Chapter 5.

THE DESIGN PHASE

Summary

5 The Design Phase

- By now **management** is getting a little edgy.
- Weeks, perhaps months, have passed since the contract was signed, and there is still no **sign** of **programs** (in fact little sign of programmers).
- It's time to show the boss that everything is right on target:
    - (1) Analysis is complete.
    - (2) A Project Plan has been written.
    - (3) The design team has been recruited and is hard at work.
- In short, you've met your milestones and are taking dead aim at the next set of **objectives**:
    - (1) **Designing** the system.
    - (2) **Refining** the Project Plan.
    - (3) **Conducting** a comprehensive review of the entire project before beginning programming.

1 Designing the System

- A key output of the **Definition Phase** was the Problem Specification, which defines the job to be done.
- The next important document to write is the **Design Specification**.
    - Design Specification is the blueprint for the program system.
    - It is the starting point for the programmers.
- It's hard to emphasize enough the importance of *having* this document and making it the **focal point** of the programmers' activity.
    - It's a *ruinous game* played by managers who don't understand that you must design before you code.
    - Such managers confuse motion with progress. *(Larry Constantine)*
- The Design Specification states the **solution** to the customer's problem.
    - (1) It is a *solution* chosen by project management from among alternatives offered by the design team.
    - (2) The design chosen must be the "*best*" one for the project.
    - (3) It may not be the best in terms of elegance, nor the one chosen if unlimited resources were available, but it must be the **best** that can be implemented, given the **constraints**, on **available time**, **talent**, **equipment**, and **money**.

    - (4) Whatever design is chosen, it must, of course, completely satisfy the requirements stated in the Problem Specification.

1.1 The Design Specification

- The Design Specification describes an *acceptable programming solution* to the problem stated in the Problem Specification.
    - The Design Specification is the **baseline** for all future detailed design and coding.
- A good design specification shows the solution in two ways: in terms of **function** and **logic**.
    - (1) The *functional description* shows what the system is to do.
    - (2) The *logic description* shows how the system is actually structured to provide those functions.
        - ***For example**: A functional description may include a box containing the words "Calculate Trajectory," but it is left to the logic descriptions to describe the method or procedures to be programmed to do the actual calculations.*
- Management and the designers must select the *appropriate tools* to use in communicating both **function** and **logic**.
    - (1) One method is to use ***HIPO charts*** to describe function and ***flow charts*** to describe logic.
    - (2) Another, more in keeping with trends in structured programming, is to use ***HIPO*** for function and ***structured charts*** or ***pseudo code*** for logic.
    - (3) In the last time, **UML diagrams** are more and more used as support for architecture design. These tools are described later.
- The **design documentation** is the result of investigation and reasoned decisions.
    - Don't just let something happen because of inaction.
    - Don't let one designer use one method and another one a different method.
    - Think about how your baseline documents *will interface* with what preceded them (the Problem Specification) and what will follow them (the *detailed program descriptions*).
- A technique such as HIPO, for **example**, can be used by the analysts in writing the Problem Specification, thus smoothing the transition between that document and the Design Specification.
    - In fact, the *functional* section of the Design Specification may look very similar to the Problem Specification.

- HIPO can, in turn, be a major part of the detailed documentation for the individual programs, thus extending the feeling of **consistency** and **continuity** in the project's technical documentation.
- The ***general content*** of the **Design Specification** consists of (fig. 5.1.1.a):
    - *(1) The overall design concept*
    - *(2) Standards and convention*
    - *(3) The program design*
    - *(4) The file design*
    - *(5) The data flow*

DESIGN SPECIFICATION (TEMPLATE)

DEPARTMENT:

PROJECT:

DOCUMENT NUMBER:

APPROVALS:

DATE OF ISSUE:

REALISED:

SECTION 1**: SCOPE**

This document defines a solution to the problem described in the Problem Specification. The Design Specification is the foundation for all program implementation. The design logic described here is detailed enough so that all requirements functions are satisfied, and all interfaces, system files, and the logic connecting all program modules are defined. The design is done in sufficient detail that all system logic problems are resolved and the complete program system "hangs together". The lowest level of program module is specified in terms of the functions it must perform and the interfaces it must have with other modules, but the actual internal design of these lowest-level modules is left to the implementing programmers.

If the project is to produce more than one program system, for example, support programs in addition to operational programs, there will be more than one Design Specification.

SECTION 2: **APPLICABLE DOCUMENTS**

SECTION 3: **OVERALL DESIGN CONCEPT**

This is an overview of the entire program system hierarchy.

3.1. **Program Hierarchy**

Definition and description of the program system hierarchy.

3.2. **Data Hierarchy**

Definition and description of the system files and their interrelationships, including simple pictures of file structures.

3.3. **Standards and Conventions**

3.3.1. **Design Standards and Conventions**

Definition of all standards and conventions adopted for use in this design document and to be observed during later detailed design.

3.3.1.1. UML Standards

3.3.1.2. Naming Standards

3.3.1.3. Interfacing Standards

3.3.1.4. Message Formats

3.3.2. **Coding Standards and Conventions**

Definition of all standards and conventions to be observed during coding.

3.3.2.1. Languages

3.3.2.2. Prohibited Coding Practices

3.3.2.3. Required Coding Practices

3.3.2.4. Recommended Coding Practices

SECTION 4: **THE BASELINE DESIGN**

This is the focal point of this document. All program and system file logic is presented here to the level of detail the designers feel is necessary before turning the document over to the programmers for implementation.

4.1. **Program Design**

4.2. **File Design**

Pictorial layouts of all system files describing all subdivisions of the files and characters. Also, a complete description of the relationships among the various files, including pointers used to link files and coverage matrices showing which programs access each file.

SECTION 5: **DATA FLOW**

This section uses flow diagrams and accompanying narrative to describe the major transactions in the system, irrespective of the actual logic structure of the system. The intent is to provide an understanding of data paths and major events in the operational system, including all subsystems, hardware as well as software. This exposition is useful as an introduction to the system and should not presume programming knowledge on the part of the reader.

Fig. 5.1.1.a. Design Specification template

- **(1) Overall design concept.**
    - This is a brief combination of *narrative* and *diagrams* providing an overview of the entire program system design at a high level.
- **(2) Standards and conventions.**
    - This section states the **rules** adopted for use in describing both the baseline design and the detailed design to be done later by the programmers.
    - It covers such items as:
        - Flow-charting.
        - HIPO standards.
        - UML standards.
        - Naming standards.
        - Interfacing conventions.
        - Message formats.
    - This section also includes:
        - Coding standards and conventions to be observed during the **Programming Phase**.

        - Prohibited, required, and recommended coding practices.
            - If such standards already are published for the organization, a simple reference to those standards here will suffice.

- **(3) Program design.**
    - This is the core of the document.
    - Through a combination of diagrams, narrative and tabular information, it describes the program system first functionally, and then in terms of functionalities, its actual structure, or logic.
    - It begins with *a look at the overall hierarchy* and then breaks the system into *smaller chunks* for a closer look, on a **WBS** basis.
        - The level of detail must be such that **no** major design problems are left to the **Programming Phase**.
    - But this baseline design should **not** be carried to the ultimate level of detail. There are two **reasons** for this:
        - (1) First, detailed design would make this a massive document, and you would find it impossible to apply effective change control.
            - *Change control* must focus on the structure of the system at a high enough level.
            - So, a change in the way a low-level module is coded is **not** subject to formal control.
        - (2) Second, don't make the programmer a robot who codes someone else's design.
            - The individual programmer should be your expert at coming up with a solution (detailed design and code) that best handles a specific problem.
    - **Very important:**
        - (1) Insist on building a **solid framework** for both your *programs* and *data files.*
        - (2) Let the **individual programmer** concentrate on devising *the best possible code* to fit within that **framework**.

- **(4) File design.**
    - This is the companion to the program design section.
    - It defines in detail all *system files* also called **system data sets**.
        - These are files which are accessed by more than one program module.
        - Thorough definition of these files will help you avoid a lot of problems later.

    - Many projects have floundered because individual programmers independently designed files and later found that other programmers had had different file designs in mind and had written their programs accordingly.
        - ***Metzger's story:*** *In one memorable case, two teams developed major program subsystems, each based on its concept of what the system files were to be like. After more than a year's work they discovered that the two subsystems were miserably incompatible because the files each assumed were worlds apart. One subsystem was scrapped; so was its manager. Avoiding such gross lack of communication is one of management's most urgent responsibilities.*
- **(5) Data flow.**
    - This is a kind of *executive summary* of the design, understandable by non-technical upper management.
        - There will be more details on this later in the chapter.

1.2 The Designers

- The **designers**:
    - Must be, above all else, *expert programmers.*
    - At least some of them should have been heavily *involved* in the problem analysis activity.
    - Ideally, some of the designers should be earmarked as *lead programmers* or *managers* during the **Programming Phase** in order to provide as much continuity as possible.
- **Good designers**:
    - (1) Can go quickly to the *heart of a problem* and not get trapped into wandering down all the little dark alleys.
    - (2) Must be *practical*.
    - (3) Must know what can *reasonably be expected* of the various components of the overall system: the machines, programs, people.
    - (4) And above all, they must *communicate*.
        - There are some superb technical programmers worth ten run-of-the-mill programmers as far as technical ability is concerned, but they can't communicate.
        - They are content and extremely productive if you carve out a big chunk of the system, define its interfaces with the rest of the system, and turn them loose.
        - These people may be more useful in the **Programming Phase** than during the **Design Phase**.

- The manager must know the *technical leanings* of the **designers**.
    - *Strong biases* could easily prevent sound design tradeoffs from being evaluated.
        - A lead designer who always tilts toward assembler language, for instance, may never give a high-level language a fair shake.
    - *Biases* are inevitable, but if you know they're there, you can probably keep them from killing you.

1.3 The Design Environment

- The designer requires *special conditions* to work.
    - If you, the manager are an orderly person who can't stand arguments and conflict, lock up your designers and leave the room.
    - If there's a place on a project for a little chaos, it's probably here.
- Don't bother the designers with constant *interruptions*, *unnecessary meetings*, or *chores* not bearing directly on their design job.
- Just be sure that the design effort is led by someone who has the spine and the technical competence to *resolve conflicts* (sometimes arbitrarily in order to get on with the job).

1.4 Design Guidelines

- Designing a program system is not so mysterious as some designers would have you believe.
    - There are some perfectly reasonable *guidelines* that apply to the *design* of any program system, and, for that matter, any other kind of system.
- Good designers will observe these guidelines almost automatically.
- (1) **Conceptual integrity.**
    - When designers are turned loose to design a new system, usually they should be full of novel ideas and clever techniques and will have an almost *irresistible urge* to incorporate them into the system.
        - But if they are really good, they'll resist that urge.
    - A well-designed system would best start with a *very small number of people* so that a ***single philosophy*** prevails.
        - A good system design is not a lot of **lovely limbs** stuck together to make a **tree**; rather, it's a *strong trunk* which supports graceful limbs.
    - It's up to *chief designer* to assure the *integrity* of the system, the *uniformity* of conception.

    - He or she must keep the design squarely aimed at the system's requirements.
  - Brooks [37] contends that "***conceptual integrity*** *is the most important consideration in system design*."
    - It is better to have a system omit certain anomalous features and improvements, but to reflect **one set of design ideas**, than to have one that contains many good but independent and uncoordinated ideas."
- (2) **Modularity.**
  - **Metzger's story**: *"When I was a boy, I dug a lot of ditches. I went about that inspiring job in a very methodical way. First I outlined where the ditch was to go, broke loose a neat two or three cubic feet of earth with the pickaxe, and then shoveled away the loose earth cleanly so that I could now see and attack the next chunk of ditch. I always felt that I could see my progress better that way — and that little game probably helped me maintain a degree of sanity. The alternative was to whack away with the pickaxe for a longer time, piling up a strip of more or less loosened earth, and then shovel for a long time; but that way there would be fewer times when I could look back and clearly see progress."*
    - A programming job is like a *ditch*. It has a beginning and (sometimes) an end.
      - You can attack it *methodically*, always having a good feel for where you are.
      - Or you can *lurch forward* with no intermediate goal in mind except to bull your way to the end.
    - Either way, you'll strike rocks.
      - Mr. Neat, however, will be able to clean around the rock, see it, pry it loose, or bypass it.
      - Mr. Bull's rock will be obscured by all that loosened earth.
  - The **designers** should *lay out* the program system in **chunks**, or **modules**, not only to aid in the design process itself, but to give a big assist to the rest of the project.
  - **Modularity** means *subdividing a job into compartments*. That has many advantages. We list them at the risk of stating the obvious:
    - (1) Modularity provides *visibility*.
      - A system quickly gets so big and complicated that it's difficult to see what's going on, unless we can look at it and understand it one piece at a time.
    - (2) Modules force *simplicity*, and as a result, *less error*.
    - (3) Modules are very often *reusable*.

