Chapter 3

MANAGING SOFTWARE PROJECTS

Summary

1.Introduction

1.1 Short history. SW Project Management Definition

- Over the time, much has *changed* in the computer business.
    - Computers have become *smaller*, more *powerful*, and dramatically *cheaper*.
    - There have been other changes, potentially even more significant in their impact on *computer usage*, but not as visible.
- During the **1970s** there was feverish activity among computer scientists and practicing programmers seeking *new methods of producing good programs*, on time and more efficiently.
- During the **1980s** the search continues, but a good many projects are already immersed in some of the *"new" methodologies*, such as structured programming and top-down development.
    - Before long, reliable programs, produced within the limitations of a budget and on time, will no longer be a rarity, although it will take some time before they become an everyday occurrence.
- Managers during the 1980s are faced with some *fundamental decisions* which were not so much in evidence during prior decades.
- Some of these decisions go to the very *core of programming* and affect everything the *manager* does from his first day on the project including:
    - (1) How he *organizes* his people,
    - (2) What kinds of *talent* he must find,
    - (3) How computer *developing* time should be spaced,
    - (4) How he defines the *development cycle* for the project.
- **Definition: *Project management*** is *"the application of knowledge, skills, tools, and techniques to project activities in order to meet or exceed stakeholder needs and expectations from a project."*

  [K. Schwalbe, "Information Technology Project Management", Thomson Learning, 2000]

  - This definition:
    - (1) Emphasizes using specific knowledge and skills
    - (2) Stresses the importance of the people involved in project management.
  - **Project managers** must:
    - (1) Strive to meet specific scope, time, cost, and quality goals of projects
    - (2) Facilitate the entire process to meet the needs and expectations of

the people involved in or affected by project activities.

- The fig. 1.1.a presents a **framework** describing the **project management**. Key elements of this framework include:
    - (1) The project stakeholders,
    - (2) Project management knowledge areas,
    - (3) Project management tools and techniques.

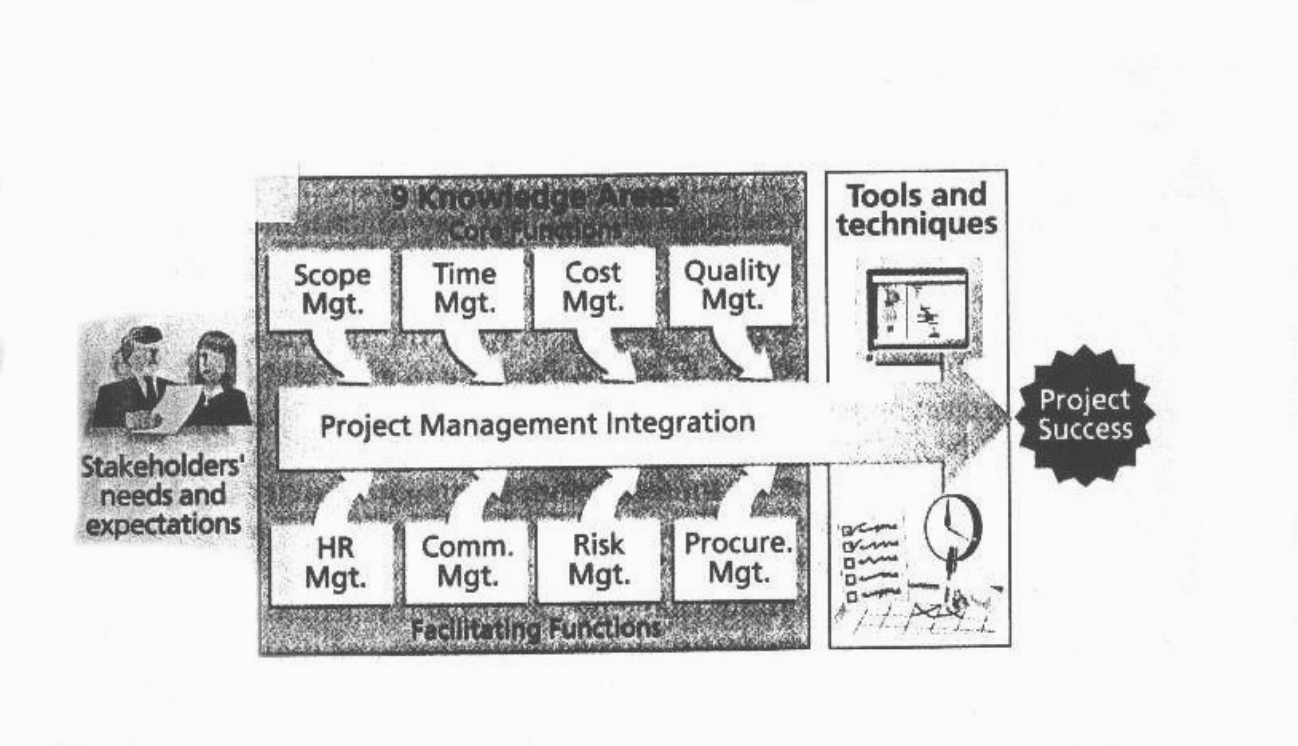

Fig. 1.1.a. Project management framework

- **(1) Stakeholders** are the people *involved in* or *affected by* project activities *and* include:
    - (a) The project sponsor
    - (b) Project team
    - (c) Support staff
    - (d) Customers, users, suppliers
    - (e) And even opponents to the project.
    - Successful project managers work on developing good relationships with **project stake-holders** to ensure their needs and expectations are understood and met.
- **(2) Knowledge areas** describe the key competencies that project managers must develop.
    - The center of Figure 1-2 shows the nine knowledge areas of project management.

- (2.1) The four **core knowledge areas** of project management include project *scope*, *time*, *cost*, and *quality management*.
- (2.2) The four **facilitating knowledge areas** of project management include *human resources, communications, risk*, and *procurement management.*
- (2.3) The **project management integration area**

- **(2.1)** The **core knowledge areas** lead to specific project objectives.
  - Brief descriptions of the **core knowledge areas** are provided below:
    - (1) **Project scope management** involves defining and managing all the work required to successfully complete the project.
    - (2) **Project time management** includes estimating how long it will take to complete the work, developing an acceptable project schedule, and ensuring timely completion of the project.
    - (3) **Project cost management** consists of preparing and managing the budget for the project.
    - (4) **Project quality management** ensures that the project will satisfy the stated or implied needs for which it was undertaken.
- **(2.2)** The four **facilitating knowledge areas** of project management are *human resources, communications, risk*, and *procurement management.*
  - These are called facilitating areas because they are the means through which the project objectives are achieved.
  - Brief descriptions of each **facilitating knowledge areas** are provided below:
    - (5) **Project human resource management** is concerned with making effective use of the people involved with the project.
    - (6) **Project communications management** involves generating, collecting, disseminating, and storing project information.
    - (7) **Project risk management** includes identifying, analyzing, and responding to risks related to the project.
    - (8) **Project procurement management** involves acquiring or procuring goods and services that are needed for a project from outside the performing organization.
- **(2.3) Project management integration** is an over-arching function that affects and is affected by all of the other knowledge areas.
  - **Project managers** must have knowledge and skills in all nine of these areas.
- **(3) Project management tools and techniques** assist project managers and their teams in carrying out scope, time, cost, and quality management.
  - Additional tools help project managers and teams carry out human resources, communications, risk, procurement, and integration management.

