Chapter 4.

THE DEFINITION PHASE

Part 1

Summary

The Definition Phase

1. Definition Phase Objectives

- (1) The *first objective* of the **Definition Phase** is ***problem analysis.***
- (2) A *second objective* is ***project planning*** *that means devising a scheme that will produce an acceptable solution to the problem.*
- (3) A *third objective* is writing and getting ***customer approval*** of the set of ***acceptance criteria*** which will help to determine whether your product fulfills the requirements of the contract.

2. Problem Analysis

- The *target* of the **problem analysis**:
  - Writing a **requirements document** which accurately and in detail describes the customer's problem.
- This document is known under different names:
  - Problem Specification
  - Requirements Specification
  - Specification (Spec)
  - Software Requirements Specification (SRS)
- In some cases it will already have been done, at least to a moderate level of detail, during proposal efforts preceding contract award.
  - In that case, the job during the *Definition Phase* may be simply to:
    - (1) *Fine-tune the existing documents*.
    - (2) *Fill* in the details.
    - (3) *Formalize* them.
- Usually, the **requirements** have only been *sketched* during the *proposal phase*.
  - In this case, this first phase of the contract will involve plenty of *analysis work*.
- It often makes sense to contract for *two separate jobs*; one to **define** *the problem* and one to **solve** *that problem*.
  - Federal government of USA often makes use of this principle, especially on very *large contracts* or when the *technical problems are formidable*.
  - The government enters into an agreement with *two or more contractors* simultaneously and funds each to work out a **problem definition** and **design concept**.

    - The government is then free to choose what it likes from among the various concepts submitted.
    - A *single* contractor may then be chosen to go ahead with full development according to the approach chosen.
    - The *period* during which the separate approaches are being worked out is usually called a ***Contract Definition Phase***.
- Some *software vendors* regularly do business this way.
    - The two-stage approach limits their liability and ensures that the customer understands what he is buying.
- Don't assume that the *problem* is obvious and that everyone understands it.
    - Write the **requirement document** even in cases of in-house jobs or internal projects where the work was being done under "shop order" (a sort of within-the-company contract) to another department.
- A serious *analysis of the problem*, avoids a great deal of aggravation and waste of time and money (contract default, loss of profit, and even court action).

2.1 Recommendations for Analyze Activity

- (1) Resist the temptation to *begin* designing programs *immediately*.
- (2) Concentrate first on *what the problem is,* not on *how you are going to solve it.*
- (3) *While the analysts* are describing the *what,* they will be thinking about and discussing *design concepts*.
    - But be sure they know that their first objective is to write a document describing the *problem*, not the *solution.*
- (4) Begin the document by *describing the customer's problem* from scratch, in non-technical language.
- (5) Identify
    - Who the *customer* is.
    - What the *problem environment* is.
    - Why a *computer solution* is sought.
- (6) Then describe the *technical problem* in increasing levels of detail.
- (7) Be very specific about what *capabilities* are to be *included* in the system;
    - Sometimes it's helpful to point out what's *not included.*
- (8) Be precise.
    - Don't leave it to the reader to infer what's *included* and what is not.

2.2 The Problem Specification Document

- The document in focus during this phase, the **Problem Specification**, *defines* in *quantitative* terms the *customer's requirements*.
- This specification states the requirements in some major *categories* (fig. 2.2.a.)
    - *(1) Performance, including file capacities, timing constraints, input rates, and system loads.*
    - *(2) Operational requirements: functions, or operations, or features to be provided by the system.*
    - *(3) Data requirements.*
    - *(4) Human interfaces: interactions between user and software product.*
    - *(5) Human performances: considerations such as minimum times for making decisions, maximum times allowable for system responses, and restrictions on program-generated displays.*

PROBLEM SPECIFICATION (TEMPLATE)

DEPARTMENT:

PROJECT:

DOCUMENT NUMBER:

APROVALS:

DATE OF ISSUE:

REALISED:

SECTION 1**: SCOPE**

The Problem Specification describes the "why" of the project and the requirements of the program system, that is, the job to be done by the programs. It is a baseline document and its most recent edition is to be adhered to strictly by all project personnel.

SECTION 2: **APPLICABLE DOCUMENTS**

SECTION 3: **REQUIREMENTS**

This section, the heart of the document, states in as much detail as possible the job the programs are to do. Almost all of the information here will be some combination of narrative description,

mathematics, and tabular data. HIPO charts and data flow diagrams are also used to express required functional relationship, but not program logic design.

3.1. **Performance Parameters**

This subsection spells out the system's requirements for transaction rates, throughput, etc., as imposed by the problem environment. These requirements may be stated in terms of file capacities, acceptable timing constraints, permissible input rates, and so on. These requirements are stated in quantitative terms, with tolerance where applicable.

3.2. **Operational Requirements**

This subsection shows the functional requirements of the program system. The intent is to show all functional operations of the programs, the relationship among those functions, and the tie-in between program functions and other subsystem functions. Each program function is further defined in separate subsection as shown below.

3.2.1. **Function (Feature) 1**

3.2.1.1. Source and Type of Inputs

3.2.1.2. Destination and Types of Outputs

3.2.1.3. Functional Diagram

.....

3.2.n. **Function(Feature) n**

3.2.n.1. Source and Types of Inputs

3.2.n.2. Destination and Types of Outputs

3.2.n.3. Functional Diagram

3.3. **Data requirements**

This section defines data parameters that affect the program system design, for example, geographic co-ordinates for operational sites. The detailed definition of parameters includes descriptions of the data, definitions of units of measure, and precision requirements.

3.4. **Human Interfaces**

This section describes the interactions between the user and the software product defining the type of interactions, the content, general requirements of interaction shape, etc.

3.5. **Human Performance**

This section describes requirements involving human interactions with the program system, such as minimum times for decisions, maximum times for system responses, restrictions on program generated-displays.

Fig.2.2.a. Problem Specification Document Template

2.3 Analyst tasks

- *(1)* ***Meet the real customer.***
  - There are usually ***many people who are the "customer"*** and they all may make different demands on your analysts.
    - (a) A buyer who has in mind holding down costs;
    - (b) A staff analyst who wants a system with a lot of fancy gadgets;
    - (c) A contract administrator who may know little about the technical part of the job;
    - (d) A user who eventually will be saddled with your system.
  - The **analysts** should not assume that the members of the customer's organization *talk* to each other;
    - Let alone agree about what it is you *should deliver* to them.
    - If the **analysts** don't talk to the *right people* at the *right time*, they may not come away with a *clear understanding* of what this many-headed **customer** really *expects*.
      - *For **example**, if the analysts ignore the user until it's time to turn over a finished system, the results may be tragic.*
    - The **user** who has no part in specifying the system may be most reluctant to accept it.
  - The **analysts** must diplomatically find out, from the user point of view:
    - (a) Who controls *what.*
    - (b) Who has the *real power.*
    - (c) Who will eventually *use* the product.
- *(2)* ***Pick the customer's brains.***
  - The **analysts** must be skilled at finding out what the customer *really wants because it may be different from what was implied in a loosely worded contract.*
  - *They need to read* both what they're given and what's between the lines.

    - They need to *interview* people and try to *understand* what's really being said.
    - They must *listen* hard, *rephrase* what the customer has said, *feed* it back to him and ask: Is this what you mean?
- *(3)* ***Write it down.***
    - The **analysts** can write the **Problem Specification** in a large number of ways.
        - Some of those ways will be easy for designers to implement, some difficult, some impossible.
    - An **analysis** team *lacking programming experience* can generate big problems.
        - By the way, so can a team *with only programming experience*.
    - The **analysts** must state the problem in *clear* and *precise language that is acceptable to both the* ***customer*** *and to* ***designers****.*
- *(4)* ***Keep it simple***.
    - Prefer simplicity instead of complication.
- *(5)* ***Get it approved — gradually.***
    - As they write sections of the **Problem Specification**, the **analysts** should get tentative *customer approval* of what they have written.
    - Don't show the **customer** the *finished document* all at once, for if he doesn't like it, you're in bad shape.
    - Let the customer in on what you're doing as you *go along*.

Remarks

- Obviously, the analyst should represent many disciplines: *programmer, salesman, engineer, psychologist, and writer*.
- The Project Manager won't often find *all* the attributes you need in one person, but he can build **a group** that *collectively* has all the right *credentials*.
- Don't rely on *titles* when you *select* your people.
    - There are many people around called "**analysts**" because they don't neatly fit any other category.

3. Project Planning Activities

- Many programming projects are treated like mystery novels.
- A list of the *problems* which most often boil to the surface:
    - ***Poor planning***
    - ***Ill-defined contract***

    - ***Unstable problem definition***
    - ***Poor risks analysis***
    - ***Inexperienced management***
    - ***Political pressures***
    - ***Ineffective change control***
    - ***Unrealistic deadlines***
    - ***Poor planning***
- The list could be several pages long and two items would remain conspicuous, one by its presence, another by its absence:
    - Ever present is ***poor planning****.*
        - *Sometimes that means **failure** to consider the job from all angles.*
        - *Sometimes it means there is essentially **no** plan at all.*
    - Absent from *the list is **technical difficulty**.*
        - *That's not to say that many jobs aren't technically tough, but there are few that are beyond the "state-of-the-art."*
        - *In fact, if you agree to do a job that is beyond* the *state-of-the-art*, a job that requires some technical breakthrough, *that's* poor planning.
        - An obvious **exception** is a project whose objective is some sort of basic research.

3.1 The System

3.1.1 Definition

- The thing we're planning to build is called a *system.*
- *A **system** is a structured combination of interacting parts satisfying a set of objectives.*
- *There are examples* of systems everywhere:
    - *The solar system;*
    - *A power distribution system;*
    - *The human body;*
    - *The digestive system;*
    - *A corporation;*
    - *A pencil sharpener;*
    - *A computer system*.

- They **all** satisfy the definition.
- Every **system** is really a *subset* of some other system.
- We will frequently mention a "programming system".
    - But of course the **programming system** is really a *subsystem* of a data-processing system,
    - Which in turn may be a *subsystem* of a weapon system,
    - Which is a *subsystem* of a defense system,
    - And on and on it goes.

