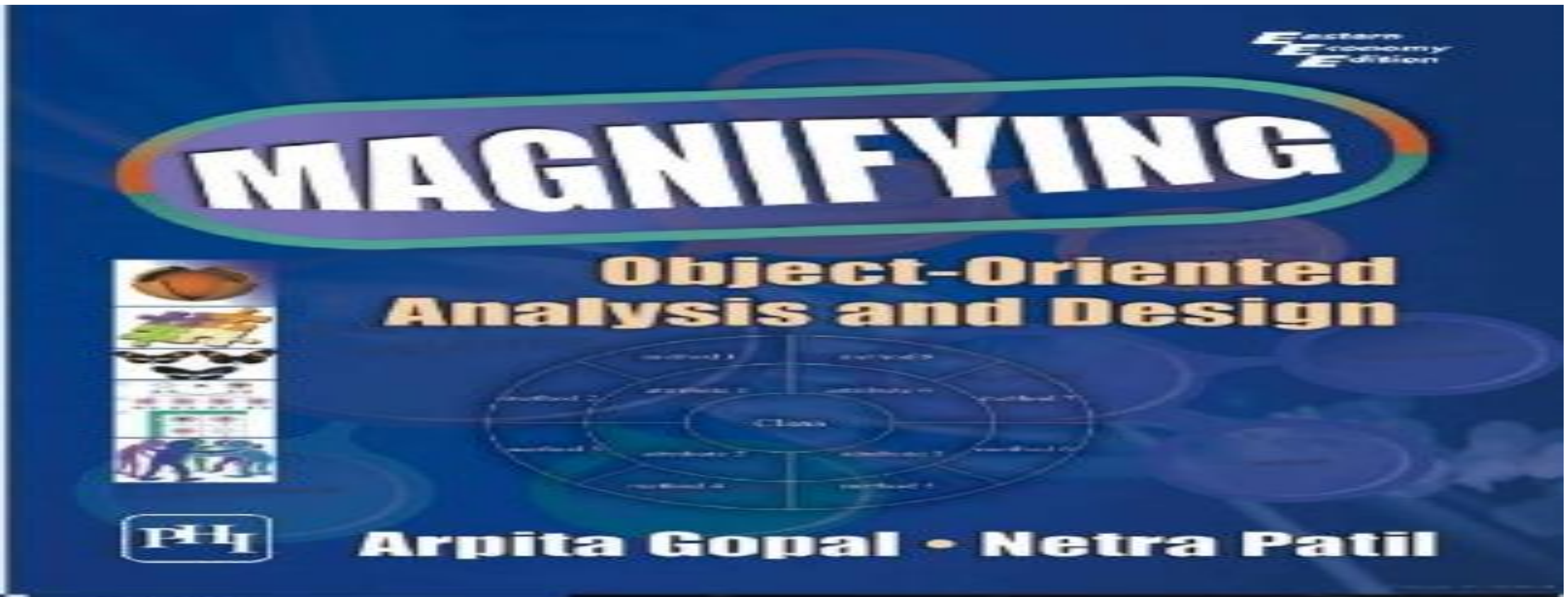
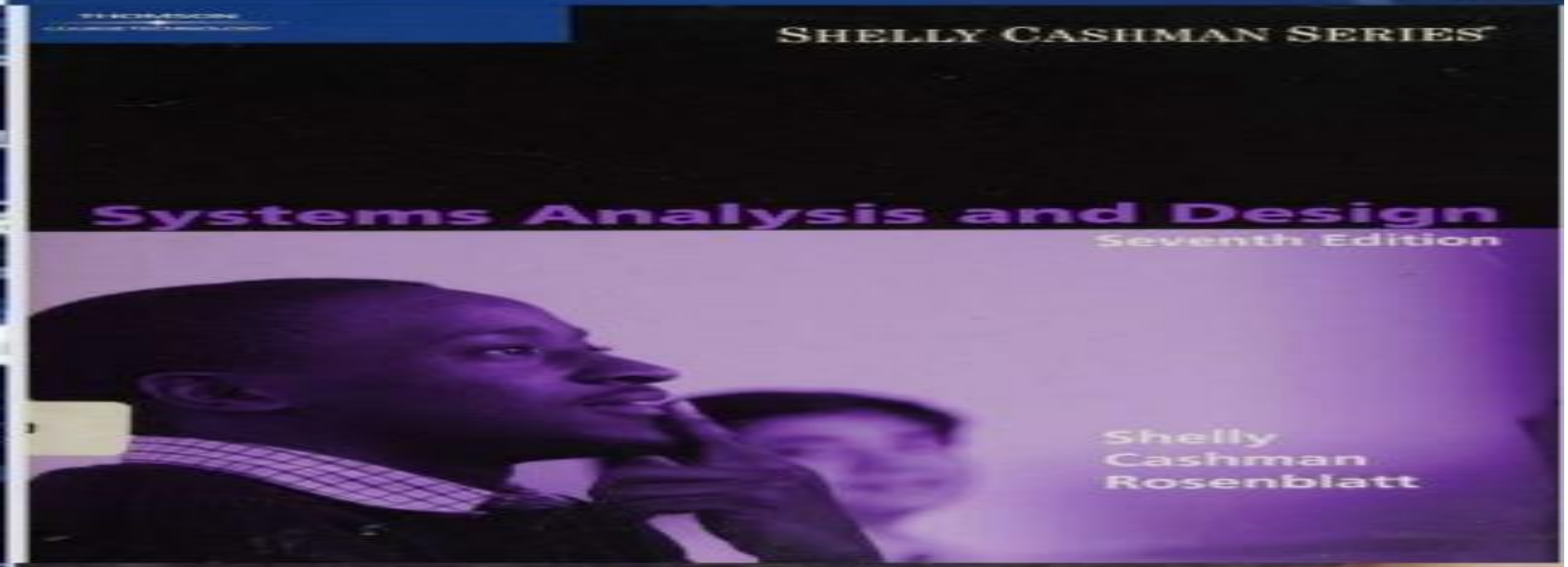

Eastern Economy Edition

MAGNIFYING

Object-Oriented Analysis and Design

PHI

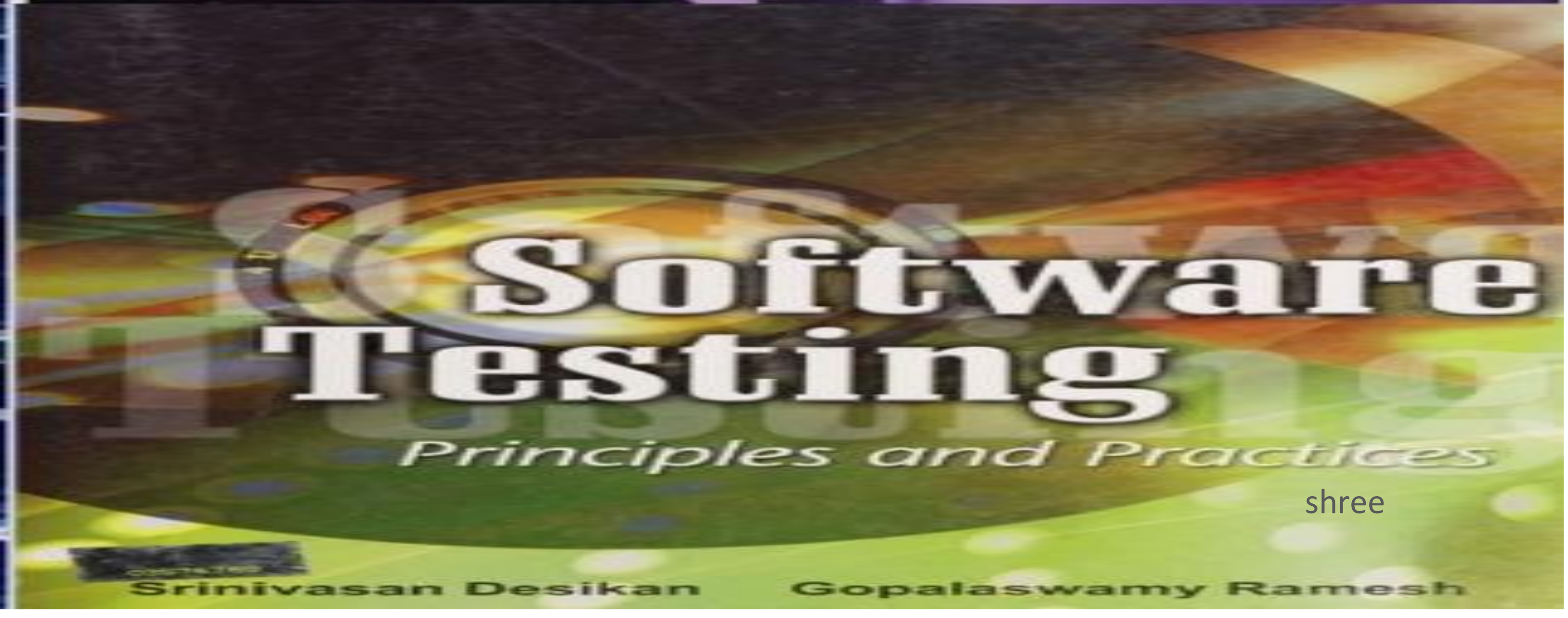
Arpita Gopal • Netra Patil


SHELLY CASHMAN SERIES

Systems Analysis and Design

Seventh Edition

Shelly
Cashman
Rosenblatt


Software Testing

Principles and Practices

shree

Srinivasan Desikan Gopalaswamy Ramesh

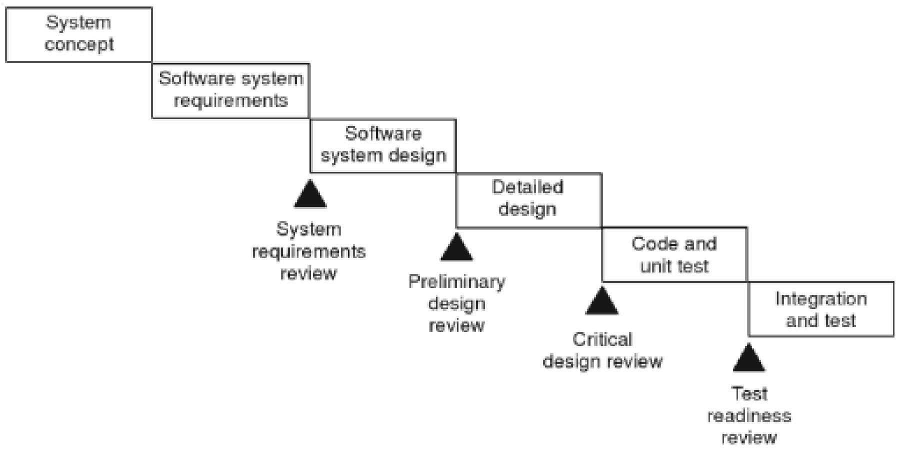
Unit-1 SYSTEM ANALYSIS AND DESIGN

- SOFTWARE DEVELOPMENT MODELS

Earlier in the process of software development, code was developed first and then debugged. There was no formal design or analysis approach. This code and debug approach was not found optimal as complexity of software systems increased. As the time progressed the approach to developing comples hardware system was well understood, and gradually different models for developing software evolved. Discussed below are a few models used for software development

- Waterfall Model

The waterfall approach to information system development emphasizes completing a phase of the development before proceeding to the next phase. After completion of the certain phases, a baseline is established that "freezes" the products of the development at that point. If need of a change is identified, a formal change process is followed to make the change. The graphic representation of these phases in software development resembles the downward flow of a waterfall. Figure 1.1 depicts the phases of the waterfall model.



Each box in Figure 1.1 represents a phase. Output from each phase includes documentation. The phases below the detailed design phase include software as part of their output. Transition from phase to phase is accomplished by holding formal reviews which provide insight into the progress. Baselines are established at critical points on the waterfall model, the last of which is the product baseline. The final baseline is accompanied by audits.

The application most suitable for the use of the waterfall model should be limited to situations where the requirements and the implementation of those requirements are very well understood.

For example, if a company has experience in building accounting systems, I/O controllers, or compilers, then building another such product based on the existing designs is best managed with the waterfall model

- The incremental model

The incremental model performs the waterfall in overlapping sections (see Figure 1.2), attempting to compensate for the length of waterfall model projects by producing usable functionality earlier. The project using the incremental model may start with general objectives, later some portion of these objectives are defined as requirements and are implemented, followed by the next portion of the objectives until all objectives are implemented.

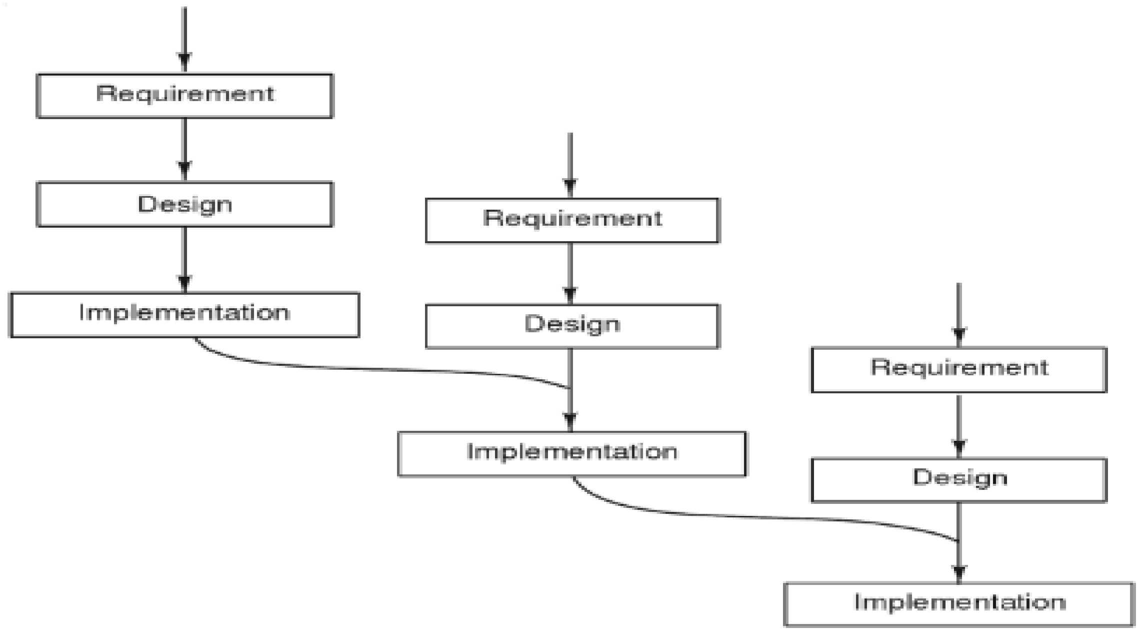


FIGURE 1.2 Incremental model.

But this approach of use of general objectives initially, rather than complete requirements can be uncomfortable for management as some modules will be completed long before the others. Another difficulty is that formal reviews and audits are more difficult to implement on increments than on a complete system. Additionally, there can be a tendency to push difficult problems to the future to demonstrate early success to management. To overcome these problems, well-defined interfaces are required.

Use of the incremental model is appropriate when, it is too risky to develop the wholetivate Windows system at once.

- Spiral Model

The spiral model (Figure 1.3) is proposed by Barry Boehm in which prototyping is used to control cost. The Boehm spiral model has become quite popular among ADE (Aerospace, Defense and Engineering) specialists, and is not so familiar among business developers. It is particularly useful in projects, that are risky in nature. Business projects are more conservative. They tend to use mature technology and to work well-known problems.

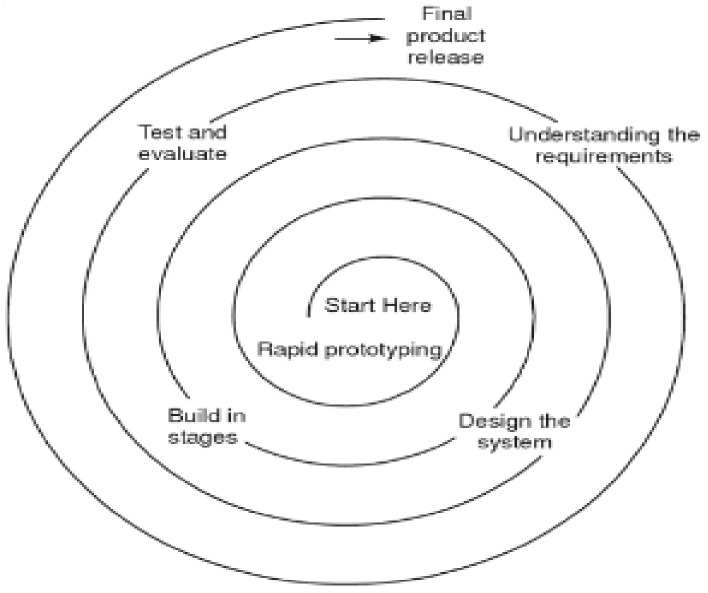


FIGURE 1.3 Spiral model.

TABLE 1.2 Strengths and weaknesses of models

	Waterfall model	*Incremental model*	*Boehm spiral model*
Strengths			
Allows for work force specialization	×	×	×
Orderliness appeals to management	×	×	×
Can be reported about	×	×	×
Facilitates allocation of resources	×	×	×
Early functionality		×	×
Does not require a complete set of requirements at the onset		×	×
Resources can be held constant		×	
Control costs and risk through prototyping			×
Weaknesses			
Requires a complete set of requirements at the onset	×		
Enforcement of non-implementation attitude hampers analyst/designer communications	×		
Beginning with less defined general objectives may be uncomfortable for management		×	×
Requires clean interfaces between modules		×	
Incompatibility with a formal review and audit procedure		×	×
Tendency for difficult problems to be pushed to the future so that the initial promise of the first increment is not met by subsequent products		×	×

The incremental model may be used with a complete set of requirements or with less defined general objectives.

- REQUIREMENTS MODELING :-

  This chapter Data and Process Modeling FIGURE 3-2 The systems analysis phase consists of requirements modeling, data and process modeling, object modeling, and consideration of development Requirements Modeling — Development Strategies describes requirements modeling, which involves fact-finding to describe the current system and identification of the requirements for the new system, such as outputs, inputs, processes, performance, and security. Outputs refer to electronic or printed information produced by the system. Inputs refer to necessary data that enters the system, either manually or in an automated manner. Processes refer to the logical rules that are applied to transform the data into meaningful information. Performance refers to system characteristics such as speed, volume, capacity, availability, and reliability. Security refers to hardware, software, and procedural controls that safeguard and protect the system and its data from internal or external threats.

- INTERVIEWS:-

  Systems analysts spend a great deal of time talking with people, both inside and outside the IT department. Much of that time is spent conducting interviews, which are the most common fact-finding technique. An interview is a planned meeting during which you obtain information from another person. You must have the skills needed to plan, conduct, and document interviews successfully. After you identify the information you need, as described earlier in the chapter, you can begin the interviewing process, which consists of seven steps for each interview:

  1.Determine the people to interview.
  2.Establish objectives for the interview.
  3.Develop interview questions.
  4.Prepare for the interview.
  5.Conduct the interview.
  6.Document the interview.
  7.Evaluate the interview.

- Step 1: Determine the People to Interview :-

  To get an accurate picture, you must select the right people to interview and ask them the right questions. During the preliminary investigation, you talked mainly to middle managers or department heads. Now, during the systems analysis phase, you might need to interview people from all levels of the organization. Although you can select your interview candidates from the formal organization charts that you reviewed earlier, you also must consider any informal structures that exist in the organization. Informal structures usually are based on interpersonal relationships and can develop from previous work assignments, physical proximity, unofficial procedures, or personal relationships such as the informal gathering shown in Figure 3-18. In an informal structure, some people have more influence or knowledge than appears on an organization chart. Your knowledge of the company's formal and informal structures helps you determine the people to interview during the systems analysis phase. Should you interview several people at the same time? Group interviews can save time and provide an opportunity to observe interaction among the participants. Group interviews also can present problems. One person might dominate the conversation, even when questions are addressed specifically to others. Organization level also can present a problem, as the presence of senior managers in an interview might prevent lower-level employees from expressing themselves candidly.