- For example, some popular time management tools and techniques include Gantt charts, network diagrams, and critical path analysis.

- Project management software is a tool that can facilitate management processes in all the knowledge areas.

1.2 The Management Business

- The *manager's* job is:
    - (1) **To plan an activity**
    - (2) **To see that it is carried out**.
- But from the instant the project begins, he must contend with the fact that *humans tend not to solve problems* until they become *crises*.
    - Only a crisis seems worthy of real attention
    - Whether it's a *strike deadline*, an *international diplomatic standoff*, *a human injustice*, nothing gets resolved until something overflows.
    - And of course, computer programs don't get serious attention until the deadline is terrifyingly close at hand or the customer is threatening to sue!
    - It's practically a **law**: *"A problem must reach crisis proportions before we act to solve it."*
- On SW projects there is a *common scenario*:
    - Time passes and problems develop.
    - Everybody knows it but the status charts are "fudged" in the name of optimism.
    - An important delivery date arrives and there's nothing to deliver. Mild panic. Meetings. When *will* the item be delivered, Mr. Manager? Next month. No question about it. We've put Charlie Superprogrammer on the job.
    - But Charlie turns out to be human; next month arrives, and still no product. Another cycle or two and in comes the Company-Vice-President-In-Charge-Of-Boondoggles. He'll fix things with his heavy hammer. But that doesn't work either.
    - Finally, a high-level decision is made to stop, take stock, and come up with a new plan for finishing. New plan? Ah, there's the rub. Quite likely it will be the first plan.
- The aim of this course is to get the *manager*, to ***plan,*** and then ***control*** his *project* according to that plan.
    - (1) Almost *any plan* is better than none at all.
    - (2) There is not other *alternative*.

1.3 Terminology

- The **target of the course**: the **manager** of a *medium-sized software project* involving a *medium number of programmers, managers, and others*.
    - Larger and smaller projects will be discussed latter
- Most of presented information is *vital* regardless of project size.
    - What *varies* from one job to the next is not *what tasks* manager need to do, but *how much horsepower* is necessary to do them.
- The term ***manager*** means different things to different organizations.
    - Usually it is used to identify two category of people:
        - (1) Who are **responsible** for *planning* and *directing* the implementation of some job, or project
        - (2) Who have **direct responsibility** for *hiring*, *firing, adjusting salaries, and promotions.*
- A ***first-level manager*** has the closest supervision over the people who actually build the product;
- A ***second-level manager*** supervises first-level managers, and so on.
    - In many organizations the term ***supervisor*** replaces what we called *a first-level manager*.
- Sometimes the two titles are synonymous, but:
    - More often the *supervisor* gives technical direction without having direct responsibility for hiring, firing, salary, and promotions;
    - But *supervisors* do have plenty of influence in all those areas.
- The terms ***program, software project*** and ***software*** are synonymous.
- ***Operational programs*** are those written to do the job for which your project exists, e.g., calculating payroll checks, directing space flights, producing management reports.
- ***Support programs*** are those utilized as aids in producing the operational programs.
- This course is divided into two parts.
    - (1) Part I describes what should happen on a well-behaved project;
    - (2) Part II outlines a planning document (the Project Plan) essential to good management.

1.4 Establishing the Ground Rules

- Reading the literature it's easy to despair of the diverse and often contradictory definitions and uses of terms such as software, module, integration, system test, and so on.
    - It's very important **to define** everything *clearly* and *unambiguously*.
- All programming people shall speak a *common tongue*, from **managerial** point of view
- The best we can do is decide *that now, for this project:*

    - (1) Which are the ***definitions*** to be used
    - (2) Which is the ***management scheme*** to be followed
- Don't fret if this does not conform to the "**right**" way of doing things;
    - There is no **single** "right" way, but only **alternatives** from which we must choose.
- The ***consistency*** within a project will contribute immensely toward a successful project.
- There are some **ground rules** to establish:
    - **(1)** Define *project's development cycle* and *relate all schedules and activities to that cycle*.
        - When we mention, say, the *Programming Phase,* we may be sure that always mean a certain time slice one can point to on a simple chart, and there are always certain activities associated with that phase. It doesn't matter, of course, what names choose; just be *consistent*.
    - **(2)** Define *activities,* such as *levels of testing*, in a *consistent* way.
        - If there is a universally accepted set of definitions, adopt those that make *sense* to your **organization**, and *stick* with them.
    - **(3)** Define a *system of documents* clearly, consistently, and early. Then hang anyone who operates outside that system.
        - There will be enough paperwork on any project without the headache of random documents that can't be controlled and whose authority is suspect.
    - **(4)** In summary
        - **(a)** *Define* the ***development process*** for your **project**
        - **(b)** *Believe* in it
        - **(c)** *Sell* it to your people
        - **(d)** *Enforce* it

1.5 The Contract

- It's *absolutely necessary*. Half the horror stories about programming involve either bad contracts or no contract at all.
- A ***contract*** is an agreement between you and a customer that you will do a certain job within specific constraints for so much money.
    - Don't operate on the basis of *verbal agreements* or *casual memos*, even if your customer happens to be your buddy down the hall and you both work for the same organization.

    - Within your company, you may call your document a *"letter of understanding"* or something similar, which sounds friendlier than "contract."

- In any case, you need a ***formal written statement*** clearly showing what the **customer wants** and what **you agree** to provide.
    - Operating without such an agreement is lunacy for both parties, as many software project managers and just as many customers have found out.