3.1.2 Characteristics of a System

- *(1)* ***Interactions***
    - ***By definition****, a system contains parts which must* interact with each other (fig.3.1.2.a).
    - *Controlling the interactions* among the parts becomes a **major task** as the system grows in size and complexity.
        - The number of *potential* ***interactions within a system grows as the number of elements in the system grows.***
    - *The management energy required to control, minimize*, and *simplify* the **interactions** within the system is significant.
        - **Interactions can be** *controlled* if we first recognize that they exist.
    - Some concepts can help to minimize the *effects of interactions*:
        - (1) *Modularity*
        - (2) *Interface definition*
        - (3) *Project organization*

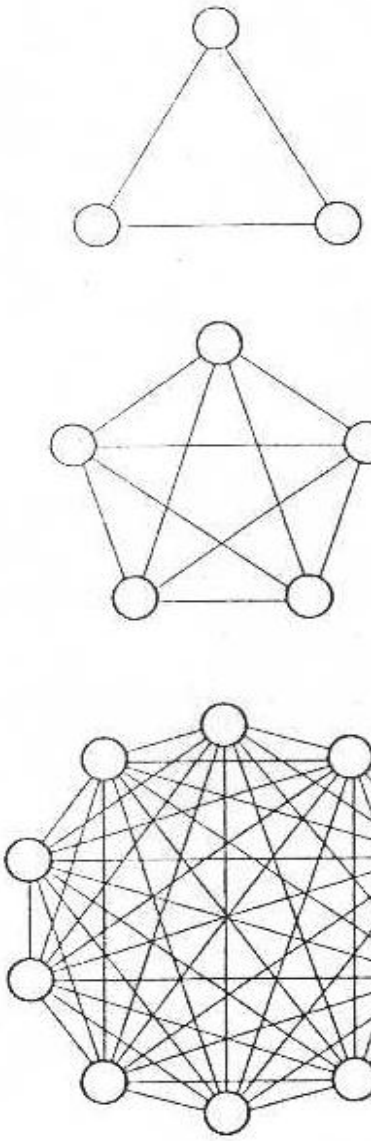

Figure 2.3. Interactions[2]

Fig.3.1.2.a. Interactions

- *(2)* ***Changes***
  - *Given any job that spans more than a couple of weeks of time, you can be sure that change will occur.*

- **Examples** of the kinds of changes are as follows:
  - (1) *Requirements changes:*
    - *The problem definition* your analysts labor over at the beginning of the job seldom stands *still*.
    - The larger the job, the more likely there will be shifts in the requirements.
  - (2) *Design changes:*
    - *The baseline design* is intended as the foundation for the programming effort.
    - But, as any homeowner knows, foundations shift, crack, and have to be patched.
    - A program system is no different.
    - Treat your baseline design as a *good start*, but expect it to be changed.
  - (3) *Technological changes:*
    - *The huge government-sponsored programming jobs (e.g., anti-ballistic missile systems, military command and control systems), but not only, are particularly vulnerable to technological change* for two reasons:
      - First, they span such *long periods* of time that new engineering and scientific developments (for example, in weaponry and data-processing equipment) are inevitable.
      - Second, the very nature of these projects is that they push the state-of-the-art and are often directly *responsible for* technological innovation.
  - (4) *Social changes:*
    - *Many projects are the unwitting victims of changes* in the way a large segment of society behaves.
      - ***For example,*** programs to handle payroll checks are constantly bombarded with **changes** because the tax laws change or because people make new demands on the system by having more and more personal deductions made.
  - (5) *People changes:*
    - *People leave, die, get sick, change jobs.*
    - *When you lose a key person from your own staff or when an important member of the customer's organization disappears, you have a potential problem*.

        - (6) *Corrections:*
            - *People make errors*.
            - They always have and they always will.
            - The errors may be major or minor, technical or administrative, outrageous or subtle, but they are errors nonetheless, and they must be fixed.
    - What's important about **changes** is that it be *controlled,* not *eliminated*.
        - ***For example*:** If there is a change in the **Problem Specification:**
            - (1) **Estimate** the *impact* of the change on project costs and delivery dates.
            - (2) If the customer still wants the change, **negotiate** a *contract* modification.
            - (3) **Issue** a formal *change notice*.
            - (4) **Get on** with the *job*.
    - In this context, and not only, it's very important to establish *accurate* and *meaningful* **baseline documents**.

3.2 Planning Tools

3.2.1 Project Plan Outline

- The toughest part of any writing job is *getting started*.
- That is the reason for is recommended to start with a *predefined* **Project Plan Outline**.

  It should be:
    - (1) A *model* published in literature [Me81], [Kr99].
    - (2) A *model* developed in your organization based on your history and experience.
- Use it as a *starter*, modify it to suit your situation, and you're on the way.
- Starting with the **Project Plan Outline** has many **advantages**:
    - (1) You *won't waste time* trying to decide how to break up the planning job.
    - (2) This outline has *built-in credibility* because it has been contributed to by many experienced programming managers.
    - (3) Having *any outline* to use as a *starter helps reduce* the number of rewrites, saving hundreds of man-hours later.

3.2.2 Bar Charts

- **Bar charts,** also called Gantt charts, are very familiar as planning tools.
- Bar charts are simple to *construct* and can be useful in depicting the *scheduling* or *expenditure* of resources versus *time* (fig. 3.2.2.a).
- Bar charts **advantages**:
  - Tells at a glance who is *assigned* to which tasks.
  - Figures in an intuitive manner the time projection.
- Bar charts usually operate with *tasks*.
  - The tasks can be as *big* or as *small* as you wish; what's important is that their size provides you the *control* you need.
  - A **project manager**, will probably want a chart that shows the large tasks;
    - The *large tasks* may be of *one to several* months' duration.

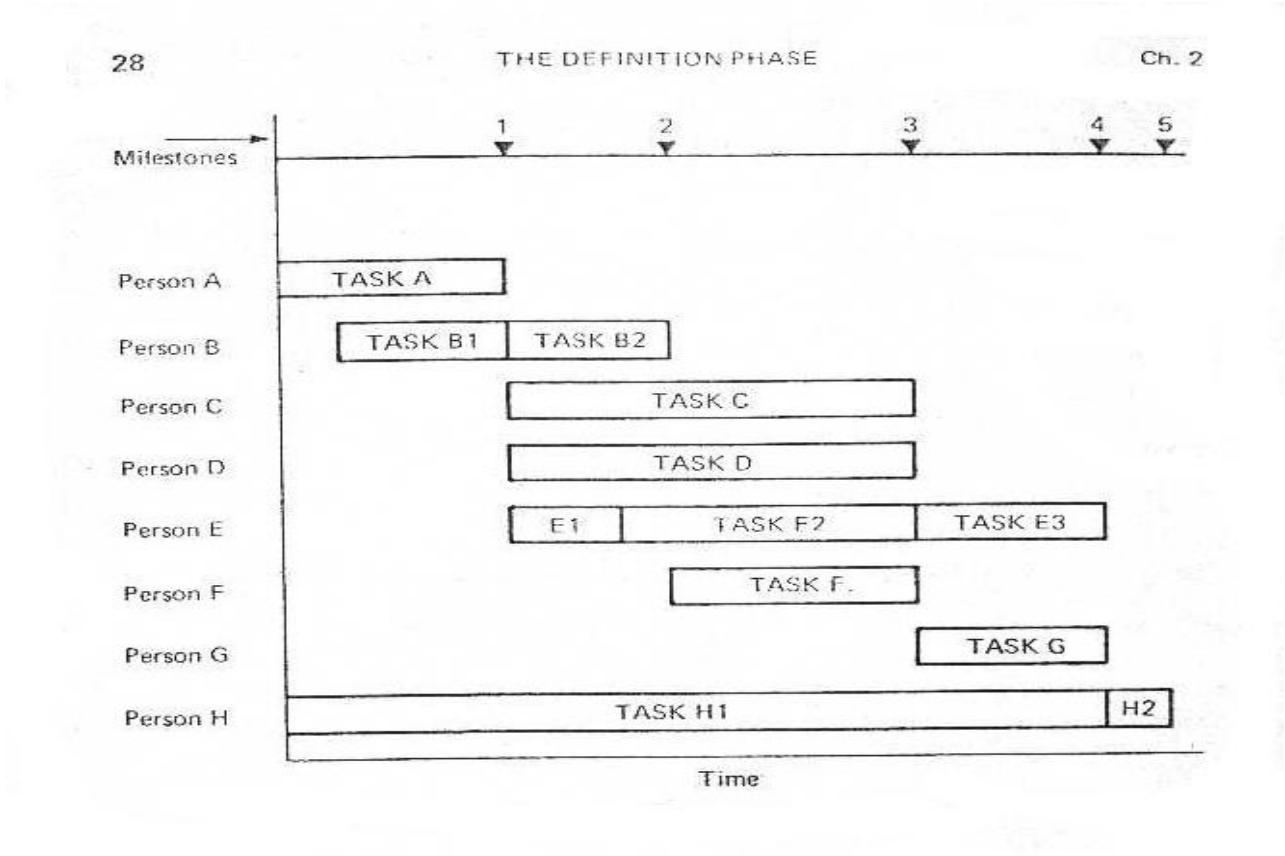

Fig.3.2.2.a. Bar Chart example

    - The **first-level managers** (those who directly supervise the programmers) will need charts showing the small tasks.
        - The **first-level manager**, however, could scarily exercise any control against a chart that is too broad.
        - He needs to *break* each task into *smaller pieces* so that trouble can be spotted early.
        - Theoretically, his maximum exposure is equal to the longest-duration task.
- How finely you break down your tasks depends on the *complexity* of that part of the job and the *experience* and *competence* of the programmers.
    - You might subdivide a given task into three *one-week* pieces for one programmer.
    - But for another you might assign that same task in one *two-week* chunk.
- As a **general guideline**:
    - The **minimum time** any task should be scheduled for is a ***week***.
    - A more **reasonable time** in most cases would be ***two weeks***.
        - If you try to **subdivide** tasks any **finer**,
            - (1) Your programmers will spend disproportionate amounts of time in progress reporting rather than in progress.
            - (2) Your first-level managers will become bookkeepers because they'll be preoccupied with logging and keeping track of all those little tasks.
    - The **maximum time** most tasks should be scheduled for is about a ***month*** — and again, ***two weeks*** would be more comfortable.
        - The point is that if a task is in trouble, you've got to know it soon enough to take remedial action.
- **Bar charts** may be used in many different ways:
    - (1) People, tasks, and time.
    - (2) Tasks versus time.
    - (3) Anything versus anything, as long as they help you in planning and controlling.
- The **tasks** shown on a bar chart:
    - The tasks shown on a chart should be *listed* and *defined* on a separate sheet.
    - Every task must require the *delivery* of a clearly defined product, such as a program module, a document, an artifact.
- There are many *planning tools* based on bar charts [**Microsoft Project Plan**]

3.2.3 Milestone Charts

- Webster Dictionary defines a *milestone* as a "significant point in development."
  - When you *define* **milestones** for your project**, don't forget** that word ***significant***.
  - *Don't necessarily make the end of each task* a **milestone**.
    - Instead, pick out those points in your schedules at which something *truly significant* should have been *completed* and
    - At which some *decision* is to be made, *(for **example**: continue, re-plan, get more resources, etc)*.
  - Base each **milestone** on something *measurable; otherwise, you won't know when you get there.*
- *Some **examples** of **poor** **milestones:***
  - *"Testing 50% complete."*
    - *Even if "50% complete" means something to you, chances are it will mean something different to someone else.*
    - *It's a poor milestone* because there is **no** sure way of knowing when you get there — it's not *measurable*.
    - *Even if based on the number of tests to be run, it's fuzzy because tests vary so much in complexity that one "half" may take a week to run and the other "half" may take three months.*
  - *"Coding 50% complete."*
    - *The same objections apply here.*
    - *Who knows that the coding is 50% complete?*
    - *Does this mean 50% of the anticipated number of lines of code?*
    - *Or 50% of the modules are coded?*
    - *Maybe 50% of the lines are coded, but they're the "easy" 50%, with the crunchers yet to come.*
    - *Again, the number is deceptive because it* means *different things* to *different people*.
- Here are a few examples of ***good** **milestones for the first-level manager.***
  - *"Detailed design on module X approved."*
    - *That's important even though that design may later be changed.*
    - *It means that the program module is laid out on paper and is ready for coding.*
  - *"Module X coded."*