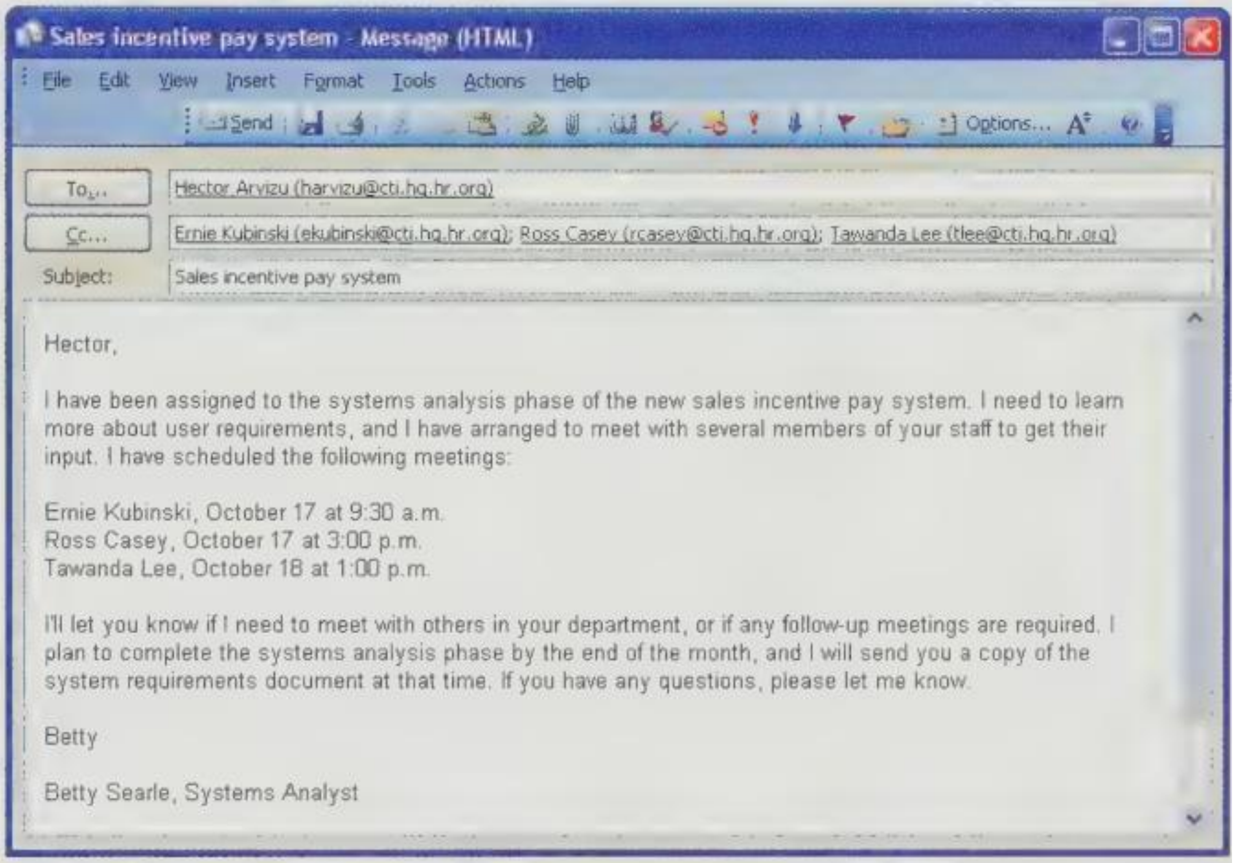
- Step 2: -Establish Objectives for the Interview:-
  After deciding on the people to interview, you must establish objectives for the session. First, you should determine the general areas to be discussed, and then list the facts you want to gather. You also should try to solicit ideas, suggestions, and opinions during the interview.
  The objectives of an interview depend on the role of the person being interviewed. Upper-level managers can provide the big picture and help you to understand the system as a whole. Specific details about operations and business processes are best learned from people who actually work with the system on a daily basis. In the early stages of systems analysis, interviews usually are general. As the factfinding process continues, however, the interviews focus more on specific topics. Interview objectives also vary at different stages of the investigation. By setting specific objectives, you create a framework that helps you decide what questions to ask and how to phrase the questions.

- Step 3: Develop Interview Questions:-

Creating a standard list of interview questions helps to keep you on track and avoid unnecessary tangents. Also, if you interview several people who perform the same job, a standard question list allows you to compare their answers. Although you have a list of specific questions, you might decide to depart from it because an answer to one question leads to another topic that you want to pursue. That question or topic then should be included in a revised set of questions used to conduct future interviews. If the question proves to be extremely important, you may need to return to a previous interviewee to query him or her on the topic. The interview should consist of several different kinds of questions: open-ended, closed-ended, or questions with a range of responses. When you phrase your questions, you should avoid leading questions that suggest or favor a particular reply. For example, rather than asking, “What advantages do you see in the proposed system?” you might ask, “Do you see any advantages in the proposed system?”

- OPEN-ENDED QUESTIONS:- Open-ended questions encourage spontaneous and unstructured responses. Such questions are useful when you want to understand a larger process or draw out the interviewee’s opinions, attitudes, or suggestions. Here are some examples of open-ended questions: What are users saying about the new system? How is this task performed? Why do you perform the task that way? How are the checks reconciled? What added features would you like to have in the new billing system? Also, you can use an open-ended question to probe further by asking: Is there anything else you can tell me about this topic?
- CLOSED-ENDED QUESTIONS-: Closed-ended questions limit or restrict the response. You use closed-ended questions when you want information that is more specific or when you need to verify facts. Examples of closed-ended questions include the following: How many personal computers do you have in this department? Do you review the reports before they are sent out? How many hours of training does a clerk receive? Is the calculation procedure described in the manual? How many customers ordered prod- ucts from the Web site last month? RANGE-OF-RESPONSE QUESTION

- RANGE-OF-RESPONSE QUESTIONS:- Range-of-response questions are closed-ended questions that ask the person to evaluate something by providing limited answers to specific responses or on a numeric scale. This method makes it easier to tabulate the answers and interpret the results. Range-of-response questions might include these: On a scale of 1 to 10, with 1 the lowest and 10 the highest, how effective was your train- ing? How would you rate the severity of the problem: low, medium, or high? Is the sys- tem shutdown something that occurs never, sometimes, often, usually, or always?
- Step 4: Prepare for the Interview After setting the objectives and developing the questions, you must prepare for the interview. Careful preparation is essential because an interview is an important meeting and not just a casual chat. When you schedule the interview, suggest a specific day and time and let the interviewee know how long you expect the meeting to last. It is also a good idea to send an e-mail or place a reminder call the day before the interview. Remember that the interview is an interruption of the other person's routine, so you should limit the interview to no more than one hour. If business pressures force a postponement of the meeting, you should schedule another appointment as soon as it is convenient. Remember to keep department managers informed of your meetings with their staff members. Sending a message to each department manager listing your planned appointments is a good way to keep them informed. Figure 3-19 is an example of such a message. You should send a list of topics to an interviewee several days before the meeting, especially when detailed information is needed, so the person can prepare for the interview and minimize the need for a follow-up meeting. Figure 3-20 shows a sample message that lists specific questions and confirms the date, time, location, purpose, and anticipated duration of the interview. If you have questions about documents, ask the interviewee to have samples available at the meeting. Your advance memo should include a list of the documents you want to discuss, if you know what they are. Also, you can make a general request for documents, as the analyst did in her e-mail shown in Figure 3-20. Two schools of thought exist about the best location for an interview. Some analysts believe that interviews should take place in the interviewee's office, whereas other analysts feel that a neutral location such as a conference room is better. Supporters of interviews in the interviewee's office believe that is the best location because it makes the interviewee feel comfortable during the meeting. A second argument in favor of the interviewee's office is that the office is where he or she has the easiest access to supporting material that might be needed during the discussion. If you provide a complete list of topics in advance, however, the interviewee can bring the necessary items to a conference room or other location. Supporters of neutral locations stress the importance of keeping interruptions to a minimum so both people can concentrate fully. In addition, an interview that is free of interruptions takes less time. If the meeting does take place in the interviewee's office, you should suggest tactfully that all calls be held until the conclusion of the interview.


Sales incentive pay system - Message (HTML)
File Edit View Insert Format Tools Actions Help
Send
Options...

To... Hector Arvizu (harvizu@cti.hq.hr.org)

Cc... Ernie Kubinski (ekubinski@cti.hq.hr.org); Ross Casey (rcasey@cti.hq.hr.org); Tawanda Lee (tlee@cti.hq.hr.org)

Subject: Sales incentive pay system

Hector,

I have been assigned to the systems analysis phase of the new sales incentive pay system. I need to learn more about user requirements, and I have arranged to meet with several members of your staff to get their input. I have scheduled the following meetings:

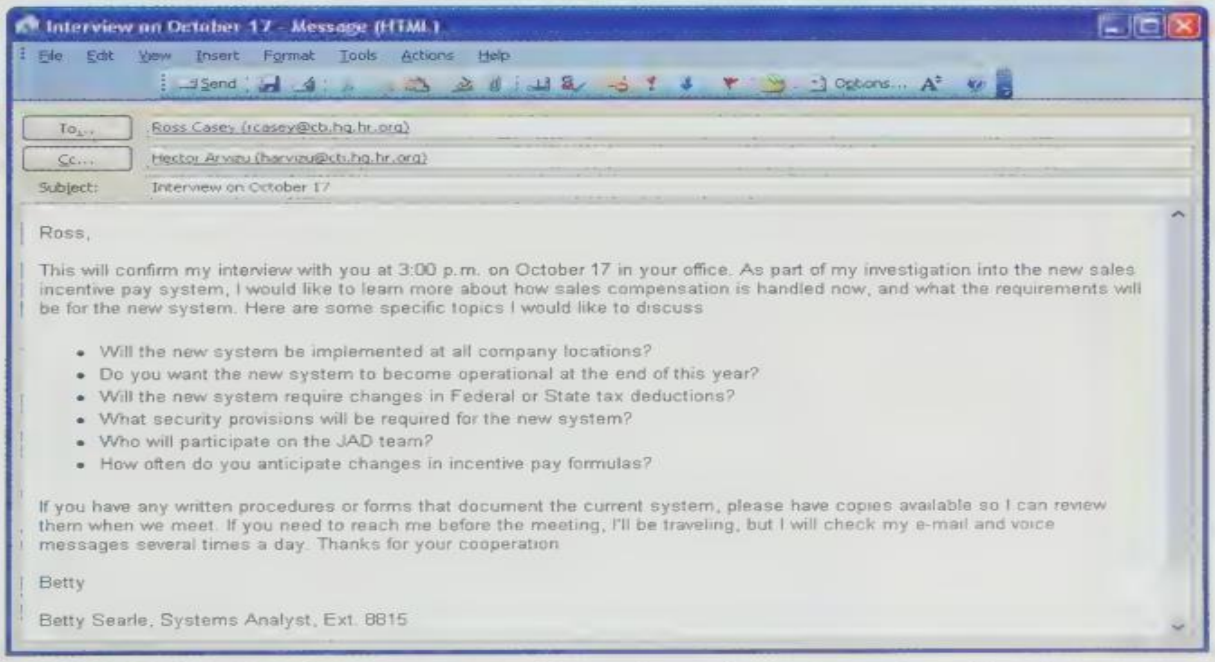
Ernie Kubinski, October 17 at 9:30 a.m.
Ross Casey, October 17 at 3:00 p.m.
Tawanda Lee, October 18 at 1:00 p.m.

I'll let you know if I need to meet with others in your department, or if any follow-up meetings are required. I plan to complete the systems analysis phase by the end of the month, and I will send you a copy of the system requirements document at that time. If you have any questions, please let me know.

Betty

Betty Searle, Systems Analyst

FIGURE 3-19 Sample message to a department head about interviews with people in his group.


Interview on October 17 - Message (HTML)
File Edit View Insert Format Tools Actions Help
Send
Options...

To... Ross Casey (rcasey@cti.hq.hr.org)

Cc... Hector Arvizu (harvizu@cti.hq.hr.org)

Subject: Interview on October 17

Ross,

This will confirm my interview with you at 3:00 p.m. on October 17 in your office. As part of my investigation into the new sales incentive pay system, I would like to learn more about how sales compensation is handled now, and what the requirements will be for the new system. Here are some specific topics I would like to discuss

- Will the new system be implemented at all company locations?
- Do you want the new system to become operational at the end of this year?
- Will the new system require changes in Federal or State tax deductions?
- What security provisions will be required for the new system?
- Who will participate on the JAD team?
- How often do you anticipate changes in incentive pay formulas?

If you have any written procedures or forms that document the current system, please have copies available so I can review them when we meet. If you need to reach me before the meeting, I'll be traveling, but I will check my e-mail and voice messages several times a day. Thanks for your cooperation

Betty

Betty Searle, Systems Analyst, Ext. 8815

FIGURE 3-20 Sample message to confirm a planned meeting and provide advance information about topics to be discussed.

argument in favor of the interviewee's office is that the office is where he or she has the easiest access to supporting material that might be needed during the discussion. If you provide a complete list of topics in advance, however, the interviewee can bring the necessary items to a conference room or other location. Supporters of neutral locations stress the importance of keeping interruptions to a minimum so both people can concentrate fully. In addition, an interview that is free of interruptions takes less time. If the meeting does take place in the interviewee's office, you should suggest tactfully that all calls be held until the conclusion of the interview.

- Step 5:Conduct the Interview :-
  After determining the people to interview, setting your objectives, and preparing the questions, you should develop a specific plan for the meeting. When conducting an interview, you should begin by introducing yourself, describing the project, and explaining your interview objectives. During the interview, ask questions in the order in which you prepared them, and give the interviewee sufficient time to provide thoughtful answers. Establishing a good rapport with the interviewee is important, especially if this is your first meeting. If the other person feels comfortable and at ease, you probably will receive more complete and candid answers. Your primary responsibility during an interview is to listen carefully to the answers. Analysts sometimes hear only what they expect to hear. You must concentrate on what is said and notice any nonverbal communication that takes place. This process is called engaged listening. After asking a question, allow the person enough time to think about the question and arrive at an answer. Studies have shown that the maximum pause during a conversation is usually three to five seconds. After that interval, one person will begin talking. You will need to be patient and practice your skills in many actual interview situations to be successful. When you finish asking your questions, summarize the main points covered in the interview and explain the next course of action. For example, mention that you will send a follow-up memo or that the interviewee should get back to you with certain information. When you conclude the interview, thank the person and encourage him or her to contact you with any questions or additional comments. Also, when the interview ends, it is a good idea to ask the interviewee whether he or she can suggest any additional topics that should be discussed. After an interview, you should summarize the session and seek a confirmation from the other person. By stating your understanding of the discussion, the interviewee can respond and correct you, if necessary. One good approach is to rephrase the intervierwee's answers. For example, you can say, "If I understand you correctly, you are saying that ..." and then reiterate the information given by the interviewee.
- step 6:Document the Interview :-Although taking notes during an interview has both advantages and disadvantages, the accepted view is that note taking should be kept to a minimum. Although you should write down a few notes to jog your memory after the interview, you should avoid writing everything that is said. Too much writing distracts the other person and makes it harder to establish a good rapport. After conducting the interview, you must record the information quickly. You should set aside time right after the meeting to record the facts and

evaluate the information. For that reason, try not to schedule back-to-back interviews. Studies have shown that SO percent of a conversation is forgotten within 30 minutes. You, there- fore, should use your notes to record the facts immediately so you will not forgetthem. You can summarize the facts by preparing a narrative describing what took place or by recording the answers you received next to each question on your prepared question list. Tape recorders are effective tools for an interview; however, many people feel uncomfortable when recorders are present. Before using a recorder, you should discuss its use with the interviewee. Assure the interviewee that you will erase the tape after you transcribe your notes and that you will stop and rewind the tape anytime during the interview at his or her request. If you ask sensitive questions or the interviewee wants to answer a question without being recorded, explain that you will turn off the tape for a period of time during the interview. Even with a tape recorder in use, you should listen carefully to the interviewee's responses so you can ask good follow-up questions. Otherwise, you might have to return for a second visit to ask the questions you missed the first time. Also, remember that each recorded interview takes twice the amount of time, because you must listen to or view the recorded meeting again after conducting the interview itself. After the interview, send a memo to the interviewee expressing your appreciation for his or her time and cooperation. In the memo, you should note the date, time, location, purpose of the interview, and the main points you discussed so the interviewee has a written summary and can offer additions or corrections.

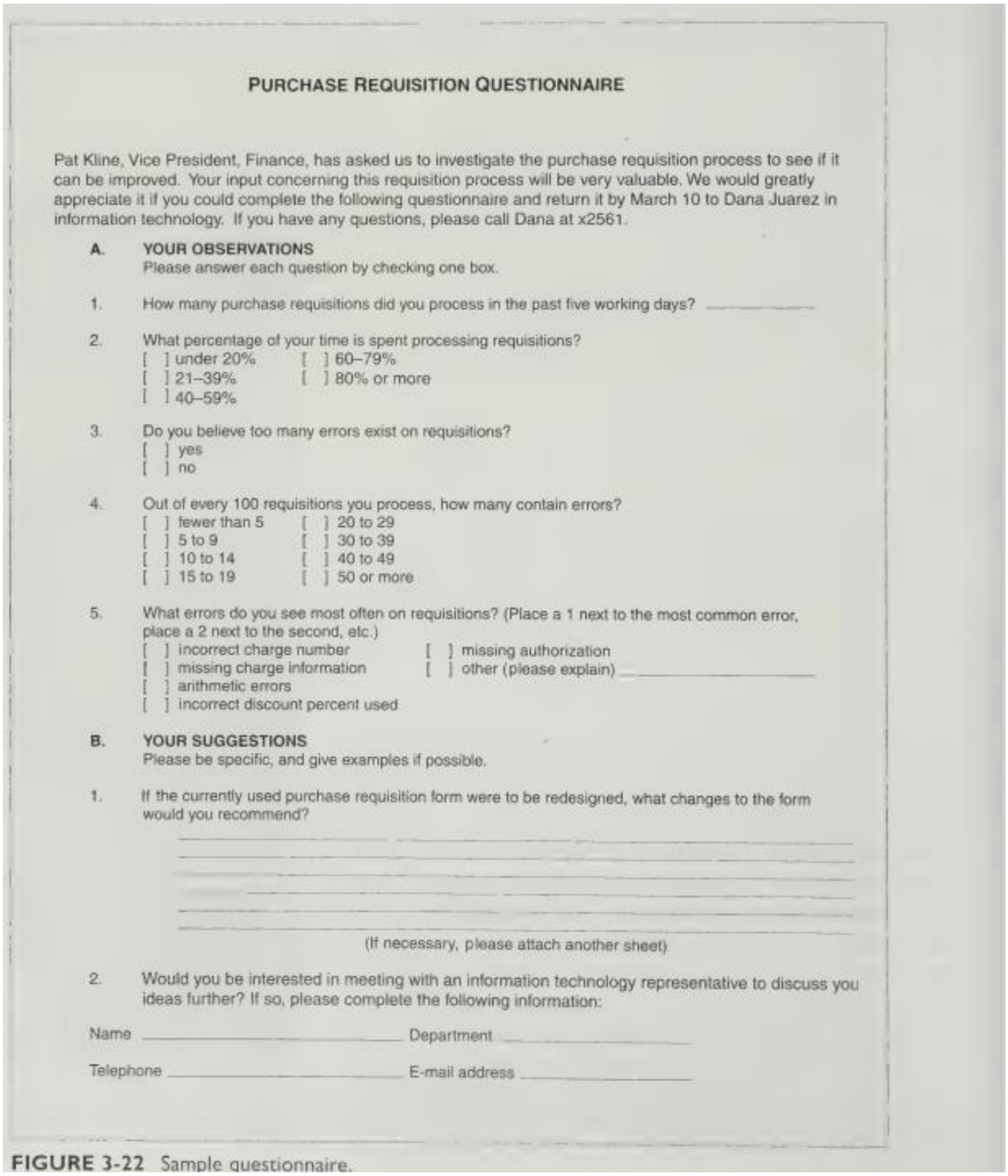
- Step 7: Evaluate the Interview:- In addition to recording the facts obtained in an interview, try to identify any possible biases. For example, an interviewee who tries to protect his or her own area or function might give incomplete answers or refrain from volunteering information. Or, an interviewee with strong opinions about the current or future system might distort the facts. Some interviewees might answer your questions in an attempt to be helpful even though they do not have the necessary experience to provide accurate information.

- Document Review:- Document review can help you understand how the current system is supposed to work. Remember that system documentation sometimes is out of date. Forms can change or be discontinued, and documented procedures often are modified or eliminated. You should obtain copies of actual forms and operating documents currently in use. You also should review blank copies of forms, as well as samples of actual com- pleted forms. You usually can obtain document samples during interviews with the peo- ple who perform that procedure. If the system uses a software package, you should review the documentation for that software.

- Observation:- The observation of current operating procedures is another fact-finding technique. Seeing the system in action gives you additional perspective and a better understanding of system procedures. Personal observation also allows you to verify statements made in interviews and determine whether procedures really operate as they are described. Through observation, you might discover that neither the system documentation nor the interview statements are accurate. Personal observation also can provide important advantages as the development process continues. For example, recommendations often are better accepted when they are based on personal observation of actual operations. Observation also can provide the knowledge needed to test or install future changes and can help build relationships with the users who will work with the new system. Plan your observations in advance by preparing a checklist of specific tasks you want to observe and questions you want to ask. Consider the following issues when you prepare your list: 1. Ask sufficient questions to ensure that you have a complete understanding of the present system operation. A primary goal is to identify the methods of handling situations that are not covered by standard operating procedures. For example, what happens in a payroll system if an employee loses a time card? What is the procedure if an employee starts a shift 10 minutes late but then works 20 minutes overtime? Often, the rules for exceptions such as these are not written or formalized; therefore, you must try to document any procedures for handling exceptions. 2. Observe all the steps in a transaction and note the documents, inputs, outputs, and processes involved. 3. Examine each form, record, and report. Determine the purpose each item of information serves. 4. Consider each user who works with the system and the following questions: What information does that person receive from other people? What information does this person generate? How is the information communicated? How often do interruptions occur? How much downtime occurs? How much support does the user require, and who provides it? 5. Talk to the people who receive current reports to see whether the reports are complete, timely, accurate, and in a useful form. Ask whether information can be eliminated or improved and whether people would like to receive additional information. As you observe people at work, as shown in Figure 3-21, consider a factor called the Hawthorne Effect. The name comes from a well-known study performed in the Hawthorne plant of the Western Electric Company in the 1920s. The purpose of the study was to determine how various changes in the work environment would affect employee productivity. The surprising result was that productivity improved during observation whether the conditions were made better or worse. Researchers concluded that productivity seemed to improve whenever the workers knew they were being observed. Thus, as you observe users, remember that normal operations might not always run as smoothly as your observations indicate. Operations also might run less smoothly because workers might be nervous during the observation. If possible, meet with workers and their supervisors to discuss your plans and objectives to help establish a good working relationship. In some situations, you might even participate in the work yourself to gain a personal understanding of the task or the environment.

