During the project, if plans must be changed and additional activities must be done, after the decision is made, the changes must be reflected in the detailed schedule, as this reflects the tasks actually planned to be performed. Hence, the detailed schedule becomes the main document that tracks the activities and schedule.

5.3.3 An Example

Consider the example of a project from [97]. The overall effort estimate for this project is 501 person-days, or about 24 person-months (this estimation was done using the bottom-up approach discussed earlier). The customer gave approximately 5.5 months to finish the project. Because this is more than the square root of effort in person-months, this schedule was accepted.

The milestones are determined by using the effort estimates for the phases and an estimate of the number of resources that can be fully occupied in this phase. Table 5.3 shows the high level schedule of the project. This project uses a process in which initial requirement and design is done in two iterations and the development is done in three iterations. The overall project duration with these milestones is 140 days.

Task	Duration (days)	Work (person -days)	Start date	End date
Project initiation	33.78	24.2	5/4/00	6/23/00
Regular activities	87.11	35.13	6/5/00	10/16/00
Training	95.11	49.37	5/8/00	9/29/00
Knowledge sharing tasks	78.22	19.56	6/2/00	9/30/00
Inception phase	26.67	22.67	4/3/00	5/12/00
Elaboration Iteration 1	27.56	55.16	5/15/00	6/23/00
Elaboration Iteration 2	8.89	35.88	6/26/00	7/7/00
Construction Iteration 1	8.89	24.63	7/10/00	7/21/00
Construction Iteration 2	6.22	28.22	7/20/00	7/28/00
Construction Iteration 3	6.22	27.03	7/31/00	8/8/00
Transition phase	56	179.62	8/9/00	11/3/00
Back-end work	4.44	6.44	8/14/00	8/18/00

Table 5.3: High-level schedule for the project.

This high-level schedule is not suitable for assigning resources and detailed planning. During detailed scheduling, these tasks are broken into schedulable activities. In this way, the schedule also becomes a checklist of tasks for the project. As mentioned before, this exploding of top-level activities is not done fully at the start but rather takes place many times during the project.

Table 5.4 shows part of the detailed schedule of the construction-iteration 1 phase of the project. For each activity, the table specifies the activity by a short name, the module to which the activity is contributing, and the duration and effort. For each task, how much is completed is given in the % Complete column. This information is used for activity tracking. The detailed schedule also specifies the resource to which the task is assigned (specified by initials of the person.) Sometimes, the predecessors of the activity (the activities upon which the task depends) are also specified. This information helps in determining the critical path and the critical resources. This project finally had a total of about 325 schedulable tasks.

Module	Task	Duration (days)	Effort (days)	Start date	End date	% done	Resource
-	Requirements	8.89	1.33	7/10	7/21	100	bb,bj
-	Design review	1	0.9	7/11	7/12	100	bb,bj,sb
-	Rework	1	0.8	7/12	7/13	100	bj, sb
History	coding	2.67	1.87	7/10	7/12	100	hp
History	Review UC17	0.89	0.27	7/14	7/14	100	bj,dd
History	Review UC19	0.89	0.27	7/14	7/14	100	bj,dd
History	Rework	0.89	2.49	7/17	7/17	100	dd,sb,hp
History	Test UC17	0.89	0.62	7/18	7/18	100	sb
History	Test UC19	0.89	0.62	7/18	7/18	100	hp
History	Rework	0.89	0.71	7/18	7/18	100	bj,sb,hp
Config.	Reconciliation	0.89	2.49	7/19	7/19	100	bj,sb,hp
Mgmt.	Tracking	7.11	2.13	7/10	7/19	100	bb
Quality	Analysis	0.89	0.62	7/19	7/19	100	bb

Table 5.4: Portion of the detailed schedule.

5.3.4 Team Structure

We have seen that the number of resources is fixed when schedule is being planned. Detailed scheduling is done only after actual assignment of people has been done, as task assignment needs information about the capabilities of the team members. In our discussion above, we have implicitly assumed that the project's team is led by a project manager, who does the planning and task assignment. This form of hierarchical team organization is fairly common, and was earlier called the Chief Programmer Team.

