During the meeting, the moderator has to make sure that all the participants contribute effectively, everyone is heard, there is an agreement on the findings of the review, and that the interest level does not drop. A key responsibility is to ensure that during the meeting the focus remains on problem identification and does not drift into problem resolution and that all reviewers remain focused on finding defects in the work product and do not get into finding faults with the author. Overall, orderly and amicable conduct of the meeting is largely the responsibility of the moderator. After the meeting, the moderator has to make sure that all participants are satisfied, the review reports have been filled and follow-up actions taken.

A reviewer is primarily responsible for finding defects. Generally, all members of the group review team are reviewers. The defects are found either through individual review or through the group review meeting. The main issues for a reviewer are:

- Be prepared for group review
- Be objective; focus on issues and not on people
- Concentrate on problems (offer solutions only after the group review)
- If something is not clear do not hesitate to stop progress until it is understood
- When proved wrong, move on

Guidelines for Work Products

All the work products in a project may not undergo group review as that may be prohibitively expensive and may not give commensurate returns. For each project it has to be decided which work products should be inspected, and the size of the inspection team. As the work products of the early part of the life cycle are very critical and defects in them have a multiplier effect in the later stages, it is recommended that early work products like the requirements document, architecture document, and project management plan, be inspected. Regarding team size, though a team size of three to five is often recommended, sometimes where the cost is not justified, an inspection team of just the author and another reviewer may be suitable [97]. This is also sometimes called one-person review.

Though the inspection process is same for any work product, the focus of the inspection is often different for different products. The constitution of the review team and the checklists used in review also depend on the nature of the work product. Some of the guidelines regarding the focus of the review and the composition of the inspection team are given in Table 2.1 [97].

It is often hard to believe that a human-intensive process like the inspections can improve quality and productivity. Due to this and other reasons, inspections are often resisted. One way to find out the utility of inspections is to conduct some experiments

Work product	Focus of Inspection	Participants
Requirement Specification	Requirements meet customer needs Requirements are implementable Omissions, inconsistencies and ambiguities in the requirements	Customer Designers Tester Developer
High Level Design	High-level design implements the requirements The design is implementable Omissions, and other defects in the design	Requirements author Detailed designer Developer
Code	Code implements the design Code is complete and correct Defects in code	Designer Tester Developer
System Test Cases	The set of test cases checks all conditions in the requirements Test cases are executable	Requirements author Tester Project leader
Project Management Plan	Plan is complete Project management plans is implementable Omissions and ambiguities	Project leader SEPG member Another project leader

Table 2.1: Guidelines for inspection of work products.

and evaluate the benefits. Two simple experiments for this purpose are described in [98], along with the data of performing one in a commercial organization.

2.4.3 Software Configuration Management Process

Changes continuously take place in a software project—changes due to the evolution of work products as the project proceeds, changes due to defects (bugs) being found and then fixed, and changes due to requirement changes. All these are reflected as changes in the files containing source, data, or documentation. Configuration management (CM) or *software configuration management (SCM)* is the discipline for systematically controlling the changes that take place during development [13, 12, 91]. The IEEE defines SCM as "the process of identifying and defining the items in the system, controlling the change of these items throughout their life cycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items" [91]. Though all three are types of changes, changes due to product evolution and changes due to bug

fixes can be, in some sense, treated as a natural part of the project itself which have to be dealt with even if the requirements do not change. Requirements changes, on the other hand, have a different dynamic. We will discuss the additional steps that need to be done for requirement changes as a separate process after discussing the CM process.

Software configuration management is a process independent of the development process largely because most development models look at the macro picture and not on changes to individual files. In a way, the development process is brought under the configuration control process, so that changes are allowed in a controlled manner, as shown in Figure 2 for a waterfall-type development process model [147]. Note that SCM directly controls only the products of a process and only indirectly influences the activities producing the product.

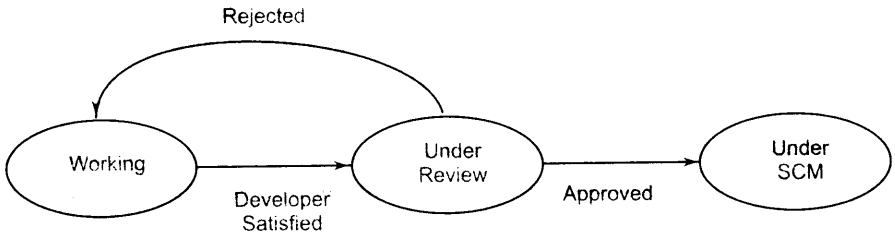

Figure 2.15: Configuration management and development process.

CM is essential to satisfy one of the basic objectives of a project—delivery of a high-quality software product to the client. What is this "software" that is delivered? At the least, it contains the various source or object files that make up the source or object code, scripts to build the working system from these files, and associated documentation. During the course of the project, the files change, leading to different versions. In this situation, how does a program manager ensure that the appropriate versions of sources are combined without missing any source, and the correct versions of the documents, which are consistent with the final source, are sent? This is ensured through proper CM.

CM Functionality

To better understand CM, let us consider some of the functionality that a project requires from the CM process. Though the requirements of a project from its CM process depends on the nature of the project, we discuss here a few functions that are generally needed.

- *Give latest version of a program.* Suppose that a program has to be modified. Clearly, the modification has to be carried out in the latest copy of that program;

otherwise, changes made earlier may be lost. A proper CM process will ensure that latest version of a file can be obtained easily.

- *Undo a change or revert back to a specified version.* A change is made to a program, but later it becomes necessary to undo this change request. Similarly, a change might be made to many programs to implement some change request and later it may be decided that the entire change should be undone. The CM process must allow this to happen smoothly.

- *Prevent unauthorized changes or deletions.* A programmer may decide to change some programs, only to discover that the change has adverse side effects. The CM process ensures that unapproved changes are not permitted.

- *Gather all sources, documents, and other information for the current system.* All sources and related files are needed for releasing the product. The CM process must provide this functionality. All sources and related files of a working system are also sometimes needed for reinstallation.

These are some of the basic needs that a CM process must satisfy. There are other advanced requirements like handling concurrent updates or handle invariance [96].

CM Mechanisms

The main purpose of CM is to provide various mechanisms that can support the functionality needed by a project to handle the types of scenarios discussed above that arise due to changes. The mechanisms commonly used to provide the necessary functionality include the following

- Configuration identification and baselining
- Version control or version management
- Access control

As discussed above, the software being developed is not a monolith. A Software configuration item (SCI), or *item* is a document or an artifact that is explicitly placed under configuration control and that can be regarded as a basic unit for modification. As the project proceeds, hundreds of changes are made to these configuration items. Without periodically combining proper versions of these items into a state of the system, it will become very hard to get the system from the different versions of the many SCIs. For this reason, *baselines* are established. A baseline, once established, captures a logical state of the system, and forms the basis of change thereafter [14]. A baseline also forms a reference point in the development of a system.

A baseline essentially is an arrangement of a set of SCIs [14]. That is, a baseline is a set of SCIs and the relationship between them. For example, a requirements baseline may consist of many requirement SCIs (e.g., each requirement is an SCI) and how these SCIs are related in the requirements baseline (e.g., in which order they appear).

It should be noted that the SCIs being managed by SCM are not independent of one another and there are dependencies between various SCIs. An SCI X is said to *depend* on another SCI Y, if a change to Y might require a change to be made to X for X to remain correct or for the baselines to remain consistent [147]. A change request, though, might require changes be made to some SCIs; the dependency of other SCIs on the ones being changed might require that other SCIs also need to be changed. Clearly, the dependency between the SCIs needs to be properly understood and documented.

Version control is a key issue for CM [14, 12, 147], and many tools are available to help manage the various versions of programs. Without such a mechanism, many of the required CM functions cannot be supported. Version control helps preserve older versions of the programs whenever programs are changed. Commonly used CM systems like SCCS, CVS (www.cvshome.org), VSS (msdn.microsoft.com/vstudio/previous/ssafe), focus heavily on version control.

Most CM systems also provide means for access control. To understand the need for access control, let us understand the life cycle of an SCI. Typically, while an SCI is under development and is not visible to other SCIs, it is considered being in the *working* state. An SCI in the working state is not under SCM and can be changed freely. Once the developer is satisfied that the SCI is stable enough for it to be used by others, the SCI is given for review, and the item enters the state "under review." Once an item is in this state, it is considered as "frozen," and any changes made to a private copy that the developer may have made are not recognized. After a successful review the SCI is entered into a *library*, after which the item is formally under SCM. The basic purpose of this review is to make sure that the item is of satisfactory quality and is needed by others, though the exact nature of review will depend on the nature of the SCI and the actual practice of SCM. For example, the review might entail checking if the item meets its specifications or if it has been properly unit tested. If the item is not approved, the developer may be given the item back and the SCI enters the working state again. This "life cycle" of an item from the SCM perspective, is shown in Figure 2 [147].

Once an SCI is in the library, any modification should be controlled, as others may be using that item. Hence, access to items in the library is controlled. For making an approved change, the SCI is checked out of the library, the change is made, the modification is reviewed and then the SCI is checked back into the library. When a new version is checked in, the old version is not replaced and both old and new versions may exist in the library—often logically with one file being maintained along with information about changes to recreate the older version. This aspect of SCM is sometimes called *library management* and is done with the aid of tools.

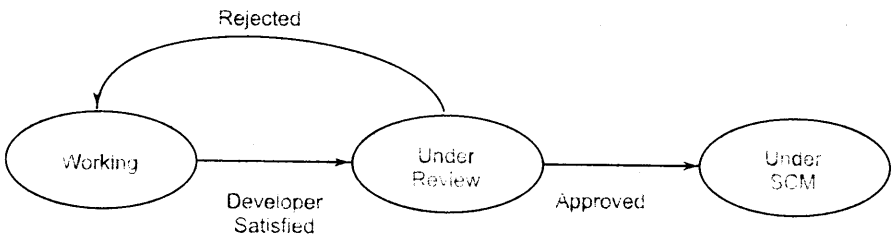

Figure 2.16: SCM life cycle of an item.

CM Process

The CM process defines the set of activities that need to be performed to control change. As with most activities in project management, the first stage in the CM process is planning. Then the process has to be executed, generally by using some tools. Finally, as any CM plan requires some discipline from the project personnel in terms of storing items in proper locations, and making changes properly, monitoring the status of the configuration items and performing CM audits are therefore other activities in the CM process.

Planning for configuration management involves identifying the configuration items and specifying the procedures to be used for controlling and implementing changes to these configuration items. Identifying configuration items is a fundamental activity in any type of CM [12, 89, 147]. Typical examples of configuration items include requirements specifications, design documents, source code, test plans, test scripts, test procedures, test data, standards used in the project (such as coding standards and design standards), the acceptance plan, documents such as the CM plan and the project plan, user documentation such as the user manual, documents such as the training material, contract documents (including support tools such as a compiler or in-house tools), quality records (review records, test records), and CM records (release records, status tracking records). Any customer-supplied products or purchased items that will be part of the delivery (called "included software product") are also configuration items.

As there are typically a lot of items in a project, how they are to be organized is also decided in the planning phase. Typically, the directory structure that will be employed to store the different elements is decided in the plan. To facilitate proper naming of configuration items, the naming conventions for CM items are decided during the CM planning stages. In addition to naming standards, version numbering must be planned. When a configuration item is changed, the old item is not replaced with the new copy;

instead, the old copy is maintained and a new one is created. This approach results in multiple versions of an item, so policies for version number assignment are needed. If a CM tool is being used, then sometimes the tool handles the version numbering. Otherwise, it has to be explicitly done in the project.

The configuration controller or the project manager do the CM planning. It is begun only when the project has been initiated and the operating environment and requirements specifications are clearly documented. The output of this phase is the CM plan.

The configuration controller (CC) is responsible for the implementation of the CM plan. Depending on the size of the system under development, his or her role may be a part-time or full-time job. In certain cases, where there are large teams or where two or more teams/groups are involved in the development of the same or different portions of the software or interfacing systems, it may be necessary to have a configuration control board (CCB). This board includes representatives from each of the teams. A CCB (or a CC) is considered essential for CM [89], and the CM plan must clearly define the roles and responsibilities of the CC/CCB. These duties will also depend on the type of file system and the nature of CM tools being used.

For a CM process to work well, the people in the project have to use it as per the CM plan and follow its policies and procedures. However, people make mistakes. And if by mistake an SCI is misplaced, or access control policies are violated, then the integrity of the product may be lost. To minimize mistakes and catch errors early, regular status checking of SCIs may be done. A configuration audit may also be performed periodically to ensure that the CM system integrity is not being violated. The audit may also check that the changes to SCIs due to change requests (discussed next) have been done properly and that the change requests have been implemented.

In addition to checking the status of the items, the status of change requests (discussed below) must be checked. To accomplish this goal, change requests that have been received since the last CM status monitoring operation are examined. For each change request, the state of the item as mentioned in the change request records is compared with the actual state. Checks may also be done to ensure that all modified items go through their full life cycle (that is, the state diagram) before they are incorporated in the baseline.