    - The more restricted is the function assigned to a module, the higher the probability it can be used elsewhere in this or some other program system.
  - (4) Modules are a *convenient basis* for *assigning work* to the programmers.
  - (5) Modules are *handy building blocks* that can be *put together* in a very deliberate, controlled manner during testing, whether you are working top-down or bottom-up.
  - (6) Modules provide a *convenient basis* for progress-reporting and statistics-keeping.
  - (7) Modularity makes *later changes* easier to effect.
- *Module,* as used in this course, is a general term applying to a *clearly identified portion* of the program system at any level in the hierarchy.
- The fig. 5.1.4.a shows a **general hierarchy** of modules comprising a program system.
  - The **modules** at each level are named as follows: *unit, CSCI, component, package, subsystem, system*.

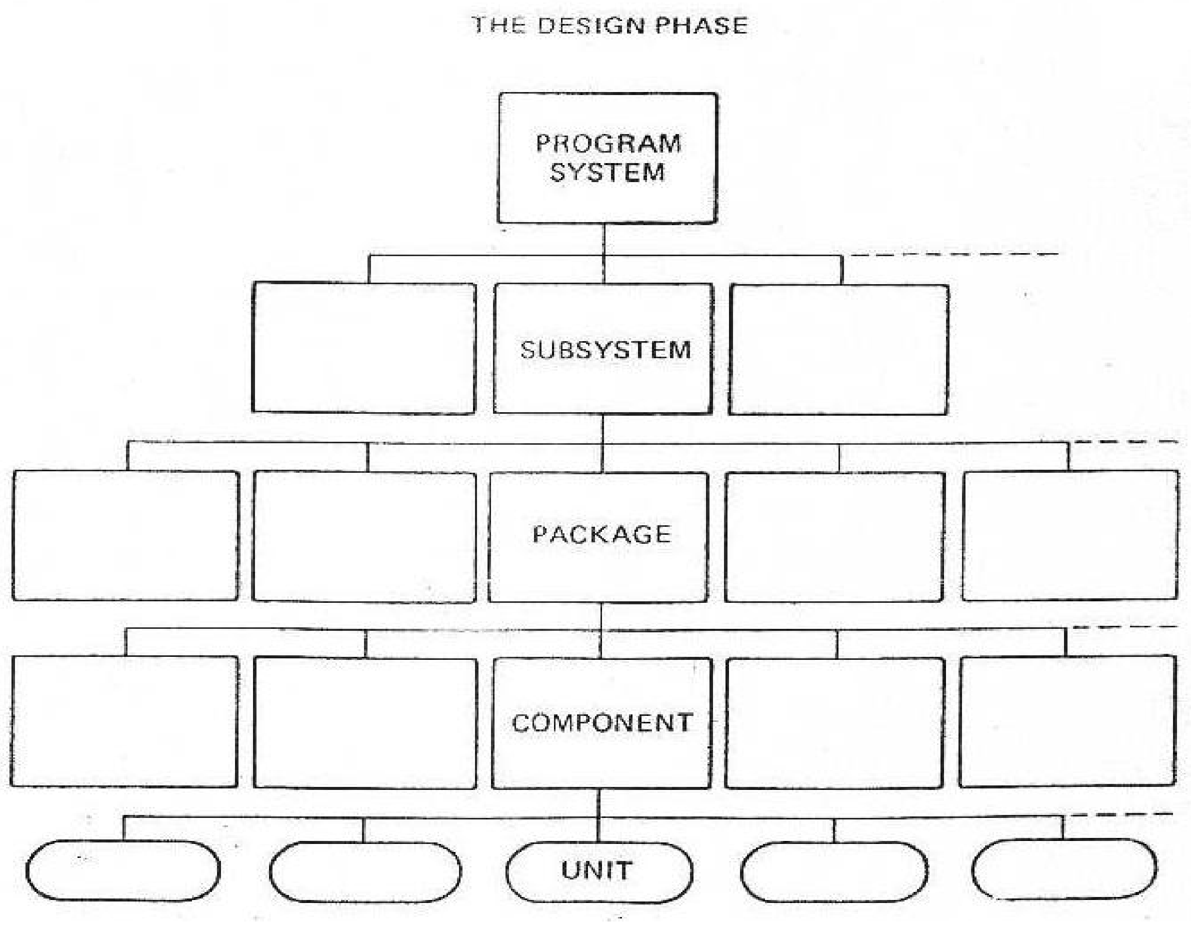


Fig.5.1.4.a. System WBS (Hierarchy)

- The diagram simply says that the **program system** is comprised of **program subsystems** which are comprised of **program packages** which are comprised of **program components** which are comprised of **program units**.
    - **Each box** in the diagram represents a **program module**.
    - You may choose to call your modules at the various levels by some other names, but whatever names you choose, use them consistently throughout the project.
    - You may also need *more* or *fewer levels*, depending on the nature of your system.
- The module called **unit** or *"SU (Software Unit)"* is the **lowest** level of module independently documented and controlled in the system.
    - A unit it's generally assigned to an *individual programmer*.
    - The programmer, in coding the unit, may break it into *smaller pieces*, such as *objects, classes, methods, routines, subroutines, macros*, or other exotic names, but when the work is *documented* it is all contained in a single tidy bundle called a *unit*.

- (3) **Interface definition.**
    - Although the **designers** must expend much energy in defining the system in terms of **modules**.
        - They must pay equal attention to *defining* and *documenting* the interfaces between modules.
    - The Design Specification should spell out exactly how the modules are to communicate.
        - Programmers writing individual units should *never* be given the *freedom to combine* their units in whatever fashion they wish.
        - This is not the place to give the individual programmer artistic license.
    - Designers have the responsibility to include in the design document **explicit** and **detailed explanations** referring of the following:
        - (1) How *modules* are to communicate with other *modules*.
        - (2) How *modules* are to communicate with data *files*.
        - (3) How data *files* are to communicate with other data *files*, including the use of "pointers" linking one file to another.
        - (4) How human *operators* are to interface with the programs.

    - (5) How the *programs* are to pass data, such as error messages, to an *operator*.
    - (6) How the *program* system is to pass information to other *program systems* or to *equipment systems* such as display devices.

- (4) **Simplicity.**
  - The apart domain of software industry encourages the **tendency** to use complicate and specific constructions.
    - If you can find a *designer* who can discard such nonsense and express his design in simple, understandable language, you've got yourself a *real professional*.
    - *"It is never unprofessional . . . to make oneself clear,"* says **Robert Gunning** [Me81].
  - As program systems are asked to satisfy *increasingly complex requirements*, computer scientists insist that program designs become simpler.
    - IBM a leader in the search for better programming and management techniques, urges us to look for the "*deep simplicities*" in program design.
  - Strive for modules so simple and explicit in their *function* and *structure* that they can be *reused* in other systems.
- (5) **Simplified module coupling.**
  - We've all seen programs in which **modules** are strongly dependent on one another because the operation of one module depends on its "knowledge" of what's happening in the other module.
    - In extreme cases, one module might alter *the contents* (sometimes instructions!) within the other.
    - Such modules are said to possess *strong coupling*, and their effect is to *complicate* the system.
  - Designers should aim for the *weakest* possible coupling — that is, the *greatest independence* — between modules.
  - **Weak coupling** makes each module:
    - (1) *Easier to treat* as an entity.
    - (2) *Easier to design* without considering its effect on other modules.
    - (3) *Easier to alter* or *to replace* later on.
- (6) **Minimum commitment.**
  - **Larry Constantine** has stressed the idea of "minimum commitment."
  - By this he means that a designer should solve detailed problems *as far down* in the system *as necessary*.

    - For ***example***:
        - When laying out the "package" level of module, don't give undue attention to a detail that belongs in a lower-level module, a "unit."
        - Only express as much as *you have to* in a design description at any given level.
        - Don't get hung up on details and lose track of the big things that are going on at that level.

- (7) **Rule of black boxes.**
    - As a corollary to "minimum commitment," Constantine advises the designer to state a required function as a "black box".
    - A **black box** presumes to define its inputs and outputs with little regard for *internal structure* until a later pass through the design.
        - **Joseph Orlicky** [6] describes the **black box** as a device "we simply postulate by defining its inputs and outputs."
        - A **black box**, he says, *"does everything we want it to do*."
    - Again, avoid getting so deeply involved with *details* that a sound basic design never shows through.
- (8) **Top-down design.**
    - If you were *designing a building*:
        - You'd start by considering the building's total environment (for example, where will it sit, on how big a piece of land, of what shape).
        - Then you might try to determine the style of building that would fit both its intended use and its surroundings, not to mention the customer's budget and his tastes.
        - After that you might be ready to sketch out the main structure, without details.
        - Given tentative agreement about building height, number of floors, general shape and style, and so on, you could proceed to designing the major parts of the structure —main entries, office areas, shop areas.
        - And then you'd get down to designing specific areas, such as offices, meeting rooms, rest rooms, shops, connecting hallways, stairwells, elevator shafts.
        - Finally, you would need to address the placement of doors, windows, lights, outlets, plumbing, decor, and a thousand other details without which the building would not function.
        - All through the process you would find that decisions about lower-level items would impact decisions already made at higher levels.

        - The very tools and materials to be used in doing the job (for example, concrete or glass) might affect, and be affected by, decisions made earlier.
    - Many iterations later, decisions would be solidified and approved (sometimes arbitrarily, in order to get on with the job).
    - And finally the design is ready for the workmen.
    - The building design did not begin by concentrating on the size of the toilets, and program designs don't begin with the layout of a housekeeping module.
- In **SW Design** is the same situation:
- **Start** with the *highest*, *grossest*, *most inclusive level of functions*, and refine it in ever smaller steps ("*stepwise refinement*") until all *functions* have been accounted for in a coherent and systematic manner.
    - This is top-down design.
- The writing of the ***baseline design*** *is the* **point of departure** *for the entire* ***detailed design****.*
    - The **Baseline Design Document** must:
        - (1) Establish *the framework* for the program system.
        - (2) Establish all the *communication conventions*.
        - (3) Solve all the *flow and control problems*.
    - The design of the *individual lower-level modules*, however, is left to *individual programmers*.
    - The **designers** *must decide where to stop* the baseline design.
        - There is no way to state exactly where the baseline design should be *stopped*.
        - That will be different for each project, and in fact will reflect the experience, even the personalities, of the designers, managers, and programmers.
    - As soon as the *baseline design* shows a complete and viable solution to the problem stated in the Problem Specification, it's finished.

- (9) **Existing programs. Reusability**
    - It's known that there are very few programmers who wouldn't rather rewrite from scratch than make use of existing programs.
        - It's very often true that rewriting is the proper course.
    - Making use of *existing code* can be a **headache** because:
        - (1) Supporting documentation is poor.
        - (2) The code is not exactly what is needed and must be modified.

        - (3) Modifying someone else’s code is not too exciting.
    - Nevertheless, your designers are derelict if they *don’t honestly* consider what exists and how it may be adapted to your system.
        - The huge *operating systems* and **libraries** of *support programs* built by computer manufacturers and some software houses cost hundreds of millions of dollars; there just *might* be something there that you can use.
        - Rather than ***build*** *a solution, you may be able to* ***buy*** *it.*
    - **Constantine**, **Stevens**, and **Meyers** [36] make a strong case for building (at least within a given organization) program modules of such *simplicity* and *independence* that they *can be used for later needs*, not just to satisfy the requirements of one system or contract.
    - The more we begin to use structured design and object oriented technologies, the more we treat modules as "black boxes," the closer we’ll come to realizing that goal.
        - How many thousands of essentially identical "edit" or "binary search" or "get" or "put" routines have been built over the years?
- (10) **Pity the user.**
    - Designers must constantly think and act with the *total system* in mind.
        - That system includes people and machines, not just programs.
    - You’re **not** designing something to show how *clever* you are.
        - You’re building something *to be used* by **human beings**.
    - Furthermore, those **human** users are usually not **computer-oriented**.
        - If the computer makes their jobs *easier*, it may be *accepted*.
            - Otherwise they’ll ignore it and continue to work in their old, comfortable ways.
            - If the computer and its complicated user’s manuals are shoved down their throats, they may even *sabotage* the entire effort.
    - **Orlicky** says on this subject: "*Coddle the user, not the computer, and remember that the primary goal is* ***not*** *the efficiency of the computer system* ***but*** *the efficiency of the business".*
- (11) **Iteration.**
    - Don’t think that the designers are finished the first time they have a flow chart that seems to include modules to cover all the required functions.
        - Good design is usually the result of much iteration.
    - Things learned while designing at the *component level* are bound to give the designer second thoughts about what he did at the *package level*.