In this hierarchical organization, the project manager is responsible for all major technical decisions of the project. He does most of the design and assigns coding of the different parts of the design to the programmers. The team typically consists of programmers, testers, a configuration controller, and possibly a librarian for documentation. There may be other roles like database manager, network manager, backup project manager, or a backup configuration controller. It should be noted that these are all logical roles and one person may do multiple such roles.

For a small project, a one-level hierarchy suffices. For larger projects, this organization can be extended easily by partitioning the project into modules, and having module leaders who are responsible for all tasks related to their module and have a team with them for performing these tasks.

A different team organization is the egoless team [114]: Egoless teams consist of ten or fewer programmers. The goals of the group are set by consensus, and input from every member is taken for major decisions. Group leadership rotates among the group members. Due to their nature, egoless teams are sometimes called *democratic teams*. This structure allows input from all members, which can lead to better decisions for difficult problems. This structure is well suited for long-term research-type projects that do not have time constraints. It is not suitable for regular tasks that have time constraints; for such tasks, the communication in democratic structure is unnecessary and results in inefficiency.

In recent times, for very large product developments, another structure has emerged. This structure recognizes that there are three main task categories in software development—management related, development related, and testing related. It also recognizes that it is often desirable to have the test and development team be relatively independent, and also not to have the developers or tests report to a nontechnical manager. In this structure, consequently, there is an overall unit manager, under whom there are three small hierarchic organizations—for program management, for development, and for testing. The primary job of developers is to write code and they work under a development manager. The responsibility of the testers is to test the code and they work under a test manager. The program managers provides the specifications for what is being built, and ensure that development and testing are properly coordinated. In a large product this structure may be replicated, one for each major unit. This type of team organization is used in corporations like Microsoft.

5.4 Software Configuration Management Plan

From the earlier discussions on software configuration management, it should be somewhat clear what the SCM plans should contain. The SCM plan, like other plans, has to identify the activities that must be performed, give guidelines for performing the activities, and allocate resources for them.

Planning for configuration management involves identifying the configuration items and specifying the procedures to be used for controlling and implementing changes to them. We have discussed CM planning while discussing the CM process in Chapter 2. To summarize, the configuration controller does the CM planning when the project has been initiated and the operating environment and requirements specifications are known. The activities in this stage include the following [97]:

- Identify configuration items, including customer-supplied and purchased items.
- Define a naming scheme for configuration items.
- Define the directory structure needed for CM.
- Define version management procedures, and methods for tracking changes to configuration items.
- Define access restrictions.
- Define change control procedures.
- Identify and define the responsibility of the CC.
- Identify points at which baselines will be created.
- Define a backup procedure and a reconciliation procedure, if needed.
- Define a release procedure.

The output of this phase is the CM plan. An example of a full CM plan for a project in a commercial organization is given in [96, 97].

5.5 Quality Plan

Earlier in Chapter 1, we discussed the notion of software quality. Even though there are different dimensions of quality, in practice, quality management often revolves around defects. Hence, we use "delivered defect density"—the number of defects per unit size in the delivered software—as the definition of quality. This definition is currently the de facto industry standard [41]. By defect we mean something in software that causes the software to behave in a manner that is inconsistent with the requirements or needs of the customer. Defect in software implies that its removal will result in some change being made to the software.

To ensure that the final product is of high quality, some quality control (QC) activities must be performed throughout the development. A QC task is one whose main purpose is to identify defects. The purpose of a quality plan in a project is to specify the activities that need to be performed for identifying and removing defects, and the tools and methods that may be used for that purpose.

In a project it is very unlikely that the intermediate work products are of poor quality, but the final product is of high quality. So, to ensure that the delivered software is of good quality, it is essential to ensure that all work products like the requirements specification, design, and test plan are also of good quality. For this reason, a quality plan should contain quality activities throughout the project.

The quality plan specifies the tasks that need to be undertaken at different times in the project to improve the software quality by removing defects, and how they are to be managed. Before we discuss these, let us first understand the defect injection and removal cycle.

5.5.1 Defect Injection and Removal Cycle

Software development is a highly people-oriented activity and hence it is error-prone. Defects can be injected in software at any stage during its evolution. That is, during the transformation from user needs to software to satisfy those needs, defects can be injected in all the transformation activities undertaken. These injection stages are primarily the requirements specification, the high-level design, the detailed design, and coding.

