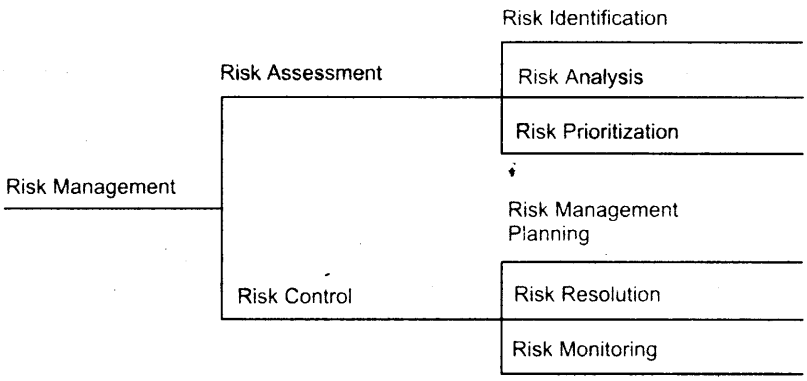

Figure 5.4: Risk management activities.

The goal of risk assessment is to prioritize the risks so that attention and resources can be focused on the more risky items. *Risk identification* is the first step in risk assessment, which identifies all the different risks for a particular project. These risks are project-dependent and identifying them is an exercise in envisioning what can go wrong. Methods that can aid risk identification include checklists of possible risks, surveys, meetings and brainstorming, and reviews of plans, processes, and work products [78].

Checklists of frequently occurring risks are probably the most common tool for risk identification—most organizations prepare a list of commonly occurring risks for projects, prepared from a survey of previous projects. Such a list can form the starting point for identifying risks for the current project.

Based on surveys of experienced project managers, Boehm [19] has produced a list of the top 10 risk items likely to compromise the success of a software project. Though risks in a project are specific to the project, this list forms a good starting point for identifying such risks. Figure 5.5 shows these top 10 items along with the techniques preferred by management for managing these risks. Top risks in a commercial software organization can be found in [97].

The top-ranked risk item is personnel shortfalls. This involves just having fewer people than necessary or not having people with specific skills that a project might require. Some of the ways to manage this risk is to get the top talent possible and to match the needs of the project with the skills of the available personnel. Adequate training, along with having some key personnel for critical areas of the project, will also reduce this risk.

The second item, unrealistic schedules and budgets, happens very frequently due to business and other reasons. It is very common that high-level management imposes a schedule for a software project that is not based on the characteristics of the project and is unrealistic. Underestimation may also happen due to inexperience or optimism.

	RISK ITEM	RISK MANAGEMENT TECHNIQUES
1	Personnel Shortfalls	Staffing with top talent; Job matching; Team building; Key personnel agreements; training; Prescheduling key people
2	Unrealistic Schedules and Budgets	Detailed multi source cost and schedule estimation; Design to cost; Incremental Development; Software reuse; Requirements scrubbing
3	Developing the Wrong Software Functions	Organization analysis; Machine analysis; Ops concept forumlation; User surveys; Prototyping; Early user's manuals
4	Developing the Wrong User Interface	Prototyping; Scenarios; Task analysis; User characterization
5	Gold Plating	Requirements scrubbing; Prototyping; Cost benefit analysis; Design to cost
6	Continuing Stream of Requirement Changes	High change threshold; Information hiding; Incremental development
7	Shortfalls in Externally Furnished Components	Benchmarking inspections; Reference checking; Compatibility analysis
8	Shortfalls in Externally Performed Tasks	Reference checking; Preaward audits; Award free contracts; Competetive design or prototyping; Team building
9	Real Time Performance Shortfalls	Simulation; Benchmarking; Modeling; Prototyping; Instrumentation; Tuning
10	Straining Computer Science Capabilities	Technical analysis; Cost benefit analysis; Prototyping; Reference checking

Figure 5.5: Top 10 risk items and techniques for managing them.

The next few items are related to requirements. Projects run the risk of developing the wrong software if the requirements analysis is not done properly and if development begins too early. Similarly, often improper user interface may be developed. This requires extensive rework of the user interface later or the software benefits are not obtained because users are reluctant to use it. Gold plating refers to adding features in the software that are only marginally useful. This adds unnecessary risk to the project because gold plating consumes resources and time with little return. Some requirement changes are to be expected in any project, but sometimes frequent changes are requested, which is often a reflection of the fact that the client has not yet understood or settled on its own requirements. The effect of requirement changes is substantial in terms of cost, especially if the changes occur when the project has progressed to later phases. Performance shortfalls are critical in real-time systems and poor performance can mean the failure of the project.

If a project depends on externally available components—either to be provided by the client or to be procured as an off-the-shelf component—the project runs some risks. The project might be delayed if the external component is not available on time. The project would also suffer if the quality of the external component is poor or if the component turns out to be incompatible with the other project components or with the environment in which the software is developed or is to operate. If a project relies on technology that is not well developed, it may fail. This is a risk due to straining the computer science capabilities.

Using the checklist of the top 10 risk items is one way to identify risks. This approach is likely to suffice in many projects. The other methods are decision driver analysis, assumption analysis, and decomposition [19]. Decision driver analysis involves questioning and analyzing all the major decisions taken for the project. If a decision has been driven by factors other than technical and management reasons, it is likely to be a source of risk in the project. Such decisions may be driven by politics, marketing, or the desire for short-term gain. Optimistic assumptions made about the project also are a source of risk. Some such optimistic assumptions are that nothing will go wrong in the project, no personnel will quit during the project, people will put in extra hours if required, and all external components (hardware or software) will be delivered on time. Identifying such assumptions will point out the source of risks. An effective method for identifying these hidden assumptions is comparing them with past experience. Decomposition implies breaking a large project into clearly defined parts and then analyzing them. Many software systems have the phenomenon that 20% of the modules cause 80% of the project problems. Decomposition will help identify these modules.

Risk identification merely identifies the undesirable events that might take place during the project, i.e., enumerates the "unforeseen" events that might occur. It does not specify the probabilities of these risks materializing nor the impact on the project if the risks indeed materialize. Hence the next tasks are *risk analysis* and *prioritization.*

In risk analysis, the probability of occurrence of a risk has to be estimated, along with the loss that will occur if the risk does materialize. This is often done through discussion, using experience and understanding of the situation. However, if cost models are used for cost and schedule estimation, then the same models can be used to assess the cost and schedule risk. For example, in the COCOMO cost model, the cost estimate depends on the ratings of the different cost drivers. One possible source of cost risk is underestimating these cost drivers. The other is underestimating the size. Risk analysis can be done by estimating the worst-case value of size and all the cost drivers and then estimating the project cost from these values. This will give us the worst-case analysis. Using the worst-case effort estimate, the worst-case schedule can easily be obtained. A more detailed analysis can be done by considering different cases or a distribution of these drivers.

The other approaches for risk analysis include studying the probability and the outcome of possible decisions (decision analysis), understanding the task dependencies to decide critical activities and the probability and cost of their not being completed on time (network analysis), risks on the various quality factors like reliability and usability (quality factor analysis), and evaluating the performance early through simulation, etc., if there are strong performance constraints on the system (performance analysis). The reader is referred to [19] for further discussion of these topics.

Once the probabilities of risks materializing and losses due to materialization of different risks have been analyzed, they can be prioritized. One approach for prioritization is through the concept of *risk exposure (RE)* [19], which is sometimes called *risk impact.* RE is defined by the relationship

$$RE = Prob(UO) * Loss(UO),$$

where $Prob(UO)$ is the probability of the risk materializing (i.e., undesirable outcome) and $Loss(UO)$ is the total loss incurred due to the unsatisfactory outcome. The loss is not only the direct financial loss that might be incurred but also any loss in terms of credibility, future business, and loss of property or life. The RE is the expected value of the loss due to a particular risk. For risk prioritization using RE, the higher the RE, the higher the priority of the risk item.

It is not always possible to use models and prototypes to assess the probabilities of occurrence and of loss associated with particular events. Due to the nonavailability of models, assessing risk probabilities is frequently subjective. A subjective assessment can be done by the estimate of one person or by using a group consensus technique like the Delphi approach [20]. In the Delphi method, a group of people discusses the problem of estimation and finally converges on a consensus estimate.

5.6.3 Risk Control

The main objective of risk management is to identify the top few risk items and then focus on them. Once a project manager has identified and prioritized the risks, the

top risks can be easily identified. The question then becomes what to do about them. Knowing the risks is of value only if you can prepare a plan so that their consequences are minimal—that is the basic goal of risk management.

One obvious strategy is risk avoidance, which entails taking actions that will avoid the risk altogether, like the earlier example of shifting the building site to a zone that is not earthquake-prone. For some risks, avoidance might be possible.

For most risks, the strategy is to perform the actions that will either reduce the probability of the risk materializing or reduce the loss due to the risk materializing. These are called risk mitigation steps. To decide what mitigation steps to take, a list of commonly used risk mitigation steps for various risks is very useful here. Generally the compiled table of commonly occurring risks also contains the compilation of the methods used for mitigation in the projects in which the risks appeared.

Note that unlike risk assessment, which is largely an analytical exercise, risk mitigation comprises active measures that have to be performed to minimize the impact of risks. In other words, selecting a risk mitigation step is not just an intellectual exercise. The risk mitigation step must be executed (and monitored). To ensure that the needed actions are executed properly, they must be incorporated into the detailed project schedule.

Risk prioritization and consequent planning are based on the risk perception at the time the risk analysis is performed. Because risks are probabilistic events that frequently depend on external factors, the threat due to risks may change with time as factors change. Clearly, then, the risk perception may also change with time. Furthermore, the risk mitigation steps undertaken may affect the risk perception.

This dynamism implies that risks in a project should not be treated as static and must be monitored and reevaluated periodically. Hence, in addition to monitoring the progress of the planned risk mitigation steps, a project must periodically revisit the risk perception and modify the risk mitigation plans, if needed. *Risk monitoring* is the activity of monitoring the status of various risks and their control activities. One simple approach for risk monitoring is to analyze the risks afresh at each major milestone, and change the plans as needed.

5.6.4 A Practical Risk Management Approach

Though the concept of risk exposure is rich, a simple practical way of doing risk planning is to simply categorize risks and the impacts in a few levels and then use it for prioritization. This approach is used in many organizations. Here we discuss a simple approach used in an organization [97]. In this approach, the probability of a risk occurring is categorized as low, medium, or high. The risk impact can be also classified as low, medium, and high. With these ratings, the following simple method for risk prioritization can be specified:

1. For each risk, rate the probability of its happening as low, medium, or high.

2. For each risk, assess its impact on the project as low, medium, or high.

3. Rank the risks based on the probability and effects on the project; for example, a high-probability, high-impact item will have higher rank than a risk item with a medium probability and high impact. In case of conflict, use judgment.

4. Select the top few risk items for mitigation and tracking.

An example of this approach is given in Table 5.5, which shows the various ratings and the risk mitigation steps [97].

Seq Num	Risk	Prob.	Impact	Exp.	Mitigation Plan
1	Failure to meet the high performance	High	High	High	Study white papers and guidelines on perf. Train team on perf. tuning. Update review checklist to look for perf. pitfalls. Test application for perf. during system testing.
2	Lack of people with right skills	Med	Med	Med	Train resources. Review prototype with customer. Develop coding practices.
3	Complexity of application	Med	Med	Med	Ensure ongoing knowledge transfer. Deploy persons with prior experience with the domain.
4	Manpower attrition	Med	Med	Med	Train a core group of four people. Rotate assignments among people. Identify backups for key roles.
5	Unclear requirements	Med	Med	Med	Review a prototype. Conduct a midstage review.

Table 5.5: Risk management plan for a project.

As we can see, the risk management part of the project management plan, which is essentially this table, can be very brief and focused. For monitoring the risks, one way is to redo risk management planning at milestones, giving more attention to the risks listed in the project plan. During risk monitoring at milestones, reprioritization may

occur and mitigation plans for the remainder of the project may change, depending on the current situation and the impact of mitigation steps taken earlier.

5.7 Project Monitoring Plan

A project management plan is merely a document that can be used to guide the execution of a project. Even a good plan is useless unless it is properly executed. And execution cannot be properly driven by the plan unless it is monitored carefully and the actual performance is tracked against the plan.