2.4.4 Requirements Change Management Process

Requirements change. And changes in requirements can come at any time during the life of a project (or even after that). The farther down in the life cycle the requirements change, the more severe the impact on the project. Instead of wishing that changes will not come, or hoping that somehow the initial requirements will be "so good" that no changes will be required, it is better that a project manager prepare to handle change requests as they come.

Uncontrolled changes to requirements can have a very adverse effect on the cost, schedule, and quality of the project. Requirement changes can account for as much as 40% of the total cost [22]. Due to the potentially large impact of requirement changes on the project, often a separate process is employed to deal with them.

The change management process defines the set of activities that are performed when there are some new requirements or changes to existing requirements (we will call both changes in the requirements). Though we are focusing on requirement changes, any major changes like design changes or major bug fixes to a system in deployment, this process can be used. Here we discuss a requirement change management process which is based on one used in a commercial organization [97]. The change management process has the following steps.

- Log the changes
- Perform impact analysis on the work products
- Estimate impact on effort and schedule
- Review impact with concerned stakeholders
- Rework work products

A change is initiated by a *change request.* A change request log is maintained to keep track of the change requests. Each entry in the log contains a change request number, a brief description of the change, the effect of the change, the status of the change request, and key dates.

The effect of a change request is assessed by performing impact analysis. Impact analysis involves identifying work products and configuration items that need to be changed and evaluating the quantum of change to each; reassessing the projects risks by revisiting the risk management plan; and evaluating the overall implications of the changes for the effort and schedule estimates.

Once a change is reviewed and approved, then it is implemented, i.e., changes to all the items are made. The actual tracking of implementation of a change request may be handled by the configuration management process, which has been discussed above.

One danger of requirement changes is that, even though each change is not large in itself, over the life of the project the cumulative impact of the changes is large. Hence, besides studying the impact of individual changes and tracking them, the cumulative impact of changes must also be monitored. For cumulative changes, the change log is used. To facilitate this analysis, the log is frequently maintained as a spreadsheet.

2.4.5 Process Management Process

A software process is not a static entity—it has to change to improve so that the products produced using the process are of higher quality and are less costly. As we

have seen, improving quality and productivity are fundamental goals of engineering. To achieve these goals the software process must continually be improved, as quality and productivity are determined to a great extent by the process. As stated earlier, improving the quality and productivity of the process is the main objective of the process management process. It should be emphasized that process management is quite different from project management. In process management the focus is on improving the process which in turn improves the general quality and productivity for the products produced using the process. In project management the focus is on executing the current project and ensuring that the objectives of the project are met. The time duration of interest for project management is typically the duration of the project, while process management works on a much larger time scale as each project is viewed as providing a data point for the process.

Process management is an advanced topic beyond the scope of this book. Interested readers are referred to the book by Humphrey [89]. We will only briefly discuss some aspects here.

To improve its software process, an organization needs to first understand the status of the current status and then develop a plan to improve the process. It is generally agreed that changes to a process are best introduced in small increments and that it is not feasible to totally revolutionize a process. The reason is that it takes time to internalize and truly follow any new methods that may be introduced. And only when the new methods are properly implemented will their effects be visible. Introducing too many new methods for the software process will make the task of implementing the change very hard.

If we agree that changes to a process must be introduced in small increments, the next question is out of a large set of possible enhancements to a process, in what order should the improvement activities be undertaken? Or what small change should be introduced first? This depends on the current state of the process. For example, if the process is very primitive there is no point in suggesting sophisticated metrics-based project control as an improvement strategy; incorporating it in a primitive process is not easy. On the other hand, if the process is already using many basic models, such a step might be the right step to further improve the process. Hence, deciding what activities to undertake for process improvement is a function of the current state of the process. Once some process improvement takes place, the process state may change, and a new set of possibilities may emerge. This concept of introducing changes in small increments based on the current state of the process has been captured in the Capability Maturity Model (CMM) framework. The CMM framework provides a general roadmap for process improvement. We give a brief description of the CMM framework here; the reader is referred to [89, 134] for more details. An example of implementation of CMM in an organization can be found in [96].

Software process capability describes the range of expected results that can be achieved by following the process [134]. The process capability of an organization determines

what can be expected from the organization in terms of quality and productivity. The goal of process improvement is to improve the process capability. A *maturity level* is a well-defined evolutionary plateau towards achieving a mature software process [134]. Based on the empirical evidence found by examining the processes of many organizations, the CMM suggests that there are five well-defined maturity levels for a software process. These are initial (level 1), repeatable, defined, managed, and optimizing (level 5). The CMM framework says that as process improvement is best incorporated in small increments, processes go from their current levels to the next higher level when they are improved. Hence, during the course of process improvement, a process moves from level to level until it reaches level 5. This is shown in Figure 2 [134].

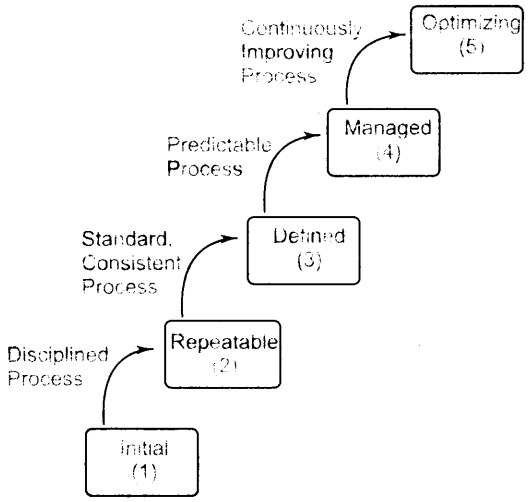

Figure 2.17: Capability Maturity Model.

The CMM provides characteristics of each level, which can be used to assess the current level of the process of an organization. As the movement from one level is to the next level, the characteristics of the levels also suggest the areas in which the process should be improved so that it can move to the next higher level. Essentially, for each level it specifies the areas in which improvement can be absorbed and will bring the maximum benefits. Overall, this provides a roadmap for continually improving the process.

The *initial process* (level 1) is essentially an ad hoc process that has no formalized method for any activity. Basic project controls for ensuring that activities are being done properly, and that the project plan is being adhered to, are missing. In crisis the project plans and development processes are abandoned in favor of a code-and-test type of approach. Success in such organizations depends solely on the quality and capability

of individuals. The process capability is unpredictable as the process constantly changes. Organizations at this level can benefit most by improving project management, quality assurance, and change control.

In a *repeatable process* (level 2), policies for managing a software project and procedures to implement those policies exist. That is, project management is well developed in a process at this level. Some of the characteristics of a process at this level are: project commitments are realistic and based on past experience with similar projects, cost and schedule are tracked and problems resolved when they arise, formal configuration control mechanisms are in place, and software project standards are defined and followed. Essentially, results obtained by this process can be repeated as the project planning and tracking is formal.

At the *defined level* (level 3) the organization has standardized a software process, which is properly documented. A software process group exists in the organization that owns and manages the process. In the process each step is carefully defined with verifiable entry and exit criteria, methodologies for performing the step, and verification mechanisms for the output of the step. In this process both the development and management processes are formal.

At the *managed level* (level 4) quantitative goals exist for process and products. Data is collected from software processes, which is used to build models to characterize the process. Hence, measurement plays an important role in a process at this level. Due to the models built, the organization has a good insight of the process capability and its deficiencies. The results of using such a process can be predicted in quantitative terms.

At the *optimizing level* (level 5), the focus of the organization is on continuous process improvement. Data is collected and routinely analyzed to identify areas that can be strengthened to improve quality or productivity. New technologies and tools are introduced and their effects measured in an effort to improve the performance of the process. Best software engineering and management practices are used throughout the organization.

This CMM framework can be used to improve the process. Improvement requires first assessing the level of the current process. Based on the current level, the areas in which maximum benefits can be derived are known from the framework. For example, for improving a process at level 1 (or for going from level 1 to level 2), project management and the change control activities must be made more formal. The complete CMM framework provides more details about which particular areas need to be strengthened to move up the maturity framework. This is generally done by specifying the key process areas of each maturity level, which in turn, can be used to determine which areas to strengthen to move up. Some of the key process areas of the different levels are shown in Figure 2 [134].

Though the CMM framework specifies the process areas that should be improved to increase the maturity of the process, it does not specify how to bring about the improvement. That is, it is essentially a framework that does not suggest detailed prescriptions for improvement, but guides the process improvement activity along the maturity lev-

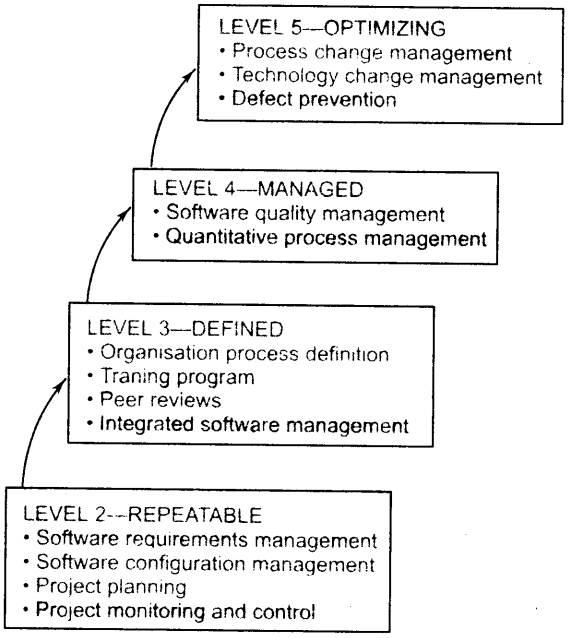

Figure 2.18: Some key process areas.

els such that process improvement is introduced in increments and the improvement activity at any time is clearly focused. Many organizations have successfully used this framework to improve their processes. It is a major driving force for process improvement. A detailed example of how an organization that follows the CMM executes its project can be found in [96].

2.5 Summary

A software process is the set of activities, together with ordering constraints, such that if the activities are performed in accordance to the process, the desired results (of high quality and productivity) will be achieved. A software process consists of different component processes like the development process, the project management process, the configuration management process, and the process management process.

In order to satisfy the basic software engineering objectives, the software process must have some desirable properties. The process must be predictable, that is, following the same process produces more or less similar results. The process must also support testability and maintainability, as testing and maintenance are the activities that consume the most resources. The process should support defect removal and prevention throughout development, as the longer a defect stays, the more costly it is to remove it. And the process must be self-improving.

A process model is a general process specification which has been found useful in many circumstances. In this chapter, we discussed some process models for the development process. The waterfall model is conceptually the simplest model of software development, where the requirement, design, coding, and testing phases are performed in linear progression. There is a defined output after each phase, which is certified before the next phase begins. It has been very widely used, even though it has limitations. The major limitations of this model are that it follows the "all or nothing" approach, is document driven, and does not permit changes.

Another major model is the prototyping model, where a prototype is built before building the final system. The prototype is used to further develop the requirements leading to more stable requirements. Experience with the prototype also results in better design and development of the system.

In the iterative development model, software is developed in iterations, each iteration resulting in a working software system. Iterative development is now widely used as it allows software to be developed and delivered in parts, hence the risks are low. Furthermore, it does not require all requirements to be known in the start, which is usually not possible. The feedback from software of earlier iterations can also be used to improve the software in later iterations.

The timeboxing model is also an iterative model, but the different iterations are of equal time duration. Each iteration is also divided into equal length stages. There is a committed team for each stage of an iteration. The different iterations are then executed in a pipelined manner, with each dedicated team working on its stage but for different iterations. As multiple iterations are concurrently active, this model reduces the average completion time of each iteration and hence is useful in situations where short cycle time is highly desirable.

Besides the development process, there are other component processes of the software process. The project management process consists of three major phases—planning, monitoring and control, and termination analysis. Much of project management revolves around the project plan, which is produced during the planing phase. The monitoring and control phase requires accurate data about the project to reach project management, which uses this data to determine the state of the project and exercise any control it requires. For this purpose, metrics play an essential role in providing the

project management quantified data about the state of development and of the products produced. In the end, a postmortem analysis is done to learn from the experience.

The inspection process is used for finding defects in a work product. In inspections, a work product is closely examined by a group of peer experts, who examine it for defects. There is a structured process, in which first the material to be inspected is examined individually by each of the reviewers. Once the reviewers have logged the defects they have found, a group review meeting is held in which the product is examined line by line. At any point, if a reviewer has an issue, it is discussed and, if needed, recorded as a defect. By the end of the process, all the defects found are recorded, and a summary of defects and effort spent prepared that can be used to judge if the review has been performed properly. Inspections are used heavily in practice and are recognized as an industry best practice.

The software configuration management (CM) process deals with managing the changes that take place during the project. The CM process typically focuses on controlling the changes to the individual CM items, such that latest copy of each item is easily available, and changes can be undone, if needed. Along with the CM process, there is a requirements change management process that focuses on handling changes in requirements. The purpose is to evaluate change requests for their impact, and then implement the changes of approved changes.

