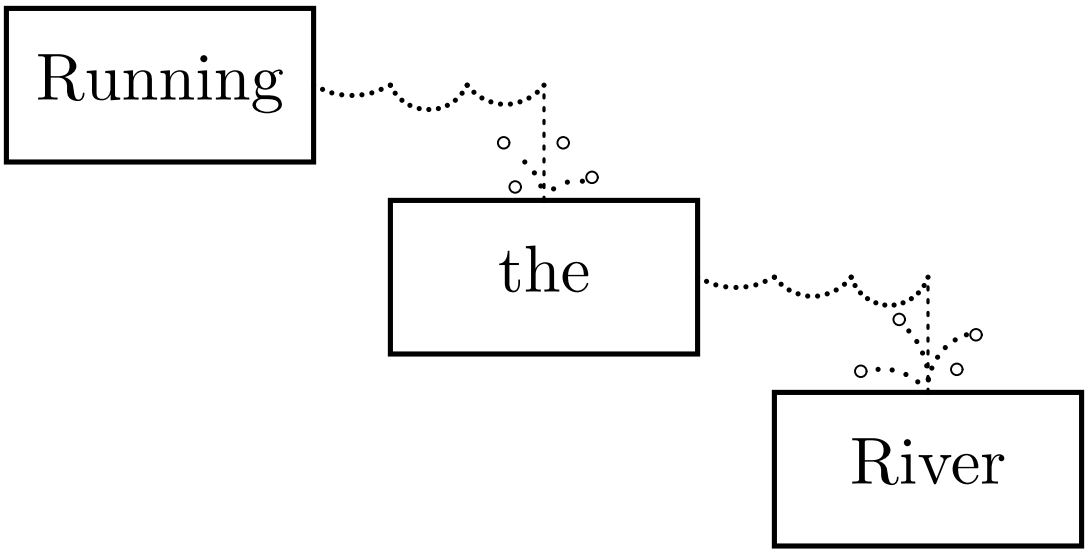


A Survival Guide for the Waterfall Model

Edward Robertson

Copyright ©2007, Edward Robertson

® Trademarks and tradenames used in this *Guide* include Access, D-BASE, emacs, Foxbase, Foxpro, IBM, Microsoft, Paradox, Post-it Notes, Sapphire, TEX, Unix, and Visual Basic.

Acknowledgement: The original draft of this *Guide* was written with the assistance of Lincoln Carr, Jeffery Evans, Vivek Gupta, Jay Hakim, and Arijit Sengupta, all students in the course *Software Engineering Managment* during the 1993-94 acadmic year.

Table of Contents

Preface

This *Guide* was developed to help chart the route towards completion of your Information Systems project. The *Running the River* metaphor fits because a river guide cannot prevent you from being overturned or sucked under; it can only warn you about the waterfalls and whirlpools you will encounter.

The target for this *Guide* is the course sequence offered to undergraduates as *Software Engineering for Information Systems* (P465/6) and to graduate students as *Software Engineering* (P565/6). This course sequence, as the rather awkward name give to its undergraduate manifestation indicates, covers a mixture of software engineering and database issues. Until 1996 there was once course sequence, *Information Systems* (C445/6), which covered basic aspects of databases and software engineering. Because the volume of material had become unmanageable, the material relating to database concepts was split off under that title as B461 and the graduate version *Advanced Database Concepts* (B561). The material remaining in the sequence covers all of the computer science side of information systems[1] while there is much in the bigger picture of software engineering that gets short shrift. Thus the title "Information Systems" is still most suitable for the entire package and is so used in this *Guide*. We also use the singular term "course" for either sequence, since the material is an integral unit.

This *Guide* was begun by students in the Software Engineering Management (currently numbered C665/6) course during the Spring Semester, 1994. Each student in the Software Engineering Management class had experienced the Information Systems sequence twice: once as a team member and once as a Supervisor for project teams. Thus they have faced all of the difficulties you are about to encounter.

The Information Systems milestones and associated process methodology are those used in professional software projects in a commercial setting. They were not adopted because of this widespread use but because they will successfully lead you through the phases of a software engineering lifecycle. This is likely the first time you have had to follow such rigorous and detailed process, with the attendant requirement for mountains of documentation. This *Guide* attempts describe the various aspects of the process and their purposes, along with the documentation required at each milestone and its purpose. Understanding both the "why" and the "how" of these various components will make your life a little easier and your project more successful.

No single document can address the multitude of issues that arise during the Information Systems projects. So this *Guide* cannot give definitive answers to all questions. It is not an answer key or a programmer's manual. It is, however, a planning guide and a reference, providing a general framework and suggesting profitable ways for thinking about issues.

1. Many disciplines teach information systems courses. Computer science tends to deal with the specification and construction of information systems while business deals with is management aspects and information science with its content, although the three are thoroughly intertwined.

While discussing the variety , it is worthwhile to relate this *Guide* to the *Course Packet*[26]. The *Course Packet* is in fact a detailed syllabus for this course (recall the convention just made, that "course" refers to both sequences P465/6 and P565/6), most importantly a schedule of material covered and assignments due. Returning to the river metaphor, the *Course Packet* says "be at the campsite by sundown", this *Guide* tells you how to arrive their safely. The *Course Packet* is specific to a particular academic year calendar, this *Guide* covers principles that not only cover all offerings of the course but are applicable is any system or software engineering endeavor.

Remember the instructors' disclaimer: "neither this *Guide* nor any other material absolves you from using intelligence and common sense." If you do something inappropriate for your particular project, claiming that the "the *Guide* told me to do it," you will get neither sympathy nor relief.

Finally, note that the Information Systems project has two distinct and often contradictory goals: it must provide a valuable educational experience and a workable software system. This means that you will be given the valuable opportunities to learn by making mistakes while still being monitored to avoid major disruptions.

I. Introduction

This chapter is a brief overview of the Information Systems project – its goals and overall structure. It also contains some supporting material for reading the rest of this *Guide*.

1. Why do a project and not just programs?

The goal of this course, as a software engineering endeavor, is to give you experience in software system development – experience of far greater size and scope than programming assignments in your previous courses. Since you're in the Information Systems class, you have a strong background in programming. You've twisted data structures inside-out in C++ or Java and done Scheme programming which defies description. These assignments have been well-defined, have been of limited size, and have always been done when they are handed in. All of those conditions are now removed: the project will be flagrantly ill-defined, it will be not only large but also ever-growing, and its useful life begins when you are done with it. This situation is important because it is "real world," but even more it is important because it forces a disciplined approach to software specification, design, and development and concern about the the reliability and maintainability of the final product. This is sometimes called "*programming in the large*".

This course shares a common goal with its companion Database Concepts courses (B461 or B561), namely is to teach the conceptual foundation of database systems and the application of these concepts in the design and implementation. Although there are many commercial products which aid database implementation in areas such as user interfaces and which do extremely well with simple database structures, these tools do not scale-up to development of larger systems. So the development of large information systems still must be based on an understanding of database principles and guided by informed, careful design.

The goals of experience with programming in the large and practice in specification and design will be met through a project whose objective project is to develop a substantial information system for a client with a significant need, taking that project from proposal to installation. As noted above, this project is very different from your previous course work. Additionally, you will not be doing this yourself, but in a group of three to five people. Dealing with a real project for real clients requires documentation, an important facet of the programming in the large experience. You will write specifications of the client's needs and document your design decisions and implementation strategies. In addition, you will test your system thoroughly and follow coding standards, making your project maintainable.

In summary, the project will give you experience and discipline in specifying, designing, implementing, testing, and documenting a full blown application for a client. Definitely a real world experience with many, many real world problems.

2. Overview of the Project

A. The Focus on Process

The most noticeable characteristic of information systems projects for this class, in contrast to programming assignments in your earlier classes, is the formal structure of the project process. This structure includes a strict schedule of project phases with associated deliverables (also known as "*milestones*") and close attention given to the process itself. This attention is especially evident in the weekly reports by which each team assesses its progress.

The focus on process gives rise to a situation known as the *product versus process* dichotomy,[1] an important issue in all software development projects. This is the tension between focusing on the final software product and focusing on the process used to develop that product. Staff directly involved in development want to get on with the task – they are programmers after all – and tend to resent the time spent on process matters.[2] However, in your professional career, as a programmer, you must appreciate the need for process standards and monitoring. When you become an manager, you will need to motivate your programmers to follow formal processes and monitor their processes and you will help define process and monitoring mechanisms that are effective yet not overly intrusive.

