Chapter 6.

THE PROGRAMMING PHASE

Part I

Summary

0 Introduction

1 Programming Techniques

- 1.1 Structured programming
  - 1.1.1 Goals of structured programming
- 1.2 Object-Oriented Programming, Design and Analysis
  - 1.2.1 Object-Oriented Programming
  - 1.2.2 Object-Oriented Design
  - 1.2 3 Object-Oriented Analysis

2 Organization Modalities

- 2.1 Conventional Organization
  - 2.1.1 Analysis and Design Group
    - 2.1.1.1 Change Control
    - 2.1.1.2 Data Control
    - 2.1.1.3 Structured Walk-Throughs and Inspections
    - 2.1.1.4 Simulation Modeling
    - 2.1.1.5 User Documentation
  - 2.1.2 Programming Group
    - 2.1.2.1 Detailed Design
    - 2.1.2.2 Coding
    - 2.1.2.3 Module Test
    - 2.1.2.4 Documentation
    - 2.1.2.5 Integration: "Top-Down"
    - 2.1.2.6 Integration: "Bottom-Up"
    - 2.1.2.7 Integration: The Test specification
  - 2.1.3 Test Group
  - 2.1.4 Staff Group
    - 2.1.4.1 Technical Staff Functions
    - 2.1.4.2 Administrative Staff Functions
  - 2.1.5 The Numbers Game
- 2.2 Team Organization. Chief Programmer Team

2.2.1 How It Works
2.2.2 Project Organization using Chief Programmer Team approach
3 Change Control
3.1 Baseline Documents
3.2 Control Procedures
4 Programming Tools
4.1 Written Specifications
4.2 Test Executives
4.3 Environment Simulators
4.4 Specialized Programming Environment
4.5 Automated Documentation Aids
4.6 Software and Hardware Monitors
4.7 The Project Library
4.7.1 General Library
4.7.2 Development Support Library

6 THE PROGRAMMING PHASE

0 Introduction

- At last you're ready to write programs, and things begin to happen.
  - Suddenly there are more people to manage;
  - The paper pile has swollen;
  - Programmers are waiting to start the coding;
  - Flaws show up in the baseline design;
  - The customer leans on your programmers to bootleg changes;
  - Your manager says you're overrunning the budget;
  - Jack programmer is a dud;
  - Jill programmer gets married and leaves;
  - And your spouse is bugging you about being married to that stupid computer.
- You'll be thankful you planned and designed well because your hands will be full tending to daily problems that no amount of planning can avert.
- This chapter focuses on the programming job and the most effective way to get it done.

1 Programming Techniques

- In ours days, two **programming techniques** are more used:
  - **(1) Structured Programming.**
  - **(2) Object Oriented Programming.**

1.1 Structured Programming

- **Structured programming** is an effort to establish order in the construction of a program.
  - There is a good deal of hesitancy on the part of most computer scientists and programmers in defining **structured programming**.
    - According to **Yourdon**, *"the notion of structured programming is a philosophy of writing programs according to a set of rigid rules in order to decrease testing problems, increase productivity, and increase the readability of the resulting program"*
    - **Hughes** and **Michton** offer this: *" .... . we could say that structured programming is the design, writing, and testing of a program in a prescribed pattern of organization"*
    - **Harlan Mills** says *". . . the essence of structured programming is the presence of rigor and structure in programming. . ."*
- The ideas that are common to all the definitions of **structured programming** are:

    - (1) Order
    - (2) Clarity
    - (3) Readability
    - (4) All leading toward the goal of error-free code which may be readily understood by people other than the program's author.
- The days of intricate secret code written by snobs or messy code written by poorly trained programmers are, we may hope, coming to an end.
    - There is such a strong drive among the leaders of the programming community to bring order to the business that sooner or later the entire complexion of the programming activity shall certainly change for the better.
- The transition from the old ways to the new is, of course, most difficult for those accustomed to the old;
    - Setting new programmers on a clearer road, before they have learned bad habits, is relatively easy.
- In our days **structured programming** is considered a **classical approach**.

1.1.1 Goals of Structured Programming

- (1) ***Correctness.***
    - Nobody wants to structure programs simply to make them pretty.
        - What counts in the end is that the programs be **correct**—that they do their prescribed functions flawlessly.
    - Using structured programming and related concepts, complex programs are now being written which run correctly the first time.
        - By practicing principles of structured programming and its mathematics, you should be able to write correct programs and convince yourself and others that they are correct by **logic and reason** rather than by **trial and error**.
    - If you are a professional programmer, errors in program logic should be extremely rare, because you can prevent them from entering your programs by positive action on your part.
        - Durring development, programs do not acquire **bugs** as people do germs—just by being around other buggy programs.
        - They acquire bugs only from their authors.
- (2) ***Readability.***
    - There is no place in today's computer business for programs which cannot be **read** and **understood** by other than the original authors.
        - Programs must be made readable *from the start* so that they can be **inspected** by managers, supervisors, and other programmers who are double-checking logic or tracking down problems in the system.
    - Programs must be readable *at the finish* so that they can be modified and maintained by other than the original programmers.

- (3) ***Testability.***
    - It follows that a readable, clearly structured program may be more easily tested (especially by someone other than the original author) than a mysterious program.
- (4) ***Increased productivity.***
    - Improvements in the first three goals (correctness, readability, testability) automatically lead to lower programming costs.

1.2 Object Oriented Programming, Design and Analysis

1.2.1 Object-Oriented Programming

- What is **object-oriented programming** (or **OOP***,* as it is sometimes written)?
- A possible definition:
    - ***Object-oriented programming*** *is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.*
- There are three important parts to this definition: **object-oriented programming**:
    - (1) Uses objects, not algorithms, as its fundamental logical building blocks (the "part of" the program hierarchy).
    - (2) Each object is an instance of some class.
    - (3) Classes are related to one another via inheritance relationships (the "is a" of the hierarchy).
- A program may appear to be object-oriented, but if any of these elements is missing, it is not an object-oriented program.
- Specifically, programming without inheritance is distinctly not object-oriented.
    - We call it programming with **abstract data types**.
- By this definition, some languages are object-oriented, and some are not.
    - From a theoretical perspective, one can fake object-oriented programming in non-object-oriented programming languages.
- **Cardelli** and **Wegner** thus say *"that a language is object-oriented if and only if it satisfies the following requirements:*
    - (1) *It supports objects that are data abstractions with an interface of named operations and a hidden local state.*
    - (2) *Objects have an associated type [class].*
    - (3) *Types [classes] may inherit attributes from supertypes [superclasses]"*
- For a language to support inheritance means that it is possible to express "is a" relationships among types,
    - If a language does not provide direct support for inheritance, then it is not object-oriented.

- **Cardelli** and **Wegner** distinguish such languages by calling them *object-based* rather than *object-oriented.* Under this definition:
    - **Object-oriented languages**: Smalltalk, Object Pascal, C++, Eiffel, CLOS, Java, C#.
    - **Object-based language**: Pascal, Ada.
- However, since objects and classes are elements of both kinds of languages, it is both possible and highly desirable for us to use object-oriented design methods for both object-based and object-oriented programming languages.

1.2.2 Object-Oriented Design

- The emphasis in **programming methods** is primarily on the proper and effective use of particular language mechanisms.
- By contrast, **design methods** emphasize **the proper and effective** structuring of a complex system.
- What then is **object-oriented design**?
- A possible **definition**:
    - ***Object-oriented design*** *is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.*
- There are two important parts to this definition. **Object-oriented design**:
    - (1) Leads an object-oriented decomposition.
    - (2) Uses different notations to express different **models**:
        - Logical (class and object structure).
        - Physical design of a system (module and process architecture).
        - Static aspects of the system.
        - Dynamic aspects of the system.
- The support for **object-oriented decomposition** is what makes object-oriented design quite different from structured design.
    - The object -oriented design uses class and object abstractions to logically structure systems.
    - The structured design uses algorithmic abstractions.
- The term *object-oriented design is used to refer to any method that leads to an object-oriented decomposition.*
    - Occasionally the acronym **OOD** (**Object Oriented Design**) is used to designate a particular method of object-oriented design.

1.2.3 Object-Oriented Analysis

- The **object model** has influenced even earlier phases of the software development life cycle.

- Traditional structured analysis techniques, best typified by the work of **DeMarco, Yourdon**, and **Gane** and **Sarson**, with real-time extensions by **Ward** and **Mellor**, and by **Hatley** and **Pirbhai**, focus upon the flow of data within a system.

- **Object-oriented analysis** (or **OOA** as it is sometimes called) emphasizes the building of real-world models, using an **object-oriented** view of the world:
  - ***Object-oriented analysis*** *is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.*
- How are **OOA, OOD**, and **OOP** related?
  - Basically, the products of object-oriented analysis serve as the models from which we may start an object-oriented design.
  - The products of object-oriented design can then be used as blueprints for the completely implementing a system using object-oriented programming methods.

2 Organization Modalities

- There are a number of basic ways of **organizing people** to do a job:
  - (1) Functional organization.
  - (2) Job-shop organization.
  - (3) Project organization.

- (1) **Functional organization**
  - The Project Manager borrows people from groups of specialists within the company.
  - Each specialist is on loan to PM to do his part of the job, and then he's gone — on loan to the next manager who needs his skills.
    - This arrangement gives the project manager, little control because the man on loan is likely to be more concerned with his home organization than with your project.
    - Typically, PM has little or no say about whom he get, and he can be frustrated by substitutions made before his job is finished.
  - Perhaps worse than that there will be little or no **continuity** of people on job.
  - In the worst case:
    - The analysts come, they analyze, they leave.
    - The designers come, they design, they leave.
    - And the same for the programmers.
- (2) **Job-shop organization**
  - The program system is broken up into several major subsystems.

    - A **manager** and his **group** is assigned with total responsibility for developing that subsystem—analysis, design, programming, the works.
        - Here the big problem is that nobody has his eye on the *system* because the managers are concerned only with the subsystems.
    - A job-shop arrangement works if there are to be done a number of relatively **small**, **unrelated jobs** (in other words, not a system).
        - If you're a manager accustomed to a job-shop organization and are about to manage the development of a system, remember that what worked before may not work now.
- (3) **Project organization**
    - Neither functional nor job-shop organization is appropriate for producing a system.
        - The kind of organization needed here is **project organization**.
    - What is implied in any such arrangement is:
        - (1) The people involved devote their efforts to a single project.
        - (2) They are all under the control of a single project manager.
    - **Project organization** may take many forms.
        - Every company has its rules about lines of authority, degree of autonomy, reporting to outside management, and so on.
        - Ignoring such considerations, we may discuss **project organization** in terms of two quite different approaches:
            - (1) **Conventional organization**.
            - (2) **Team organization**.

2.1 Conventional Organization

- Figure 6.2.1.a illustrates two conventional ways to organize a medium size project.
- The only real difference between **(a)** and **(b)** is the number of management levels between the project manager, and the people who do the technical work.
- The choice of **(a)** or **(b)** depends on project manager's strengths and weaknesses and those of the managers who are available to him.
    - (1) If the project manager is **technically strong**, able to absorb much detail, and can handle as many as seven managers reporting to him (a hefty number), then **(a)** might well be the choice.
        - The danger here is that the project manager may become swamped in details, lose sight of broader project objectives, and lose control.
    - (2) If the project manager prefers to **delegate more responsibility** so that he can concentrate on the important problems that arise, then **(b)** might be the choice.
        - In that case the project manager has four managers reporting to him.