Monitoring requires measurements to be made to assess the situation of a project. If measurements are to be taken during project execution, we must plan carefully regarding what to measure, when to measure, and how to measure. Hence, measurement planning is a key element in project planning. In addition, how the measurement data will be analyzed and reported must also be planned in advance to avoid the situation of collecting data but not knowing what to do with it. Without careful planning for data collection and its analysis, neither is likely to happen. In this section we discuss the issues of measurements and project tracking.

5.7.1 Measurements

The basic purpose of measurements in a project is to effectively monitor and control the project. For monitoring a project schedule, size, effort, and defects are the basic measurements that are needed [76, 134]. Schedule is one of the most important metrics because most projects are driven by schedules and deadlines. Only by monitoring the actual schedule can we properly assess if the project is on time or if there is a delay. It is, however, easy to measure because calendar time is usually used in all plans.

Effort is the main resource consumed in a software project. Consequently, tracking of effort is a key activity during monitoring; it is essential for evaluating whether the project is executing within budget. For effort data some type of timesheet system is needed where each person working on the project enters the amount of time spent on the project. For better monitoring, the effort spent on various tasks should be logged separately. Generally effort is recorded through some online system (like the weekly activity report system in [96]), which allows a person to record the amount of time spent on a particular activity. At any point, total effort on an activity can be aggregated.

Because defects have a direct relationship to software quality, tracking of defects is critical for ensuring quality. A large software project may include thousands of defects that are found by different people at different stages. Just to keep track of the defects found and their status, defects must be logged and their closure tracked. Once each defect found is logged (and later closed), analysis can focus on how many defects have been found so far, what percentage of defects are still open, and other issues. Defect

tracking is considered one of the best practices for managing a project [26]. We will discuss it in Chapter 10.

Size is another fundamental metric because many data (for example, delivered defect density) are normalized with respect to size. The size of delivered software can be measured in terms of LOC (which can be determined through the use of regular editors and line counters) or function points. At a more gross level, just the number of modules or number of features might suffice.

5.7.2 Project Monitoring and Tracking

The main goal of monitoring is for project managers to get visibility into the project execution so that they can determine whether any action needs to be taken to ensure that the project goals are met. Different types of monitoring might be done for a project. The three main levels of monitoring are activity level, status reporting, and milestone analysis. Measurements taken on the project are employed for monitoring.

Activity-level monitoring ensures that each activity in the detailed schedule has been done properly and within time. This type of monitoring may be done daily in project team meetings or by the project manager checking the status of all the tasks scheduled to be completed on that day. A completed task is often marked as 100% complete in detailed schedule—this is used by tools like the Microsoft Project to track the percentage completion of the overall project or a higher level task.

Status reports are often prepared weekly to to take stock of what has happened and what needs to be done. Status reports typically contain a summary of the activities successfully completed since the last status report, any activities that have been delayed, any issues in the project that need attention, and if everything is in place for the next week.

The *milestone analysis* is done at each milestone or every few weeks, if milestones are too far apart. Analysis of actual versus estimated for effort and schedule is often included in the milestone analysis. If the deviation is significant, it may imply that the project may run into trouble and might not meet its objectives. This situation calls for project managers to understand the reasons for the variation and to apply corrective and preventive actions if necessary.

A graphical method of capturing the basic progress of a project as compared to its plans is the cost-schedule-milestone [20] graph. The graph shows the planned schedule and cost of different milestones, along with shows the actual cost and schedule of achieving the milestones achieved so far. By having both the planned cost versus milestones and the actual cost versus milestones on the same graph, the progress of the project can be grasped easily.

The x-axis of this graph is time, where the months in the project schedule are marked. The y-axis represents the cost, in dollars or PMs. Two curves are drawn. One curve is the planned cost and planned schedule, in which each important milestone of the project is marked. This curve can be completed after the project plan is made. The

second curve represents the actual cost and actual schedule, and the actual achievement of the milestones is marked. Thus, for each milestone the point representing the time when the milestone is actually achieved and the actual cost of achieving it are marked. A cost-schedule-milestone graph for the example is shown in Figure 5.6.

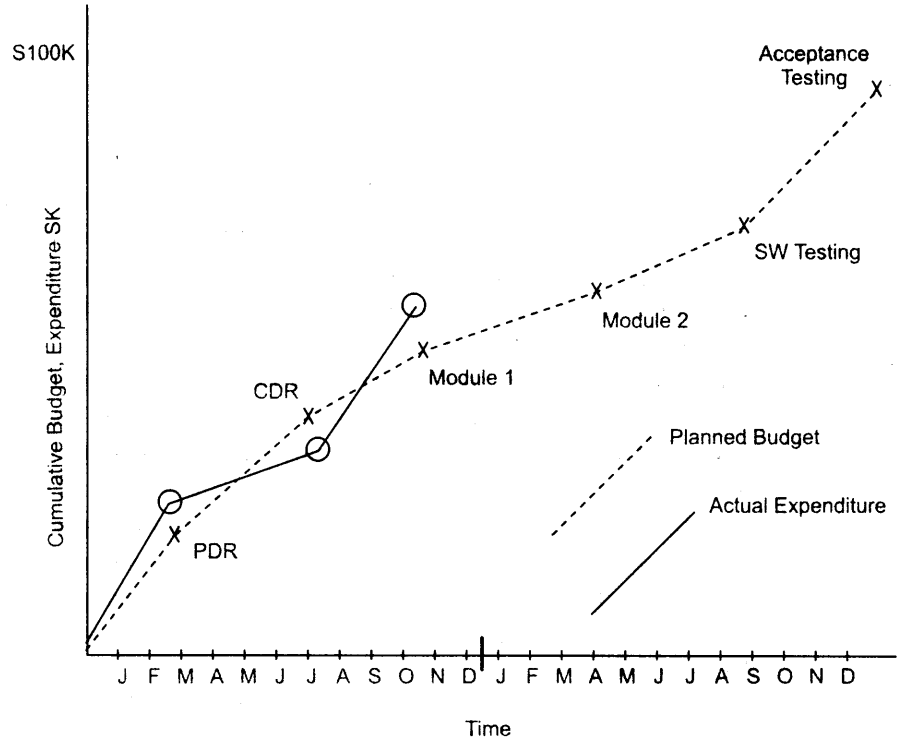

Figure 5.6: A cost-schedule-milestone graph.

The chart shown in Figure 5.6 is for a hypothetical project whose cost is estimated to be $100K. Different milestones have been identified and a curve is drawn with these milestones. The milestones in this project are PDR (preliminary design review), CDR (critical design review), Module 1 completion, Module 2 completion, integration testing, and acceptance testing. For each of these milestones some budget has been allocated based on the estimates. The planned budget is shown by a dotted line. The actual expenditure is shown with a bold line. This chart shows that only two milestones have been achieved, PDR and CDR, and though the project was within budget when PDR was complete, it is now slightly over budget.

5.8 Summary

A proper project plan is an important ingredient for a successful project. Without proper planning, a software development project is unlikely to succeed. Good planning can be done after the requirements and architecture for the project are available. The important planning activities are: process planning, effort estimation, scheduling and staffing planning, quality planning, configuration management planning, project monitoring planning, and risk management.

Process planning generally involves selecting a proper process model and tailoring it to suit the project needs. In effort estimation overall effort requirement for the project and the breakup of the effort for different phases is estimated. In a top-down approach, total effort is first estimated, frequently from the estimate of the size, and then effort for different phases or tasks is determined. In a bottom-up approach, the main tasks in the project are identified, and effort for them is estimated first. From the effort estimates of the tasks, the overall estimate is obtained.

The overall schedule and the major milestones of a project depend on the effort estimate and the staffing level in the project and simple models can be used to get a rough estimate of schedule from effort. Often, an overall schedule is determined using a model, and then adjusted to meet the project needs and constraints. The detailed schedule is one in which the tasks are broken into smaller, schedulable tasks, and then assigned to specific team members, while preserving the overall schedule and effort estimates. The detailed schedule is the most live document of project planning as it lists the tasks that have to be done; any changes in the project plan must be reflected suitably in the detailed schedule.

Quality plans are important for ensuring that the final product is of high quality. The project quality plan identifies all the V&V activities that have to be performed at different stages in the development, and how they are to be performed.

The goal of configuration management is to control the changes that take place during the project. The configuration management plan identifies the configuration items which will be controlled, and specifies the procedures to accomplish this and how access is to be controlled.

Risks are those events which may or may not occur, but if they do occur, they have a negative impact on the project. To meet project goals even under the presence of risks requires proper risk management. Risk management requires that risks be identified, analyzed, and prioritized. Then risk mitigation plans are made and performed to minimize the effect of the highest priority risks.

For a plan to be successfully implemented it is essential that the project be monitored carefully. Activity level monitoring, status reports, and milestone analysis are the mechanisms that are often used. For analysis and reports, the actual effort, schedule,

defects, and size should be measured. With these measurements, it is possible to monitor the performance of a project with respect to its plan. And based on this monitoring, actions can be taken to correct the course of execution, if the need arises.

Overall, project planning lays out the path the project should follow in order to achieve the project objectives. It specifies all the tasks that the project members should perform, and specifies who will do what, in how much time, and when in order to execute this plan. With a detailed plan, what remains to be done is to execute the plan, which is done through the rest of the project. Of course, plans never remain unchanged, as things do not always work as planned. With proper monitoring in place, these situations can be identified and plans changed accordingly. Basic project planning principles and techniques can be used for plan modification also.

Exercises

1. Suppose that the requirements specification phase is divided into two parts: the initial requirements and feasibility study and the detailed requirements specification. Suppose that first part costs about 25% of the total requirement cost. Based on the cost distribution data given earlier, develop a cost estimation model that can be used to predict the cost *after* (a) the feasibility study and (b) the detailed requirements. What are the basic parameters for this cost model? How accurate is this cost model?

2. For the above, if effort is estimated after the feasibility study, some clear risks emerge. What are these and what will be your mitigation plan?

3. Consider a project to develop a full-screen editor. The major components identified are (1) screen edit, (2) command language interpreter, (3) file input and output, (4) cursor movement, and (5) screen movement. The sizes for these are estimated to be 4K, 2K, 1K, 2K, and 3K delivered source code lines. Use the COCOMO model to determine overall effort and schedule estimates (assume values for different cost drivers, with at least three of them being different from 1.0) and effort and schedule estimates for different phases. (b) Use the bottom-up approach given in the chapter to estimate the effort.

4. For the preceding example, assuming that you have a team of 3 people, develop a high level schedule, and a detailed schedule.

5. What are the limitations of the cost estimation models?

6. Assume that testing (and bug fixing) effort is proportional to the number of errors detected (regardless of the nature of errors). Suppose that testing detects 90% of the total errors (10% remain undetected). By adding design and code reviews, suppose the cost of the design and coding phases increases by 10% each (from the base distribution given earlier), and 10% of the errors are detected in design reviews and 10% in code reviews. (So, testing now detects only 70% of errors.) What is the impact on the overall cost of reviews?

7. You want to monitor the effort spent on different phases in the project and the time spent on different components. Design a time sheet or form to be filled in by the programmers that can be used to get this data. The design should be such that automated processing is possible.

8. For a student project being done in a semester course, list the major risks and risk mitigation strategy for them.

9. For a group student project in the software engineering course, device a suitable monitoring plan, and plans for data collection for this monitoring.

10. For a project to manage enrollment and activities in a hobby club, design a suitable [illegible]

11. Suppose a customer gives a project to build parts of a larger system to your group, and other parts to some other groups. Your group has to use an internal tool of the customer, whose new version is to come out soon. Prepare a risk management plan for your project.

12. In a defect injection and removal cycle, suppose the defect injection rates in requirements, design, and coding are 5 defects per KLOC, 10 defects per KLOC, and 60 defects per KLOC respectively. Develop a quality plan and give some removal rates for the different QC tasks in your plan such that the final quality is less than 2 defects per KLOC.

13. For the injection rates given above, suppose the defect removal efficiency of requirement review, design review, unit testing, and system testing are 80% each. What would be the final delivered quality, assuming that these are the only QC tasks performed in the project.

14. In the example above, suppose there are different effort for removing defects in different QC tasks, and that the effort increases as the removal efficiency of the task increases. On what basis would you allocate effort to different QC tasks? (An approach for a general form of this problem can be found in [102].)