A complex process has a natural structure deriving from the progression of different activities required for its completion - this progression defines the phases of the project. These phases are then formalized and codified with procedural guidelines and milestones. There are many reasons to formalize the structure of the process:

subdivide tasks/allocate resources/schedule
monitor/control
differing activities/tools
communication

The formal project structure used in this course is known as the *waterfall lifecycle* (also called *waterfall model*). Both the title of this guide and the graphic used on its cover are jocular allusions to the waterfall lifecycle. The phases and milestones of the waterfall lifecycle, along with a rough characterization according to the questions to be answered by each phase, are:

phase	**milestones**	**questions**
Preliminary	Proposal	why, for whom
Analysis	Requirements Specification	what
	Project Plan	when, by whom
	Prototype	
Design	Preliminary Design	how
	Detailed Design	
	Test Plan	
Implementation	Coding	

1. The Terminology section which follows, §I.3 has a further discussion about the uses of the word "process."
2. This dichotomy, as tension between programmer and manager, is humorously but bitingly explored in many Dilbert cartoons[1].

	Test Log
	Installation
	User's & Programmer's Manuals
Maintenance	*start all over*

The waterfall lifecycle, in its many variations, is by far the most commonly used structure for software projects. The deliverables and phase activities we will use are specifically for information systems for an external client, but different milestones and activities adapt the waterfall lifecycle to other types of software development, from the ignition controller for an automobile engine to the next release of a video game.

In spite of this, the waterfall lifecycle is far from perfect. Particularly, the assumption that a project progresses in a straightforward manner, from specification through design to implementation, is idealism at its most extreme. Other lifecycle models have been proposed which feature iteration, in recognition of the iterative nature of the analyze-design-implement cycle,[3] but these models are less able to accommodate the realities imposed by calender and budget. Another view is that the phases are handled recursively, with every phase having a certain amount of analysis, design, implementation, and testing.[4] The relative proportion of effort in the sub-phases reflects the major phase, so the that the design phase, for example, involves more design than analysis or implementation. This recursive approach is particularly suited for explaining why prototyping, an implementation activity, takes place early in the project.

In any case, we must recognize that the lifecycle phases are not pure. Nonetheless, with a fixed two semester deadline for completion of our projects, we must define and adhere to a strict schedule. Nonetheless, our schedule and the associated process recognize the inherent iterativeness of the project. Two milestones are provided during the design phase to allow for one iterative cycle where it is most necessary.

Table I.2 on the following page shows how the major aspects of project tasks appear in the various milestones.

B. Carry Forward Material

This section could equally well be called "Consistency," because that is the ultimate goal of the discussion in this section. But this is one occasion when the project requirements actually reduce your effort. The idea is that a substantial amount of material is carried forward from one milestone document to the next. For example, the diagrams and the descriptive text for the information model, developed first for the Requirements Specification, should reappear in each of the subsequent milestones.

Any design and development process, such as one characterized by the waterfall lifecycle, is transformational. That is, each stage (after the first, of course) takes the result of the

3. See, for instance, Boehm's spiral model [5]). Boehm's model is particularly appealing because it involves an explicit risk analysis at each iteration. Thus Boehm's model is carefully controlled and involves more rather than less process management.

4. "Evaluation" is a better word than "testing" when applied to all the phases.

Milestone	Major Project Tracks				
	Context	Database Development	Program Development	Interface Development	Software Engineering Process
Proposal	project goals				
Feasibility Study	client information; project goals	*get current forms & reports*	*read "polcies & procedures" manual* *client interviews*	*get current forms & reports*	benefit/cost analysis
Requirements Specification	identify hardware and software platforms; context DFD	ER model; data dictionary	functional requirements	functional and non-functional requirments	project plans
Prototype	platform architecture & tools	prototype database	sample modules	sample screens; *experiment with interface tools*	
Preliminary Design	platform tools	relational schema; data dictionary	module hierarchy and interfaces	design standards; screen and report specifications; links to other systems	test plan
Detailed Design		relational implementation (all details)	module details	screen layouts; report layouts	complete test specifications
Implementation		test-data sets	code; test scripts	screens (using application generator if available)	*configuration management*
Testing & Integration		*begin populating with real data;* *data conversion*	*unit and integration testing*	*testing by client*	*configuration management*
Delivery		Programmer's Manual		User's Manual	final acceptance

Key: deliverables in roman; *activities in italics*

Table 1.2: Project Activities and Deliverables by Milestone

previous stage(s) and modifies it.[5] In a simple programming task, these transformations are computerized, implemented by compilers and loaders, and the stage to stage consistency follows from the (hopeful) correctness of these tools. In a large system development project, those transformations are mostly done by human hands, notoriously inconsistent when compared to even the worst compilers. Thus close attention must be paid to make certain that these transformations do not introduce errors or omit important elements.

There are two mechanisms to ensure consistency: one is to use the text from each milestone as the basis for the next and the other is to make explicit correspondences between each milestone and its predecessor. Of course not every paragraph of one milestone is useful for the next, and much material must be added. But the Requirements Specification, for example, can comprise most of the first five sections of the Preliminary Design, according to the outline suggested below (§IX.3.B). An example of an explicit correspondence is given in the chapter on Design (chapter IX), where the use of tables showing how the system design meets the functional requirements is suggested.

C. Quality Assurance

The motivations for the focus on process are especially pertinent to the desire for Quality Assurance (QA). In truth, the reliance QA places on process factors is a fallback position – the root of *product versus process* dichotomy. It would clearly be best if the quality of the software product were exhibited directly, but evaluating software quality is notoriously – maddeningly – difficult. Hence we place heavy emphasis on monitoring the process by which software is developed, ensuring that it meets the necessary quality standards. Better understanding of and proficiency with testing reduces this heavy reliance on process monitoring as a means for achieving QA, but testing itself is built upon a process foundation.

D. The Team

The fact that this project occurs in a very specific team context is dominant among all the process issues. As in the real world, "human resources" is the most difficult to manage. Interactions between people, variations between people, and even variations between the same person on different occasions are problematic, unpredictable, and even sometimes infuriating. Fortunately these variations are manageable, given the proper process and attendant mechanisms.

On the positive side, the managed team experience is what most students and almost all potential employers cite as the most significant feature of this course.

Chapter VI, Teams, goes into may aspects of the team experience and how it is managed. You will note, however, that is several chapters away. The reason for this is that team size (and the concomitant management structure) ramps up over the first few milestones. The first milestone, the Project Proposal, is done individually. The second, the Feasibility Study, is done in pairs. All the rest are done in teams, typically comprise four individuals

5. A suitable analogy here is a chemical plant, where the initial goo flows from stage to stage, with different reagents and under different conditions. This is contrasted with the process of building a house, where adding the first floor doesn't change the basement, adding the second floor doesn't change the first, and so on.

but occasionally more or fewer as circumstances dictate. Note that team memberships are assigned and not of your choosing, another way in which in this course reflects real-world situations.

3. Terminology

The following terms are used throughout this *Guide*. Although many of these terms are common usage elsewhere (such as the "goal" *verses* "objective" distinction), this list is specifically intended to help you read this *Guide* and is not a general glossary for information systems projects.

Unfortunately several of these terms have different usage at different levels in the project; without care, these different usages can be sources of serious confusion. For example, in the example law firm project used later, the team may think "clients" designates lawyers but those lawyers think clients are plaintiffs and defendants. Indeed, the chapter on Requirements Specification has a section, §VII.5.B, which focuses on this issue where it is most problematic. Only one of the following terms, "process", mentions such a level difference, but other terms are also susceptible, as the "client" example shows. In general, watch for such possible confusions and avoid them by being specific.

Upon reading this section for the first time, get a general feeling for these terms. But remember that this section is here and return to it for clarification.

Checklist: At places is this *Guide*, and in many other resources, you will find outlines and checklists. Do not confuse them. An outline gives structure (and some content), a checklist only content. An outline is top-down; a checklist is bottom-up, in that it may cover low-level details but not general structure. An outline is a design, a checklist is a specification. Even though the name "checklist" indicates an after-the-fact reading of the document to verify the contents, you can (and should) use such checklists during the early stages. What you are "checking" is the task assignment – go through the list, ask whether each item is relevant, and assign each relevant item to one team member.

Client: (see also "Users") The term "client" generally means not just one person but the entire organization for which the project is being developed. However, sometimes "client" ambiguously refers to both the organization and the person within the organization who is responsible for the project. When it is necessary to distinguish this particular individual, the term "primary client" will be used (see section 2.3 of the Feasibility Study outline, page 46).

Component: A component is an independent part or piece of a deliverable. "Component" is used in place of it's synonym "module", for reasons explained below under "Module".

Customer: Since a discussion of "client's clients" would be hopelessly confusing, we will frequently use "customer" to designate people and organizations served by the project's client. The one exception is the running example of a law firm, since the term "lawyer's client" is so well established.

Goal: (see also "Objective") A goal is a general aim toward which effort is directed. While "goal" and "objective" may be synonymous in everyday speech, they have very specific

and distinct meanings as management concepts. It is sometimes said that "goal" refers to strategy while "objective" to tactics. In setting "goals and objectives" for a meeting or a system, there should be very few, rather general, concisely expressed goals and a longer list of specific, measurable objectives that lead toward the general goals. Goals are sometimes so general that they sound like platitudes or advertising copy: "The goal of ACME Company is to be the world's major supplier of widgets."