        - *This is helpful but not as helpful as some other milestones because the code may change radically as testing goes on.*
    - *"Module testing of module X completed."*
        - *This means that the programmer has tested his individual module of code to his satisfaction and it's ready for integration with other tested modules.*
        - *It's a good milestone because the programmer at this point actually submits his physical program for the next level of testing.*
        - *It also means, as will be explained later, that the descriptive documentation for that module is completed in draft form, and this is something else that is measurable — something you can actually hold in your hands and see.*
    - *"Specification Y drafted."*
        - *Again, the document is either physically done or it isn't.*
        - *The manager can take a quick look at it and decide whether or not it's in good enough shape to be considered done, and hence whether or not the milestone has been met.*
- *The figure 3.2.3.a presents a list of* ***basic milestones*** for each *development phase* for the *project manager*.
    - You *can* modify the *milestone list* to suit to your project.
- How many **milestones** should you have?
- As usual, there is no magic number, but consider these **guidelines:**
    - (1) Each manager should have *his own* **set of milestones** *for the work to be done by the people reporting to him*.
    - (2) This means that the *first-level manager* has **milestones** *for the programmers' work*.
    - (3) The *project manager* has **milestones** *for the first-level managers' work*, and so on.
    - (4) However, one manager's **milestones** are not simply *the sum* of all those used by the managers *who report* to him.
    - (5) Each *manager* at each higher level must concentrate his energy on broader problems.
        - That, after all, is what *hierarchical organizations* are all about.
    - (6) Don't define the end of *every task* as a **milestone**.
        - Allow some *flexibility*.
    - (7) **Milestones** should be considered *important enough* that the people on the project will put out some *extra effort* in order to meet a milestone.

        - This means that you should have few enough that you don't have a *weekly milestone crisis*.
        - After three or four of these crises, your people will begin to yawn when you scream for help.
        - It is quite obvious that if a real schedule crisis should arise, this man would not get excited, because as far as he was concerned, his manager cried "wolf" too often.
    - (8) The **milestones:**
        - Used by your first-level managers should ordinarily be spaced at *least two weeks* apart.
        - For the next higher level, at *least three weeks* apart.
        - Next level, at *four weeks* apart.
        - And so on.

MILESTONE	WHEN
• Problem Specification written • Accepted by customer • Preliminary Acceptance Test Specification written • Accepted by customer	End of Definition Phase
• Preliminary Design Specification written • Accepted by customer	Middle of the Design Phase
• Design Specification completed • Accepted by customer • First distribution of the Programmer's Handbook • Design phase review completed • Integration Test Specification completed	End of the Design Phase

• System Test Specification completed • Final Acceptance Test Specification written • Site Test Specification written • Approved by customer • Program Documentation in clean draft form	End of Programming Phase
• System Test completed	End of System Test Phase
• Acceptance agreement signed • Customer training completed	End of Acceptance Phase
• Program Documentation corrected and delivered • System operational • Project History completed	End of Installation and Operation Phase

Figure 3.2.3.a. Project milestones

3.2.4 Activity Network Charts

- **Bar charts** are useful, but they have one **significant weakness:**
  - The bar charts don't adequately show *interdependencies among tasks or among people.*
- *To show these interdependencies an **activity network chart** is needed.*
- *An **activity network chart** is in fact a* weighted oriented graph. (fig.3.2.4.a) [Me81].

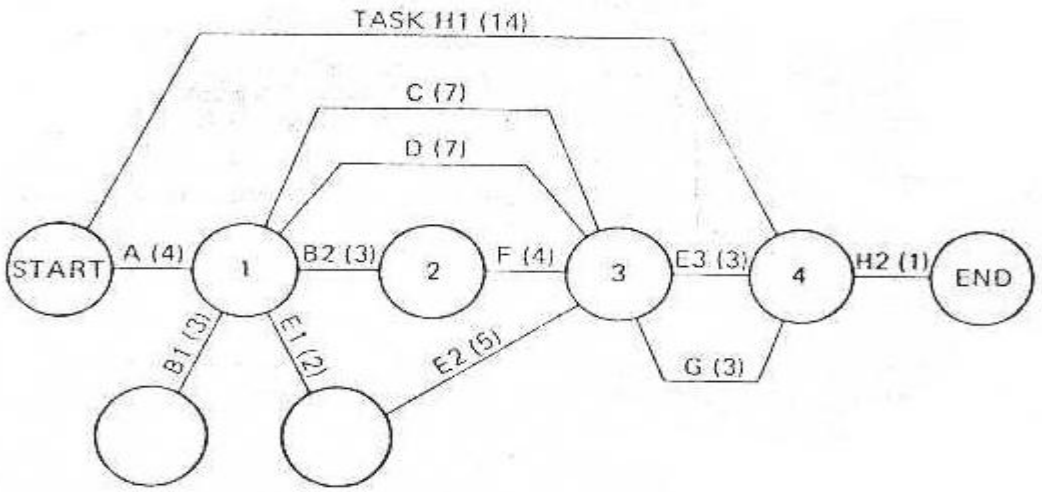

Fig.3.4.2.a. Activity network

- (1) The circles represent *events*;
- (2) The lines show *activities* that are required to get from one event to the next.
- (3) The estimated *units of time* (or other resource) for each activity, or task, are shown in parentheses.
- (4) The *lines* feeding an *event circle* from the left represent activities that must be finished *before* that event can occur.

- Normally, when a **network chart** is finished, there will be a *path* from beginning to end over which the total time required will be greater than for any other path.
  - This route is called the ***critical path***.
  - It demands *extra* monitoring and management attention, for if it slips, the end date slips.
- The **activity network:**
  - (1) It is **most valuable** during the *design* and *integration test* activities when management attention is directed toward the problems of **fitting things together.**
  - (2) It is **less useful** during the middle time when individual program modules are being written and tested, because then the focus is on **individual pieces** of the system more than on their interactions.

    - (3) Since the usefulness of these charts ails off after design is completed, they are often *abandoned* at that time.
- This may explain why **PERT** has had only spotty success in programming.
    - **PERT (Program Evaluation and Review Technique)** is a formalized, usually computerized implementation of the activity network idea.
        - There are several basic forms and many specific versions of **PERT** networks.
        - The literature abounds with books and articles on the subject.
        - The effort required to keep **PERT activity networks updated as the project progresses is** *costly* and it *diverts* managers from supervising individual tasks.
- It is not uncommon for an **activity network** to *deteriorate* during the **Programming Phase**.
    - If it falls behind, abandon it.
    - Don't blindly update it if it's no longer useful.
- The **activity network** could be drawn according to a *time scale*, with the spacing between event circles representing *calendar time*.
    - These charts require a lot of work and tend to cover walls.
- **Recommendation**
    - **Bar charts** to be used to show *calendar schedules*,
    - **Activity networks** to be limited to showing *interdependencies*.

4. Software Size Estimation

4.1 Background

- The principal *reason* you should *estimate the size* of a software product is to help you to *plan* the product's development.
- The quality of a *software development plan*, in turn, generally depends on the quality of the *size estimate*.
- With a good plan:
    - (1) You have the basis for *funding* and *staffing* the work.
    - (2) You know *what* has to be done, *when*, and by *whom*.
    - (3) You also know *how long* it will take and understand the *critical dependencies* or other *constraints*.

- *Poor software planning* is one of the principal reasons software projects get into trouble. [Humphrey 89].
    - A frequent cause of poor plans is a ***poor size estimate***.
- Accepted practice in *engineering*, *manufacturing*, and *construction* is to base development plans on *product size estimates*.
- ***Building construction*** provides a good example.
    - Competent builders get *detailed information* about the *kind of building* you want before they give you a construction estimate.
    - Then they make *materials and labor cost estimates* based on their prior experiences.
    - They use *historical factors* for such items as the *costs per square foot* for wiring, carpentry, plastering, and painting.
    - Experienced builders can often *estimate large projects* to within one or two percent of the actual finished cost.
- The degree to which you can *accurately and precisely plan* a job depends on what you know about it.
    - (1) At the *earliest* or *pre-proposal stage*, you have only a general *idea* of the product requirements.
        - About the only way to make an estimate is by *analogy* to previous products.
            - This situation is like that faced by a *builder* when a home buyer wants the price for a *one-story ranch house* with *three bedrooms*.
            - An experienced builder could show the buyer several homes that fit this description and give their prices, but would not quote a price for such a general requirement.
    - (2) After the *pre-proposal phase*, you know progressively more about the planned product.
        - You can then make *more-refined estimates* of job size.
            - Again, it is instructive to consider what *builders* do when they need a competitive estimate for a multi-million dollar construction job.
            - They work with detailed *architectural specifications* and a group of *proven subcontractors*.
            - They calculate the linear feet of interior and exterior walls, the square feet of floor and ceiling, and the square feet of concrete.
            - They can then determine the number of studs and the amount of linear feet of finished lumber required, and the plumbing, wiring, and masonry costs.

        - When they need an accurate estimate, they define the project in **great detail**.
- ***Estimates for large software jobs*** can be handled in much the same way.
- To make an **accurate estimate**
    - Start with a *design specification*.
    - *Examine* and *estimate* each part of the job.
    - This estimate requires **separate estimates** for:
        - Each *software component*,
        - Each major *document*,
        - The *test cases*,
        - The *installation planning,*
        - *File conversion,*
        - *User training*.
    - The **program components** may have different sub-elements.
        - If there are *screens* to develop, *reports* to generate, or *functional logic* to design, these must also be estimated.
    - For **large software products**, there is a great deal of potentially useful *detail*.
        - If you want to make accurate estimates, you will have to *define* all these details and *consider* each one in the estimate.

4.2 The Size Estimating Framework

- The generalized planning framework is presented in the figure 4.2.a. [Humphrey97].
    - The tasks are shown in the *rectangles.*
    - The various data, reports, and products are shown in the *ovals*.
- In *estimating the size*
    - (1) First define the *requirements*.
    - (2) Then produce a *conceptual design*.
    - (3) Compare the elements of this conceptual design with the *known size* of program elements you have *previously developed*.
        - This process helps you to judge the size of the parts of the **new product.**

- The most complicated part of SW size estimating resides in *characterizing* the product elements and *relating* them to your historical experience.