- Either way the project has **many** managers (seven or eight besides project manager), and that may horrify the boss - "*Where are the workers!?*".
    - (1) In fact, these managers are not paperwork shufflers.
    - (2) Since they are very much involved in technical decisions, the ratio of managers to workers is not so bad as it looks.
- In most situations is recommended choose **(b)** over **(a).**

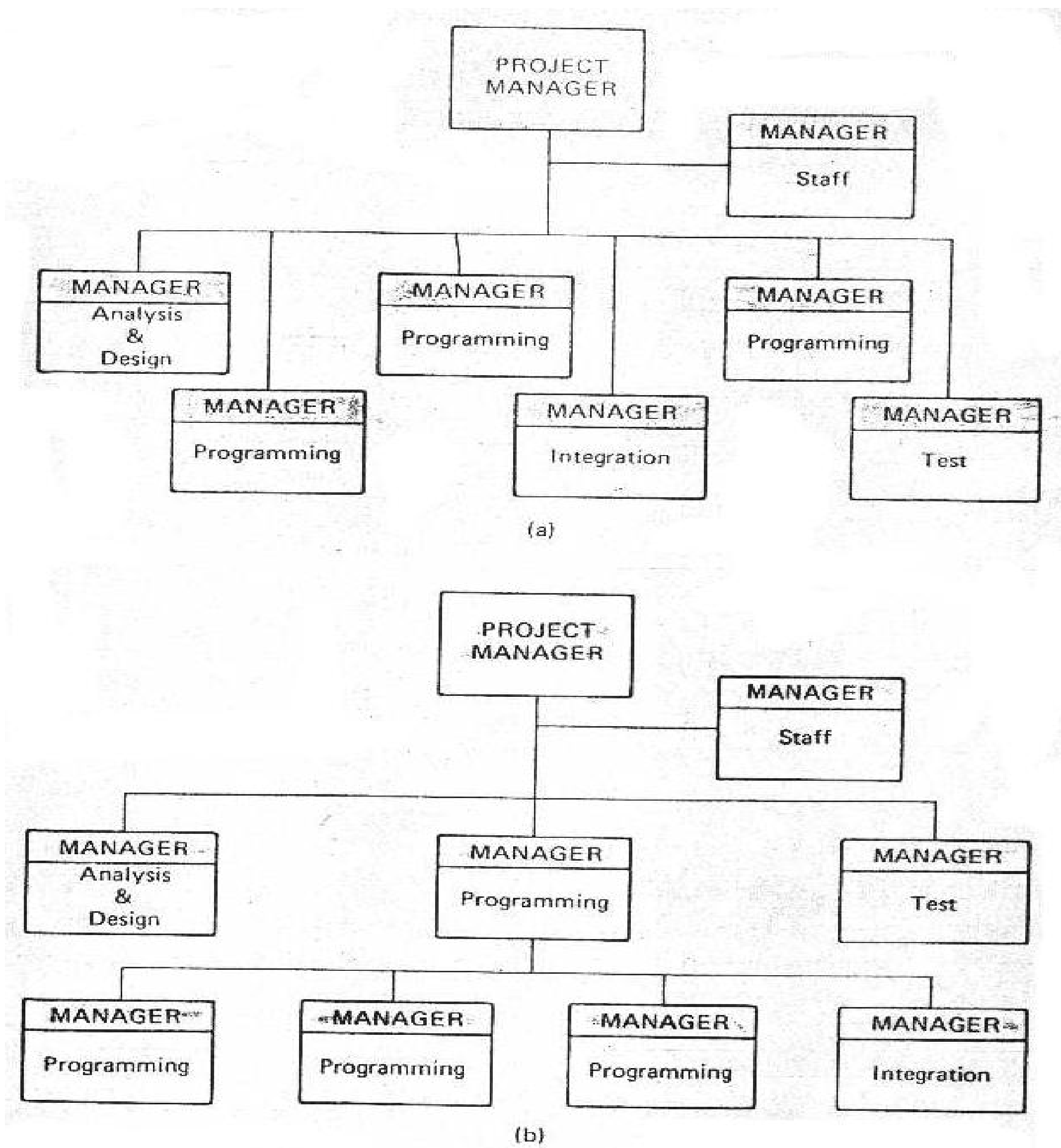

Fig. 6.2.1.a. Conventional Project Organization

- The real importance, however, is not in the exact number of boxes on the organization chart nor their titles but:
    - (1) The PM (project manager) must account for all jobs that have to be done and do it in a workable way.
    - (2) PM must be sure that every member of the organization knows both his and other people's objectives.

- The remainder of this section describes the functions of the various groups shown in Figure 6.2.1.a. (b) and ends by considering some typical numbers of people in the various roles.

2.1.1 Analysis and Design Group (A&D Group)

- We are now in the **Programming Phase** and, therefore, the programmers are at center stage.
- However, the **analysts and designers** still play a very strong **supporting role**.
- A subset of the original analysts and designers have the following jobs to do:
  - (1) Change Control.
  - (2) Data Control.
  - (3) Structured Walk-Throughs and Inspections.
  - (4) Simulation Modeling.
  - (5) User Documentation.

2.1.1.1 Change Control

- The most important function of Analysis and Design Group is to carry out the **change control procedures** which will be described later.
- This means:
  - (1) Investigating proposed changes.
  - (2) Recommending adoption or rejection.
  - (3) Documenting the results.
- The group acts as a filter.
  - It relieves other project members, particularly the programmers, from much of the burden of digging into a proposed change and tracking down the consequences of making the change.
    - On many projects the investigation of a change proposal falls on the programmer.
    - The programmer is constantly sidetracked from his main job to run down this or that idea suggested by the customer or by someone in his own organization.
  - When a person is doing something as logic-oriented as programming, every interruption means a loss of efficiency.
    - When the interruption ends, he must say, "*Now, where was I*?"
    - In addition to the wasted time backtracking, he may well end up with a bug at the point of interruption.
    - Very often the frustration of constant interruption causes the programmer to give a hasty answer and to agree to the change just to get the problem off his back so that he can get on with his programming.

- Having a group handle change proposals concentrates a vital function in one place rather than spreading it thinly over the project.

2.1.1.2 Data Control

- The Analysis and Design Group is also involved in **Data Control**.
  - This is really part of the change control function, but it needs emphasis.
- **Data control** means keeping an eye on all system files so that their structures are not violated.
  - By **system files** we mean those organizations of data that are accessed (either stored into or retrieved from) by more than a single program module.
- The following should have been spelled out as part of the baseline design:
  - (1) The system file structures.
  - (2) A dictionary defining each data item.
  - (3) All the rules for using system files.
- In a great many program systems these data files hold the system together.
- Just as it is necessary to control changes to program logic after the baseline has been established, so too must changes to the system files be controlled.
  - Don't leave it to the programmers to form ad hoc agreements as they go along.

2.1.1.3 Structured Walk-Throughs and Inspections

- The Analysis and Design Group is a good place to assign responsibility for **scheduling** and **conducting** continuing detailed reviews of technical progress.
- There are two closely related means of conducting such reviews:
  - **(1) Structured walk-throughs**.
  - **(2) Inspections**.
- Both these terms came into use during the 1970s; some people make a distinction between the two, some use the terms interchangeably.
  - *A **structured walk-through** is simply an organized (structured) review of a project member's work by other project members.*
- During a **walk-through**:
  - (1) The developer (the person whose work is being reviewed) first gives a tutorial description of his project.
    - This may be a design, code, a set of documentation, a test plan, an artifact or any other item.
  - (2) Then the project members "**walks through**" the product verbally, step by step, giving the reviewers a "**guided tour**" and inviting them to find flaws.

- (3) The vehicle the developer uses may be whatever is appropriate to his **product**.
  - If a design is involved, then **HIPO charts** and other design documents would be pertinent.
  - If a module of code is involved, then the **actual code** would be used — or **pseudo code** or similar if the actual code has not yet been written.
  - If reviewing a test plan or user's manual, then either a detailed outline or a **draft version** of those documents would be walked through.
- (4) The idea is that there be a definite, very specific look in detail at each product.
  - Rather than relying on the testing process to show up problems or simply passing documents around hoping that people will review and comment on them.
- (5) There are generally 4-6 participants in a structured walk-through.
  - One of them is always a **moderator**.
- (6) Is recommended that the Analysis and Design Group to be **responsible** for scheduling and conducting the walk-throughs.
  - In consequence a member of that group should act as **moderator**.
- (7) The **moderator** has as main tasks:
  - (a) Schedules the meetings and meeting places.
  - (b) Helps select participants.
  - (c) Reports results immediately after the meeting.
  - (d) Follows up to see that any rework to be done is done and presented again if necessary.
  - (e) But most important, he must keep the walk-through sessions moving along toward their objectives without getting sidetracked and without allowing animosities or bruised egos to destroy the effectiveness of the review.
- (8) The other participants in the walk-through include:
  - The developer whose work is being reviewed.
  - Two to four others who are competent enough to understand his work and its place in the system.
  - If the work being reviewed is a module of code, one of the participants might be a programmer responsible for similar code or code which interfaces this module directly; another might be a programmer responsible for code elsewhere in the system, say, in the control program.
  - If the module was designed by someone other than the coder, the original designer should be present.
  - The makeup of the review groups can be quite flexible.
- (9) The moderator must select reviewers thoughtfully.

- In most cases, managers are **not** included in walk-throughs.
- These sessions are **not** intended as vehicles for appraising employees; a manager's presence would inevitably put a huge damper on the proceedings.

- (10) The aim of the walk-through is to find errors, **not** to correct them.
  - Corrections must be assumed to be within the province and capabilities of the developer.
- (11) A review session might last for fifteen minutes to two hours.
  - If more than two hours is needed, a second session can be scheduled after an appropriate break (probably later in the same day, so that continuity is not lost).
- (12) There are some extremely important **benefits** as a result of conducting serious and frequent walk-throughs of all the project's products:
  - (a) Where the product is actual design or code, there is a demonstrable and significant saving when errors are found early.
    - The later in a project's life an error is found, the greater the cost of fixing the error.
    - A good deal of expensive, time-consuming regression testing might have to be performed to ensure that making a change to fix an error embedded deep in the system will not adversely affect other code already tested and presumed clean.
  - (b) There can be an enormous benefit in promoting what's called "egoless programming" or egoless anything, for that matter.
    - In an excellent book, "*Psychology of Computer Programming*" **Gerald Weinberg** makes a strong case for taking steps toward making the programmer less defensive about errors in his work by promoting the idea of programmers reading each other's code in order to find problems.
    - He cites evidence that a great number of bugs are discovered early when code reading is practiced.
    - The specific techniques for this purpose are known as ***peer programming*** or ***peer review***.
  - (c) The advantages go beyond that, however.
    - Extensive and regular reading of code provides a beautiful opportunity for helping **to train** newer people.
    - The process fosters a feeling **of openness** on the project, in direct contrast to the situation in which a programmer treats a module of code as his own private property.
    - Is a guaranty for the **quality** of the **process**
  - (d) When the product is a document, say a test plan or a user's manual, savings are effected not only by avoiding errors in those documents which might affect the testing or the use of the system, but by cutting down on republication and distribution costs, as well.