Case Studies

Case Study 1—Course Scheduling

Here we present some aspects of developing the project plan. The complete plan is available from the book's Web site. The project has three main modules. The size estimates for these in lines of code are:

Input	650
Schedule	650
Output	150
TOTAL	1450 = 1.45 KLOC

Because this project is somewhat small and straightforward, a waterfall type of process will be used. We use the simple method of determining the total effort from the size based on average productivity. Based on experience and capability of programmers (though no data has been formally collected for this), it is felt that for a project of this size the productivity will be of the order of 600 LOC per PM. From this, we get the effort estimate:

$$E = 1.45 / .6 = 2.4 PM$$

To get the phase-wise breakup of cost we use the distribution of costs given earlier for COCOMO. The phase-wise cost breakup for the project is

Design	2.4 * 0.16 = 0.38 PM
Detailed Design	.26 * 2.4 = 0.62 PM
Coding and Testing	.42 * 2.4 = 1.0 PM
Integration	0.16 * 2.4 = 0.38 PM

The total coding and unit testing effort is one PM, in which the different modules will be coded and tested. We approximate the effort for the different modules in this phase by dividing one PM in the ratio of the sizes of the modules. From this we get the estimate for coding and unit testing of different modules.

The team consists of three persons, all of whom are students who will devote about one-third to one-fourth of their time to the project. A relatively flat team structure will be used with a leader who will allocate tasks to team members. During system design, only the two members will be involved. During detailed design, coding and unit testing, all three will work. There will be no librarian or configuration controller in the project, as it is a small project, and the programmers themselves will do the documentation and configuration management tasks.

The project will produce the following documents (besides the SRS): System design, code, system test plan, and system test report. No unit testing report is needed.

Similarly, detailed design is treated as an activity to help the programmer but its output need not be submitted or reviewed. The quality plan will be fixed accordingly.

The final project plan for the project is available from the Web site.

Case Study 2—PIMS

In this case study, as it was felt that the requirements are not fully clear and may evolve, an interactive development process was chosen, with two iterations. What will be done in the two iterations was decided, as given below.

Iteration 1. Basic functionality of PIMS without authentication and without getting current value data from the Web. That is, all modules related to data access and main control, and modules for key computations.

Iteration 2. Enhance to get current data from the Web, build security, installation module, and the alert system.

A bottom-up estimation was done for these two iterations. The effort and schedule estimates for the two iterations were.

- Iteration 1: 192 person days; 27 days.
- Iteration 2: 88 person-days; 10 days.

The assignment to team members was straightforward. The risk management plan was also simple. The complete project management plan is available from the Web site—it is self explanatory.

Chapter 6

Function-Oriented Design

The design activity begins when the requirements document for the software to be developed is available and the architecture has been designed. During design we further refine the architecture. Generally, design focuses on the what we have called the *module view* in Chapter 4. That is, during design we determine what modules should the system have and which have to be developed. Sometimes, the module view may effectively be a module structure of each component in the architecture. That is, the design exercise determines the module structure of the components. However, this simple mapping of components and modules may not always hold. In that case we have to ensure that the module view created in design is consistent with the architecture.

The design of a system is essentially a blueprint or a plan for a solution for the system. Here we consider a system to be a set of modules with clearly defined behavior which interact with each other in a defined manner to produce some behavior or services for its environment. A module of a system can be considered a system, with its own modules.

The design process for software systems often has two levels. At the first level the focus is on deciding which modules are needed for the system, the specifications of these modules, and how the modules should be interconnected. This is what is called the *system design* or *top-level design.* In the second level, the internal design of the modules, or how the specifications of the module can be satisfied, is decided. This design level is often called *detailed design* or *logic design.* Detailed design essentially expands the system design to contain a more detailed description of the processing logic and data structures so that the design is sufficiently complete for coding.

A *design methodology* is a systematic approach to creating a design by applying of a set of techniques and guidelines. Most design methodologies focus on the system design, and do not reduce the design activity to a sequence of steps that can be blindly followed by the designer.

In this chapter we discuss the function-oriented methods for design and describe one particular methodology—the structured design methodology—in some detail. In

a function-oriented design approach, a system is viewed as a transformation function, transforming the inputs to the desired outputs. The purpose of the design phase is to specify the components for this transformation function, so that each component is also a transformation function. That is, each module in design supports a functional abstraction. The basic output of the system design phase, when a function oriented design approach is being followed, is the definition of all the major data structures in the system, all the major modules of the system, and how the modules interact with each other.

In this chapter, we first discuss some general design principles. Then we discuss a notation for expressing function-oriented designs and describe the structured design methodology for developing a design. Then we discuss some verification methods for design and some metrics that are applicable to function-oriented designs. As in most chapters, we will end with the case studies.

6.1 Design Principles

The design of a system is *correct* if a system built precisely according to the design satisfies the requirements of that system. Clearly, the goal during the design phase is to produce correct designs. However, correctness is not the sole criterion during the design phase, as there can be many correct designs. The goal of the design process is not simply to produce *a* design for the system. Instead, the goal is to find the *best* possible design within the limitations imposed by the requirements and the physical and social environment in which the system will operate.

To evaluate a design, we have to specify some properties and criteria that can be used for evaluation. Ideally, these properties should be as quantitative as possible. In that situation we can precisely evaluate the "goodness" of a design and determine the best design. However, criteria for quality of software design is often subjective or non-quantifiable. In such a situation, criteria are essentially thumb rules that aid design evaluation.

A design should clearly be verifiable, complete (implements all the specifications), and traceable (all design elements can be traced to some requirements). However, the two most important properties that concern designers are efficiency and simplicity. *Efficiency* of any system is concerned with the proper use of scarce resources by the system. The need for efficiency arises due to cost considerations. If some resources are scarce and expensive, it is desirable that those resources be used efficiently. In computer systems, the resources that are most often considered for efficiency are processor time and memory. An efficient system is one that consumes less processor time and requires less memory. In earlier days, the efficient use of CPU and memory was important due to the high cost of hardware. Now that the hardware costs are low compared to the software costs, for many software systems traditional efficiency concerns now take a

back seat compared to other considerations. One of the exceptions is real-time systems, for which there are strict execution time constraints.

Simplicity is perhaps the most important quality criteria for software systems. We have seen that maintenance of software is usually quite expensive. Maintainability of software is one of the goals we have established. The design of a system is one of the most important factors affecting the maintainability of a system. During maintenance, the first step a maintainer has to undertake is to understand the system to be maintained. Only after a maintainer has a thorough understanding of the different modules of the system, how they are interconnected, and how modifying one will affect the others should the modification be undertaken. A simple and understandable design will go a long way in making the job of the maintainer easier.

These criteria are not independent, and increasing one may have an unfavorable effect on another. For example, often the "tricks" used to increase efficiency of a system result in making the system more complex. Therefore, design decisions frequently involve trade-offs. It is the designers' job to recognize the trade-offs and achieve the best balance. For our purposes, simplicity is the primary property of interest, and therefore the objective of the design process is to produce designs that are simple to understand.

Creating a simple (and efficient) design of a large system can be an extremely complex task that requires good engineering judgment. As designing is fundamentally a creative activity, it cannot be reduced to a series of steps that can be simply followed, though guidelines can be provided. In this section we will examine some basic guiding principles that can be used to produce the design of a system. Some of these design principles are concerned with providing means to effectively handle the complexity of the design process. Effectively handling the complexity will not only reduce the effort needed for design (i.e., reduce the design cost), but can also reduce the scope of introducing errors during design. The principles discussed here form the basis for most of the design methodologies.

It should be noted that the principles that can be used in design are the same as those used in problem analysis. In fact, the methods are also similar because in both analysis and design we are essentially constructing models. However, there are some fundamental differences. First, in problem analysis, we are constructing a model of the problem domain, while in design we are constructing a model for the solution domain. Second, in problem analysis, the analyst has limited degrees of freedom in selecting the models as the problem is given, and modeling has to represent it. In design, the designer has a great deal of freedom in deciding the models, as the system the designer is modeling does not exist; in fact the designer is creating a model for the system that will be the basis of building the system. That is, in design, the system depends on the model, while in problem analysis the model depends on the system. Finally, as pointed out earlier, the basic aim of modeling in problem analysis is to understand, while the basic aim of modeling in design is to optimize (in our case, simplicity and

performance). In other words, though the basic principles and techniques might look similar, the activities of analysis and design are very different.

6.1.1 Problem Partitioning and Hierarchy

When solving a small problem, the entire problem can be tackled at once. The complexity of large problems and the limitations of human minds do not allow large problems to be treated as huge monoliths. For solving larger problems, the basic principle is the time-tested principle of "divide and conquer." Clearly, dividing in such a manner that all the divisions have to be conquered together is not the intent of this wisdom. This principle, if elaborated, would mean "divide into smaller pieces, so that each piece can be conquered separately."

For software design, therefore, the goal is to divide the problem into manageably small pieces that can be solved separately. It is this restriction of being able to solve each part separately that makes dividing into pieces a complex task and that many methodologies for system design aim to address. The basic rationale behind this strategy is the belief that if the pieces of a problem are solvable separately, the cost of solving the entire problem is more than the sum of the cost of solving all the pieces.

However, the different pieces cannot be entirely independent of each other, as they together form the system. The different pieces have to cooperate and communicate to solve the larger problem. This communication adds complexity, which arises due to partitioning and may not have existed in the original problem. As the number of components increases, the cost of partitioning, together with the cost of this added complexity, may become more than the savings achieved by partitioning. It is at this point that no further partitioning needs to be done. The designer has to make the judgment about when to stop partitioning.

As discussed earlier, two of the most important quality criteria for software design are simplicity and understandability. It can be argued that maintenance is minimized if each part in the system can be easily related to the application and each piece can be modified separately. If a piece can be modified separately, we call it *independent* of other pieces. If module A is independent of module B, then we can modify A without introducing any unanticipated side effects in B. Total independence of modules of one system is not possible, but the design process should support as much independence as possible between modules. Dependence between modules in a software system is one of the reasons for high maintenance costs. Clearly, proper partitioning will make the system easier to maintain by making the design easier to understand. Problem partitioning also aids design verification.

Problem partitioning, which is essential for solving a complex problem, leads to hierarchies in the design. That is, the design produced by using problem partitioning can be represented as a hierarchy of components. The relationship between the elements in this hierarchy can vary depending on the method used. For example, the most common is the "whole–part of" relationship. In this, the system consists of some parts, each

part consists of subparts, and so on. This relationship can be naturally represented as a hierarchical structure between various system parts. In general, hierarchical structure makes it much easier to comprehend a complex system. Due to this, all design methodologies aim to produce a design that employs hierarchical structures.

6.1.2 Abstraction

Abstraction is a very powerful concept that is used in all engineering disciplines. It is a tool that permits a designer to consider a component at an abstract level without worrying about the details of the implementation of the component. Any component or system provides some services to its environment. An abstraction of a component describes the external behavior of that component without bothering with the internal details that produce the behavior. Presumably, the abstract definition of a component is much simpler than the component itself.

Abstraction is an indispensable part of the design process and is essential for problem partitioning. Partitioning essentially is the exercise in determining the components of a system. However, these components are not isolated from each other; they interact with each other, and the designer has to specify how a component interacts with other components. To decide how a component interacts with other components, the designer has to know, at the very least, the external behavior of other components. If the designer has to understand the details of the other components to determine their external behavior, we have defeated the purpose of partitioning—isolating a component from others. To allow the designer to concentrate on one component at a time, abstraction of other components is used.

Abstraction is used for existing components as well as components that are being designed. Abstraction of existing components plays an important role in the maintenance phase. To modify a system, the first step is understanding what the system does and how. The process of comprehending an existing system involves identifying the abstractions of subsystems and components from the details of their implementations. Using these abstractions, the behavior of the entire system can be understood. This also helps determine how modifying a component affects the system.

During the design process, abstractions are used in the reverse manner than in the process of understanding a system. During design, the components do not exist, and in the design the designer specifies only the abstract specifications of the different components. The basic goal of system design is to specify the modules in a system and their abstractions. Once the different modules are specified, during the detailed design the designer can concentrate on one module at a time. The task in detailed design and implementation is essentially to implement the modules so that the abstract specifications of each module are satisfied.

There are two common abstraction mechanisms for software systems: *functional abstraction* and *data abstraction*. In *functional abstraction*, a module is specified by the function it performs. For example, a module to compute the log of a value can

be abstractly represented by the function log. Similarly, a module to sort an input array can be represented by the specification of sorting. Functional abstraction is the basis of partitioning in function-oriented approaches. That is, when the problem is being partitioned, the overall transformation function for the system is partitioned into smaller functions that comprise the system function. The decomposition of the system is in terms of functional modules.