1.5.1 Contract Template

- Usually a contract have to cover the following **essentials elements**:
- **(1)** *Scope of the work.*
    - What is the *job* to be done?
    - If the job definition is too vague, maybe you need **two contracts:**
        - One to *define* the job
        - One to *write* programs.
- **(2)** *Schedule and deliverables.*
    - *What* specific items (programs, documents) are to be delivered to the customer?
    - *When* are they to be delivered?
    - *Where* are they to be delivered?
    - In *what form* (diskettes, CDs, drafts or clean documents)?
    - How *many* copies?
- **(3)** *Key people.*
    - *Who* is authorized to *approve changes* and *accept* the finished product?
- **(4)** *Review schedule.*
    - *When* and *how* shall the customer be given reviews and reports of progress?
    - *What* is required of the customer if he *disapproves* of a report?
- **(5)** *Change control procedures.*
    - *What* will be the *mechanism* for dealing with items the customer demands, which you consider *changes* to the original work scope?
- **(6)** *Testing constraints.*

    - *Where* and under whose *control* will computer or other test time be obtained?
    - During which work shifts?
    - Exactly what priority will your testers have?
- **(7)** *Acceptance criteria.*
    - What are the specific *quantitative criteria* to be used in judging whether your finished product is *acceptable*?
- **(8)** *Additional constraints.*
    - Are there items which may be *peculiar* to your working environment?
    - Are you to use customer *personnel*? If so, what control do you have over them?
    - Are there special *data security problems*?
    - Is the customer required to supply **test data**? If so, what kinds of data, in what form, when, and how clean?
- **(9)** *Price.*
    - What is your price for doing the job?
    - Is it fixed or variable?
    - If variable, under what circumstances?
- All of these items and more will be addressed in much or less detail in the contract.
- The last item, price, is handled in a good many different ways depending on the type of contract agreed upon.
- Here is a brief summary of formal contract types.

1.5.2 Types of Prices

- **(1) Firm Fixed Price (FFP)**
    - (a) The price *is set* and not subject to change even if you have estimated badly.
    - (b) This is the *most risky* type of contract to use on a programming job.
    - (c) It should *never* be used without at least *a very clear statement of work, no fuzzy areas, no dangling definitions.*
    - (d) Many a project has experienced severe losses operating under such a contract.
- **(2) Fixed Price with Escalation (FP-E)**

- (a) The price is set
- (b) Some allowance is made for both *upward* and *downward adjustments* in case certain things happen, for **example**, *labor rates* or *material costs change*.

- **(3) Fixed Price Incentive (FPI)**
  - (a) A target price is set
  - (b) Some formulas are established that allow the contractor:
    - (1) *A higher percentage of profit* if the contractor exceeds selected targets, (such as *cost)*
    - (2) *A lower percentage of profit* (even a loss) if the contractor misses the selected targets.

- **(4) Cost Shared (CS)**
  - (a) This type of contract reimburses the contractor for part or all of his costs but allows *no profit*, or *fee*.
  - (b) It's used either:
    - (1) For research work with *nonprofit organizations*
    - (2) In joint projects between the customer and the contractor where there is *anticipated benefit* to the contractor.
      - For **example**, the job may result in a product for which the contractor will have the exclusive right to sell.

- **(5) Cost Plus Incentive Fee (CPIF)**
  - (a) The contractor will be paid *all his costs* plus a *fee*
  - (b) The *fee* **varies** depending on:
    - (1) How close the contractor comes to meeting the established target costs
    - (2) How well he does in other areas spelled out in the contract.
  - (c) The **criteria** which determine the fee must be *objective and measurable*

- **(6) Cost Plus Award Fee (CPAF)**.
  - (a) Is similar with CPIF
  - (b) The difference is the **criteria** which are *more subjective* and are weighed by a *Board of Review*.

- **(7) Cost Plus Fixed Fee (CPFF)**
  - (a) The contractor is paid *allowable costs* and a set *fee*.

- **(8) Time & Materials (T&M)**

- (a) The contractor is paid for *labor hours* actually worked and the *cost of materials* used.

- **(9) Labor Hour (LH)**
  - (a) *Labor hours* are paid for, but nothing else.

- **(10) Level of Effort (LOF)**
  - (a) Is similar with Labor Hours type of contract
  - (b) The effort is paid, but nothing else.

- **Conclusions:**
  - (1) The last two contract types (*Labor hours* and *Level of effort*) are pretty much risk-free for the contractor. He provides people to do as the customer directs.
  - (2) The other contract types involve varying degrees of risk for the contractor and the customer.
  - (3) When the deliverable product can be *well defined* in advance, the contractor may propose a ***fixed price*** *and* a ***high fee***.
  - (4) When the end product is *poorly defined* or *subject to change*, a *cost type of contract* is appropriate, from the contractor's point of view. His profit is lower, but so is his risk.

1.6 Customer Rights and Responsibilities

- Software *success* depends on developing a collaborative partnership between software *developers* and their *customers*.
  - Too often, though, the customer-developer relationship becomes **strained** or even **adversarial**.
- Problems arise partly because people don't share *a clear understanding of what requirements are* and *who the customers are*.
- To clarify key aspects of the customer-developer partnership**,** ***Karl Wiegers*** [*] proposes two documents:
  - (1) **Requirements Bill of Rights** for **software customers**
  - (2) **Requirements Bill of Responsibilities** for **software customers**
- Because it's impossible to identify every requirement early in a project, the *commonly used*—and sometimes *abused*—practice of requirements sign-off bears *further examination*.

1.6.1 Who Is the Customer?

- A **customer** is anyone who derives direct or indirect benefit from a product. This includes:
    - (1) People who request, pay for, select, specify or use a software product, or
    - (2) Those who receive the product's outputs.
- There are **different types** of customers:
- **(1)** Customers who *initiate* or *fund* a software project.
    - They supply the high-level product concept and the project's business rationale.
    - These **business requirements** describe the **value** that the *users*, *developing organization* or other *stakeholders* want to receive from the system.
- **(2)** Customers who will *actually use* the product. **Users** can describe:
    - The tasks they need to perform—the *use cases*—with the product
    - The product's desired *quality attributes*.
    - **Analysts** interact with customers to gather and document requirements and derive specific *software functional requirements* from the user requirements.
    - There are different possible **scenarios**:
        - Customers often feel they don't have time to participate in requirements elicitation.
        - Sometimes customers expect developers to figure out what the users need *without* a lot of discussion and documentation.
        - Development groups sometimes exclude customers from requirements activities, believing that they already know best, will save time or might lose control of the project by involving others.
        - It's not enough to use customers just to answer questions or to provide selected feedback *after* something is developed.
        - *Proactive* customers will insist on being partners in the venture.
- **(3)** For ***commercial software*** development, the customer and user are often the same person.
    - Even for commercial software you should get *actual users* involved in the requirements-gathering process, perhaps through focus groups or by building on your existing beta testing relationships.
- **(4) Customer surrogates**, such as the marketing department, attempt to determine what the *actual customers* will find appealing.