Module: Any piece of a deliverable that stands alone in representation or concept is a module. In a modern information systems development toolkit, a module may be a table definition (including constraints), a menu or form, or a more traditional collection of procedural code. Even documents have modules, which may be high-level (such as chapters or sections) or low-level (such as macros for the document preparation tool). However, since "module" still carries an implication of "code module", we will use the term "component" for the more general concept.

Objective: (see also "Goal") An objective is a tangible and specifically attainable target of effort and action. It is often said that an objective must be "measurable," which means that it is possible to determine whether or not the objective has been reached and possibly even how close the objective is. However, "measurable" does not necessarily imply "quantifiable."

Platform: (see also "Tool") The term "platform" may refer to any of the technology required to run your system, individually ("The database platform is Oracle.") or collectively ("The platform is a PC with $\cdots$ running $\cdots$.").

Process versus *Product*: When discussing software or system engineering, we often must distinguish the way we are doing things (the "process") from the things that result from our doing (the "product"). Unfortunately, systems are themselves made up of processes – causing a level confusion as discussed above. Emerging IEEE standards use "enabling process" to designate a process of the development cycle. We will use a different convention, where "the process" refers to the specific milestones and mechanisms used in this course, that is, our variation of the waterfall.

Scope: The dictionary entries that most closely capture our notion of scope are "extent of treatment, activity, or influence" and "range of operation". For our needs, concerns about scope are largely concerns about the boundaries that delimit our extent or range and the interactions across those boundaries. Once we know the boundaries, the activities and operations are easier to delineate.

System: Unless otherwise qualified, "system" always refers to an information system. That includes not only hardware and software, but application programs, policies and procedures, forms and documents to implement those policies and procedures, and the staff and other users who use all of the above. We will explicitly use "hardware system," "operating system," and the like when we wish to mean something less than the information system as a whole.

Tool: (see also "Platform") A tool is any piece of software that assists you in building upon your platform(s). The tool may be integrated with the platform (*e.g.* Microsoft Ac-

cess's form building tools) or independently supplied (*e.g.* comparable form building tools for the Web).

Users: (see also "Client") The users are all of the individuals who will regularly (as a group, not always the same individuals) interact with the system. This includes managers, office staff, staff in warehouses or factories, and visitors who may be different from day to day. In IU's on-line registration system, students are users even though one individual student makes up a tiny fraction of the use of the system. A user may be a member of the client's organization or may be merely an individual who interacts with that organization.

Consideration of the above list reveals that the product *verses* process dichotomy even appears in our terminology. The same word takes very different meanings when used in the context of product or process. The need to use "customer" for "client's client" is a consequence of this fact. In the phrase "client's client", the first "client" indicates the client of the process and the second indicates clients tracked within the product. Another more blatant example would occur in a project for a sports league: the process "team" refers to those computing professionals implementing the system while within the product "team" refers to a collection of players wearing the same uniform. Unless you intend to lumber around in a helmet and shoulder pads, it behooves you to keep such terminology straight.

Word usage in this guide is deliberately moderated, taking into account the fact that many readers are from outside the US and have only learned English in recent years. Nonetheless, there are occasions where unfamiliar words are used because they have unique combinations of meaning. Consequently, if words such as "rubric" are unfamiliar, you should read this *Guide* with a dictionary at hand.

4. Overview of this Document

This *Guide* has chapters which deal in turn with each of the required milestones, with the exception that the two design milestones are covered in a single chapter and the implementation phase, so dependent on a particular software platforms, is not covered at all. In addition, there are chapters on Client Issues and on Teams at the points they become relevant, before the Feasibility Study and the Requirements Specification respectively. Tools and methodologies required at each step are discussed in the relevant chapter – for example the Requirements Specification chapter discusses data flow modeling and the Testing chapter, glass-box testing. However, the important Entity-Relationship methodology used for information modeling is not covered in Chapter VII because this material is thoroughly presented in the lecture notes.

In addition, certain chapters contain sections that are particularly of broad relevance. The preceding list of Terminology and the following Hints are such sections as are the discussion of writing, Chapter II, and of modeling frameworks, chapter V.

As with any serious text, this *Guide* is not intended for single pass reading. Skim each chapter and then revisit it for details. After the first two milestones, go over the chapter with your team to make certain you have identified all salient issues.

5. Hints, Tips and Suggestions

This entire *Guide* is, in some sense, just a compendium of tips. But there are a few issues which the Software Engineering Management students, who had experienced the Information Systems sequence, feel are especially important. It is impossible to order these admonitions; they are all important. If you ignore any one of the monsters discussed, it will sneak up and bite you.

Look ahead: A project that flows cleanly from stage to stage will be easier on the team and has a much higher chance of success. Such fluidity does not occur by happenstance; it is the result of knowing how each stage makes use of the previous stages' results. Therefore, understand future stages in preparation for the current one.

Size and scope: No one can understand the entire project. Modularize and delegate.

Design before implementing: The victim of the "shoot first and ask questions later" syndrome is invariably the one who pulls the trigger.

Learn your tools: Try to get started early on understanding your development environment so that you can find out early what resources will be available to you to assist you later in your development.

Throw away prototype: The purpose of the prototype is to further understand your tools and the client's needs. Plan to keep *only* these lessons from the prototype.

Plan to work in parallel: Does it seem like there is a lot of documentation to do? Well you're right and it means you should get started as early as possible on each document. Design your project with components and then do some parallel processing. This is necessary if you intend to finish things on schedule.

Client relations: It is important that good relations be kept with the client at all times, especially keep promises made to the client.

Timely problem solving: Deal with both process and product problems early on, otherwise, when it comes time to turn in documentation or do a milestone demonstration, you may find that things are not working or looking quite as you expected, or not there at all!

6. Further Reading

Most software engineering texts are targeted at areas other information systems and hence tend to underplay important topics such as client relationships and information models. On the other hand, typical texts on information systems (typically used in business school courses) underplay implementation and testing. While there are other books that do better on specific subjects, the text by Pressman[25] is the best software engineering text across the board; there are many editions and almost any will do.

Brooks' *Mythical Man-Month* [10] (or the article [9], which is included in the second edition) is an enjoyable and highly readable, although somewhat dated, motivation for software engineering. It is suitable for "light reading" but contains important insights.

II. Technical Writing

English is your most important professional tool, use it with precision.[1]

This chapter contains a summary of important points on writing – sometimes to reemphasize things that you were taught in secondary school and sometimes to indicate ways in which technical writing differs from the art practiced by Charles Dickens or Stephen King. The principles of good writing are not long or deep. All that is required is the discipline to use them.

Discussion of writing involves two intertwined but vastly different aspects: the characteristics of a well-written document and the process by which ideas become a document. Chapter I introduced the product *verses* process dichotomy; here we see it occur for the first time.

This chapter is far from complete as a discussion of writing. It does not repeat material from the many books on writing which cover the subject in great depth and breadth. The best is certainly the gem *The Elements of Style* by Strunk and White [31]. You are urged to buy that book and study it. In contrast to *The Elements of Style* and most other books, this chapter narrows its focus to technical writing and even more narrowly to systems- or software-related documents.

Writing is the process by which you put meaning into a document. You will spend considerable effort doing so. To the extent that your reader does not get the same meaning out, your effort has been wasted.

This chapter begins with a section applying the preceding idea through understanding how your reader will comprehend the document's meaning. That is followed by a description of a well-written document. The chapter ends with a brief discussion of process: team and technology mechanics. At several places, considerations that went into writing this *Guide* are discussed as examples, typically presented in footnotes.[2] In addition to the material in this chapter, Chapter XI, Documentation, toward the end of this *Guide*, contains further points on writing specific to user's and programmer's manuals.

1. Targeting the Reader

The goal of writing is to communicate, not merely to fill up a few pages with words, so it is important to understand what communication entails.. Communication involves two parties: the sender (writer, in this context) and receiver (reader). The formal model of communication (as used by, say, computer network engineers) begins with a meaning which is encoded into a message by the sender. That message is then transmitted to the receiver, who decodes the message to retrieve a meaning. Ideally, the same meaning is extracted as first was entered, requiring (1) that transmission is error free and (2) that the

1. The origins of this exhortation are lost in academic legend, but it reportedly originated with a professor of Physics, not of English.

2. Such extensive use of footnotes is generally frowned upon. However, in this case it is easily justified because writing about a document in that very document can create hopeless circularities in the mind of the reader unless the distinction is clearly delineated.

decoding is the inverse of the encoding. The first condition is always satisfied by written documents. The second condition requires that, because you have no control over the decoder (reader), you must encode (write) to satisfy the equation $encoder = decoder^{-1}$. An English professor would say this directly: you must write for the reader.[3]

Although most of the discussion in this chapter is phrased in terms of "the reader", there may be a variety of different readers. The term "the reader" is used because each reader reads the document as an individual. The first step toward writing for the reader is, of course, to identify the readers, however varied they may be. While some documents are intended for one or a few readers (a Project Proposal may be directed specifically toward one key manager who has budgetary authority over the project's domain), others have wide and varied audiences (especially the User's Manual). So identification of readers may involve listing specific names or may involve categorizing broad classes.