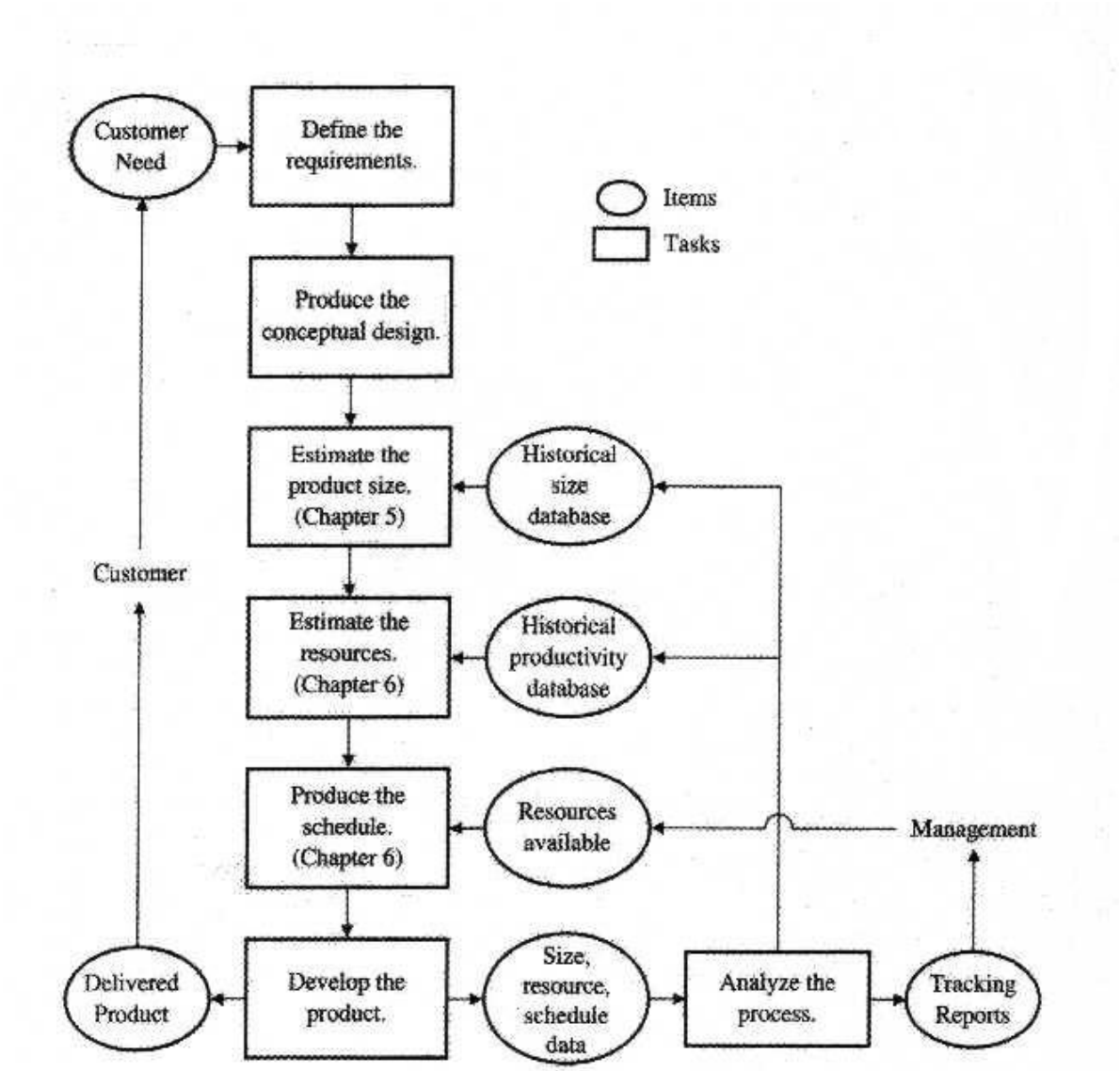

Fig.4.2.a. Project Planning Framework

4.2.1 The Size-Resource Relationship

- Size is an accurate *predictor* of the resources required to develop a product [Boehm]

- The relation between *size* and *required resources* has led to the development of a number of **cost models** such:
  - **COCOMO**
  - **PriceS**
  - **SLIM** [Boehm, Park 89, Putnam]
- While these models can be generally useful, they must be *calibrated* for *each using organization*.
  - Each such organization must thus gather its own *historical data* and use them to set the *factors* in its cost models.
- The models also require a ***size estimate*** as input.
- Before you can use them, you must estimate the *product size*.
  - The accuracy of the *models* is thus limited by the accuracy of the *size estimates*.
  - So, even when you use an *estimating model*, you need an accurate size estimate.

4.2.2 Some Estimating Experiences

- Hihn reports several studies that show *size estimation* errors as high as 100 percent [Hihn].
- *Size estimates* that are seriously in error result in:
  - (1) Poor *resource estimates.*
  - (2) Unrealistic *project schedules*.
- This is one reason why software groups get into trouble.
  - They either had no *initial estimate* or the one they had was seriously in error.
  - Under these conditions, projects often start in trouble and never recover.
- The unknowns early in the software development process cause *major cost-estimating inaccuracies*.
  - Boehm states that:
    - (1) These inaccuracies range up to **400%** in the early feasibility phases.
    - (2) Up to **60%** or more during the requirements phase.
    - (3) They only decline to **25%** or less during and after detailed design.
  - Clearly, if you make a development estimate *too early*, when you know less about the product to be built, your estimate will likely be *less accurate*.

4.2.3 Size Estimating Criteria

- A widely usable ***size estimating method*** should meet the following criteria:
- (1) It should use *structured* and *trainable* methods.
    - A *structured method* facilitates training and process improvement.
    - It also permits you to track and improve the *estimating method* itself.
- (2) It should be a method you can use during all phases of *software development* and *maintenance*.
    - You use size estimates *early* in the development cycle to make realistic plans and commitments.
    - During development, you may need to *modify* your plans to *adjust* for changes.
    - On larger projects, it is also good practice to *periodically* re-estimate the product size and needed resources at the end of every major phase.
    - As these plans become more accurate, they provide a *firmer basis* for *project management* and *tracking*.
- (3) It should be usable for all *software product elements*.
- (4) It should handle not only the *code*, but also for:
    - *Files*
    - *Reports*
    - *Screens*
    - *Documentation*
- (5) It should also be suitable for the *types of systems* and *applications* you work on as well as for *new product* development, *enhancement*, and *repair*.
- (6) It should be suitable for statistical analysis.
    - A statistically based size estimating method will provide the means for you to *adjust* your estimating parameters based on your *historical data*.
- (7) It should also be adaptable to the types of work you are likely to do in the *future*.
    - As you build estimating data and experience, you will then be building an asset of continuing value.
- (8) It should provide the means to judge the *accuracy* of your *estimates*.
    - You should always compare each estimate with the *size of the actual resulting product*.

- By reviewing these comparisons, you will often see the causes of your errors and be better able to *adjust* and *improve* your estimating method.

4.3 Size Estimating Methods

- To measure SW projects different *metrics* are used.
- A **metric** for a SW project is in fact a method permitting to characterize from *quantitative* point of view the *product*.
  - In other words a **metric** permits a *quantitative estimate* a SW project.
- There are 2 categories of metrics used in SW project *evaluation*:
  - (1) **Size oriented metrics**
    - Mainly consisting in estimation of the *number of source line of code*.
  - (2) **Function oriented metrics**
    - Mainly consisting in estimating of the *complexity of the functions* realized by the software product resulted from the development process.

4.3.1 Size Oriented Metrics

- **Size oriented metrics** use as measure the *number of source line of code* or the *number of delivered source instructions*.
- Usually the *number of source line of code* is grater than the *number of delivered source instructions* for at least two reasons:
  - (1) During the development process there is the possibility that certain *modules* to be coded more times, in a successive refinements process until the final version is achieved, or the *optimal variant* to be selected from different issues.
  - (2) Some parts of the *constructed models* will be not **included** in the final product because there are *simulations*, *experiments* or *trial versions* of the developed product
- The usual *measurement software size* units are:
  - **Number of line of code LOC** (**L**ine **O**f **C**ode)
  - **Number of kilo-line of code KLOC** (**K**ilo-**L**ine **O**f **C**ode)
  - **Number of source line of code SLOC** (**S**ource **L**ine **O**f **C**ode)
  - **Number of kilo-source line of code KSLOC (Kilo-Source Line Of Code)**
  - **Number of delivered source lines DSI** (**D**elivered **S**ource **I**nstructions)

- **Number of delivered kilo source lines KDSI** (**K**ilo **D**elivered **S**ource **I**nstructions)

- There are some **classical methods for size estimation** of the SW projects based on these metrics:
  - (1) **Wideband-Delphi Method**
  - (2) **Fuzzy-Logic Method**
  - (3) **Standard-Component Method**

4.3.1.1 Wideband-Delphi Method

- The **Wideband-Delphi** estimating method was originated by the Rand Corporation and refined by Boehm.
  - (1) It calls for *several engineers* to individually produce *estimates.*
  - (2) Then uses a Delphi process to *converge* on a consensus estimate.
- The **procedure** is essentially as follows [Humphrey 89]:
  - (1) A group of **experts** is each given the program's specifications and an *estimation form*.
  - (2) The members of the group meet to discuss *project goals*, *assumptions*, and *estimation issues*.
  - (3) The members of the group then each *anonymously* list *project tasks* and estimate *size*.
  - (4) The estimates are given to the estimate moderator, who tabulates the *results* and returns them to the *experts*, as illustrated in figure 4.3.1.1.a:
    - For each expert only *personal estimate* is identified.
    - All other's estimates are *anonymous*.
    - There is also identified the *median estimation.*
  - (5) The **experts** meet to discuss the anonymous results.
  - (6) The **experts** each eventually review the *tasks* they have defined and their *size estimates*.
  - (7) The cycle continues at step (3) until the estimates *converge* to within an acceptable range.
- For reasonably large programs, the estimators make *simultaneous estimates* for several *product components*.
  - At the end of the estimating process, these *estimates* are combined to produce the *total final estimate*.

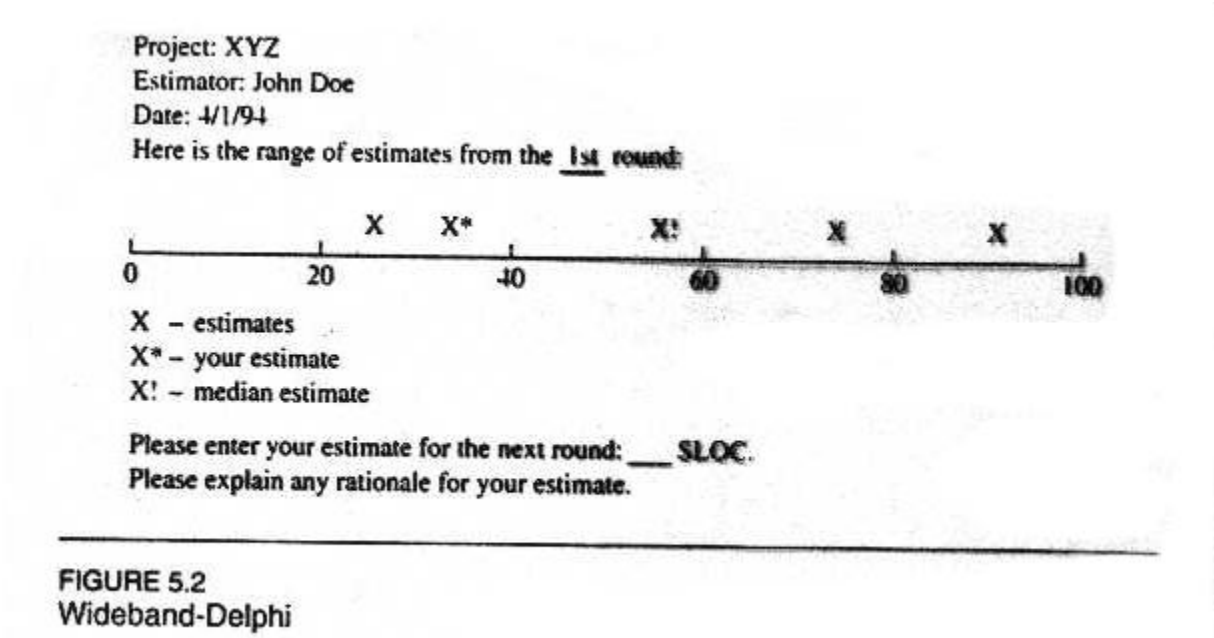

FIGURE 5.2
Wideband-Delphi

Fig.4.3.1.1.a. Estimation Form for Delphi Method

- The estimating process is run by a ***moderator*** who should always be careful never to reveal the *source* of any *individual estimate*.
- The appropriate attitude is that no one knows the *right answer*.
  - Everyone has a *partial view*.
  - The purpose of the Delphi process is to *share* those views.
- By encouraging the participants to discuss the project tasks, a skilled moderator can facilitate very *informative discussions*.
  - The person who made a particularly high or low *estimate* is sometimes willing to explain why that value was picked.
  - The resulting discussions often shed light on the problem and surprisingly often convince the other engineers to *change* their estimates.
  - While confidentiality will often become a non-issue, the decision to reveal any *estimator's identity* must be left up to *that estimator*.
- This method can produce quite *accurate estimates*.
- It also often provides:
  - (1) A solid foundation from which to *start the work.*

- (2) The commitment of the engineers *to get the job done* for the cost and time *they put* in the estimate.