1.6.2 The Customer-Development Partnership

- **Quality software** is the product of a *well-executed design* based on *accurate requirements*, which are in turn the result of effective **communication and collaboration,** a real ***partnership*** between **developers** and **customers**.
- **Collaborative efforts** only work *when*:
  - (1) All parties involved know *what they need to be successful*
  - (2) They understand and respect *what their collaborators need to succeed*.
- As project pressures rise, it's easy to forget that **everyone shares** a ***common objective***:
  - *To build a successful product that provides business value, user satisfaction and developer fulfillment.*
- *Figure 1.6.2.a.* [Wi99] presents a Requirements Bill of Rights for Software Customers
  - There are10 ***expectations*** which **customers** can place on their interactions with **analysts** and **developers** during requirements engineering.
  - Each of these ***rights*** implies a corresponding *software developer's* ***responsibility*****.**

Figure 1.6.2.a. A Bill of Rights for Software Customers

As a software customer, you have the right to:

1. Expect analysts to speak your language.
2. Expect analysts to learn about your business and your objectives for the system.
3. Expect analysts to structure the requirements information you present into a software requirements specification.
4. Have developers explain requirements work products.
5. Expect developers to treat you with respect and maintain a professional attitude.
6. Have analyst present ideas and alternatives both for your requirements and for implementation.
7. Describe characteristics that will make the product easy and enjoyable to use.
8. Be presented with opportunities to adjust your requirements to permit reuse.
9. Be given good-faith estimates of the costs and trade-offs when you request a change.
10. Receive a system that meets your functional and quality needs, to the extent that those needs have been communicated to the developers and agreed upon.

- *Figure 1.6.2.b* [Wi99] proposes 10 ***responsibilities*** the **customer** has to the **developer** during the requirements process.
  - These rights and responsibilities apply to ***actual user representatives*** for internal corporate software development.
  - For mass-market product development, they apply more to ***customer surrogates***, such as the marketing department.

Figure 1.6.2.b. A Bill of Responsibilities for Software Customers

As a software customer, you have the responsibility to:

1. Educate analysts about your business and define jargon.
2. Spend the time to provide requirements, clarify them, and iteratively flesh them out.
3. Be specific and precise about the system's requirements.
4. Make timely decisions about requirements when requested to do so.
5. Respect developers' assessments of cost and feasibility.
6. Set priorities for individual requirements, system features, or use cases.
7. Review requirements documents and prototypes.
8. Promptly communicate changes to the product's requirements.
9. Follow the development organization's defined requirements change process.
10. Respect the requirements engineering processes the developers use.

- Early in the project, customer and development representatives should *review* these two lists and reach a meeting of the minds.
    - If you encounter some sticking points, ***negotiate*** to reach a clear understanding regarding your responsibilities to each other.
    - This understanding *can reduce friction later*, when one party expects something the other isn't willing or able to provide.
    - These lists aren't all-inclusive, so *feel free to change* them to meet your specific needs.

1.6.3 Customer Rights

- ***Right #1: To expect analysts to speak your language.***
    - *Requirements* discussions should center on your *business needs* ***and*** *tasks*, using your *business vocabulary* (which you might have to convey to the analysts).
    - You shouldn't have to wade through *computer jargon*.

- ***Right #2: To expect analysts to learn about your business.***
    - By *interacting* with **users** while eliciting requirements, the **analysts** can better understand your *business tasks* and how the *product fits* into your world.
    - This will help **developers** design software that truly meets your *needs*.
    - Consider inviting **developers** or **analysts** to *observe* what you do on the job.
    - If the new system is replacing an *existing application*, the **developers** should use the system as you do to see how it works, how it fits into your workflow, and where it can be improved.

- ***Right #3: To expect analysts to write a Software Requirements Specification (SRS).***
    - The analyst will sort through the customer-provided information, separating *actual user needs* from other items, such as:
        - *Business requirements and rules*

        - *Functional requirements*
        - *Quality goals*
        - *Solution ideas*.
    - The analyst will then write a *structured* ***Software Requirements Specification***, which constitutes an agreement between **developers** and **customers** about the proposed product.
    - *Review* these specifications to make sure they accurately and completely represent your *requirements*.

- ***Right #4: To have developers explain requirements work products.***
    - The analyst might represent the requirements using various *diagrams* that complement the *written SRS*.
        - These graphical views of the requirements express certain aspects of system behavior more clearly than words can.
    - Although unfamiliar, the *diagrams* aren't difficult to understand.
    - The analysts should:
        - (1) Explain the *purpose* of each diagram
        - (2) Describe the *notations* used
        - (3) Demonstrate how to examine it for *errors*.

- ***Right #5: To expect developers to treat you with respect.***
    - Requirements discussions can be *frustrating* if users and developers don't understand each other.
    - Working *together* can open each group's eyes to the problems the other faces.
    - Customers who participate in requirements development have the right to have developers treat them with *respect* and to *appreciate the time* they are investing in project success.
    - Similarly, demonstrate *respect* for the developers as they work with you toward your common objective of a successful project.

- ***Right #6: To have analysts' present ideas and alternatives for requirements and implementation.***
    - The analysts should explore *ways* your *existing systems* don't fit well with your *current business processes*, to make sure the new product doesn't automate ineffective or inefficient processes.
    - Analysts who thoroughly understand the application domain can sometimes *suggest improvements* in your business processes.
    - An experienced and creative analyst also adds value by *proposing valuable capabilities* the new software could provide that the users haven't even envisioned.

- ***Right #7: To describe characteristics that will make the product easy and enjoyable to use.***

- The analyst should ask you about *characteristics* of the software that go beyond your functional needs.
- These *"quality attributes"* make the software *easier* or *more pleasant* to use, letting you accomplish your tasks accurately and efficiently.
    - For **example**, customers sometimes state that the product must be "*user-friendly"* or *"robust"* or *"efficient,*" but these terms are both subjective and vague.
- The analyst should *explore and document* the specific characteristics that signify "*user-friendly*," "*robust*," or "*efficient*" to the users.

- ***Right #8: To be presented with opportunities to adjust your requirements to permit reuse.***
    - The analyst might know of *existing software components* that come close to addressing some need you described.
    - In such a case, the analyst should give you a chance to *modify* your requirements to allow the developers to *reuse* existing software.
    - Adjusting your requirements when sensible *reuse opportunities* are available can save time that would otherwise be needed to build precisely what the original requirements specified.

- ***Right #9: To be given good-faith estimates of the costs of changes.***
    - People sometimes make *different choices* when they know one alternative is more expensive than another.
    - Estimates of the *impact and cost* of a proposed requirement change are necessary to make *good business decisions* about which requested changes to approve.
    - Developers should present their **best** *estimates of impact, cost, and trade-offs*, which won't always be what you want to hear.
    - Developers must not inflate the *estimated cost* of a proposed change just because they don't want to implement it.