The second unit for abstraction is *data abstraction.* Any entity in the real world provides some services to the environment to which it belongs. Often the entities provide some fixed predefined services. The case of data entities is similar. Certain operations are required from a data object, depending on the object and the environment in which it is used. Data abstraction supports this view. Data is not treated simply as objects, but is treated as objects with some predefined operations on them. The operations defined on a data object are the only operations that can be performed on those objects. From outside an object, the internals of the object are hidden; only the operations on the object are visible. Data abstraction forms the basis for *object-oriented design*, which is discussed in the next chapter. In using this abstraction, a system is viewed as a set of objects providing some services. Hence, the decomposition of the system is done with respect to the objects the system contains.

6.1.3 Modularity

As mentioned earlier, the real power of partitioning comes if a system is partitioned into modules so that the modules are solvable and modifiable separately. It will be even better if the modules are also separately compilable (then changes in a module will not require recompilation of the whole system). A system is considered *modular* if it consists of discreet components so that each component can be implemented separately, and a change to one component has minimal impact on other components.

Modularity is a clearly a desirable property in a system. Modularity helps in system debugging—isolating the system problem to a component is easier if the system is modular; in system repair—changing a part of the system is easy as it affects few other parts; and in system building—a modular system can be easily built by "putting its modules together."

A software system cannot be made modular by simply chopping it into a set of modules. For modularity, each module needs to support a well-defined abstraction and have a clear interface through which it can interact with other modules. Modularity is where abstraction and partitioning come together. For easily understandable and maintainable systems, modularity is clearly the basic objective; partitioning and abstraction can be viewed as concepts that help achieve modularity.

6.1.4 Top-Down and Bottom-Up Strategies

A system consists of components, which have components of their own; indeed a system is a hierarchy of components. The highest-level component correspond to the total

system. To design such a hierarchy there are two possible approaches: top-down and bottom-up. The top-down approach starts from the highest-level component of the hierarchy and proceeds through to lower levels. By contrast, a bottom-up approach starts with the lowest-level component of the hierarchy and proceeds through progressively higher levels to the top-level component.

A top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved. Top-down design methods often result in some form of *stepwise refinement.* Starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly. The top-down approach has been promulgated by many researchers and has been found to be extremely useful for design. Most design methodologies are based on the top-down approach.

A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower-level components. Bottom-up methods work with *layers of abstraction.* Starting from the very bottom, operations that provide a layer of abstraction are implemented. The operations of this layer are then used to implement more powerful operations and a still higher layer of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.

A top-down approach is suitable only if the specifications of the system are clearly known and the system development is from scratch. However, if a system is to be built from an existing system, a bottom-up approach is more suitable, as it starts from some existing components. So, for example, if an iterative enhancement type of process is being followed, in later iterations, the bottom-up approach could be more suitable (in the first iteration a top-down approach can be used).

Pure top-down or pure bottom-up approaches are often not practical. For a bottom-up approach to be successful, we must have a good notion of the top to which the design should be heading. Without a good idea about the operations needed at the higher layers, it is difficult to determine what operations the current layer should support. Top-down approaches require some idea about the feasibility of the components specified during design. The components specified during design should be implementable, which requires some idea about the feasibility of the lower-level parts of a component. A common approach to combine the two approaches is to provide a layer of abstraction for the application domain of interest through libraries of functions, which contains the functions of interest to the application domain. Then use a top-down approach to determine the modules in the system, assuming that the abstract machine available for implementing the system provides the operations supported by the abstraction layer. This approach is frequently used for developing systems. It can even be claimed that it is almost universally used these days, as most developments now make use of the

layer of abstraction supported in a system consisting of the library functions provided by operating systems, programming languages, and special-purpose tools.

6.2 Module-Level Concepts

In the previous section we discussed some general design principles. Now we turn our attention to some concepts specific to function-oriented design. Before we discuss these, let us define what we mean by a module. A *module* is a logically separable part of a program. It is a program unit that is discreet and identifiable with respect to compiling and loading. In terms of common programming language constructs, a module can be a macro, a function, a procedure (or subroutine), a process, or a package. In systems using functional abstraction, a module is usually a procedure of function or a collection of these.

To produce modular designs, some criteria must be used to select modules so that the modules support well-defined abstractions and are solvable and modifiable separately. In a system using functional abstraction, coupling and cohesion are two modularization criteria, which are often used together.

6.2.1 Coupling

Two modules are considered independent if one can function completely without the presence of other. Obviously, if two modules are independent, they are solvable and modifiable separately. However, all the modules in a system cannot be independent of each other, as they must interact so that together they produce the desired external behavior of the system. The more connections between modules, the more dependent they are in the sense that more knowledge about one module is required to understand or solve the other module. Hence, the fewer and simpler the connections between modules, the easier it is to understand one without understanding the other. The notion of coupling [138, 154] attempts to capture this concept of "how strongly" different modules are interconnected.

Coupling between modules is the strength of interconnections between modules or a measure of interdependence among modules. In general, the more we must know about module A in order to understand module B, the more closely connected A is to B. "Highly coupled" modules are joined by strong interconnections, while "loosely coupled" modules have weak interconnections. Independent modules have no interconnections. To solve and modify a module separately, we would like the module to be loosely coupled with other modules. The choice of modules decides the coupling between modules. Because the modules of the software system are created during system design, the coupling between modules is largely decided during system design and cannot be reduced during implementation.

Coupling increases with the complexity and obscurity of the interface between modules. To keep coupling low we would like to minimize the number of interfaces per module and the complexity of each interface. An interface of a module is used to pass information to and from other modules. Coupling is reduced if only the defined entry interface of a module is used by other modules (for example, passing information to and from a module exclusively through parameters). Coupling would increase if a module is used by other modules via an indirect and obscure interface, like directly using the internals of a module or using shared variables.

Complexity of the interface is another factor affecting coupling. The more complex each interface is, the higher will be the degree of coupling. For example, complexity of the entry interface of a procedure depends on the number of items being passed as parameters and on the complexity of the items. Some level of complexity of interfaces is required to support the communication needed between modules. However, often more than this minimum is used. For example, if a field of a record is needed by a procedure, often the entire record is passed, rather than just passing that field of the record. By passing the record we are increasing the coupling unnecessarily. Essentially, we should keep the interface of a module as simple and small as possible.

The type of information flow along the interfaces is the third major factor affecting coupling. There are two kinds of information that can flow along an interface: data or control. Passing or receiving control information means that the action of the module will depend on this control information, which makes it more difficult to understand the module and provide its abstraction. Transfer of data information means that a module passes as input some data to another module and gets in return some data as output. This allows a module to be treated as a simple input-output function that performs some transformation on the input data to produce the output data. In general, interfaces with only data communication result in the lowest degree of coupling, followed by interfaces that only transfer control data. Coupling is considered highest if the data is hybrid, that is, some data items and some control items are passed between modules. The effect of these three factors on coupling is summarized in Table 6 [138].

	Interface Complexity	Type of Connection	Type of Communication
Low	Simple obvious	To module by name	Data
			Control
High	Complicated obscure	To internal elements	Hybrid

Table 6.1: Factors affecting coupling.

6.2.2 Cohesion

We have seen that coupling is reduced when the relationships among elements in different modules are minimized. That is, coupling is reduced when elements in different modules have little or no bonds between them. Another way of achieving this effect is to strengthen the bond between elements of the same module by maximizing the relationship between elements of the same module. Cohesion is the concept that tries to capture this intra-module [138, 154]. With cohesion, we are interested in determining how closely the elements of a module are related to each other.

Cohesion of a module represents how tightly bound the internal elements of the module are to one another. Cohesion of a module gives the designer an idea about whether the different elements of a module belong together in the same module. Cohesion and coupling are clearly related. Usually, the greater the cohesion of each module in the system, the lower the coupling between modules is. This correlation is not perfect, but it has been observed in practice. There are several levels of cohesion:

- Coincidental
- Logical
- Temporal
- Procedural
- Communicational
- Sequential
- Functional

Coincidental is the lowest level, and functional is the highest. These levels do not form a linear scale. Functional binding is much stronger than the rest, while the first two are considered much weaker than others. Often, many levels can be applicable when considering cohesion between two elements of a module. In such situations, the highest level is considered. Cohesion of a module is considered the highest level of cohesion applicable to all elements in the module.

Coincidental cohesion occurs when there is no meaningful relationship among the elements of a module. Coincidental cohesion can occur if an existing program is "modularized" by chopping it into pieces and making different pieces modules. If a module is created to save duplicate code by combining some part of code that occurs at many different places, that module is likely to have coincidental cohesion. In this situation, the statements in the module have no relationship with each other, and if one of the modules using the code needs to be modified and this modification includes the common code, it is likely that other modules using the code do not want the code modified. Consequently, the modification of this "common module" may cause other modules to behave

incorrectly. The modules using these modules are therefore not modifiable separately and have strong interconnection between them. We can say that, generally speaking, it is poor practice to create a module merely to avoid duplicate code (unless the common code happens to perform some identifiable function, in which case the statements will have some relationship between them) or to chop a module into smaller modules to reduce the module size.

A module has logical cohesion if there is some logical relationship between the elements of a module, and the elements perform functions that fall in the same logical class. A typical example of this kind of cohesion is a module that performs all the inputs or all the outputs. In such a situation, if we want to input or output a particular record, we have to somehow convey this to the module. Often, this will be done by passing some kind of special status flag, which will be used to determine what statements to execute in the module. Besides resulting in hybrid information flow between modules, which is generally the worst form of coupling between modules, such a module will usually have tricky and clumsy code. In general, logically cohesive modules should be avoided, if possible.

Temporal cohesion is the same as logical cohesion, except that the elements are also related in time and are executed together. Modules that perform activities like "initialization," "clean-up," and "termination" are usually temporally bound. Even though the elements in a temporally bound module are logically related, temporal cohesion is higher than logical cohesion, because the elements are all executed together. This avoids the problem of passing the flag, and the code is usually simpler.

A procedurally cohesive module contains elements that belong to a common procedural unit. For example, a loop or a sequence of decision statements in a module may be combined to form a separate module. Procedurally cohesive modules often occur when modular structure is determined from some form of flowchart. Procedural cohesion often cuts across functional lines. A module with only procedural cohesion may contain only part of a complete function or parts of several functions.

A module with communicational cohesion has elements that are related by a reference to the same input or output data. That is, in a communicationally bound module, the elements are together because they operate on the same input or output data. An example of this could be a module to "print and punch record." Communicationally cohesive modules may perform more than one function. However, communicational cohesion is sufficiently high as to be generally acceptable if alternative structures with higher cohesion cannot be easily identified.

When the elements are together in a module because the output of one forms the input to another, we get sequential cohesion. If we have a sequence of elements in which the output of one forms the input to another, sequential cohesion does not provide any guidelines on how to combine them into modules. Different possibilities exist: combine all in one module, put the first half in one and the second half in another, the first third in one and the rest in the other, and so forth. Consequently, a sequentially

bound module may contain several functions or parts of different functions. Sequentially cohesive modules bear a close resemblance to the problem structure. However, they are considered to be far from the ideal, which is functional cohesion.

Functional cohesion is the strongest cohesion. In a functionally bound module, all the elements of the module are related to performing a single function. By function, we do not mean simply mathematical functions; modules accomplishing a single goal are also included. Functions like "compute square root" and "sort the array" are clear examples of functionally cohesive modules.

How does one determine the cohesion level of a module? There is no mathematical formula that can be used. We have to use our judgment for this. A useful technique for determining if a module has functional cohesion is to write a sentence that describes, fully and accurately, the function or purpose of the module. The following tests can then be made [138]:

1. If the sentence must be a compound sentence, if it contains a comma, or it has has more than one verb, the module is probably performing more than one function, and it probably has sequential or communicational cohesion.

2. If the sentence contains words relating to time, like "first," "next," "when," and "after" the module probably has sequential or temporal cohesion.

3. If the predicate of the sentence does not contain a single specific object following the verb (such as "edit all data") the module probably has logical cohesion.

4. Words like "initialize" and "cleanup" imply temporal cohesion.

Modules with functional cohesion can always be described by a simple sentence. However, if a description is a compound sentence, it does not mean that the module does not have functional cohesion. Functionally cohesive modules can also be described by compound sentences. If we cannot describe it using a simple sentence, the module is not likely to have functional cohesion.