        - (e) Frequent and productive walk-throughs, once they become an **accepted way of project life**, lead to better products in the first place, because developers will not knowingly submit sloppy work for such scrutiny.
            - It's very common for a programmer or a writer to throw together a "quick-and-dirty" first hack of a program or a document, intending to "clean it up" later. But often, later never comes.
            - Walk-throughs can go a long way toward eliminating such sloppy and dangerous habits.
        - (f) There is an enormous educational benefit as a result of walk-throughs.
            - It becomes impossible for individuals to work for long periods in isolation from other project members, with their work hidden from scrutiny.
            - Everybody knows what everybody else is doing.
- The term **"inspection"** is preferred by some over "structured walk-through" to denote a similar but much more rigorous activity.
    - **Inspections** are more intensive examinations of detailed design and code, with much more emphasis on keeping statistics on types of errors found to help guide subsequent inspections.
    - The rigor and careful record-keeping of inspections become important as projects grow large and loss of control becomes a problem.

2.1.1.4 Simulation Modeling

- The Analysis and Design Group is responsible for continuing simulation modeling activities begun during earlier phases.
    - It conducts simulation runs and evaluates and distributes results.
    - It may propose design changes as a result of some simulations.
- On a very large project in which much simulation is done it may be necessary to form a separate simulation modeling group.

2.1.1.5 User Documentation

- The are two major category of **Documentation**:
    - (1) **User Documentation** includes anything you are responsible for writing that will help the customer to use the system.
        - This is responsibility of A&D Group.
    - (2) **Descriptive Documentation** something telling how the system is put together.
        - That's the programmers' job.
- **User Documentation** may include the following topics:
    - (1) Installing the system.

- (2) Periodic testing of the system after installation.
- (3) Daily start-up procedures.
- (4) Daily operating procedures, options, and error correction.
- (5) Preparing inputs for the system.
- (6) Analyzing outputs from the system.

- This is a job that requires much assistance from the programmers, but it should be the responsibility of analysts and designers since they presumably have a better understanding of the customer viewpoint.
- On some projects, the user writes these documents with the assistance of the Analysis and Design group.

2.1.2 Programming Group

- The programmers are the focal point in the organization.
- Their job may be thought of as a series of five steps:
  - (1) Detailed design.
  - (2) Coding.
  - (3) Module test.
  - (4) Documentation.
  - (5) Integration.
- The individual programmer is responsible for the first four and he at least assists in the fifth.

2.1.2.1 Detailed Design

- The programmer inherits from the designers the document called the Design Specification.
  - This is the **baseline** for all his work.
  - The programs he writes must mesh **perfectly** with the **baseline design**; otherwise, either his program or the baseline design must change.
- The individual programmer is assigned a piece of the baseline design by his manager or supervisor. Let's assume that this is a single module:
  - His first job is to design the module in detail, living within all the rules laid down in the Design Specification.
    - The programmer's vehicle for expressing this detailed design is the document called a Coding Specification (described later).
    - The programmer is expected to devise the best detailed design possible, consistent with the baseline design.
    - Violation of the baseline design is a capital offense. Off with the head!
- A problem arises:

  - Some programmers have no use for detailed design documents. They would rather code directly from the **baseline design** and skip the **detailed design**.
  - Other programmers would rather code first and design later.
- What to do?
  - First: it's a fair assumption that someday someone will need to modify your program system.
  - In order to do so that person will have to understand it and will require detailed documentation.
  - Unless one of these assumptions is false in your case, you'll need Coding Specifications.
- The next question is when to produce the Coding Specification?
  - Must they be done before or after coding?
  - Clearly, if our only concern is for the customer who might later modify the programs, the answer is after.
- In fact, all the customer cares about is that the detailed design documents be delivered with the programs.
- He doesn't care whether the documents are written before or after coding as long as they're accurate.
  - If the customer doesn't care, who does? The Project Manager needs the documentation **before** for the following reasons:
  - (1) The Coding Specification is the only vehicle to use in **reviewing** the programmer's work before it gets too far into coding and testing.
  - (2) It's the only reasonable document to use in continuing **design review**.
  - (3) Writing a detailed planning document forces a better product.
  - (4) If for any reason a programmer leaves your project, you'll be better off with a decent Coding Specification than with a half-coded, undocumented program module.
- Despite these arguments, you may occasionally allow coding to be done directly from the baseline design.
  - Some portions of the baseline design may have been done in sufficient detail to allow this.
- Like everything else on the project, if you go that way, make it the result of a reasoned decision.
  - Don't just shrug and let it happen.
- As an **alternative**, the Coding Specification can be substituted by the code itself, inserting substantial comments inside the code.
  - (1) Each module, procedure, function, object, etc can be preceded by a description of the functionalities, inputs, outputs, data structures description, relationship with other modules.
  - (2) The code itself can be well commented, including the description of the coding philosophy.

- (3) For the programmers is easier and in the same time very appropriate to explain their own coding decisions just inside the code.
- (4) There are special designed tools which can derive from the comments the code documentation in an automatic manner.

2.1.2.2 Coding

- **Coding** is the translation of the detailed design into computer instructions.
- As coding proceeds, changes in the detailed design will often be found advisable or necessary.
  - Making these changes is the **responsibility** of the programmer, except when the baseline design is affected.
- A manager should watch for programmers who have a tendency for writing unnecessarily tight, complex code.
  - Although there will be times when it will be necessary to save every bit and every microsecond possible, there usually are much more important considerations.
- The code should be readable by another competent programmer.
  - **Metzger** cites more than one instance when a pseudoprofessional programmer left behind a batch of code that worked but was unintelligible to anyone else but the programmer.
    - *In one case, the programmer left a marvelously efficient major program, but one day it became necessary to modify that program. The unsuspecting manager promised the customer that the modifications would be done and delivered in four weeks. Six months later, the job was not done, and the embarrassed manager finally had to have the program rewritten from scratch. An extreme example? Not at all. Watch out for it.*
- In programming **simplicity** pays off.
  - If you want to challenge your programmers, challenge them to write efficient code that even you (PM) can understand.

2.1.2.3 Module Test

- **Module test** is the process of testing an individual module in an isolated environment before combining it with other tested modules.
- The term **module** is equivalent in this context with: software units **SU**, software components **CSCI's** (Computer Software Configuration Items), or other specific terms.
  - Ordinarily, an individual programmer would be assigned what we have earlier called a "unit" the lowest-level module in the system; it might follow, then, that this discussion should be about "unit test".
  - Often, however, the lowest-level module in a given path will be at some higher level than "unit," perhaps what we have called "component."

- In those cases, the term "unit test" would be incorrect.
- So, the use of "module test," is an attempt to be more general.

- The **objective** is to determine that this module when inserted into the system, will do its job as a black box:
    - It should be capable of accepting its specified inputs and producing exactly the right outputs.
- Although the project may supply various test aids, **module testing** is the **programmer's job**.
- It is not recommended to impose any rigid, formal module test scheme only general **guidelines**.
    - (1) The programmer should put on paper, in his own words and in his own format, the steps he proposes to execute in order to test the module.
    - (2) He should discuss this informal "module test plan" with his manager, subject it to a walk-through, modify it if necessary, and execute it.
    - (3) Module-testing may involve no more than thorough desk-checking and clean compilation.
    - (4) A step further would be to "walk through" the code with another programmer.
- Many modules will need to be tested further in a "stand-alone" manner, that means, not yet combined with other system modules.
    - This can be achieved by providing test data and test drivers.
    - **Test Drivers** are programs whose purpose is to supply a special test environment for the module.
- Decisions concerning the nature and extent of module test will be influenced by whether you are testing top-down or bottom-up.
- (1) In **bottom-up** testing, **test drivers** of various kinds would normally be used to represent the "**top**" of the **system**.
    - A **Test Driver** is the part of the program system above the tested module in the hierarchy and responsible for invoking the module in the first place must be ready.
- (2) In **top-down** testing, the "**top**" of the system already exists.
    - The module can be added to the existing system.
    - What needs to be simulated in this case is any relevant module below the one being tested.
    - In this case, the programmer writes code called "**stubs**" to stand in for the missing lower modules.
        - Stubs are generally simpler than drivers and test executives. They may simply record that they have been called and return control to the invoking module.
        - Stubs may go further and simulate the actions that will eventually be taken by the real modules for which they are temporarily substituting.

    - The most efficient way to provide stubs is to build the entire system of stubs in advance, rather than have them introduced by individual programmers as they are needed.
        - As new modules are completed and inserted into the system, corresponding stubs can be deleted.

2.1.2.4 Documentation

- *"Document unto others as you would have them document unto you!"* say **Kreitzberg** and **Schneiderman**.
    - Here is the point where an otherwise good product may be poorly represented.
- The programmers are responsible for the documents that describe in detail how the system has been constructed.
- The vehicle they use is the Coding Specification and also the Documentation Plan.
    - This is the same document the programmer used to show his detailed design before the module was coded (fig. 6.2.1.2.4.a.)

CODING SPECIFICATION (TEMPLATE)

DEPARTMENT:

PROJECT:

DOCUMENT NUMBER:

APPROVALS:

DATE OF ISSUE:

REALISED:

SECTION 1**: SCOPE**

A standard statement: This document contained the detailed description of program module ______________.

SECTION 2: **APPLICABLE DOCUMENTS**

A standard statement keying this detailed specification to the appropriate part of the Design Specification: "The design described in this document represents the portion of the baseline design shown in the Design Specification, document number __________________, subsection _________________."

SECTION 3**: THE DETAILED DESIGN**

3.1. **Program Structure**

This section describes the logic of the program module according to the standards an conventions adopted and stated in the Design Specification, subsection 3.3.

3.2. **File Structures**

3.2.1. System Files

This subsection makes explicit references to the system file layouts contained in the Design Specification. File layouts may be repeated here if the programmer feels this would enhance the clarity of this document.

3.2.2. Local Files

A complete, detailed description of all local files. Local files are unique to this program module. They are not accessed by other modules.

SECTION 4: **LISTINGS**

This is a standard reference to the detailed, machine-produced instruction listings showing the complete set of object code for this module, including any local files.

Fig. 6.2.1.2.4.a. Coding Specification Template

- When the module has been tested, its Coding Specification should be corrected and completed by the addition of the machine listing of the actual coded module.
  - Thus, the original Coding Specification showed detailed design intent, and the completed document shows final design along with resultant code.
- The logic and the code described in the Coding Specification should be completely accurate and consistent.
- The connective tissue tying all the individual Coding Specifications together is the Design Specification.
  - That combination should completely and adequately describe the program system structure.
- As was mentioned, Coding Specification can be included as comments in code.
  - In this case, the Coding Specification document can miss or can be more formal including general considerations.

2.1.2.5 Integration: "Top-Down"

- **Integration**, or **integration testing**, is the process of gradually adding new modules to the evolving system and testing to assure that the new module and the system perform properly.