Having identified the readers, the next step is to characterize them. Some of the characteristics that should be evaluated for each class of readers are

- knowledge of computers.
  - ◇ specific task-oriented skills
  - ◇ deep understanding
- knowledge of the application area.
  - ◇ cursory familiarity
  - ◇ fixed, simple operations
  - ◇ deep understanding
- motivation for reading the document.
  - ◇ find a particular fact
  - ◇ develop general understanding
  - ◇ evaluate for action or approval
- manner of reading the document.
  - ◇ start to finish
  - ◇ as a reference
- skill in reading comprehension

3. This *Guide* was targeted toward advanced computer science students. You have technical and mathematical skills, so the rather extravagant occurrence of the above equation should not cause you the consternation it might cause for an English major.

2. The Well-written Document

This section is organized by the three general categories by which written presentation is evaluated in this course – Organization, Explanation, and Grammar/Usage. In general terms, "Organization" deals with issues at the level of the entire document or major subsections thereof, "Explanation" deals with paragraphs, and "Grammar/Usage" deals with sentences and words. The order of three subsections – Organization, Explanation, and Grammar/Usage – is the same as that in which they should be addressed during a writing task. Of course each topic has both product and process aspects, which are interwoven in the following discussion. This section ends with a proofreading checklist, which serves as a summary.

A. Organization

i. General Principles

A well-organized document does not just happen. It is the result of a careful design process. The most evident aspect of a document's design is the arrangement of related ideas into chapters, sections, and the like. Strunk and White capture the difference between writing words and writing a document: "Writing, to be effective, must follow closely the thoughts of the writer, but not necessarily the order in which those thoughts occur"[31, rule 12].

Computer folk typically have a harder time organizing paper documents. We're used to working with more powerful (but more complex) structures that do not have the strict linear organization of a book. The general public is beginning to use richer directed graph structures, since that is the abstract characterization of web hyperlinks, but they never have seen topological sort.

There are several principles for organizing a document. As with any such "obvious" principles, it is easy to forget them. Therefore we list them here. Items marked "•" are firm rules; items marked "◦" should be weighed against other considerations.

- Define before use.
- Give context before details.
- ◦ Cluster related ideas.
- ◦ Organize subsections at the same level of abstraction.

The last point deserves explanation, which is best done using an example. In discussing the various functionalities in a Requirements Specification, you might discover three general categories: data entry and update, security and user control, and report generation. Each category becomes a major section of the document, with several sub-sections. However, you wonder where to place backup, which is a singular activity and therefore would need only a simple, short section. But backup is at the same level of abstraction as report generation and hence deserves its own section.

The process by which you organize a document may be top-down or bottom-up. Writing an outline first is a top-down approach. It is suitable if you know the overall material very well. Curiously, it is also suitable if you know very little about the topic, since building an outline is a good technique for researching the material. A bottom-up approach begins with collecting all of the information and then arranging this information. The old "technology" of $3'' \times 5''$ cards still works well because it enables you to lay out the material on a table. Often a good way to organize a document is to combine these two approaches in a multi-pass strategy, alternating top-down and bottom-up.

As you struggle with organizing a document, you face choices in the placement of material. Whenever you make such a choice and place the material in some spot, make a note of this placement in the locations where the material did not go. These notes later become valuable cross-references.

If you carefully organized your document, you had a reason for placing each piece of material where you did. When the reader understands these reasons, reading is easier and comprehension is better. One way to facilitate this is to write chapter titles and section headers which clearly indicate what the chapter or section contains. Another way is to have a few sentences of introduction at the beginning of a chapter or section; however, doing this for sub-sub-sub-sub-sections is overkill.

Examples will be easier to write and to follow if you plan them while you are organizing your document. As you lay out the material, note where an example would contribute to the presentation and what it should exemplify. These notes can then be drawn together to suggest one or a few examples that thread throughout your document. Take care not to pick examples which require vast amounts of extra knowledge. The example of an attorney's office was chosen for this *Guide* because the basic operation should be familiar to all readers; thus only minimal explanation was required to set the stage for the example.

Do not repeat the obvious. For example, section 3.2 of the recommended Feasibility Study checklist calls for a description of your client's business; if your client is IU or the Monroe County Government, the name is sufficient.

ii. Executive summary *verses* introduction

One common problem people have with organizing documents comes from the confusion between the executive summary and the introduction. Both are relatively short, toward the beginning of the document, and rather broad in content. However their differing purposes are clearly indicated in their names. An executive summary is meant to summarize the entire document, with as much or more attention on conclusions as preceding material; as its name implies, it is intended to be read by a busy senior executive who likes to keep tabs on what is happening across the organization.[4] An introduction, on the other hand, indeed introduces both the subject and the document to a reader who intends to read it in its entirety.

An executive summary is meant to be read alone; hence it should be understandable by itself. On the other hand, the document should also be understandable without the

4. Or, as disgruntled staff claim, the executive's reading skills are challenged by more than two pages.

executive summary. Do not depend upon ideas defined in the executive summary elsewhere in the document. If necessary, material from the executive summary may be repeated the introduction.

An executive summary should focus on the problem and its solution. Where necessary (almost always the case in this course), the problem statement may be preceded by a paragraph or two setting the context and the solution description may be followed by an overview of the path to the solution. The most difficult aspect of an executive summary[5] is to be specific without being detailed. An easily understood example is how project budgets are included in an executive summary: the figures must be exact but must not itemize individual expenses (a few broad categories, such as personnel, hardware, and software are permissible). An executive summary must always be one and a half to two pages in length.

An introduction should guide the reader through the document. First, it must connect the document to things with which the reader is familiar; that is, it should provide context. This context includes the environment in which the problem occurs, the goals of the project addressing the problem, previous attempts at solutions, *etc.* Unlike the executive summary, it is sometimes valuable to be neither specific nor detailed. Subsequent parts of the document should provide specifics and details; the introduction should indicate how those parts go together.

This leads to the second purpose of an introduction: introducing the document itself. This usually happens after the subject and its various parts have been discussed and is accomplished by a paragraph or two describing how the parts of the material fit into the parts of the document. This should be quite specific (as in section I.4). These paragraphs are easy to write if you start by translating the table of contents into sentences.

You will be reminded of this distinction when specific documents are described, such as the Feasibility Study in section V.2.

B. Explanation

Recall that "Explanation" identifies the paragraph-level aspects of the document. This rubric is suitable because each idea, assembled during the organization stage, must now have an explanation constructed for it. This section briefly discusses individual paragraphs and then the use of lists as an alternative to paragraphs for providing structure in a technical document.

Paragraphs can be of any length. Some are long, others short.

Rule 13 of Strunk and White[31] has many good pointers on constructing paragraphs. Two which deserve repetition are that each paragraph should have a single theme and that the first sentence of the paragraph should set that theme. The idea of a "theme sentence" is probably familiar from as far back as elementary school. However, not every paragraph can naturally be introduced by just one sentence. The first paragraph of section

5. Indeed, writing executive summaries is one of the most difficult writing tasks; hence skill in this area is valuable and very much worth developing.

II.2.C begins with two sentences that are more polemic than informative. This is just one example of the use of the beginning sentence to aid the flow of the document.

Sometimes a well-written theme sentence can also facilitate flow from the previous paragraph. For example, the first sentence of the second paragraph of Section V.1.C links the notion of a variety of models to frameworks that organizes such models. Unfortunately, such well-written transition sentence are difficult to craft.

Specificity and organization are often more important than flow, hence technical documentation is often organized around lists and tables rather than paragraphs. Lists are easier for the writer because there is no need to craft sentences tying the ideas together. They are easier for the reader (if those paragraphs are well-written) because the reader sees the various ideas in relation to each other and can digest each fact separately.

List are, however, not merely an agglomeration of items. Items in a list should have substantial commonality and the presentation of the list should reflect that commonality.

The first step in building a list is to introduce it. This introduction should provide the reader with understanding of the list's context and the commonality of its items. A list may be introduced by a partial sentence, such that each list item completes the sentence.[6]

The items in a list should have parallel form. The following list explains these points while simultaneously illustrating *in*correct form. Items in a list should

- have used the same tense, preferably present.
- be written in the same voice, preferably active.
- share all aspects of construction with each other – that is they should have basically the same parse structure. However additional sentences may be added which do not conform, although it is not a good idea to have one item in a list ramble on for several lines if the rest are only a few words long. See also Strunk and White, rule 19.

The layout of a document can be as important as word choice. Use white space to emphasize organization and important topics. Do not, however, get carried away with visual clutter, such as lots of boxes with different borders.