4.3.1.2 Fuzzy-logic Method

- Putnam describes the **fuzzy-logic** *estimating method* where:
  - (1) Estimators *assess* a planned product.
  - (2) Then *roughly judge* how its size *compares* with historical data on prior products. [Putnam].
- ***For example***, you could break the *sizes* of all your organization's ***previously developed products*** into **ranges** and **subranges** size categories, as shown in Table 4.3.1.*2*.a.

Subrange Range	Low-LOC	Size-LOC	High-LOC
Very Small	1000	2000	4000
Small	4000	8000	16000
Medium	16000	32000	64000
Large	64000	128000	256000
Very Large	256000	512000	1024000

Table 4.3.1.2.a. Fuzzy-logic size rangers

- With sufficient data, these *gross size ranges* can be subdivided into **more** detailed categories *(subranges)* as shown in Table 4.3.1.2.b.

Subrange Range	Very Small LOC lg(LOC)	Small LOC lg(LOC)	Medium LOC lg(LOC)	Large LOC lg(LOC)	Very Large LOC lg(LOC)
Very Small	1,148 (3.06)	1,514 (3.18)	2,000 (3.30)	2,630 (3.42)	3,467 (3.54)
Small	4,570 (3.66)	6,025 (3.78)	8,000 (3.90)	10,471 (4.02)	13,804 (4.14)
Medium	18,197 (4.26)	23,988 (4.38)	32,000 (4.50)	41,687 (4.62)	54,954 (4.74)
Large	72,444 (4.86)	95,499 (4.98)	128,000 (5.10)	165,958 (5.22)	218,776 (5.34)
Very Large	288,403 (5.46)	380,189 (5.58)	512,000 (5.70)	660,693 (5.82)	870,964 (5.94)

Table 4.3.1.2.b. Fuzzy-logic size rangers and logarithms

- To show how these ranges are **constructed**, Table 4.3.1.2.b. gives **the decimal logarithms** of the ranges (in blue).
  - (1) The *size ranges* are evenly spaced on a *logarithmic scale* with ratio 0.60 (vertical).
  - (2) Each *division* is subdivided into *five subrange categories*, with the increment 0.60/5=0.12 (horizontal)
  - (3) The resulting *25 categories* provide the basis for making reasonably *precise estimates*.
- ***EXAMPLE*: The **construction** of a simple **estimation table.**
  - Suppose your smallest program had 173 LOC and your largest 10,341 LOC.

- How you might divide a size range into categories (Table 4.3.1.2.c)?

Subranges Ranges	Small	Medium	Large
Very Small	104 (2.016)	173 (2.238)	288 (2.460)
Small	288 (2.460)	481 (2.682)	802 (2.904)
Medium	802 (2.904)	1338 (3.126)	2230 (3.348)
Large	2230 (3.348)	3719 (3.570)	6202 (3.792)
Very Large	6202 (3.792)	10341 (4.014)	17234 (4.236)

Table 4.3.1.2.c. Construction of a simple ranges table

- (1) Divide the range into *five equal sets* on a logarithmic scale:
  - (1)Take the *base 10 logarithms* of 173 and 10,341, giving *lg(173) = 2.238* and *lg(10,341) = 4.014*, respectively.
- (2) Calculate the *difference* between these two numbers: *4.014 - 2.238 = 1.776.*
- (3) To get five ranges, *divide* the difference between these numbers by 4, giving *1.774/4= 0.444.*
- This is the *logarithmic increment* between each **range category**.
- (2) Calculate the range values:
  - The lowest range value is the 173 LOC, number you started with, that means as logarithm *2.238.*

        - The next range has a logarithm of *2.238+0.444=2.682* and a value of 481 LOC.
        - Similarly, the other values are 1338, 3719, and 10341 LOC (increasing by *logarithmic increment 0.444*).
    - (3) To divide each range into *two subranges*, the *logarithmic increment (0.444)* would be cut in half, to *0.222*.
        - The smallest subrange will be 2.238-0.222=2.016 that means 104 LOC.
        - The next subrange of Very Small category is 2.238+0.222=2.460 that means 288 LOC.
    - (4) Continue in the same manner for each of the ranges' value.
    - (5) Now you know the top points, bottom points, and midpoints of each of the *five size ranges* and you can *construct the table* (Table 4.3.1.2.c):

- To make an **estimate** is necessary to judge which of this *categories* most closely resembles the *new project.*
- To do this :
    - (1) First decide into which *gross size range* the new project fits.
    - (2) Then, by comparing the new project with the *known characteristics* of the projects in this size range, you place it in a *subrange*.
    - (3) While this estimating technique gives only a crude size judgment, it does provide an orderly way to *compare* the **sizes** of planned projects with the **sizes** of previously developed ones.
    - (4) The **Wideband-Delphi technology** can be used as support for this method in order to obtain *more accurate results*.
- In using this method, several **considerations** are important:
    - (1) *Historical data* are required on a large number of projects.
        - You need to have a *reasonable number* of historical products in each *size category*.
    - (2) Is necessary to have data on some products in every category.
        - Otherwise you will not have examples with which to compare all the new products you might have to estimate.
    - (3) You can use *any number of size ranges*, but they should cover the *entire span* of expected project sizes.
    - (4) You should then *extend* these ranges up or down as you get further data.
    - (5) You should *not change the existing ranges*, however, because you will become adept at judging programs in terms of the ranges you have learned.

        - When you change the ranges, your estimating accuracy will likely suffer, at least until you **relearn** the new ranges.
    - (6) To provide a meaningful number of examples in each size range, you will need a considerable amount of historical data.
        - If there is not at least a modest distribution in a given range, an estimator will have trouble judging the relative size of a new program.
    - (7) Until you have a *large base of data*, therefore, it is wisest to distribute your historical data into no more than about *five major categories*
        - Make **comparative judgments** at this level.
        - To do this properly, of course, you should be familiar with all the programs in the database.
- One **problem** with the fuzzy logic method is that the **size** of major programming applications has *historically grown* by about an order of magnitude *every 10 years*.
- Reasonably accurate estimates of the very largest category are thus particularly important.
- Unfortunately, any gross sizing method will not likely be of much help in estimating a new program that is much larger than anything you have previously built.
- To handle this situation, you would have to *subdivide* the new product into *smaller components* and *estimate each component*.

4.3.1.3. Standard-Component Method

- The **standard-component** estimating method is described by Putnam as a way to use an *organization's historical data* to make *progressively more-refined* program *size estimates* [Putnam].
    - (1) Start by *gathering data* on the *sizes* of the various levels of program abstractions you use, for example, *subsystems*, *modules*, and *screens*.
    - (2) Then judge *how many* of these components will likely be in your new program.
    - (3) Also judge the largest number you could imagine being in the program as well as the smallest number.
    - (4) Combine these estimates by taking *four times the most likely number* and *adding* both the *smallest* and *largest* numbers.
    - (5) Then divide this total by six to give the **likely number of that component**.
        - The *standard deviation* of the final estimate should be roughly one sixth of the difference between the smallest conceivable number and the largest conceivable number.

- In **equation form**, this procedure is:

Estimated Number = [smallest_conceivable_number + 4*(likely number)
+ largest_conceivable_number]/6.

- (6) Based on this **numbers**, and based on **historical data** about standard components acquired by your organization during the time, multiply the estimated numbers of components with their length in **SLOC**.
- (7) Add the calculated numbers to obtain the **total size** of the product.
- (8) Don't forget after project finalization to *review the dimensions in SLOC of the standard components* on the basis of the **post-mortem analysis.**

- **Example:** Some of the values Putnam uses for standard-component estimating are shown in an **example** in Table 4.3.1.3.a. [Hu95]
  - This **example** includes several *files, modules*, *screens*, and *reports*.
  - Here, the **file components** are each average *2535* SLOC.
  - In this application, the estimator was confident that there would be:
    - Least 3 file components.
    - The largest number expected was 10.
    - And the more likely number was about 6.
  - Using the formula given above, you get an estimate of *6.17 file components*.
    - Because each file is expected to contain the historical average of *2535* SLOC, you get a total of *15,633* SLOC.
  - The other components' SLOCs are calculated in the same **way** and totaled to give the final estimate *46,359* SLOC.

Standard Component	**SLOC *per* Component**	**S**	**M**	**L**	**X=[S+4*M+L]/6**	**SLOC**
SLOC	1					
Object	0.28					

Instructions						
Files	2,535	3	6	10	6.17	15,633
Modules	932	11	18	22	17.5	16,310
Subsystemes	8,175					
Screens	818	5	9	21	10.3	8,453
Reports	967	2	6	11	6.17	5,963
Interactive Programs	1,769					
Batch Programs	3,214					
Total						**46,359**

Table.4.3.1.3.a. Example of component estimating

4.3.2 Function Oriented Metrics

- Function Oriented Metrics consists in *estimating* of the complexity of the functions implemented by the SW product to be developed.
- There are 2 categories of **function oriented metrics**:
  - (1) **Functional points oriented metrics**
  - (2) **Feature points oriented metrics**
- There are also some **specific estimation methods** based on these metrics:
  - (1) **Function-Point Method**
  - (2) **Characteristic-Point Method**

- (3) **Proxy-Based Method**

4.3.2.1 Function-Point Method

- The *function-point method*, invented by Albrecht at IBM in 1979, is probably the most popular method for estimating the size of commercial software applications.
- Albrecht:
  - (1) Has identified ***five basic functions*** that occur frequently in commercial software development.
  - (2) Has categorized them according to their *relative development complexities*.
- The definitions of these **five basic functions** are as follows: [Jones]
  - (1) ***Inputs:***
    - Inputs are *screens* or *forms* through which human users of an application or other programs add new data or update existing data.
    - If an input screen is too large for a single normal display (usually 80 columns by *25* lines) and flows over onto a second screen, the set counts as one input.
    - Inputs that require a unique processing are what should be considered.
  - (2) ***Outputs:***
    - Outputs are *screens* or *reports* that the application produces for human use or for other programs.
    - Note that outputs requiring separate processing are the units to count;
      - For **example**, *in a payroll application, an output function that created, say, 100 checks would still count as one output.*
  - (3) ***Inquiries:***
    - Inquiries are *screens* that allow users to interrogate an application and ask for assistance or information.
      - For **example** Help screens.
  - (4) ***Data files:***
    - Data files are *logical collections of records* that the application modifies or updates.
    - *A file can be, for **example**:*
      - *A flat file such as a tape file.*
      - *One leg of a hierarchical database.*

        - *One table within a relational database.*
        - *One path through a network database.*
    - (5) ***Interfaces:***
        - Interfaces are *files shared* with other applications and include:
            - *Incoming* or *outgoing tape files*.
            - *Shared databases*.
            - *Parameter lists*.