    - This process is repeated until the changes are minor and the chief designer calls for the rubber stamp that says *done.*

1.5 Designing Tools

- Most *design tools* are more helpful in trying out or documenting a design idea than in coming up with the idea in the first place.
- This is not to discount the use of such aids; it is *simply* to indicate that conceptualizing a design remains very much a cerebral process.
- Below are some of the most often used *design tools*, almost all of which are also helpful during problem analysis.

1.5.1 Flow Charts

- A ***flow chart*** (also called a flow diagram) is a diagram that combines symbols and abbreviated narrative to describe a sequence of operations in a program system.
- The symbols include:
  - (1) *Geometric shapes* as boxes, circles, triangles, and diamonds.
  - (2) The shapes are connected by *lines* indicating direction, or sequence.
  - (3) Each *symbol* has a specific meaning:
    - A *rectangle* may signify a process of some kind,
    - A *diamond* usually is a decision point, and so on.
  - (4) A *few words* accompanying each symbol define the *operation* represented by the symbol.
- This is one of the few places in the programming business where some *degree of standardization* has been achieved.
  - The International Organization for Standardization (ISO) and the American National Standards Institute (ANSI) have adopted compatible **flow chart standards.**
- **Flow charts** were once used as the *primary form of documentation* for almost all programs.
  - Useful as they are, they have a rather spotty reputation.
  - They may be **much more helpful** as a *programmer's private thinking tool* **than as** *a formal and final means of documenting programs.*
  - Programmers are notoriously lax in keeping flow charts *updated* and in *step* with the code.
- More fruitful may be *the use of other tools* easier to update (for example, HIPO charts) and more reliance on the *code itself* to document the program.

- The use of **structured code**, described later, is the next step in this direction.
- Sometimes there are pressures which dictate the continued use of flow charts even if you feel that some other tool is better.
    - One such pressure may be *customer insistence*, because he is used to flow charts and gets nervous when someone rocks the boat by suggesting a change.
    - In that case, you have an education job ahead of you.

1.5.2 HIPO

- Traditionally, designers have laid out proposed programs using various combinations of *tables*, *flow charts*, and *narrative*.
- The ***flow charts*** have very often been a cross between *functional diagrams and logic diagrams.*
    - That is, they sometimes describe in the same set of diagrams:
        - (1) **The functions** the programs are to perform *(the what).*
        - (2) **The structure or logic** of the programs *(the how).*
- What is needed during the **Design Phase** is:
    - (1) **First** a concentration on *functions.*
    - (2) **Then** enough expression of *logic* to assure the builders and managers that all **functions** are accounted for completely.
- There is a reasonable way to build a **set of programs** to provide those **functions**.
- To satisfy the *functional* design, a documentation method called **HIPO** has been devised and is enjoying acceptance among designers, managers, and programmers.
    - The description of *logic* is to a very great **extent** implicit in the **HIPO** charts, but it is left to other forms of documentation to describe logic fully.
- **HIPO** stands for **H**ierarchy plus **I**nput-**P**rocess-**O**utput, and represents a **design oriented** WBS technology. It consists of:
    - (1) A set of diagrams which show the functional breakdown of a program system (or any system) in the form of traditional **hierarchy charts**.
    - (2) Separate diagrams which explode each box on the hierarchy chart into a set of three boxes showing inputs, processes, and outputs.
- The figure 5.1.5.2.a shows one way of functionally representing a program using HIPO.

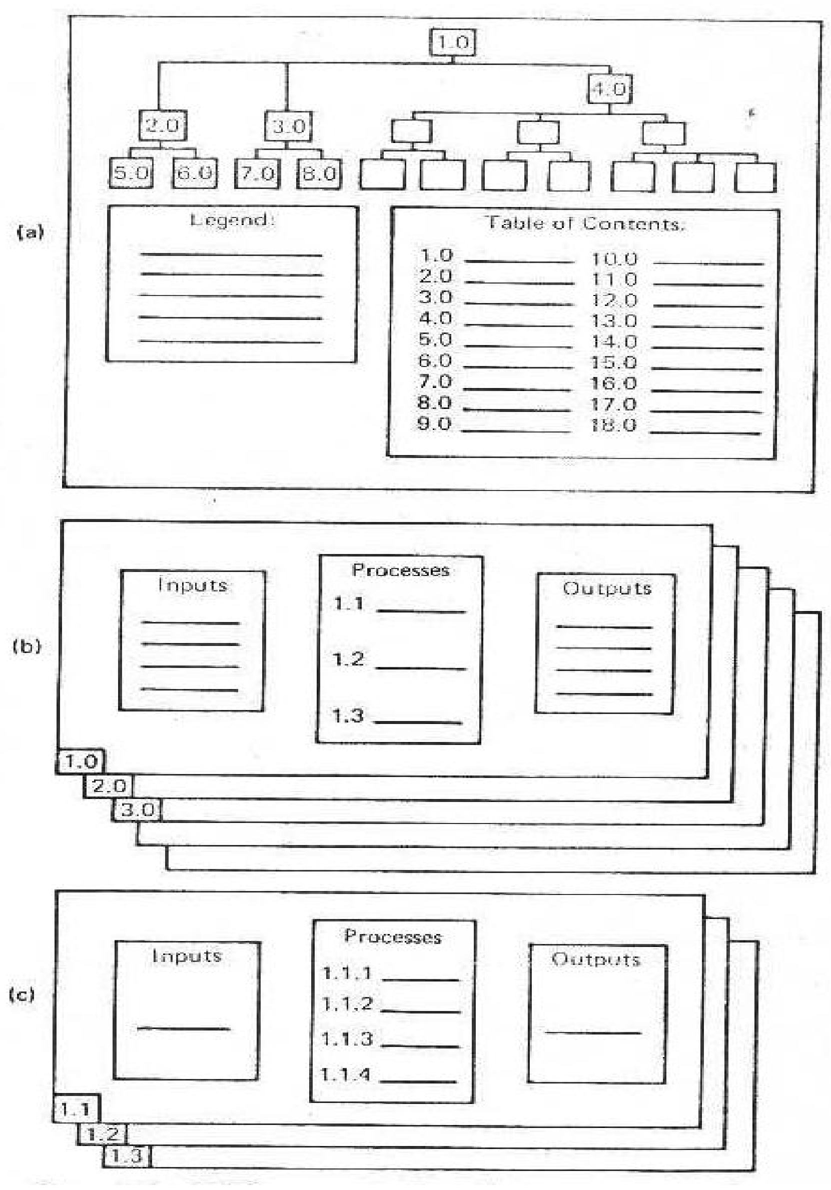


Fig. 5.1.5.2.a. HIPO diagrams

- Each organization using the HIPO idea may use *slightly different formats*, but the **basics** remain the same.
- For **every box** shown in the overview there is a **separate chart** showing:
  - (1) The inputs to that box.
  - (2) The processes (or "functions" or "transformations") the box is to perform.
  - (3) The outputs of the box.

- The scheme may be extended to *any level of detail* required to account for all functions.
    - Normally, each box in the **overview** (Fig.5.1.5.2.a (a)) will be shown in *separate charts* (Fig.5.1.5.2.a (b)) as indicated in slide.
    - Any or all of those charts, in turn, may be further exploded, as in (Fig.5.1.5.2.a (c)), until all functions are accounted for.
- HIPO can be used effectively by other than the baseline designers.
    - It can be used by the **analysts** to help *express the **requirements*** of the system.
    - HIPO could well serve as the *analysts' primary documentation*.
    - Using HIPO to express the **system's requirements** would ease the way for their use in describing the system's functional design.
    - In fact, there could be a *close correlation*, visually and in substance, between **HIPO *requirements* charts** and corresponding **HIPO *functional design* charts**.
    - Furthermore, HIPO charts can be used as *basic program documentation*.
    - When combined with structured code or pseudo code, HIPO charts can eliminate the need for flow charts as a deliverable item to the customer.

1.5.3 Pseudocode

- As mentioned earlier, *flow charts* are not always a suitable vehicle for either developing or documenting design.
- With the increased use of *structured code for programming*, a tool called **pseudocode**, also called Program Design Language (**PDL**) is used in many installations in place of flow charts to describe design logic.
- **Pseudocode** is a *notation* similar in look, form, and meaning to both spoken language and programming languages; it's a **bridge** between the two.
- There is no single pseudocode.
    - Rather, there are a *number of schemes* used in different organizations, tailored to both the needs of that organization and the coding language(s) used.
- In fact, pseudocode can take *whatever form* an individual designer or programmer wishes; that, however, is certainly to be avoided like plague.
- Pseudocode must be **standardized** *at least* throughout a given project, if not throughout an installation or company.
    - If it is not standardized it will of course lose its value as a crisp means of communication.

1.5.4 Structured Charts

- ***Structured charts*** are a pictorial way of expressing program *logic*, or *structure*, in a rigorous way, readable from top to bottom.
    - These charts are based on the **restricted** *set of conventions* used in structured programming.
- There are a number of different schemes for drawing such charts.
    - Figure 5.1.5.4.a (a) shows a set of conventions for representing each type of **program structure**.
    - Figure 5.1.5.4.a (b) shows a portion of an actual structured chart where the program is to make a **series of choices**.
- Charts like those shown in these figures are often called **Nassi**/**Shneiderman** or **Chapin** charts, after their authors.