6.3 Design Notation and Specification

During the design phase there are two things of interest: the design of the system, the producing of which is the basic objective of this phase, and the process of designing itself. It is for the latter that principles and methods are needed. In addition, while designing, a designer needs to record his thoughts and decisions and to represent the design so that he can view it and play with it. For this, design notations are used.

Design notations are largely meant to be used during the process of design and are used to represent design or design decisions. They are meant largely for the designer so

that he can quickly represent his decisions in a compact manner that he can evaluate and modify. These notations are frequently graphical.

Once the designer is satisfied with the design he has produced, the design is to be precisely specified in the form of a document. Whereas a design represented using the design notation is largely to be used by the designer, a design specification has to be so precise and complete that it can be used as a basis of further development by other programmers. Often, design specification uses textual structures, with design notation helping understanding.

6.3.1 Structure Charts

For a function-oriented design, the design can be represented graphically by structure charts. The structure of a program is made up of the modules of that program together with the interconnections between modules. Every computer program has a structure, and given a program its structure can be determined. The structure chart of a program is a graphic representation of its structure. In a structure chart a module is represented by a box with the module name written in the box. An arrow from module A to module B represents that module A invokes module B. B is called the *subordinate* of A, and A is called the *superordinate* of B. The arrow is labeled by the parameters received by B as input and the parameters returned by B as output, with the direction of flow of the input and output parameters represented by small arrows. The parameters can be shown to be data (unfilled circle at the tail of the label) or control (filled circle at the tail). As an example consider the structure of the following program, whose structure is shown in Figure 6.1.

```
{}
        main()
        {
            int sum, n, N, a[MAX];
            readnums(a, &N); sort(a, N); scanf(&n);
            sum = add_n(a, n); printf(sum);
        }

        readnums(a, N)
        int a[], *N;
        {
            :
        }

        sort(a, N)
        int a[], N;
        {
            :
            if (a[i] > a[t]) switch(a[i], a[t]);
            :
```

```
}

/* Add the first n numbers of a */
add_n(a, n)
int a[], n;
{
    :
}
```

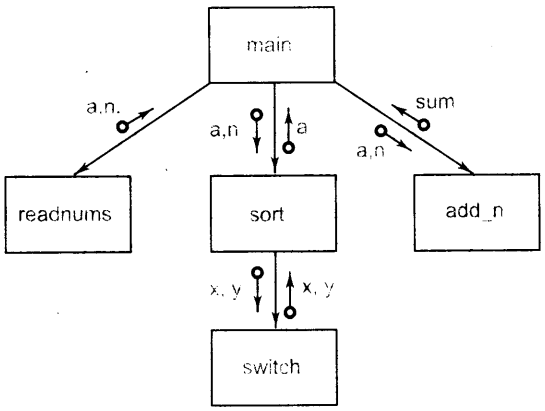

Figure 6.1: The structure chart of the sort program.

In general, procedural information is not represented in a structure chart, and the focus is on representing the hierarchy of modules. However, there are situations where the designer may wish to communicate certain procedural information explicitly, like major loops and decisions. Such information can also be represented in a structure chart. For example, let us consider a situation where module A has subordinates B, C, and D, and A repeatedly calls the modules C and D. This can be represented by a looping arrow around the arrows joining the subordinates C and D to A, as shown in Figure 6.2. All the subordinate modules activated within a common loop are enclosed in the same looping arrow.

Major decisions can be represented similarly. For example, if the invocation of modules C and D in module A depends on the outcome of some decision, that is represented by a small diamond in the box for A, with the arrows joining C and D coming out of this diamond, as shown in Figure 6.2.

Modules in a system can be categorized into few classes. There are some modules that obtain information from their subordinates and then pass it to their superordinate. This kind of module is an *input module.* Similarly, there are *output modules.* that take

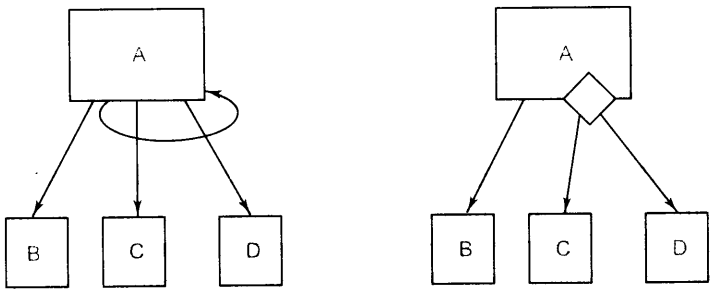

Figure 6.2: Iteration and decision representation.

information from their superordinate and pass it on to its subordinates. As the name suggests, the input and output modules are typically used for input and output of data from and to the environment. The input modules get the data from the sources and get it ready to be processed, and the output modules take the output produced and prepare it for proper presentation to the environment. Then there are modules that exist solely for the sake of transforming data into some other form. Such a module is called a *transform module.* Most of the computational modules typically fall in this category. Finally, there are modules whose primary concern is managing the flow of data to and from different subordinates. Such modules are called *coordinate modules.* The structure chart representation of the different types of modules is shown in Figure 6.3.

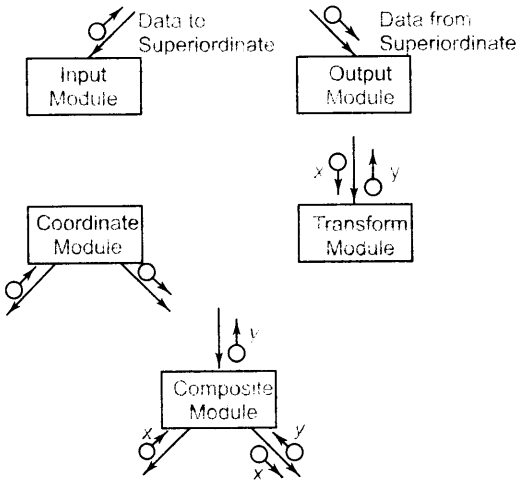

Figure 6.3: Different types of modules.

A module can perform functions of more than one type of module. For example, the composite module in Figure 6.3 is an input module from the point of view of its superordinate, as it feeds the data Y to the superordinate. Internally, A is a coordinate module and views its job as getting data X from one subordinate and passing it to another subordinate, which converts it to Y. Modules in actual systems are often composite modules.

A structure chart is a nice representation mechanism for a design that uses functional abstraction. It shows the modules and their call hierarchy, the interfaces between the modules, and what information passes between modules. It is a convenient and compact notation that is very useful while creating the design. That is, a designer can make effective use of structure charts to represent the model he is creating while he is designing. However, it is not sufficient for representing the final design, as it does not give all the information needed about the design. For example, it does not specify the scope, structure of data, specifications of each module, etc. Hence, it is generally supplemented with textual specifications to convey design to the implementer.

We have seen how to determine the structure of an existing program. But once the program is written, its structure is fixed and little can be done about altering the structure. However, for a given set of requirements many different programs can be written to satisfy the requirements, and each program can have a different structure. That is, although the structure of a given program is fixed, for a given set of requirements, programs with different structures can be obtained. The objective of the design phase using function-oriented method is to control the eventual structure of the system by fixing the structure during design.

6.3.2 Specification

Using some design rules or methodology, a conceptual design of the system can be produced in terms of a structure chart. As seen earlier, in a structure chart each module is represented by a box with a name. The functionality of the module is essentially communicated by the name of the box, and the interface is communicated by the data items labeling the arrows This is alright while the designer is designing but inadequate when the design is to be communicated. To avoid these problems, a design specification should define the major data structures, modules and their specifications, and design decisions.

During system design, the major data structures for the software are identified; without these, the system modules cannot be meaningfully defined during design. In the design specification, a formal definition of these data structures should be given.

Module specification is the major part of system design specification. All modules in the system should be identified when the system design is complete, and these modules should be specified in the document. During system design only the module specification is obtained, because the internal details of the modules are defined later. To specify

a module, the design document must specify (a) the *interface of the module* (all data items, their types, and whether they are for input and/or output), (b) the *abstract behavior* of the module (*what* the module does) by specifying the module's functionality or its input/output behavior, and (c) all other modules used by the module being specified—this information is quite useful in maintaining and understanding the design.

Hence, a design specification will necessarily contain specification of the major data structures and modules in the system. After a design is approved (using some verification mechanism), the modules will have to be implemented in the target language. This requires that the module "headers" for the target language first be created from the design. This translation of the design for the target language can introduce errors if it's done manually. To eliminate these translation errors, if the target language is known (as is generally the case after the requirements have been specified), it is better to have a design specification language whose module specifications can be used almost directly in programming. This not only minimizes the translation errors that may occur, but also reduces the effort required for translating the design to programs. It also adds incentive for designers to properly specify their design, as the design is no longer a "mere" document that will be thrown away after review—it will now be used directly in coding. In the case study, a design specification language close to C has been used. From the design, the module headers for C can easily be created with some simple editing.

To aid the comprehensibility of the design, all major *design decisions* made by the designers during the design process should be explained explicitly. The choices that were available and the reasons for making a particular choice should be explained. This makes a design more *visible* and will help in understanding the design.

6.4 Structured Design Methodology

Creating the software system design is the major concern of the design phase. Many design techniques have been proposed over the years to provide some discipline in handling the complexity of designing large systems. The aim of design methodologies is not to reduce the process of design to a sequence of mechanical steps but to provide guidelines to aid the designer during the design process. Here we describe the structured design methodology [138, 154] for developing system designs.

Structured design methodology (SDM) views every software system as having some inputs that are converted into the desired outputs by the software system. The software is viewed as a transformation function that transforms the given inputs into the desired outputs, and the central problem of designing software systems is considered to be properly designing this transformation function. Due to this view of software, the structured design methodology is primarily function-oriented and relies heavily on functional abstraction and functional decomposition.

The concept of the structure of a program lies at the heart of the structured design method. During design, structured design methodology aims to control and influence the structure of the final program. The aim is to design a system so that programs implementing the design would have a hierarchical structure, with functionally cohesive modules and as few interconnections between modules as possible.

In properly designed systems it is often the case that a module with subordinates does not actually perform much computation. The bulk of actual computation is performed by its subordinates, and the module itself largely coordinates the data flow between the subordinates to get the computation done. The subordinates in turn can get the bulk of their work done by their subordinates until the "atomic" modules, which have no subordinates, are reached. *Factoring* is the process of decomposing a module so that the bulk of its work is done by its subordinates. A system is said to be completely factored if all the actual processing is accomplished by bottom-level atomic modules and if non-atomic modules largely perform the jobs of control and coordination. SDM attempts to achieve a structure that is close to being completely factored.

The overall strategy is to identify the input and output streams and the primary transformations that have to be performed to produce the output. High-level modules are then created to perform these major activities, which are later refined. There are four major steps in this strategy:

1. Restate the problem as a data flow diagram
2. Identify the input and output data elements
3. Perform factoring
4. Design the input, output, and transform modules

We will now discuss each of these steps in more detail. The design of the case study using structured design will be given later. For illustrating each step of the methodology as we discuss them, we consider the following problem: there is a text file containing words separated by blanks or new lines. We have to design a software system to determine the number of unique words in the file.

6.1.1 Restate the Problem as a Data Flow Diagram

To use the SD methodology, the first step is to construct the data flow diagram for the problem. We studied data flow diagrams in Chapter 3. However, there is a fundamental difference between the DFDs drawn during requirements analysis and those drawn during structured design. In the requirements analysis, a DFD is drawn to model the problem domain. The analyst has little control over the problem, and hence his task is to extract from the problem all the information and then represent it as a DFD.

During design activity, we are no longer modeling the problem domain, but rather are dealing with the solution domain and developing a model for the *eventual system.* That is, the DFD during design represents how the data will flow in the system when it is built. In this modeling, the major transforms or functions in the software are decided, and the DFD shows the major transforms that the software will have and how the data will flow through different transforms. So, drawing a DFD for design is a very creative activity in which the designer visualizes the eventual system and its processes and data flows. As the system does not yet exist, the designer has complete freedom in creating a DFD that will solve the problem stated in the SRS. The general rules of drawing a DFD remain the same; we show what transforms are needed in the software and are not concerned with the logic for implementing them. Consider the example of the simple automated teller machine that allows customers to withdraw money. A DFD for this ATM is shown in Figure 6.4.