- To make a **function-point** estimate:
    - (1) Review the *requirements* and count the *numbers* of *each* **Function Type** the program will likely need.
    - (2) Enter these *numbers* in the **Basic Counts** area in Table 4.3.2.1.a.
    - (3) *Multiply* them by the **Weights** area to produce the **Total** *numbers* of function points in each category.
    - (4) Add the obtained values.
    - (5) The obtained number is **Unadjusted Total** of functional points of the SW you have estimated.

Basic Counts	**Function Type**	**Weights**	**Total**
	Inputs	x 4	
	Outputs	x 5	
	Inquiries	x 4	
	Logical Files	x 10	
	Interfaces	x 7	
	Unadjusted Total		

Table 4.3.2.1.a. Function-Point Categories

- **An Example** [Hu95]
  - The example is stated in Table 4.3.2.1.b.
  - A count of the application's requirement statement might yield a total of:
    - 8 inputs
    - 12 outputs
    - 4 inquiries
    - 2 logical files
    - 1 interface.
  - Based on proposed estimation algorithm, multiply these numbers by the weights to give the values in the total column.
    - For **example,** the 8 inputs are multiplied by a weight of 4, giving a total of 32 (Table 4.3.2.1.b.).
  - Add the obtained values for each of the function type
  - The ***Unadjusted total*** *of function-point* would then be *135.*

Basic Counts	Function Type	Weights	Total
8	Inputs	8 x 4	32
12	Outputs	12 x 5	60
4	Inquiries	4 x 4	16
2	Logical Files	2 x 10	20
1	Interfaces	1 x 7	7
	Unadjusted Total		**135**

Table 4.3.2.1.b. Function-point category example

- Jones, in his definitive text on this subject, suggests that this *unadjusted function-point total* to be **adjusted** by a **Complexity Multiplier** based on 14 **influence factors**
    - In Table 4.3.2.1.c are shown as an example the proposed 14 *influence factors.*
    - The *influence factors values* are selected from 0 to *5,* depending on the degree to which you judge that the particular factor falls between very simple to very complex or very low to very high, and so on.
    - Because there are *14 factors* and their values can each range from 0 to *5,* the total *Sum_of_the_influence_factors* then ranges from 0 to 70.
    - The formula to calculate the **Complexity Multiplier** is:

      Complexity Multiplier=0.65+0.01*(Sum_of_the_influence_factors)

    - From the above formula, it can be seen that the value of the **Complexity Multiplier** varies from *0.65 (when Sum=0) to 1.35 (when Sum=70)*
        - *That means the influence of the* **Complexity Multiplier** *is ranged from minus 35 percent to plus 35 percent.*
- To finalize the **size estimation** in this new conditions:
    - (6) Estimate the **impact** of each influence factor by values from 0 to 5.
    - (7) Add the assigned values for the factors and obtain the Sum_of_influence_factors.
    - (8) Calculate the **Complexity Multiplier** based on the formula:

      Complexity Multiplier=0.65+0.01*(Sum_of_influence_factors)

    - (9) Multiply the Unadjusted Total of functional points with the **Complexity Multiplier** in order to obtain the **Function points** number for the estimated SW.
- **For example:**
    - Suppose that you have selected the following values for the influence factors for your project (Table 4.3.2.1.c)

Nr.crt	Factor	Influence 0 = non influence 1 = little influence 2 = moderate influence 3 = average influence 4 = significant influence 5 = strong influence
1.	Data communications	2
2.	Distributed functions	0
3.	Performance objectives	3
4	Heavily used configuration	3
5.	Transaction rate	4
6.	On line data entry	4
7.	End-user efficiency	3
8.	On line update	2
9.	Complex processing	3
10.	Reusability	2

11.	Installation ease	3
12.	Operational ease	4
13.	Multiple sites	5
14.	Facilitate change	3
	Sum of influence factors =	**41**
	Complexity Multiplier = 0.65+0.01(Sum of influence factors)	**1.06**
	Function points=Complexity Multiplier ***Unadjusted Function Points	**135*1.06=143**

Table 4.3.2.1.c. Function-point influence factors (an example)

- The influence factors you selected from Table 4.3.2.1.c. are used to calculate the function point adjustment (**Complexity Multiplier**) of 1.06.
- Multiply the unadjusted function-point total of *135* by this factor and you get a **total** of *143* function points.

- As useful as they are, function points are **not** *fully satisfactory* for two reasons.
  - (1) **First**, they cannot be *directly measured.*
  - (2) **Second**, they are not sensitive to *implementation decisions.*
- Jones addressed this first problem by producing a number of *function-point* ***conversion*** *factors* that permit to count **LOC** and calculate the program's likely function point content [Jones].
- The second issue, *implementation independence*, is more debatable.
  - While implementation independence is an advantage in *cross-language* or cross-system comparisons, it is a disadvantage in development *cost estimates*.
  - Development costs are typically sensitive to:
    - (1) *Implementation language.*

        - (2) *Design style.*
        - (3) *Application domain.*
    - So either the estimating method must consider them or some **other allowance** must be made.
- The function-point method continues to be refined.
    - Copies of the latest **International Function Point Users Group (IFPUG)** guidelines and standards can be obtained on internet. [Sprouts, Zwanzig]

Function-Point Method Summary

- **"Basic" Function Points (BFP):** 4(EI) + 5(EO) + 4(EQ) + 10(ILF) + 7(EIF) (with ± 35% Complexity Adjustment)
- **Unadjusted Function Points (UFP):** Weight Five Attributes as Simple, Average, or Complex (Table 4.3.2.1.d.)

Attribute	Complexity			Total
	Simple	Average	Complex	
External Inputs	3	4	6	
External Outputs	4	5	7	
External inQuiries	3	4	6 (or 7)	
Internal Files	7	10	15	
External InterFaces	5	7	10	

Table 4.3.2.1.d. Weight attributes

- **Adjusted Function Points (AFP):** UFP (0.65 + [0.01(SIF)])
  (SIF is Sum of the Influence Factors: Sum of 14 Factors, Rated 1 to 5 for Influence [0 - None, 1 - Little, 2 - Moderate, 3 - Average, 4 - Significant, 5 - Strong]; Ratings Defined for Each Factor) (Table 4.3.2.1.e.)

14 Factors:	
1. Data Communications	2. Distributed Data Processing
3. Performance Objectives	4. Heavily-Used Configuration
5. Transaction Rate	6. On-Line Data Entry
7. End-user Efficiency	8. On-Line Update
9. Complex Processing	10. Reusability
11. Conversion and Installation Ease	12. Operational Ease
13. Multiple Site Usage	14. Facilitate Change

Table 4.3.2.1.e. Influence Factors

4.3.2.2 Conversion of Function Point to SLOC

- It is sometimes necessary to convert from *SLOC* to *function points*, or vice-versa.
- Several software cost models, such as Stage 2 of **COCOMO II**, allow the user to input *function points* (or a variant of a function point).
  - As result they must **convert** function points to SLOC because the model's algorithms are based on SLOC.
- The opposite situation occurs in the CHECKPOINT® model where *SLOC inputs* must be converted to *function points*.
  - This conversion process is sometimes called "backfiring."

- To help in this *conversion process*, sets of SLOC to function point ratios have been developed.
  - Table 4.3.2.2.a shows ratios for various languages based on research by **Jones** and **Reifer**.
  - Jones specifies ***language levels*** that show the number of equivalent assembly language SLOC generated by one SLOC of the specified language.
  - Jones further states that ***language levels*** are useful for converting size **from one language to another**, and for assessing relative ***language productivity***.
  - The relationship between **language level** and **productivity** is not linear.

Language	Jones	Jones	Reifer
	Language Level	**SLOC/FP**	**SLOC/FP**
Assembler	1	320	400
COBOL	3	107	100
FORTRAN	3	107	105
Ada (1983)	4.5	71	72
PROLOG	5.0	64	64
Pascal	3.5	91	70
PL/1	4.0	80	65

Table 4.3.2.2.a. Conversion from SLOC/FP

- While function point to SLOC conversion ratios is useful, and often necessary, they should be used with caution.

- Table 4.3.2.2.a. illustrates that:
  - While the two researchers agree on the ratios for some languages such as **Ada**, they differ on the ratios for other programming languages such as Pascal and PL/1.
  - For the **COBOL** language, research by Henderson showed that the conversion ratios are inconsistent both within and between databases.
  - Therefore, for some languages it appears that backfiring should not be used, and for *cost estimation*, it is probably best to use a model for which the algorithms are based on the *user's size measure* (i.e., calibrated parametric sizing models).

4.3.2.3 Characteristic-Point Method

- The metrics based on *characteristic points* was proposed by C.A. Jones
  - [*C.A. Jones, "A short History of Function Point Method and Feature Points Software Productivity" Research, Inc., Burlington, MA, June, 1986*].
- The method is **not** very much different from ***function-point method***.
- In fact there are three differences:
  - (1) **Function-points** are substituted with **characteristic points**.
  - (2) The associated weights have different values.
  - (3) A new parameter is added: *algorithms* with weight 3.
- The **characteristic-point method's steps** are:
  - (1) Estimate the *basic count* associated to the each characteristic point.
  - (2) Multiply each counter with its *weight*.
  - (3) Calculate the Total for each characteristic point.
  - (4) Calculate de general Total by summing the partial totals (Table 4.3.2.3.a.).
    - There are versions of method in which weights differs function of program complexity (*simple, average, complex*).
    - The determined Total can be *adjusted* in a similar manner as in functional point method.
  - (5) Establish the weights of the 14 influence factors.
  - (6) Calculate the value of the ***complexity adjustment*** factor (**SIF-**Sum of Influence Factors) as sum of the 14 influence factors.
  - (7) Calculate the value of the Adjusted Total :

    Adjusted Total =Total*(0.65 + 0.01*SIF)

Characteristic Type	Basic Counts	Weights	Total
External Inputs		4	
External Outputs		5	
External Inquiries		4	
Internal Files		7	
External Interfaces		7	
Algorithms		3	
Total			

Table 4.3.2.3.a. Characteristic-point and associated weights

4.3.2.4 Proxy-based Estimation

- In project planning, *estimates* are generally required before development can begin.
  - At this early stage, the requirements may be understood but little is generally known about the product itself.
- The estimating problem is thus to *predict* the likely finished size of the required product.
  - Because no one can know in advance how big a planned product will be, software size estimating will always be an uncertain process.
- The need, however, is to make an **estimate** as accurate as possible.
- **In general**, all estimating methods use data on previously developed similar programs to establish some basis for judging the size of the new program.
- This suggests a generalized estimating process (Fig.4.3.2.4.a) [Humphrey95].