- Let's assume we've chosen **top-down integration testing** as our project's approach, consistent with our use of top-down design and top-down structured programming. How to proceed? There are several avenues:
- (1) All integration testing could be done by a separate group **Integration Test Team** (see Fig. 6.2.1.a) whose sole function is just that.
    - The members of the group would **not** themselves actually write any of the programs.
    - Individually tested modules would be turned over to this Integration Test Team.
    - The team would add each module to the developing system and test it according to a predetermined **integration test plan**.
- (2) All integration testing could be done by individual programmers (eliminate the "Integration" group in fig.6.2.1.a.).
    - Each programmer would be responsible for adding his modules and running tests according to the test plan.
- (3) Integration testing could be handled by a group which also has programming responsibility.
    - Logically, this would be the group charged with writing the higher-level modules in the program hierarchy —the "executive" program, "control" program, or whatever we name the set of code which serves as the system's framework.
- **Metzger** suggests that the third alternative **Choice** (3) is generally best.
    - This choice guarantees that the people responsible for integration have the most intimate knowledge of the overall system.
- **Choice** (1) is workable, perhaps even best, for large projects, where there are so many programs involved that integration is a huge task. But keep in mind:
    - A separate group with no part in the actual programming may be too far removed from the system.
    - They would be in a less favorable position to spot problems and devise solutions.
    - They might be less motivated, since they have no code of their own at stake.
    - A strong counter-argument, of course, is that such a separate group could be more objective, for the very reason that their own code is not under question.
- **Choice** (2) invites chaos.

2.1.2.6 Integration: "Bottom-Up"

- As tested modules become available from the programmers, the process of integration begins.
- Theoretically, this means:
    - (1) The units are combined and tested together to form components.
    - (2) These components are grouped and tested to form packages.

- (3) And so on up the pyramid (see Fig. 6.2.1.2.6.a) until the complete system has been put together and progressively, exhaustively tested.

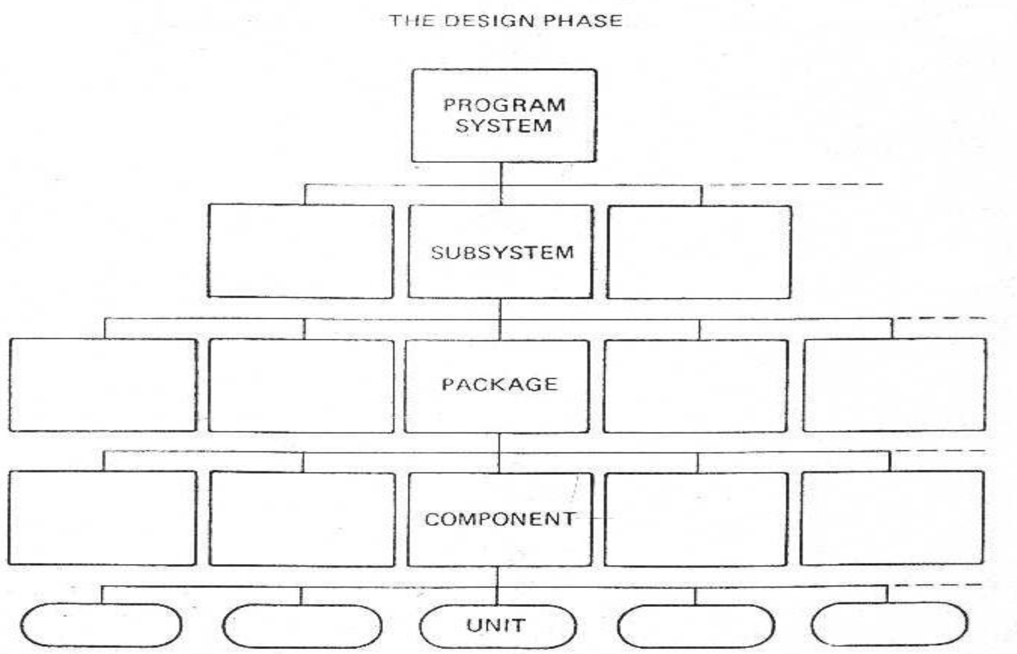

Fig. 6.2.1.2.6.a. Program System Hierarchy

- In practice, you will usually find that no matter how neatly you lay things out on paper, the process is not quite that clean and orderly.
  - (a) One reason is that some "components" will be ready while other "units" are still being coded.
  - (b) Another is that when bugs show up at each level of test, buggy units have to be sent back to the drawing board for more work.
- Nevertheless, **integration testing** should be planned as an orderly building process allowing for **detours**.
- There are at least two ways to proceed with **bottom-up integration.**
  - (1) One is to have programmers produce the lowest-level modules, that is, units, and turn them over to a separate group for integration.
  - (2) Another is to have the programming groups integrate their portions of the system and turn their work over in much larger chunks to a separate group.
    - The first way may be theoretically more attractive.
    - The second way is more practical and vastly more satisfying to the programmers because it gives the individual programmer more responsibility than to simply keep producing small parts (units) for someone else to assemble.
- As in the case of top-down testing, the group responsible for integration also could be responsible for writing the basic control program for the system.

- It will be necessary during earlier test planning to decide at what level work done by the Programming Groups will be turned over to the Integration Group.
  - For **example**, you might give your Programming Groups responsibility for detailed design, coding, module test, documentation, and integration up through the **program package level** (see fig. 6.2.1.2.6.a).
  - When integration of an individual package has been completed, the package is turned over to the Integration Group for final merging of packages into **subsystems** and subsystems into a **system**.
  - You may, of course, choose a different level at which to submit modules to the Integration Group.

2.1.2.7 Integration: The Test specification

- Since the Integration Test Specification is key to the formal testing process, let's look at it in a little more detail

--

TEST SPECIFICATION (TEMPLATE)

(INTEGRATION, SYSTEM, ACCEPTANCE, SITE)

--

DEPARTMENT:

PROJECT:

DOCUMENT NUMBER:

APPROVALS:

DATE OF ISSUE:

REALISED:

--

SECTION 1**: SCOPE**

There are four separate sets of test specifications: Integration, System, Acceptance, and Site Test Specifications. The outlines for all four are identical, except that the appropriate qualifier ("integration", "system", "acceptance", or "site") must be inserted. The content of the specifications may, of course, vary considerably, although two of them (acceptance and site) will often be identical. This section, Scope, should serve in each case as an introduction to the document, describing its intent and how it is to be used.

SECTION 2: **APPLICABLE DOCUMENTS**

SECTION 3**: (INTEGRATION, SYSTEM, ACCEPTANCE, SITE) TEST OVERVIEW**

3.1. Testing Philosophy

3.2. General Objectives

3.3. General Procedures

3.4. Success Criteria

SECTION 4: **COVERAGE MATRIX**

*A chart listing along the vertical axis the **areas** to be tested and along the horizontal axis the **test case number**(s) covering each area. When complete, this chart amounts to a cross-reference between all areas to be tested and all test cases covering those areas. See also* Test Case Template.

Fig.6.2.1.2.7.a. Test Specification Template

- The Integration Test Specification must be ready to use early in the **Programming Phase**, when integration actually begins.
  - The document must, therefore, be finished during the **Design Phase** (see fig. 6.2.1.2.7.b.)

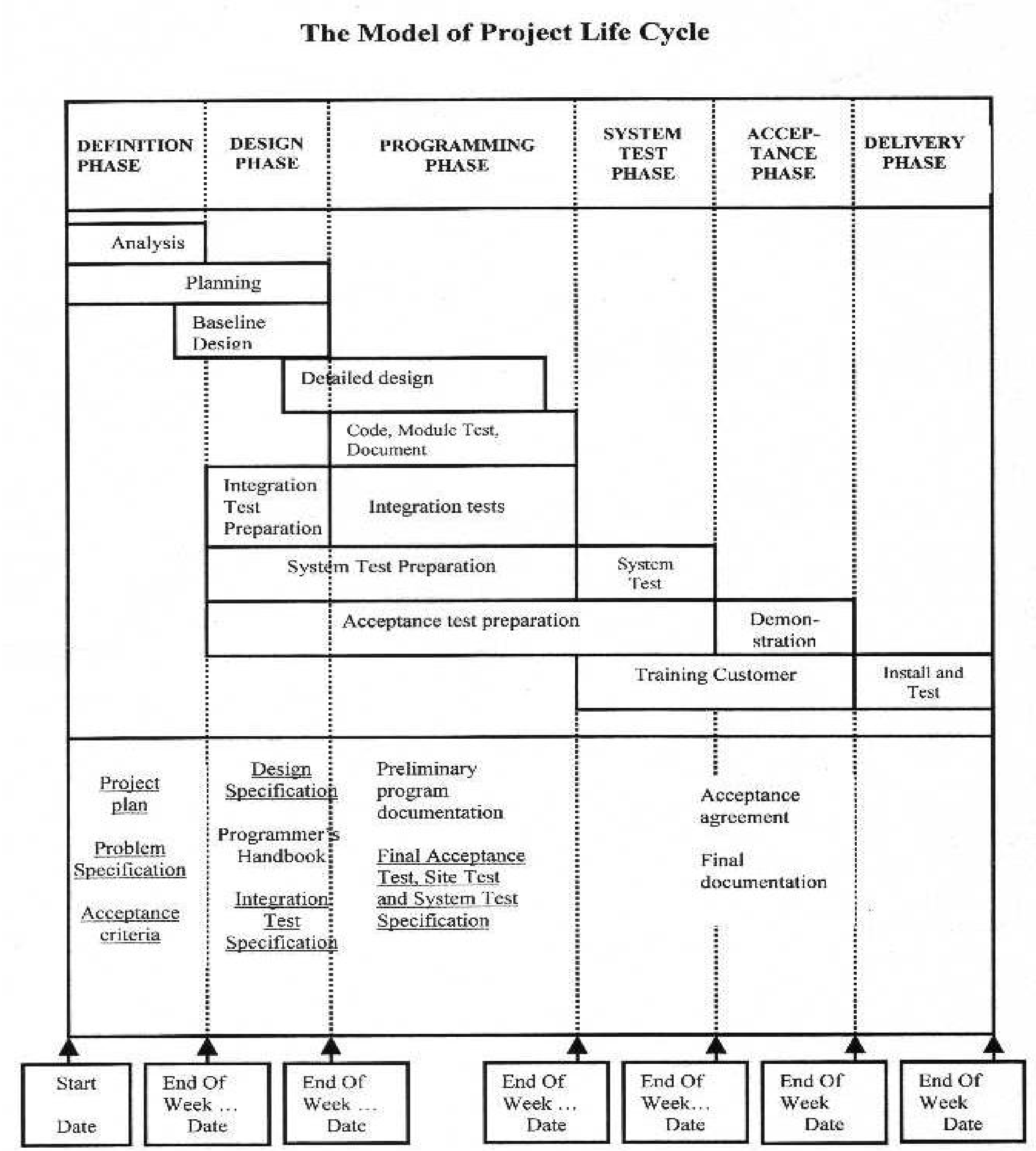

Fig. 6.2.1.2.7.b. The model of the Project Life Cycle

- Integration Test Specification describes:
  - (1)Test philosophy.
  - (2) Objectives.
  - (3) General procedures and tools.
  - (4) Success criteria.