FIGURE 3-21 The Hawthorne study suggested that worker productivity improves during observation. Always consider the Hawthorne Effect when observing the operation of an existing system.

- Questionnaires and Surveys:- In projects where it is desirable to obtain input from a large number of people, a questionnaire can be a valuable tool. A questionnaire, also called a survey, is a document containing a number of standard questions that can be sent to many individuals. Questionnaires can be used to obtain information about a wide range of topics, including workloads, reports received, volumes of transactions handled, job duties, difficulties, and opinions of how the job could be performed better or more efficiently. Figure 3-22 shows a sample questionnaire that includes several different question and response formats.



PURCHASE REQUISITION QUESTIONNAIRE

Pat Kline, Vice President, Finance, has asked us to investigate the purchase requisition process to see if it can be improved. Your input concerning this requisition process will be very valuable. We would greatly appreciate it if you could complete the following questionnaire and return it by March 10 to Dana Juarez in information technology. If you have any questions, please call Dana at x2561.

A. YOUR OBSERVATIONS
Please answer each question by checking one box.

1. How many purchase requisitions did you process in the past five working days? ____________

2. What percentage of your time is spent processing requisitions?
[ ] under 20% [ ] 60–79%
[ ] 21–39% [ ] 80% or more
[ ] 40–59%

3. Do you believe too many errors exist on requisitions?
[ ] yes
[ ] no

4. Out of every 100 requisitions you process, how many contain errors?
[ ] fewer than 5 [ ] 20 to 29
[ ] 5 to 9 [ ] 30 to 39
[ ] 10 to 14 [ ] 40 to 49
[ ] 15 to 19 [ ] 50 or more

5. What errors do you see most often on requisitions? (Place a 1 next to the most common error, place a 2 next to the second, etc.)
[ ] incorrect charge number [ ] missing authorization
[ ] missing charge information [ ] other (please explain) ____________
[ ] arithmetic errors
[ ] incorrect discount percent used

B. YOUR SUGGESTIONS
Please be specific, and give examples if possible.

1. If the currently used purchase requisition form were to be redesigned, what changes to the form would you recommend?
____
____
____
____
____
____
(If necessary, please attach another sheet)

2. Would you be interested in meeting with an information technology representative to discuss you ideas further? If so, please complete the following information:

Name ____________ Department ____________

Telephone ____________ E-mail address ____________

FIGURE 3-22 Sample questionnaire.

A typical questionnaire starts with a heading, which includes a title, a brief statement of purpose, the name and telephone number of the contact person, the deadline date for completion, and how and where to return the form. The heading usually is followed by general instructions that provide clear guidance on how to answer the questions. Headings also are used to introduce each main section or portion of the survey and include instructions when the type of question or response changes. A long questionnaire might end with a conclusion that thanks the participants and reminds them how to return the form. What about the issue of anonymity? Should people be asked to sign the questionnaire, or is it better to allow anonymous responses? The answer depends on two questions. First, does an analyst really need to know who the respondents are in order to match or correlate information? For example, it might be important to know what percentage of users need a certain software feature, but specific user names might not be relevant. Second, does the questionnaire include any sensitive or controversial topics? Many people do not want to be identified when answering a question such as "How well has your supervisor explained the system to you?" In such cases, anonymous responses might provide better information. When designing a questionnaire, the most important rule of all is to make sure that your questions collect the right data in a form that you can use to further your fact-finding. Here are some additional ideas to keep in mind when designing your questionnaire:

- ➔ Keep the questionnaire brief and user-friendly.
- ➔ Keep the questionnaire brief and user-friendly.
- ➔ Arrange the questions in a logical order, going from simple to more complex topics.
- ➔ Phrase questions to avoid misunderstandings; use simple terms and wording.
- ➔ 'Try not to lead the response or use questions that give clues to expected answers.
- ➔ Limit the use of open-ended questions that are difficult to tabulate.
- ➔ Limit the use of questions that can raise concerns about job security or other negative issues.
- ➔ Include a section at the end of the questionnaire for general comments.
- ➔ Test the questionnaire whenever possible on a small test group before finalizing it and distributing to a large group.

A questionnaire can be a traditional paper form, or you can create a fill-in form and collect data on the sInternet or a company intranet. For example, you can use Microsoft Word, as shown in Figure 3-23 on the next page, to create form fields, including text boxes, check boxes, and drop-down lists where users can click selections. Before you publish the form, you should protect it so users can fill it in but cannot change the layout or design. Forms also can be automated, so if a user answers no to question three, he or she goes directly to question eight, where the form-filling resumes.

- OVERVIEW OF FEASIBILITY Study:-

A systems request must pass several tests, called a feasibility study, to see whether it is worthwhile to proceed further. As shown in Figure 2-14, a feasibility study uses four main yardsticks to measure a proposal: operational feasibility, technical feasibility, economic feasibility, and schedule feasibility. Sometimes a feasibility study is quite simple and can be done in a few hours. If the request involves a new system or a major change, however, extensive fact-finding and investigation is required. How much effort needs to go into a feasibility study? That depends on the request. For example, if a department wants an existing report sorted in a different order, the analyst can decide quickly whether the request is feasible. On the other hand, a proposal by the marketing department for a new market research system to predict sales trends requires more effort. In both cases, the systems analyst asks these important questions:

- Is the proposal desirable in an operational sense? Is it a practical approach that will solve a problem or take advantage of an opportunity to achieve company goals?
- Is the proposal technically feasible? Are the necessary technical resources and people available for the project?
- Is the proposal economically desirable? What are the projected savings and costs? Are other intangible factors involved, such as customer satisfaction or company image? Is the problem worth solving, and will the request result in a sound business investment?
- Can the proposal be accomplished within an acceptable time frame?

To obtain more information about a systems request, you might perform initial fact-finding by studying organization charts, performing interviews, reviewing current documentation, observing operations, and surveying users. If the systems request is approved, more intensive fact-finding will continue during the systems analysis phase.

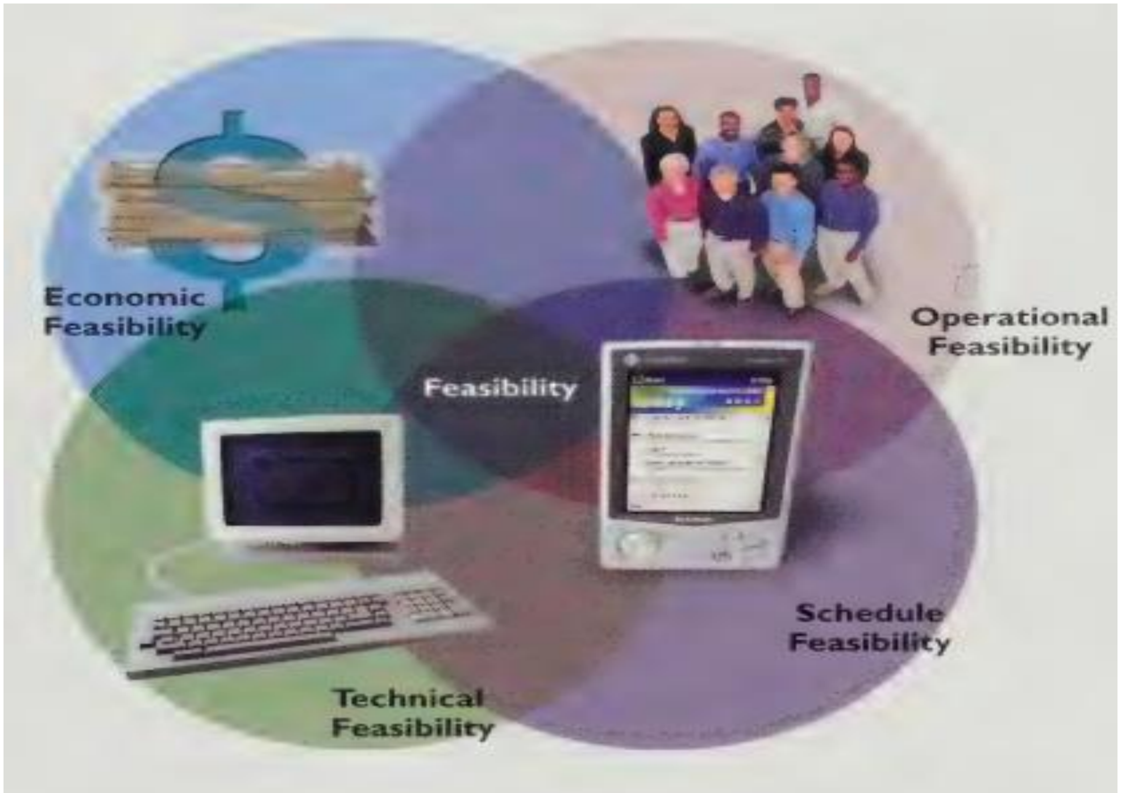


FIGURE 2-14 A feasibility study includes tests for operational, technical, economic, and schedule feasibility.

- Operational Feasibility
  Operational feasibility means that a proposed system will be used effectively after it has been developed. If users have difficulty with a new system, it will not produce the expected benefits. Operational feasibility depends on several vital issues. For example, consider the following questions:

- ➔ Does management support the project? Do users support the project? Is the current system well liked and effectively used? Do users see the need for change?
- ➔ Will the new system result in a workforce reduction? If so, what will happen to affected employees?
- ➔ Will the new system require training for users? If so, is the company prepared to provide the necessary resources for training current employees?
- ➔ Will users be involved in planning the new system right from the start?
- ➔ Will the new system place any new demands on users or require any operating changes? For example, will any information be less accessible or produced less frequently? Will performance decline in any way? If so, will an overall gain to the organization outweigh individual losses?
- ➔ Will customers experience adverse effects in any way, either temporarily or permanently?
- ➔ Will any risk to the company's image or goodwill result?
- ➔ Does the development schedule conflict with other company priorities?
- ➔ Do legal or ethical issues need to be considered?

- Technical Feasibility
  Technical feasibility refers to the technical resources needed to develop, purchase, install, or operate the system. When assessing technical feasibility, an analyst must con- sider the following points:

- ➔ Does the company have the necessary hardware, software, and network resources? If not, can those resources be acquired without difficulty?
- ➔ Does the company have the needed technical expertise? If not, can it be acquired?
- ➔ Does the proposed platform have sufficient capacity for future needs? If not, can it be expanded?
- ➔ Will a prototype be required?
- ➔ Will the hardware and software environment be reliable? Will it integrate with other company information systems, both now and in the future? Will it interface properly with external systems operated by customers and suppliers?
- ➔ Will the combination of hardware and software supply adequate performance? Do clear expectations and performance specifications exist?
- ➔ Will the system be able to handle future transaction volume and company growth?

- Economic Feasibility

Economic feasibility means that the projected benefits of the proposed system outweigh the estimated costs usually considered the total cost of ownership (TCO), which includes ongoing support and maintenance costs, as well as acquisition costs. To determine TCO, the analyst must estimate costs in each of the following areas:

- ➔ People, including IT staff and users
- ➔ Hardware and equipment
- ➔ Software, including in-house development as well as purchases from vendors
- ➔ Formal and informal training
- ➔ Licenses and fees
- ➔ Consulting expenses
- ➔ Facility costs
- ➔ The estimated cost of not developing the system or postponing the project

In addition to costs, you need to assess tangible and intangible benefits to the company. The systems review committee will use those figures, along with your cost estimates, to decide whether to pursue the project beyond the preliminary investigation phase. Tangible benefits are benefits that can be measured in dollars. Tangible benefits result from a decrease in expenses, an increase in revenues, or both. Examples of tangible benefits include the following:

- ➔ A new scheduling system that reduces overtime
- ➔ An online package tracking system that improves service and decreases the need for clerical staff
- ➔ A sophisticated inventory control system that cuts excess inventory and eliminates production delays

Intangible benefits are advantages that are difficult to measure in dollars but are important to the company. Examples of intangible benefits include the following:

- ➔ A user-friendly system that improves employee job satisfaction
- ➔ A sales tracking system that supplies better information for marketing decisions
- ➔ A new Web site that enhances the company's image

You also must consider the development timetable, because some benefits might occur as soon as the system is operational, but others might not take place until later.

- Schedule feasibility
  Schedule feasibility means that a project can be implemented in an acceptable time frame. When assessing schedule feasibility, a systems analyst must consider the interaction between time and costs. For example, speeding up a project schedule might make a project feasible, but much more expensive. Other issues that relate to schedule feasibility include the following:

➔ Can the company or the IT team control the factors that. affect schedule feasibility?
➔ Has management established a firm timetable for the project?
➔ What conditions must be satisfied during the development of the system?
➔ Will an accelerated schedule pose any risks? If so, are the risks acceptable?
➔ Will project management techniques be available to coordinate and control the project?
➔ Will a project manager be appointed?

- Data FLow DIAGRAMS:
  In Part 1 of the Systems Analyst's Toolkit, you learn how to use visual aids to help explain a concept, as shown in Figure 4-2. Similarly, during the systems analysis phase, you learn how to create a visual model of the information system using a set of data flow diagrams. A data flow diagram (DFD) shows how data moves through an information system but does not show program logic or processing steps. A set of DFDs provides a logical model that shows what the system does, not how it does it. That distinction is important because focusing on implementation issues at this point would restrict your search for the most effective system design.

- DFD Symbols:-
  DFDs use four basic symbols that represent processes, data flows, data stores, and entities. Several different versions of DFD symbols exist, but they all serve the same purpose. DFD examples in this textbook use the Gane and Sarson symbol set. Another popular symbol set is the Yourdon symbol set. Figure 4-3 shows examples of both versions. Symbols are referenced by using all capital letters for the symbol name.

- PROCESS SYMBOL:-
  A process receives input data and produces output that has a different content, form, or both. For instance, the process for calculating pay uses two inputs (pay rate and hours worked) to produce one output (total pay). Processes can be very simple or quite complex. In a typical company, processes might include calculating sales trends, filing online insurance claims, ordering inventory from a supplier's system, or verifying e-mail addresses for Web customers. Processes contain the business logic, also called business rules, that transform the data and produce the required results. The symbol for a process is a rectangle with rounded corners. The name of the process appears inside the rectangle. The process name identifies a specific function and consists of a verb (and an adjective, if necessary) followed by a singular noun. Examples of process names are APPLY RENT PAYMENT, CALCULATE COMMISSION, ASSIGN FINAL GRADE, VERIFY ORDER, and FILL ORDER. Processing details are not shown in a DED. For example, you might have a process named DEPOSIT PAYMENT. The process symbol does not reveal the business logic for the DEPOSIT PAYMENT process. To document the logic, you create a process description, which is explained later in this chapter.

FIGURE 4-2 Systems analysts often use visual aids during presentations.

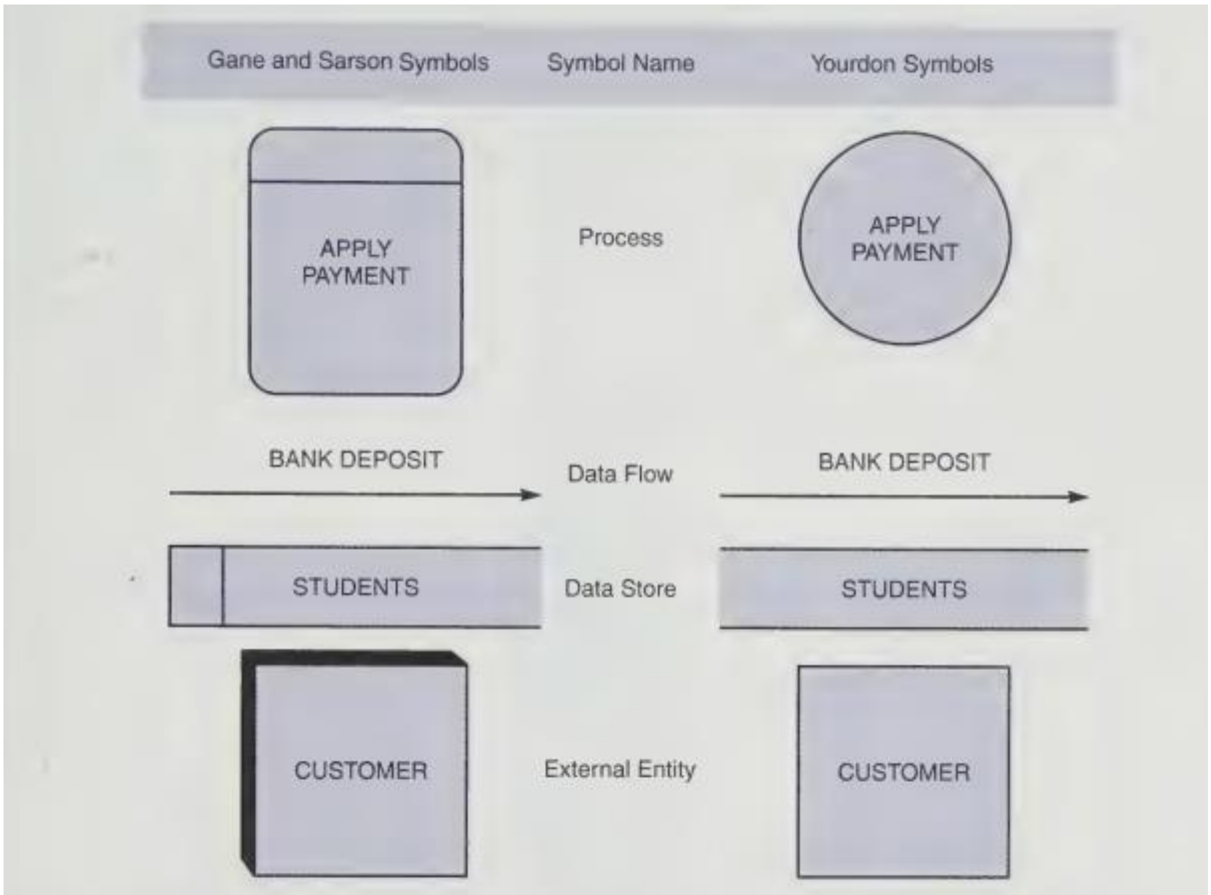


FIGURE 4-3 Data flow diagram symbols, symbol names, and examples of the Gane and Sarson and Yourdon symbol sets.

In DFDs, a process symbol can be referred to as a black box, because the inputs, outputs, and general functions of the process are known, but the underlying details and logic of the process are hidden. By showing processes as black boxes, an analyst can create DFDs that show how the system functions, but avoid unnecessary detail and clutter. When the analyst wishes to show additional levels of detail, he or she can zoom in on a process symbol and create a more in-depth DFD that shows the process's internal workings — which might reveal even more processes, data flows, and data stores. In this manner, the information system can be modeled as a series of increasingly detailed pictures. The network router shown in Figure 4-4 is an example of a black box. An observer can see cables that carry data into and out of the router, but the router's internal operations are not revealed — only the results are apparent.

FIGURE 4-4 Networks use various devices that act like black boxes. Cables carry data in and out, but internal operations are hidden inside the case.

- DATA FLOW SYMBOL
  A data flow is a path for data to move from one part of the information system to another. A data flow in a DFD represents one or more data items. For example, a data flow could consist of a single data item (such as a student ID number) or it could include a set of data (such as a class roster with student ID numbers, names, and registration dates for a specific class). Although the DFD does not show the detailed contents of a data flow, that information is included in the data dictionary, which is described later in this chapter. The symbol for a data flow is a line with a single or double arrowhead. The data flow name appears above, below, or alongside the line. A

data flow name consists of a singular noun and an adjective, if needed. Examples of data flow names are DEPOSIT, INVOICE PAYMENT, STUDENT GRADE, ORDER, and COMMISSION. Exceptions to the singular name rule are data flow names, such as GRADING PARAMETERS, where a singular name could mislead you into thinking a single parameter or single item of data exists. Figure 4-5 shows correct examples of data flow and process symbol connections. Because a process changes the data's content or form, at least one data flow must enter and one data flow must exit each process symbol, as they do in the CREATE INVOICE process. A process symbol can have more than one outgoing data flow, as shown in the GRADE STUDENT WORK process, or more than one incoming data flow, as shown in the CALCULATE GROSS PAY process. A process also can connect to any other symbol, including another process symbol, as shown by the connection between VERIFY ORDER and ASSEMBLE ORDER in Figure 4-5. A data flow, therefore, must have a process symbol on at least one end.