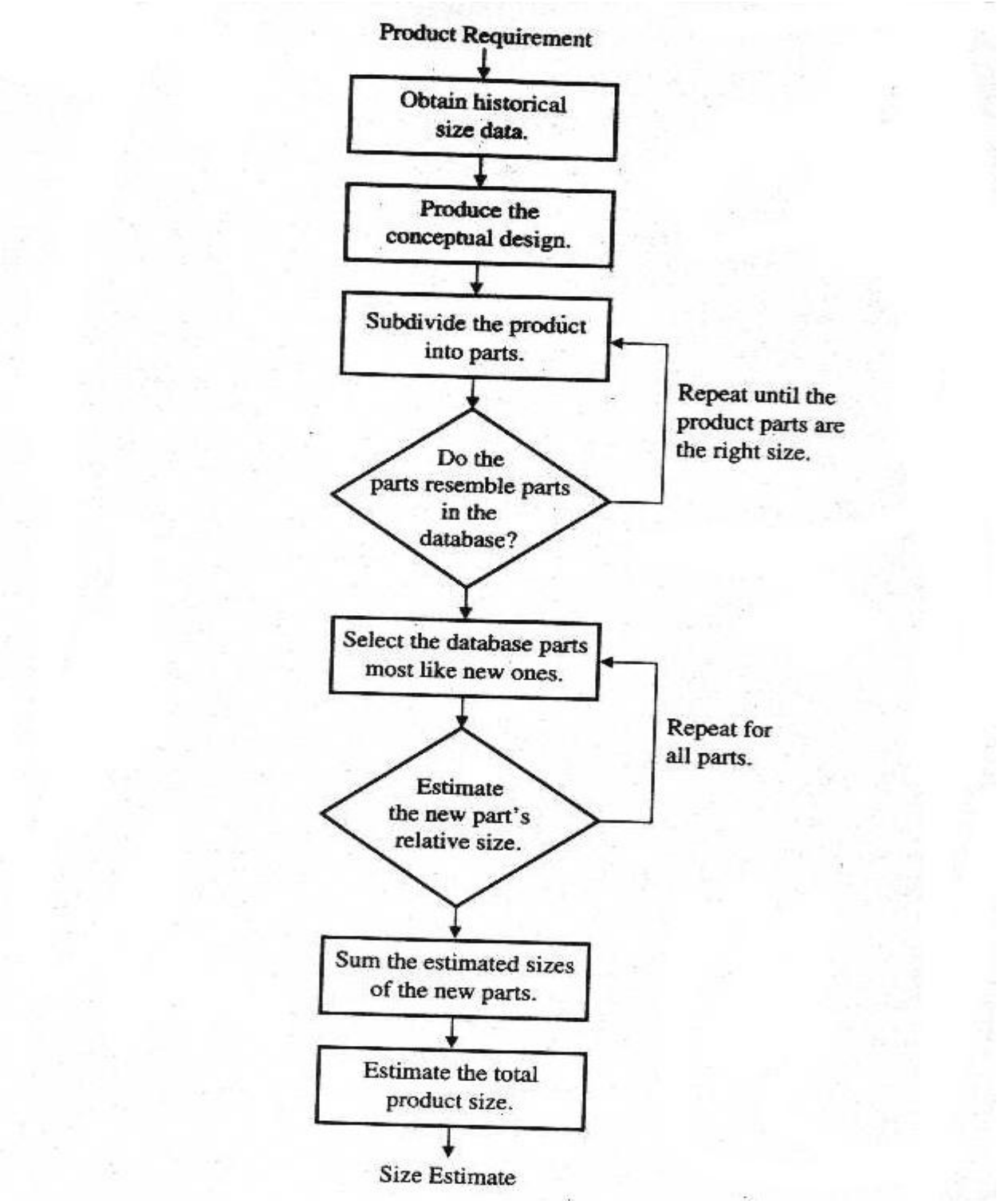

Fig.4.3.2.4.a. Size estimating overview

- **Proxies**
  - Consider again the example of construction.

    - In home building, the number of square feet of living space provides a basis for estimating construction costs.
    - Few people, however, can visualize the house they want in terms of square feet.
- We have the same problem with software.
    - If we could accurately judge the **numbers of LOC** in a planned software product, we could make good development estimates.
    - Unfortunately, few people can directly judge how many LOC it will take to meet a software requirement.
- The need is for some *proxy* that relates product size to the functions the estimator can visualize and describe.
    - A ***proxy*** is a ***substitute*** or a ***stand-in***.
- Assuming it, is easier for you to visualize the proxy than the size measure you are using, a proxy can help you judge product size.
- Therefore proxies can be used to estimate a product's LOC.
    - Examples of *proxies* are:
        - Objects
        - Screens
        - Files
        - Scripts
        - Function points

- **Selecting a Proxy**
    - The **criteria** for a good proxy are as follows:
    - (1) The proxy size measurement should closely **relate to the effort** required to develop the product.
        - *Related to Development Effort**.***
            - A proxy, to be useful, must have a demonstrably *close relationship to the resources* required to develop the product.
            - By estimating the size of the proxy, you can then accurately judge the size of the job.
            - You determine the effectiveness of a proxy by obtaining historical data on a number of products you have developed and comparing the proxy values with the development costs.
            - Using the correlation method, you can then determine if this or some other proxy is a better predictor of product size or development effort.

        - If the proxy does not pass this test, there is no point in using it.

- (2) The proxy content of a product should be **automatically countable**.
    - *Automatically Countable Proxy Content**.***
        - Because historical proxy data are needed for making new estimates, it is desirable to have a large amount of proxy data.
        - This requires that the data be automatically countable, which in turn suggests the proxy must be a *physical entity* that can be precisely defined and algorithmically identified.
        - If you cannot automatically count the proxy content of a program, there is **no convenient way** to reliably obtain the statistical data you need to generate estimating factors customized for your particular development process and design style.

- (3) The proxy should be **easy to visualize** at the beginning of a project.
    - *Easily Visualized at the Project's beginning**.***
        - If the **proxy** is harder to visualize than the *number of programmer hours* required to do the development, you may as well **estimate hours** directly and not worry about the proxy.
        - The usefulness of a **proxy** thus depends on the degree to which it helps you to visualize the size of the project to be done.
        - This in turn depends on your background and preferences. So there will likely not be one best proxy for all purposes.
        - With suitable historical data, you could even end up using several different proxies in one estimate.
        - The multiple regressions can be helpful for this purpose.

- (4) The proxy should be **customizable** to the special needs of using organizations
    - *Customizable to the Needs of the Using Organization**.***
        - Much of the difficulty organizations have with various estimating methods comes from their attempts to use data from one development group for planning developments by another group.
        - While this is principally a problem with resource models, it also applies to the use of proxies. It is important, therefore, to gather and use data that are relevant to the particular project being estimated.
        - This suggests that large organizations have size and resource databases for **each major software product type**.

        - Similarly, every **individual programmer** should gather and use data on his own work.
        - If the selected proxies are not suitable for some of the work you do, then your data should help you to identify other more suitable proxies and their estimating factors.

- (5) The proxy should be **sensitive to any implementation variations** that impact development costs or effort.
    - *Sensitive to Implementation Variations**.***
        - This issue involves a difficult trade-off.
        - The proxies that are most easily visualized at the beginning of a project are application entities, such as inputs, outputs, files, screens, and reports.
        - Unfortunately, a good development estimate requires entities that closely relate to the product to be built and the labor to build it.
        - This need also requires that data be available on proxy and product size for each implementation language, design style, and application category.
        - Thus it is essential that the languages, design styles, and application categories to be used on a project be represented in the data used to calculate the estimating factors to be used.

- **Possible Proxies**
    - Many potential types of proxies could meet the previously outlined criteria.
    - The ***function-point method*** is an obvious candidate because it is widely used.
        - Many people have found *function points* to be helpful for resource estimating, so you should consider them.
        - However, as noted earlier in this chapter, the function-points' principal disadvantage is that they *are not directly countable* in the finished product.
    - Other possible proxies are:
        - *Objects*
        - *Screens*
        - *Files*
        - *Scripts*
        - *Document chapters*
        - *etc.*

- **Objects as Proxies**
  - The principles of object-oriented design suggest that **objects** would be good estimating *proxies*.
  - During initial **program analysis** and **design**, *application entities* are used as the basis for selecting system objects. [Booch, Coad, Shlaer, Wirfs-Brock].
    - An *application entity* is something that exists in the application environment.
  - In an **object-based design**, you select program objects that will **model** these real-world entities.
    - This means that your highest-level product objects could be visualized during requirements analysis.
    - Objects thus potentially meet one of the basic requirements for a **proxy**.