C. Grammar and Word Usage

Grammar deals with sentence-level aspects of writing, including word choice and word use as well as punctuation.

i. Sentence structure

There are many suggestions for writing sentences which will not be repeated here, although this section will briefly illustrate the application of such suggestions. Once again Strunk and White is an excellent source – chapter I covers matters of punctuation while chapter II is mostly about sentence characteristics. One suggestion omitted by Strunk and

6. The introductory sentence to the following list reflects this technique. The skeleton of this sentence is "Items should ____", where each list item fills in the blank.

White is that several short, focused sentences are better than one long sentence with many ideas.

It is difficult to write while keeping all the suggestions and rules about grammar in mind, but this difficulty may be overcome if you first write the sentence as it comes to mind and then re-craft it.[7] The following example shows how recrafting both shortens and strengthens a sentence (numbers in brackets indicate the rule from Strunk and White that was applied in the rewriting):

- "There are ten procedures that may be called by procedure ScanHeader." Stripping the needless words [17] "There are ... that" gives the simpler sentence:
- "Ten procedures may be called by procedure ScanHeader." Moving to active voice [14] give the even simpler sentence:
- "Procedure ScanHeader contains calls to ten procedures." An extra benefit of this last rewriting is that the true intent of the sentence, describing the program syntax, is evident in the phrase "contains calls", rather than the confusing and problematic subjective phrase "may be called".

An important precursor to rewriting is to parse sentences as you revise or proofread a document. For example, make certain that each component of a conjunctive phrase makes sense by itself. For example, in evaluating the sentence "I saw Bill and she at the movies", replace "Bill and she" with each of its separate components. Using "Bill" results in a grammatically correct sentence while using "she" does not.

ii. Word choice

This is not Alice's wonderland. Humpty Dumpty's statement "When *I* use a word, it means just what I choose it to mean – neither more nor less"[11, Chapter VI] does not apply. If you wish to convey meaning, you must use words with the meaning they will have for the reader.

Use words that:

- have a familiar meaning. If you must introduce new or technical words, define them before they are used. Don't use synonyms (especially for technical nouns) – this rule may produce a dull document but it will be more understandable.
- have a single, consistent meaning. The worst confusion arises when a word is used with two specific distinct, yet close meanings. To avoid this confusion, database texts talk about "implementing an ER relationship with a database relation." Without the "ship" the sentence would be correct but hopelessly confusing.
- fit. For example, do not use words that confuse implementation with function, as happened in the early days of computers, when US engineers used the term "vacuum tube" to describe the building block of circuits while those in the UK used "valve".

7. This difficulty is the kernel of "writer's block" and indeed this *Guide* exemplifies that too. The author did indeed use the suggested technique to keep the words coming – and hasn't had sufficient time to do all the required recrafting.

With the invention of the transistor, the word "vacuum tube" went out of date while a transistor is still a valve.

One particular problem in word use is the tendency to make relative notions seem absolute. It is common to say such things a "Uwe Blab is tall".[8] But this statement is only true when we understand that "tall" means "taller than most humans". The statement "General Grant was tall" is deceiving because the implicit context – humans – is not the correct context. The correct context is trees: General Grant was a sequoia tree in the California mountains that happened to be named after a US military office who later became President. This difficult with making relative terms absolute is as much conceptual as verbal and is often confusing in technical realms where assumptions about explicit context are often wrong. A good example here is the "logical/physical" dichotomy of database systems – an operating systems programmer sees this division as much closer to the actual hardware than does an application programmer. The solution, of course, is to be more explicit – using "at the level of device interfaces" (the system programmer's interpretation of "physical") or, respectively, "at or below the relational table representation" (the application programmer's interpretation).

Of course Strunk and White have much valuable advice here as well; chapter IV is devoted to misused words and expressions. Take that chapter to heart and you will never misuse "which" and "that". One set of words they missed, however, is "if", "when", and "whether": "whether" indicates a choice of conditions, "if" a hypothetical or stipulated condition, and "when" the occurrence of a certain condition.

While having the right choice of words makes documents clearer and more readable, choosing those words is sometimes difficult. Waiting for "the prefect word" is an invitation to writer's block. A technique to handle word choosing is put down a reasonably close but inadequate word and flag that word as problematic.[9]

Section IV.3.D, admonishing you to 'Present Ideas in Ways Appropriate to the Client', has further discussion of word choice and related matters.

iii. Pronoun referents

Perhaps the greatest damage to the meaning of a sentence comes from incorrect pronoun referents. A pronoun is a pointer and programmers, of all people, should be aware of the damage caused by misdirected pointers. When you write a simple "it" or "this" or "that", you know perfectly what the word denotes and the sentence is perfectly clear to you; but without that same knowledge the reader perceives only murkiness unless you are careful with the referent.

The referent of a pronoun always precedes the pronoun (and in fact is called the pronoun's *antecedent* for exactly that reason), but even experts cannot give hard and fast

8. Uwe Blab was the only IU Compter Science major to play professional basketball. He got into the NBA not because of his stellar record in CS but because his height was over 7 feet.

9. This author uses "[[$\cdots$]]" as such a flag, so early drafts of this *Guide* had may places where [[wrong]] words were flagged in this manner. Of course "wrong" in the preceding sentence is itself misleading but not really wrong.

rules governing how far back to look for the antecedent. Sometimes it is the immediately preceding noun phrase (as in this case). Sometimes it is the subject of the preceding sentence (as in this case). Sometimes it is even further away (as in this case).

When the referent of a pronoun is not obvious, use only the noun from the referent phrase preceded by "this", "such," *etc.* For example, the second sentence in the section II.3.B below could be (poorly) written: "A few hints about it are $\cdots$". The intended referent of "it" is 'effective use", but that is not obvious. Replacing "it" by "such use" avoids any ambiguity.

D. Proofreading Checklist

Table II.1 summarizes the above ideas (and adds a few from other sources) by giving a checklist to be used (repeatedly) while proofreading your document.

Is this idea or term previously defined?	_____
Is this idea or term used consistently?	_____
Does this paragraph have topic sentence(s)?	_____
Does this paragraph match its topic sentence(s)?	_____
Does this sentence parse as intended?	_____
Is this the proper word?	_____
Is the referent of this pronoun obvious and correct?	_____

Table II.1: Proofreading Checklist

3. Mechanics

Technical writing as a process differs from "artistic" writing in two ways: it is often a team activity and it benefits more from computer tools. We discuss those topics in turn.

A. Writing a Document with a Team

Technical documentation is often the result of group authorship, hence it requires preparatory planning and final melding.

Decide on format standards first, before writing. This way all group members can use the same format in writing their parts of the manuals and much time can be saved when integrating the parts into a manual with a consistent style. Table II.2 on the following page is a checklist for a document standard. It does not define a standard; instead it indicates the factors you must consider while defining your own standard.

When a team is organizing a document, first have each team member think about the relevant topics (the bottom-up phase) and write those topics on Post-It notes. Then come together in a room with a large white board and start pasting the notes in related categories (see also section VI.3.C).

Page layout

- page header _____
- page footer _____
- odd/even page differences _____
- margins _____
- footnote conventions _____

Chapters & Sections (for each level:)

- rules for numbering _____
- rules for header capitalization _____
- header font & point size _____
- mapping to Table of Contents _____

Paragraphs

- indentation _____

Lists (for each list type:)

- tagging _____
- capitalization and punctuation rules _____

Tables & Figures

- numbering _____
- label format _____
- placement _____
- references to _____

Cross References (relates to Chapters & Sections)

- format _____
- consistency enforcement _____

Index

- choosing entries _____
- recording _____

Tools

Table II.2: Document Standards Checklist

Even if a thorough standard has been developed and carefully followed, a draft with many authors will display many styles. Therefore one writer should meld all the sections with a final pass through the entire document to insure consistency of style. This melding should consider both content and format. This melding is also a planing issue: make certain that you allow enough time for this melding.

B. Technical Hints

Effective use of software tools can enhance the quality of documentation and facilitate its preparation. A few hints about such use are:

- Use macros or stylesheets to standardize formating and defer layout decisions.[10] Document preparation postprocessors, where the author defines the content and layout before software "sets" it on pages, have more powerful macro facilities. Typical WYSIWYG (What You See Is What You Get) word processors have macro tools which are used during text entry, not post processing.
- Use the automatic indexing and cross referencing tools provided by your word processor. This is very helpful in providing consistency.[11]
- Apply modularity to document preparation. Enable individual pieces to be written and run-off independently, but have a method which automatically generates a complete document. Also use good configuration management. This is valuable in general but essential for documents written by a team.
- Avoid the "n^2 writing syndrome." With modern document processing tools, it is common iterate revising existing text and entering further text, starting with the first section and working toward the last. This means that if you have n sections, you will revise the first $n-1$ times before doing any revisions on the last, so writing quality falls off greatly further in the document. This is also inefficient use of your time, since you will have done $\sum_{i=1}^{n} i = O(n^2)$ section revisions for n sections.[12]

10. This *Guide* was prepared in this manner using TEX. Macros were defined for chapter, section, *etc.* Initially these macros only echoed their parameters. But as the document took shape they were enhanced. For example, including sections as well as chapters in the Table of Contents was a trivial change, even at a late stage, because all the machinery was in place. On the other hand, there was no such prearranged format for document outlines, and it shows.