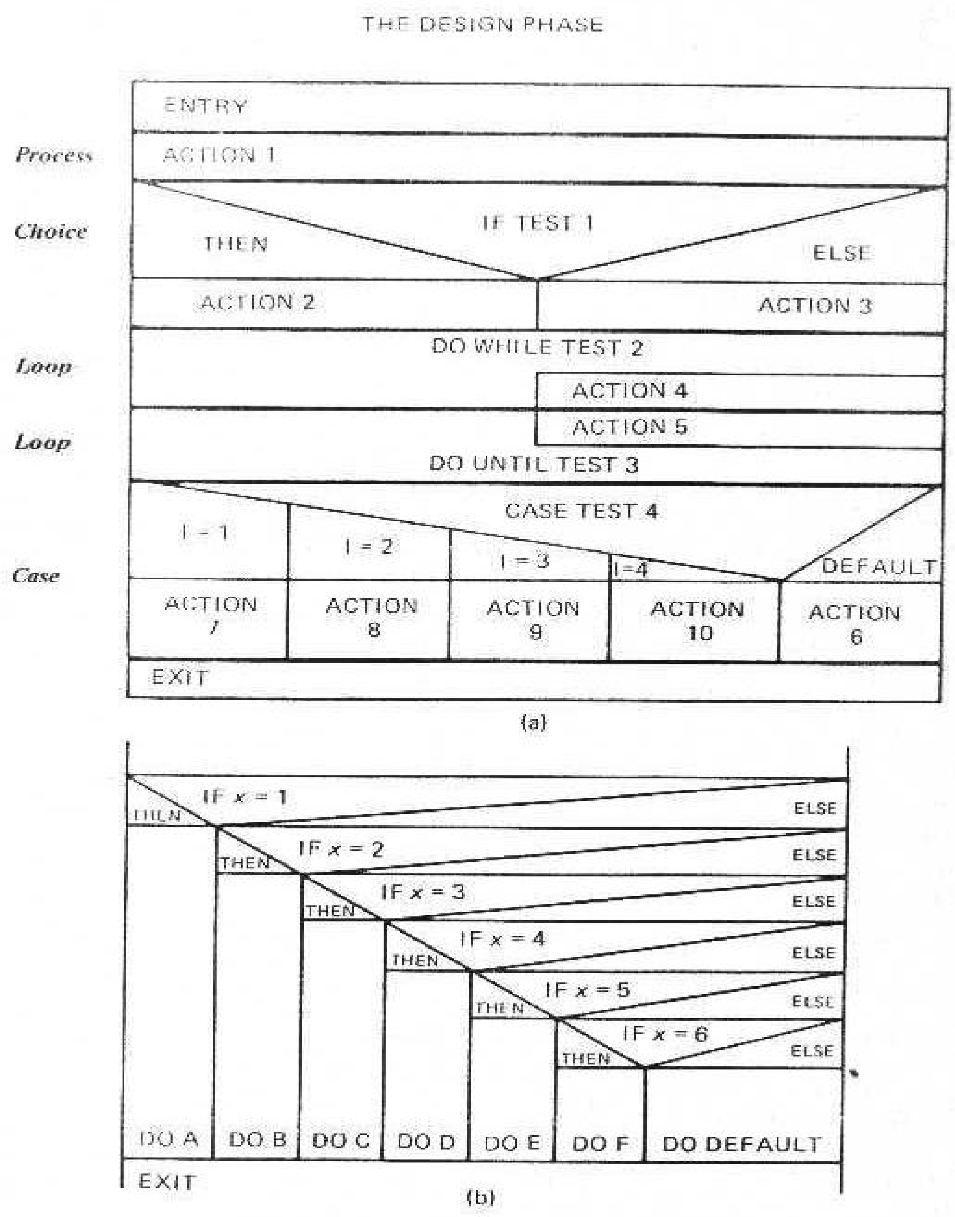


Fig. 5.1.5.4.a. Structured Chart

1.5.5 Data Flow Diagrams

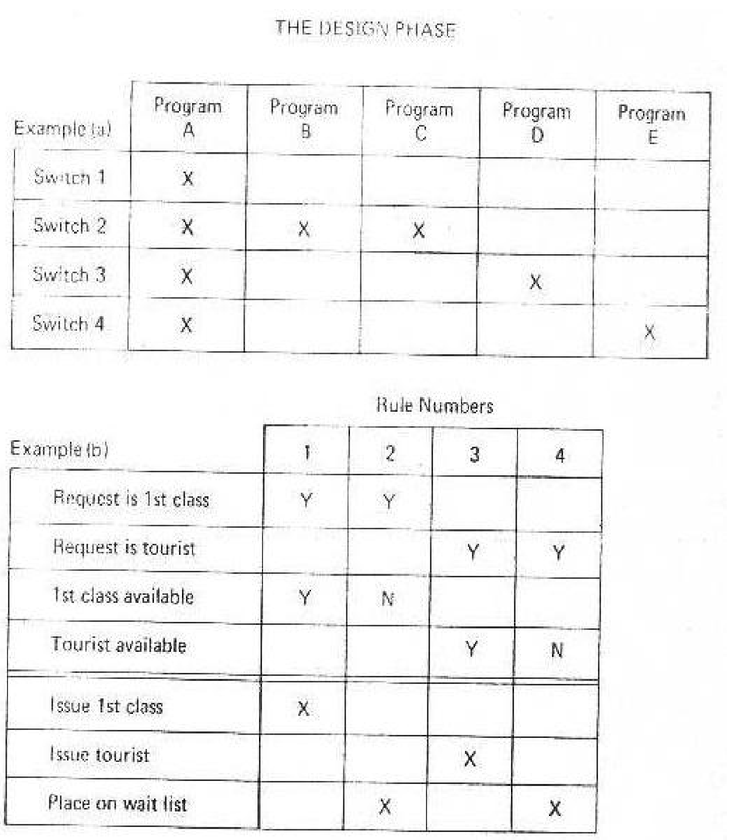
- ***Data flow diagrams*** are a special form of flow chart, useful to both analysts and designers.
    - Its intent, like that of any flow chart, is to use symbols and text to describe a *sequence of operations*,

- But there is a significant difference: a data flow diagram describes the **system functionally** but has little or no regard for the actual system ***structure***.
    - *For **example**, in a message processing system it may be very useful to depict what happens to a message as it passes through the system: acceptance and decoding of the message; performing error checks and error corrections; extracting relevant data from the message; filing the data; updating displays affected by the message content; printing summary reports of all messages received; and eventually purging the message data from the system.*
    - *These operations may be performed by several different sets of programs over a long period of time (for instance, days), with frequent involvement by human operators.*
- Ordinary flow charts or other means of describing the programs involved may not get across a clear idea of what's happening in the system unless the reader is already very familiar with it.
- A ***data flow diagram*** can help illuminate the process:
    - (1) By showing *events* **rather** than *programs.*
    - (2) By tracing the *wanderings of a major chunk of data* **rather** than by *showing program logic*.
- Here is a **convenient format** for a data flow diagram:
    - (1) Divide a page down the middle and show connected *flow symbols* down the **right half** of the page with the *running text, or commentary*, down the **left half**.
    - (2) **Symbols** and their **explanatory text** should always be kept on the same page.
        - In this case, the text is not the abbreviated kind usually found in flow charts.
        - Instead, it's plain English commentary describing what's happening in the diagram on the right hand side of the page.
    - (3) This flow description is particularly useful to people whose main interest is in overviews.
- The symbols you use in a data-flow diagram may be quite different from those used to describe program logic.
    - *For **example**, if your system involves spaceships, radars, and display devices, you could use symbols that represent those devices in order to make it easier for the reader to understand the diagram.*

1.5.6 Decision Table

- A Decision Table is a simple, convenient way of summarizing a number of "if-then" situations.

- A Decision Table shows at a glance what **action** is to be taken if a given **condition** or **set of conditions** exists (fig. 5.1.5.6.a.).



THE DESIGN PHASE

Example (a)	Program A	Program B	Program C	Program D	Program E
Switch 1	X				
Switch 2	X	X	X		
Switch 3	X			X	
Switch 4	X				X

Example (b)	Rule Numbers			
	1	2	3	4
Request is 1st class	Y	Y		
Request is tourist			Y	Y
1st class available	Y	N		
Tourist available			Y	N
Issue 1st class	X			
Issue tourist			X	
Place on wait list		X		X

Fig.5.1.5.6.a. Decision tables

  - ***Example*** *(a) in Figure 5.1.5.6.a shows which program (named A, B, C, D, and E) or combination of programs must be called and executed in response to any one of four switch actions by an operator. If switch 1 is on, only program A is executed; if switch 2 is on, programs A, B, and C are executed; and so on.*

- The table doesn't necessarily say anything about combinations of switches, but it could easily be constructed to do so. The example is very simple, but it shows some vital information at a glance.

- The **greater** the number of switches and the **greater** the number of programs, the more useful the table becomes.

  - *In* ***example*** *(b) in Figure 5.1.5.6.a a somewhat different use of a decision table is shown. Here programs are being used to determine what kind of ticket to issue to an airlines passenger. In the table, Y means yes, N means no, and X means action. Again the example is simple, but consider how this table might be expanded to cover a more complex set of circumstances. A line* *entitled, "Is alternate acceptable?" might be added. Then, if first class is requested, but unavailable, and tourist is available, the action would be to issue a tourist ticket rather than put the passenger on a waiting list.*

- There is no end to the number of practical uses of such tables because they can be designed to show in one place all possible decisions for a given set of conditions.
    - Ordinary flow charts containing the same logic might cover many sheets of paper and would be much more difficult to read and understand
- In addition, decision tables are easy to check for **completeness** by inspecting rows versus columns.
    - It's much more difficult to look at a flow chart and see whether or not all required combinations of conditions have been accounted for.

1.5.7 UML

- (1) **Definition**
    - The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of *software systems*, as well as for *business modeling* and other *non-software systems*.
    - The UML represents a collection of the **best engineering practices** that have proven successful in the modeling of large and complex systems.
    - The concept of Artifact plays a central role in UML philosophy.
    - The *Primary Artifacts of the UML* are the UML definition itself and how it is used to produce project artifacts.
    - The *Development Project Artifacts* are **views**, **models** and **diagrams** defined by UML
    - In terms of the views of a model, the UML defines the following graphical diagrams:
        - Use case diagram.
        - Class diagram.
        - Behavior diagrams:
            - Statechart diagram.
            - Activity diagram.
            - Interaction diagrams:
                - Sequence diagram.
                - Collaboration diagram.
        - Implementation diagrams:
            - Component diagram.

        - Deployment diagram.
    - Although other names are sometimes given to these diagrams, this list constitutes the canonical diagram names.
        - These diagrams provide multiple perspectives of the system under analysis or development.
        - The underlying model integrates these perspectives so that a self-consistent system can be analyzed and built.
        - These diagrams, along with supporting documentation, are the primary artifacts that a modeler sees, although the UML and supporting tools will provide for a number of derivative views.
            - A frequently asked question has been: *"Why doesn't UML support data-flow diagrams (DFD)?"* The answer: *"Data-flow and other diagram types that were not included in the UML do not fit as cleanly into a consistent object-oriented paradigm".*
        - Activity diagrams and collaboration diagrams accomplish much of what people want from DFDs, and then some.
        - Activity diagrams are also useful for modeling workflow.
- (2) **Motivation to Define the UML**
    - *Why We Model.*
        - Developing a model for an industrial-strength software system prior to its construction or renovation is as essential as having a blueprint for large building.
        - Good models are essential for communication among project teams and to assure architectural soundness.
        - We build models of complex systems because we cannot comprehend any such system in its entirety.
        - As the complexity of systems increase, so does the importance of good modeling techniques.
        - There are many additional factors of a project's success, but having a rigorous modeling language standard is one essential factor.
        - A **modeling** language must include:
            - (1) Model elements — fundamental modeling concepts and semantics.
            - (2) Notation — visual rendering of model elements.
            - (3) Guidelines — idioms of usage within the trade.
        - In the face of increasingly complex systems, visualization and modeling become essential.

        - The UML is a well-defined and widely accepted response to that need.
        - It is the visual modeling language of choice for building object-oriented and component-based systems.
- *Industry Trends in Software*
    - (1) As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software.
    - (2) We look for techniques to improve quality and reduce cost and time-to-market.
        - These techniques include:
            1. (1) Component technology.
            2. (2) Visual programming.
            3. (3) Patterns.
            4. (4) Frameworks.
    - (3) We also seek techniques to manage the complexity of systems as they increase in scope and scale.
        - In particular, we recognize the need to solve recurring architectural problems, such as:
            1. (1) Physical distribution.
            2. (2) Concurrency.
            3. (3) Replication.
            4. (4) Security.
            5. (5) Load balancing.
            6. (6) Fault tolerance.
    - (4) Development for the worldwide web makes some things simpler, but exacerbates these architectural problems.
    - (5) **Complexity** will vary by application domain and process phase.
        - One of the key motivations in the minds of the UML developers was to create a set of semantics and notation that adequately addresses all scales of architectural complexity, across all domains.
- *Prior to Industry Convergence*
    - Prior to the UML, there was no clear leading modeling language. Users had to choose from among many similar modeling languages with minor differences in overall expressive power.

- Most of the modeling languages shared a set of commonly accepted concepts that are expressed slightly differently in various languages.
  - This lack of agreement discouraged new users from entering the object technology market and from doing object modeling, without greatly expanding the power of modeling.
  - Users longed for the industry to adopt one, or a very few, broadly supported modeling languages suitable for general-purpose usage.
  - Some vendors were discouraged from entering the object modeling area because of the need to support many similar, but slightly different, modeling languages.
  - In particular, the supply of add-on tools has been depressed because small vendors cannot afford to support many different formats from many different front-end modeling tools.
- It is important to the entire object industry to encourage broadly based tools and vendors, as well as niche products that cater to the needs of specialized groups.
- The perpetual cost of using and supporting many modeling languages motivated many companies producing or using object technology to endorse and support the development of the UML.
- While the UML does not guarantee project success, it does **improve** many things.
  - (1) Significantly lowers the perpetual cost of training and retooling when changing between projects or organizations.
  - (2) Provides the opportunity for new integration between tools, processes, and domains.
  - (3) Enables developers to focus on delivering business value and gives them a paradigm to accomplish this.