- ***Right #10: To receive a system that meets your functional and quality needs.***

- **Conclusion:**
    - This desired **project outcome** is achievable only if:
        - You as **customer** clearly communicate *all the information* that will let developers build the product that satisfies your needs,
        - **Developers** communicate *options* and *constraints*,
        - You *state* any assumptions or implicit expectations you might hold; otherwise, the developers probably can't address them to your satisfaction.

1.6.4 Customer Responsibilities

- ***Responsibility #1: To educate analysts about your business.***
  - **Analysts** depend on you to educate them about your *business concepts and terminology*.
  - The intent is not to transform analysts into domain experts, but to help them *understand* your problems and objectives.
  - Don't expect analysts to have knowledge you and your peers take for granted.

- ***Responsibility #2: To spend the time to provide and clarify requirements.***
  - You have a responsibility to invest time in *workshops*, *interviews* and other *requirements elicitation activities*.
  - Sometimes the analyst might think he *understands* a point you made, only to realize later that she needs further *clarification*.
  - Be patient with this *iterative approach to developing and refining the requirements*, as it is the nature of complex human communication and essential to software success.

- ***Responsibility #3: To be specific and precise about requirements.***
  - Writing *clear, precise requirements* is hard
  - It's tempting to leave the requirements *vague*, because *pinning down details* is tedious and time-consuming.
  - At some point during development, though, someone must resolve the *ambiguities* and *imprecision*.
    - You are most likely the best person to make those decisions; otherwise, you're relying on the developers to *guess* correctly.
  - Do your best to *clarify* the intent of *each requirement*, so the analyst can express it accurately in the Software Requirement Specification.
  - If you can't be precise, agree to a process to generate the necessary precision, perhaps through some *type of prototyping*.

- ***Responsibility #4: To make timely decisions.***
  - The analyst will ask you to make many *choices* and *decisions*.
  - These decisions include resolving *inconsistent requests* received from multiple users and making *trade-offs* between conflicting quality attributes.
  - Customers who are authorized to make such decisions *must do so promptly* when asked.
  - The developers often can't proceed *until you render your decision*, so time spent waiting for an answer can delay progress.
  - If customer decisions aren't forthcoming, the *developers* might make the decisions for you and charge ahead, which often won't lead to the outcome you prefer.

- ***Responsibility #5: To respect a developer's assessment of cost and feasibility.***

- All software functions have a price, and **developers** are in the best position to *estimate those costs*.
- Some features you would like might not be *technically feasible* or might be surprisingly *expensive* to implement.
- The developer can be the bearer of bad news about *feasibility or cost*, and you should respect that judgment.
- Sometimes you can *rewrite requirements* in a way that makes them *feasible or cheaper*.
    - For **example**, asking for an action to take place "instantaneously" isn't feasible, but a more specific timing requirement ("within 50 milliseconds") might be achievable.

- ***Responsibility #6: To set requirement priorities.***
    - Most projects don't have the *time or resources* to implement every desirable bit of *functionality*, so you must determine
        - (1) Which features are essential,
        - (2) Which are important to incorporate *eventually*,
        - (3) Which would just be nice extras.
    - **Developers** usually can't determine priorities from your perspective, but they should estimate the *cost and technical risk of each feature, use case, or requirement* to help you make the *decision*.
    - When you *prioritize*, you help the **developers** deliver the greatest value at the lowest cost.
    - No one likes to hear that something he or she wants can't be completed within the project bounds, but that's just a reality.
    - A *business decision* must then be made to reduce project scope based on *priorities,* or to extend the schedule, provide additional resources or compromise on quality.

- ***Responsibility #7: To review requirements documents and prototypes.***
    - Having customers participate in *formal and informal reviews* is a valuable quality control activity.
        - It's the **only way** to evaluate whether the requirements are complete, correct and necessary.
    - It's difficult to *envision* how the software will actually work by reading a specification.
    - To better understand your needs and explore the best ways to satisfy them, developers often build prototypes.
        - Your *feedback* on these preliminary, partial or possible implementations helps ensure that everyone understands the requirements.
        - Recognize, however, that a *prototype* is not a *final product*;
        - Allow developers to build fully *functioning systems* based on the prototype.

- ***Responsibility #8: To promptly communicate changes to the product's requirements.***
    - Continually *changing requirements* pose a serious risk to the development team's ability to deliver a *high-quality product* within the *planned schedule*.
    - *Change* is inevitable, but the later in the development cycle a change is introduced, the greater its impact.
    - Extensive *requirements changes* often indicate that the *original requirements elicitation* process wasn't adequate.
    - Changes can cause expensive rework and schedules can slip if new functionality is demanded after construction is well under way.
    - Notify the analyst with whom you're working as soon as you become aware of any change needed in the requirements.
    - Key *customers* also should participate in the process of deciding whether to approve or reject change requests.

- ***Responsibility #9: To follow the development organization's requirements change process.***
    - To minimize the negative impact of change, *all participants* must follow the *project's change-control process*. This ensures that:
        - (1) Requested changes are not lost,
        - (2) The impact of each requested change is evaluated,
        - (3) All proposed changes are considered in a consistent way.
        - (4) As a result, you can make good business decisions to incorporate certain changes into the product.

- ***Responsibility #10: To respect the requirements engineering processes the developers use.***
    - Gathering *requirements* and verifying their *accuracy* are among the greatest challenges in software development.
    - Although you might become frustrated with the process, it's an excellent investment that will be less painful if you *understand* and *respect* the techniques analysts use for gathering, documenting, and assuring the quality of the software requirements.
    - Customers should be *educated* about the requirements process, ideally attending classes together with developers.
        - An efficient modality is to present seminars to audiences that included developers, users, managers, and requirements specialists.
        - People can *collaborate* more effectively when they learn together.

1.6.5 What About Sign-Off?

- Agreeing on a *new product's requirements* is a critical part of the *customer-developer partnership*.