For high-quality software, the final product should have as few defects as possible. Hence, for delivery of high-quality software, active removal of defects through the quality control activities is necessary. The QC activities for defect removal include requirements reviews, design reviews, code reviews, unit testing, integration testing, system testing, and acceptance testing (we do not include reviews of plan documents, although such reviews also help in improving quality of the software). Figure 5.3 shows the process of defect injection and removal.

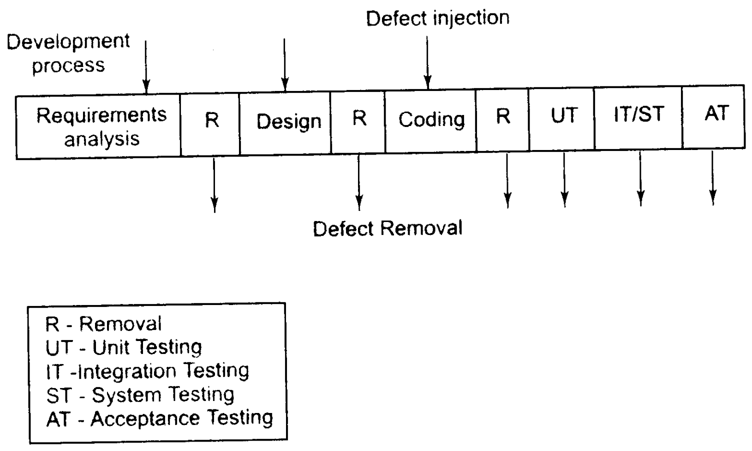

Figure 5.3: Defect injection and removal cycle.

The task of quality management is to plan suitable quality control activities and then to properly execute and control them so the projects quality goals are achieved. With respect to quality control the terms *verification* and *validation* are often used. *Verification* is the process of determining whether or not the products of a given phase of software development fulfill the specifications established during the previous phase. *Validation* is the process of evaluating software at the end of the software development to ensure compliance with the software requirements. Clearly, for high reliability we need to perform both activities. Together they are often called *V&V activities.*

The major V&V activities for software development are inspection and testing (both static and dynamic). The quality plan identifies the different V&V tasks for the different phases and specifies how these tasks contribute to the project V&V goals. The methods to be used for performing these V&V activities, the responsibilities and milestones for each of these activities, inputs and outputs for each V&V task, and criteria for evaluating the outputs are also specified.

5.5.2 Approaches to Quality Management

Reviews and testing are two most common QC activities. Whereas reviews are structured, human-oriented processes, testing is the process of executing software (or parts of it) in an attempt to identify defects. In the *procedural approach to quality management*, procedures and guidelines for the review and testing activities are planned. During project execution, they are carried out according to the defined procedures. In short, the procedural approach is the execution of certain processes at defined points to detect defects.

The procedural approach does not allow claims to be made about the percentage of defects removed or the quality of the software following the procedures completion. In other words, merely executing a set of defect removal procedures does not provide a basis for judging their effectiveness or assessing the quality of the final code. Furthermore, such an approach is highly dependent on the quality of the procedure and the quality of its execution. For example, if the test planning is done carefully and the plan is thoroughly reviewed, the quality of the software after testing will be better than if testing was done but using a test plan that was not carefully thought out or reviewed.

To better assess the effectiveness of the defect detection processes, metrics-based evaluation is necessary. Based on analysis of the data, we can decide whether more testing or reviews are needed. If controls are applied during the project based on quantitative data to achieve quantitative quality goals, then we say that a *quantitative quality management* approach is being applied. Quantitative quality management is an advanced concept, and we only briefly discuss it.

One approach to quantitative quality management is defect prediction. In this approach, the quality goal is set in terms of delivered defect density. Intermediate goals are set by estimating the number of defects that may be identified by various defect detection activities; then the actual number of defects are compared to the estimated defect levels. The effectiveness of this approach depends on how well you can predict the defect levels at various stages of the project. An approach like this requires past data for estimation—an example of this can be found in [97].