- (3) **Goals of the UML** The primary design goals of the UML are as follows:
  - (1) Provide users with a ready-to-use, expressive visual modeling language to develop and exchange meaningful models.
  - (2) Furnish extensibility and specialization mechanisms to extend the core concepts.
  - (3) Support specifications that are independent of particular programming languages and development processes.
  - (4) Provide a formal basis for understanding the modeling language.
  - (5) Encourage the growth of the object tools market.
  - (6) Support higher-level development concepts such as components, collaborations, frameworks and patterns.

- (7) Integrate best practices.
- These goals are discussed in detail below.

- *(1)* ***Provide users with a ready-to-use, expressive visual modeling language to develop and exchange meaningful models***
  - The Object Analysis and Design (OA&D) standard supports a modeling language that can be used "out of the box" to do normal general-purpose modeling tasks.
  - The standard merely provides a meta-meta-description that requires tailoring to a particular set of modeling concepts, then it will not achieve the purpose of allowing users to exchange models without losing information or without imposing excessive work to map their models to a very abstract form.
  - The UML consolidates a set of core modeling concepts that are generally accepted across many current methods and modeling tools.
  - These concepts are needed in many or most large applications, although not every concept is needed in every part of every application.
  - Specifying a meta-meta-level format for the concepts is not sufficient for model users, because the concepts must be made concrete for real modeling to occur.
  - If the concepts in different application areas were substantially different, then such an approach might work, but the core concepts needed by most application areas are similar and should be supported directly by the standard without the need for another layer.

- *(2)* ***Furnish extensibility and specialization mechanisms to extend the core concepts***
  - OMG expects that the UML will be tailored as new needs are discovered and for specific domains.
  - At the same time, it's not necessary to force the common core concepts to be redefined or re-implemented for each tailored area.
  - Therefore, the extension mechanisms should support deviations from the common case, rather than being required to implement the core modeling concepts themselves.
  - The core concepts should not be changed more than necessary. Users need to be able to:
    - Build models using core concepts without using extension mechanisms for most normal applications,
    - Add new concepts and notations for issues not covered by the core,
    - Choose among variant interpretations of existing concepts, when there is no clear consensus,

        - Specialize the concepts, notations, and constraints for particular application domains.

- *(3) **Support specifications that are independent of particular programming languages and development processes***
    - The UML must and can support all reasonable programming languages.
    - It also must and can support various methods and processes of building models.
    - The UML can support multiple programming languages and development methods without excessive difficulty.
- *(4) **Provide a formal basis for understanding the modeling language***
    - Because users will use formality to help understand the language, it must be both precise and approachable; a lack of either dimension damages its usefulness.
    - The formalisms must not require excessive levels of indirection or layering, use of low-level mathematical notations distant from the modeling domain, such as set-theoretic notation, or operational definitions that are equivalent to programming an implementation.
    - The UML provides a formal definition of the static format of the model using a metamodel expressed in UML class diagrams.
        - This is a popular and widely accepted formal approach for specifying the format of a model and directly leads to the implementation of interchange formats.
    - UML expresses well-formedness constraints in precise natural language plus Object Constraint Language expressions.
    - UML expresses the operational meaning of most constructs in precise natural language.
- *(5) **Encourage the growth of the object tools market***
    - By enabling vendors to support a standard modeling language used by most users and tools, the industry benefits.
    - While vendors still can add value in their tool implementations, enabling interoperability is essential.
    - Interoperability requires that models can be exchanged among users and tools without loss of information. This can only occur if the tools agree on the format and meaning of all of the relevant concepts.
    - Using a higher meta-level is no solution unless the mapping to the user-level concepts is included in the standard.
- *(6) **Support higher-level development concepts such as components, collaborations, frameworks, and patterns***

    - Clearly defined semantics of these concepts is essential to reap the full benefit of object-orientation and reuse.
    - Defining these within the holistic context of a modeling language is a unique contribution of the UML.
- *(7)* ***Integrate best practices***
    - A key motivation behind the development of the UML has been to integrate the best practices in the industry, encompassing widely varying views based on levels of abstraction, domains, architectures, life cycle stages, implementation technologies, etc.
    - The UML is indeed such an integration of best practices.
- (4) **Scope of the UML**
    - The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system.
    - (1) First and foremost, the Unified Modeling Language fuses the concepts of Booch, OMT, and OOSE. The result is a single, common, and widely usable modeling language for users of these and other methods.
    - (2) Second, the Unified Modeling Language pushes the envelope of what can be done with existing methods.
        - *As an **example**, the UML authors targeted the modeling of concurrent, distributed systems to assure the UML adequately addresses these domains.*
    - (3) Third, the Unified Modeling Language focuses on a standard modeling language, not a standard process.
        - Although the UML must be applied in the context of a process, it is our experience that different organizations and problem domains require different processes.
        - *For **example**, the development process for shrink-wrapped software is an interesting one, but building shrink-wrapped software is vastly different from building hard-real-time avionics systems upon which lives depend.*
    - (4) Therefore, the efforts concentrated first on a common metamodel (which unifies semantics) and second on a common notation (which provides a human rendering of these semantics).
    - (5) The UML authors promote a development process that is:
        - Use-case driven.
        - Architecture centric.
        - Iterative and incremental.

    - (6) The UML specifies a modeling language that incorporates the object-oriented community's consensus on core modeling concepts. It allows deviations to be expressed in terms of its extension mechanisms.
    - (7) The Unified Modeling Language provides the following:
        - Semantics and notation to address a wide variety of contemporary modeling issues in a direct and economical fashion.
        - Semantics to address certain expected future modeling issues, specifically related to component technology, distributed computing, frameworks, and executability.
        - Extensibility mechanisms so individual projects can extend the metamodel for their application at low cost. We don't want users to directly change the UML metamodel.
        - Semantics to facilitate model interchange among a variety of tools.
        - Semantics to specify the interface to repositories for the sharing and storage of model artifacts.
- (5) **UML - Present and Future**
    - The UML is nonproprietary and open to all.
    - It addresses the needs of user and scientific communities, as established by experience with the underlying methods on which it is based.
    - Many methodologists, organizations, and tool vendors have committed to use it.
    - Since the UML builds upon similar semantics and notation from Booch, OMT, OOSE, and other leading methods and has incorporated input from the UML partners and feedback from the general public, widespread adoption of the UML should be straightforward.
    - There are two aspects of "unified" that the UML achieves:
        - First, it effectively ends many of the differences, often inconsequential, between the modeling languages of previous methods.
        - Secondly, and perhaps more importantly, it unifies
            - The perspectives among many different kinds of systems (business versus software).
            - Development phases (requirements analysis, design, and implementation).
            - Internal concepts.
- (6) **Standardization of the UML**
    - Many organizations have already endorsed the UML as their organization's standard, since it is based on the modeling languages of leading object methods. The UML is ready for widespread use.

    - The Unified Modeling Language v. 1.1 specification which was added to the list of OMG Adopted Technologies in November 1997. Since then the OMG has assumed responsibility for the further development of the UML standard.

- **(7) Industrialization**
    - Many organizations and vendors worldwide have already embraced the UML. The number of endorsing organizations is expected to grow significantly over time.
    - These organizations will continue to encourage the use of the Unified Modeling Language by making the definition readily available and by encouraging other methodologists, tool vendors, training organizations, and authors to adopt the UML.
    - The real measure of the UML's success is its use on successful projects and the increasing demand for supporting tools, books, training, and mentoring.
- **(8) Future UML Evolution**
    - Although the UML defines a precise language, it is not a barrier to future improvements in modeling concepts. We have addressed many leading-edge techniques, but expect additional techniques to influence future versions of the UML.
    - Many advanced techniques can be defined using UML as a base.
    - The UML can be extended without redefining the UML core.
    - The UML, in its current form, is expected to be the basis for many tools, including those for visual modeling, simulation, and development environments.
    - As interesting tool integrations are developed, implementation standards based on the UML will become increasingly available.
    - The UML has integrated many disparate ideas, so this integration will accelerate the use of object-orientation.
    - Component-based development is an approach worth mentioning.
        - It is synergistic with traditional object-oriented techniques.
        - While reuse based on components is becoming increasingly widespread, this does not mean that component-based techniques will replace object-oriented techniques.
        - There are only subtle differences between the semantics of components and classes.

1.5.8 Coverage Matrices

- *Coverage matrices.* A *coverage matrix* is a means of showing the relationship between two kinds of information.

    - An index in a book is a coverage matrix for it shows what pages in the book cover what subjects.
- A *coverage matrix* in a program design document might show system functions versus module names;
    - That is, for a given capability stated in the Problem Specification, what program module in the Design Specification provides that capability
- Such a matrix is even more useful in testing where it can be used to list functions to be tested **versus** identifications of the tests that are to cover those functions.
    - ***Example****: One IBM manager used a coverage matrix in another way. He listed all the items of work specifically called out in the contract down one axis and the names of the people responsible for those tasks along the other axis. The first attempt at using this matrix showed some tasks not covered by anyone. When the coverage matrix was passed among the workers, it also brought to light the fact that the boss thought person A was responsible for a particular task and A thought that task was being handled by person B.*

1.5.9 Storage maps.

- *Storage maps* are pictures describing how the various storage devices (core, disk, tape, CD's etc.) are being utilized.
    - In most cases, a simple diagram showing how various blocks of storage are being used (that is, by what programs and for what purposes) is sufficient.
- If *storage* is being **allocated dynamically** in your system, *storage maps* can still be used to answer the following basic questions:
    - Where does each kind of program reside?
    - Where does each kind of data file reside?
    - Where are the overflow storage areas, if any?
    - What provisions are made for backup storage?

1.5.10 Programming languages

- The way you design may affect, and will certainly be affected by, the language or languages in which you choose to code your system.
    - If your language is "higher-level" (for example, FORTRAN, COBOL, PL/I, PASCAL, Basic, Visual Basic, Delphi, C, C++, C#, Java, J+) as opposed to assembly language, you may use the **language itself** to express some of your design.
- In any case, be sure to treat language selection as one of your important design decisions.

    - (1) Frequently a job is done in language X because that's the language this group of programmers knows best —and sometimes this makes sense. This decision should be a positive one and not arrived at by default.
    - (2) Even though your programmers may be assembly language experts, maybe they should switch to other language for the current job.
    - (3) Or perhaps part of your job, such as a supervisor program, should be coded in assembly language and the rest in some other language.
- Mixing languages is often sound.
    - If you elect to mix languages, be sure that your designers understand what interfacing problems may be introduced between the languages chosen.
- Here are some questions you need to consider before you **select** a language for a given program:
    - (1) How frequently is the program to be executed?
        - If infrequently, it may be a candidate for the less efficient code usually (but not always) resulting from the use of a high-level language.
        - If the program is to be executed frequently (as in the case of supervisory programs, dispatchers, schedulers, input-output programs, embedded applications), it might make sense to code in assembly language or C.
        - In the case of programs written to solve **specific engineering problems** in which a mathematical solution is sought, the choice is usually clear: use the language that enables you to code, test, and arrive at an answer in the shortest calendar time.
            - The running-time efficiency of the program can be ignored because it's a throwaway, that is, it won't be used again once it has given an answer.
            - These one-shot programs, however, are generally independent and not part of a program *system*.
        - Is in discussion the possibility to use an object-oriented language?
    - (2) Is computer internal storage space an important constraint?
        - If storage is limited, assembler code may be necessary.
        - A given program can be coded using less core storage in assembler code than in high-level code, provided you have programmer experts in the use of assembly language.
    - (3) Is calendar time critical?
        - If it is, and if you have a choice between competent assembly language programmers and equally competent high-level language programmers, you can probably save calendar time by going with the latter.

        - High-level code can usually be written and tested faster than assembler code, with fewer errors.
    - (4) How competent and experienced are your programmers in the candidate languages? The answer to this may override all other considerations.
    - (5) Does each language processor or controller you are considering support all the input-output devices necessary for this job? Careful. It's very easy to make mistakes with unpleasant consequences.
        - *Metzger presents a story about of a job in which the contract specified the use of COBOL, but the project manager learned later on in the project that COBOL did not support all the input-output devices he was using. For months the customer insisted on adherence to the contract, and he agreed to change it only after many time-consuming and embarrassing meetings and compromises between contractor and customer management at the highest levels. You can say that the contractor should have caught the error before the contract was signed and that the customer shouldn't have been so pig-headed, but the point is, it happened.*
    - (6) Can you afford the time to train your programmers in a new language? (You should have thought of that before you signed a contract.)
        - A COBOL programmer does not attend a two-week course in assembly language coding and emerge an assembly language expert. Can you afford his inevitable false starts?
    - (7) Is the language you have selected fully supported on all machines that you will be using?
        - Suppose you develop your programs on one machine and install and maintain them on another, the operational machine.
        - Unless the development machine is to be retained for making future modifications to the programs, the operational machine must include in its support program repertoire the same assembler or high-level language processing capabilities used on the development machine.
    - (8) Would it make sense in your case to **code** some programs **first** in *a high-level language* (to get something running early in order to test design concepts) and **then** later **recode** critical modules in *assembler language*?
        - That's a luxury that may apply only to very large projects, but consider it especially for embedded systems projects.
- These questions are intended to get you to think about alternatives.
    - In the end, however, you must choose the language (or languages), that seems best for your job.
    - Again, such factors as the competence and experience of your programmers may override everything else.