The process management process is frequently performed by the software engineering process group. The basic objective of this process is to improve the process such that the quality and productivity improves. A key aspect of this process is to understand the capability of the current process and characterize it so that the expected outcomes are known. The other major activity of this process is to improve the process so that the cost and quality of future products are improved. Frameworks like the CMM help in the process management process.

In the rest of the book we will focus on the important activities that are generally performed during a software development project, and discuss each one of them in more detail. The activities covered include project management as well as development process activities. A knowledge of these activities will enable a person to successfully execute a software project.

Exercises

1. What is the relationship between a process model, process specification, and process for a project?

2. What are the key outputs in a development project that follows the prototyping model? Write a ETVX specification for this process.

3. For the next project, the project manager wants to predict the number of defects she is likely to find in each of the quality control tasks in the process. How will you do this? Assume that all the past data you need is available. Also make suitable assumptions on process predictability.

4. You have to design a process for a project using the iterative development model. If the main objective of this project is high quality, what are the quality control tasks you will have in the process?

5. It is reasonable to assume that if software is easy to test, it will be easy to maintain. Suppose that by putting extra effort in design and coding you increase the cost of these phases by 15%, but you reduce the cost of testing and maintenance by 5%. Will you put in the extra effort?

6. Which of the development process models discussed in this chapter would you follow for the following projects? Give justifications.

7. A simple data processing project.

8. A data entry system for office staff that has never used computers before. The user interface and user-friendliness are extremely important.

9. A new system for comparing fingerprints. It is not clear if the current algorithms can compare fingerprints in the given response time constraints.

10. A spreadsheet system that has some basic features and many other desirable features that use these basic features.

11. A new missile tracking system. It is not known if the current hardware/software technology is mature enough to achieve the goals.

12. An online inventory management system for an automobile industry.

13. A flight control system with extremely high reliability. There are many potential hazards with such a system.

14. A Web site for an online store which always has a list of desired features it wants to add and add them quickly.

15. Suppose that the stages in a time box in the timeboxing model are unequal. What will be the impact on delivery time and resource utilization?

16. A project uses the timeboxing process model with three stages in each time box (as discussed in the chapter), but with unequal length. Suppose the requirement specification stage takes 2 weeks with a team of 2 people, the build stage takes 3 weeks with a team of 4 people, and deployment takes 1 week with a team of 2 people. Design the process for this project that maximizes resource utilization. Assume that each resource can do any task. (Hint: Exploit the fact that the sum of durations of the first and the third stage is equal to the duration of the second stage.)

17. In the timeboxing process model, what will happen if one stage in an iteration takes longer or shorter than its allocated time?

18. Why is the CM process needed in addition to the development process?

19. **What types of effect will the project monitoring activity of the project management process have on the development process? Explain with examples.**

20. Suppose the SEPG undertakes some initiatives to improve the existing process. How will you verify that the initiatives are indeed improving the process?

Chapter 3

Software Requirements Analysis and Specification

The complexity and size of software systems are continuously increasing. As the scale changes to more complex and larger software systems, new problems occur that did not exist in smaller systems (or were of minor significance), which leads to a redefining of priorities of the activities that go into developing software. Software requirements is one such area, to which little importance was attached in the early days of software development, as the emphasis was on coding and design. The tacit assumption was that the developers understood the problem clearly when it was explained to them, generally informally.

As systems grew more complex, it became evident that the goals of the entire system could not be easily comprehended. Hence the need for more rigorous requirements analysis arose. Now, for large software systems, requirements analysis is perhaps the most difficult and intractable activity; it is also very error-prone. Many believe that the software engineering discipline is weakest in this critical area.

Some of the difficulty is due to the scope of this activity. The software project is initiated by the client's needs. In the beginning, these needs are in the minds of various people in the client organization. The requirements analyst has to identify the requirements by talking to these people and understanding their needs. In situations where the software is to automate a currently manual process, many of the needs can be understood by observing the current practice. But no such methods exist for systems for which manual processes do not exist or for "new features," which are frequently added when automating an existing manual process. For such systems, the requirements problem is complicated by the fact that the needs and requirements of the system many not be known even to the user—they have to be visualized and created.

Hence, identifying requirements necessarily involves specifying what some people have in their minds (or what will come to their minds when they visualize it). As the information in their minds is, by nature, not formally stated or organized, the input

to the software requirements specification phase is inherently informal and imprecise, and it is likely to be incomplete. When inputs from multiple people are to be gathered, these inputs are likely to be inconsistent as well.

The requirements phase translates the ideas in the minds of the clients (the input), into a formal document (the output of the requirements phase). Thus, the output of the phase is a set of precisely specified requirements, which hopefully are complete and consistent, while the input has none of these properties. Clearly, the process of specifying requirements cannot be totally formal; any formal translation process producing a formal output must have a precise and unambiguous input. This is why the software requirements activity cannot be fully automated, and any method for identifying requirements can be at best a set of guidelines.

In this chapter we will discuss what requirements are, why requirement specification is important, how requirements are analyzed and specified, how requirements are validated, and some metrics that can be applied to requirements. It ends with a discussion of the SRS of the case studies used in the book.

3.1 Software Requirements

IEEE defines a requirement as "(1) A condition of capability needed by a user to solve a problem or achieve an objective; (2) A condition or a capability that must be met or possessed by a system ... to satisfy a contract, standard, specification, or other formally imposed document." [91]. Note that in software requirements we are dealing with the requirements of the proposed system, that is, the capabilities that the system, which is yet to be developed, should have. It is because we are dealing with specifying a system that does not exist that the problem of requirements becomes complicated. The goal of the requirements activity is to produce the Software Requirements Specification (SRS), that describes *what* the proposed software should do without describing *how* the software will do it.

Producing the SRS is easier said than done. A basic limitation for this is that the user needs keep changing as the environment in which the system is to function changes with time. Even while accepting that some requirement change requests are inevitable, there are still pressing reasons why a thorough job should be done in the requirements phase to produce a high-quality and relatively stable SRS. Let us first look at some of these reasons.

3.1.1 Need for SRS

The origin of most software systems is in the needs of some clients. The software system itself is created by some developers. Finally, the completed system will be used by the end users. Thus, there are three major parties interested in a new system: the client, the developer, and the users. Somehow the requirements for the system that will satisfy

the needs of the clients and the concerns of the users have to be communicated to the developer. The problem is that the client usually does not understand software or the software development process, and the developer often does not understand the client's problem and application area. This causes a communication gap between the parties involved in the development project. A basic purpose of software requirements specification is to bridge this communication gap. SRS is the medium through which the client and user needs are accurately specified to the developer. Hence one of the main advantages is:

- An SRS establishes the basis for agreement between the client and the supplier on what the software product will do.

This basis for agreement is frequently formalized into a legal contract between the client (or the customer) and the developer (the supplier). So, through SRS, the client clearly describes what it expects from the supplier, and the developer clearly understands what capabilities to build in the software. Without such an agreement, it is almost guaranteed that once the development is over, the project will have an unhappy client, which almost always leads to unhappy developers. (The classic situation is, client: "Hey! there is a bug"; Developer: "No, it is a software feature.") Actually, the reality of the situation is that even with such an agreement, the client is frequently not satisfied! A related, but important, advantage is:

- An SRS provides a reference for validation of the final product.

That is, the SRS helps the client determine if the software meets the requirements. Without a proper SRS, there is no way a client can determine if the software being delivered is what was ordered, and there is no way the developer can convince the client that all the requirements have been fulfilled.

Providing the basis of agreement and validation should be strong enough reasons for both the client and the developer to do a thorough and rigorous job of requirement understanding and specification, but there are other very practical and pressing reasons for having a good SRS.

We have seen that the primary forces driving a project are cost, schedule, and quality. Consequently, anything that has a favorable effect on these factors should be considered desirable. Boehm found (as reported in [45]) that in some projects 54% of all the detected errors were detected after coding and unit testing was done and that 45% of these errors actually originated during requirement and early design stages. That is, a total of approximately 25% errors occur during requirement and early design stages. A report on errors in the A-7 project shows that about 80 errors were detected in the requirements document over a period of few months that resulted in change requests [8]. Another report indicates that more than 500 errors were found in an SRS that was earlier approved (as reported in [45]). Similarly, another project reported that more

than 250 errors were found in a previously reviewed SRS by stating the requirements in a structured manner and using tools to analyze the document [47].

It is clear that many errors are made during the requirements phase. And an error in the SRS will most likely manifest itself as an error in the final system implementing the SRS; after all, if the SRS document specifies a wrong system (i.e., one that will not satisfy the client's objectives), then even a correct implementation of the SRS will lead to a system that will not satisfy the client. Clearly, if we want a high-quality end product that has few errors, we must begin with a high-quality SRS. In other words, we can conclude that:

- A high-quality SRS is a prerequisite to high-quality software.

Finally, we show that the quality of SRS has an impact on cost (and schedule) of the project. We have already seen that errors can exist in the SRS. We saw earlier that the cost of fixing an error increases almost exponentially as time progresses. That is, a requirement error, if detected and removed after the system has been developed, can cost up to 100 times more than removing it during the requirements phase itself. Based on the data given in [8], which reported that on average it took about 2.4 person-hours to make a change to the requirements to correct an error (this average was without considering the outliers; with the outliers the average was about 5.0 person-hours), we assume that the average cost of fixing a requirement error in the requirement phase is about 2 person-hours. From this and the relative cost of fixing errors as reported in a multicompany study in [17, 20] (these costs were reflected graphically in Figure 1), the approximate average cost of fixing requirement errors (in person-hours) depending on the phase is shown in Table 3.

Phase	Cost (person-hours)
Requirements	2
Design	5
Coding	15
Acceptance test	50
Operation and maint.	150

Table 3.1: Cost of fixing requirement errors.

Clearly, we can have a tremendous reduction in the project cost by reducing the errors in the SRS. A simplified example will illustrate this point. Using the costs given earlier, by investing an additional 100 person-hours in the requirements phase, an average of about 50 new requirements errors will be detected and removed. (This oversimplification is likely to hold only for the errors detected early in the phase. As the number of remaining errors is reduced, the effort required to detect each error is

likely to increase.) If these errors are not detected in the requirements phase, they will be detected in some later phase. In the A-7 project the following distribution was found [8]: of the requirements errors that remain after the requirements phase, about 65% are detected during design, 2% during coding, 30% during testing, and 3% during operation and maintenance. This type of distribution can be expected in general, as most of the requirements errors are likely to be caught in the design phases and the acceptance test phase, while the rest will be caught in other phases. Assume that these 50 requirement errors, if not removed, would have been detected (and fixed) in the later phases with the distribution given earlier. The total cost of fixing the errors in this case will be

$$32.5 * 5 + 1 * 15 + 15 * 50 + 1.5 * 150 = 1152 \text{ person-hours!}$$

In other words, by investing additional 100 person-hours in the requirements phase in this example, the development cost could be reduced by 1152 person-hours—a net reduction in cost of 1052 person-hours!

This can be viewed in another manner. An error that remains in the requirements will be detected in the later phases with the following probabilities: 0.4 in design, 0.1 in coding and unit testing, 0.4 in acceptance testing, 0.1 during operation and maintenance. The cost of fixing a requirement error in these phases was given earlier. Hence, the expected cost of fixing a requirement error that is not removed during the requirements phase is $0.4 * 5 + 0.1 * 15 + 0.4 * 50 + 0.1 * 150 = 38.5$ person-hours. Therefore, if the expected effort required to detect and remove an error during the requirements phase is less than this, it makes economic sense to spend the extra effort in the requirements phase and remove the error.

This is not the complete story. We know that requirements frequently change. As mentioned earlier, though some of the changes are inevitable due to the changing needs and perceptions, many changes come as the requirements were not properly analyzed and not enough effort was expended to validate the requirements. With a high-quality SRS, requirement changes that come about due to improperly analyzed requirements should be reduced considerably. And as changes tend to escalate the cost and throw the project schedule haywire, a reduction in the requirement change traffic will reduce the project cost, in addition to improving its chances of finishing on schedule.

Let us illustrate this with another simplified example. It is estimated that 20% to 40% of the total development effort in a software project is due to rework, much of which occurs due to change in requirements [22]. The cost of the requirement phase is typically about 6% of the total project cost, according to the COCOMO model [20] (the model is discussed in more detail in Chapter 4). Consider a project whose total effort requirement is estimated to be 50 person-months. For this project, the requirements phase consumes about 3 person-months. If by spending an additional 33% effort in the requirements phase we reduce the total requirement change requests by 33%, then the total effort due to rework (assuming all rework is due to requirement change requests)

will reduce from 10 to 20 person-months to 6 to 12 person-months, resulting in a total saving of 5 to 11 person-months, i.e., a saving of 10% to 22% of the total cost! From these, we can conclude that

- A high quality SRS reduces the development cost.

Hence, the quality of the SRS impacts customer (and developer) satisfaction, system validation, quality of the final software, and the software development cost. The critical role the SRS plays in a software development project should be evident from these.