    - (5) Coverage Matrix - showing which **specific tests** (or "test cases") cover which **functional areas** of the program system.
- Integration Test Specification is required whether top-down or bottom-up testing is used.
- The Integration Test Specification calls for a number of test cases.
    - A **test case** contains the detailed objectives, data, and procedures required for a given test.
    - A look at the coverage matrix mentioned earlier should show which test case or cases apply to a given functional area.
- The key items in a **test case** are:
    - (1) The data required for the test.
    - (2) A script.
        - A script (often called a scenario) is a set of step-by-step procedures telling, for this test,
            - (a) What is to be done.
            - (b) Who is to do it.
            - (c) When it is to be done.
            - (d) What to look for.
            - (e) What to record.
        - Similar scripts are described in the next chapter in which system testing is discussed.
- Tacking test cases onto a basic test specification, rather than writing one huge testing document, is another example of modularity.
    - It's so much easier to see where you are when things are done in clean, finite chunks.
    - And going back to repeat a test is simple when you can point to a single test case and say, "Do it again."
- Fig. 6.2.1.2.7.c. illustrates four formal test specifications that are discussed here and in the next two chapters

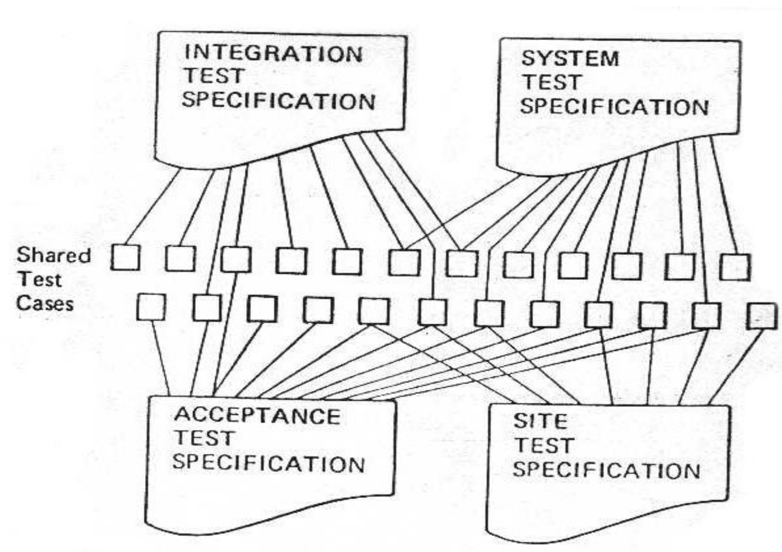

Fig. 6.2.1.2.7.c. Types of Test Specification

- Each type of test has the same conceptual organization.
    - In fact, as the fig.6.2.1.2.7.c. shows, certain test cases may serve equally well during integration testing, system testing, acceptance testing, and site testing.
- Good planning early in the project will enable you to maximize the multiple use of test cases.
- It's recommended that the tests to be carefully laid out in advance so that when integration time arrived to concentrate on running the tests, evaluating results, and making fixes.
    - It's too late to begin test planning when testing actually begins.
        - All you can do then is fumble and pray that baseline designing and module testing have been done so well that things fall into place easily.

2.1.3 Test Group

- During the **Programming Phase**, the job of the Test Group is to get ready for **system** test, **acceptance** test, and **site** test.
- This is not the same group responsible for integration test.
    - Its orientation is quite different.
- The **integration testers** were concerned with:
    - (1) Putting program modules together.
    - (2) Testing interfaces.
    - (3) Testing both system logic and function.
- The **system and acceptance testers** are almost solely concerned with testing function.
    - They are not directly concerned with the structure of the program system.
    - They are concerned with:

        - (1) How the program system performs.
        - (2) How well it satisfies the requirements stated in the **Problem Specification**.
- The Test Group comes into prominence in the next two chapters, but they must prepare now, during the **Programming Phase**.
- During the **Programming Phase** their job includes:
    - (1) Writing test specifications.
    - (2) Building specific test cases.
    - (3) Predicting results.
    - (4) Getting test data ready.
    - (5) Making tentative arrangements for computer time.
    - (6) Setting up test schedules.
    - (7) Organizing test libraries.
    - (8) Choosing and securing test tools.

2.1.4 Staff Group

- Some technical people look at staff groups as hangers-on, paperpilers, drains on the overhead, and general pains in the neck.
    - Occasionally that view is justified, for some managers surround themselves with so many assistants of various kinds that it's difficult to determine who is the manager.
- This happens in big organizations because rules, regulations, and associated paperwork get out of hand and staff people are hired to control them.
- Some staff people create more **rules**, **regulations**, and **paperwork**, causing the disease to spread rapidly.
    - This occurs even in smaller organizations, particularly when the customer happens to be big.
- What stands out in this observation of staff groups:
    - After a while no one knows why they're there, let alone why they were originally formed.
    - A manager often takes on a staff member to work in an area but doesn't really define that person's job.
        - The result is that his job overlaps other people's jobs.
        - This situation is known as the concept of stuff syndrome.
    - If the staff member is industrious, he will define his work scope and very soon will generate requirements for more staff help.
- There's only one way to avoid amoebalike growth of staff functions: PM must define the staff member's job as clearly as he would define a programmer's job.

    - Surely PM wouldn’t hire a programmer and tell him to find a piece of programming work to do. PM’d say:
        - Here’s the overall job,
        - Here’s the piece I’d like you to do,
        - Here’s the schedule,
        - This is how I want you to report progress!
- Do the same with a staff member.
    - Don’t hire one unless you can assign specific responsibilities.
- The two kinds of staff functions that you’re likely to need on your project are technical and administrative.

2.1.4.1 Technical Staff Functions

- The people supplying technical support must themselves be technically competent.
- Their function is to focus on tasks that help all the other technical people on the project.
- Their specific jobs are:
- (1) Controlling computer time. (If the situation requires this activity)
    - All computer time needs should be funneled to one person who should secure the time each week, schedule it as equitably as possible among those who request it, resolve conflicts, observe priorities, keep accurate records of time requested and used, plan for time needed weeks and months ahead, and dispense the aspirin when time is cancelled.
    - Part of this job is to set up and enforce the rules for use of computer time. The staff member should write the procedures (crisply and clearly) for submitting remote runs or for "hands-on" use of the machine, arrange for pickup and delivery of test runs and computer outputs, and provide for such physical facilities as bins and cabinets and for courier service if necessary.
    - In short, this person should be the interface between the computer installation and its users.
- (2) Supplying technical assistance.
    - The same staff member should also supply technical assistance:
        - Estimate the amount of service needed and arrange for it.
        - Take care of incidental problems, such as the need for special requirements.
        - Determine priorities whenever necessary.
- (3) Maintaining the Programmer’s Handbook.
    - Organizing the handbook, getting it distributed, and keeping it updated should be done by the technical staff.
- (4) Training.

    - Unless training is a very large function for your project, the Staff Group should be responsible for both internal and external training and should provide for instructors, training facilities, written training materials, schedules, and training cost estimates.
- (5) Handling special technical assignments.
    - Occasionally, there are specific, short-range technical jobs to be done, but there's no specific place to assign them.
    - *For **example,** there may be a troublesome problem that cuts across several of your groups and must be tracked down. It's recommended that you include someone in your staff estimates to allow for this fire-fighting.*

2.1.4.2 Administrative Staff Functions

- Before presenting the specific functions of Administrative Staff let say what an administrative staff is not:
    - (1) It's not project management.
        - It's an aid to project management.
    - (2) It's not a quality control department.
        - Quality control is a management function, and quality will not be assured by having ten thousand administrators looking over the programmer's shoulder and filling out forms and reports.
    - (3) It's neither a personnel management group nor a salary administration group.
        - Those are management jobs.
- The functions of the administrative staff are as follows:
    - (1) Document control.
        - This is as vital a function as any on the project. If documentation goes careening out of control, the project will wind up on the sick list.
        - The administrative staff has the job of handling documentation as laid out in the Documentation Plan.
        - The job includes:
            - (a) Setting up and operating the Project General Library.
            - (b) Handling all interfacing between the project and any outside technical publications organization.
            - (c) Keeping track of document numbers and issuing new ones on request.
            - (d) Publishing or updating a periodic documentation index listing the names and numbers of all project documents.
            - (e) Providing for all reproduction services and equipment.
    - (2) Report control.

        - (a) The staff assists Project Manager by gathering status data and drafting status reports from PM (Project Manager) to his management and from PM to the customer.
        - (b) It also obtains and distributes to PM and all managers on the project periodic financial status reports.
        - (c) The staff prepares a final report, the Project History described earlier.
        - (d) If PERT or other automated report and control systems are used on the project, the staff prepares its inputs (from data obtained from line managers) and distributes its outputs.
    - (3) Contract change control.
        - When a contract change has been agreed to on a technical level, the staff handles the job of completing the paperwork showing that the customer formally agrees to the change.
        - Part of the job is assessing the cost of the change. In doing this, the staff will coordinate among four parties:
            - (1) The technical people who make the first estimate of cost.
            - (2) The PM (Project Manager).
            - (3) The company's financial and legal services.
            - (4) The customer.
    - (4) Secretarial and typing support for the project.

2.1.5 The Numbers Game

- The Metzger's version:
- Figure 6.2.1.5.a. is identical to Figure 6.2.1.a (b) but with two pieces of information added:
    - (1) A summary of the jobs of each group
    - (2) The numbers of people in each group. The numbers are, of course, subject to argument and will vary from one project to another
- The rationale for choosing the numbers for a 20-30 people software project:
    - (1) Project manager.
        - Of course there is a "one".
        - There are sometimes two people.
            - The second is called an assistant project manager or some other meaningless term.
            - Avoid having two because it makes it difficult to know who's responsible for what.