1.5.11 Simulation models

- The dictionary defines a model as "a description or analogy used to help visualize something that cannot be directly observed."
- There are many kinds of models, including physical scale models, mathematical models, blueprints, and artists' models.
    - Tom Humphrey, a simulation modeling expert, points out that even your desk calendar on which you jot down dates and times for appointments and meetings is a model of a portion of your life.
- The kind of model of interest here is a *computer simulation model.*
    - In this case, **the system being modeled** is expressed in some computer language and the computer executes the resultant programs in imitation of the described system.
    - *For **example**, if you are interested in a system of automated traffic control along city streets, you might describe a tentative control system in an appropriate simulation modeling language, run the model on a computer under varying traffic conditions, and determine from the model where bottlenecks are likely to occur. You can alter parameters (such as duration of red and green lights) and run the model again to see what effect your alterations have on the traffic flow.*
- *Simulation modeling* can be a powerful tool for the designer and, in fact, can be **extremely helpful** all the way through the project.
- Although it may be costly, depending on how much you want to simulate, it can pay for itself by providing the following **advantages**:
    - (1) **Alternative design approaches** can be tried under simulation before committing the project to one approach or another.
    - (2) **System bottlenecks** can be discovered and possible remedies investigated.
    - (3) **Problems**, such as **running out of internal storage**, can be predicted in advance so that there is time to avoid them.
    - (4) Building a simulation model **to test a proposed design** tends to force **design completeness** because the modelers must keep asking questions of the designers until the model itself is complete.
- The new SW development technologies are mainly based on different **types of models** (RUP).

1.6 Assessing Design Quality

- One test of design quality is obviously the degree of success of the resultant program system in meeting the customer's requirements.

    - However, that's not enough.
    - The program system may run perfectly and do the job intended, and yet be **inadequate** because it's difficult to alter to meet future needs.
- Long before acceptance time, in fact *now,* before programs are actually written, managers must be able to get some handle on the **quality** of the baseline design.
- The following **set of questions** might be used as a starting point in **assessing design quality:**
    - (1) Are all functions spelled out in the Problem Specification fully accounted for in the Design Specification?
    - (2) Have the interfaces between the program system and human operators or machines been designed with ease of use in mind, even at the sacrifice of some programming simplicity?
    - (3) Has the program system been broken into modules small enough to fit on a single computer printout sheet (say, 40 to 60 lines of code) or on a reduced number of screens?
    - (4) Has each module been designed to perform a single specific function (such modules are said to possess "**functional strength**")?
    - (5) Has each module been designed with maximum independence in mind?
        - The best means of communication between modules is *data coupling,* wherein module A calls module B, passing to B needed data and receiving data back from B.
        - Other means of coupling provide less module independence; for example, when module A directly refers to the contents of module B.
        - Does each module have complete predictability?
    - (6) Modules whose behavior is dependent on self-contained status indicators are less predictable and more difficult to test.
    - (7) Are the **rules** of module-to-module communication and access to all data clearly and completely stated?
        - Leave no room for private agreements among programmers concerning inter-module communication.
    - (8) Is the set of documents representing the design clear, complete, and ready for further refinement and coding by the programmers?

2 PROJECT PLANNING

- In parallel preparation with the baseline design effort, further planning and preparation are done by the **Project Manager** during the **Design Phase**.

- Most of this work falls into these areas:
  - (1) Change control.
  - (2) Preparation for testing.
  - (3) Resource training.
  - (4) Documentation.

2.1 Change control

- The design process does not end when you have produced the Design Specification.
- Throughout the life of the project, and especially during the **Programming Phase**, changes will be proposed either to the Problem Specification or to the Design Specification, or to both.
- A mechanism for dealing with and controlling change will be described in the **next** chapter.
  - But *now* is the time to **set up the apparatus**.
  - When **change proposals** come in, you should be ready to dispose of them quickly.

2.2 Preparation for testing

- Like most activities during the life of a project, testing must be prepared for in advance of when it will actually take place.
- Testing will begin during the **Programming Phase**.
  - Getting ready for it must be done during the **Design Phase**.
- Detailed discussion of testing is left to later chapters, but now you must **set the stage**.

2.2.1 Defining a test hierarchy

- You need to **define** the types, or levels, of **testing** to be done on your project.
  - Define them, publish them, and stick with them.
- It is know that when terms such as "integration test" and "system test" are used, their meanings depend on who is using them.
  - You need to be sure that ***for your project*** you have unambiguous definitions understood by all the project members, management outside the project, and the customer.
- There are **fife test levels** of modules comprising the program system.
  - Metzger offers the following definitions of test levels in a test hierarchy (they are further discussed in succeeding chapters).

- *(1)* ***Module test*** is the testing done on any **individual module** before it is combined (integrated) with the rest of the system.
    - Module test is done by **individual programmers**.
- *(2)* ***Integration test***. Also simply called "**integration**" this is the process of:
    - (1) Adding **a new module** to the evolving system.
    - (2) Testing this **new combination**.
    - (3) Repeating the process until finally the **entire system** has been brought together and thoroughly tested.
- *(3)* ***System test***. The integrated program system that the programmers consider clean is now run through a **new series of tests**.
    - This series is *not* prepared or executed by the programmers.
    - These new tests are run in as nearly a "live" final environment as possible.
    - Their main objective is to test the programs against the original Problem Specification to determine whether or not the system does the job it was intended to do.
- *(4)* ***Acceptance test.*** The program system is tested under conditions agreed to by the **customer**, with the objective of demonstrating to him that the system satisfies the contract requirements.
    - In many cases, acceptance test is completely controlled by the customer or his representative.
- *(5)* ***Site test***. After installation in its ultimate operating environment, the program is tested once again to assure complete readiness for operation.

2.2.2 Top-down vs. bottom-up integration testing

- Like many other aspects of the programming business, the theories and practices concerning **testing** have undergone a good deal of change.
    - The changes have been both philosophical and practical.
- Traditionally, a system made up of several levels of modules suggested in a generalized way in Figure 5.1.4.a has been tested "**bottom-up**."

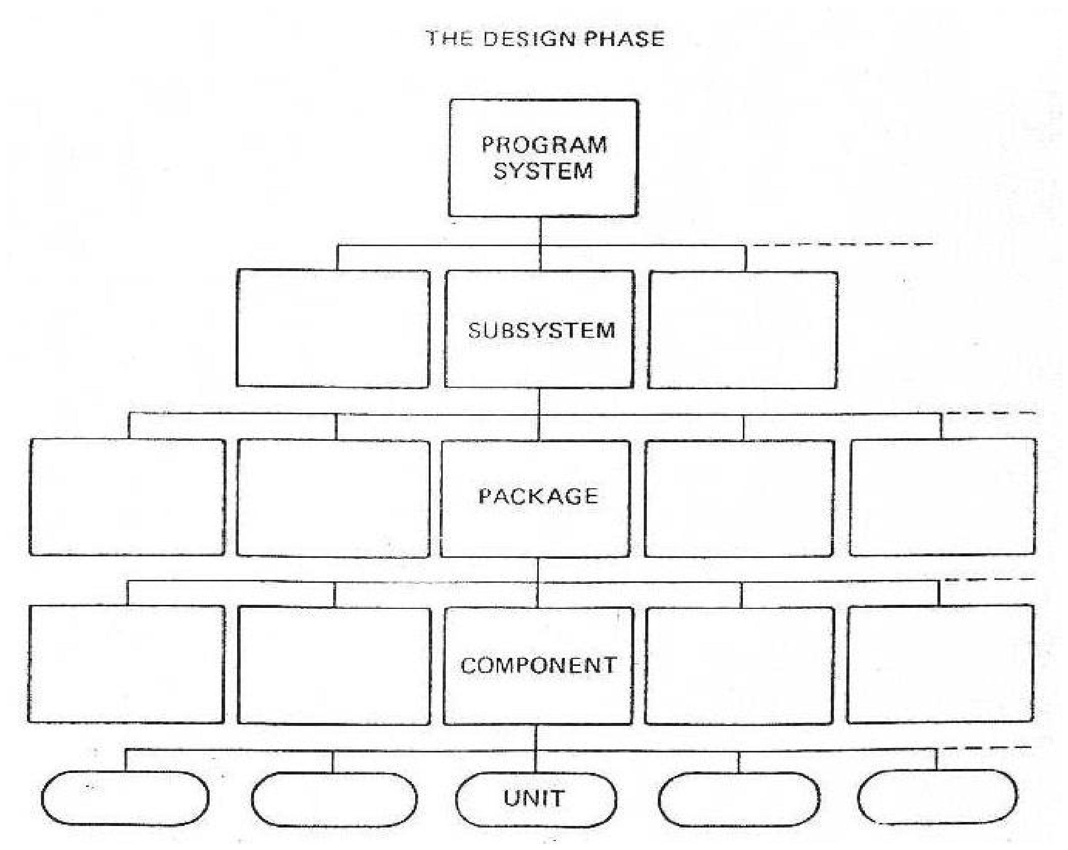


Fig.5.1.4.a. System WBS (Hierarchy) (replicate)

- **Bottom-up** testing presumes:
    - (1) Lowest-level modules (in Fig. 5.1.4.a, "units") would be coded and tested first on a "stand-alone" basis.
    - (2) When the units comprising a higher-level module ("component") were ready, they would be combined and the combination tested until that component ran successfully.
    - (3) Meanwhile, other components would have been readied in a similar manner, and eventually appropriate groupings of components would be tested.
    - (4) And so on up the hierarchy, until one day —lo! the system was tested.
- **Top-down** testing is just the reverse, at least philosophically.
    - (1) Testing begins with the top-level modules and proceeds down through the hierarchy.
    - (2) The lowest-level modules are the last to be added to the system and tested.

- Either method of testing can be chosen and either will work, given adequate planning and control.
    - But you, the manager, need to understand what's involved in each method and what's at stake in making your choice.
- The first thing to understand is that the choice between top-down and bottom-up testing is a choice between two **philosophies**, or **basic approaches**.
    - They are not necessarily **mutually exclusive** testing methods, each involves some of the other.
- For **example**:
    - In using the **bottom-up** method, it would normally be practical or necessary to provide a **framework** into which the modules could be inserted for test purposes.
        - This **framework** is usually a **bare-bones** version of the system's control program (a high-level module in the eventual system).
    - In **top-down** testing, it's often necessary to code and test **very early** some modules (such as an output module) which have been shown at a very low level in the hierarchy.
    - So neither approach is completely sanitary. There will generally be some **mixing** of the two.
        - There's nothing wrong with that; what counts, after all, is that the system be well tested and done on time.
        - What *is important* is that **one approach** or the other be selected for your project.
- The approach recommend by Metzger is **top-down,** but later discussions in this book allow for either approach to be used.
- Following is a summary of the reasons for making this choice:
    - (1) *Top-down* is a "natural" method; it involves building a **framework** before adding details.
    - (2) It fits comfortably with the ideas of top-down design and top-down coding.
        - The whole idea of top-down development is that of a natural progression.
    - (3) As new modules in the hierarchy are added to the system and tested with the already-tested higher-level modules, the system evolves as a living, growing entity, "complete" at any given stage.
        - Bottom-up is a more piecemeal approach, involving more finger-crossing and more surprises when groups of modules are thrown together for the first time.