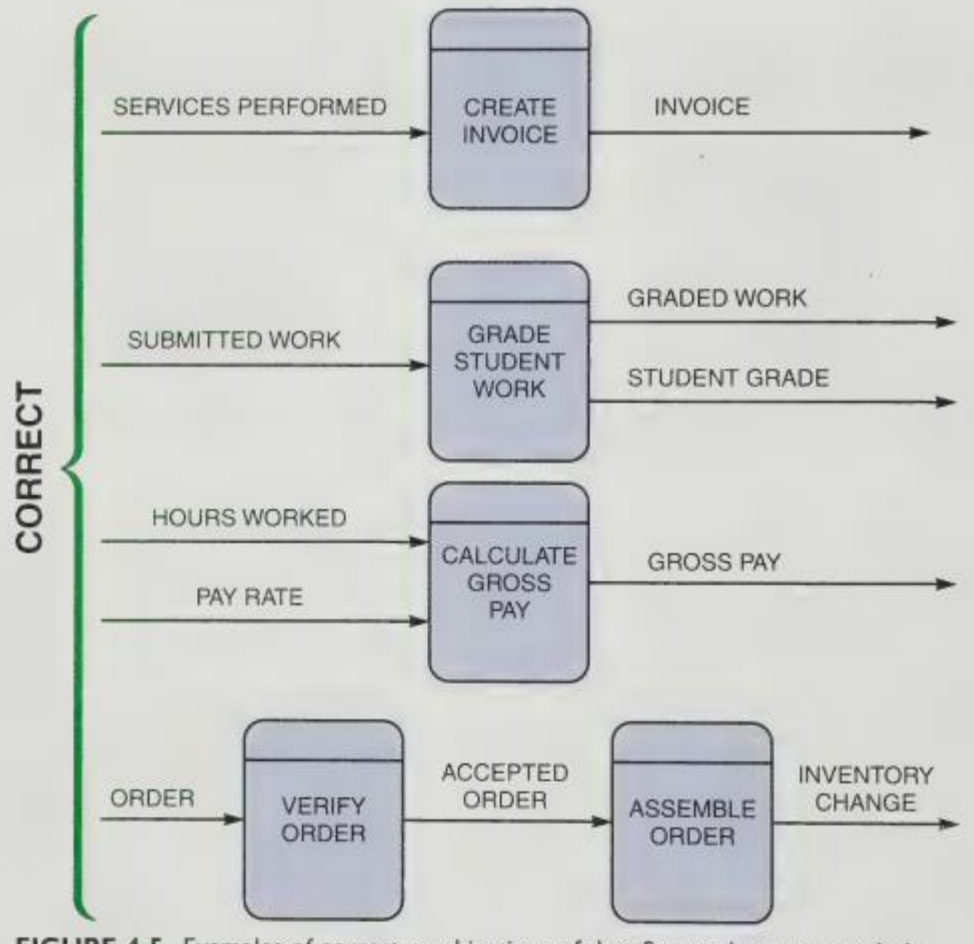


FIGURE 4-5 Examples of correct combinations of data flow and process symbols.

Figure 4-6 shows three data flow and process combinations that you must avoid:

- ➔ Spontaneous generation: The APPLY INSURANCE PREMIUM process, for instance, produces output, but has no input data flow. Because it has no input, the process is called a spontaneous generation process.
- ➔ Black hole: The CALCULATE GROSS PAY is called a black hole process, which is a process that has input, but produces no output.
- ➔ Gray hole: A gray hole is a process that has at least one input and one output, but the input obviously is insufficient to generate the output shown. For exam- ple, a date of birth input is not sufficient to produce a final grade output in the CALCULATE GRADE process.

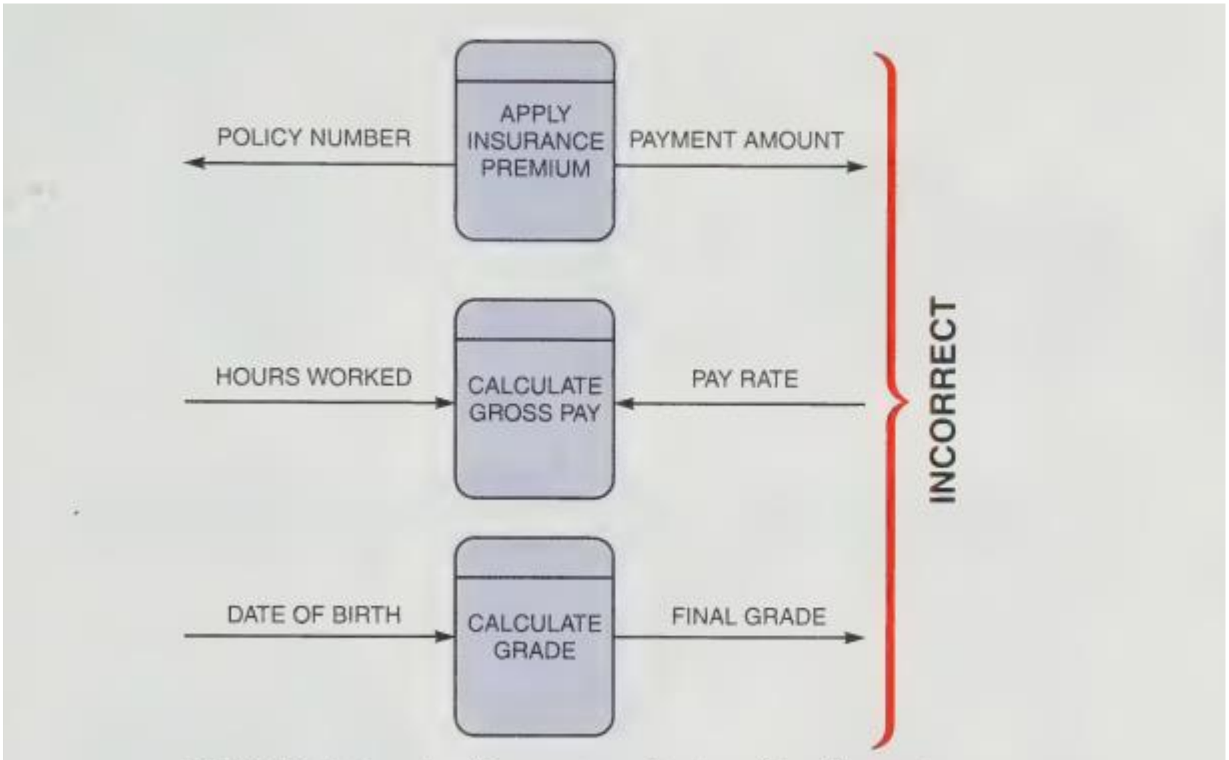


FIGURE 4-6 Examples of incorrect combinations of data flow and process symbols. APPLY INSURANCE PREMIUM has no input and is called a spontaneous generation process. CALCULATE GROSS PAY has no outputs and is called a black hole process. CALCULATE GRADE has an input that is obviously unable to produce the output. This process is called a gray hole.

Spontaneous generation, black holes, and gray holes are impossible logically in a DED because a process must act on input, shown by an incoming data flow, and produce output, represented by an outgoing data flow.

- DATA STORE SYMBOL

A data store is used in a DFD to represent data that the system stores because one or more processes need to use the data at a later time. For instance, instructors need to store student scores on tests and assignments during the semester so they can assign final grades at the end of the term. Similarly, a company stores employee salary and deduction data during the year in order to print W-2 forms with total earnings and deductions at the end of the year. A DFD does not show the detailed contents of a data store — the specific structure and data elements are defined in the data dictionary, which is discussed later in this chapter. The physical characteristics of a data store are unimportant because you are concerned only with a logical model. Also, the length of time that the data is stored is unimportant — it can be a matter of seconds while a transaction is processed or a period of months while data is accumulated for year-end processing. What is important is that a process needs access to the data at some later time. In a DFD, the Gane and Sarson symbol for a data store is a flat rectangle that is open on the right side and closed on the left side. The name of the data store appears between the lines and identifies the data it contains. A data store name is a plural name consisting of a noun and adjectives, if needed. Examples of data store names are STUDENTS, ACCOUNTS RECEIVABLE, PRODUCTS, DAILY PAYMENTS, PURCHASE ORDERS, OUTSTANDING CHECKS, INSURANCE POLICIES, and EMPLOYEES. Exceptions to the plural name rule are collective nouns that represent multiple occurrences of objects. For example, GRADEBOOK represents a group of students and their scores. A data store must be connected to a process with a data flow. Figure 4-7 illustrates typical examples of data stores. In each case, the data store has at least one incoming and one outgoing data flow and is connected to a process symbol with a data flow.

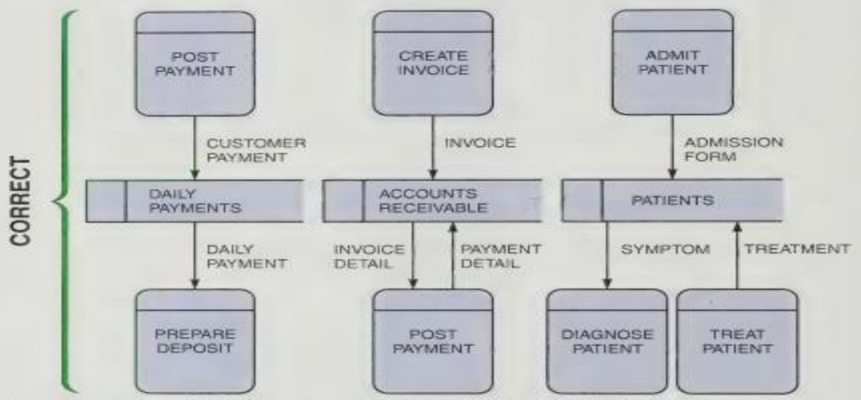


FIGURE 4-7 Examples of correct uses of data store symbols in a data flow diagram.

Violations of the rule that a data store must have at least one incoming and one outgoing data flow are shown in Figure 4-8. In the first example, two data stores are connected incorrectly because no process is between them. Also, COURSES has no incoming data flow and STUDENTS has no outgoing data flow. In the second and third examples, the data stores lack either an outgoing or incoming data flow.

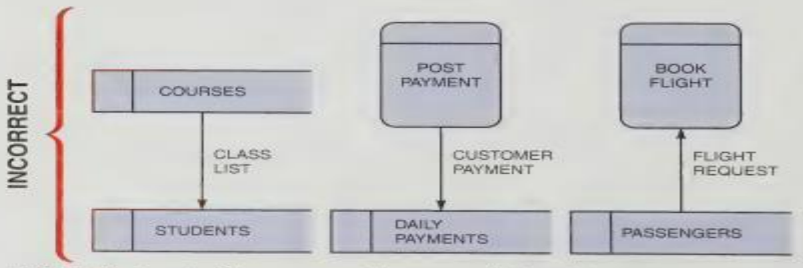


FIGURE 4-8 Examples of incorrect uses of data store symbols: two data stores cannot be connected by a data flow without an intervening process, and each data store should have an outgoing and incoming data flow.

There is an exception to the requirement that a data store must have at least one incoming and one outgoing data flow. In some situations, a data store has no input data flow because it contains fixed reference data that is not updated by the system. For example, consider a data store called TAX TABLE, which contains withholding tax data that a company downloads from the Internal Revenue Service. When the company runs its payroll, the CALCULATE WITHHOLDING process accesses data from this data store. On a DFD, this would be represented as a one-way outgoing data flow from the TAX TABLE data store into the CALCULATE WITHHOLDING process.

- ENTITY SYMBOL
  The symbol for an entity is a rectangle, which may be shaded to make it look threedimensional. The name of the entity appears inside the symbol. A DED shows only external entities that provide data to the system or receive output from the system. A DFD shows the boundaries of the system and how the system interfaces with the outside world. For example, a customer entity submits an order to an order processing system. Other examples of entities include a patient who supplies data to a medical records system, a homeowner who receives a bill from a city property tax system, or an accounts payable system that receives data from the company's purchasing system. DFD entities also are called terminators, because they are data origins or final destinations. Systems analysts call an entity that supplies data to the system a source, and an entity that receives data from the system a sink. An entity name is the singular form of a department, outside organization, other information system, or person. An external entity can be a source or a sink or both, but each entity must be connected to a process by a data flow. Figures 4-9 and 4-10 show correct and incorrect examples of this rule.
  With an understanding of the proper use of DFD symbols, you are ready to construct diagrams that use these symbols. Figure 4-11 on the next page shows a summary of the rules for using DFD symbols.

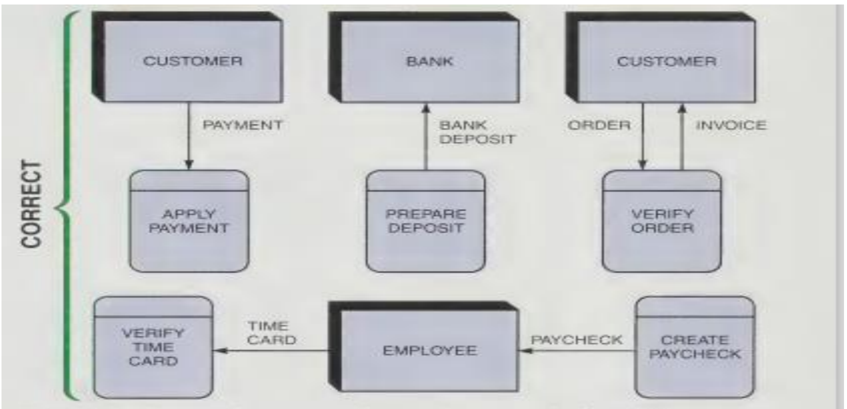


FIGURE 4-9 Examples of correct uses of external entities in a data flow diagram.

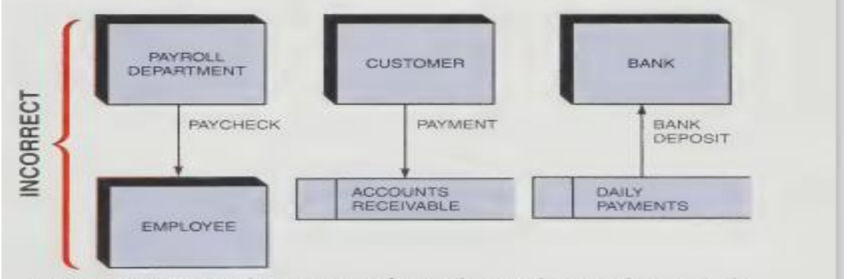
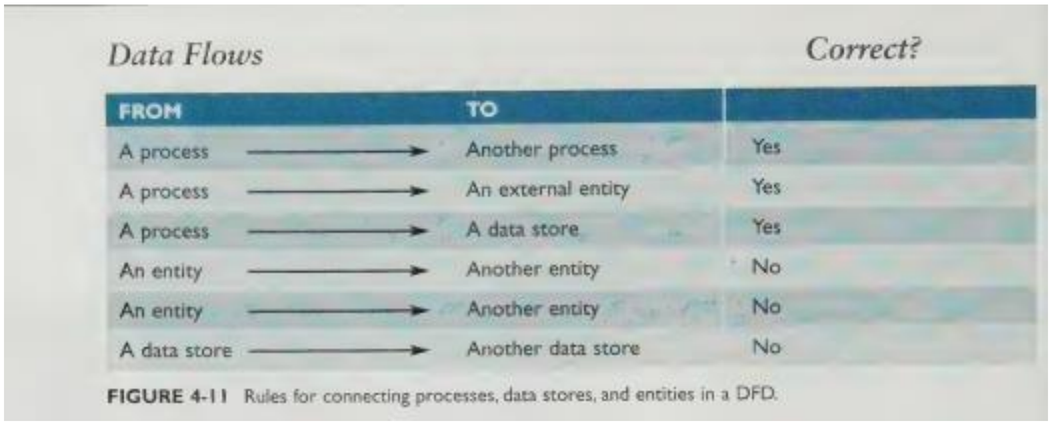


FIGURE 4-10 Examples of incorrect uses of external entities. An external entity must be connected by a data flow to a process, and not directly to a data store or to another external entity.



Data Flows *Correct?*

FROM	TO	
A process →	Another process	Yes
A process →	An external entity	Yes
A process →	A data store	Yes
An entity →	Another entity	No
An entity →	Another entity	No
A data store →	Another data store	No

FIGURE 4-11 Rules for connecting processes, data stores, and entities in a DFD.

- Construction Context Diagram :

  Phase 2 Systems Analysis 157 The first step in constructing a set of DFDs is to draw a context diagram. A context diagram is a top-level view of an information system that shows the system's boundaries and scope. To draw a context diagram, you start by placing a single process symbol in the center of the page. The symbol represents the entire information system, and you identify it as process 0 (the numeral zero, and not the letter O). Then you place the system entities around the perimeter of the page and use data flows to connect the entities to the central process. Data stores are not shown in the context diagram because they are contained within the system and remain hidden until more detailed diagrams are created. How do you know which entities and data flows to place in the context diagram? You begin by reviewing the system requirements to identify all external data sources and destinations. During that process, you identify the entities, the name and content of the data flows, and the direction of the data flows. If you do that carefully, and you did a good job of fact-finding in the previous stage, you should have no difficulty drawing the context diagram. Now review the following context diagram examples.

  EXAMPLE: CONTEXT DIAGRAM FORA GRADING SYSTEM The context diagram for a grading system is shown in Figure 4-12 on the previous page. The GRADING SYSTEM process is at the center of the diagram. The three entities (STUDENT RECORDS SYSTEM, STUDENT, and INSTRUCTOR) are placed around the central process. Interaction among the central process and the entities involves six different data flows. The STUDENT RECORDS SYSTEM entity supplies data through the CLASS ROSTER data flow and receives data through the FINAL GRADE data flow. The STUDENT entity supplies data through the SUBMITTED WORK data flow and receives data through the GRADED WORK data flow. Finally, the INSTRUCTOR entity supplies data through the GRADING PARAMETERS data flow and receives data through the GRADE REPORT data flow.

  EXAMPLE: CONTEXT DIAGRAM FORAN ORDER SYSTEM The context diagram for an order system is shown in Figure 4-13. Notice that the ORDER SYSTEM process is at the center of the diagram and five entities surround the process. Three of the entities, SALES REP, BANK, and ACCOUNTING, have single incoming data flows for COMMISSION, BANK DEPOSIT, and CASH RECEIPTS ENTRY, respectively. The WAREHOUSE entity has one incoming data flow — PICKING

LIST — that is, a report that shows the items ordered and their quantity, location, and sequence to pick from the warehouse. The WAREHOUSE entity has one outgoing data flow, COMPLETED ORDER. Finally, the CUSTOMER entity has two outgoing data flows, ORDER and PAYMENT, and two incoming data flows, ORDER REJECT NOTICE and INVOICE.

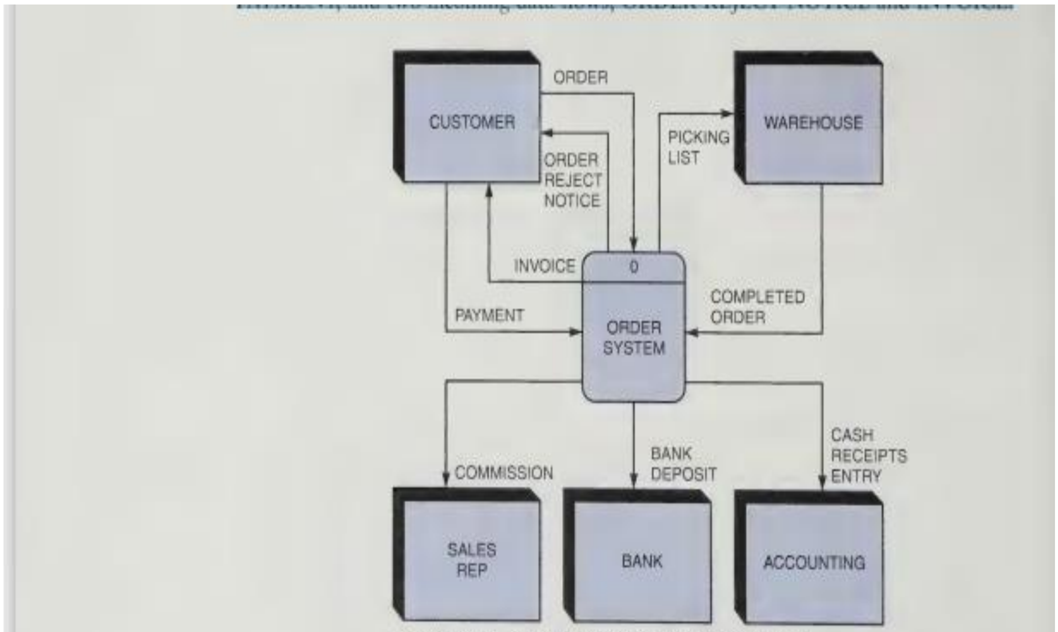


FIGURE 4-13 Context diagram DFD for an order system.

e context diagram for the order system appears more complex than the grading system because it has two more entities and three more data flows. What makes one system more complex than another is the number of components, the number of levels, and the degree of interaction among its processes, entities, data stores, and data flows.

The real-life scene in Figure 4-14 represents a complex manufacturing system with many interactive processes and data. In a large system such as this, each process in diagram 0 could represent a separate system such as inventory, production control, and scheduling. Diagram 0 provides an overview of all the components that interact to form the overall system. Now review the following diagram 0 examples.

FIGURE 4-14 Complex manufacturing systems require many interactive processes and data sources.

EXAMPLE: DIAGRAM 0 DFD FOR A GRADING SYSTEM Figure 4-15 on the next page shows a context diagram at the top and diagram 0 beneath it. Notice that diagram 0 is an expansion of process 0. Also notice that the three same entities (STUDENT RECORDS SYSTEM, STUDENT, and INSTRUCTOR) and the same six data flows (FINAL GRADE, CLASS ROSTER, SUBMITTED WORK, GRADED WORK, GRADING PARAMETERS, and GRADE REPORT) appear in both diagrams. In addition, diagram 0 expands process 0 to reveal four internal processes, one data store, and five additional data flows. Notice that each process in diagram 0 has a reference number: ESTABLISH GRADEBOOK is 1, ASSIGN FINAL GRADE is 2, GRADE STUDENT WORK is 3, and PRODUCE GRADE REPORT is 4. These reference numbers are important because they identify a series of DFDs. If more detail were needed for ESTABLISH GRADEBOOK, for example, you would draw a diagram 1, because ESTABLISH GRADEBOOK is process 1. The process numbers do not suggest that the processes are accomplished in a sequential order. Each process always is considered to be available, active, and awaiting data to be processed. If processes must be performed in a specific sequence, you document the information in the process descriptions (discussed later in this chapter), not in the DFD.