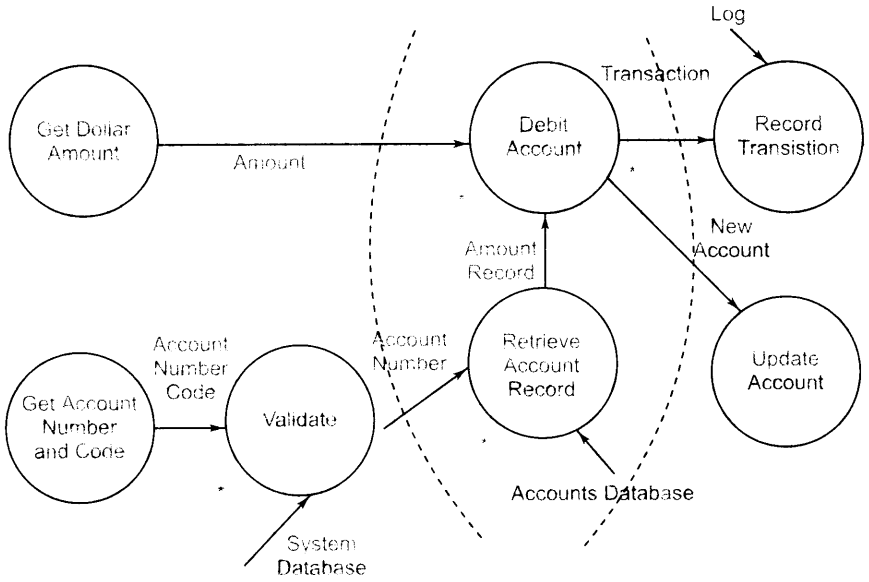

Figure 6.4: Data flow diagram of an ATM.

There are two major streams of input data in this diagram. The first is the account number and the code, and the second is the amount to be debited. The DFD is self-explanatory. Notice the use of * at different places in the DFD. For example, the transform "validate," which verifies if the account number and code are valid, needs not only the account number and code, but also information from the system database to do the validation. And the transform debit account has two outputs, one used for recording the transaction and the other to update the account.

As another example, consider the problem of determining the number of different words in an input file. The data flow diagram for this problem is shown in Figure 6.5.

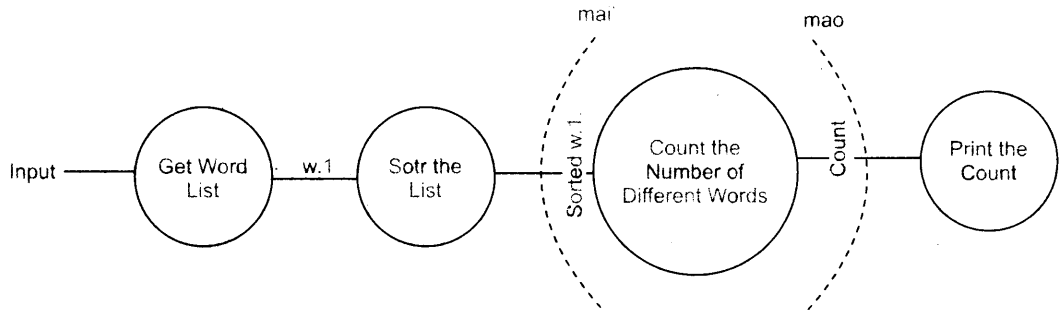

Figure 6.5: DFD for the word-counting problem.

This problem has only one input data stream, the input file, while the desired output is the count of different words in the file. To transform the input to the desired output, the first thing we do is form a list of all the words in the file. It is best to then sort the list, as this will make identifying different words easier. This sorted list is then used to count the number of different words, and the output of this transform is the desired count, which is then printed. This sequence of data transformation is what we have in the data flow diagram.

6.1.2 Identify the Most Abstract Input and Output Data Elements

Most systems have some basic transformations that perform the required operations. However, in most cases the transformation cannot be easily applied to the actual physical input and produce the desired physical output. Instead, the input is first converted into a form on which the transformation can be applied with ease. Similarly, the main transformation modules often produce outputs that have to be converted into the desired physical output. The goal of this second step is to separate the transforms in the data flow diagram that convert the input or output to the desired format from the ones that perform the actual transformations.

For this separation, once the data flow diagram is ready, the next step is to identify the highest abstract level of input and output. *The most abstract input data elements* are those data elements in the data flow diagram that are farthest removed from the physical inputs but can still be considered inputs to the system. The most abstract input data elements often have little resemblance to the actual physical data. These are often the data elements obtained after operations like error checking, data validation, proper formatting, and conversion are complete.

Most abstract input (MAI) data elements are recognized by starting from the physical inputs and traveling toward the outputs in the data flow diagram, until the data elements are reached that can no longer be considered incoming. The aim is to go as far as possible from the physical inputs, without losing the incoming nature of the data element. This process is performed for each input stream. Identifying the most abstract data items represents a value judgment on the part of the designer, but often the choice is obvious.

Similarly, we identify the *most abstract output data elements* (MAO) by starting from the outputs in the data flow diagram and traveling toward the inputs. These are the data elements that are most removed from the actual outputs but can still be considered outgoing. The MAO data elements may also be considered the logical output data items, and the transforms in the data flow diagram after these data items are basically to convert the logical output into a form in which the system is required to produce the output.

There will usually be some transforms left between the most abstract input and output data items. These *central transforms* perform the basic transformation for the system, taking the most abstract input and transforming it into the most abstract output. The purpose of having central transforms deal with the most abstract data items is that the modules implementing these transforms can concentrate on performing the transformation without being concerned with converting the data into proper format, validating the data, and so forth. It is worth noting that if a central transform has two outputs with a + between them, it often indicates the presence of a major decision in the transform (which can be shown in the structure chart).

Consider the data flow diagram shown in Figure 6.5. The arcs in the data flow diagram are the most abstract input and most abstract output. The choice of the most abstract input is obvious. We start following the input. First, the input file is converted into a word list, which is essentially the input in a different form. The sorted word list is still basically the input, as it is still the same list, in a different order. This appears to be the most abstract input because the next data (i.e., count) is not just another form of the input data. The choice of the most abstract output is even more obvious; count is the natural choice (a data that is a form of input will not usually be a candidate for the most abstract output). Thus we have one central transform, count-number-of-different-words, which has one input and one output data item.

Consider now the data flow diagram of the automated teller shown in Figure 6.4. The two most abstract inputs are the dollar amount and the validated account number. The validated account number is the most abstract input, rather than the account number read in, as it is still the input—but with a guarantee that the account number is valid. The two abstract outputs are obvious. The abstract inputs and outputs are marked in the data flow diagram.

6.4.3 First-Level Factoring

Having identified the central transforms and the most abstract input and output data items, we are ready to identify some modules for the system. We first specify a main module, whose purpose is to invoke the subordinates. The main module is therefore a coordinate module. For each of the most abstract input data items, an immediate subordinate module to the main module is specified. Each of these modules is an input module, whose purpose is to deliver to the main module the most abstract data item for which it is created.

Similarly, for each most abstract output data item, a subordinate module that is an output module that accepts data from the main module is specified. Each of the arrows connecting these input and output subordinate modules are labeled with the respective abstract data item flowing in the proper direction.

Finally, for each central transform, a module subordinate to the main one is specified. These modules will be transform modules, whose purpose is to accept data from the main module, and then return the appropriate data back to the main module. The data items coming to a transform module from the main module are on the incoming arcs of the corresponding transform in the data flow diagram. The data items returned are on the outgoing arcs of that transform. Note that here a module is created for a transform, while input/output modules are created for data items. The structure after the first-level factoring of the word-counting problem (its data flow diagram was given earlier) is shown in Figure 6.6.

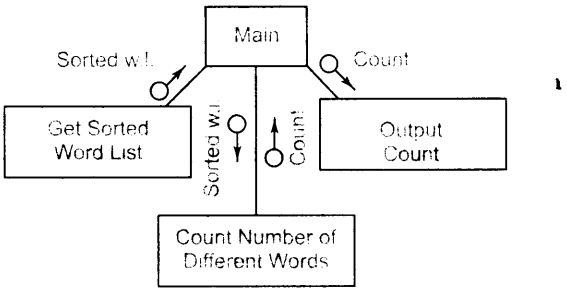

Figure 6.6: First-level factoring.

In this example, there is one input module, which returns the sorted word list to the main module. The output module takes from the main module the value of the count. There is only one central transform in this example, and a module is drawn for that. Note that the data items traveling to and from this transformation module are the same as the data items going in and out of the central transform.

Let us examine the data flow diagram of the ATM. We have already seen that this has two most abstract inputs, two most abstract outputs, and two central transforms. Drawing a module for each of these, we get the structure chart shown in Figure 6.7.

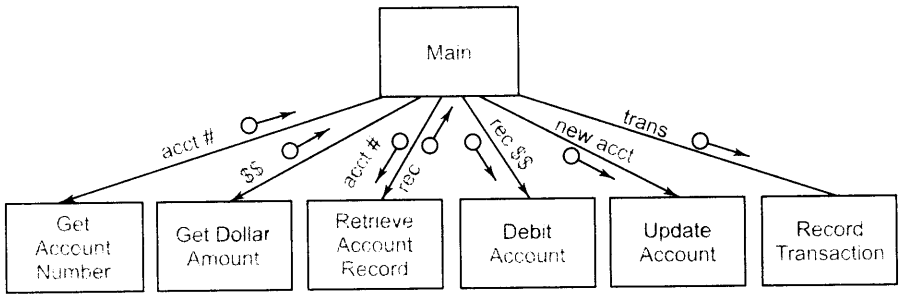

Figure 6.7: First-level factoring for ATM.

As we can see, the first-level factoring is straightforward, after the most abstract input and output data items are identified in the data flow diagram. The main module is the overall control module, which will form the main program or procedure in the implementation of the design. It is a coordinate module that invokes the input modules to get the most abstract data items, passes these to the appropriate transform modules, and delivers the results of the transform modules to other transform modules until the most abstract data items are obtained. These are then passed to the output modules.

6.4.4 Factoring the Input, Output, and Transform Branches

The first-level factoring results in a very high-level structure, where each subordinate module has a lot of processing to do. To simplify these modules, they must be factored into subordinate modules that will distribute the work of a module. Each of the input, output, and transformation modules must be considered for factoring. Let us start with the input modules.

The purpose of an input module, as viewed by the main program, is to produce some data. To factor an input module, the transform in the data flow diagram that produced the data item is now treated as a central transform. The process performed for the first-level factoring is repeated here with this new central transform, with the input module being considered the main module. A subordinate input module is created for each input data stream coming into this new central transform, and a subordinate transform module is created for the new central transform. The new input modules now created can then be factored again, until the physical inputs are reached. Factoring of input modules will usually not yield any output subordinate modules.

The factoring of the input module get-sorted-list in the first-level structure is shown in Figure 6.8. The transform producing the input returned by this module (i.e., the sort

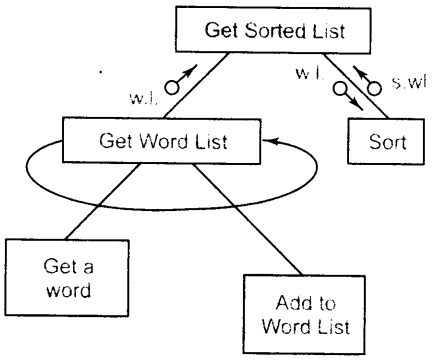

Figure 6.8: Factoring the input module.

transform) is treated as a central transform. Its input is the word list. Thus, in the first factoring we have an input module to get the list and a transform module to sort the list. The input module can be factored further, as the module needs to perform two functions, getting a word and then adding it to the list. Note that the looping arrow is used to show the iteration.

The factoring of the output modules is symmetrical to the factoring of the input modules. For an output module we look at the next transform to be applied to the output to bring it closer to the ultimate desired output. This now becomes the central transform, and an output module is created for each data stream going out of this transform. During the factoring of output modules, there will usually be no input modules. In our example, there is only one transform after the most abstract output, so this factoring need not be done.

If the data flow diagram of the problem is sufficiently detailed, factoring of the input and output modules is straightforward. However, there are no such rules for factoring the central transforms. The goal is to determine subtransforms that will together compose the overall transform and then repeat the process for the newly found transforms, until we reach the atomic modules. Factoring the central transform is essentially an exercise in functional decomposition and will depend on the designers' experience and judgment.