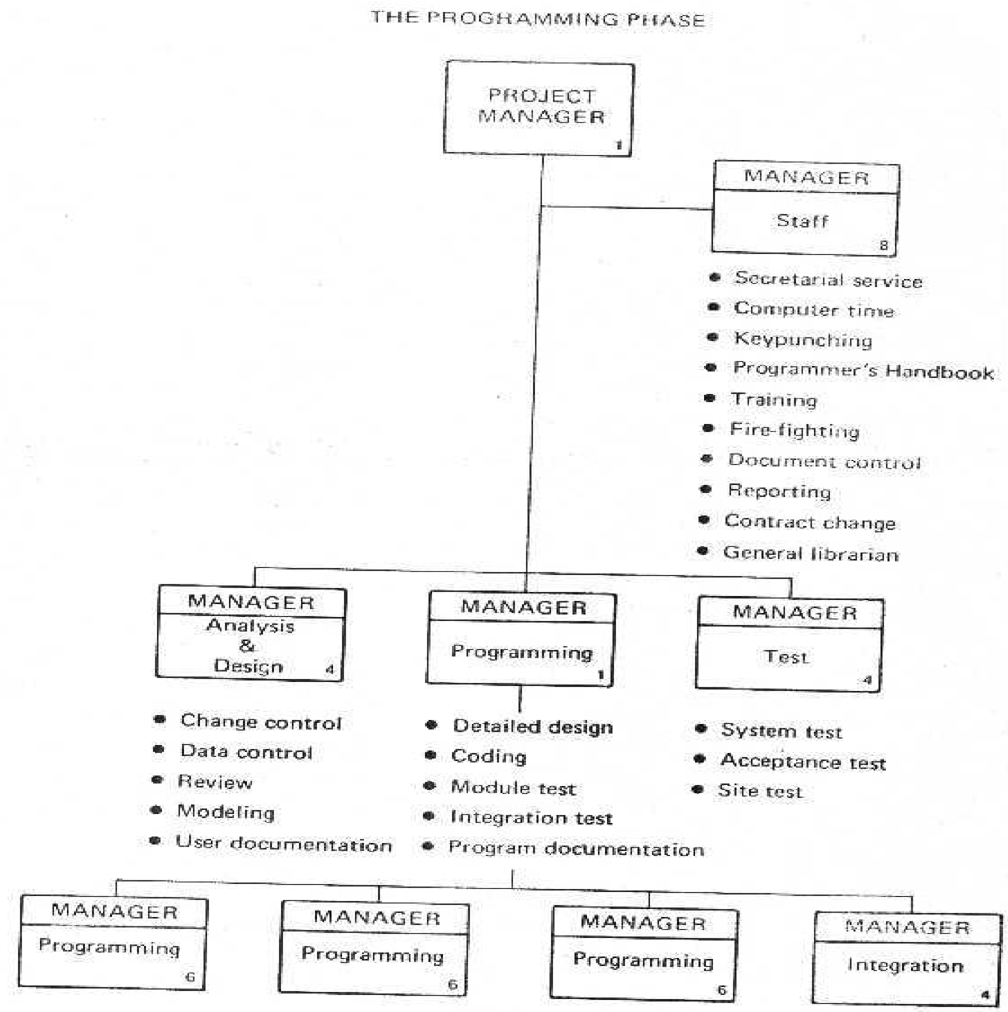

Fig.6.2.1.5.a. The numbers game

- (2) Staff
  - One manager and 1-2 workers.
  - The 1-2 are technical staff and administrative workers.
  - The administrative member has somtime the role of secretary.
  - The administrative member, along with the group manager, handles contract matters, documentation control, and report preparation.
- (3) Analysis and design.
  - 2-3 workers plus the manager should be sufficient to handle the presented functions.
  - Toward the end of the **Programming Phase**, the number may be reduced, or more likely, the group may merge with the Test Group in order to help perform system test and acceptance test.
- (4) Test.
  - During this phase, the Test Group's main responsibility is getting ready for system test, acceptance test, and site test.

        - 3-4 people should suffice.
    - (5) Programming.
        - Three relatively small groups composed by 3-6 people.
        - All the groups in this organization are intentionally small.
        - The job of a first-level manager is tough.

If you define properly this person’s job, you shouldn’t give him ten or twelve or twenty people to manage and then expect a first-rate job.

        - Keep the programming groups small enough that the manager can become intimately involved with the details of the work.
            - Working with a horde, you can expect this person to become a paper shuffler.
    - (6) Integration
        - The group responsible with integration of the modules.
        - The integration group may contain 1-2 persons.
        - If you are doing top-down integration, the group of four called "Integration" could be labeled "Programming."
        - It would be responsible for the high-level system modules and for integrating the work of the other groups with its own.
    - (7) In the same time a SCM Group (Software Configuration Management) should be added in parallel with Integration Group containing 1-2 persons.

2.2 Team Organization. Chief Programmer Team

- The team approach is a way of organizing around a group of specialists.
    - The embodiment of the approach in programming is called the **Chief Programmer Team**.
- IBM’s **Harlan Mills**, originator of the concept, compares the Chief Programmer Team to a surgical team, where a chief surgeon plans and performs an operation with vital help and backup from highly skilled assistants, both surgeons and nonsurgeons.
    - What follows is an overview of how this idea is put into practice.

2.2.1 How It Works

- The core of a Chief Programmer Team would normally be three people:
    - (1) Chief programmer.
    - (2) Backup programmer.
    - (3) Technical librarian.
- (1) The **Chief Programmer**
    - **Chief programmer** is the technical manager responsible for the development of the program system.

    - This person will normally write at least the critical "system" modules — that is, the portion of the program system exercising control over, and interfacing with, all the lower-level "working" modules.
    - Depending on the total size and complexity of the job, he and the backup might write the entire program system.
        - Where others are involved, the chief programmer assigns work to them and integrates all their modules with his own.
    - The chief programmer is the main interface with the customer, at least in technical matters
        - There may be a managerial counterpart who handles non-technical tasks.
- (2) The **backup programmer**
    - Assists in any way assigned by the chief programmer,
        - His primary function is to understand all facets of the system as well as the chief programmer does.
        - His second function is to be ready at any time to take over as chief programmer.
    - The backup programmer is normally assigned specific portions of the system to design, code, and test, as well as other duties for example, preparation of a test plan.
- (3) The **technical librarian (configurator)**
    - The technical librarian is a different person from the "general librarian" who runs the project's general library and has the normal responsibility usually associated with a library.
    - The technical librarian is responsible for running the Development Support Library or to manage the SW Configuration Tool used by the organization (CVS - Control Version System, Continuous, ClearCase, CM Synergy, etc).
        - This person is a full and vital member of the team, not on part-time loan from somewhere else.
    - The librarian 's duties include preparing machine inputs as directed by the programmers, submitting and picking up computer runs; and filing all outputs.
- This team of three may be augmented by other people, such as:
    - (1) Programmers who are specialists in a given area.
    - (2) Less senior programmers who code a specific portion of the system designed by the chief or the backup programmer.
        - There is no sure limit to the size of such a team, but six to eight, by consensus and experience thus far, seems to be the top of the range.
- The **idea** of the Chief Programmer Team was born as a result of the search for better, more efficient ways of producing complex programs free of errors.
    - Mammoth undertakings, such as IBM's OS/360, had made clear that ways must be found to dramatically improve the quality and to reduce the cost in future software development efforts.

    - The thrust of the **team concept** is that those goals of quality and efficiency can be achieved through a very tight, disciplined organization of a small number of highly motivated people very experienced and skilled in all aspects of program development, from analysis down through design, code, test, and documentation.
    - In keeping the number of people small, the human communication problems (and therefore the program communication problems) could be drastically reduced.
        - Just compare the possible interfaces among, say, six people as opposed to thirty!
- But simply choosing top people and organizing them in a small group is not enough. They need better tools with which to work:
    - (1) Development Support Library or SW Configuration tools.
    - (2) Top-down development, including design, code, test and documentation tools.
    - (3) Structured programming, Object oriented technologies, or UML tools.
    - (4) These kind of tools are now recommended, of course, whatever the organizational structure is.
- As the Chief Programmer Team concept is tried by more and more organizations, it will inevitably be refined and will lead to other ideas and tools.
    - One natural outgrowth is the use of a "team of teams," where a large problem is broken down in a hierarchical manner, and major subsystems assigned to individual teams which are responsible to a "higher-level" team, which is in turn responsible for the system as a whole.

2.2.2 Project Organization usig Chief Programmig Team Approach

- Suppose you are to use the team approach on your project.
- How might the total organization look, and how would the various functions be handled?
- Your organization might look something like Figure 6.2.2.2.a. which shows a little more than half as many people as under conventional organization.
- Numbers here are not very meaningful, since we're not discussing a job of known size.
- For a given system, you might be able to eliminate half the Staff Group, all of the Analysis and Design Group, and some of the Test Group, as in Figure 6.2.2.2.b.

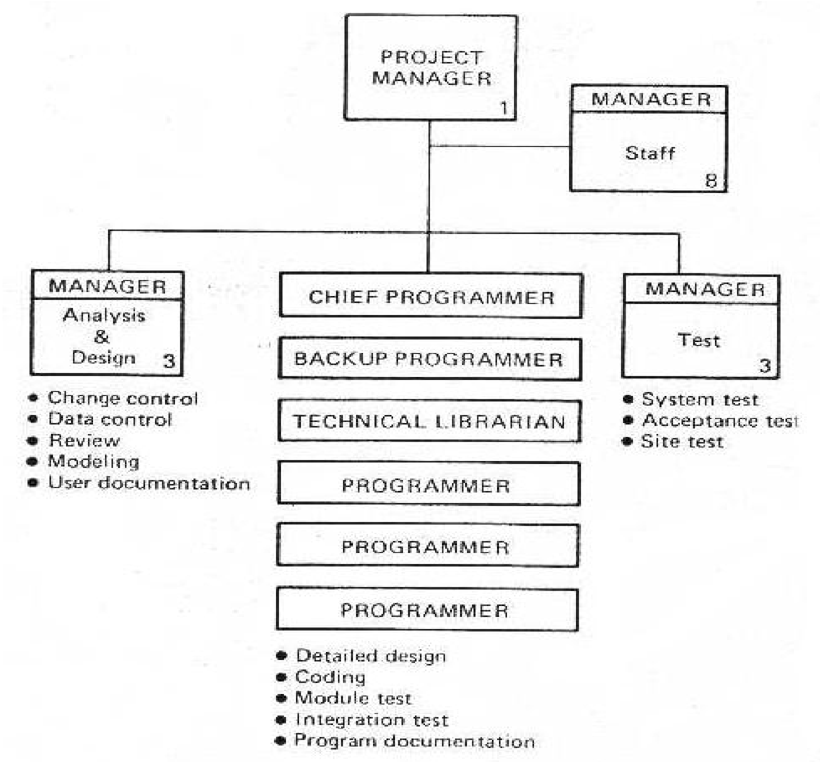

Fig.6.6.2.a. Team Project Organization (1)

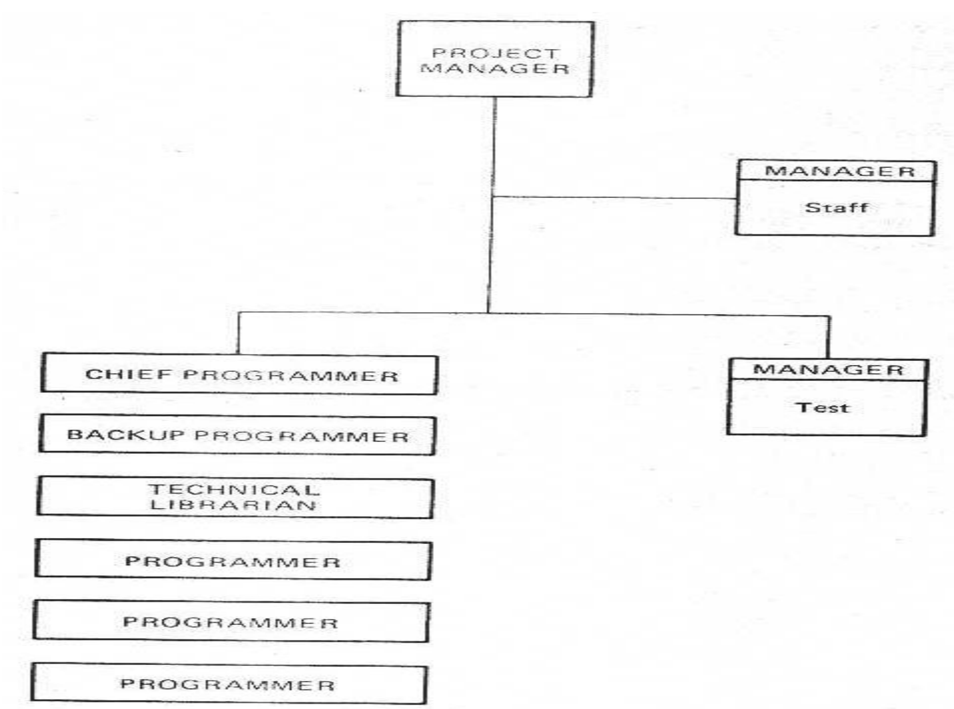

Fig. 6.6.2.b. Team Project Organization (2)

- Those functions might all be handled by the Chief Programmer Team.
    - The answers concerning numbers of people depend not only on the **nature and complexity** of the job, but on the **talents** of the people comprising the Chief Programmer Team.
    - The team approach assumes highly skilled and dedicated team members, but there is nothing quantitatively fixed about those terms.
        - How highly skilled?
        - How dedicated?
        - There is nothing about the team approach to relieve management from making critical judgements and decisions.