11. Section number references in this *Guide*, as in "see Section III.2", are automatically generated. Thus the document could be rearranged without worry that section pointers were incorrect.

12. A final reminder of the importance of targeting the reader. This document could well be the only writer's guide that includes a use of the big-Oh notation, since it is assumed that a computer scientist's technical background includes this notation.

III. Proposal

The proposal is the key document necessary to secure a contract in the business world; similarly in this course it will be the first factor in deciding which projects proceed through the remainder of the course. Your task is to find a potential client and write a description of the client's proposed project. The client may be any individual or organization who is interested in having a project done and who is capable of supporting the design and development of that project.

This chapter begins with a discussion of the criteria by which projects are evaluated. It then suggests a general outline for the proposal.

1. Criteria for Project Evaluation

Both the client and the development team should evaluate a project, choosing among the possible alternatives, including the *status quo*. In the "real world," the client's judgment typically matters most, since it is the client who must pay for the development. Developers more often exert their influence by setting the price. However, developers may also make a judgment whether they wish to continue – especially, a small consulting firm may reject a project that is too risky or too far beyond their resources.

We also make that judgment in the Information Systems course. It is in fact a two-stage judgment, the first stage (based on the Proposal) concentrating on the client's characteristics goals, the second (on the Feasibility Study) on details of the project's environment, its technical characteristics, and its suitability for the course.

A few criteria by which a project is judged are itemized below. Those marked ⋆ are absolute, while those marked ∘ require careful consideration. These items not only determine whether a project continues but also may be used to adjust the project objectives and processes; for example, a project with uncertain goals might assume more characteristics of a large prototype. In any case, you should not by yourself reject a project unless it is completely out of the question. Instead, if you suspect difficulties with a possible project, gather as much information as possible and report that information, including an evaluation of the difficulties, in your proposal. This discussion belongs in the final section of the outline suggested below.

⋆ *consequences of failure*

This course does not have adequate controls to attempt systems whose failure might imperil life and limb (for example, medical patient records) or might have serious commercial or legal consequences (for example, accounting systems).

⋆ *relevance*

This is an information systems class and projects must reflect this. However, non-traditional database applications are certainly welcome. For example, a project that does numeric calculations with scientific data does not satisfy the relevance criterion but a project that manages that same data does.

⋆ *location*

Projects outside the Bloomington area have rarely been successful.

⋆ *managerial infighting*

A project reflecting managerial infighting in the client organization is not a pleasant experience. However, it is likely that such problems will not surface during the Proposal phase. Still, you should be on the lookout for such problems (discussed in chapter IV, "Client Issues").

◦ *size and scope*

It's usually quickly clear when a project is too small. Determining that one is too large or complex often requires the more detailed examination of the Feasibility Study.

◦ *client relationships*

The client must be willing to commit time to meet with the team during specification and design and to commit the resources to install or operate the system. The client needs to have experience with or at least proper appreciation of computer systems and realistic expectations about the proposed software. Finally, difficulties inevitably ensue if the client has ties (employment, family, personal, *etc.*) with some team member. To remedy this latter situation, if a good project is proposed by someone too close to the client, the proposer will not be on the final project team.

◦ *clarity and certainly of goals*

Projects with unclear goals are usually problematic. Even a goal as general as "computerize this process" is adequate provided the process is well-defined. Occasionally the client's real expectation is that computerization will organize an ill-structured situation and this puts the team in an awkward position.

◦ *platform*

With the low price of PC's and database software, lack of suitable a platform is rarely a difficulty; but occasionally the client does not have facilities or resources to acquire those facilities. More problematic is a situation where the client insists on the use of inappropriate tools.

2. Proposal Outline

Your proposal will not necessarily be a long document but it should clearly propose your project. The following items describe the content of a *typical* Project Proposal. Many of the issues raised here recur in the Feasibility Study and perhaps yet again in the Requirements Specification; each subsequent stage requires consideration in greater detail. Hence you should read ahead to see the level of detail at which these questions should be answered at the various stages.

- Client

Identify and describe the relevant clients – in particular the client organization and the primary client (defined below). The term "client" has many meanings, generally indicating those with a stake in the outcome of the project. To avoid the ambiguity of "client", we use "*primary client*" to designate the (commonly one) key person who initiated the project, defines its goals, and allocates resources to it (resources include not only the hardware and software to implement the system but the substantial staff time required for analysis and validation). Later in the project, the primary client will be asked to confirm the project's specifications.

Clients should be distinguished from users, who also have a stake in the operation of the system. Although clients may be users of the proposed system, it is very common they make up only a small portion of the user base. Indeed, clients may use the information of the proposed system but not the software that maintains that information Users of the system, both internal and external, are discussed under Scope below.

Client commitment is crucial, so discuss it here or in the final section. If you have any doubts on the client's goals and commitments, make sure to discuss them here or be prepared to face a rough road.

- System Goals and Objectives

Discuss the ways in which the client expects to benefit from this project. This section sets the tone for your presentation. It is very important that the reasons for the project are clearly organized and thoroughly presented.

Recall the distinction between goals and objectives discussed in section I.3. Because goals are broader, they should be presented first. Note that technology in and of itself is never a goal and rarely an objective – the one exception is when the goal is compatibility with some externally defined technical standard and the corresponding objective is of course implementation of a conforming technology.

- Scope

Describe the extent of the project: its boundaries and interfaces with other pre-existing or proposed software or manual routines. In order to clearly understand (and thus communicate) the scope, first obtain a global view of how the client organization operates. Then ask your client whether changes in operations are anticipated. Finally, fit your project into this global view, discussing what part(s) of the operation your project must facilitate and how it needs to interface with other systems.

The proposed system will certainly interact with users, perhaps several classes of users. Therefore the description of scope should identify the project's users, but only in the broadest terms. In the law firm example that we use later, it is sufficient to say that the users are clerical staff and attorneys. In a web-based ordering system, it is important to distinguish external customers from the internal staff who maintain catalog data, fulfill orders, and manage the web site. Individual staff from the client's organization and other users will be discussed in the Requirements Specification.

- Software and Hardware Platforms

Verify that there will be platforms for the development and installation of your system. Hopefully a development environment equivalent to the client's facilities will be available on campus, since this greatly facilitates development by the project team. This verification of course requires knowledge of the final platform (in general terms).

In order to provide that required knowledge (and of course to set the stage for the final installation), the client must either already own the necessary hardware/software or they must have a firm commitment to purchase it. If the client already has the necessary computing environment (or if they are already planing hardware or software acquisitions), describe the environment and evaluate how well it meets your system's requirements. If they do not already own a system (or must upgrade), discuss their commitments to make the necessary investment. If you propose new acquisitions at this point, specify them only in the broadest terms; if necessary, you will provide detailed specifications as part of the Feasibility Study.

Since it is necessary to have ready access to the client's computer in order to test and install your system, insure that such access in readily available. Ideally, you will be able to use their office after hours so you have full use without disrupting their daily activities.

- Conclusion

At this point, sum up your own feelings about the project and the work involved. Discuss whether or not you think it is likely to succeed. This is also a good place to explain your views on the client's commitment, if this has not be thoroughly discussed previously.

You should not expect your Project Proposal to be a "closed" product, with no loose ends. It may be that the most appropriate things you can write at this point are further questions.

IV. Client Issues

This chapter discusses ways in which client behavior can impact information system specification and development and ways to ameliorate the negative consequences of such behavior. It is important to understand that this is not negative behavior on the client's part but potential negative consequences of the client's natural behavior. That behavior, which may be overt action or merely thought and concept, is a direct outcome of the client's situation and is to be expected and anticipated. The only negative aspect occurs if you, as the information systems professional, allow the client's behavior to diminish the overall success of the project. Following our "river" metaphor, if you are careless and are mauled by a bear, it is your fault, not the bear's.[1]

We are concerned with client issues because it is essential that you gather the best information possible from the client. In fact, good information is all you really need from the client to develop a successful system, although additional resources may be required to deploy the system.[2]

We have already noted that the term "client" is seriously ambiguous. It may refer to the person who originated the systems project, the person who will be responsible for its development or its operation, or the person(s) who will use the system on a daily basis. In this chapter, we actually make use of this ambiguity, in that our discussion may refer to any or all of the above roles. None the less, it is essential that you understand (and record in your notes) the roles of various individual clients. In this *Guide*, we will use the term "primary client" to mean the individual responsible for the proposed system, in particular that person who must sign-off on the Requirements Specification and on the completed system upon installation.

The first section of this chapter discusses the delicate issue of client's view *versus* analyst's view. The second discusses some other common pitfalls and ways to avoid being trapped by them. Note that pitfalls are problems only if you fall into them; the purpose of this discussion is to forewarn you. The final section gives some suggestions on dealing with clients.