One way to factor a transform module is to treat it as a problem in its own right and start with a data flow diagram for it. The inputs to the data flow diagram are the data coming into the module and the outputs are the data being returned by the module. Each transform in this data flow diagram represents a subtransform of this transform. The central transform can be factored by creating a subordinate transform module for each of the transforms in this data flow diagram. This process can be repeated for the new transform modules that are created, until we reach atomic modules. The factoring of the central transform count-the-number-of-different-words is shown in Figure 6.9.

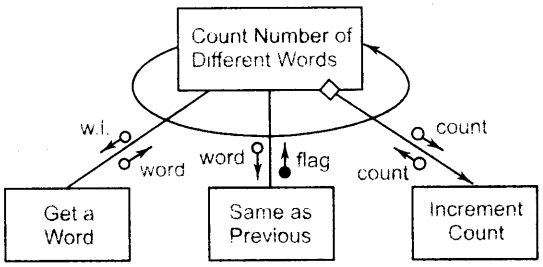

Figure 6.9: Factoring the central transform.

This was a relatively simple transform, and we did not need to draw the data flow diagram. To determine the number of words, we have to get a word repeatedly, determine if it is the same as the previous word (for a sorted list, this checking is sufficient to determine if the word is different from other words), and then count the word if it is different. For each of the three different functions, we have a subordinate module, and we get the structure shown in Figure 6.9.

It should be clear that the structure that is obtained depends a good deal on what are the most abstract inputs and most abstract outputs. And as mentioned earlier, determining the most abstract inputs and outputs requires making a judgment. However, if the judgment is different, though the structure changes, it is not affected dramatically. The net effect is that a bubble that appears as a transform module at one level may appear as a transform module at another level. For example, suppose in the word-counting problem we make a judgment that word-list is another form of the basic input but sorted-word-list is not. If we use word-list as the most abstract input, the net result is that the transform module corresponding to the sort bubble shows up as a transform module one level above. That is, now it is a central transform (i.e., subordinate to the main module) rather than a subordinate to the input module "get-sorted-word-list." So, the SDM has the desired property that it is not very sensitive to some variations in the identification of the most abstract input and most abstract output.

6.4.5 Design Heuristics

The design steps mentioned earlier do not reduce the design process to a series of steps that can be followed blindly. The strategy requires the designer to exercise sound judgment and common sense. The basic objective is to make the program structure reflect the problem as closely as possible. With this in mind the structure obtained by the methodology described earlier should be treated as an initial structure, which may need to be modified. Here we mention some heuristics that can be used to modify the structure, if necessary. Keep in mind that these are merely pointers to help the

designer decide how the structure can be modified. The designer is still the final judge of whether a particular heuristic is useful for a particular application or not.

Module size is often considered an indication of module complexity. In terms of the structure of the system, modules that are very large may not be implementing a single function and can therefore be broken into many modules, each implementing a different function. On the other hand, modules that are too small may not require any additional identity and can be combined with other modules.

However, the decision to split a module or combine different modules should not be based on size alone. Cohesion and coupling of modules should be the primary guiding factors. A module should be split into separate modules only if the cohesion of the original module was low, the resulting modules have a higher degree of cohesion, and the coupling between modules does not increase. Similarly, two or more modules should be combined only if the resulting module has a high degree of cohesion and the coupling of the resulting module is not greater than the coupling of the submodules. Furthermore, a module usually should not be split or combined with another module if it is subordinate to many different modules. As a rule of thumb, the designer should take a hard look at modules that will be larger than about 100 lines of source code or will be less than a couple of lines.

Another parameter that can be considered while "fine-tuning" the structure is the fan-in and fan-out of modules. *Fan-in* of a module is the number of arrows coming in the module, indicating the number of superordinates of a module. *Fan-out* of a module is the number of arrows going out of that module, indicating the number of subordinates of the module. A very high fan-out is not very desirable, as it means that the module has to control and coordinate too many modules and may therefore be too complex. Fan-out can be reduced by creating a subordinate and making many of the current subordinates subordinate to the newly created module. In general the fan-out should not be increased above five or six.

Whenever possible, the fan-in should be maximized. Of course, this should not be obtained at the cost of increasing the coupling or decreasing the cohesion of modules. For example, implementing different functions into a single module, simply to increase the fan-in, is not a good idea. Fan-in can often be increased by separating out common functions from different modules and creating a module to implement that function.

Another important factor that should be considered is the correlation of the scope of effect and scope of control. The scope of effect of a decision (in a module) is the collection of all the modules that contain any processing that is conditional on that decision or whose invocation is dependent on the outcome of the decision. The scope of control of a module is the module itself and all its subordinates (not just the immediate subordinates). The system is usually simpler when the scope of effect of a decision is a subset of the scope of control of the module in which the decision is located. Ideally, the scope of effect should be limited to the modules that are immediate subordinates

of the module in which the decision is located. Violation of this rule of thumb often results in more coupling between modules.

There are some methods that a designer can use to ensure that the scope of effect of a decision is within the scope of control of the module. The decision can be removed from the module and "moved up" in the structure. Alternatively, modules that are in the scope of effect but are not in the scope of control can be moved down the hierarchy so that they fall within the scope of control.

6.4.6 Transaction Analysis

The structured design technique discussed earlier is called *transform analysis*, where most of the transforms in the data flow diagram have a few inputs and a few outputs. There are situations where a transform splits an input stream into many different substreams, with a different sequence of transforms specified for the different substreams. For example, this is the case with systems where there are many different sets of possible actions and the actions to be performed depend on the input command specified. In such situations the transform analysis can be supplemented by *transaction analysis*. and the detailed data flow diagram of the transform splitting the input may look like the DFD shown in Figure 6.10.

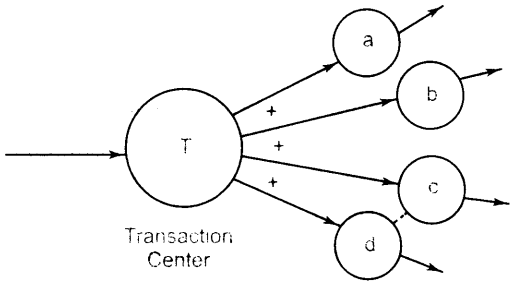

Figure 6.10: DFD for transaction analysis.

The module splitting the input is called the *transaction center*; it need not be a central transform and may occur on either the input branch or the output branch of the data flow diagram of the system. One of the standard ways to convert a data flow diagram of the form shown in Figure 6.10 into a structure chart is to have an input module that gets the analyzed transaction and a dispatch module that invokes the modules for the different transactions. This structure is shown in Figure 6.11.

For smaller systems the analysis and the dispatching can be done in the transaction center module itself, giving rise to a flatter structure. For designing systems that require transaction analysis, start with a data flow diagram, as in transform analysis, and

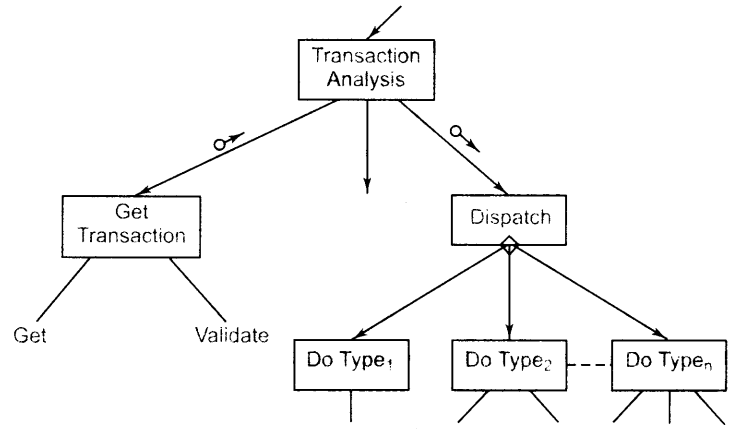

Figure 6.11: Factored transaction center.

identify the transform centers. Factor the data flow diagram, as is done in transform analysis. For the modules corresponding to the transform centers, draw the detailed data flow diagram, which will be of the form shown in Figure 6.11. Choose one of the transaction-centered organizations, either one with a separate dispatch and input module or one with all combined in one module. Specify one subordinate module for each transaction. Temptations to combine many similar transactions into one module should be avoided, as it would result in a logically cohesive module. Then each transaction module should be factored, as is done in transform analysis. There are usually many distinct actions that need to be performed for a transaction; they are often specified in the requirements for each transaction. In such cases one subordinate module to the transaction module should be created for each action. Further factoring of action modules into many detailed action modules may be needed. In many transaction-oriented systems, there is a lot of commonality of actions among the different transactions. This commonality should be exploited by sharing the modules at either the action level or the detailed action level.

6.4.7 Discussion

No design methodology reduces design to a series of steps that can be mechanically executed. All design methodologies are, at best, a set of guidelines that, if applied, will most likely give a design that will satisfy the design objectives. The basic objective is to produce a design that is modular and simple. One way to achieve modularity is to have a design that has highly cohesive modules with low coupling between different modules. In other words, the basic objective of the design activity using a function-oriented approach is to create an architecture, that, if implemented, will satisfy the SRS, and

that contains cohesive modules that have low coupling with others. Structured design methodology is an approach for creating a design that is likely to satisfy this objective. Now that we have studied the methodology, let us see how it actually achieves this goal.

The basic principle behind the SDM, as with most other methodologies, is problem partitioning, in which the problem is partitioned into subproblems that can be solved separately. In SDM, at the very basic level, this is done by partitioning the system into subsystems that deal with input, subsystems that deal with output, and subsystems that deal with data transformation.

The rationale behind this partitioning is that in many systems, particularly data processing systems, a good part of the system code deals with managing the inputs and outputs. The components dealing with inputs have to deal with issues of screens, reading data, formats, errors, exceptions, completeness of information, structure of the information, etc. Similarly, the modules dealing with output have to prepare the output in presentation formats, make charts, produce reports, etc. Hence, for many systems, it is indeed the case that a good part of the software has to deal with inputs and outputs. The actual transformation in the system is frequently not very complex—it is dealing with data and getting it in proper form for performing the transformation or producing the output in the desired form that requires considerable processing.

Structured design methodology clearly separates the system at the very top level into various subsystems, one for managing each major input, one for managing each major output, and one for each major transformation. The modules performing the transformation deal with data at an abstract level, that is, in the form that is most convenient for processing. Due to this, these modules can focus on the conceptual problem of how to perform the transformation without bothering with how to obtain "clean" inputs or how to "present" the output. And these subsystems are quite independent of each other, interacting only through the main module. Hence, this partitioning leads to independent subsystems that do not interact directly, and hence can be designed and developed separately.

This partitioning is at the heart of SDM. In the SDM itself, this partitioning is obtained by starting with a data flow diagram. However, the basic idea of the SDM can be effectively used even if one wants to go directly to the first structure (without going through a DFD).

Besides this central idea, another basic idea behind the SDM is that processing of an input subsystem should be done in a progressive manner, starting from the raw input and progressively applying transformations to eventually reach the most abstract input level (what this input subsystem has to produce). Similar is the case with the structure for the subsystems dealing with outputs. The basic idea here is to separate the different transformations performed on the input before it is in a form ready to be "consumed." And if the SDM is followed carefully, this leads to a "thin and tall" tree as a structure for the input or output subsystem. For example, if an input goes through a series of bubbles in the DFD before it is considered most abstract, the structure for this will be a

tree with each node having two subordinates—one obtaining the input data at its level of abstraction and the other a transform module that is used to transform the data to the next abstract level (which is passed to the superordinate). Similar effect can also be obtained by the main input module having one input module and then a series of transform modules, each performing one transform. In other words, the basic idea in SDM for processing an input is to partition the processing of an input into a series of transforms. As long as this approach is followed, it is not terribly important how the structure for the input subsystem is obtained.

These ideas that the methodology uses to partition the problem into smaller modules lead to a structure in which different modules can be solved separately and the connections between modules are minimized (i.e., the coupling is reduced)—most connections between modules go through some coordinate modules. These ideas of structuring are sound and lead to a modular structure. It is important that these fundamental ideas behind the SDM be kept in mind when using this approach. It may not be so important to follow SDM down to the smallest detail. This is how experienced designers use most methodologies; the detailed steps of the methodology are not necessarily followed, but the philosophy is. Many experienced designers do not start with a detailed DFD when using the SDM; they prefer to work directly with the structure or with a very high-level DFD. But they do use these principles when creating the structure. Such an approach is recommended only when one has some experience with the SDM.