    - (4) It's easier to produce intermediate versions of the final program system; effectively you have an intermediate version right from the beginning — something is cycling and showing results. This makes it easier:
        - (a) To show the customer intermediate results faster, thus avoiding end-of-project shocks.
        - (b) To deliver interim, incomplete versions of the system.
        - (c) To show management that something really is being produced. Upper management has always been in the untenable position of having to accept too much on faith; it's so hard to actually see those darned programs!
    - (5) Program system integration is being continually performed as each new module is added to, and tested with, the higher-level modules already existing.
    - (6) Much less scaffolding (specially written test support code, such as dummy driver programs) is needed in top-down development.

2.2.3 Writing test specifications

- Module testing is done by the individual programmer **without** a formal test document.
- The other four levels — integration, system, acceptance, and site testing — are performed according to previously defined **test specifications**.
- There is a **separate set** of test specifications for each level.
- Many individual tests will serve **more** than one test category.
    - For example, many, perhaps all, of the acceptance demonstration tests can be taken from the set of system tests.
- Figure 5.2.2.3.a. indicates at what point in the development cycle each **specification** should be ready.
    - It's evident that if the Integration Test Specification is to be ready for use when integration testing begins, it will have to be prepared during the **Design Phase**.
    - Similarly the System Test Specification must be written during the **Programming Phase,** and so on.
    - **Test specifications** are described later. Here it's sufficient to indicate that:
        - Each **test specification** contains a definition of:
            - (1) Test objectives.
            - (2) Success criteria.
            - (3) Test data.
            - (4) Test procedures.

- Each **specification** is supported by **test cases**.
- A **test case** contains:
  - All the background information.
  - Test data.
  - Detailed procedures required to execute one specific set of tests.

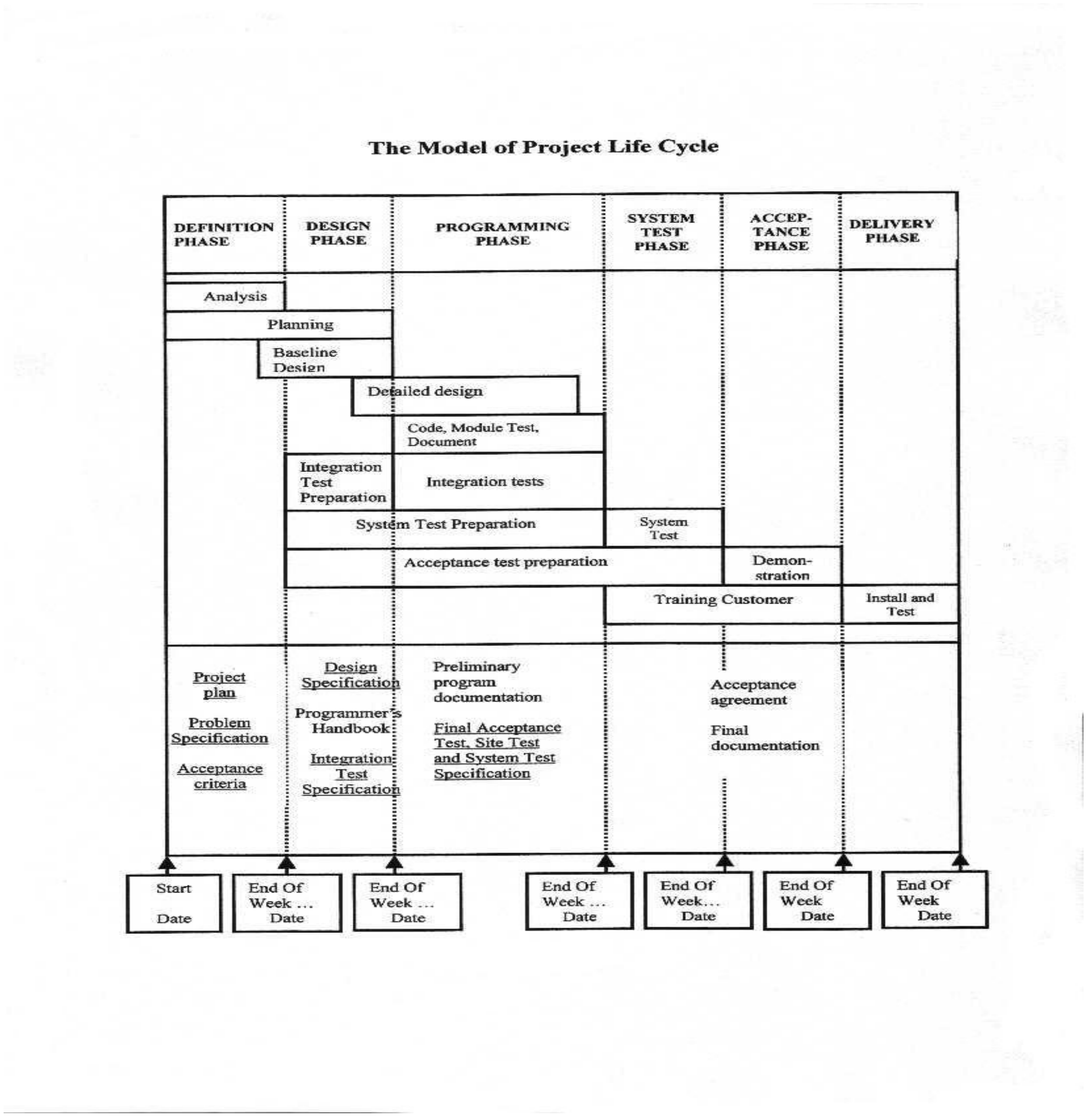


Fig.5.2.2.3.a. The life cycle model of a SW project

2.2.4 Defining test procedures

- The place to allow for creativity in testing is when you are devising the tests in the first place — not when you're executing them.
    - There will always be nail-biting moments during testing (especially during acceptance testing) no matter how well you prepare; don't add to the tension by flying blind.
- Write your **test specifications** in such a way that procedures, responsibilities, and predicted results are spelled out ahead of time.

2.3 Resource estimating

- As the end of the **Design Phase** nears, you should **re-estimate** the resources needed to finish the job.
    - Your early guesses about manpower and computer time can now be made more realistic because you've learned so much more about the job to be done.
- If you're new estimate is much higher than the original, that's a problem you and the customer must somehow resolve.
    - Better now than halfway through the **Programming Phase**.

2.4 Documentation

- The Documentation Plan should be in good shape at the end of the **Design Phase**, with most documents defined and outlined.
- The first version of the Programmer's Handbook should now be ready for distribution, and the project library should be set up.

2.4.1 Programming Manual

- The Programmer's Handbook is a loose-leaf binder containing the written information most vital to the programmer in doing his job.
- Programmer's Handbook includes (fig. 5.2.4.1.a):
    - (1) A description of the technical requirements
    - (2) The baseline design
    - (3) Support software
    - (4) Test procedures
    - (5) Hardware information
    - (6) A summary of the Documentation Plan.

PROGRAMER’S HANDBOOK (TEMPLATE)

DEPARTMENT:

PROJECT:

DOCUMENT NUMBER:

APROVALS:

DATE OF ISSUE:

REALISED:

SECTION 1: INTRODUCTION

1.1 Objectives

The Handbook is intended to be the source of the basic technical information required by all programmers in the project. The information in the Handbook is to be considered “law” until a change is approved and distributed.

The Handbook is not a single document, but a collection of documents that every programmer on the project should have close at the hand. It is extremely important that the Technical Staff (responsible for issuing and updating the Handbook) not allow additional materials to be added randomly.

The Handbook is in loose-leaf notebook form in order to facilitate updates. It can be also in HTML form in the company intranet. It is divided in sections with major tab for each and sub-tabs where appropriate. In the electronic form a search mechanism is usually added.

1.2. Scope

The Handbook should be restricted to the topics listed in this outline. There is a great deal of information pertinent to the project (plans, status reports, etc.) that is not included. The Handbook must be concise and usable from the programmer’s point of view.

1.2. Publication

Initial issue is made near the end of the Design Phase. Subsequent updates are made in two ways:

1. Routine weekly update (blue color)

2. Emergency 24-hour update (red color)

In either case, updates are handled in this manner:

1. Anyone drafts update and give it to Technical Staff.

2. Technical Staff gets the draft, analyze it, proofread, approves it and distributes it.

SECTION 2: THE PROBLEM

2.1. Introduction

A tutorial description of the customer, the environment, and the job to be done (if necessary). This should start from scratch and to be written so that a new project member can easily know what the job is all about. The limit should be about 2 pages.

2.2. The Problem Specification

The entire Problem Specification *is included here.*

SECTION 3: TESTING

The entire Test Plan *is included here.*

SECTION 4: SUPPORT PROGRAMS

Description of the programming environments and tools available to the programmers and how to use them. Each main category of tools should be separately tabbed within this section.

- *Programming environments*
- *Version control tool*

SECTION 5: THE DESIGN SPECIFICATION

The entire Design Specification *is included here. This document contains several main subsections as: Overall Design Concept, Design Standards and Conventions, Coding Standards and Conventions, Baseline Design.*

SECTION 6: DOCUMENTATION

6.1 Documentation Summary

A chart of Documentation Summary *is included here.*

6.2. Documentation tools

The used Documentation Tool is described here.

6.3. Documentation Index

A detailed cross reference index of all project documents, updated weekly (See Documentation Index*).*

SECTION 7: EQUIPMENT

A description of operational and support hardware to be used on the project (if necessary). The kind of information included here:

- *The topology of the network*
- *Technical characteristics of the equipment*
- *Geographical distribution of the network*

SECTION 8: GLOSSARY

Definition of the project terms, including the names of the program levels, and testing levels, customer jargon, equipment nomenclature.

Fig.5.2.4.1.a. Programmer's Handbook template

- The **handbook** should be ready when programming begins.
  - Its preparation and upkeep should be entrusted to a technical person, not to an administrator, because it is an important set of information for the *programmer.*
- Don't let it become too broad, with a section for filing every conceivable kind of document on the project.
  - And above all don't include in it such items as the detailed program description.
- Limit not only the size of the handbook but also the distribution.
- Keep it as small as possible.
  - The bigger it gets, the less likely that anyone will consult it; and the longer the distribution list, the tougher and more expensive it is to update the handbook.

- Today, based on Intranet facilities, many projects are reducing the need for handbooks by storing critical data in the computer and making them accessible through terminals where selected portions can be either printed or displayed on a screen, or both.
    - A huge advantage in any such centralized library is that it can always be kept current; the often slow publication and distribution cycle is avoided.

2.5. Configuration Management

- The Configuration Management should be fully organized and operating at the end of the **Design Phase**.
    - It can be implemented in different ways.
- (1) One way of functionally organizing the *configuration management* is the **project library** (Metzger).
    - This concept is shown in Figure 5.2.4.a and described more fully in next chapter.

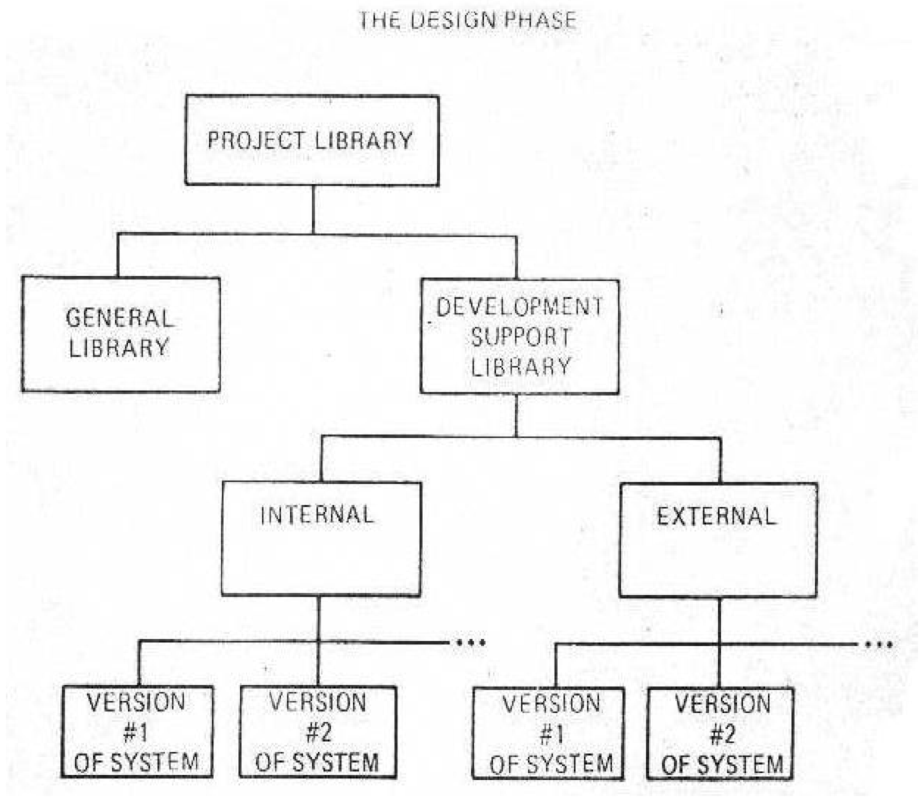


Fig.5.2.4.a. Project Library model

    - Every item (document or program module) should be given a unique identification.
    - Nobody but the librarian should ever get his hands on the master copy of a document or a module.
    - This is a convenient control point for the project, but it will lose much of its value if project members are allowed free access to it.
    - The use of the library will be discussed further in the next chapters.
- (2) Another way is to use a **specific tool** as:
    - **ClearQuest** (Rational).
    - **CVS** (Control Versions System).
    - **ClearCase** (Rational).
    - **Continuos.**
    - **CM Sinergy** (Telelogic).