3.1.2 Requirement Process

The requirement process is the sequence of activities that need to be performed in the requirements phase and that culminate in producing a high-quality document containing the software requirements specification (SRS). The requirements process typically consists of three basic tasks: problem or requirement analysis, requirement specification, and requirements validation.

Problem analysis often starts with a high-level "problem statement." During analysis the problem domain and the environment are modeled in an effort to understand the system behavior, constraints on the system, its inputs and outputs, etc. The basic purpose of this activity is to obtain a thorough understanding of what the software needs to provide. The understanding obtained by problem analysis forms the basis of *requirements specification*, in which the focus is on clearly specifying the requirements in a document. Issues such as representation, specification languages, and tools, are addressed during this activity. As analysis produces large amounts of information and knowledge with possible redundancies; properly organizing and describing the requirements is an important goal of this activity. *Requirements validation* focuses on ensuring that what has been specified in the SRS are indeed all the requirements of the software and making sure that the SRS is of good quality. The requirements process terminates with the production of the validated SRS.

Though it seems that the requirements process is a linear sequence of these three activities, in reality it is not so for anything other than trivial systems. In most real systems, there is considerable overlap and feedback between these activities. So, some parts of the system are analyzed and then specified while the analysis of the other parts is going on. Furthermore, if the validation activities reveal problems in the SRS, it is likely to lead to further analysis and specification. However, in general, for a part of the system, analysis precedes specification and specification precedes validation. This requirement process is shown in Figure 3.

As shown in the figure, from the specification activity we may go back to the analysis activity. This happens as frequently some parts of the problem are analyzed and then specified before other parts are analyzed and specified. Furthermore, the process of specification frequently shows shortcomings in the knowledge of the problem, thereby necessitating further analysis. Once the specification is "complete" it goes through

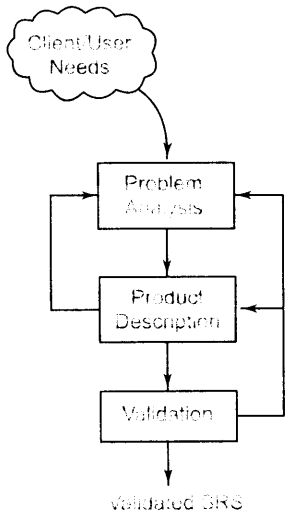

Figure 3.1: The requirement process

the validation activity. This activity may reveal problems in the specifications itself, which requires going back to the specification step, or may reveal shortcomings in the understanding of the problem, which requires going back to the analysis activity.

During requirements analysis the focus is on understanding the system and its requirements. For a complex system, this is a hard task, and the time-tested method of "divide-and-conquer," i.e., decomposing the problem or system into smaller parts and then understanding the parts and their relationships, is inevitably applied to manage the complexity. Also, for managing the complexity and the large volume of information that becomes available during analysis, various structures are used during analysis to represent the information to help view the system as a series of abstractions. Examples of these structures are data flow diagrams and object diagrams (more about these in the next section). For a portion of the system, analysis typically precedes specification. Once the analysis is complete and the structures built, the system part has to be specified.

The transition from analysis to specification, though, seems as if it should be simple; this is not so. In fact, this transition can be quite hard. The reason for this transition being hard is the different objectives of the two activities. In specification, we have to specify only what the software is supposed to do, i.e., focus on the external behavior of the system. In order to identify all the external behaviors, the *structure* of the problem and its various components need to be clearly understood besides understanding its inputs and outputs. However, the structure itself may not be of much use in specification,

as its focus is exclusively on the external behavior or the eventual system, not the internal structure of the problem domain. Due to this, one should not expect that once the analysis is done, specification will be straightforward. Furthermore, many "outputs" of the analysis are not used directly in the SRS. This does not mean that these outputs are not useful—they are essential in modeling the problem that leads to the proper understanding of the requirements, which is a prerequisite to specification. Hence, the use of the analysis activity and structures that it built may be indirect, aiding understanding rather than directly aiding specification.

It is worth noting that some similarities exist in the analysis activity and the design activity. As pointed out by Davis [45], the basic problem during software design is the same—managing the complexity. The approach used there is similar—decomposition and building structures to represent the system as a series of abstractions. Due to this similarity, the approaches used for problem analysis and design are frequently similar (e.g., data flow diagrams and object diagrams are used in analysis as well as design). However, although the approaches are similar, the objective of the two activities is completely different. Whereas analysis deals with the problem domain, with the basic objective of understanding the problem, design deals with the solution domain with the basic objective of optimizing the design [45]. Because to this, the application of similar approaches produces different structures during analysis and design. It is sometimes mistakenly believed that the structures produced during analysis will and should be carried through in design. This comes from a basic misunderstanding about the objectives of the two activities. Though some of the structures may eventually get used in design, this should be done only if the analysis structures are consistent with the design objective.

Finally, there is the issue of the level of detail that the requirement process should aim to uncover and specify. This is also an issue that cannot be easily resolved and that depends on the objective of the requirement specification phase. If the objective is to define the overall broad needs of the system, the requirements can be very abstractly stated. Generally, the purpose of such requirements is to perform some feasibility analysis or use the requirements for competitive bidding. At the lower level are the requirements where all the behavior and external interfaces of the software are clearly specified. Such requirements are clearly very detailed and are suitable for software development.

An example can illustrate this point. Suppose a car manufacturer wants to have an inventory control system. At an abstract level, the requirements of the inventory control system could be stated in terms of the number of parts it has to track, level of concurrency it has to support, whether it will be on-line or batch processing, what types of information and reports it will provide (e.g., status of each item on demand, purchase orders for items that are low in inventory, consumption patterns), etc. Requirements specification at this level of abstraction can be used to estimate the costs and perform a cost-benefit analysis. It can also be used to invite tenders from various developers. However, such a requirements specification is of little use for a developer given the

contract to develop the software. That developer needs to know the exact format of the reports, all the queries that can be performed and their structure, total number of terminals the system has to support, the structure of the major databases that will exist, etc. These are all specifying the external behavior of the software, but when viewed from the higher level of abstraction they can be considered as specifying how the abstract requirements should be implemented (e.g., the details of a report can be viewed as defining how the basic objective of providing information is satisfied).

As should be clear, the abstract requirement level is not suitable for software development. Hence, we will focus mostly on the requirements at the lower level in which all the details about external behavior that are needed for the developer to build a software system are specified. That is, we view the SRS as providing all the detailed information needed by a software developer for properly developing the system. However, it is worth pointing out that even when obtaining the detailed requirements is the objective, abstract requirements can still play a useful role for complex systems. As the problem analysis starts with some initial description of the system's behavior or needs, the abstract requirements can play this role (besides being used for competitive bidding and/or feasibility analysis). In other words, specifying the requirements at an abstract level is likely to be useful in producing the SRS that contains the detailed requirements of the system.

The following sections will be devoted to providing a more detailed description of the activities in the three major activities in the requirements phase: analysis, specification, and verification.

3.2 Problem Analysis

The basic aim of problem analysis is to obtain a clear understanding of the needs of the clients and the users, what exactly is desired from the software, and what the constraints on the solution are [45]. Frequently the client and the users do not understand or know all their needs, because the potential of the new system is often not fully appreciated. The analysts have to ensure that the real needs of the clients and the users are uncovered, even if they don't know them clearly. That is, the analysts are not just collecting and organizing information about the client's organization and its processes, but they also act as *consultants* who play an *active* role of helping the clients and users identify their needs.

The basic principle used in analysis is the same as in any complex task: divide and conquer. That is, *partition* the problem into subproblems and then try to understand each subproblem and its relationship to other subproblems in an effort to understand the total problem.

The concepts of *state* and *projection* can sometimes also be used effectively in the partitioning process. A state of a system represents some conditions about the system.

Frequently, when using state, a system is first viewed as operating in one of the several possible states, and then a detailed analysis is performed for each state. This approach is sometimes used in real-time software or process-control software.

In *projection*, a system is defined from multiple points of view [152]. While using projection, different viewpoints of the system are defined and the system is then analyzed from these different perspectives. The different "projections" obtained are combined to form the analysis for the complete system. Analyzing the system from the different perspectives is often easier, as it limits and focuses the scope of the study.

In the remainder of this section we will discuss a few methods for problem analysis. As the goal of analysis is to understand the problem domain, an analyst must be familiar with different methods of analysis and pick the approach that he feels is best suited to the problem at hand.

3.2.1 Informal Approach

The informal approach to analysis is one where no defined methodology is used. Like in any approach, the information about the system is obtained by interaction with the client, end users, questionnaires, study of existing documents, brainstorming, etc. However, with this approach no formal model is built of the system. The problem and the system model are essentially built in the minds of the analysts (or the analysts may use some informal notation for this purpose) and are directly translated from the minds of the analysts to the SRS.

Frequently, with such an approach, the analyst will have a series of meetings with the clients and end users. In the early meetings, the clients and end users will explain to the analyst about their work, their environment, and their needs as they perceive them. Any documents describing the work or the organization may be given, along with outputs of the existing methods of performing the tasks. In these early meetings, the analyst is basically the listener, absorbing the information provided. Once the analyst understands the system to some extent, he uses the next few meetings to seek clarifications of the parts he does not understand. He may document the information in some manner (he may even build a model if he wishes), and he may do some brainstorming or thinking about what the system should do. In the final few meetings, the analyst essentially explains to the client what he understands the system should do and uses the meetings as a means of verifying if what he proposes the system should do is indeed consistent with the objectives of the clients. An initial draft of the SRS may be used in the final meetings.

The informal approach to analysis is used widely and can be quite useful. The reason for its usefulness is that conceptual modeling-based approaches frequently do not model all aspects of the problem and are not always well suited for all the problems. Besides, as the SRS is to be validated and the feedback from the validation activity may require further analysis or specification (see Figure 3), choosing an informal approach to

analysis is not very risky—the errors that may be introduced are not necessarily going to slip by the requirements phase. Hence such approaches may be the most practical approach to analysis in some situations.

3.2.2 Data Flow Modeling

Data-flow based modeling, often referred to as the structured analysis technique [48, 130], uses function-based decomposition while modeling the problem. It focuses on the functions performed in the problem domain and the data consumed and produced by these functions. It is a top-down refinement approach, which was originally called *structured analysis and specification*, and was proposed for producing the specifications. However, we will limit our attention to the analysis aspect of the approach. Before we describe the approach, let us the describe the data flow diagram and data dictionary on which the technique relies heavily.

Data Flow Diagrams and Data Dictionary

Data flow diagrams (also called *data flow graphs*) are commonly used during problem analysis. Data flow diagrams (DFDs) are quite general and are not limited to problem analysis for software requirements specification. They were in use long before the software engineering discipline began. DFDs are very useful in understanding a system and can be effectively used during analysis.

A DFD shows the flow of data through a system. It views a system as a function that transforms the inputs into desired outputs. Any complex system will not perform this transformation in a "single step," and a data will typically undergo a series of transformations before it becomes the output. The DFD aims to capture the transformations that take place within a system to the input data so that eventually the output data is produced. The agent that performs the transformation of data from one state to another is called a *process* (or a *bubble*). So, a DFD shows the movement of data through the different transformations or processes in the system. The processes are shown by named circles and data flows are represented by named arrows entering or leaving the bubbles. A rectangle represents a source or sink and is a net originator or consumer of data. A source or a sink is typically outside the main system of study. An example of a DFD for a system that pays workers is shown in Figure 3.

In this DFD there is one basic input data flow, the weekly timesheet, which originates from the source *worker*. The basic output is the paycheck, the sink for which is also the worker. In this system, first the employee's record is retrieved, using the employee ID, which is contained in the timesheet. From the employee record, the rate of payment and overtime are obtained. These rates and the regular and overtime hours (from the timesheet) are used to compute the pay. After the total pay is determined, taxes are deducted. To compute the tax deduction, information from the tax-rate file is used. The amount of tax deducted is recorded in the employee and company records. Finally,

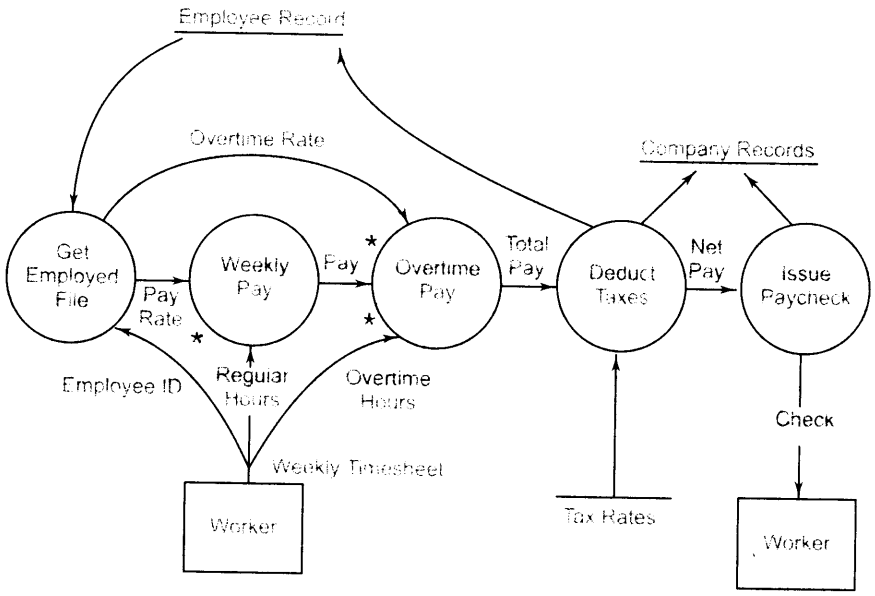

Figure 3.2: DFD of a system that pays workers.

the paycheck is issued for the net pay. The amount paid is also recorded in company records.