> Inadequate appreciation of clients' and users' characteristics and perspectives
> is the single most serious cause of information systems failures.

It must be emphasized that clients do not behave as they do in order to frustrate the analysis team, although frustration is often a consequence of client pitfalls. Client staff want to solve problems, but the problems they see tend to be immediate ones – often the problem is just to make it through the day.

1. Or perhaps the fault lies with your guide, who did did not yank you back into line when you saw the cute, fuzzy bear cub. Expect to be yanked!

2. Of course marketing your efforts involves a whole different dimension in client relationships. While the specifics of marketing are beyond the purview of this course, nothing is more important for long-term marketing than your reputation.

1. Influence of the Client's Perspective

The thorny issue during the analysis phase[3] is to gather the right information from the client and to reflect this information in an accurate analysis. The difficulty is that an accurate rendition of the client's statements and desires does not always yield an accurate analysis.

On one hand, it is correct that "the client knows the business, you don't." On the other hand, the client often knows the business too well and is necessarily focused on day-to-day details instead of the big picture. The client know how things *do* work; the analyst should perceive how things *could* work better. Furthermore, the analyst should be able to predict how things will change with the new system, including negative consequences and positive opportunities which the client does not see.

A feasibility analysis is somewhat like a medical diagnosis. A physician begins not by identifying the disease but by exploring the symptoms.[4] The symptoms reported by the patient need to be clarified – sometimes they are quite misleading – and additional investigation is needed to determine other symptoms. Similarly, the analyst must reflect upon the client's reported symptoms, politely scrutinizing any attempted diagnosis by the client, and examine the client organization for all symptoms.

As a simple example, consider a business owner who said "my company needs a better data entry system for it's billing program. It takes a secretary all day to enter the bills." If you had followed that statement as the real problem and implemented a much superior data entry system, you would have delivered a great solution for the wrong problem. The real problem was that all the information was already in the order-entry system and the process of re-keying that data needed to be changed to a process feeding directly from the order entry system to the billing system. Thus the client described the immediate solution without considering the underlying problem; you must delve beneath the client's statements to the real problem and discover the better solution.

A. Common Perspective Pitfalls

There are far too many pitfalls in this arena to describe them all, but some of the more common include a client who:

- *thinks in terms of derived data*: "We need to record the age of each employee," ignoring the fact that age changes annually.
- *wants to do only part of problem*: "We need to record when items are removed from the warehouse," ignoring that inventory must track both removals and additions.
- *wants to treat symptom, not disease*: "We need to track help desk calls better," when the problem is that the current software configuration confuses too many users.

3. Recall that the analysis phase includes the Feasibility Study and Requirements Specification milestones and perhaps some of the initial design work.

4. An experienced physician may make a quick diagnosis based on common symptoms of a disease which is "going around." An experience analyst often makes a similar quick judgment. Both understand this is a calculated risk and monitor the patient/client's progress.

- *thinks only in terms of linear processes*: "After A, we always do B, then C and D," when each step leads to several possible subsequent actions depending upon a variety of tests.
- *does not generalize adequately*: "There are six groups of products in our inventory," when what is important is the the products fall into groups and not the fact that there are six such groups.
- *does not recognize that structures as well as values change over time*: "The database should record the stock we have on seven shelves," ignoring the fact that last year two new shelves we installed. This is closely related to the previous item, where it is generalization over time that is inadequate.
- *describes normative, not actual, behavior*: "Customer complaints are always handled by George in Customer Service," when smart customers have figured out that calling Sue in Engineering works better. This terminology is further explained in section IV.2.B below.
- *confuses the improbable and the impossible*: "It never happens ..." really means it hasn't happened in the last few weeks. The client's current system may be flexible enough to handle rare cases, relying on Post-it Notes and personal memory, but a computer system is not naturally so flexible.[5]
- *injects largely irrelevant issue into the discussion.*: For example, a member of the client's staff is more concerned about the brand of PC than about the functionality of the system.
- *knows just enough to be dangerous*: "The system should have one table for customers and another for suppliers," knowing that tables are the heart of a database system but not knowing the principles of data modeling or relational design.

✝ *focuses on short-term needs exclusively*: "We need a summary of stock on hand for next month's year-end closing," when the real need is for an inventory system.

✝ *wants an information system to shore up poor business practices*: "We need an inventory system," when the real problem is that stock is misplaced in the warehouse. See also § IV.2.C, Hidden Agenda, below.

The term "pitfall" as used here indicates a danger which is hidden but generally avoidable. Another metaphor is a bomb or booby trap which can hurt someone who blunders into it unaware but which can be defused by someone vigilant. However the last two (marked ✝) are deadly; do not attempt to defuse them.[6]

B. Aspects of Perspective and Perception

There are several dimensions to a client's perspective. Variation along these dimensions

5. Systems allowed to develop naturally, without attention to design, are inflexible and indeed brittle. A major goal of careful analysis and design is to provide flexibility.

6. As an aside, this symbolism violates the spirit of section IV.3.D below, in that a cross has inappropriate religious connotations. If a skull and crossbones symbol were available, it would be much better.

may arise from different people, from one person in different roles, or just from the human trait of inconsistency.

The normative/actual distinction is discussed in section IV.2.B.

The official/informal distinction arises when organization policies prescribes one behavior but staff members say they do another. This is not the same as the normative/actual distinction but a refinement of the normative. Staff members may tell you policies and procedures but not follow them (the normative-official conflicting with the actual). Staff members may well know their actions violate policies or procedures (in which case the normative-informal coincides with the actual but not the normative-official). Staff members may even say they do not follow the rules when in fact they do (in which case the actual coincides with the official but not the informal).

The human/instrumentality distinction spans the difference between concern over people impacted by the system and the means used to implement it. Indiana University's two character building codes are an example where technological concern (space used by the codes) predominated over the human (ability of people to comprehend the codes), seen each fall when hordes of freshmen descend upon `LI`ndley Hall (but coded `LH`) looking of the `LI`brary (coded `LI`).

On the personal/professional dimension, people naturally perceive the personal ("how does it affect me?") more easily than the professional ("how does it affect my job?"). Major discrepancies between personal and professional aspects of a job is often a source of resistance to change (see § IV.2.A immediately following). The personal perspective, in this classification, is related to, but not the same as, the fact that most people have an egocentric view, in that they perceive the world first in its relationship to them.

Along the goal/method dimension, it is much more common for staff to perceive the method (what they do) rather than the goal (why they do it). If you can get them to think beyond "the boss told me to do it," you will obtain better insights.

The manager/subordinate dimension is just a reflection of the organizational hierarchy. The higher up the organization hierarchy, the broader the view.[7] This is also a good example of the fact that these dimensions are not independent, in that a manger is typically more toward the normative, official, professional, and goal-oriented ends of the respective scales. The manager/subordinate dimension is actually broader, in that any of the distinct roles within the client organization may have particular perspectives.

Although we may not approve, we must recognize that perceptions are sometimes made according to gender or other stereotypes. A once-common example was the attitude that data entry and other computer interaction was "women's work" because it involved a keyboard.[8] Sadly, stereotypes and prejudices sometimes are more persistant than rational evaluations and all we can do during the analysis phase is to note their presence. However, we should then try to design and implement a system to "empower" employees hurt by

7. At least that's the normative statement; *Dilbert* maintains the actuality is just the opposite.

8. The immediate popularity of the point-and-click interface among managers may be a consequence of this stereotype.

prejudice and discrimination – thus enabling the secretary who wields the keyboard and controls the information to rise to an executive assistant.

C. Dealing with Perspective Issues

Dealing with clients' perspectives involves two steps. First, you must recognize the other person's perspective and be able to view the world from their. Second, you must transform transform their world view into a standard frame of reference.

It helps to think of perspective as used in graphic arts, by painters from Giotto to da Vinci (where perspective provided a meeting place for mathematics and art). In order to have realistic sizing and placement of objects in a scene, it is necessary to understand the geometry of light traveling from the objects to the eye of the artist and to recreate that geometry between the painting and the eye – thus the scene and the painting share the same perspective. Now computer graphics engines readily transform one perspective to another, allowing real-time rotations of objects and scenes. This metaphor thus encompasses both notions of viewpoint and transformation. Indeed, this metaphor extends into areas of Enterprise Architecture, where both viewpoint and view are prominent factors.

It is not a trival task to "put yourself in someone else's shoes". Substantial practice helps. If you have explained, say, a web browser to an elderly relative you have begun such practice. The first step is to understand their vocabulary and adapt to it. Very often you use the same words, with quite different meanings – you say "Use the mouse" and your relative thinks of an unplessant rodent. Next, understand and characterize their point of view, classifying the individual's perspective according to as many of the above dimensions as are relevant and trying to capture what aspects of the system are really important to them. Confidentiallity and respect for the person are essential here; an employee is not going to let you know that chatting with relatives is imporant if that will get back to the boss. Choose your words carefully, direct phrasing of a question will often disquiet the employee and cut off much valuable communication. As an extreme example, "Do you like your boss" will only raise concerns while discussion of job satisfaction will most likely address the boss, albeit more circumspectly.