    - Many **organizations** use the act of signing off on the *requirements document* to indicate *customer approval*.
    - All participants in the requirements approval process need to know exactly what sign-off means.
- **(1)** One potential problem is the **customer** representative who regards signing off on the requirements as a *meaningless ritual*
    - *"I was given a piece of paper that had my name printed beneath a line, so I signed on the line because otherwise the developers wouldn't start coding."*
    - This attitude can lead to future conflicts when that customer wants to change the requirements or when he's surprised by what is delivered:

        "Sure, I signed off on the requirements, but I didn't have time to read them all. I trusted you guys—you let me down!"
- **(2)** Equally problematic is the **development manager** who views sign-off as a way to ***freeze the requirements***.
    - Whenever a change request is presented, the development manager can point to the *Software Requirements Specification* and protest

        "You signed off on these requirements, so that's what we're building. If you wanted something else, you should have said so."
- Both of these attitudes fail to acknowledge the reality that
    - (1) It's impossible to know all the requirements early in the project
    - (2) The requirements will undoubtedly change over time.
- Requirements sign-off is an appropriate action that *brings closure* to the requirements development process.
    - However, the participants have to agree on precisely what they're saying with their signatures.
- More important than the sign-off ritual is the concept of *establishing a "baseline" of the requirements agreement,* a snapshot at some point in time.
- The subtext of a signature on an *Software Requirements Specification* sign-off page should therefore read something like:
    - (1) I agree that this document represents *our **best** understanding* of the requirements for the project today.
    - (2) *Future **changes*** to this baseline can be made through the *project's defined change process*.
    - (3) I realize that *approved changes* might require us to **renegotiate** the *project's costs, resources and schedule commitments*.
- A *shared understanding* of the requirements approval process should alleviate the friction that can arise as the project progresses and requirements oversights are revealed, or as marketplace and business demands evolve.
- Sealing the *initial requirements development* activities with such an explicit agreement helps you forge a *continuing customer-developer partnership* on the way to project success.

1.7 Top-Down Development

- Traditionally, most programs have been analyzed and designed *top-down*, and coded and tested from the *bottom up*.
- During analysis and design it seemed natural to start by first considering the system as a whole and then to *break* it down into smaller and smaller pieces which individuals could handle.
- Then the pieces were coded and tested, and combined ("integrated") into increasingly larger and more complex groupings until finally the entire system had been assembled from the bottom up.
- Many systems are still being built that way. But the *trend* is toward complete top-down development, wherein not only analysis and design are attacked from the top, but so are coding and integration testing.

1.8 An ideal project

- The paragraph describes briefly a well-run and successful project unfortunately, not so typical.

- **(1) Proposal stage**
  - A **programming project** begins with an idea some user has about a need the computer might handle.
  - The user solicits *ideas* from **associates** and **contractors** about the reasonableness of the idea and possible embellishments.
  - After some incubation and revision, the now firmer idea is *formally submitted*, usually to competing contractors, for ***bids***.
  - The **competitors** start a feverish activity called ***proposal writing***.
    - Each tries to figure how he can meet this user's needs at lower cost and with better quality than the others are likely to propose.
    - Each writes a statement of his understanding of the problem and how he would solve it with computer programs.
    - Each adds a layer of boilerplate to try to *impress* the customer with his credentials
  - The proposals are submitted for ***evaluation***.
  - Of the contenders (competitors) considered responsive to his needs, the customer *selects one*, usually the lowest bidder, to do the job.
  - If none are responsive enough, he *redefines* the requirements and asks for new proposals.
  - The winner celebrates his good fortune while the losers applaud, and the project begins.
  - The winner appoints a ***project manager*** who organizes a ***team*** (partly kept in readiness since the proposals were first submitted) to do this job.

- **(2) Definition, Planning**
  - The team tackles two immediate *tasks*:
    - (1) To *define* in clear detail the **customer's needs**,

        - (2) To *write* a **plan** for filling those needs.
            - Both tasks were "done" during the proposal stage, but now they must be refined.
    - A very precise, **structured problem description** document must be written to serve as the *baseline* for subsequent design and programming;
    - A **detailed project's plan**, minus the public-relations boilerplate, must be *written* to guide all the remaining phases of the project.

- **(3) Team selection, Design**
    - Now, the PM *(Project Manager)* must *recruit* and *organize* the talent needed for the next phase: designing the program system.
    - PM selects the very best *designers* he can, including some of the *analysts*, and directs them to design the *best possible architecture for the program system* to match the problem defined by the analysts.
    - While design is going on, the Project Manager is busy *recruiting* people and *finding* other resources for the remaining work to be done.
    - PM *keeps* his eye on the project's plan and takes steps to meet all the *milestones* stated in it.
    - Sometimes PM sees a need to *change* the plan, and he does so.

- **(4) Specification approval**
    - When the overall program system design is *ready*
        - It's reviewed and approved by **project management** and the **customer**
        - It's established as the **baseline** for detailed design and coding
        - It's turned over to the **programmers**

- **(5) Detailed Design, Programming**
    - Programmers further *refine* the baseline design into smaller pieces, until the refinements reach the level of actual code.
    - The pieces ("*modules*") are coded and tested and carefully merged ("integrated") with one another in a preplanned manner.
    - As modules are added successfully, the *program system* grows in complexity and usefulness.
        - It reaches a number of releases where it can be shown to be performing some subset of its intended functions.
    - Because design integrity was sought and achieved in earlier phases, the *system* fits together well.
    - In all probability there will be analysis, design, or coding mistakes.
    - *Changes* are made as necessary, but they are strictly controlled through a simple *mechanism* earlier planned for.

- **(6) Documentation, Testing**

- Finally, the *program system* is ready for the **customer**, along with its set of *descriptive documentation* and a set of draft *user documentation*.
- But the customer doesn't yet get his hands on the product.
  - First it's wrung out through another set of tests called *"system tests."*
  - To assure integrity and objectivity, these tests are devised and conducted by a separate group rather than by the programmers.
  - This group imagines itself the **user** and tries to "raise hell" with the system to make it fail.
  - It will fail, but only in trifling ways because the *requirements* were well analyzed and the system well designed to meet those requirements.
- *Changes* are made to correct the problems found, and finally a cleanly compiled system, complete with clean documentation, is ready for delivery.

- **(7) Systems Acceptance**
  - Now the system is demonstrated to the **customer**, probably with his *direct involvement*, in order to win his formal acceptance.
  - The terms of acceptance are not subjective; they were established and agreed to *early* in the life of the project.
  - All that's needed now is to show that the programs meet those criteria earlier agreed to.

- **(8) Delivery**
  - Once accepted, the system is *delivered* to the customer.
  - If it was not possible or feasible to do acceptance testing at his **installation** under live conditions, there may be still *another set of tests*.
  - At the conclusion of these on-site tests, the project is finished, except for any agreed follow-on work to help maintain or improve the system.
  - In the case of large systems, the next versions of the system may then be built.

- **(9) End of the Project**
  - Now the *project manager* :
    - (1) Writes a history of the project's activities
    - (2) Makes a *comparison* between what was originally planned and what actually took place.
    - (3) Then promotes everybody and goes home to get acquainted with his family.

1.9 Project Lifecycle

- The *central problem* with so many failed projects is loss of control because things are not kept ***visible*** enough.