Some conventions used in drawing this DFD should be explained. All external files such as employee record, company record, and tax rates are shown as a labeled straight line. The need for multiple data flows by a process is represented by a '*' between the data flows. This symbol represents the AND relationship. For example, if there is a '*' between the two input data flows A and B for a process, it means that A AND B are needed for the process. In the DFD, for the process "weekly pay" the data flow "hours" and "pay rate" both are needed, as shown in the DFD. Similarly, the OR relationship is represented by a '+' between the data flows.

This DFD is an abstract description of the system for handling payment. It does not matter if the system is automated or manual. This diagram could very well be for a manual system where the computations are all done with calculators, and the records are physical folders and ledgers. The details and minor data paths are not represented in this DFD. For example, what happens if there are errors in the weekly timesheet is not shown in this DFD. This is done to avoid getting bogged down with details while constructing a DFD for the overall system. If more details are desired, the DFD can be further refined.

It should be pointed out that a DFD is *not* a flowchart. A DFD represents the flow of data, while a flowchart shows the flow of control. A DFD does not represent procedural information. So, while drawing a DFD, one *must not* get involved in procedural details, and procedural thinking must be consciously avoided. For example, considerations of loops and decisions must be ignored. In drawing the DFD, the designer has to specify the major transforms in the path of the data flowing from the input to output. *How* those transforms are performed is *not* an issue while drawing the data flow graph.

There are no detailed procedures that can be used to draw a DFD for a given problem. Only some directions can be provided. One way to construct a DFD is to start by identifying the major inputs and outputs. Minor inputs and outputs (like error messages) should be ignored at first. Then starting from the inputs, work toward the outputs, identifying the major transforms in the way. An alternative is to work down from the outputs toward the inputs. (Remember that it is important that procedural information like loops and decisions not be shown in the DFD, and the designer should not worry about such issues while drawing the DFD.) Following are some suggestions for constructing a data flow graph [154, 48]:

- Work your way consistently from the inputs to the outputs, or vice versa. If you get stuck, reverse direction. Start with a high-level data flow graph with few major transforms describing the entire transformation from the inputs to outputs and then refine each transform with more detailed transformations.
- Never try to show control logic. If you find yourself thinking in terms of loops and decisions, it is time to stop and start again.
- Label each arrow with proper data elements. Inputs and outputs of each transform should be carefully specified.
- Make use of * and + operations and show sufficient detail in the data flow graph.
- Try drawing alternate data flow graphs before settling on one.

Many systems are too large for a single DFD to describe the data processing clearly. It is necessary that some decomposition and abstraction mechanism be used for such systems. DFDs can be hierarchically organized, which helps in progressively partitioning and analyzing large systems. Such DFDs together are called a *leveled DFD set* [48].

A leveled DFD set has a starting DFD, which is a very abstract representation of the system, identifying the major inputs and outputs and the major processes in the system. Then each process is refined and a DFD is drawn for the process. In other words, a bubble in a DFD is expanded into a DFD during refinement. For the hierarchy to be consistent, it is important that the net inputs and outputs of a DFD for a process are the same as the inputs and outputs of the process in the higher-level DFD. This refinement stops if each bubble is considered to be "atomic," in that each bubble can be easily specified or understood. It should be pointed out that during refinement,

```
weekly timesheet =
        Employee_name +
        Employee_Id +
        [Regular_hours + Overtime_hours] *

pay_rate =
        [Hourly | daily | weekly] +
        Dollar_amount

Employee_name =
        Last + First + Middle_initial

Employee_Id =
        digit + digit + digit + digit
```

though the net input and output are preserved, a refinement of the data might also occur. That is, a unit of data may be broken into its components for processing when the detailed DFD for a process is being drawn. So, as the processes are decomposed, data decomposition also occurs.

In a DFD, data flows are identified by unique names. These names are chosen so that they convey some meaning about what the data is. However, the precise structure of data flows is not specified in a DFD. The *data dictionary* is a repository of various data flows defined in a DFD. The associated data dictionary states precisely the structure of each data flow in the DFD. Components in the structure of a data flow may also be specified in the data dictionary, as well as the structure of files shown in the DFD. To define the data structure, different notations are used. These are similar to the notations for regular expressions (discussed later in this chapter). Essentially, besides sequence or composition (represented by '+') selection and iteration are included. Selection (represented by vertical bar '|') means one OR the other, and repetition (represented by '*') means one or more occurrences. In the DFD shown earlier, data flows for weekly timesheet are used. The data dictionary for this DFD is shown in Figure 3.

Most of the data flows in the DFD are specified here. Some of the more obvious ones are not shown here. The data dictionary entry for weekly timesheet specifies that this data flow is composed of three basic data entities—the employee name, employee ID, and many occurrences of the two-tuple consisting of regular hours and overtime hours.

The last entity represents the daily working hours of the worker. The data dictionary also contains entries for specifying the different elements of a data flow.

Once we have constructed a DFD and its associated data dictionary, we have to somehow verify that they are "correct." There can be no formal verification of a DFD, because what the DFD is modeling is not formally specified anywhere against which verification can be done. Human processes and rules of thumb must be used for verification. In addition to the walkthrough with the client, the analyst should look for common errors. Some common errors are [48]:

- Unlabeled data flows
- Missing data flows, information required by a process is not available
- Extraneous data flows, some information is not being used in the process
- Consistency not maintained during refinement
- Missing processes
- Contains some control information

Perhaps the most common error is unlabeled data flow. If an analyst cannot label the data flow, it is likely that he does not understand the purpose and structure of that data flow. A good test for this type of error is to see that the entries in the data dictionary are precise for all data flows.

To check if there are any missing data flows, for each process in the DFD the analyst should ask, "Can the process build the outputs shown from the given inputs?" Similarly, to check for redundant data flows, the following question should be asked: "Are all the input data flows required in the computation of the outputs?"

In a leveled set of DFDs it is important that consistency be maintained. Consistency can easily be lost if new data flows are added to the DFD during modification. If such changes are made, appropriate changes should be made in the parent or the child DFD. That is, if a new data flow is added in a lower-level DFD, it should also be reflected in the higher-level DFDs. Similarly, if a data flow is added in a higher-level DFD, the DFDs for the processes affected by the change should also be appropriately modified.

The DFDs should be carefully scrutinized to make sure that all the processes in the physical environment are shown in the DFD. None of the data flows should actually carry control information. A data flow without any structure or composition is a potential candidate for control information.

The Structured Analysis Method

Now let us return to the structured analysis method. The basic system view of this approach is that each system can be viewed as a transformation function operating

within an environment that takes some inputs from the environment and produces some outputs for the environment. And as the overall transformation function of the entire system may be too complex to comprehend as a single function, the function should be partitioned into subfunctions that together form the overall function. The subfunctions can be further partitioned and the process repeated until we reach a stage where each function can be comprehended easily. And the basic approach used to uncover the functions being performed in the system (or the functions that are part of the overall system function) is to track the data as it flows through the system—from the input to the output. It is believed that in any complex system the data transformation from the input to the output will not occur in a single step; rather the data will be transformed from the input to the output in a series of transformations starting from the input and culminating in the desired output. By understanding the "states" the data is in as it goes through the transformation series, the functions in the system can be identified; each transformation of the data in the transformation series is performed by a transformation function. Hence, by tracking as the data flows through the system, the various functions being performed by a system can be identified. As this approach can be modeled easily by data flow diagrams, DFDs are used heavily in this method.

The first step in this method is to study the "physical environment." During this, a DFD of the current nonautomated (or partially automated) system is drawn, showing the input and output data flows of the system, how the data flows through the system, and what processes are operating on the data. This DFD might contain specific names for data flows and processes, as used in the physical environment. For example, names of departments, persons, local procedures, and organizational files can occur in the DFD for the physical environment. While drawing the DFD for the physical environment, an analyst has to interact with the users to determine the overall process from the point of view of the data. This step is considered complete when the entire physical data flow diagram has been described and the user has accepted it as a true representation of the operation of the current system. The step may start with a *context diagram* in which the entire system is treated as a single process and all its inputs, outputs, sinks, and sources are identified and shown.

The basic purpose of analyzing the current system is to obtain a logical DFD for the system, where each data flow and each process is a logical entity or operation, rather than an actual name. Drawing a DFD for the physical system is only to provide a reasonable starting point for drawing the logical DFD. Hence, the next step in the analysis is to draw the logical equivalents of the DFD for the physical system. During this step, the DFD of the physical environment is taken and all specific physical data flows are represented by their logical equivalents (for example, file 12.3.2 may be replaced by the employee salary file). Similarly, the bubbles for physical processes are replaced with logical processes. For example, a bubble named "To_John's_office" in the physical system might be replaced by "issue checks" in the logical equivalent. Bubbles that do

not transform the data in any form are deleted from the DFD. This phase also ends when the DFD has been verified by the user.

In the first two steps, the current system is modeled. The next step is to develop a logical model of the new system after the changes have been incorporated, and a DFD is drawn to show how data will flow in the new system. During this step the analyst works in the logical mode, specifying only what needs to be done, not how it will be accomplished. No separation between the automated and nonautomated processes is made.

No general rules are provided for constructing the DFD for the new system. The new system still does not exist; it has to be invented. Consequently, what will be the data flows and major processes in this new system must be determined by the analyst, based on his experience and vision of the new system. No rules can be provided for this decision. However, before this can be done, the boundaries of change have to be identified in the logical DFD for the existing system. This DFD models the entire system, and only parts of it may be modified in the new system. Based on the goals of the clients and a clear concept about what the client wants to change, the boundaries of change have to be established in the logical DFD. The DFD for the new system will replace only that part of the existing DFD within this boundary. The inputs and outputs of the new DFD should be the same as the inputs and outputs for the DFD within the boundary.

The next step is to establish the man-machine boundary by specifying what will be automated and what will remain manual in the DFD for the new system. Note that even though some processes are not automated, they could be quite different from the processes in the original system, as even the manual operations may be performed differently in the new system. Often there is not just one option for the man-machine boundary. Different possibilities may exist depending on what is automated and the degree of automation. The analyst should explore and present the different possibilities.

The next two steps are evaluating the different options and then packaging or presenting the specifications.

For drawing a DFD, a top-down approach is suggested in the structured analysis method. In the structured analysis method, a DFD is constructed from scratch when the DFD for the physical system is being drawn and when the DFD for the new system is being drawn. The second step largely performs transformations on the physical DFD. Drawing a DFD starts with a top-level DFD called the context diagram, which lists all the major inputs and outputs for the system. This diagram is then refined into a description of the different parts of the DFD showing more details. This results in a leveled set of DFDs. As pointed out earlier, during this refinement, the analyst has to make sure consistency is maintained and that net input and output are preserved during refinement.

Clearly, the structured analysis provides methods for organizing and representing information about systems. It also provides guidelines for checking the accuracy of the

information. Hence, for understanding and analyzing an existing system, this method provides useful tools. However, most of the guidelines given in the structured analysis are only applicable in the first two steps, when the DFD for a current system is to be constructed. For analyzing the target system and constructing the DFD or the data dictionary for the new system to be built (done in step three), this technique does not provide much guidance. Of course, the study and understanding of the existing system will help the analyst in this job, but there is no direct help from the method of structured analysis.

An Example

A restaurant owner feels that some amount of automation will help make her business more efficient. She also believes that an automated system might be an added attraction for the customers. So she wants to automate the operation of her restaurant as much as possible. Here we will perform the analysis for this problem. Details regarding interviews, questionnaires, or how the information was extracted are not described. First let us identify the different parties involved.