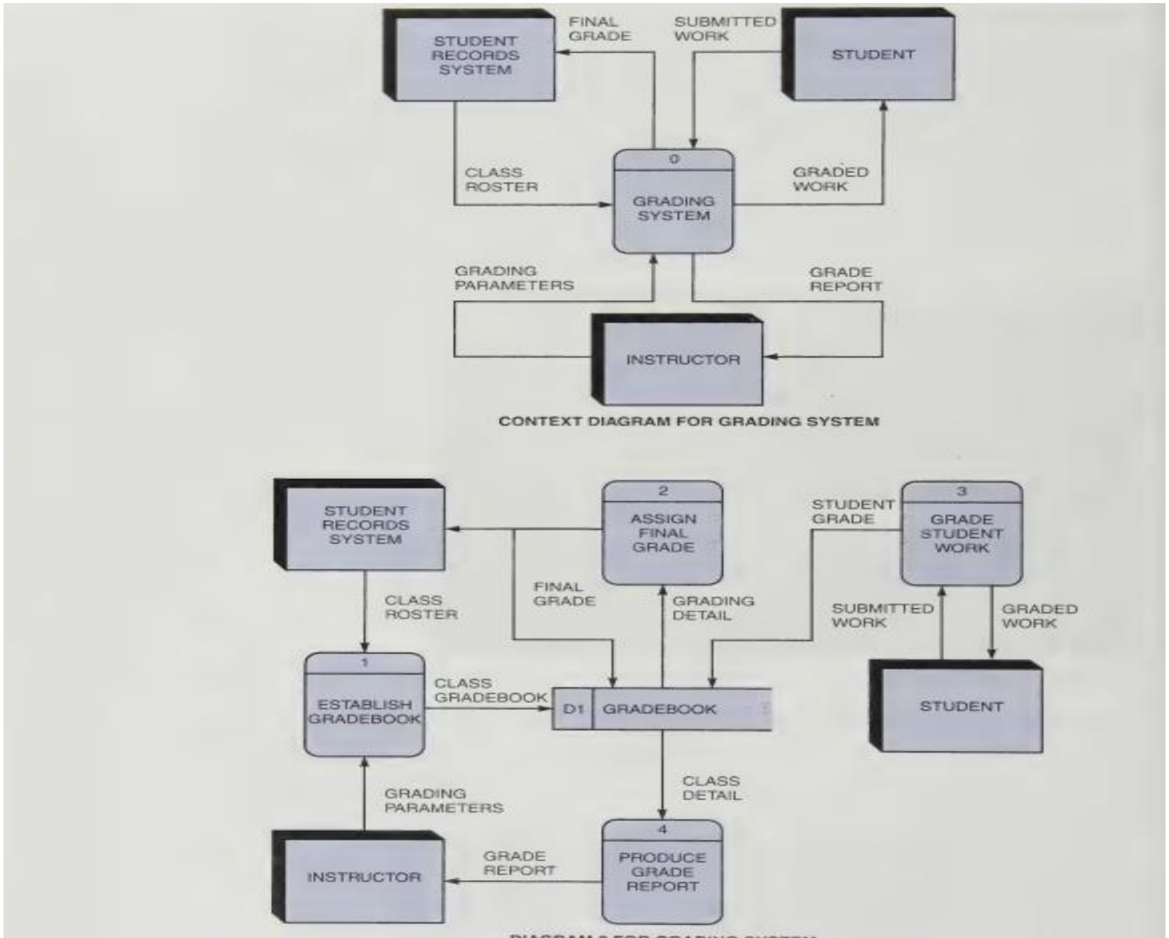


FIGURE 4-15 Context diagram and diagram 0 for the grading system.

The FINAL GRADE data flow output from the ASSIGN FINAL GRADE process is a diverging data flow that becomes an input to the STUDENT RECORDS SYSTEM entity and to the GRADEBOOK data store. A diverging data flow is a data flow in which the same data travels to two or more different locations. In that situation, a diverging data flow is the best way to show the flow rather than showing two identical data flows, which could be misleading. If the same data flows in both directions, you can use a double-headed arrow to connect the symbols. To identify specific data flows into and out of a symbol, however, you use separate data flow symbols with single arrowheads. For example, in Figure 4-14 the separate data flows (SUBMITTED WORK and GRADED WORK) go into and out of the GRADE STUDENT WORK process. Because diagram 0 is an exploded version of process 0, it shows considerably more detail than the context diagram. You also can refer to diagram 0 as a partitioned or decomposed view of process 0. When you explode a DFD, the higher-level diagram is called the parent diagram, and the lower-level diagram is referred to as the child diagram. The grading system is simple enough that you do not need any additional DFDs to model the system. At that point, the four processes, the one data store, and the 10 data flows can be documented in the data dictionary. When you create a set of DFDs for a system, you break the processing logic down into smaller units, called functional primitives, that programmers will use to develop code. A functional primitive is a process that consists of a single function that is not exploded further. For example, each of the four processes shown in the lower portion of Figure 4-15 is a functional primitive. You document the logic for a functional primitive by writing a process description in the data dictionary. Later, when the logical design is implemented as a physical system, programmers will transform each functional primitive into program code and modules that carry out the required steps. Deciding whether to explode a process further or determine that it is a functional primitive is a matter of experience, judgment, and interaction with programmers who must translate the logical design into code.

EXAMPLE: DIAGRAM 0 DFD FORAN ORDER SYSTEM Figure 4-16 on the next page shows the diagram 0 for an order system. Process 0 on the order system's context diagram is exploded to reveal three processes (FILL ORDER, CREATE INVOICE, and APPLY PAYMENT), one data store (ACCOUNTS RECEIVABLE), two additional data flows (INVOICE DETAIL and PAYMENT DETAIL), and one diverging data flow (INVOICE). The following walkthrough explains the DFD shown in Figure 4-16: 1. A CUSTOMER submits an ORDER. Depending on the processing logic, the FILL ORDER process either sends an ORDER REJECT NOTICE back to the customer or sends a PICKING LIST to the WAREHOUSE.

2. A COMPLETED ORDER from the WAREHOUSE is input to the CREATE INVOICE process, which outputs an INVOICE to both the CUSTOMER process and the ACCOUNTS RECEIVABLE data store.
3. A CUSTOMER makes a PAYMENT that is processed by APPLY PAYMENT. APPLY PAYMENT requires INVOICE DETAIL input from the ACCOUNTS RECEIVABLE data store along with the PAYMENT. APPLY PAYMENT also outputs PAYMENT DETAIL back to the ACCOUNTS RECEIVABLE data store and outputs COMMISSION to the SALES DEPT, BANK DEPOSIT to the BANK, and CASH RECEIPTS ENTRY to ACCOUNTING. The walkthrough of diagram 0 illustrates the basic

requirements of the order system. To learn more, you would examine the detailed description of each separate process.

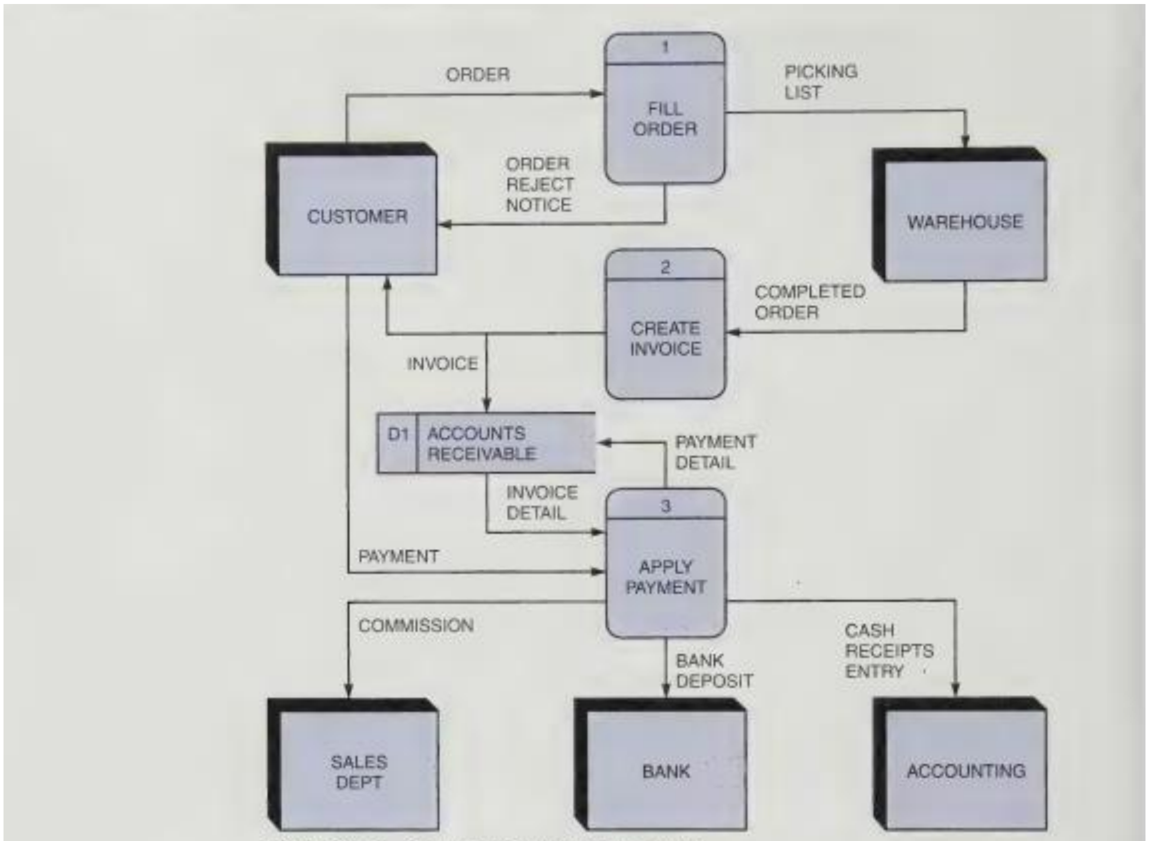


FIGURE 4-16 Diagram 0 DFD for the order system.

Unit-2 Object Oriented Analysis

- THE Constituents of OOAD:-
- Objects and Classes

The concepts of objects and classes are intrinsically linked with each other and form the foundation of object–oriented paradigm.

Object

An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Each object has –

- ➔ Identity that distinguishes it from other objects in the system.
- ➔ State that determines the characteristic properties of an object as well as the values of the properties that the object holds.
- ➔ Behavior that represents externally visible activities performed by an object in terms of changes in its state.

Objects can be modelled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

Class

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.

The constituents of a class are –

- ➔ A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes. Attributes are often referred as class data.
- ➔ A set of operations that portray the behavior of the objects of the class. Operations are also referred as functions or methods.

Example

Let us consider a simple class, Circle, that represents the geometrical figure circle in a two–dimensional space. The attributes of this class can be identified as follows –

- ➔ x–coord, to denote x–coordinate of the center

- y–coord, to denote y–coordinate of the center
- a, to denote the radius of the circle

Some of its operations can be defined as follows –

- findArea(), method to calculate area
- findCircumference(), method to calculate circumference
- scale(), method to increase or decrease the radius

During instantiation, values are assigned for at least some of the attributes. If we create an object my_circle, we can assign values like x-coord : 2, y-coord : 3, and a : 4 to depict its state. Now, if the operation scale() is performed on my_circle with a scaling factor of 2, the value of the variable a will become 8. This operation brings a change in the state of my_circle, i.e., the object has exhibited certain behavior.

- Links and Association:
  A link represents a connection through which an object collaborates with other objects. Rumbaugh has defined it as "a physical or conceptual connection between objects". Through a link, one object may invoke the methods or navigate through another object. A link depicts the relationship between two or more objects.

- Association:
  Association is a group of links having common structure and common behavior. Association depicts the relationship between objects of one or more classes. A link can be defined as an instance of an association.

- Degree of an Association:
  Degree of an association denotes the number of classes involved in a connection. Degree may be unary, binary, or ternary.
- **unary relationship** connects objects of the same class.
- **binary relationship** connects objects of two classes.
- **ternary relationship** connects objects of three or more classes.

- **Cardinality Ratios of Associations**
  Cardinality of a binary association denotes the number of instances participating in an association. There are three types of cardinality ratios, namely –

➔ **One–to–One** − A single object of class A is associated with a single object of class B.
➔ **One–to–Many** − A single object of class A is associated with many objects of class B.
➔ **Many–to–Many** − An object of class A may be associated with many objects of class B and conversely an object of class B may be associated with many objects of class A.

- Generalization and Specialization

Generalization and specialization represent a hierarchy of relationships between classes, where subclasses inherit from super-classes.

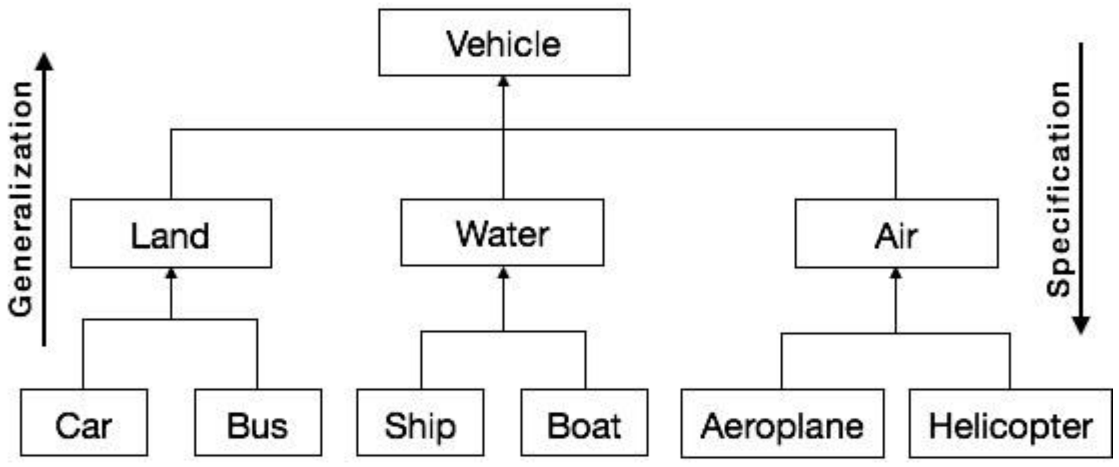
- **Generalization**

In the generalization process, the common characteristics of classes are combined to form a class in a higher level of hierarchy, i.e., subclasses are combined to form a generalized super-class. It represents an "is – a – kind – of" relationship. For example, "car is a kind of land vehicle", or "ship is a kind of water vehicle".

- **Specialization**

Specialization is the reverse process of generalization. Here, the distinguishing features of groups of objects are used to form specialized classes from existing classes. It can be said that the subclasses are the specialized versions of the super-class.

The following figure shows an example of generalization and specialization.



- Aggregation or Composition

Aggregation or composition is a relationship among classes by which a class can be made up of any combination of objects of other classes. It allows objects to be placed directly within the body of other classes. Aggregation is referred as a "part–of" or "has–a" relationship, with the ability to navigate from the whole to its parts. An aggregate object is an object that is composed of one or more other objects.

Example

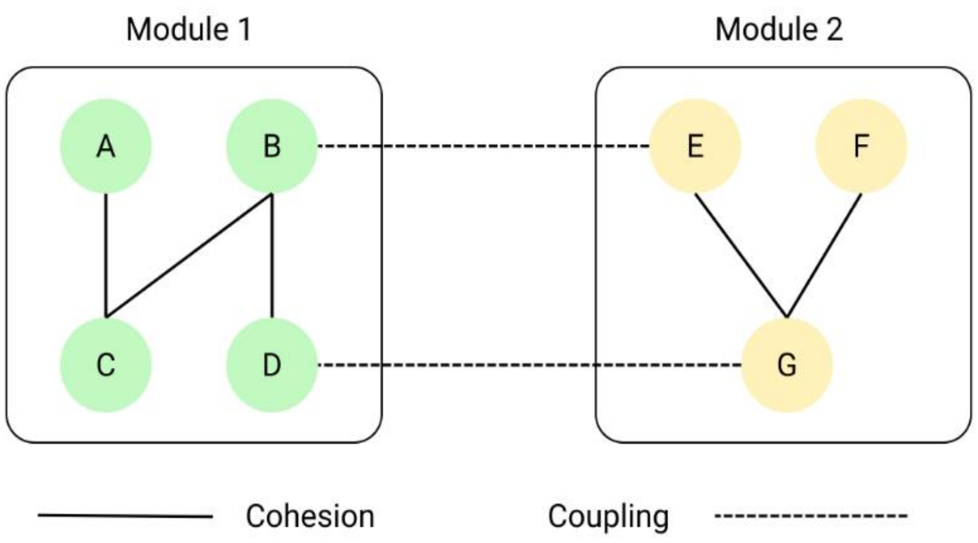
In the relationship, "a car has–a motor", car is the whole object or the aggregate, and the motor is a "part–of" the car. Aggregation may denote –

➔ **Physical containment** – Example, a computer is composed of monitor, CPU, mouse, keyboard, and so on.
➔ **Conceptual containment** – Example, shareholder has–a share.

- Coupling and Cohesion:
  As software developers, we may have heard of the concepts of Cohesion and Coupling in Object-Oriented Programming. These two concepts are used to measure our code quality and ensure it is maintainable and scalable. Let's understand the meaning of both terms.

➔ Cohesion:
  defines how effectively elements within a module or object work together to fulfil a single well-defined purpose. A high level of cohesion means that elements within a module are tightly integrated and working together to achieve the desired functionality.

➔ Coupling:
defines the degree of interdependence between software modules. Tight coupling means that modules are closely connected and changes in one module can affect others. On the other hand, loose coupling means that modules are independent and changes in one module have minimal impact on others.



So, it's best practice in OOPS to achieve loose coupling and high cohesion because it helps us to create classes or modules that are more flexible and less prone to break when changes are made.
Let's move forward to understand the idea of low and high cohesion with an example. After this section, we will also discuss the idea of tight and loose coupling

- **Components**

The main components of an object-oriented model are:

- Objects

Objects represent real-world entities by combining data with related operations. For example, an object could contain a customer’s name, address, and account balance data along with methods to modify or retrieve that data. This allows for an intuitive model that more closely mirrors real objects compared to just tables of data. Objects enable encapsulation of both state through data attributes and behavior through methods. This provides a cohesive and modular approach to structure data.

- Classes

Classes act as templates or blueprints for creating objects with the same kinds of data and behaviors. A class defines the structure for its instances – what attributes or data members the object will contain as well as what methods or functions it will have. Classes allow for organization of objects that share similar properties and behaviors. This promotes reusability and maintainability in database design.

- Inheritance

Inheritance allows classes to inherit attributes and methods from parent classes. This enables code reuse and the creation of specialized child classes. The class hierarchy represents these parent-child relationships. A base or superclass provides common data and behaviors shared by derived subclasses down the hierarchy. Subclasses can override or extend parent functionality.

- Methods

Methods are functions defined within a class that represent the behaviors or actions that objects created from that class can perform. Methods operate on the object's data members and encapsulate the logic for manipulating the object's state. This provides an organized approach to implementing functionality for the class.

- Attribute
  An attribute describes the properties of object. For example: Object is STUDENT and its attribute are Roll no, Branch, Setmarks() in the Student class.

- Interface

An Interface is a collection of operations that are used to specify a service provided by a class or component. It represents a contract with the user. In UML an Interface is represented either by a "lollipop" or by a rectangle with the word "interface" above the name of the Interface.

- Uml Diagrams:

  Software engineers create UML diagrams to understand the designs, code architecture, and proposed implementation of complex software systems. UML diagrams are also used to model workflows and business processes. Coding can be a complicated process with many interrelated elements.

- **Use-case diagrams**

In UML, use-case diagrams model the behavior of a system and help to capture the requirements of the system.

Use-case diagrams describe the high-level functions and scope of a system. These diagrams also identify the interactions between the system and its actors. The use cases and actors in use-case diagrams describe what the system does and how the actors use it, but not how the system operates internally.

Use-case diagrams illustrate and define the context and requirements of either an entire system or the important parts of the system. You can model a complex system with a single use-case diagram, or create many use-case diagrams to model the components of the system. You would typically develop use-case diagrams in the early phases of a project and refer to them throughout the development process.

Use-case diagrams are helpful in the following situations:

- ➔ Before starting a project, you can create use-case diagrams to model a business so that all participants in the project share an understanding of the workers, customers, and activities of the business.
- ➔ While gathering requirements, you can create use-case diagrams to capture the system requirements and to present to others what the system should do.
- ➔ During the analysis and design phases, you can use the use cases and actors from your use-case diagrams to identify the classes that the system requires.
- ➔ During the testing phase, you can use use-case diagrams to identify tests for the system.

The following topics describe model elements in use-case diagrams:

- **Use cases**
  A use case describes a function that a system performs to achieve the user's goal. A use case must yield an observable result that is of value to the user of the system.
- **Actors**
  An actor represents a role of a user that interacts with the system that you are modeling. The user can be a human user, an organization, a machine, or another external system.
- **Subsystems**
  In UML models, subsystems are a type of stereotyped component that represent independent, behavioral units in a system. Subsystems are used in class, component, and use-case diagrams to represent large-scale components in the system that you are modeling.
- **Relationships in use-case diagrams**
  In UML, a relationship is a connection between model elements. A UML relationship is a type of model element that adds semantics to a model by defining the structure and behavior between the model elements.
- Benefits of Use Cases

Use case diagrams help stakeholders to understand the nature and scope of the business area or the system under development, thus, use case modeling is generally regarded as an excellent technique for capturing the functional requirements of a system. They can be served as the basis for the estimating, scheduling, and validating effort.