    - The **requirements** are often invisible, or at least obscure, because we don't take the time to make them explicit.
    - **Design** and **code** are often invisible because, if they exist at all, they're carried around in people's heads and on private listings or scraps of paper.
- One of the thrusts of the newer programming technologies is to make *each stage* of the emerging program system **visible** and **available** for all to see.
- But that's not enough.
    - The *project itself* needs to be visible.
    - It must not become an amorphous collection of people, documents, and activities.
    - It must be *divided* into pieces you can get your arms around, just as a program must be divided into people-sized chunks.
    - Make your project manageable; make it ***modular***.
- The way to make your project modular is by providing a **framework** called a ***development cycle*** and breaking it into a **series of modules** called ***phases***.
    - Dream up any number of phases you want, as long as they enable you to see and exert *control* over your project.
- What's important is that each phase to have a very clear *set of objectives* and *definable outputs* so that all those you deal with, understand your planned development cycle completely.
- The *classical generic* development cycle consists of the following phases *(waterfall lifecycle)* (Fig. 1.9.a.)
    - **(1) Definition Phase**
    - **(2) Design Phase**
    - **(3) Programming Phase**
    - **(4) System Test Phase**
    - **(5) Acceptance Phase**
    - **(6) Installation and Operation Phase**
- In the fig.1.9.a the phases are depicted as *vertical slices of time*, implying that one phase ends and the next begins, all at some instant in time.
    - This is unrealistic, that in practice the phases will overlap to some degree.
    - Although this will sometimes be true, your *aim* should be to begin a phase only when the preceding phase has been *satisfactorily completed*.

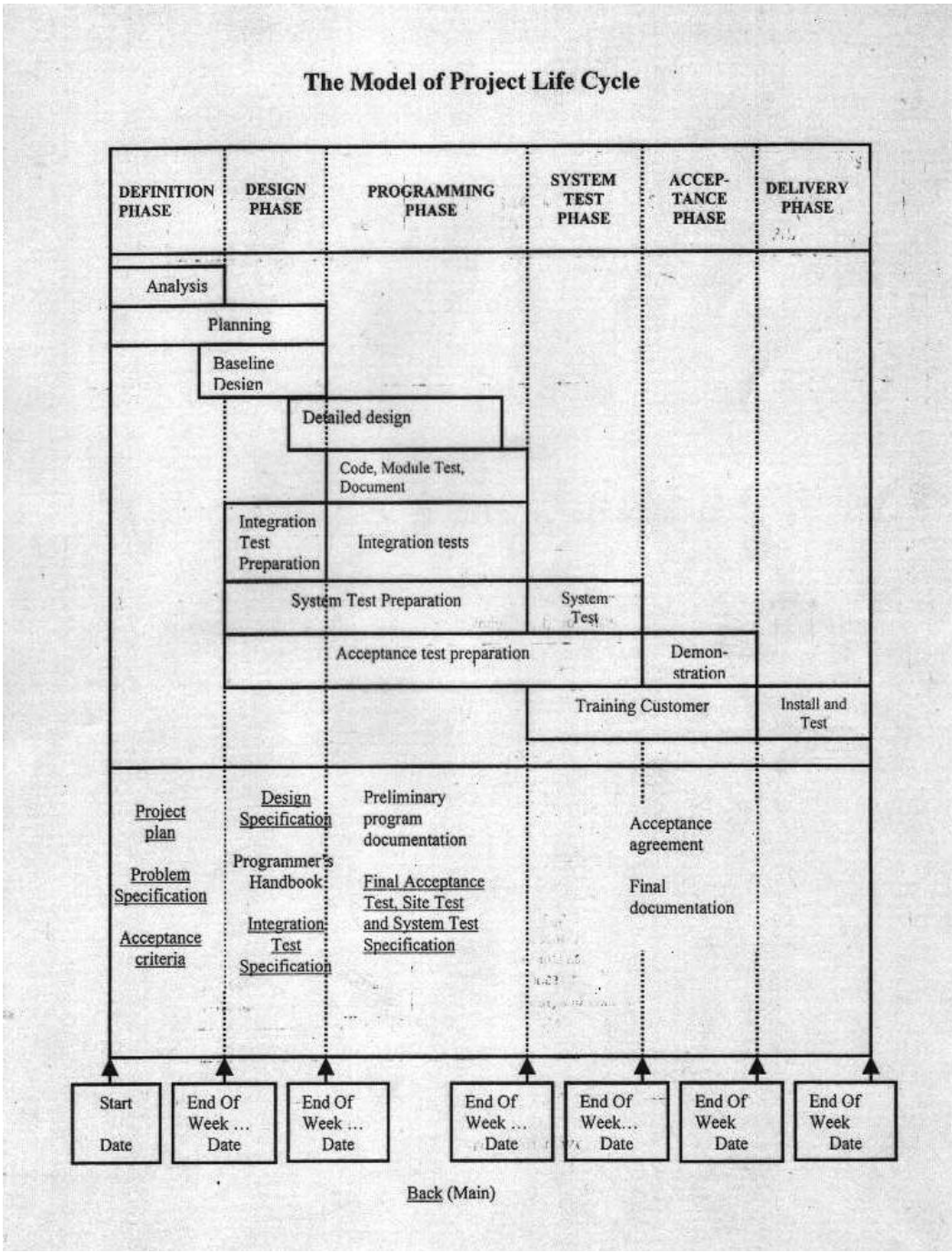

Fig.1.9.a. The Model of Project Life Cycle

- Fig.1.9.a shows a typical division of total project time among the phases. *There may be very large departures from this timing for some projects.*
    - It's easily possible for a Definition Phase to consume a *third* of the total time.
    - On a large defense project, the last two phases could take *half* the project's time.
    - The parts of the development cycle most often short-changed are the *front end* and the *rear end*.
        - (1) On the *front end*, planning is often haphazard (let's get writing programs), analysis is weak (we all understand the customer's problem), and baseline design is nonexistent.

    - (2) On the *rear end*, system testing is sometimes not even included in a plan (there's no time left, and anyway, the programmers' integration test does the same job).
- There is no reliable standard for *time allocation*.
    - Experience with similar projects is the best guide.
- In general, most projects will not be far wrong allowing *one-third* of total calendar time for the Definition and Design Phases, *one-third* for the Programming Phase, and *one-third* for the rest.
- Figure 1.9.b summarizes the **phases**, their **activities**, and important associated **documents**.