Client: The restaurant owner
Potential Users: Waiters, cash register operator

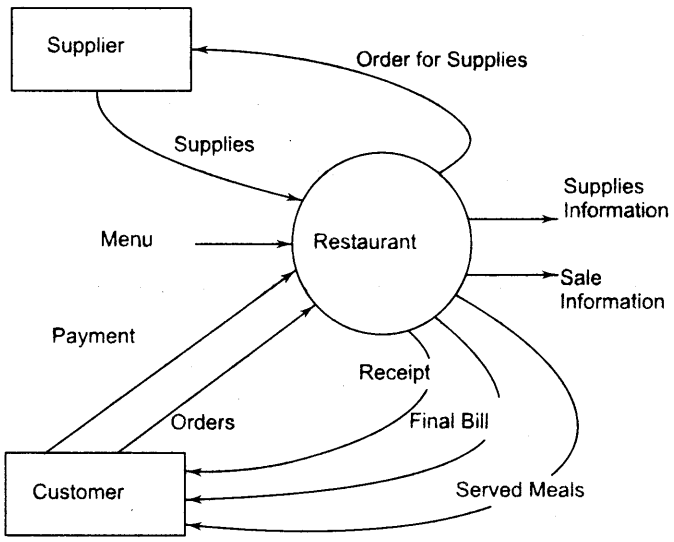

Figure 3.4: Context diagram for the restaurant.

The context diagram for the restaurant is shown in Figure 3. The inputs and outputs of the restaurant are shown in this diagram. However, no details about the functioning of the restaurant are given here. Using this as a starting point, a logical DFD of the physical system is given in Figure 3 (the physical DFD was avoided for this, as the logical DFD is similar to the physical and there were no special names for the data or the processes in the physical system). Observing the operation of the restaurant and interviewing the owner were the basic means of collecting raw information for this DFD.

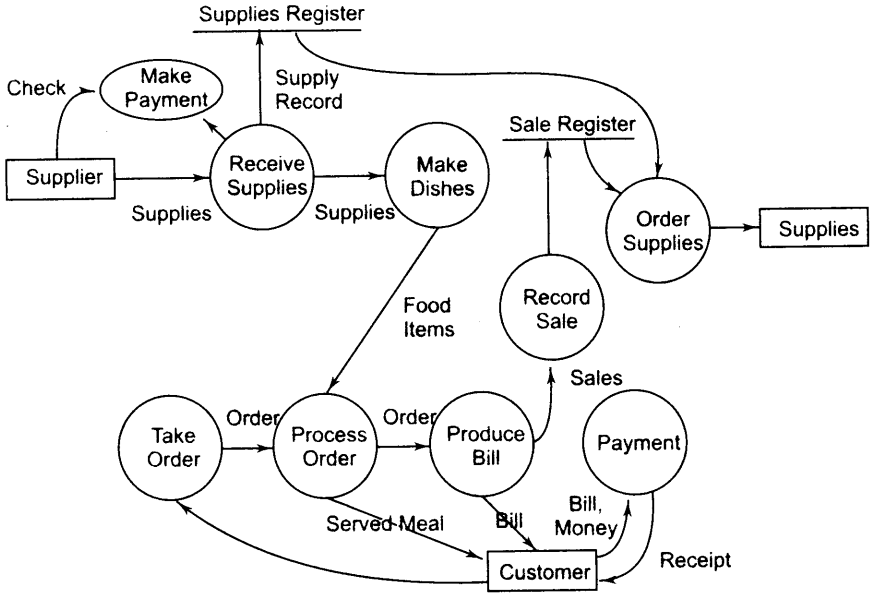

Figure 3.5: DFD for the existing restaurant system.

Now we must draw a DFD that models the new system to be built. After many meetings and discussions with the restaurant owner, the following goals for the new system were established:

- Automate much of the order processing and billing.
- Automate accounting.
- Make supply ordering more accurate so that leftovers at the end of the day are minimized and the orders that cannot be satisfied due to nonavailability are also minimized. This was being done without a careful analysis of sales.
- The owner also suspects that the staff might be stealing/eating some food/supplies. She wants the new system to help detect and reduce this.

- The owner would also like to have statistics about sales of different items.

With these goals, we can define the boundaries for change in the DFD. It is clear that the new system will affect most aspects of the previous system, with the exception of making dishes. So, except for that process, the remaining parts of the old system all fall within our boundary of change. The DFD for the new system is shown in Figure 3. Note that although taking orders might remain manual in the new system, the process might change, because the waiter might need to fill in codes for menu items. That is why it is also within the boundary of change.

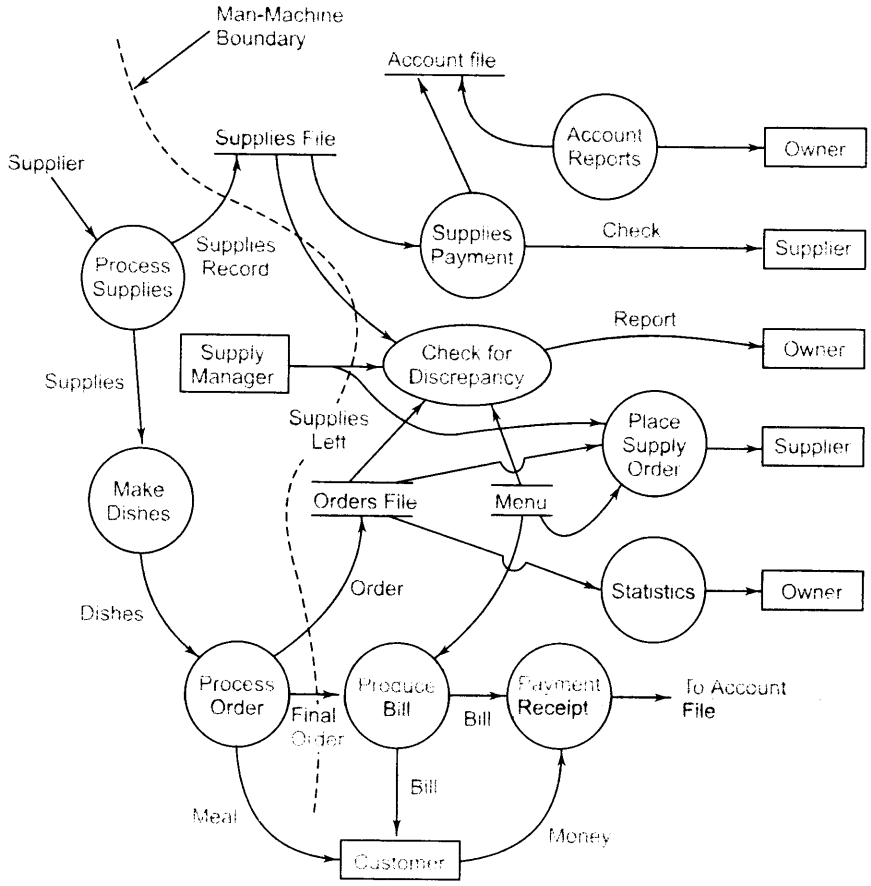

Figure 3.6: The DFD for the new restaurant system.

The DFD is largely self-explanatory. The major files in the system are: Supplies file, Accounting file, Orders file, and the Menu. Some new processes that did not have

```
Supplies_file = [date + [item_no + quantity + cost]* ]*
Orders_file = [date + [menu_item_no + quantity + status]* ]*
status = satisfied | unsatisfied
order = [menu_item_no + quantity]*
menu = [menu_item_no + name + price + supplies_used]*
supplies_used = [supply_item_no + quantity]*
bill = [name + quantity + price]* +
       total_price + sales_tax + service_charge + grand_total
discrepancy_report = [supply_item_no +
       amt_ordered + amt_left + amt_consumed + descr]*
```

Figure 3.7: Data dictionary for the restaurant.

equivalents earlier have been included in the system. These are "check for discrepancy," "accounting reports," and "statistics." Note that the processes are consistent in that the inputs given to them are sufficient to produce the outputs. For example, "checking for discrepancy" requires the following information to produce the report: total supplies received (obtained from the supplies file), supplies left at the end of the day, total orders placed by the customers (from the orders file), and the consumption rate for each menu item (from the menu). All these are shown as inputs to the process. Supplies required for the next day are assessed from the total orders placed in the day and the orders that could not be satisfied due to lack of supplies (both kept in the order file). To see clearly if the information is sufficient for the different processes, the structure and exact contents of each of the data flows has to be specified. The data dictionary for this is given in Figure 3.

The definitions of the different data flows and files are self-explanatory. Once this DFD and the data dictionary have been approved by the restaurant owner, the activity of understanding the problem is complete. After talking with the restaurant owner the man-machine boundary was also defined (it is shown in the DFD). Now, such tasks as determining the detailed requirements of each of the bubbles shown in the DFD, and determining the nonfunctional requirements, deciding codes for the items in the menu and in the supply list remain. Further refinement for some of the bubbles might be needed. For example, it has to be determined what sort of accounting reports or statistics are needed and what their formats should be. Once these are done, the analysis is complete and the requirements can then be compiled in a requirements specification document.

3.2.3 Object-Oriented Modeling

In object-oriented modeling, a system is viewed as a set of objects. The objects interact with each other through the services they provide. Some objects also interact with the users through their services such that the users get the desired services. Hence, the goal of modeling is to identify the objects (actually the object classes) that exist in the problem domain, define the classes by specifying what state information they encapsulate and what services they provide, and identify relationships that exist between objects of different classes, such that the overall model is such that it supports the desired user services.

Object-oriented modeling and systems have been getting a lot of attention in the recent past. The basic reason for this is the belief that object-oriented systems are going to be easier to build and maintain. It is also believed that transitioning from object-oriented analysis to object-oriented design (and implementation) will be easy, and that object-oriented analysis is more immune to change because objects are more stable than functions. That is, in a problem domain, objects are likely to stay the same even if the exact nature of the problem changes, while this is not the case with function-oriented modeling. Some approaches to object-oriented modeling and design were proposed early [36, 133, 23, 95]. Goals of many of these techniques regarding what to produce are quite similar, and their approaches and notations are also similar. Here, we briefly describe the approach proposed in [36]; the notation we use is UML ([24, 64] or www.uml.org), which is now the de-facto standard notation for OO modeling. We will discuss UML further in Chapter 7 when we discuss OO design; here we discuss some concepts needed to discuss analysis.

Basic Concepts and Notation

In understanding or modeling a system using an object-oriented modeling technique, the system is viewed as consisting of *objects*. Each object has certain *attributes*, which together define the object. Separation of an object from its attributes is a natural method that we use for understanding systems (a man is separate from his attributes of height, weight, etc.). In object-oriented systems, attributes hold the state (or define the state) of an object. An attribute is a pure data value (like integer, string, etc.), not an object.

Objects of similar type are grouped together to form an *object class* (or just *class*). A class is essentially a type definition, which defines the state space of objects of its type and the operations (and their semantics) that can be applied to objects of that type. Formation of classes is also a general technique used by humans for understanding systems and differentiating between classes (e.g., an apple tree is an instance of the class of trees, and the class of trees is different from the class of birds).

An object also provides some *services* or *operations*. These services are the only means by which the state of the object can be modified or viewed from outside. For operating a service, a *message* is sent to the object for that service. In general, these

services are defined for a class and are provided for each object of that class. Encapsulating services and attributes together in an object is one of the main features that distinguishes an object-oriented modeling approach from data modeling approaches, like the ER diagrams.

Class diagrams represent a structure of the problem graphically using a precise notation. In a class diagram, a class is represented as a portrait-style rectangle divided into three parts. The top part contains the name of the class. The middle part lists the attributes that objects of this class possess. And the third part lists the services provided by objects of this class

To model relationship between classes, a few structures are used. The *generalization-specialization* structure can be used by a class to inherit all or some attributes and services of a general class and add more attributes and services. This structure is modeled in object-oriented modeling through inheritance. By using a general class and inheriting some of its attributes and services and adding more, one can create a class that is a specialized version of the general class. And many specialized classes can be created from a general class, giving us class hierarchies. The *aggregation* structure models the whole-part relationship. An object may be composed of many objects; this is modeled through the aggregation structure. The representation of these in a class diagram is shown in Figure 3.

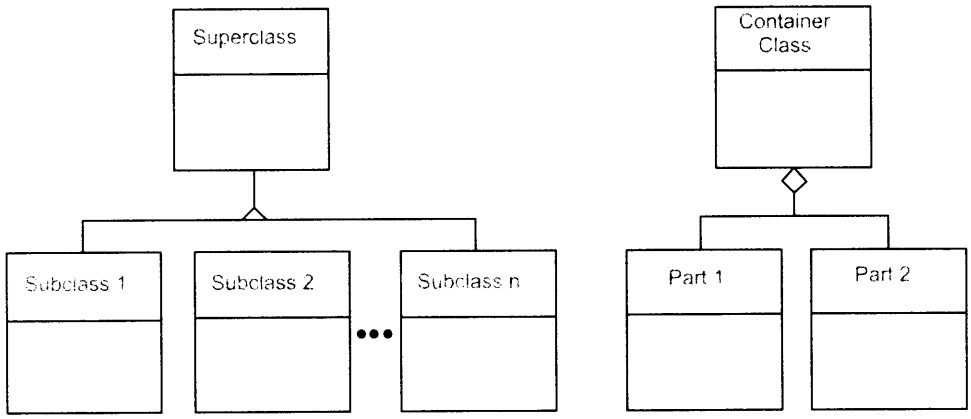

Figure 3.8: Class structures.