Use cases and use case diagrams can be fully integrated with other analysis and design using a CASE tool to produce a complete requirements, design and implementation repository. It don't use a special language and thus can be written in a variety of styles to suit the particular needs of the project.

The Benefits of Use Case Approach:

1. use cases are aligned with business requirements – use cases put requirements in context; their relationship to business tasks is clearly described.

2. From scenarios to whole stories – Use cases enable us to tell stories. It is very easy to describe a use case in a concrete way by turning it into a story or scenario.

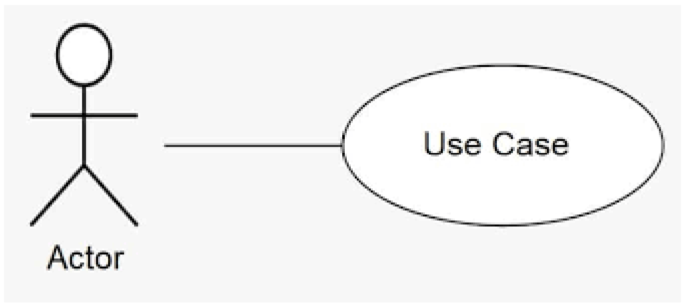
3. Create test cases from use cases – Use cases are reusable in a project. Use cases can be used in a way that captures requirements from each iteration, into a development guide for programmers, into test cases, and finally into user documentation.

4. Derive wireframes and user interfaces from use cases – Use cases focus on the interaction between the user and the system. They make it possible for the user interface designer to participate in the development process before or in parallel with the software developer.

5. Decompose system scope to prioritize functionality – Use cases are useful for scoping. Use cases make it easy for projects to take a phased approach to delivery; they can be added and removed from software projects with relative ease as priorities change.

- Actors

Actors are external entities that interact with the system. These can include users, other systems, or hardware devices. In the context of a Use Case Diagram, actors initiate use cases and receive the outcomes. Proper identification and understanding of actors are crucial for accurately modeling system behavior.



- **Relationships Between Actors and Use Cases**

The relationships between actors and use cases are crucial for illustrating the dynamics of system interactions:

  - **Association:** Represented by a line connecting an actor to a use case, it signifies the communication or interaction between the actor and the use case.
  - **Include Relationship:** Indicated by a dashed arrow, it signifies that a use case includes the functionality of another use case, providing a modular and reusable design.
  - **Extend Relationship:** Denoted by a dashed arrow with an "extend" keyword, it demonstrates that a use case can be extended by another use case under specific conditions.

- **sequence diagram:**
- **sequence diagram elements:**

- **Lifelines** represent an individual participant within an interaction. Lifelines are represented by vertical dashed lines, connecting at the top to an object or actor.
- **Actors** represent the user, external hardware or other subject setting off a particular sequence. They often appear as a stick figure icon and are placed to the left of the first object on the horizonal axis.
- **Activation bars** -- sometimes called *focus of control* -- are thin rectangular boxes that sit vertically on top of an object's lifeline to indicate the duration of a particular operation.
- **combination fragments** Commonly used **combination fragments** · It is possible to combine frames in order to capture, e.g., loops or branches. · Combined fragment keywords: alt, opt, break ...

  Combined **fragments** place multiple interaction **fragments** together and represent specific conditions that affect sequences. They are enclosed in boxes that ..

  Combined **fragments** in a sequence diagram determine the behavior of several interaction **fragments**. However, they only form the framework. They are defined by ...

Sequence **Fragments**. In a UML sequence diagram, combined **fragments** let you show loops, branches, and other alternatives. A combined fragment consists of one

- **Guards**, also known as *interaction constraints*, are square brackets positioned around an interaction as a Boolean expression. They represent a check or pause to make sure certain conditions have been met.
- **Messages** represent communication from one actor or object to another. The first message typically starts at the top left of the sequence diagram and moves horizontally across lifelines from left to right, with each subsequent message one step lower than the one before.
- **Object**

  Each **object** also has its timeline represented by a dashed line below the **object**. Messages between objects are represented by arrows that point from sender ...

  In the UML, an **object** in a sequence diagram is drawn as a rectangle containing the name of the **object**, underlined. An **object** can be named in one of three ways: ..

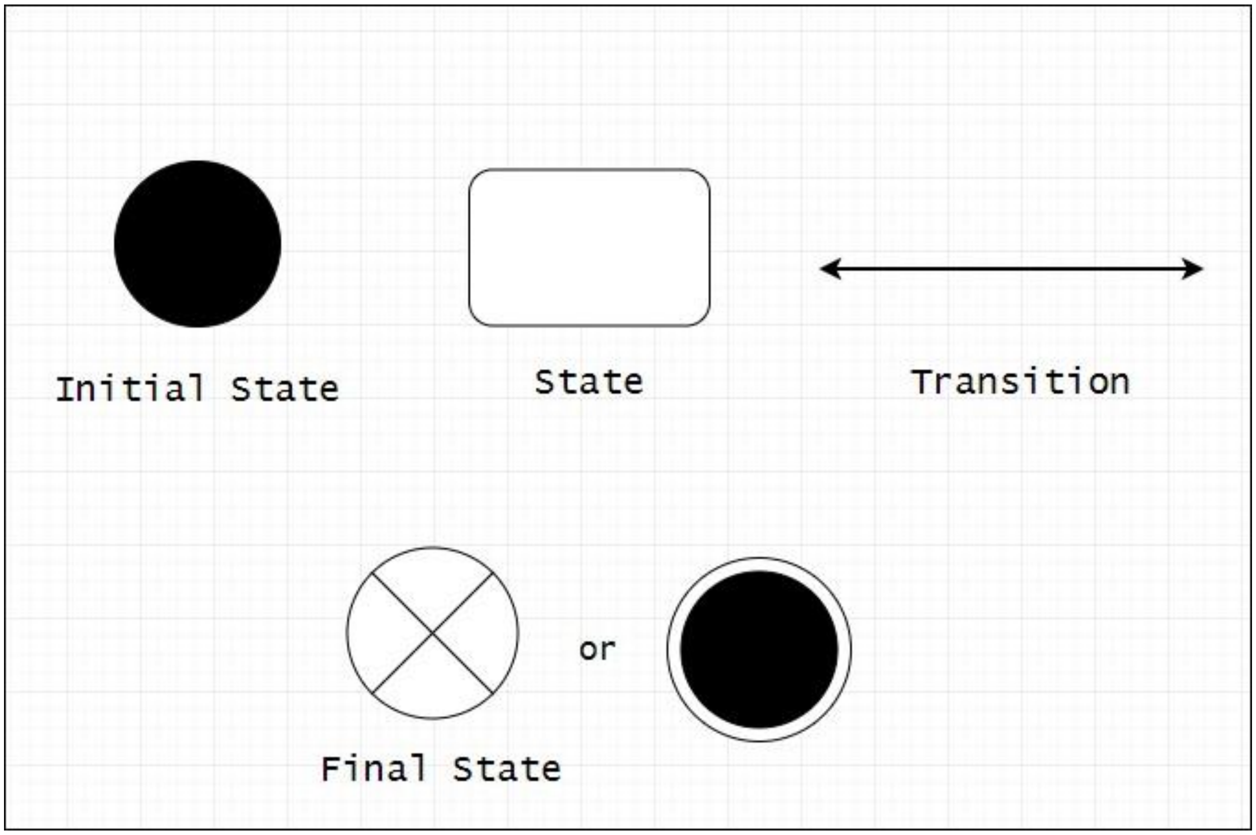
.

Unit-3 UML DIAGRAMS

- Activity Diagram:

- Elements of Activity Diagram:

- **Initial state:** It defines the initial state (beginning) of a system, and it is represented by a black filled circle.

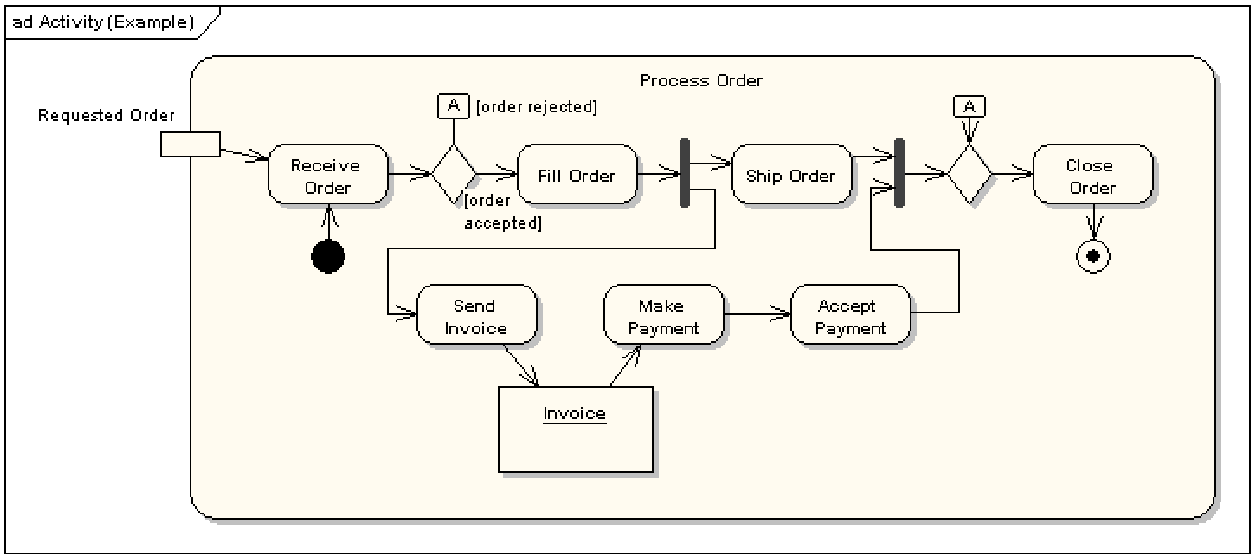


- **Final state:** It represents the final state (end) of a system. It is denoted by a filled circle present within a circle.

- **Activity Diagrams**

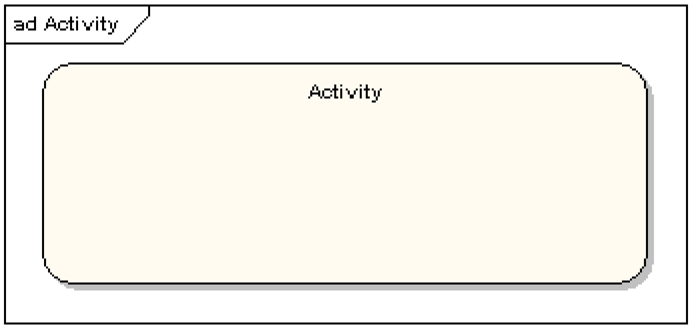
In UML, an activity diagram is used to display the sequence of activities. Activity diagrams show the workflow from a start point to the finish point detailing the many decision paths that exist in the progression of events contained in the activity. They may be used to detail situations where parallel processing may occur in the execution of some activities. Activity diagrams are useful for business modelling where they are used for detailing the processes involved in business activities.

An Example of an activity diagram is shown below.



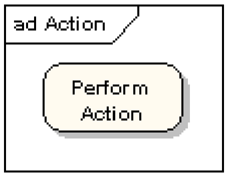
- Activities

An activity is the specification of a parameterized sequence of behaviour. An activity is shown as a round-cornered rectangle enclosing all the actions, control flows and other elements that make up the activity.



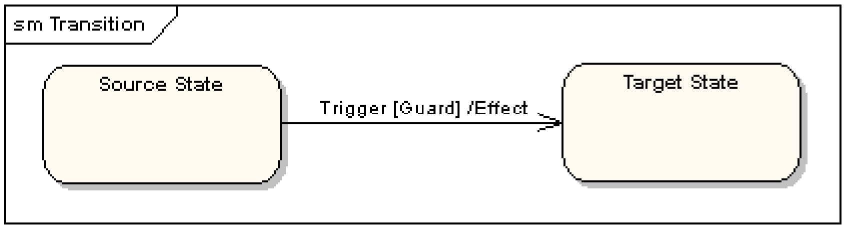
- Actions

An action represents a single step within an activity. Actions are denoted by round-cornered rectangles.



- Transitions

Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below.



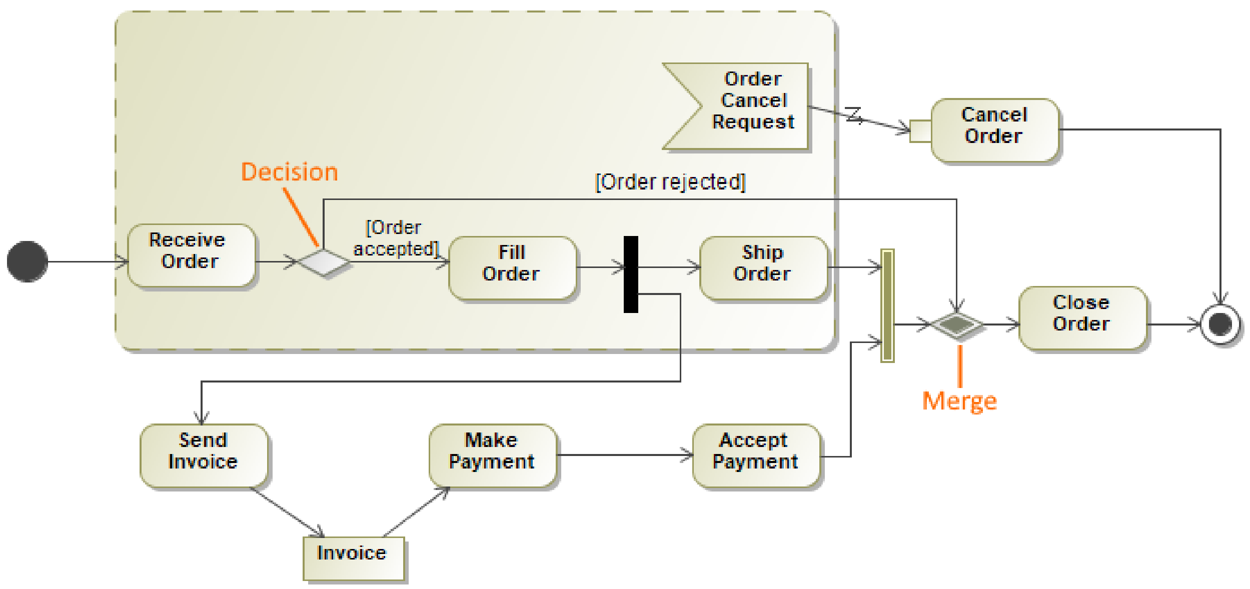
"Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

- Descision

  A Decision Node in an Activity diagram is used much like a choice or junction point in the State diagrams. Decision nodes allow you to separate the Activity Edges. Merge nodes allow you to combine the Activity Edges together.

1. The symbol used for the Decision Node is a large diamond shape. It can have one or two incoming Activity Edges, and at least one outgoing Activity Edge
2. The symbol used for the Merge Node is a large diamond shape. It can have exactly one outgoing Activity Edge but may have multiple incoming Activity Edges.

The Decision and Merge are represented by the same symbol. Although, you can set the **Use different Fork/Join and Decision/Merge notations** project option to draw different notations for the **Decision** and **Merge**.



- Synchronization:

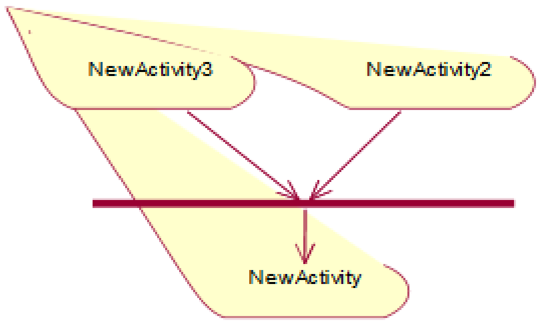
  The contents of the semantic UML model are synchronized with the corresponding diagrams by default. This canonical behavior means that any changes that you make to the model are reflected in the diagrams, and any changes that you make to the diagrams are reflected in the model.
  Canonical behavior occurs in specific elements in a UML model. You can disable the canonical mode for specific diagram element compartments, such as a state region and an activity partition. Canonical mode is always enabled for all list compartments and interaction diagrams.
  When the canonical mode is turned off, changes that you make to the semantic model are not reflected in the diagram. The ability to turn off canonical behavior is useful if you import Rose models, which have no canonical reflection of semantic content.

- Fork and Join

Join in Activity diagram:

1. A join in Activity diagram represents the synchronization of two or more concurrent flows of control.
2. A join may have two or more incoming transitions and one outgoing transition.
3. Above the join, activities associated with each of these paths continue in parallel.
4. At the join, the concurrent flows synchronize, means each waits until all incoming flows have reached at the join, at which point one flow of control continues on below the join.

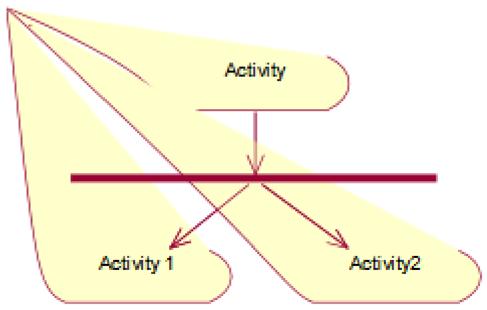
Notation:



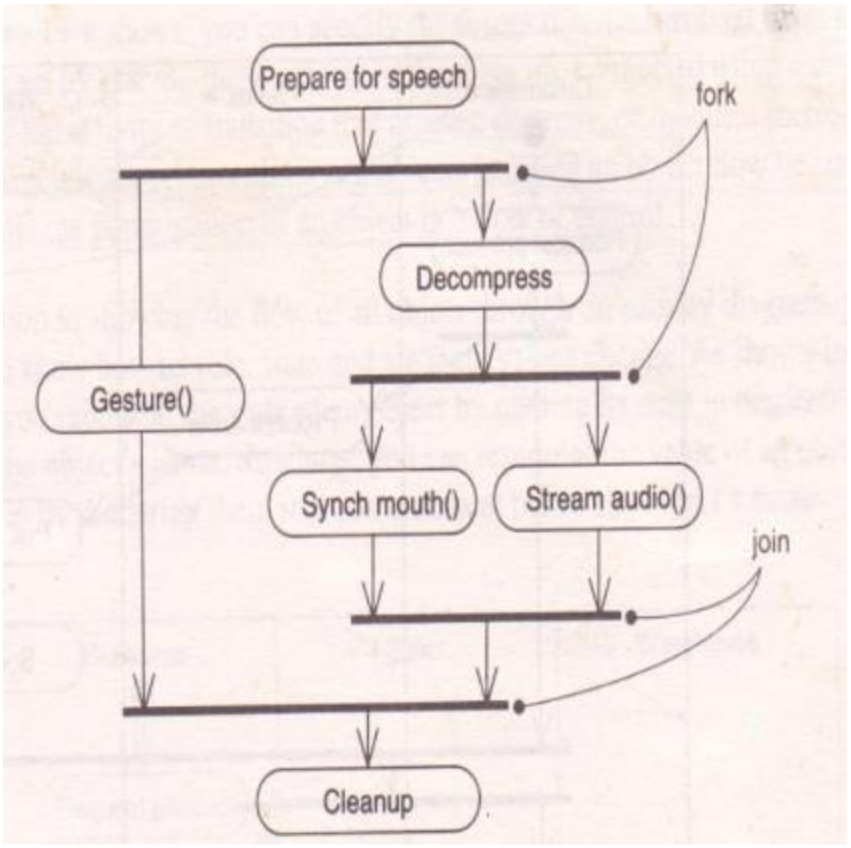
Fork in Activity diagram:

1. A fork in activity diagram represents the splitting of a single flow of control into two or more concurrent flows of control.
2. A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control.
3. Below the fork, the activities associated with each of these paths continue in parallel.