2.6 Training

- During the **Design Phase** you should train the programmers for their jobs during the **Programming Phase.**
- By the time the **Programming Phase** begins, they should all know:
    - The equipment.
    - The programming language.
    - The test facilities to be used.
    - The problem definition.
    - The baseline design.

3 Design Phase Review

- **Status reviews** should be held **at the end of every phase** in the development cycle as was established in the Review and Reporting Plan.
- Because the review held at the end of the **Design Phase** is probably the most critical of all, we'll look at it in some detail.
    - At the end of the **Design Phase** you're almost at a **point-of-no return.**
    - You're about to commit **major resources** (programming manpower, testing manpower, and computer time) and you'd better have a warm feeling that you're really ready.
    - Once you begin implementing that baseline design, it's exhausting, expensive, and morale busting to have to stop to do a major overhaul.

- The **objectives** of this review are:
    - (1) To assess the completeness and adequacy of the Baseline Design and the Project Plan.
    - (2) To provide management with sufficient information to decide whether:
        - To proceed to the next phase, or
        - Do some rework, or
        - Kill the project.

3.1 Preparation

- Don't mumble something about a review and hope everyone knows what you're talking about.
- State the objectives of the review and give someone complete responsibility for making all arrangements.
- This may be a **full-time job** for several days or weeks.

3.1.1 Scheduling people

- (1) Include as reviewers:
    - A cross-section of your own project members.
    - Representatives of management above you.
    - In addition, be sure to **invite** outside reviewers or consultants who are not only competent in both the technical side of your project and in project management but who are also disinterested in your project.
    - Don't invite your buddy, hoping he'll give you good marks.
- (2) Give this review a chance to uncover problems.
    - If problems are there, they will surface sooner or later, and the later they are found, the more difficulty you'll have in fixing them.
        - **For instance**, a design problem might be solved in a week or so during the **Design Phase**.
        - The same problem, not caught until programming is well under way, might cause a great deal of reprogramming and retesting.
- (3) Should you invite the customer? Metzger suggests **not**.
    - You may get into many **internal problems** during this review, and the customer's presence would be inhibiting.
    - It's better to solve as many problems as possible and ***then*** brief the customer.

    - This isn't double work at all, for you can treat your internal review as a warm-up for a customer review.
    - The intent is **not** to hide anything from the customer but to present him with solutions and alternatives, not problems.
- (4) Choose as project speaker members who are most competent in the areas being discussed.
    - Try to avoid starring someone who may be so glib that he glosses over problems and lulls the listeners into thinking things are in better shape than they are.
    - Sometime a review session begins with a "the-project-is-in-good-shape" attitude and ends in grief.
- (5) How much time you need for the review?
    - That depends on:
        - The size and nature of your project.
        - The complexity of the technical problems.
        - The difficulties you see in future phases.
    - On a one-year project you can probably plan on spending three to five days profitably in this review.
    - You should anticipate some prolonged discussions of problems and schedule people accordingly.

3.1.2 Scheduling meeting rooms

- Hold the meetings as far away from your office (and telephone) as possible.
    - If that means renting a hotel conference room for a week, do it.
- Arrange for frequent coffee breaks and, if necessary, for luncheons.
    - Find a place that's air conditioned and quiet; it's tough to compete with cigar smoke and the jackhammers next door.
- In some cases, you may want to break people away from a main meeting room to form smaller "task forces" in order to look in depth at a specific problem.
    - If so, you'll need to arrange for space for them.
- But resist the temptation to break up the group, at least until all main presentations are complete.
- Since this review is intended to show whether or not the whole project hangs together, the reviewers should hear all presentations, both technical and non-technical.

3.1.3 Preparing presentation aids

- Decide what presentation media (for example, slides, power-point presentations, flip charts, chalkboards, paper boards,) will be used and make sure that the appropriate equipment is available for each presenter.
- If some other group is to help you prepare charts and slides, find out how much lead time they need.

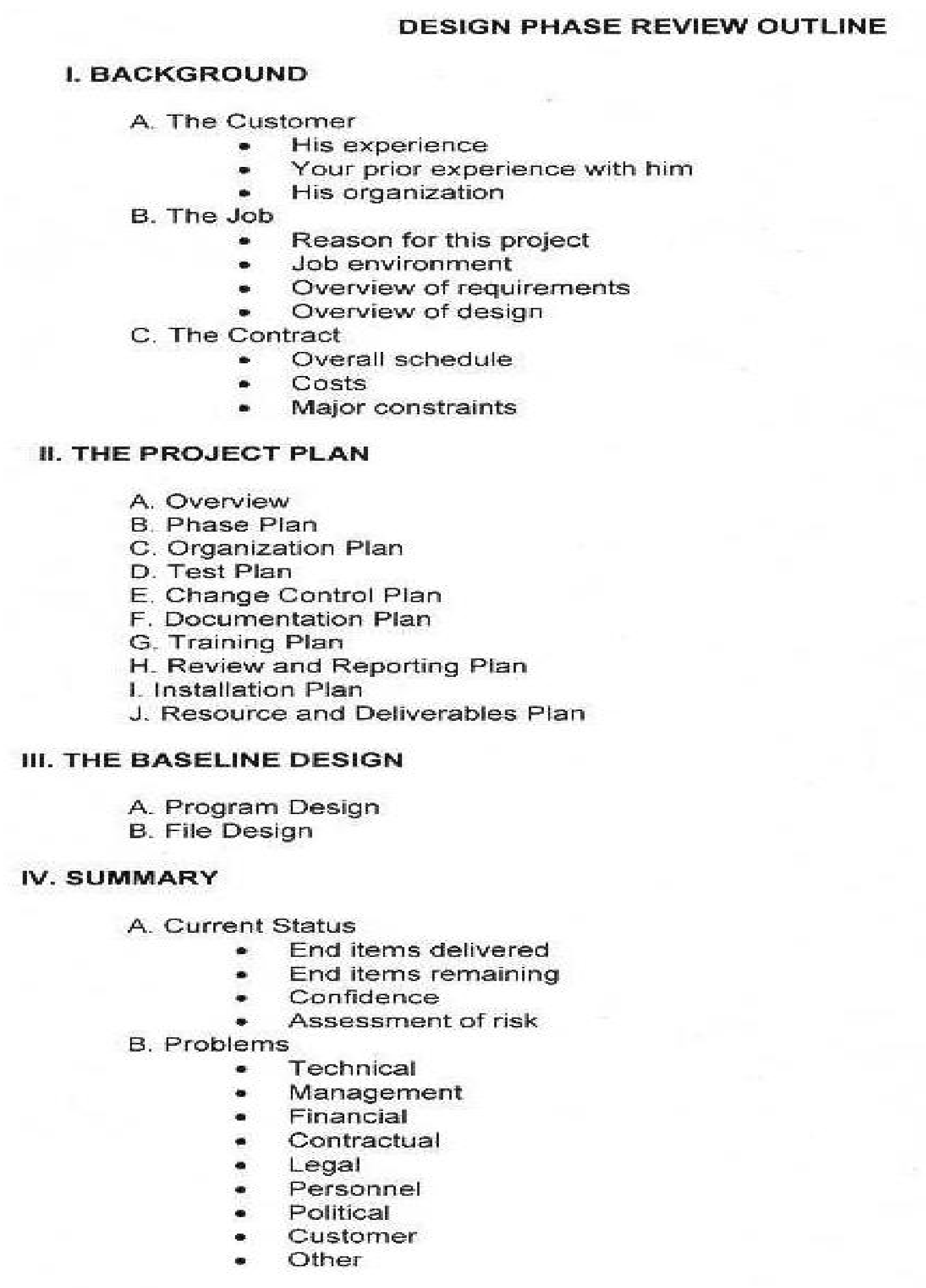
3.1.4 Preparing handout materials

- Be very selective about what you give the reviewers if you expect them to read it.
- The two documents that they should surely get are the Design Specification and the Project Plan. Beyond that, it's up to you.
- Whatever you hand out, make sure it's clean and readable.
- Send the documents to the reviewers well in advance of the actual meeting.

3.2 What to Cover

- The general objectives are to review **plans** and **the baseline design**.
- The figure 5.3.2.a suggests an outline of review topics you might use.
    - (I) The section *Background* should give the reviewers a feel for the environment in which you are working and a general idea of the technical problem as well as your proposed solution.
    - (II) Under the *Project Plan* heading, present at least a capsule description of each plan section and a more detailed look at the:
        - Phase Plan.
        - Organization Plan.
        - Test Plan.
        - Resources and Deliverables Plan.
    - (III) The *Baseline Design* section should describe in increasing levels of detail the design you have produced.
        - It may be a good idea to break for a day or so, once a first level of detail has been presented.
        - This will give the reviewers a chance to absorb what they have heard, look through the design document, and then return better able to hear more detail.
    - (IV) The *Summary* presentation should paint an **honest picture** of **where** you think you stand and **what major problems** you face.

- Distinguish between problems you feel you can solve and problems requiring your management's help.



DESIGN PHASE REVIEW OUTLINE

I. BACKGROUND

A. The Customer
- His experience
- Your prior experience with him
- His organization

B. The Job
- Reason for this project
- Job environment
- Overview of requirements
- Overview of design

C. The Contract
- Overall schedule
- Costs
- Major constraints

II. THE PROJECT PLAN

A. Overview
B. Phase Plan
C. Organization Plan
D. Test Plan
E. Change Control Plan
F. Documentation Plan
G. Training Plan
H. Review and Reporting Plan
I. Installation Plan
J. Resource and Deliverables Plan

III. THE BASELINE DESIGN

A. Program Design
B. File Design

IV. SUMMARY

A. Current Status
- End items delivered
- End items remaining
- Confidence
- Assessment of risk

B. Problems
- Technical
- Management
- Financial
- Contractual
- Legal
- Personnel
- Political
- Customer
- Other

Fig. 5.3.2.a. Design phase review outline

3.3 Results

- What you want from the review is the go-ahead to get on with the job.
  - In most cases it will be clear when the review ends what kind of shape you're in.
  - In other cases, you or your management will insist that certain problems be solved before programming can proceed.
- When you set up the review, ask certain participants to be prepared to put in writing:
  - (1) **Their opinions** of your project's status.
  - (2) **Their listing** of your outstanding problems.
  - (3) **Any suggestions** for dealing with those problems.
- Encourage reviewers to state what they see as problems even if they have no suggested solutions.
- You should solicit these written comments from each outside reviewer and from selected project members.
  - Their comments should be given to your management along with your own recommendations.
  - Every review should end with a formal written report from you.
- At the **conclusion** of a successful review, or after making changes as a result of the review, the **green light** is on and the **Programming Phase** begins.

Exercises #9

1. What are the objectives of the Design Phase?

2. What is the structure of the Design Specification document? Describe its main sections.

3. Which are the gold rules of a reasonable design?

4. What kind of design tools do you know? Describe their main features and applicability domain.

5. How can we asses the quality of the design activity?

6. What are the main activities of the project manager during Design Phase? Detail them.

7. How do we prepare the testing activity? Which are the main test philosophies? What kind of test levels do you know?

8. What can you say about the Documentation and Configuration Management during Design Phase?

9. Which are the objectives and how must be organized the Design Phase Review?