In addition to these, instances of a class may be related to objects of some other class. For example, an object of the class Employer may be related to many objects of the class Employee. This relationship between objects also has to be captured if a system is to be modeled properly. This is captured through *associations*. An association is shown in the class diagram by having a line between the two classes. The multiplicity

of an association specifies how many instances of one class may relate to an instances of the other class through this association. An association between two classes can be one-to-one (i.e., one instance of one class is related to exactly one instance of the other class), one-to-many, or some other special cases. Multiplicity is specified by having a star (*) on the line adjacent to the class representing zero or more instances of the class may be related to an instance of the other class.

Let us illustrate the use of some of these relationships and their representation through the use of an example. Suppose a system is being contemplated for a drugstore that will compute the total sales of the drugstore along with the total sales of different chemists that man the drugstore. The drugs are of two major types—off-the-shelf and prescription drugs. The system is to provide help in procuring drugs when out of stock, removing them when expired, replenishing the off-the-shelf drugs when needed, etc. A model of the system is shown in Figure 3.

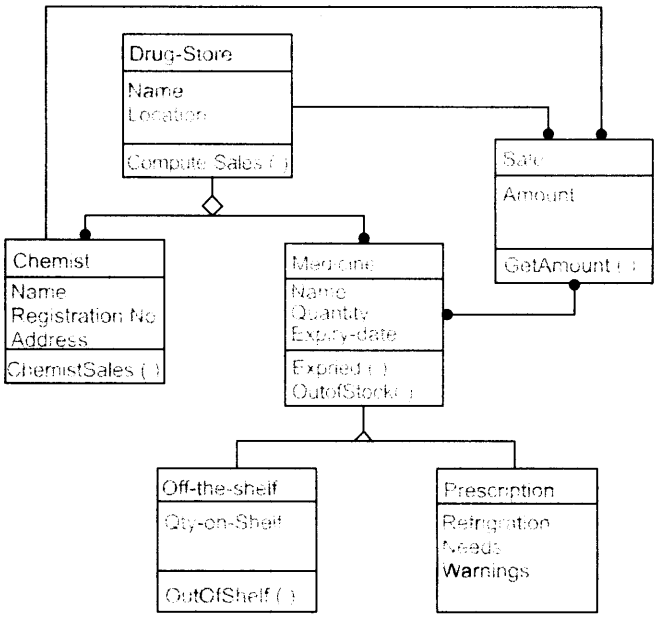

Figure 3.9: Model of a drugstore.

Let us briefly explain this class diagram. It has five classes of objects, each with a defined name, some attributes, and services. For example, an object of class `Chemist` has the attributes Name, Registration number, and Address. It has one service Chemist-Sales(), which computes the total sales by this chemist. The `Drug-Store` class is an aggregation of the class `Medicine` and the class `Chemist` (representing that a drugstore is composed of medicines and chemists). A `Medicine` may either be `Off-the-shelf` or

`Prescription`. The class `Medicine` has some attributes like Name, Quantity in stock, and Expiry-date, and has services like Expired() (to list the expired medicines), OutOf-Stock() (to list medicines that are no longer in stock), etc. These attributes and services are inherited by the two specialized classes. In addition to these, the `Off-the-shelf` class has another attribute qty-on-shelf, representing how many have been put on the shelf and have services related to shelf stock. On the other hand, the `Prescription` class has Refrigeration-needs and Warnings as specialized attributes and services related to them. There are various associations in this model. For example, there is an association between `Sale` and `Medicine`. This association is one-to-many, that is, one sale could be of many medicines. Similarly, `Drug-Store` is associated to `Medicine` and `Chemist`, and `Chemist` is associated with `Sale`.

Performing Analysis

Now that we know what a model of a system consists of, the next question that arises is how to obtain the model for a system. In other words, how do we actually perform the analysis? As mentioned earlier, there can be no "algorithm" to perform the analysis or generate the SRS. Here we briefly discuss the set of guidelines given in [36], according to which the major steps in the analysis are:

- Identifying objects and classes
- Identifying structures
- Identifying attributes
- Identifying associations
- Defining services

Identifying Objects and Classes. An object during analysis is an encapsulation of attributes on which it provides some exclusive services [36]. It represents something in the problem space. It has been argued that though things like interfaces between components, functions, etc. are generally volatile and change with changing needs, objects are quite stable in a problem domain.

To identify analysis objects, start by looking at the problem space and its description. Obtain a brief summary of the problem space. In the summary and other descriptions of the problem space, consider the nouns. Frequently, nouns represent entities in the problem space which will be modeled as objects. Structures, devices, events remembered, roles played, locations, organizational units, etc. are good candidates to consider. A candidate should be included as an object if the system needs to remember something about the object, the system needs some services from the object to perform its own services, and the object has multiple attributes (i.e., it is a high-level object encapsulating some attributes). If the system does not need to keep information about some

real-world entity or does not need any services from the entity, it should not be considered as an object for modeling. Similarly, carefully consider objects that have only one attribute; such objects can frequently be included as attributes in other objects. Though the analysis focuses on identifying objects, in modeling, classes for these objects are represented.

Identifying Structures. Structures represent the hierarchies that exist between object classes. All complex systems have hierarchies. In object-oriented modeling, the hierarchies are defined between classes that capture generalization-specialization and whole-part relationships. To identify the classification structure, consider the classes that have been identified as a generalization and see if there are other classes that can be considered as specializations of this. The specializations should be meaningful for the problem domain. For example, if the problem domain does not care about the material used to make some objects, there is no point in specializing the classes based on the material they are made of. Similarly, consider classes as specializations and see if there are other classes that have similar attributes. If so, see if a generalized class can be identified of which these are specializations. Once again, the structure obtained must naturally reflect the hierarchy in the problem domain; it should not be "extracted" simply because some classes have some attributes with the same names.

To identify assembly structure, a similar approach is taken. Consider each object of a class as an assembly and identify its parts or components. See if the system needs to keep track of the parts. If it does, then the parts must be reflected as objects; if not, then the parts should not be modeled as separate objects. Then, consider an object of a class as a part and see to which class's object it can be considered as belonging. Once again, this separation is maintained only if the system needs it. As before, the structures identified should naturally reflect the hierarchy in the problem domain and should not be "forced."

Identifying Attributes. Attributes add detail about the class and are the repositories of data for an object. For example, for an object of class Person, the attributes could be the name, sex, and address. The data stored in forms of values of attributes are hidden from outside the objects and are accessed and manipulated only by the service functions for that object. Which attributes should be used to define the class of an object depends on the problem and what needs to be done. For example, while modeling a hospital system, for the class Person attributes of height, weight, and date of birth may be needed, although these may not be needed for a database for a county that keeps track of populations in various neighborhoods.

To identify attributes, consider each class and see which attributes are needed by the problem domain. This is frequently a simple task. Then position each attribute properly using the structures; if the attribute is a common attribute, it should be placed in the superclass, while if it is specific to a specialized object it should be placed with the subclass. While identifying attributes, new classes may also get defined or old classes may disappear (e.g., if you find that a class really is an attribute of another).

Identifying Associations. Associations capture the relationship between instances of various classes. For example, an instance of the class Company may be related to an instance of the class Person by an "employs" relationship. This is similar to what is done in ER modeling. And like in ER modeling, an instance connection may be of 1:1 type representing that one instance of this type is related to exactly one instance of another class. Or it could be 1:M, indicating that one instance of this class may be related to many instances of the other class. There are M:M connections, and there are sometimes multi-way connections, but these are not very common. The associations between objects are derived from the problem domain directly once the objects have been identified. An association may have attributes of its own; these are typically attributes that do not naturally belong to either object. Although in many situations they can be "forced" to belong to one of the two objects without loosing any information, it should not be done unless the attribute naturally belongs to the object.

Defining Services. An object performs a set of predefined services. A service is performed when the object receives a message for it. Services really provide the active element in object-oriented modeling; they are the agent of state change or "processing." It is through the services that desired functional services can be provided by a system. To identify services, first identify the *occur* services, which are needed to create, destroy, and maintain the instances of the class. These services are generally not shown in the class diagrams. Other services depend on the type of services the system is providing. A method for identifying services is to define the system states and then in each state list the external events and required responses. For each of these, identify what services the different classes should possess.

All the classes and their relationships are shown in a class diagram. The class diagram, clearly, gets large and complex for large systems. To handle the complexity, a *subject layer* in which the class model is partitioned into various subjects, with each subject containing some part of the diagram is suggested. Typically, a subject will contain many related classes.

An Example

Let us consider the example of the restaurant, whose structured analysis was performed earlier. By stating the goals of the system (i.e., automate the bill generation for orders given by customers, obtain sale statistics, determine discrepancy between supplies taken and supplies consumed, automate ordering of supplies) and studying the problem domain (i.e., the restaurant with customer, supplier, menu, etc.), we can clearly see that there are at least the following classes of objects: `Restaurant`, `Restaurant owner`, `Bill`, `Menu`, `CustomerOrder`, `Supplier`, `SupplyOrder`, `Supply Handling`, and `Dishes`. Each of these entities plays an important role in the system. We consider this as the starting point and the initial classes. By looking at the objectives and scope of the system, we find that no information about the `Supplier` or the `Operator` needs to

be maintained in the system. Hence, they need not be modeled in the system as objects. For the same reason, entities like `Dishes` and `Restaurant owner` are not modeled as objects. The initial classes are shown in Figure 3. Note that this model will further evolve and new classes may get added and some of these may eventually not be needed.

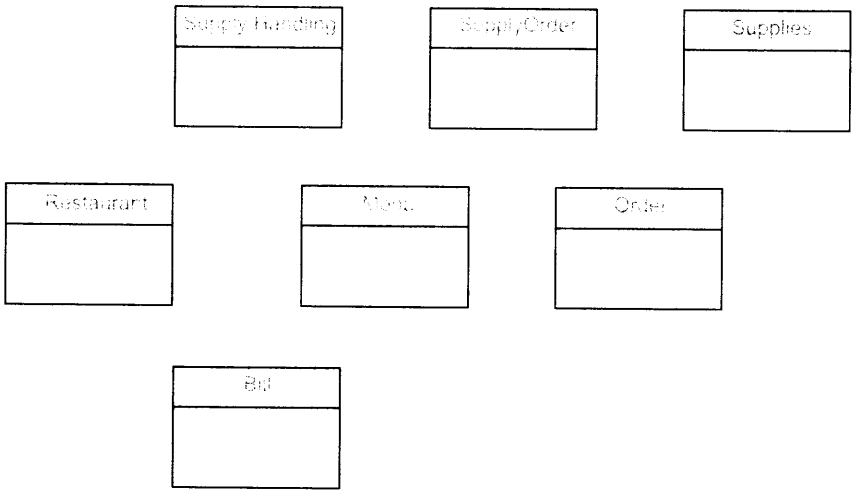

Figure 3.10: Initial classes for the restaurant example

Now let us try to identify structure between these classes. Clearly, a `Restaurant` is an aggregation of `Menu` and `Supply handling`. As dishes, operator, etc. are not considered objects, they do not show up as components of `Restaurant`. Further, a `Menu` is an aggregation of many `MenuItems`. This requires us to add `MenuItem`. Similarly, the `SupplyOrder` is an aggregation of (many) `SupplyItems`. This also requires us to add `SupplyItem` as a new class. Furthermore, the association between `SupplyOrder` and `SupplyItem` has an attribute quantity, reflecting the quantity ordered for a particular item by an order. There is no generalization-specialization hierarchy in this.

Many attributes of various items can be directly identified. A `MenuItem` has attributes of Number, Name, Price, Supplies used (i.e., which supplies it uses and quantity; this is needed to detect discrepancies in consumption and supplies used). Similarly, `SupplyItem` has Item name and Unit price as attributes.

With this, we are ready to identify relationships between objects. The `Supply handling` (unit) is related to (many) `SupplyOrder`. Similarly, an `Order` (by a customer) is related to many `MenuItem`. Furthermore, this association has an attribute of its own—quantity. The quantity of the particular `MenuItem` ordered in a particular `Order` is naturally a property of the association between the specific `Order` and the specific `MenuItem`.

Finally, we have to identify the services. Keeping our basic services of the system in mind (generate sales statistics, bill, discrepancy report, sale order), we define services of various classes. `Supply handling` object has the services CreditSupply() (used to record the receipt of supplies), DebitSupply() (used to record the supplies taken out), and PlaceOrder() (to place order of supplies). For `SupplyOrder`, one service is identified—ProduceCheck() to produce the check for the particular order. The object `Order` has one service—ProduceBill()—to produce the bill for the particular order. Handling the bill as essentially something generated for each order, we remove the object `Bill` from the object layer. The main object `Restaurant` has the services SaleStat() to generate the sale statistics for which it will require all order information (which it will obtain through its association with `Order`). It also has the service Discrepancy() to generate a discrepancy report. For this, it will need to find out what items have been consumed their quantity, and how much supply was debited. The former it can obtain from all the orders and the latter from `Supply handling`. The final class diagram is shown in Figure 3.