3 Change Control

- Choose your change control procedure thoughtfully.
    - Too much control will suffocate you; too little and you will drift.
- Don't build a change control empire in which there are volumes of procedures describing how to handle any conceivable change.
    - Think about the critical items over which you really need control, and leave the rest for day-to-day management action.

3.1 Baseline Documents

- First, you must decide what to control against, that is, what are the things you want to use as foundations, or baselines.
- **Metzger** recommends **two** baseline documents:
    - (1) The **Problem Specification**.
    - (2) The **Design Specification**.
- If you put your effort into making these two documents fine pieces of work in the first place and set up your procedures to control changes to them, then it's hard to go wrong.
    - Conversely, if your baseline documents are shallow and poorly done, or if you fall to control changes to them, it's hard to go right.
- There is a third kind of baseline you may need to consider.
    - The two mentioned above are established early in the development cycle and are used to guide production of the program system.
    - If you are responsible for maintenance or for more versions of the system beyond the initial delivery, then (3) The **Delivered Program System** becomes a new baseline.
    - In other words, in working on a second or third or n-th version of the program system, you may use the last delivery as the baseline.
- Here, however, we will discuss only a single-delivery development cycle.

3.2 Control Procedures

- If we agree on controlling change against the Problem Specification and the Design Specification, we can now consider a **simple control mechanism** that you can tailor to fit your needs.
- Whenever an individual sees a need for a **change** that he thinks may affect one of the baselines the **folowing steps** are to be accomplished:
  - (1) He proposes a **formal change**.
  - (2) The **Analysis and Design Group** analyzes the proposed change and recommends adoption or rejection.
  - (3) The recommendation is then submitted to the **Change Control Board** which makes its decision, subject to override by either you or the customer.
  - (4) The **Analysis and Design Group** documents the decision, and the change, if adopted, is implemented.
- Now let's take a closer look at how this procedure might work.
- (1) **Proposing a change.**
  - Anyone, either in your organization or the customer's can propose a change.
    - To do so, a simple Change Proposal form is filled out that describes the need for the change, and, if possible, the way to make the change.
  - **As a rule**, a programmer proposes a change only if he thinks one of the baselines might be affected.
    - A Change Proposal is not submitted every time a piece of detailed design for a module is slightly altered.
  - There is one kind of change that falls outside this formal control procedure, but it must be mentioned in passing:
    - Suppose a programmer wants to make a change to one of the modules already submitted for integration test.
      - The change affects neither the Problem Specification nor the Design Specification, but it does affect the detailed design — the Coding Specification for the module.
      - The change might be to correct a late-found bug or to improve a piece of code.
    - Whether or not to accept the change in this case should be up to whoever is in charge of integration testing involving that module.
      - If the change makes sense, it should be accepted only in the form of a new copy of the program module, a corrected Coding Specification, and an updated module identification.
      - **No change** should ever be accepted without a change in the module identifier. Every time you let one through you lose a little more control.
- (2) **Investigation.**

  - **All** proposed changes are investigated by the **Analysis and Design Group**.
  - One investigator is assigned to any given proposed change.
    - The investigator scans the proposal to get an idea of its importance and impact, and then schedules it for a decision at a future meeting (usually the next scheduled meeting) of the Change Control Board.
  - If the change is urgent, a special meeting may be called as soon as the investigator has enough information to make a recommendation.
  - The **investigator**:
    - (a) Looks into all pertinent aspects of the change.
    - (b) Writes down a report.
    - (c) Sends a copy to each member of the board within a reasonable time (say, two working days) before the board is to meet.
  - Each investigator's report should include:
    - (a) A summary of the proposed change.
    - (b) The originator's name and organization.
    - (c) Classification of the change (Type 1 or 2) as determined by the investigator (see below).
    - (d) The impact of the change on costs, schedules, or other programs.
    - (e) A recommendation **for** or **against** adoption.
- (3) **Kinds of changes.**
  - The investigator may put the change into either of two categories:
    - **Type 1** - if the change affects either of the baseline documents or would cause a cost, schedule, or other impact.
    - **Type 2** if the change affects neither baseline and has negligible cost, schedule, or other impact.
  - Be sure that changes don't too easily become categorized as Type 2, when they really do cost something and ought to be Type 1.
    - Don't be "nice guys" and allow the project to be nibbled to death by too many Type 2 changes.
  - You can make things a lot more complicated, but don't.
    - There's no sense inventing a dozen different change categories to cover combinations of situations.
    - Either the change will cause some problems (Type 1) or it's no sweat (Type 2).
    - Even the ponderous machinery of the federal government's Configuration Management gets by with only two categories of change. Surely you don't want to be put to shame by the world's greatest bureaucracy!
- (4) **Change Control Board.**

    - The board should be comprised of representatives from various project groups.
    - At periodic meetings (say, once a week) the board should consider all scheduled change proposals.
    - The board discusses each change and decides how to dispose of it.
    - Project Manager will have to decide whether: (a) the board will operate democratically by voting on each issue or (b) whether it will allow the chairman to make the decision after hearing the arguments.
        - Democracy is great, but you may find things move much faster if you give the chairman the power to decide what the board's recommendation should be.
        - PM (Project Manager) can always overrule if one of the other board members convinces him that a particular decision was a bad one.
            - Don't overrule too often or PM'll destroy the chairman.
    - The **Change Control Board** should be comprised of the following:
        - **Chairman**: the manager of the Analysis and Design Group. He's got to be tough, fair, technically sharp and politically savvy.
        - **Permanent members**: the manager of the Programming Group, the manager of the Test Group, and the manager of the Staff Group.
        - **Others**: the investigator for the proposal being considered; technical personnel invited by any of the permanent members.
    - At any board meeting, then, there will be at least five participants, four regular members and one investigator.
        - It's important not to let these meetings get too big by inviting too many extras, but obviously if there is someone who can shed more light on the proposal than anyone else, that person should be present.
        - Often this will mean that the person who proposed the change will be there.
    - Should the customer be invited to board meetings?
        - Generally, yes, although there may be times when you would rather not have him around.
            - For **example**, the customer should be excluded whenever company proprietary data are to be exposed or discussed.
        - The best way to handle this question is simply to level with the customer.
            - Then he can be invited whenever the cost is clear.
- (5) **Types of recommendations.**
    - **If** the board agrees with the investigator that a change is a Type 2, the change should be automatically accepted and no further board action is necessary.
        - It is treated following the Bug Fixing Procedure.

  - **If** it is a Type 1, it must recommend to you how to dispose of the change. There are two possibilities:
    - **Acceptance** of the change and an indication when the change should be made (immediately or in some future version of the program).
    - **Rejection** of the change.
- (6) **Customer directed changes.**
  - Some changes will be insisted on by the customer.
  - They must still be investigated, considered by the board, their cost and impact estimated, and formally approved by the customer.
  - It is always the customer's right to override any decision by the board, as result, appropriate contract changes are negotiated.
- (7) **Implementing a change.**
  - Depending on the board's recommendation for a Type 1 change, two concluding actions are possible:
  - If the board recommends **rejection**, the proposal is logged as closed.
  - If the board recommends **adoption**,
    - (1) The **investigator** writes up a summary of the change, its cost, and the schedule for making the change.
    - (2) The package is then given to PM (Project Manager) for signature.
    - (3) PM signs it.
    - (4) If there is a cost or schedule impact, PM sends the package to the customer for approval.
    - (5) When the customer approves it in writing, the investigator finally distributes to all concerned a written description of the change.
    - (6) Now the change can be implemented by the programmers.
- The above is the formal procedure.
- There will be many instances when a change cannot be held up for days or weeks.
  - Here PM (Project Manager) can **speed up** the process by investigating immediately, calling a quick, special board meeting, writing out the recommendation in longhand, approving it verbally, getting the customer's agreement verbally and telling the programmer to go ahead.
  - But, make sure that the formal paperwork follows — customer's approval, change notice, and so on. Otherwise, PM will soon lose track of things.
- Periodically, the board chairman should give to PM a summary list of all changes considered, the board's recommendation in each case, and a very brief statement of the main arguments for and against.
  - Then if PM spots something with which he thinks he disagrees, PM can have the item reconsidered.
- The Analysis and Design Group is also responsible for keeping track of a schedule for all changes.

        - On many projects changes come thick and fast.
    - Some are designated for immediate implementation; some are deferred to some specific later version of the program system.
    - It's important that everyone — you and the customer — know exactly when accepted changes will be made.

4 Programming Tools

- Tools for doing the programming job must be selected before the **Programming Phase** actually begins.
- Some tools, such as operating systems, simulation models, HIPO, pseudo code, structured charts, flow charts, decision tables, and coverage matrices, were mentioned in the last chapter.
    - Useful as analysis and design tools, they continue to serve as programming tools, as well.
- The following are brief descriptions of other tools that you as PM should consider during programming.
    - Some of them are programs, some are hardware, and some are simply documents.
- In larger projects (see Chapter 10) it is often necessary to have a separate group of people to provide the rest of the project with suitable programming aids.

4.1 Written Specifications

- Our approach outlined three key documents: **Problem Specification, Design Specification,** and **Coding Specification**.
- Earlier in this chapter it was mentioned another document called an **Integration Test Specification.**
- These tools are so important that they deserve repeated emphasis.
- (1) **Problem Specification.**
    - Is written by analysts during the **Definition Phase**.
    - It's a baseline.
    - it describes the problem that your project is all about.
- (2) **Design Specification.**
    - Is written by program designers during the **Design Phase** to describe the overall program system, the solution to the problem.
    - It is also a baseline that sets the stage for all ensuing detailed design.
- (3) **Coding Specification.**
    - These specifications are written by programmers during the **Programming Phase**.
    - Each specification describes in detail the design for a portion of the overall system laid out in the Design Specification.

- Coding is done from these specifications.
- Project Manager can decide that the Coding Specification to be included as an extended comment inside the code. In this case, special specific rules should be provided

- (4) **Integration Test Specification.**
  - Written during the **Design Phase,** this specification describes the objectives for integration testing, and spells out specific test procedures and test data for reaching those objectives.