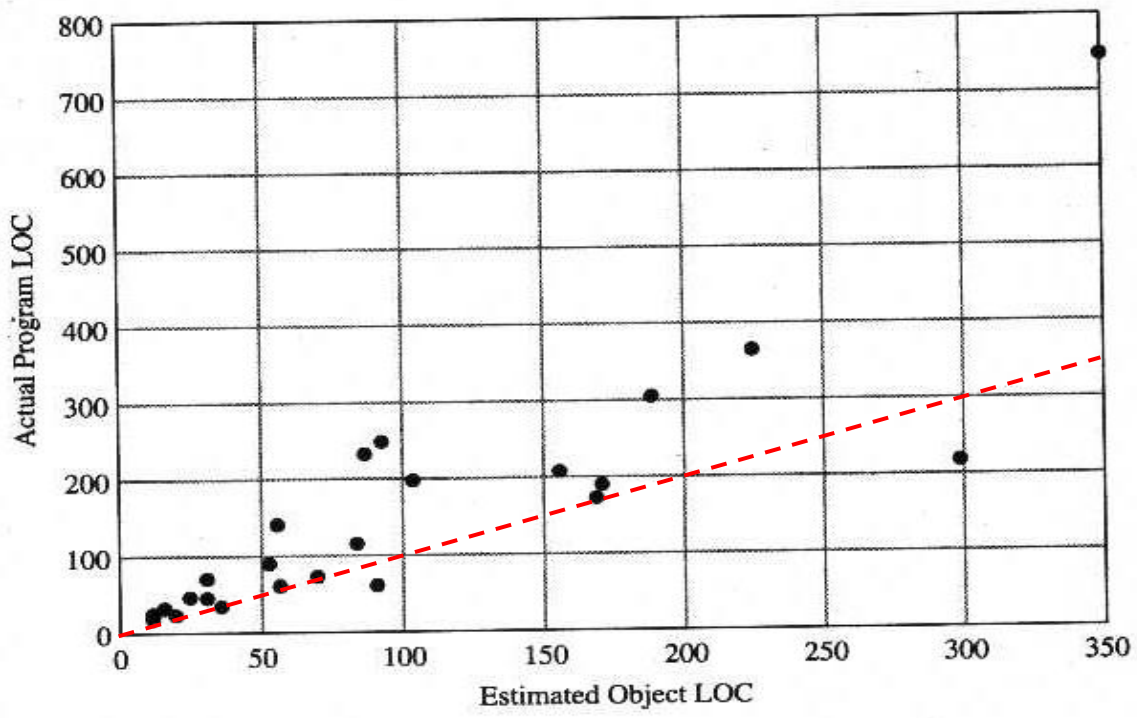

Fig.4.3.2.4.b. Object LOC versus program LOC

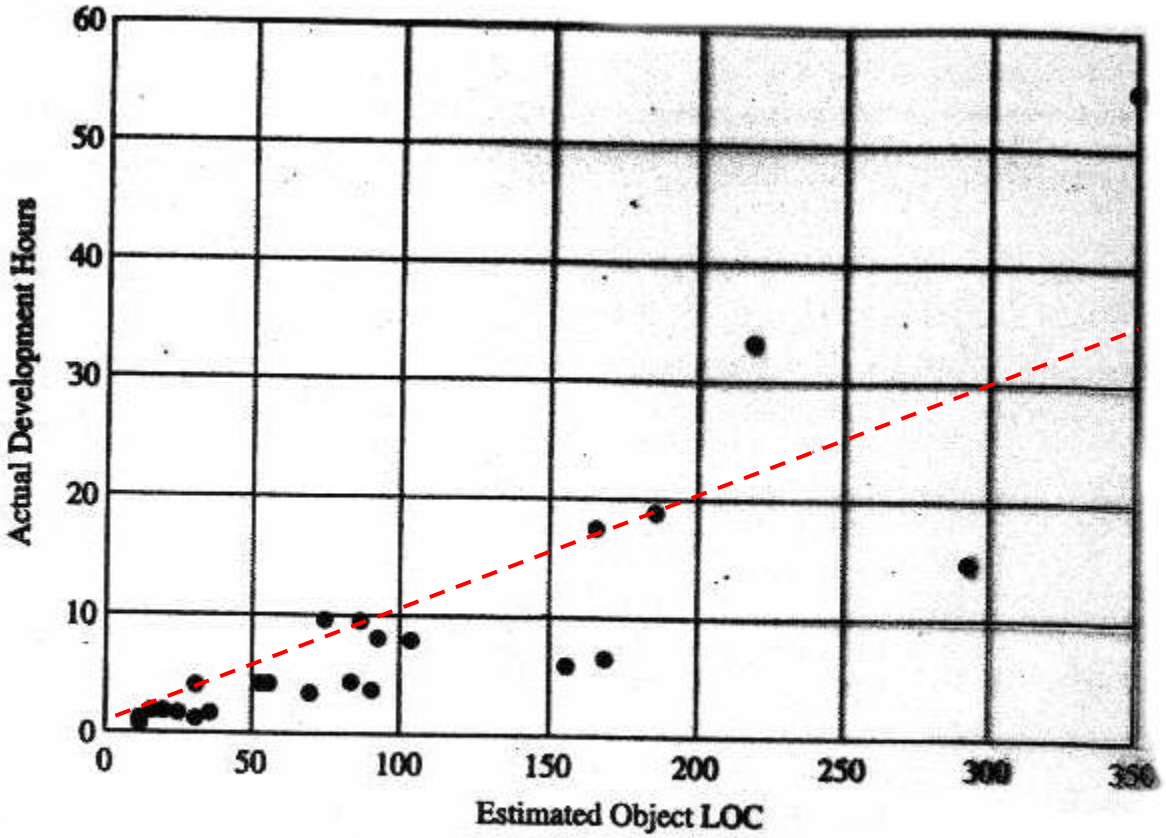

Fig.4.3.2.4.b. Object LOC versus programming hours

- In both cases, the **correlation** and **significance** are very high.

- **Conclusions:**
  - (1) Be*cause total program LOC* correlates *to development hours* and *objects* correlate highly with *total program LOC*, objects thus meet the requirement that they *closely relate to development effort*.
  - (2) Objects are physical entities that can be *automatically counted*.
  - (3) Assuming you use an *object-oriented design methodology* and *implementation language* and you *gather size data on your objects*, objects will meet *all the criteria for a proxy*.

- To use **objects** as **proxies**:
- (1) Divide your *historical object data* into categories and size ranges.
    - The construction example illustrates why you do this.
    - When estimating the square feet in a house, a builder needs to think about big rooms and small rooms and about how many of each the buyer wants.
    - It is even important the builder think about room categories; for example, a big bathroom would be much smaller than even a small family room.
  - (1.1) For estimating how many LOC objects your new product contains, you should first group these objects into **functional object categories**.
    - Example of **functional object categories**:
      - **PASCAL**: Control, Display, File, Logic, Print, Text.
      - **C++**: Calculation, Data, I/O, Logic, Set-up, Text.
  - (1.2) Using Putnam's fuzzy-logic method, divide these **functional object categories** into **ranges** of *very small, small, medium, large, very large* objects. [Putnam].
    - In the top part of Table 4.3.2.4.c [Hu95] the C++ objects are listed in six categories.
    - These data show that a *medium* or *average-sized* C++ object for mathematical calculations would have about 11 LOC, while a very large calculation object would have 54 LOC.
- (2) You then make estimates by judging:
  - (2.1) *How many objects* of *each category* you need for the new product.
  - (2.2) The *relative size* of *each object* in its category.
  - Object size is also a *matter of style*.
    - Some people prefer to group many functions in a few objects; others prefer to create many objects of relatively **few functions**.
    - The data used for estimating purposes should accurately reflect the languages and practices you use in developing software.
  - Therefore, it is a good idea to use **object size data** on your own personal work.
  - Because of the wide variations in the *numbers of methods* or *procedures* per object, is helpful to **normalize objects** by their **number of methods** (or procedures) and to judge object size on a *per-method basis*.
    - The data in Table 4.3.2.4.c are thus given in **LOC per method**.
      - In Pascal terms, this would be **LOC per procedure;**

- For **example**, if you have a medium-sized Pascal text object that has four methods, it would average 16.48*4, or about 66 LOC.

TABLE 5.7 OBJECT CATEGORY SIZES IN LOC PER METHOD

C++ Object Size in LOC per Method					
Category	*Very Small*	*Small*	*Medium*	*Large*	*Very Large*
Calculation	2.34	5.13	11.25	24.66	54.04
Data	2.60	4.79	8.84	16.31	30.09
I/O	9.01	12.06	16.15	21.62	28.93
Logic	7.55	10.98	15.98	23.25	33.83
Set-up	3.88	5.04	6.56	8.53	11.09
Text	3.75	8.00	17.07	36.41	77.66

Object Pascal Object Size in LOC per Method					
Category	*Very Small*	*Small*	*Medium*	*Large*	*Very Large*
Control	4.24	8.68	17.79	36.46	74.71
Display	4.72	6.35	8.55	11.50	15.46
File	3.30	6.23	11.74	22.14	41.74
Logic	6.41	12.42	24.06	46.60	90.27
Print	3.38	5.86	10.15	17.59	30.49
Text	4.63	8.73	16.48	31.09	58.62

Fig.4.3.2.4.c. Object category sizes in LOC per method (C++, PASCAL)

- (3) In **estimating** LOC per object:
  - (3.1) First, decide which functional category object you are considering.

- (3.2) Then, judge how many methods it likely will contain.
- (3.3) Determine whether it falls into the very small, small, medium, large or very large size range.
- (3.4) Calculate the size in LOC by summing.

- Humprey has developed **PROBE** Size Estimating Method using objects as proxies.
  - **PROBE** means ***PROxy-Based Estimating*** [Hu95].
  - The principle of this method is presented in figure 4.3.2.4.d.

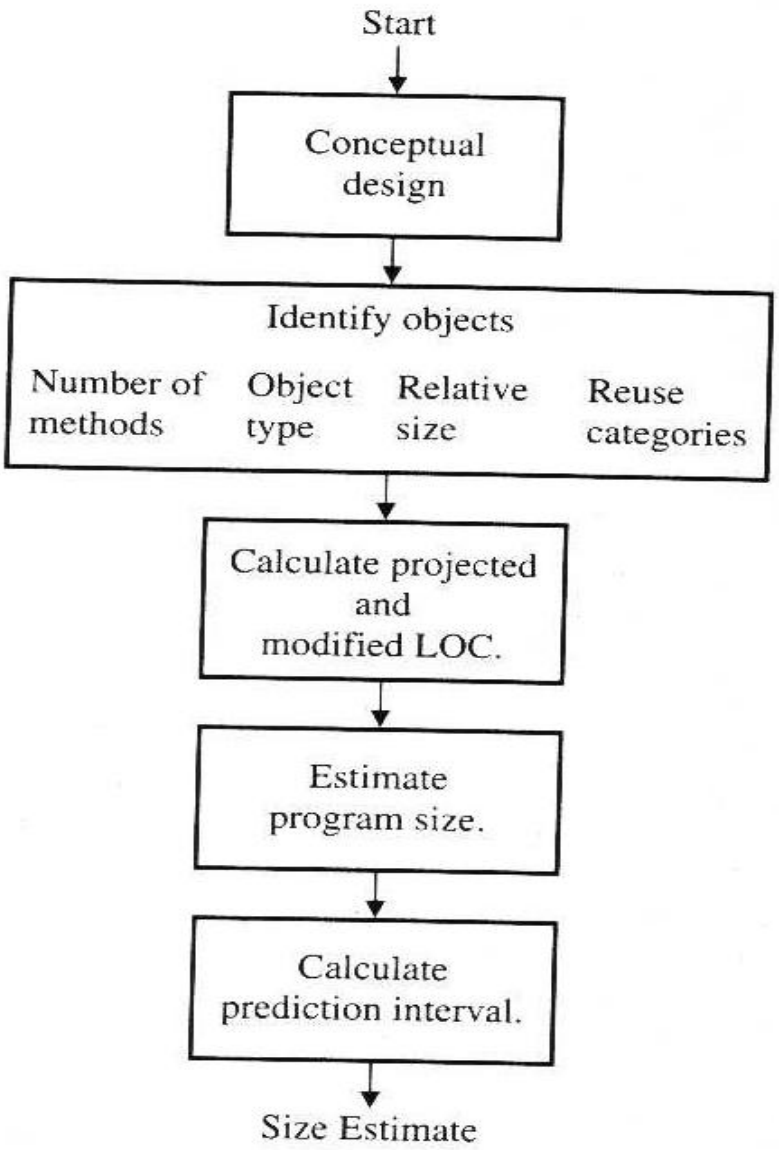

Fig.4.3.2.4.d. PROBE Size Estimating Method

Exercise #5

1. Which are the *objectives* of the Definition Phase?

2. What are the *targets* of the Problem Analysis? What are the *recommendations for the Analyze activity*?

3. Describe the contents of the Problem Specification document.

4. What are the analysts' main tasks?

5. What is a system? Which are the main characteristics of a system?

6. Describe the most well known planning tools emphasizing their advantages, disadvantages and application area.

7. What is the relationship between size and resources for a SW project? What are the size estimation criteria that should be meet by an estimation method?

8. What size oriented metrics do you know?

9. Describe Wideband-Delphi Method. Highlight the advantages and disadvantages.

10. Describe Fuzzy-Logic Method. Highlight the advantages and disadvantages.

11. Describe Standard Component Method. Highlight the advantages and disadvantages.

12. Describe Function-Point Method. Highlight the advantages and disadvantages. What do you know about function-point/sloc conversion?

13. Describe Characteristic-Point Method. Highlight the advantages and disadvantages.

14. What is a proxy? What are its main characteristics? Can you describe a proxy based size estimation method?