3.2.1 Prototyping

Prototyping takes a different approach to problem analysis as compared to modeling-based approaches. In prototyping, a partial system is constructed, which is then used by the client, users, and developers to gain a better understanding of the problem and the needs. Hence, actual experience with a prototype that implements part of the eventual software system are used to analyze the problem and understand the requirements for the eventual software system. A software prototype can be defined as a partial implementation of a system whose purpose is to learn something about the problem being solved or the solution approach [46]. As stated in this definition, prototyping can also be used to evaluate or check a design alternative (such a prototype is called a *design prototype* [46]). Here we focus on prototyping used primarily for understanding the requirements.

The rationale behind using prototyping for problem understanding and analysis is that the client and the users often find it difficult to visualize how the eventual software system will work in their environment just by reading a specification document. Visualizing the operation of the software that is yet to be built and whether it will satisfy the ultimate objectives, merely by reading and discussing the paper requirements, is indeed difficult. This is particularly true if the system is a totally new system and many users and clients do not have a good idea of their needs. The idea behind prototyping is that clients and the users can assess their needs much better if they can see the working of a system, even if the system is only a partial system. Prototyping emphasizes that actual practical experience is the best aid for understanding needs. By actually experimenting with a system, people can say, "I don't want this feature" or "I wish it had this feature" or "This is wonderful."

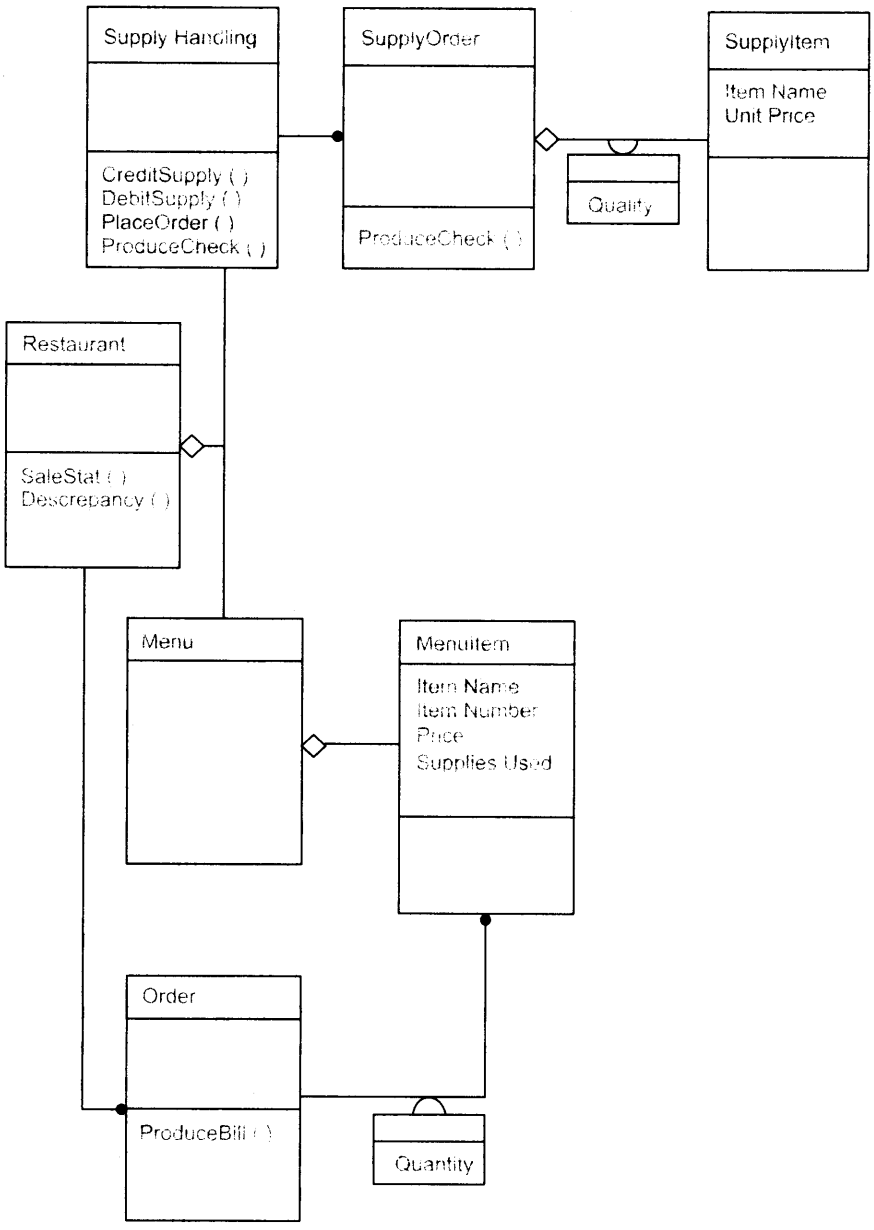

Figure 3.11: Class diagram for the restaurant.

There are two approaches to prototyping: throwaway and evolutionary [44, 46]. In the *throwaway* approach the prototype is constructed with the idea that it will be discarded after the analysis is complete, and the final system will be built from scratch. In the *evolutionary* approach, the prototype is built with the idea that it will eventually be converted into the final system. From the point of view of problem analysis and understanding, the throwaway prototypes are more suited. For the rest of the discussion we limit our attention to throwaway prototypes.

The first question that needs to be addressed is whether or not to prototype. In other words, it is important to clearly understand when prototyping should be done. The requirements of a system can be divided into three sets—those that are well understood, those that are poorly understood, and those that are not known [46]. In a throwaway prototype, the poorly understood requirements are the ones that should be incorporated. Based on the experience with the prototype, these requirements then become well understood, as shown in Figure 3.

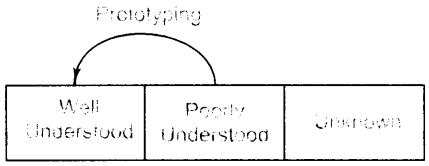

Figure 3.12: Throwaway prototyping.

It might be possible to divide the set of poorly understood requirements further into two sets—those critical to design, and those not critical to design [46]. The requirements that can be easily incorporated in the system later are considered noncritical to design. If which of the poorly understood requirements are critical and which are noncritical can be determined, then the throwaway prototype should focus mostly on the critical requirements. Overall, we can say that if the set of poorly understood requirements is substantial (in particular the subset of critical requirements), then a throwaway prototype should be built.

The development process using a throwaway prototype was discussed earlier in Chapter 2. The development activity starts with an SRS for the prototype. However, developing the SRS for the prototype requires identifying the functions that should be included in the prototype. This decision is typically application-dependent. As mentioned earlier, in general, those requirements that tend to be unclear and vague, or where the clients and users are unsure or keep changing their mind, are the ones that should be implemented in the prototype. User interface, new features to be added (beyond automating what is currently being done), and features that may be infeasible, are common candidates for prototyping. Based on what aspects of the system are included in the prototype, the prototyping can be considered *vertical* or *horizontal* [110]. In hori-

zontal prototyping the system is viewed as being organized as a series of layers and some layer is the focus of prototyping. For example, the user interface layer is frequently a good candidate for such prototyping, where most of the user interface is included in the prototype. In vertical prototyping, a chosen part of the system, which is not well understood, is built completely. This approach is used to validate some functionality or capability of the system.

Development of a throwaway prototype is fundamentally different from developing final production-quality software. The basic focus during prototyping is to keep costs low and minimize the prototype production time. Due to this, many of the bookkeeping, documenting, and quality control activities that are usually performed during software product development are kept to a minimum during prototyping. Efficiency concerns also take a back seat, and often very high-level interpretive languages are used for prototyping. For these reasons, temptation to convert the prototype into the final system should be resisted.

Experience is gained by putting the system to use by the actual client and users. Constant interaction is needed with the client/users during this activity to understand their responses. Questionnaires and interviews might be used to gather user response.

The final SRS is developed in much the same way as any SRS is developed. The difference here is that the client and users will be able to answer questions and explain their needs much better because of their experience with the prototype. Some initial analysis is also available.

For prototyping for requirements analysis to be feasible, its cost must be kept low. Consequently, only those features that will have a valuable return from the user experience are included in the prototype. Exception handling, recovery, conformance to some standards and formats are typically not included in prototypes. Because the prototype is to be thrown away, only minimal development documents need to be produced during prototyping; for example, design documents, a test plan, and a test case specification are not needed during the development of the prototype. Another important cost-cutting measure is reduced testing. Testing consumes a major part of development expenditure during regular software development. By using cost-cutting methods, it is possible to keep the cost of the prototype to less than a few percent of the total development cost.

The cost of developing and running a prototype can be around 10% of the total development cost [72]. However, it should be pointed out that if the cost of prototyping is 10% of the total development cost, it does not mean that the cost of development has increased by this amount. The main reason is that the benefits obtained due to the use of prototype in terms of reduced requirement errors and reduced volume of requirement change requests are likely to be substantial (see the examples given earlier in this chapter), thereby reducing the cost of development itself.

An Example

Consider the example of the restaurant automation discussed earlier. An initial structured analysis of the problem was also shown earlier. During the analysis the restaurant

owner was quite apprehensive about the ability and usefulness of the system. She felt that it was risky to automate, as an improper system might cause considerable confusion and lead to a loss of clientele. Due to the risks involved, it was decided to build a throwaway prototype. Note that the basic purpose of the prototype in this situation is not to uncover or clarify requirements but to ascertain the utility of the automated system. Of course, the experience with the prototype will also be used to ensure that the requirements are correct and complete.

The first step in developing a prototype is to prepare an SRS for the prototype. The SRS need not be formal but should identify the different system utilities to be included in the prototype. As mentioned earlier, these are typically the features that are most unclear or where the risk is high. It was decided that the prototype will demonstrate the following features:

1. Customer order processing and billing
2. Supply ordering and processing

The first was included, as that is where the maximum risk exists for the restaurant (after all, customer satisfaction is the basic objective of the restaurant, and if customers are unhappy the restaurant will lose business). The second was included, as maximum potential benefit can be derived from this feature. Accounting and statistics generation were not to be included in the prototype.

The prototype was developed using a database system, in which good facilities for data entry and form (bill) generation exist. The user interface for the waiters and the restaurant manager was included in the prototype. The system was used, in parallel with the existing system, for a few weeks, and informal surveys with the customers were conducted.

Customers were generally pleased with the accuracy of the bills and the details they provided. Some gave suggestions about the bill layout. Based on the experience of the waiters, the codes for the different menu items were modified to an alphanumeric code. They found that the numeric codes used in the prototype were hard to remember. The experience of the restaurant manager and feedback from the supplier were used to determine the final details about supply processing and handling.

3.3 Requirements Specification

The final output is the software requirements specification document (SRS). For smaller problems or problems that can easily be comprehended, the specification activity might come after the entire analysis is complete. However, it is more likely that problem analysis and specification are done concurrently. An analyst typically will analyze some parts of the problem and then write the requirements for that part. In practice, problem analysis and requirements specification activities overlap, with movement from both

activities to the other, as shown in Figure 3. However, as all the information for specification comes from analysis, we can conceptually view the specification activity as following the analysis activity.

The first question that arises is: If formal modeling is done during analysis, why are the outputs of modeling—the structures that are built (e.g., DFD and DD, Object diagrams)—not treated as an SRS? The main reason is that modeling generally focuses on the problem structure, not its external behavior. Consequently, things like user interfaces are rarely modeled, whereas they frequently form a major component of the SRS. Similarly, for ease of modeling, frequently "minor issues" like erroneous situations (e.g., error in output) are rarely modeled properly, whereas in an SRS, behavior under such situations also has to be specified. Similarly, performance constraints, design constraints, standards compliance, recovery, etc., are not included in the model, but must be specified clearly in the SRS because the designer must know about these to properly design the system. It should therefore be clear that the outputs of a model cannot form a desirable SRS.

For these reasons, the transition from analysis to specification should also not be expected to be straightforward, even if some formal modeling is used during analysis. It is not the case that in specification the structures of modeling are just specified in a more formal manner. A good SRS needs to specify many things, some of which are not satisfactorily handled during modeling. Furthermore, sometimes the structures produced during modeling are not amenable for translation into external behavior specification (which is what is to be specified in an SRS). For example, the object diagram produced during an OO analysis is of limited use when specifying the external behavior of the desired system.

Essentially, what passes from requirements analysis activity to the specification activity is the knowledge acquired about the system. The modeling is essentially a tool to help obtain a thorough and complete knowledge about the proposed system. The SRS is written based on the knowledge acquired during analysis. As converting knowledge into a structured document is not straightforward, specification itself is a major task, which is relatively independent.

A consequence of this is that it is relatively less important to model "completely," compared to specifying completely. As the primary objective of analysis is problem understanding, while the basic objective of the requirements phase is to produce the SRS, the complete and detailed analysis structures are not critical. In fact, it is possible to develop the SRS without using formal modeling techniques. The basic aim of the structures used in modeling is to help in knowledge representation and problem partitioning, the structures are not an end in themselves.

With this in mind, let us start our discussion on requirements specification. We start by discussing the desirable characteristics of an SRS.