6.5 Verification

The output of the system design phase, like the output of other phases in the development process, should be verified before proceeding with the activities of the next phase. If the design is expressed in some formal notation for which analysis tools are available, then through tools it can be checked for internal consistency (e.g., those modules used by another are defined, the interface of a module is consistent with the way others use it, data usage is consistent with declaration, etc.) If the design is not specified in a formal, executable language, it cannot be processed through tools, and other means for verification have to be used. The most common approach for verification is design review or inspections. We discuss this approach here.

The purpose of design reviews is to ensure that the design satisfies the requirements and is of "good quality." If errors are made during the design process, they will ultimately reflect themselves in the code and the final system. As the cost of removing faults caused by errors that occur during design increases with the delay in detecting the errors, it is best if design errors are detected early, before they manifest themselves in the system. Detecting errors in design is the purpose of design reviews.

The system design review process is similar to the inspection process, in that a group of people get together to discuss the design with the aim of revealing design errors or undesirable properties. The review group must include a member of both the system

design team and the detailed design team, the author of the requirements document, the author responsible for maintaining the design document, and an independent software quality engineer. As with any review, it should be kept in mind that the aim of the meeting is to uncover design errors not to try to fix them; fixing is done later.

The number of ways in which errors can come in a design is limited only by the creativity of the designer. However, there are some forms of errors that are more often observed. Here we mention some of these [52]. Perhaps the most significant design error is omission or misinterpretation of specified requirements. Clearly, if the system designer has misinterpreted or not accounted for some requirement it will be reflected later as a fault in the system. Sometimes, this design error is caused by ambiguities in the requirements.

There are some other quality factors that are not strictly design errors but that have implications on the reliability and maintainability of the system. An example of this is weak modularity (that is, weak cohesion and/or strong coupling). During reviews, elements of design that are not conducive to modification and expansion or elements that fail to conform to design standards should also be considered "errors."

A Sample Checklist: The use of checklists can be extremely useful for any review. The checklist can be used by each member during private study of the design and during the review meeting. For best results the checklist should be tailored to the project at hand, to uncover problem-specific errors. Here we list a few general items that can be used to construct a checklist for a design review [52]:

- Is each of the functional requirements taken into account?
- Are there analyses to demonstrate that performance requirements can be met?
- Are all assumptions explicitly stated, and are they acceptable?
- Are there any limitations or constraints on the design beyond those in the requirements?
- Are external specifications of each module completely specified?
- Have exceptional conditions been handled?
- Are all the data formats consistent with the requirements?
- Are the operator and user interfaces properly addressed?
- Is the design modular, and does it conform to local standards?
- Are the sizes of data structures estimated? Are provisions made to guard against overflow?

6.6 Metrics

We have already seen that the basic purpose of metrics is to provide quantitative data to help monitor the project. Here we discuss some of the metrics that can be extracted from a design and that could be useful for evaluating the design. We do not discuss the standard metrics of effort or defect that are collected (as per the project plan) for project monitoring.

Size is always a product metric of interest, as size is the single most influential factor deciding the cost of the project. As the actual size of the project is known only when the project ends, at early stages the project size is only an estimate. As we saw in Figure 5.1, our ability to estimate size becomes more accurate as development proceeds. Hence, after design, size (and cost) re-estimation are typically done by project management. After design, as all the modules in the system and major data structures are known, the size of the final system can be estimated quite accurately.

For estimating the size, the *total number of modules* is an important metric. This can be easily obtained from the design. By using an average size of a module, from this metric the final size in LOC can be estimated. Alternatively, the size of each module can be estimated, and then the total size of the system will be estimated as the sum of all the estimates. As a module is a small, clearly specified programming unit, estimating the size of a module is relatively easy.

Another metric of interest is complexity, as one of our goals is to strive for simplicity and ease of understanding. A possible use of complexity metrics at design time is to improve the design by reducing the complexity of the modules that have been found to be most complex. This will directly improve the testability and maintainability. If the complexity cannot be reduced because it is inherent in the problem, complexity metrics can be used to highlight the more complex modules. As complex modules are often more error-prone, this feedback can be used by project management to ensure that strict quality assurance is performed on these modules as they evolve. Overall, complexity metrics are of great interest at design time and they can be used to evaluate the quality of design, improve the design, and improve quality assurance of the project. We will describe some of the metrics that have been proposed to quantify the complexity of design.

6.6.1 Network Metrics

Network metrics for design focus on the structure chart (mostly the call graph component of the structure chart) and define some metrics of how "good" the structure or network is in an effort to quantify the complexity of the call graph. As coupling of a module increases if it is called by more modules, a good structure is considered one that has exactly one caller. That is, the call graph structure is simplest if it is a pure tree. The more the structure chart deviates from a tree, the more complex the system.

Deviation of the tree is then defined as the *graph impurity* of the design [153]. Graph impurity can be defined as

$$Graph\ impurity = n - e - 1$$

where n is the number of nodes in the structure chart and e is the number of edges. As in a pure tree the total number of nodes is one more than the number of edges, the graph impurity for a tree is 0. Each time a module has a fan-in of more than one, the graph impurity increases. The major drawback of this approach is that it ignores the common use of some routines like library or support routines. An approach to handle this is not to consider the lowest-level nodes for graph impurity because most often the lowest-level modules are the ones that are used by many different modules, particularly if the structure chart was factored. Library routines are also at the lowest level of the structure chart (even if they have a structure of their own, it does not show in the structure chart of the application using the routine).

Other network metrics have also been defined. For most of these metrics, significant correlations with properties of interest have not been established. Hence, their use is limited to getting some idea about the structure of the design.

6.6.2 Stability Metrics

We know that maintainability of software is a highly desired quality attribute. Maintenance activity is hard and error-prone as changes in one module require changes in other modules to maintain consistency, which require further changes, and so on. It is clearly desirable to minimize this ripple effect of performing a change, which is largely determined by the structure of the software. *Stability* of a design is a metric that tries to quantify the resistance of a design to the potential ripple effects that are caused by changes in modules [151]. The higher the stability of a program design, the better the maintainability of the program. Here we define the stability metric as defined in [151].

At the lowest level, stability is defined for a module. From this, the stability of the whole system design can be obtained. The aim is to define a measure so that the higher the measure the less the ripple effect on other modules that in some way are related to this module. The modules that can be affected by change in a module are the modules that invoke the module or share global data (or files) with the module. Any other module will clearly not be affected by change in a module. The potential ripple effect is defined as the total number of assumptions made by other modules regarding the module being changed. Hence, counting the number of assumptions made by other modules is central to determining the stability of a module.

As at design time only the interfaces of modules are known and not their internals, for calculating design stability only the assumptions made about the interfaces need be considered. The interface of a module consists of all elements through which this module can be affected by other modules, i.e., through which this module can be coupled with

other modules. Hence, it consists of the parameters of the modules and the global data the module uses. Once the interface is identified, the structure of each element of the interface is examined to determine all the *minimal* entities in this element for which assumptions can be made. The minimal entities generally are the constituents of the interface element. For example, a record is broken into its respective fields as a calling module can make assumptions about a particular field.

For each minimal entity at least two categories of assumptions can be made—about the type of the entity and about the value of the entity. (The assumption about the type is typically checked by a compiler if the programming language supports strong typing.) Each minimal entity in the interface is considered as contributing one assumption in each category. A structured type is considered as contributing one more assumption about its structure in addition to the assumptions its minimal elements contribute. The procedure for determining the stability of a module x and the stability of the program can be broken into a series of steps [151]:

Step 1: From the design, analyze the module x and all the modules that call x or share some file or data structure with x, and obtain the following sets.

I_x = {modules that invoke x}

I_x^i = {modules invoked by x}

R_{xy} = {passed parameters returned from x to y, $y \in I_x$}

R_{xy} = {parameters passed from x to y, $y \in I_x^i$}

GR_x = {Global data referenced in x}

GD_x = {Global data defined in x}

Note that determining GR_x and GD_x is not always possible when pointers and indirect referencing are used. In that case, a conservative estimate is to be used. From these, for each global data item i, define the set G_i as

$$G_i = \{x \mid i \in GR_x \cup GD_x\}.$$

The set G_i represents the set of modules where the global data i is either referenced or defined. Where it is not possible to compute G accurately, the worst case should be taken.

Step 2: For each module x, determine the number of assumptions made by a caller module y about elements in R_{xy} (parameters returned from module x to y) through these steps:

1. Initialize assumption count to 0.

2. If i is a structured data element, decompose it into base types, and increment the assumption count by 1; else consider i minimal.

3. Decompose base types, and if they are structured, increment the count by 1.

4. For each minimal entity i, if module y makes some assumption about the value of i, increment the count by 2; else increment by 1.

Let TP_{xy} represent the total number of assumptions made by a module y about parameters in R_{xy}.

Step 3: Determine TP'_{xy}, the total number of assumptions made by a module y called by the module x about elements in R'_{xy} (parameters passed from module x to y). The method for computation is the same as in the previous step.

Step 4: For each data element $i \in GD_x$ (i.e., the global data elements modified by the module x), determine the total number of assumptions made by other modules about i. These will be the modules other than x that use or modify i, i.e., the set of modules to be considered is $\{G_i - \{x\}\}$. The counting method of step 2 is used. Let TG_x be the total number of assumptions made by other modules about the elements in GD_x.

Step 5: For a module x, the design logical ripple effect (DLRE) is defined as:

$$DLRE_x = TG_x = \sum_{i} TP_{xi} + \sum_{j} TP'_{xj}$$

$DLRE_x$ is the total number of assumptions made by other modules that interact with x through either parameters or global data. The design stability (DS) of a module x is then defined as

$$DS_x = 1/(1 + DLRE_x).$$

Step 6: The program design stability (PDS) is computed as

$$PDS = \sum_{x} DLRE_x$$

By following this sequence of steps, the design stability of each module and the overall program can be computed. The stability metric, in a sense, is trying to capture the notion of coupling of a module with other modules. The stability metrics can be used to compare alternative designs—the larger the stability, the more maintainable the program. It can also be used to identify modules that are not very stable and that are highly coupled with other modules with a potential of high ripple effect. Changes to these modules will not be easy, hence a redesign can be considered to enhance the stability. Only a limited validation has been done for this metric. Some validation has been given in [151], showing that if programming practices are followed which are generally recognized as enhancing maintainability, then higher program stability results.

Another stability metric was described in [121]. In this formulation, the effect of a change in a module i on another module j is represented as a probability. For the entire system, the effect of change is captured by the probability of change metrics C. An element $C[i, j]$ of the matrix represents the probability that a change in module i will result in a change in module j. With this matrix the ripple effect of a change in a module can also be easily computed. This can then be used to model the stability of the system. The main problem with this metric is to estimate the elements of the matrix.

6.6.3 Information Flow Metrics

The network metrics of graph impurity had the basis that as the graph impurity increases, the coupling increases. However, it is not a very good approximation for coupling, as coupling of a module increases with the complexity of the interface and the total number of modules a module is coupled with, whether it is the caller or the callee. So, if we want a metric that is better at quantifying coupling between modules, it should handle these. The information flow metrics attempt to define the complexity in terms of the total information flowing through a module.

In one of the earliest work on information flow metrics [84, 85], the complexity of a module is considered as depending on the intramodule complexity and the intermodule complexity. The intramodule complexity is approximated by the size of the module in lines of code (which is actually the estimated size at design time). The intermodule complexity of a module depends on the total information flowing in the module (*inflow*) and the total information flowing out of the module (*outflow*). The inflow of a module is the total number of abstract data elements flowing in the module (i.e., whose values are used by the module), and the outflow is the total number of abstract data elements that are flowing out of the module (i.e., whose values are defined by this module and used by other modules). The module design complexity, D_c, is defined as

$$D_c = \text{[illegible]}$$

The term $(inflow * outflow)$ refers to the total number of combinations of input source and output destination. This term is squared, as the interconnection between the modules is considered a more important factor (compared to the internal complexity) determining the complexity of a module. This is based on the common experience that the modules with more interconnections are harder to test or modify compared to other similar-size modules with fewer interconnections.

The metric defined earlier defines the complexity of a module purely in terms of the total amount of data flowing in and out of the module and the module size. A variant of this was proposed based on the hypothesis that the module complexity depends not only on the information flowing in and out, but also on the number of modules to or