Notation:



Example:

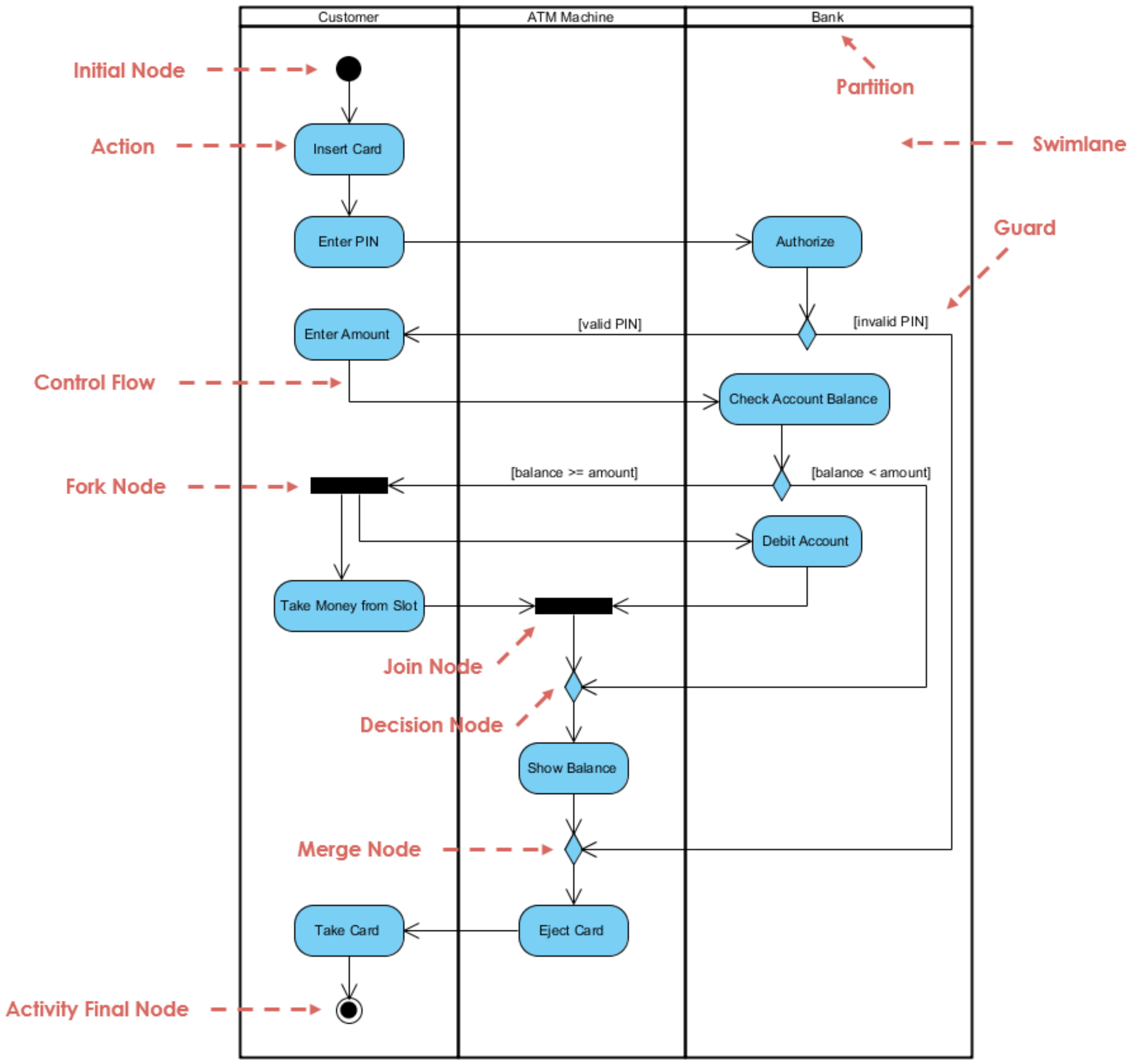


In the above example there are two forks and two joins. Joins and forks should balance, meaning that the number of flows that leave a fork should match the number of flows that enter its corresponding join.The activity Stream audio needed to tell the activity Synch mouth when important pauses and intonations occurred. Similarly, for Synch mouth, we would see transitions triggered by these same signals, to which the Synch mouth state machine would respond.

- Swim lanes

  Withdraw money from an ATM Account - The three involved classes (people, etc.) of the activity are Customer, ATM, and Bank. represented **swimlanes** that determine which object is responsible for which activity. The process begins at the black start circle at the top and ends at the concentric white/black stop circles at the bottom. The activities are rounded rectangles.

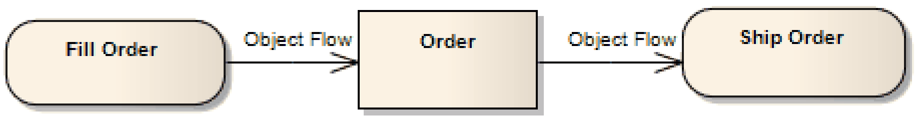
  A single transition comes out of each activity, connecting it to the next activity, which may branch into two or more mutually exclusive transitions. Guard expressions (inside [ ]) label the transitions coming out of a branch. A branch and its subsequent merge marking the end of the branch appear in the diagram as hollow diamonds. A transition may fork into two or more parallel activities. The fork and the subsequent join of the threads coming out of the fork appear in the diagram as solid bars.



- Object and object flow

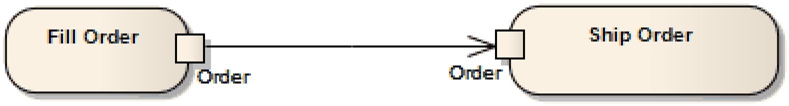
  In Activity diagrams, there are several ways to define the flow of data between objects.

  This diagram depicts a simple Object Flow between two actions, Fill Order and Ship Order, both accessing order information.



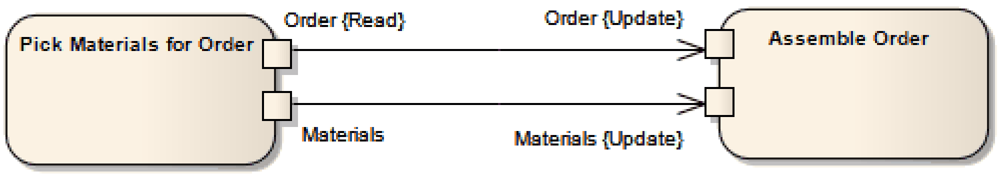
  See UML Superstructure Specification, v2.1.1, figure 12.110, p.391.

This explicit portrayal of the data object Order, connected to the Activities by two Object Flows, can be refined by using this format. Here, Action Pins are used to reflect the order.



See UML Superstructure Specification, v2.1.1, figure 12.110, p.391.

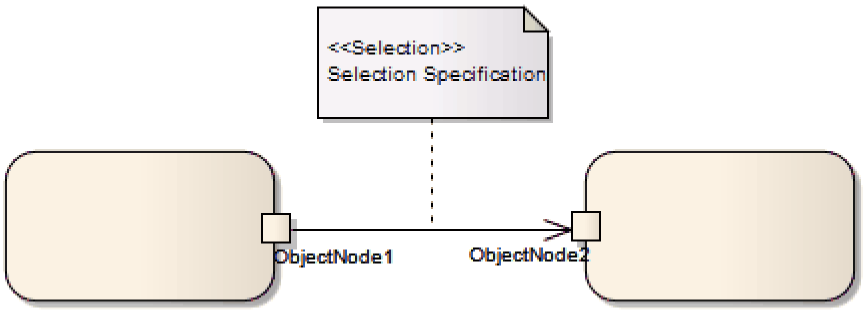
This diagram is an example of multiple Object Flows exchanging data between two actions.



See UML Superstructure Specification, v2.1.1, figure 12.111, p.391.

Selection and transformation behavior, together composing a sort of query, can specify the nature of the Object Flow's data access. Selection behavior determines which objects are affected by the connection. Transformation behavior might then further specify the value of an attribute pertaining to a selected object.

Selection and transformation behaviors can be defined by attaching a note to the Object Flow. To do this, right-click on the Object Flow and select the 'Attach Note or Constraint' option. A dialog lists other flows in the diagram to which you can select to attach the note, if the behavior applies to multiple flows. To comply with UML 2, preface the behavior with the notation «selection» or «transformation».



- Class diagram
- Class diagram Elements:

- Class:
  On a Class diagram you can illustrate relationships between **Classes** and Interfaces using Generalizations, Aggregations and Associations, which are valuable in . Some examples include classifiers such as actors, **classes**, components, information items, and nodes.

Behavioral model elements, These elements model the . UML class diagrams show the **classes** of the system, their inter-relationships, and the operations and attributes of the **classes**.

- **Interface:**
  Interfaces are implemented, “realized” in UML parlance, by classes and components – to realize an interface a class or component must implement the operations . These include **interfaces**, the three remaining types of associations, visibility, and other additions in the UML 2 specification. **Interfaces**. Earlier, I . On a Class diagram you can illustrate relationships between Classes and **Interfaces** using Generalizations, Aggregations and Associations, which are valuable .

- Association:
  An **association** indicates that objects of one class have a relationship with objects of another class, in which this connection has a specifically defined . Aggregation is an **association** with the relation between the whole and its parts, and is represented as empty diamond on the Class Diagram. Composition is a . An **association** is a linkage between two classes. Associations are always assumed to be bi-directional; this means that both classes are aware of each other

- User interfaces and Layouts:

- ViewGroups

A ViewGroup is a special view that can contain other views. The ViewGroup is the base class for Layouts in android, like LinearLayout, RelativeLayout, FrameLayout etc.

In other words, ViewGroup is generally used to define the layout in which views(widgets) will be set/arranged/listed on the android screen.
ViewGroups acts as an invisible container in which other Views and Layouts are placed. Yes, a layout can hold another layout in it, or in other words a **ViewGroup** can have another **ViewGroup** in it.

- built in layout classes:

Frame Layout: FrameLayout is a ViewGroup subclass, used to specify the position of View elements it contains on the top of each other to display only a single View inside the FrameLayout.

Linear Layout: LinearLayout is a ViewGroup subclass, used to provide child View elements one by one either in a particular direction either horizontally or vertically based on the orientation property.

Relative Layout: RelativeLayout is a ViewGroup subclass, used to specify the position of child View elements relative to each other like (A to the right of B) or relative to the parent (fix to the top of the parent).

Table Layout: TableLayout is a ViewGroup subclass, used to display the child View elements in rows and columns.

Grid View: GridView is a ViewGroup that is used to display a scrollable list of items in a grid view of rows and columns.

- **multiple Layouts:**
  As we have studied about different Views, ViewGroups and Layouts in the previous tutorials, now it's time to study how to use all of them together in our android project to design great user interfaces. In this tutorial, we will learn how we can put different layouts, views and viewgroups inside another layout(hierarchical arrangement) to design the perfect GUI for your android application.

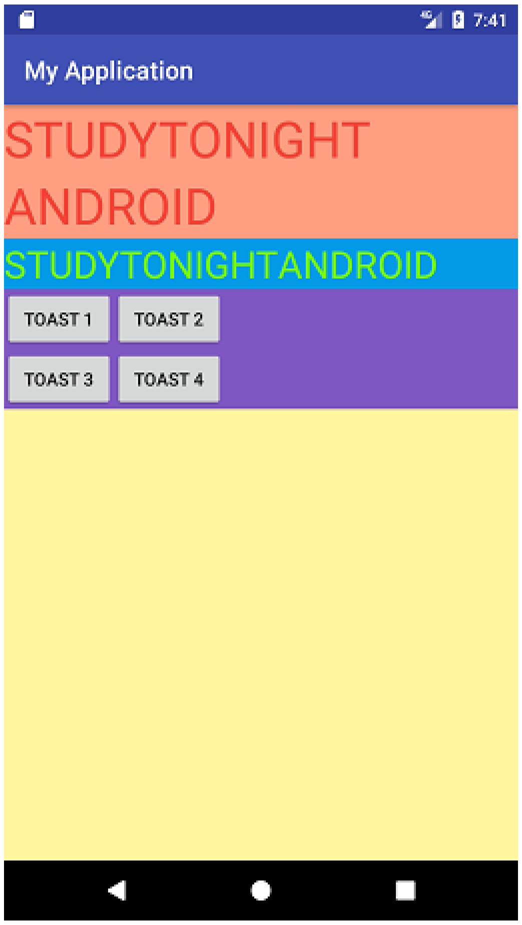
  We have a very basic example below to demonstrate how we can use multiple layouts, views and viewgroups together.

Defining the design in layout XML file

```
<?xml version="1.0" encoding="utf-8"?>

<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/a
ndroid"

    xmlns:app="http://schemas.android.com/apk/res-
auto"

    xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"

    android:layout_height="match_parent"
    android:background="#FFF59D"

tools:context="com.example.android.myapplication.Ma
inActivity">

    <!--Light Yellow Color-->


    <LinearLayout

        android:layout_width="match_parent"

        android:layout_height="wrap_content"

        android:orientation="vertical"

        android:id="@+id/l1"

        android:background="#FF9E80">

        <!--Light Pink Color-->


        <TextView

            android:layout_width="wrap_content"

            android:layout_height="wrap_content"

            android:text="Studytonight"

            android:textAllCaps="true"
            android:textSize="40dp"

            android:textColor="#F44336"/>


        <TextView

            android:layout_width="wrap_content"

            android:layout_height="wrap_content"

            android:text="Android"

            android:textAllCaps="true"

            android:textSize="40dp"

            android:textColor="#F44336"/>


    </LinearLayout>


    <LinearLayout

        android:layout_width="match_parent"

        android:layout_height="wrap_content"

        android:orientation="horizontal"

        android:layout_below="@id/l1"
        android:background="#039BE5"

        android:id="@+id/l2">

        <!--Light Blue Color-->


        <TextView

            android:layout_width="wrap_content"

            android:layout_height="wrap_content"

            android:text="Studytonight"

            android:textAllCaps="true"

            android:textSize="30dp"

            android:textColor="#76FF03"

            />


        <TextView

            android:layout_width="wrap_content"

            android:layout_height="wrap_content"

            android:text="Android"

            android:textAllCaps="true"
```

```xml
        android:textSize="30dp"

        android:textColor="#76FF03"

        />


</LinearLayout>


<RelativeLayout

    android:layout_width="match_parent"

    android:layout_height="wrap_content"

    android:layout_below="@id/l2"

    android:background="#7E57C2"

    >

    <!--Light Purple Color-->


    <Button

        android:layout_width="wrap_content"

        android:layout_height="wrap_content"

        android:text="Toast 1"
        android:id="@+id/b1"

        />

    <Button

        android:layout_width="wrap_content"

        android:layout_height="wrap_content"

        android:text="Toast 2"

        android:layout_toRightOf="@id/b1"

        android:id="@+id/b2"

        />

    <Button

        android:layout_width="wrap_content"

        android:layout_height="wrap_content"

        android:text="Toast 3"

        android:layout_below="@id/b1"

        android:id="@+id/b3"

        />

    <Button

        android:layout_width="wrap_content"
```

```
            android:layout_height="wrap_content"

            android:text="Toast 4"

            android:layout_below="@id/b2"

            android:layout_toRightOf="@id/b3"

            />



    </RelativeLayout>



</RelativeLayout>
```



- In the UI design above, we have a root element which is RelativeLayout. This means that all its children, whether it is layouts or views will be arranged in a relative fashion. Thus, it is very important to choose our root layout carefully.
- Next, we have a Linear Layout, whose orientation is set to **vertical**. This means that all the elements inside this Linear Layout will be arranged in a vertical fashion.
- Next, we have another Linear Layout, whose orientation is set to **horizontal**. Thus, as you can see, we can have different layouts placed inside a root layout.
- Next, we have another Relative Layout placed inside the root Relative Layout. This Relative Layout has 4 buttons inside it, arranged in a relative fashion to each other.
- Remember, all these layouts are placed inside a Relative Layout. Thus, all these layouts are relative to each other.

Android provides a very structured and extensible way of designing the user interface. Now that you have understood how to use various layouts, views and viewgroups, go ahead design great some great UI screens for your app.

Unit-4 QA and Testing

- Quality, Quality Assurance, and Quality Control:

  A software product is designed to satisfy certain requirements of a given customer (or set of customers). How can we characterize this phrase—"satisfying requirements"? Requirements get translated into software features, each feature being designed to meet one or more of the requirements. For each such feature, the expected behavior is characterized by a set of fest cases. Each test case is further characterized by

  1. The environment under which the test case is to be executed;

  2. Inputs that should be provided for that test case;

  3. How these inputs should get processed;

  4. What changes should be produced in the internal state or environment; and

  5. What outputs should be produced.

  The actual behavior of a given software for a given test case, under a given set of inputs, in a given environment, and in a given internal state is characterized by

  1. How these inputs actually get processed;

  2. What changes are actually produced in the internal state or environment; and

  3. What outputs are actually produced.

  If the actual behavior and the expected behavior are identical in all their characteristics, then that test case is said to be passed. If not, the given software is said to have a defect on that test case. How do we increase the chances of a product meeting the requirements expected of it, consistently and predictably? There are two types of methods— quality control and quality assurance. Quality control attempts to build a product, test it for expected behavior after it is built, and if the expected behavior is not the same as the actual behavior of the product, fixes the product as is necessary and rebuilds the product. This iteration is repeated till the expected behavior of the product matches the actual behavior for the scenarios tested. Thus quality control is defect-detection and defect-correction oriented, and works on the product rather than on the process. Quality assurance, on the other hand, attempts defect prevention by concentrating on the process of producing the product rather than working on defect detection/correction after the product is built. For example, instead of producing and then testing a program code for proper behavior by exercising the built product, a quality assurance approach would be to first review the design before the product is built and correct the design errors in the first place. Similarly, to ensure the production of a better code, a quality assurance process may mandate coding standards to be followed by all programmers. As can be seen from the above examples, quality assurance normally tends to apply to all the products that use a process. Also, since quality assurance continues throughout the life of the product it is everybody's responsibility; hence it is a staff function. In contrast, the responsibility for quality control is usually localized to a quality control team. Table 2.1 summarizes the key distinctions between quality control and quality assurance.

Difference between quality assurance and quality control.

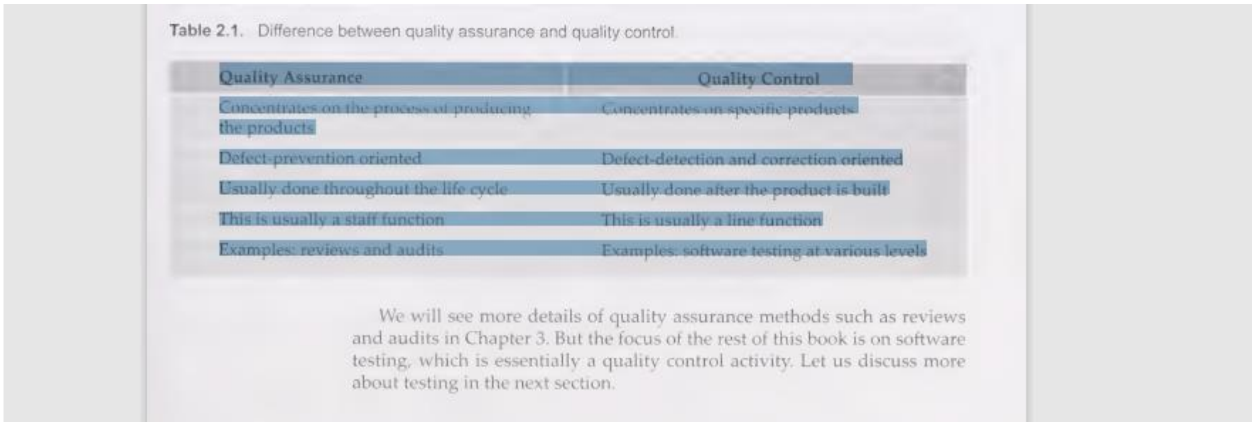


Table 2.1. Difference between quality assurance and quality control

Quality Assurance	Quality Control
Concentrates on the process of producing the products	Concentrates on specific products
Defect-prevention oriented	Defect-detection and correction oriented
Usually done throughout the life cycle	Usually done after the product is built
This is usually a staff function	This is usually a line function
Examples: reviews and audits	Examples: software testing at various levels

We will see more details of quality assurance methods such as reviews and audits in Chapter 3. But the focus of the rest of this book is on software testing, which is essentially a quality control activity. Let us discuss more about testing in the next section.

- White Box Testing:

Every software product is realized by means of a program code. White box testing is a way of testing the external functionality of the code by examining and testing the program code that realizes the external functionality. This is also known as clear box, or glass box or open box testing. White box testing takes into account the program code, code structure, and internal design flow. In contrast, black box testing, to be discussed in Chapter 4, does not look at the program code but looks at the product from an external perspective. A number of defects come about because of incorrect translation of requirements and design into program code. Some other defects are created by programming errors and programming language idiosyncrasies. The different methods of white box testing discussed in this chapter can help reduce the delay between the injection of a defect in the program code and its detection. Furthermore, since the program code represents what the product actually does (rather than what the product is intended to do), testing by looking at the program code makes us get closer to what the product is actually doing. As shown in Figure 3.1, white box testing is classified into "static" and "structural" testing. The corresponding coloured version of Figure 3.1 is available on page 458. We will look into the details of static testing in Section 3.2 and take up structural testing in Section 3.3.