Another approach is to use statistical process control (SPC) for managing quality. In this approach, performance expectations of the various QC processes are set, such as testing and reviews, in terms of control limits. If the actual performance of the QC task is not within the limits, the situation is analyzed and suitable action taken. The control limits resemble prediction of defect levels based on past performance but can also be used for monitoring quality activities at a finer level, such as review or unit testing of a module.

5.5.3 Quality Plan

The quality plan for a project is what drives the quality activities in the project. The sophistication of the plan depends on the type of data or prediction models available. At the simplest, the quality plan specifies the quality control tasks that will be performed in the project. Typically, these will be schedulable tasks in the detailed schedule of the project. For example, it will specify what documents will be inspected, what parts of the code will be inspected, and what levels of testing will be performed. The plan will be considerably enhanced if some sense of defect levels that are expected to be found for the different quality control tasks are mentioned—these can then be used for monitoring the quality as the project proceeds.

Much of the quality plan revolves around testing and reviews. Testing will be discussed in detail in a later Chapter. Effectiveness of reviews depends on how they are conducted. One particular process of conducting reviews called inspections was discussed earlier in Chapter 2. This process can be applied to any work product like requirement specifications, design document, test plans, project plans, and code.

5.6 Risk Management

A software project is a complex undertaking. Unforeseen events may have an adverse impact on a projects cost, schedule, or quality. Risk management is an attempt to minimize the chances of failure caused by unplanned events. The aim of risk management is not to avoid getting into projects that have risks but to minimize the impact of risks in the projects that are undertaken.

A risk is a probabilistic event—it may or may not occur. For this reason, we frequently have an optimistic tendency to simply not see risks or to wish that they will not occur. Social and organizational factors also may stigmatize risks and discourage clear identification of them [30]. This kind of attitude gets the project in trouble if the risk events materialize, something that is likely to happen in a large project. Not surprisingly, then, risk management is considered first among the best practices for managing large software projects [26]. It first came to the forefront with Boehm's tutorial on risk management [19]. Since then, several books have targeted risk management for software [29, 78].

5.6.1 Risk Management Concepts

Risk is defined as an exposure to the chance of injury or loss. That is, risk implies that there is a possibility that something negative may happen. In the context of software projects, negative implies that there is an adverse effect on cost, quality, or schedule. *Risk management* is the area that tries to ensure that the impact of risks on cost, quality, and schedule is minimal.

Risk management can be considered as dealing with the possibility and actual occurrence of those events that are not "regular" or commonly expected, that is, they are probabilistic. The commonly expected events, such as people going on leave or some requirements changing, are handled by normal project management. So, in a sense, risk management begins where normal project management ends. It deals with events that are infrequent, somewhat out of the control of the project management, and which can have a major impact on the project.

Most projects have risk. The idea of risk management is to minimize the possibility of risks materializing, if possible, or to minimize the effects if risks actually materialize. For example, when constructing a building, there is a risk that the building may later collapse due to an earthquake. That is, the possibility of an earthquake is a risk. If the building is a large residential complex, then the potential cost in case the earthquake risk materializes can be enormous. This risk can be reduced by shifting to a zone that is not earthquake prone. Alternatively, if this is not acceptable, then the effects of this risk materializing are minimized by suitably constructing the building (the approach taken in Japan and California). At the same time, if a small dumping ground is to be constructed, no such approach might be followed, as the financial and other impact of an actual earthquake on such a building is so low that it does not warrant special measures.

It should be clear that risk management has to deal with identifying the undesirable events that can occur, the probability of their occurring, and the loss if an undesirable event does occur. Once this is known, strategies can be formulated for either reducing the probability of the risk materializing or reducing the effect of risk materializing. So the risk management revolves around *risk assessment* and *risk control.* For each of these major activities, some subactivities must be performed. A breakdown of these activities is given in Figure 5.4 [19].

5.6.2 Risk Assessment

Risk assessment is an activity that must be undertaken during project planning. This involves identifying the risks, analyzing them, and prioritizing them on the basis of the analysis. Due to the nature of a software project, uncertainties are highest near the beginning of the project (just as for cost estimation). Due to this, although risk assessment should be done throughout the project, it is most needed in the starting phases of the project.