4.2 Test Executives

- Most program systems include some form of control program, or test executive.
- (1) In **bottom-up testing**
  - A **test executive** is a modified version of the eventual **full executive program**.
  - It begins as a simplified, perhaps only a skeletal form of the ultimate program.
  - It's written early to provide a framework for integration testing.
  - Some **test executives** contain "dummy" modules or "stubs," that are gradually replaced as their "real" counterparts emerge from module test.
  - The stubs may do little more than record and print an indication that they have been invoked.
    - *For **example*** they may perform some simple operation in imitation of what the real module will do when it is eventually inserted it into the system.
  - As stubs are replaced by real modules, the program system begins to take shape.
- (2) In **top-down testing**, the test executive is essentially **the code for the higher-level ("system") modules**.
- Some test executives contain special test aids that are removed during the final stages of integration test.
  - (1) One such test aid may be a trace program to keep track of sequences of events within the system for later analysis.
  - (2) Another aid is a program to provide displays or "snapshots," of key computer registers or storage areas at strategic times.
    - This kind of test aids are called Debuggers and they usually are part of the Program Developing Tools.
  - *An **example** of a test executive used in the early stages of development of "real-time" systems is a non-real-time version of the system's control program. Similarly, a single-processor control program is often written prior to development of a full multiprocessing capability.*
- A test executive can be complex, but as a rule it should be kept simple for two reasons:

- (1) First, its value lies partly in having it ready early, before the "real" executive program is finished. If you put too much into the test executive, it won't be ready much sooner than the real one.
- (2) Second, the more complex you make the executive, the tougher it will be to separate its problems, or bugs, from those of the modules that you are trying to test.

4.3 Environment Simulators

- An environment simulator is a program that temporarily, for testing purposes, replaces some part of the world with which your program system must eventually interface.
  - *For **example**, suppose you are writing a program system for directing air traffic. One of the pieces of equipment with which your programs must eventually communicate is a radar set. But because the radar is still being developed, or because it's not yet feasible to hook up to it, you may need to develop programs that "look" like a radar. These special programs would attempt to simulate both the radar's inputs to your operational program system and the radar's responses to your outputs.*
- Other simulation programs might replace special display consoles still under development.
  - Still others might be used to feed your system sets of data representing real-world operational conditions; in our example operational conditions might be traffic loads or weather information.
- To be most effective, an environment simulator must be transparent to the using programs, that is, your programs should require little or no special modification in order to communicate with the simulators.
  - Your programs should think that they are dealing with the real world.
  - Any change you make to allow your program system to run with the simulator lessens your confidence that the system will run properly when the real thing is substituted for the simulator.
- Depending on the size and nature of your job, your **environment simulation programs** may:
  - Operate in the same computer as do your operational programs.
  - Run in a separate computer that interfaces with yours.
- The cost of developing **environmental simulators** can be enormous.
  - For example, in the air traffic control job suggested above, the cost of the simulators could easily be greater than that of the operational programs themselves.
- The need for these tools must be addressed in the **Definition** and **Design Phases**.
  - The simulators require the same care in design and programming as do the operational programs.

4.4 Specialized Programming Environments

- In ours day, developing organizations use as developing tools Specialized Programming Environments.
- Such developing tools are specialized on different programming languages and provide the necessary features for code development in form of integrated editors, compilers, debuggers, code optimizations aids, etc.
  - *For **example**: Developer Studio (Microsoft),Visual Studio, JavaBuilder, Borland C and Pascal, Rational Rose, Development Kits for different microcontrollers, different SDK's (Software Development Kits), etc.*

4.5 Automated Documentation Aids

- You'll need to find out what automated aids, if any, are available in your company or from other companies.
- There are a good many systems which purpose to automate the drawing and updating of flow charts, for example or documenting the code.
- If you consider using an automated system, make your decision early.
  - It's expensive, wasteful, and aggravating to decide halfway through a project that you're going to switch to an automated system.
- There are a lot of such automated documentation tools offered by different SW companies (Doxigen, JavaDoc, SoDA, etc).

4.6 Software and Hardware Monitors

- A **software monitor** is code added to a system for the purpose of inspecting and gathering data at key points during execution.
  - *A highly successful **example** is the Statistics Gathering System (SGS) used years ago in the Apollo program at Houston. Data gathered by SGS during operation of the Apollo system showed the developers a great deal about the execution times and frequency of operation of key modules. These data were also used to calibrate and improve the accuracy of the Apollo simulation models.*
- A software monitor might be a large effort, as in the case of SGS, or it might amount to inserting a few small routines at critical points in your program system to take counts of module usage.
- Some manufacturers offer for rent or sale hardware devices that may be attached to a computer to make certain measurements.
- Most of these **monitors** are intended to give you data to help determine how efficiently your system is utilizing various input-output channels or how your system's workload is divided between computation and input-output activities.

4.7 The Project Library

- A library is an organized collection of information.

- Your project's library should consist of two sections:
    - (1) The General Library.
    - (2) The Development Support Library.

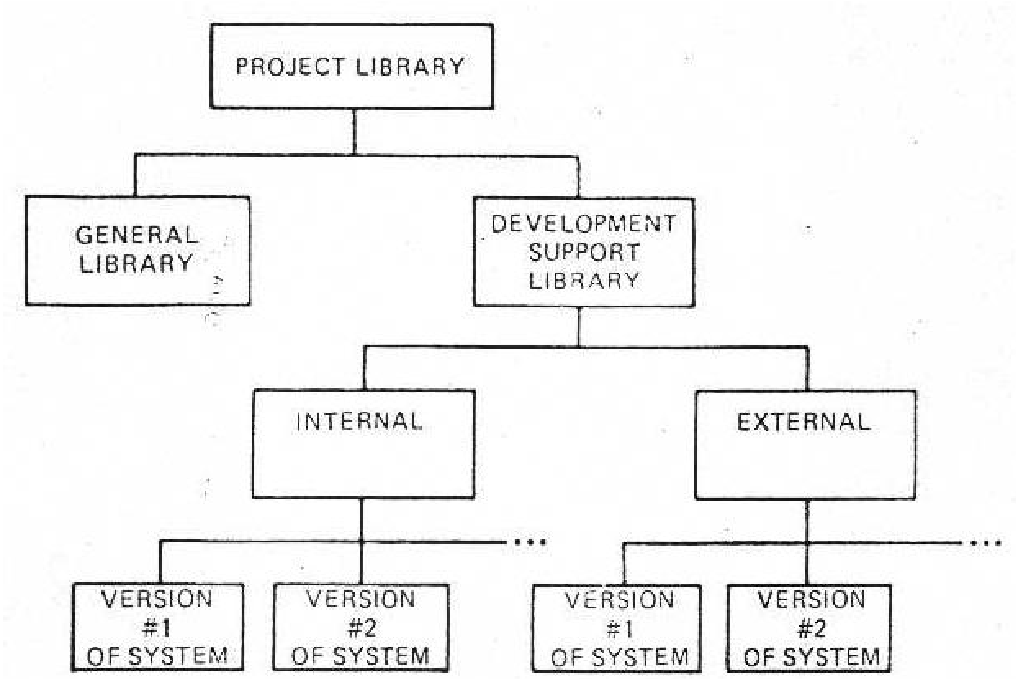

Fig.4.7.a. Project Library

4.7.1 General Library

- Keep **master copies** or **master files** of all project documents in this section, other than those in the Development Support Library.
- A basic list of documents to include:
    - Project Plan.
    - Problem Specification.
    - Design Specification.
    - Test Specifications.
    - Technical Notes.
    - Administrative Notes.
    - Change Documents.
    - Test Reports.
    - Status Reports.
    - Project History.
    - Forms.

  - Documentation Index.

- In addition, keep a **copy** of the modules and documentation for any previous version of the system you have completed and delivered.
- Every document in the library should be given a unique identifying number by the librarian.
  - The librarian should have quick access to reproduction equipment and be able to run a copy of a master document when requested.
- He should never let the master copy out of his hands.
- He may keep on hand a number of copies of often-requested documents rather than run them off only on request.
- The librarian should keep a log of document numbers so that when any project member is ready to issue a new document in some category, for instance, a Design Change Notice, he need only call the library to get a unique document number.
- Periodically, the librarian should update the Documentation Index and send out or to publish on Intranet the new version.
  - This is a listing of all documents currently in the library.
  - The listing should show document titles, authors, dates of issue, and identification numbers.
- The librarian should also periodically gather vital records and store them as **back-ups** in a facility physically separate from your facility.
  - Vital records are whatever materials PM decide are necessary in order to reconstruct the system if a fire or other catastrophe were to wipe out.
  - Vital records might include a CD (DVD) copy of your program system as it stands at some instant in time, along with a copy of the specifications describing the system at that time.
  - It costs relatively little to do this job, and it can save you a lot.
- **Metzger** gives an ***example***. *One project during the 1960s had made no such provisions. The programs were being developed in a "fire-proof" building at an Air Force base in Florida. The system was nearly completed. One night the building was gutted by fire and practically everything was lost — cards, tapes, listings, the works. And, of course, no other copy of the system existed. The story has a happy ending because the programmers had enough bootlegged listings in their homes to piece the system together again. Contract saved, payment made.*

4.7.2 Development Support Library

- Included among the important innovations and improvements in the programming process is the **Development Support Library** (DSL) (also called by other names, including Program Support Library or Programming Production Library).
- The DSL is the project's central storage place for the official version of the developing program system. It consists of two sections, internal and external, and procedures governing their use.
- (1) The internal library contains, on disk, tape, CD's or other computer storage:

- (a) The programs being developed and data relating to their development.
- (b) Exactly what is stored depends on the nature of the project and especially the computer system being used,
- (c) Typically the internal library would include the most current source code and object code for all modules in the developing system, test data, control language statements, and so on.

- (2) The external library consists of:
  - (a) Listings corresponding to the current status of each type of data stored in the internal library, and run notebooks showing the output results of test runs.
  - (b) The external library also contains archives documents — older versions of the current-status listings to be used for historical purposes and as backups in case of loss of current documents.
- (3) Procedures.
  - (a) The Development Support Library serves as the single location for the official version of the program system being developed.
  - (b) It should be supported by specialized tools for Configuration Management (CVS, Continuous, ClearCase, etc)
  - (c) It virtually eliminates the retention of private versions of modules by individual programmers, and it makes the current system completely visible and open to inspection by all project members.
  - (d) All submissions of new code, changes in existing code, requests for test runs, and so on, are submitted through the technical librarian, someone specially trained for the job.
  - (e) This librarian is the interface between the programmer and the project; he enters all new inputs into the system, and distributes all new outputs to the appropriate notebooks and binders.
- A given Development Support Library may contain more than one program system at a time because:
  - There may be versions of a system at different levels of completion at any one time, especially on larger projects;
  - The library may serve a larger community than just your project. There is no reason why a number of projects may not make use of the same library facility. In this case, more than one technical librarian may be needed.
- Every set of data, whether program code, control code, or test data, is uniquely identified within the library.
- Unique identifiers are used to separate different versions of a program from one another and entirely different projects from one another.
- Of course, it's important that the technical librarian be well trained and capable: his job is an important one.
- It's important, too, that the technical librarian to be firm and not easily bribed.
  - Programmers should not be allowed to skirt the library procedures; otherwise, the library immediately loses its value as a **vital project control point.**

- Programmers are infamous for putting in those little last-minute changes. There are no little changes. Each change is potential dynamite, especially if you're nearing system or acceptance test.
- Guard against those midnight patches by making internal library storage virtually inaccessible.

- Your programmers may be irked, but that's better than having an acceptance test blow up in your face while the customer is looking on.