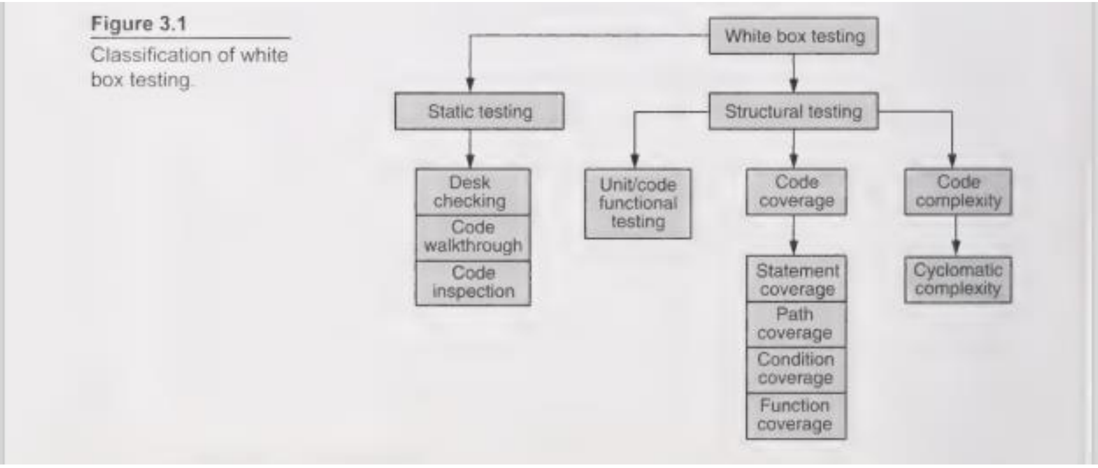
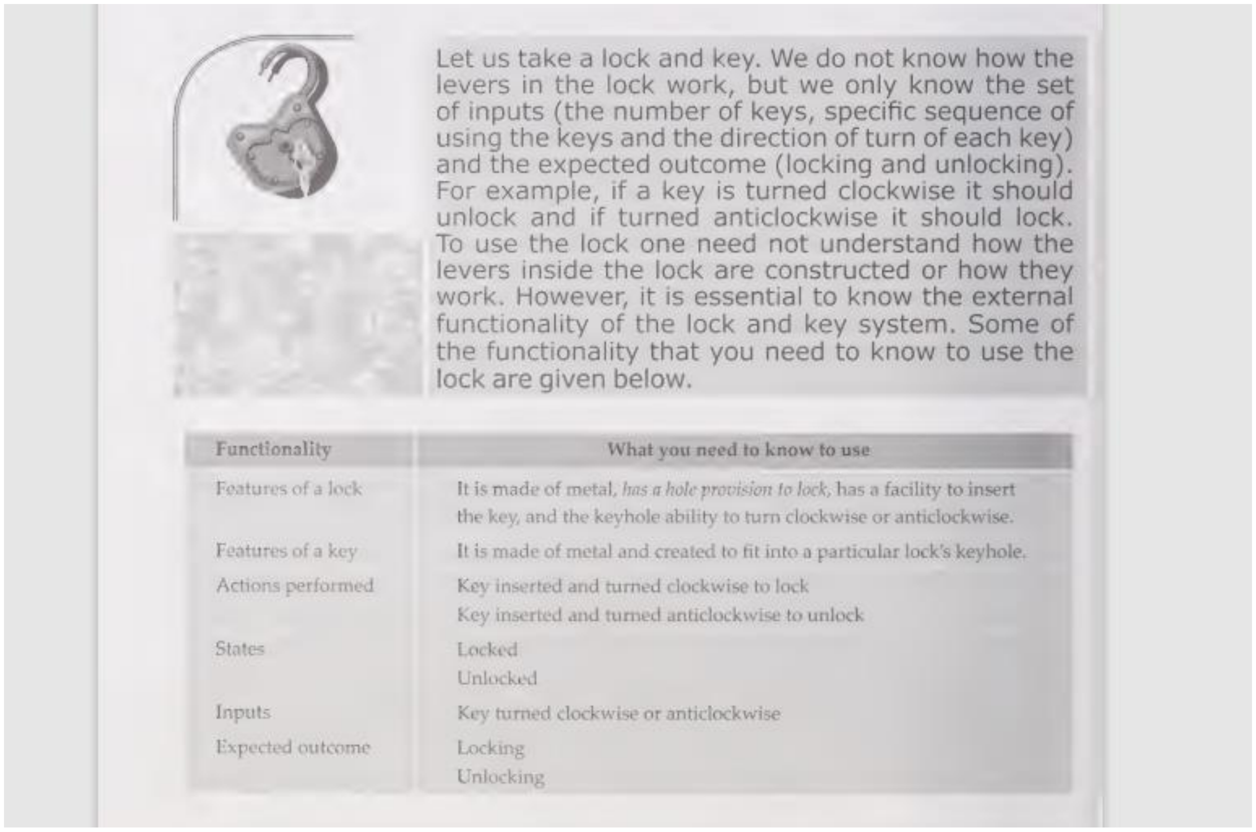


Figure 3.1
Classification of white box testing.

- Black Box Testing:

Black box testing involves looking at the specifications and does not require examining the code of a program. Black box testing is done from the customer's viewpoint. The test engineer engaged in black box testing only knows the set of inputs and expected outputs and is unaware of how those inputs are transformed into outputs by the software. Black box tests are convenient to administer because they use the complete finished product and do not require any knowledge of its construction. Independent test laboratories can administer black box tests to ensure functionality and compatibility. Let us take a lock and key. We do not know how the levers in the lock work, but we only know the set of inputs (the number of keys, specific sequence of using the keys and the direction of turn of each key) and the expected outcome (locking and unlocking). For example, if a key is turned clockwise it should unlock and if turned anticlockwise it should lock. To use the lock one need not understand how the levers inside the lock are constructed or how they work. However, it is essential to know the external functionality of the lock and key system. Some of the functionality that you need to know to use the lock are given below. Black box testing thus requires a functional knowledge of the product to be tested. It does not mandate the knowledge of the internal logic of the system nor does it mandate the knowledge of the programming language used to build the product. Our tests in the above example were focused towards testing the features of the product (lock and key), the different states, we already knew the expected outcome. You may check if the lock works with some other key (other than its own). You may also want to check with a hairpin or any thin piece of wire if the lock works. We shall see in further sections, in detail, about the different kinds of tests that can be performed in a given product.


Let us take a lock and key. We do not know how the levers in the lock work, but we only know the set of inputs (the number of keys, specific sequence of using the keys and the direction of turn of each key) and the expected outcome (locking and unlocking). For example, if a key is turned clockwise it should unlock and if turned anticlockwise it should lock. To use the lock one need not understand how the levers inside the lock are constructed or how they work. However, it is essential to know the external functionality of the lock and key system. Some of the functionality that you need to know to use the lock are given below.

Functionality	What you need to know to use
Features of a lock	It is made of metal, *has a hole provision to lock*, has a facility to insert the key, and the keyhole ability to turn clockwise or anticlockwise.
Features of a key	It is made of metal and created to fit into a particular lock's keyhole.
Actions performed	Key inserted and turned clockwise to lock Key inserted and turned anticlockwise to unlock
States	Locked Unlocked
Inputs	Key turned clockwise or anticlockwise
Expected outcome	Locking Unlocking

- INTEGRATION TESTING:
  A system is made up of multiple components or modules that can comprise hardware and software. Integration is defined as the set of interactions among components. Testing the interaction between the modules and interaction with other systems externally is called integration testing. Integration testing starts when two of the product components are available and ends when all component interfaces have been tested. The final round of integration involving all components is called Final Integration Testing (FIT), or system integration. Integration testing is both a type of testing and a phase of testing. As integration is defined to be a set of interactions, all defined interactions among the components need to be tested. The architecture and design can give the details of interactions within systems; however, testing the interactions between one system and another system requires a detailed understanding of how they work together. This knowledge of integration (that is, how the system or modules work together) depends on many modules and systems. These diverse modules could have different ways of working when integrated with other systems. This introduces complexity in procedures and in what needs to be done. Recognizing this complexity, a phase in testing is dedicated to test these interactions, resulting in the evolution of a process. This ensuing phase is called the integration testing phase. Since integration testing is aimed at testing the interactions among the modules, this testing—just like white box, black box, and other types of testing —comes with a set of techniques and methods, which we will see in the following sections. Hence

integration testing is also viewed as a type of testing (and thus fits into the canvas of this part of the book). In the next section, we will look at integration as a type of testing and in the section thereafter, we will view integration as a phase of testing.

- QA and TESTING:
  Quality Assurance in Software Testing is defined as a procedure to ensure the quality of software products or services provided to the customers by an organization. Quality assurance focuses on improving the software development process and making it efficient and effective as per the quality standards defined for software products. Quality Assurance is popularly known as QA Testing.
  As the programs are coded (in the chosen programming language), they are also tested. In addition, after the coding is (deemed) complete, the product is subjected to testing. Testing is the process of exercising the software product in pre-defined ways to check if the behavior is the same as expected behavior. By testing the product, an organization identifies and removes as many defects as possible before shipping it out.

- System and Acceptance Testing:
  System testing is defined as a testing phase conducted on the complete integrated system, to evaluate the system compliance with its specified requirements. It is done after unit, component, and integration testing phases. A system is a complete set of integrated components that together deliver product functionality and features. A system can also be defined as a set of hardware, software, and other parts that together provide product features and solutions. In order to test the entire system, it is necessary to understand the product's behavior as a whole. System testing helps in uncovering the defects that may not be directly attributable to a module or an interface. System testing brings out issues that are fundamental to design, architecture, and code of the whole product. System testing is the only phase of testing which tests the both functional and non-functional aspects of the product. On the functional side, system testing focuses on real-life customer usage of the product and solutions. System testing simulates customer deployments. For a general-purpose product, system testing also means testing it for different business verticals and applicable domains such as insurance, banking, asset management, and so on. On the non-functional side, system brings in different testing types (also called quality factors), some of which are as follows.
  1. Performance/Load testing To evaluate the time taken or response time of the system to perform its required functions in comparison with different versions of same product(s) or a different competitive product(s) is called performance testing. This type of testing is explained in the next chapter.
  2. Scalability testing A testing that requires enormous amount of resource to find out the maximum capability of the system parameters is called scalability testing. This type of testing is explained in subsequent sections of this chapter.
  3. Reliability testing To evaluate the ability of the system or an independent component of the system to perform its required functions repeatedly for a specified period of time is called reliability testing. This is explained in subsequent sections of this chapter.

4. . Stress testing Evaluating a system beyond the limits of the specified requirements or system resources (such as disk space, memory, processor utilization) to ensure the system does not break down unexpectedly is called stress testing. This is explained in subsequent sections of this chapter.
5. . Interoperability testing This testing is done to ensure that two or more products can exchange information, use the information, and work closely. This is explained in subsequent sections of this chapter.
6. Figure 6.1 Different perspectives of system testing. System and Acceptance Testing 129 6. Localizationtesting Testing conducted to verify that the localized product works in different languages is called localization testing. This is explained in Chapter 9, Internationalization Testing. The definition of system testing can keep changing, covering wider and more high-level aspects, depending on the context. A solution provided to a customer may be an integration of multiple products. Each product may bea combination of several components. A supplier of a component of a product can assume the independent component as a system in its own right and do system testing of the component. From the perspective of the product organization, integrating those components is referred to as sub-system testing. When all components, delivered by different component developers, are assembled by a product organization, they are tested together as asystem. At the next level, there are solution integrators who combine products from multiple sources to provide a complete integrated solution for a client. They put together many products as a system and perform system testing of this integrated solution. Figure 6.1 illustrates the system testing performed by various organizations from their own as well as from a global perspective. The coloured figure is available on page 463.

System testing is performed on the basis of written test cases according to information collected from detailed architecture/design documents, module specifications, and system requirements specifications. System test cases are created after looking at component and integration test cases, and are also at the same time designed to include the functionality that tests the system together. System test cases can also be developed based on user stories, customer discussions, and points made by observing typical customer usage. System testing System testing may not include many negative scenario verification, such as testing for incorrect and negative values. This is because such negative testing would have been already performed by component and integration testing and may not reflect real-life customer usage. System testing may be started once unit, component, and integration testing are completed. This would ensure that the more basic program logic errors and defects have been corrected. Apart from verifying the business requirements of the product, system testing is done to ensure that the product is ready for moving to the user acceptance test level.

- Performance Testing:
  In this Internet era, when more and more of business is transacted online, there is a big and understandable expectation that all applications run as fast as possible. When applications run

fast, a system can fulfill the business requirements quickly and put it in a position to expand its business and handle future needs as well. A system or a product that is not able to service business transactions due to its slow performance is a big loss for the product organization, its customers, and its customers' customers. For example, it is estimated that 40% of online marketing/ shopping for consumer goods in the USA happens over a period of November—December. Slowness or lack of response during this period may result in losses to the tune of several million dollars to organizations. In yet another example, when examination results are published on the Internet, several hundreds of thousands of people access the educational websites within a very short period. If a given website takes a long time to complete the request or takes more time to display the pages, it may mean a lost business opportunity, as the people may go to other websites to find the results. Hence, performance is a basic requirement for any product and is fast becoming a subject of great interest in the testing community.

- regression ion testing:
  Software undergoes constant changes. Such changes are necessitated because of defects to be fixed, enhancements to be made to existing functionality, or new functionality to be added. Anytime such changes are made, it is important to ensure that
  1. The changes or additions work as designed; and
  2. The changes or additions do not break something that is already working and should continue to work.

  Regression testing is designed to address the above two purposes. Let us illustrate this with a simple example. Assume that ina given release of a product, there were three defects—D1, D2, and D3. When these defects are reported, presumably the development team will fix these defects and the testing team will perform tests to ensure that these defects are indeed fixed. When the customers start using the product (modified to fix defects D1, D2, and D3), they may encounter new defects —D4 and D5. Again, the development and testing teams will fix and test these new defect fixes. But, in the process of fixing D4 and D5, as an unintended side-effect, D1 may resurface. Thus, the testing team should not only ensure that the fixes take care of the defects they are supposed to fix, but also that they do not break anything else that was already working. Regression testing enables the test team to meet this objective. Regression testing is important in today's context since software is being released very often to keep up with the competition and increasing customer awareness. It is essential to make quick and frequent releases and also deliver stable software. Regression testing enables that any new feature introduced to the existing product does not adversely affect the current functionality. Regression testing follows selective re-testing technique. Whenever the defect fixes are done, a set of test cases that need to be run to verify the defect fixes are selected by the test team. An impact analysis is done to find out Regression Testing 195 what areas may get impacted due to those defect fixes. Based on the impact analysis, some more test cases are selected to take care of the impacted areas. Since this testing technique focuses on reuse of existing test cases that have already been executed, the technique is called selective re-testing. There may be situations where new test cases need to be developed to take care of some impacted areas. However, by and large, regression testing reuses the test cases that are available, as it focuses on testing the features that are already available and tested at least once already.

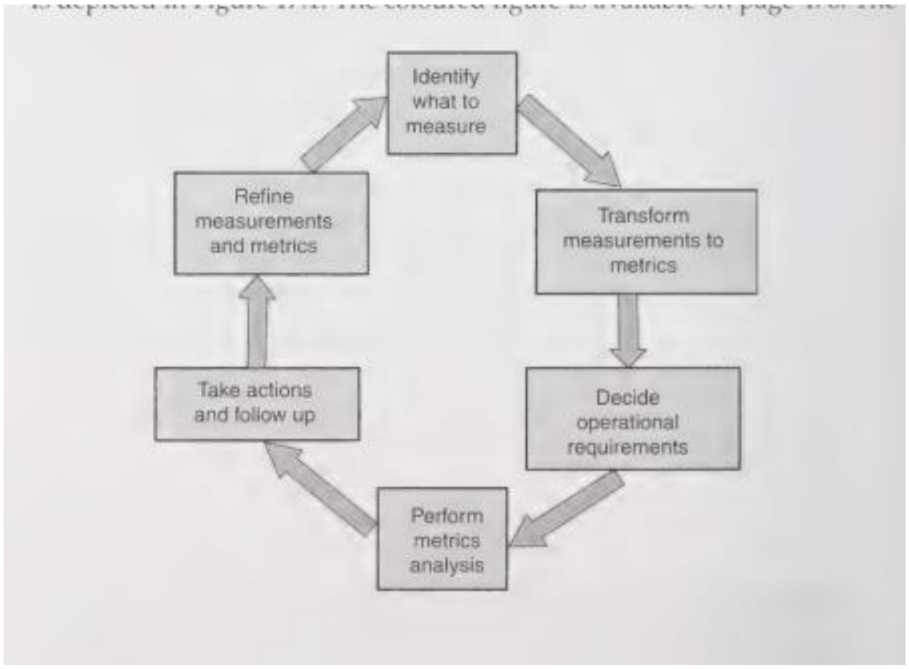
- Test metrics and measurements
  All significant activities in a project need to be tracked to ensure that measurement of key parameters is an integral part of tracking. Measurements first entail collecting a set of data. But, raw data by itself may not throw light on why a particular event has happened. The collected data have to be analyzed in totality to draw the appropriate conclusions. In the above cartoon, the two data points were that the boss had gone on vacation and the profits zoomed in the previous quarter. However (hopefully!) the two events are not directly linked to each other! So the conclusion from the raw data was not useful for decision making. Metrics derive information from raw data with a view to help in decision making. Some of the areas that such information would shed light on are

  1 Relationship between the data points;
  2. Any cause and effect correlation between the observed data points; and
  3. Any pointers to how the data can be used for future planning and continuous improvements.

  Metrics are thus derived from measurements using appropriate formulae or calculations. Obviously, the same set of measurements can help product different set of metrics, of interest to different people. From the above discussion, it is obvious that in order that a project performance be tracked and its progress monitored effectively,

  1. The right parameters must be measured; the parameters may pertain to product or to process.
  2. The right analysis must be done on the data measured, to draw correct conclusions about the health of the product or process within a project or organization.
  3. The results of the analysis must be presented in an appropriate form to the stakeholders to enable them to make the right decisions on improving product or process quality (or any other relevant business drivers).

  Since the focus of this book is on testing and products under test, only metrics related to testing and product are discussed in this chapter and not those meant for process improvements. The metrics and analysis of metrics may convey the reason when data points are combined. Relating several data points and consolidating the result in terms of charts and pictures simplifies the analysis and facilitates the use of metrics for decision making. A metrics program is intended to achieve precisely the above objectives and is the intended focus of this chapter. We will first baseline certain terminology we will be using in this chapter. Effort is the actual time that is spent on a particular activity or a phase. Elapsed days is the difference between the start of an activity and the completion of the activity. For example, ordering a product through the web may involve five minutes of effort and three elapsed days. It is the packaging and shipping that takes that much duration, not the time spent by the person in ordering. However, in the schedule, this latency or delay needs to be entered as three days. Of course, during these three days, the person who ordered the product can get on to some other activity and do it in simultaneously. In general, effort is derived from productivity numbers, and elapsed days is the number of days required to complete the set of activities. Elapsed days for a complete set of activities becomes the schedule for the project. Collecting and analyzing metrics involves effort and several steps. This is depicted in Figure 17.1. The coloured figure is available on page 476. The first step involved in a metrics program is to decide what measurements are important and collect data accordingly. The effort spent on testing, number of defects, and number of test cases, are some examples of measurements. Depending on what the data is used for, the granularity of measurement will vary.


Identify what to measure
Transform measurements to metrics
Decide operational requirements
Perform metrics analysis
Take actions and follow up
Refine measurements and metrics

While deciding what to measure, the following aspects need to be kept in mind.

1. What is measured should be of relevance to what we are trying to achieve. For testing functions, we would obviously be interested in the effort spent on testing, number of test cases, number of defects reported from test cases, and so on
2. . The entities measured should be natural and should not involve too many overheads for measurements. If there are too many overheads in making the measurements or if the measurements do not follow naturally from the actual work being done, then the people who supply the data may resist giving the measurement data (or even give wrong data).
3. . What is measured should be at the right level of granularity to satisfy the objective for which the measurement is being made.

   Let us look at the last point on granularity of data in more detail. The different people who use the measurements may want to make inferences on different dimensions. The level of granularity of data obtained depends on the level of detail required by a specific audience. Hence the measurements — and the metrics derived from them—will have to be at different levels for different people. An approach involved in getting the granular detail is called data drilling. Given in the next page is an example of a data drilling exercise. This is what typically happens in many organizations when metrics/test reports are presented and shows how different granularity of data is relevant for decision making at different levels. The conversation in the example continues till all questions are answered or till the defects in focus becomes small in number and can be traced to certain root causes. The depth to which data drilling happens depends on the focus area of the discussion or need. Hence, it is important to provide as much granularity in measurements as possible. In the above example, the measurement was “number of defects.” Not all conversations involve just one measurement as in the example. A set of measurements can be combined to generate metrics that will be explained in further sections of this chapter. An example question involving multiple measurements is “How many test cases produced the 40 defects in data migration involving different schema?” There are two measurements involved in this question: the number of test cases and the number of defects. Hence, the second step involved in metrics collection is defining how to combine data points or measurements to provide meaningful metrics. A particular metric can use one or more measurements.

Knowing the ways in, which a measurement is going to be used and knowing the granularity of measurements leads us to the third step in the metrics program—deciding the operational requirement for measurements. The operational requirement for a metrics plan’should lay down not only the periodicity but also other operational issues such as who should collect measurements, who shouldi receive the analysis, and so on. This step helps to decide on the appropriate periodicity for the measurements as well as assign operational responsibility for collecting, recording, and reporting the measurements and dissemination of the metrics information. Some measurements need to be made on a daily basis (for example, how many test cases. were executed, how many defects found, defects fixed, and so on). But the metrics. involving a question like the one above (“how many test cases produced 40 defects”) isa type of metric that needs to be monitored at extended periods of time, say, once ina: week or at the end of a test cycle. Hence, planning metrics generation also needs to’ consider the periodicity of the metrics. . sis The fourth step involved ina metrics program is to analyze the metrics to identify both positive areas and improvement areas on product quality. Often, only the improvement aspects pointed to by the metrics are analyzed and focused; it is important to also highlight and sustain the positive areas of the product. This will ensure that the best practices get institutionalized and also motivate the team better. The final step involved in a metrics plan is to take necessary action and follow up on the action. The purpose of a,metrics program will be defeated Test Metrics and Measurements A225 ‘) af the actionitems are not followed through to. completion, This is especially ‘itrueof testing, which is the penultimate phase before release; Any, delay in analysis'and following through'with action ‘items to completion ’can result in undue delays in product release. Any metrics program, as described above, is a continuous and ongoing process. As we make measurements, transform the measurements into metrics, analyze the metrics, and take corrective action, the issues for which the measurements were made in the first place will become resolved. Then, we would have to continue the next iteration of metrics programs, measuring (possibly) a different set of measurements, leading to more refined metrics addressing (possibly) different issues. As shown in Figure 17.1, metrics programs continually go through the steps described above with different measurements or metrics.