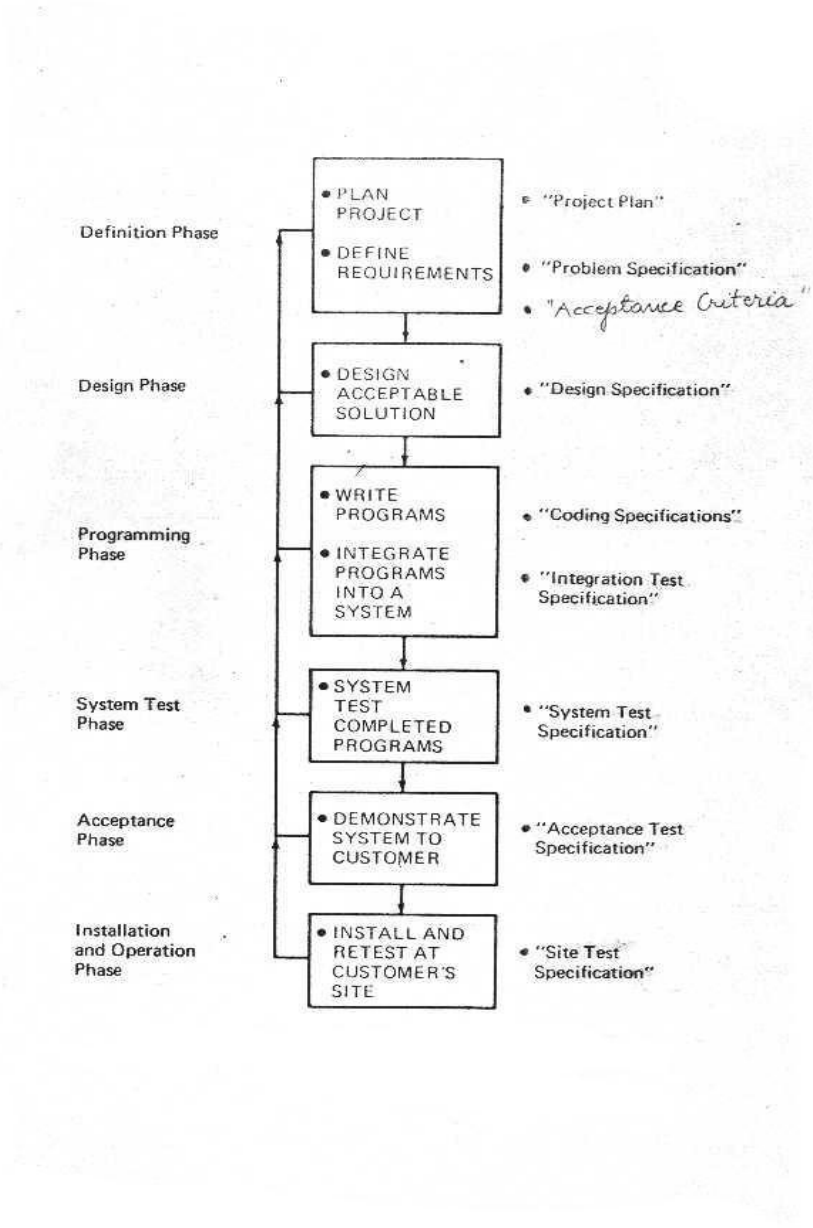

Fig.1.9.b. Phases, functions and key documents

- **(1) Definition Phase** –
    - During this phase, a plan for the project is written and the customer's *problem* is defined.
    - During the problem definition activity, ideas about *solutions* will inevitably be discussed, but adoption of any specific solution is deferred until the Design Phase.

- **(2) Design Phase** –
    - Now that you and the customer have agreed on what the problem is, write a design document describing the *architecture* of an acceptable solution to the problem.
    - Usually many solutions are feasible, but you and the customer must pick one and stick with it.

- **(3) Programming Phase –**
    - You've defined the problem and blueprinted a solution; now *build* and *test* a program system according to that blueprint.

- **(4) System Test Phase –**
    - After the programmers have built a product they're happy with, a separate group performs a *new set of tests* in as nearly a "live" environment as possible.

- **(5) Acceptance Phase –**
    - The finished program system, including its documentation, is *demonstrated* to the customer in order to gain his formal agreement that the system satisfies the contract.
    - *Acceptance* is based on *meeting criteria* that you and the customer agreed to earlier in the development cycle.

- **(6) Installation and Operation Phase –**
    - The accepted programs are introduced into their ultimate *operating environment* on the customer's equipment, retested in that environment, and then *put into operation*.

1.10 Some Key Documents

- There are three **key technical documents** (Fig.1.10.a)

DOCUMENT NAME	WHEN WRITTEN	WHAT IT DOES	WHO WRITES	SOME POSSIBLE FORMS
"Problem Specification"	Definition Phase	Defines the problem for which a solution is needed	Analysts	Narrative HIPO Tables Data flow diagrams Data dictionaries
"Design Specification"	Design Phase	Describes the overall solution	Designers	Narrative HIPO Tables Flow charts
"Coding Specification"	Programming Phase	Describes the detailed solution	Programmers	HIPO Pseudo code Procedure charts Code Flow charts

Fig.1.10.a. Key technical documents

  - **(1) Problem Specification** is the document your analysts produce describing the *customer's problem*. It defines the *requirements* of the job to be done.
  - **(2) Design Specification** is written during the Design Phase. It describes the *architecture* of the overall solution to the problem.
  - **(3) Coding Specification**, is the detailed extension of the Design Specification

- This is the really a *set* of documents describing from **technical** point of view the *program system* in detail.
- There are also a set of **management documents** gathered together in the **Project Plan**
- It's very important that the *Project Manager* to develop a Documentation System in order to manage the project documents.

1.10.1 Document Testing

- Get used to the idea of *testing your documents* thoroughly, just as you test your programs.
- Documents such as those discussed in the preceding section, as well as *user's manuals, test specifications,* and so forth, are as critical to a **successful project** as anything you'll produce.
- A good way **to test a document** is to *submit* it to close scrutiny by others during a "structured walk-through".
  - Don't simply pass a document around for comment; it's too easy for readers to be lazy and assume that the next reader will be more thorough.
- There are ***two criteria*** for testing your documents:
  - (1) **First**, they must be *complete* and *absolutely accurate*.
  - (2) **Second**, they must be *readable* and *easily understood*.

- Your **documents** represent your **product** to the user; they're *tangible*, *visible*, while the *programs* are not. Making them easy to read is *as important as any job on your project*.

Summary

2. SW Project Management Exercise #4

1. What is the definition of the term *Project management*?
2. Describe the key elements of the *Project Management Framework*: *stakeholders, knowledge areas, tools and techniques*.
3. Explain the following terms: *manager, first-level manager, second level-manager, program, software*.
4. What are the *ground rules* for a SW project? Describe their meaning.
5. What is a *contract*? What does a *contract template* contain? What is the meaning of *sign-off*? What *type of contract* from the *price* point of view do you know? Describe them.
6. Describe the main *customer rights* and the main *customer responsibilities* in the relation with a contractor.
7. Describe the development of an *ideal project.*
8. Describe a *Generic Life Cycle* for a SW Project. Emphasize the content of the main *phases.*
9. Which are *the main activities* and *documents* for a SW Project development?