Pertaining to transformations, recall the perspective metaphor and identify the differences in the coordinates of the two viewpoints. In this case, the goals and objectives of the individual and the proposed system serve as coordinates. Of course, some viewpoints cannot be transformed into others; the view of the mountain from the east at best provides a mere outline of the mountain's west face.

2. Other Pitfalls

There are several additional pitfalls – potential albeit unlikely pitfalls – in the information received from the clients which may cloud the analysis and even distort the entire project.

A. Resistance to Change

Perhaps the most commonplace difficulty facing new computerization projects is resistance from the client's staff. This resistance is sometimes overt, which is a relative benefit because the problem is out in the open and can be dealt with there. Sometimes it is covert, which is more difficult to deal with because it is not evident. Many kinds of resistance are operational (a high incidence of data entry errors, for example) and as such fall outside of the purview of the Information Systems project.[9] But staff resistance that takes the form of withholding or shading information can make project analysis more difficult.

The most common sources of resistance, and some hints for dealing with them, include:

i. "I'll never learn how to use a computer."

In spite of the wide presence of computers in the modern economy, *computer phobia* is still a serious factor. Many people, when hearing of a mouse, respond by grabbing a broom. When dealing with such people during the analysis phase, focus the interview away from the potential computerization. Make the interviewee feel confident in the skills they do have, by carefully but respectfully looking at their daily routines.

When a client has staff with computer phobia, special attention must be paid to make the user interface non-threatening and reversible and to make the documentation clear and simple. Good management should be aware of this problem or should be made aware. Emphasize to management that a few computer games will help the staff learn to deal with computers and greatly alleviate stress; in this way games can actually increase productivity if they do not get out of hand.

ii. "The computer has it in for me."

Personification of the computer is really an extreme case of computer phobia.

iii. "It will take my job."

Concern about loss of employment has been felt since the first commercial applications of computers. Although this is sadly coming to pass at last, the kind of projects that Information Systems teams implement will never take anyone's job away.[10] But this historical fact will not reassure someone who feels threatened. The best you can do is emphasize that the proposed system is a tool that will help them, not only keeping them in the loop but making their skill and judgment more essential. Do this not by lecturing about the

9. In other professional settings, you may be serving both as a computer consultant and a management consultant and have to deal with all of the staff resistance issues in the latter capacity.

10. A computer will only replace a person if that person is doing a very limited job, with a small task set that can be entirely automated, or if there are many people whose collective task load will be reduced. This situations never occurs in the small office environments in which Information Systems projects occur.

future of computing but by constantly presenting the interviewee with the attitude "we want to help *you* do *your* job better."

iv. Mr. Lucas

"Mr. Lucas" is named for Lazy User Computer Apathy Syndrome.[11] It describes users who "won't bother to learn to use software effectively, even when they receive training."

However, there is an issue of perception here too. If an employee appears lazy because he won't lift heavy boxes, perhaps a joint is stiff and physical therapy is necessary. If an employee appears lazy because she won't learn the difference between "Word" and "Windows," perhaps she lacks the cognitive framework to make this a meaningful distinction.

B. Normative *versus* Actual Behavior

This pitfall is to some extent a matter of perspective, but it is such a major trap that it deserves special emphasis.

There is often a big difference in what people say they do and how they really behave. This clash is humorously seen in statements that violate their own prescriptions, as in "a preposition is a poor thing to end a sentence with." Anthropologists have long understood that this difference could confound their field research and have given names to these tendencies: *normative behavior* is what people say they do, as opposed to their *actual behavior*. And this distinction is as relevant in an office as it is on Pago Pago. Many management texts point out that informal communications are often more important than communications that follow formal reporting channels.

In the area of information systems, the normative/actual distinction applies to what information is important, who maintains and uses the information, and what decisions are based on which data. It also applies to personnel matters such as the staff's experience and comfort with computing technology.

You must use the same technique for dealing with the normative/actual dichotomy as anthropologists use: do not rely only on what people say; validate their statements by observing actual behavior.[12]

C. Hidden Agenda and Other Problems of Client Politics

Sometimes, fortunately quite rarely, an information systems project is initiated for entirely wrong reasons.

The first such political pitfall is the "hidden agenda". For example, a project is initiated by one department in order to send a message that there are too many errors in data entered by a second department. In such cases there will be a serious difference between the stated goals and the real, but hidden, goals of the project.

11. *Information Week*, Feb. 20, 1995, p. 44. While such jargon is fun for us to use, you must never, ever, use it to the client's face.

12. Other aspects of anthropologist's field methodology are not necessarily recommended. It has been said that anthropology requires a strong stomach (to consume fermented mare's milk or some other local brew) and a high tolerance for alcohol (to be able to remember the tidbits learned while consuming that brew).

The second type of problem is when there are strong conflicts in the goals of different individuals or different units. If these are true differences, there is nothing that you, as an outsider, can do to influence the situation – it is likely that the conflicts run deep. However, if these differences are due to misperceptions or other inadequate information, you may be able to resolve the situation, particularly if past information systems did not serve all units.

The third problem occurs when the responsible people in the client organization themselves have inadequate goals. This may happen when the primary client is responding to an order from above without adequate guidance and motivation. A symptom for this problem is lack of response or indecision by the primary client.

In the Information Systems class, report such a situation to the Instructors once you have strong suspicions it is occurring. There is much agony if a project team is caught in the cross-fire of the client's political battles.

In other professional situations, leave immediately once you have verified that you are truly in such a nasty predicament. You may loose in the short-term, but in the long-term your professional reputation will be intact.

D. Lag Time in the Client's Organization

Excessive lag time can appear as delays in making decisions, as the need to schedule meetings weeks in advance, as delays in delivering materials, and in many other ways. This is a pitfall of a different flavor than the above, in that it involves the process within the client's organization rather than the frame of mind of members of the client's staff. It is still a serious pitfall into which many teams have fallen. Moreover, it is particularly serious because this project is free to the client. If they were paying their money for the services you provide, they certainly would perceive a higher stake in your efforts. This is unfortunate because the biggest stake in any project like this is not the cost of making it succeed but the potential cost of failure.

Because the causes and symptoms of lag-time problems are highly varied, there can be no general prescription for dealing with these problems. If you team gets bogged down because the client does not respond in a timely manner, ask the Supervisors and Instructors for assistance.

3. Dealing with the Client Issues

There is a variety of ways in which you can improve the client's perspective and facilitate your interactions with the client. Issues related to client interviewing are discussed in § V.3.A on page 57.

A. Anticipate

As with virtually everything in this area, using anticipation to handle client factors has both process and product components. On the process side, anticipate those things you will need to do to rectify for the clients perspective. On the product side, design and build a system that accommodates change.

The most important factor in handling the client's perspective is for *you* to be aware of and compensate for that perspective.

In building a system to accommodate change, you can sometimes anticipate specific changes but other times you can only know that some changes will occur. One change that almost always occurs when a new information system is installed is that the management begins to ask questions; hence a statement that a query capability is not necessary only applies to the present and not to the future. The good side of this anticipation is knowing that users can and will adapt, provide you have addressed the resistance issues discussed in IV.2.A above.

B. Build Confidence

You should conduct yourself so as to cause the client to have confidence in you. One facet of this is almost literally to put yourself in the client's shoes - or at least the client's "uniform." That is, when meeting with a client, wear a costume that the client will recognize and classify you as a knowledgeable professional. For many years, IBM was known for the blue suits and stripped ties its staff always wore.[13] This dress code existed not merely because someone at IBM liked striped ties – the IBM "uniform" was recognized as a sign of professionalism by current and, most importantly, potential customers.

Dress code is a simple but tangible example of building confidence by fitting into your client's environment and expectations. More significant (but more difficult than putting on a tie) is to fit in with the client's environment by learning the vocabulary and practices for the client's area of business and, if possible, specifically for the client's organization. Anyone who has dealt with an auto mechanic, for example, understands the benefit of "knowing what you are talking about." Vocabulary and business practice are in any case essential to the specification and design (as "domain knowledge," discussed in § V.1.B) – the point here is that doing your homework before the meeting with the client has extra benefits.

On the other hand, avoid being labeled a "techno-nerd." This means not only avoiding introducing jargon but avoiding being drawn in when someone else introduces jargon into the conversation. For example, a client may say "my FAT is corrupted", having heard this from someone else without being really certain while this affects his computer or his diet. If you begin a detailed discussion of Microsoft's disk representation you have been unwittingly drawn into the techno-nerd trap.

On the other hand, a professional commentator notes that "begin seen as a nay-sayer is an even more grievous sin than being marked as a nerd."[14] He recommends that we adopt such tactics as saying "There is a fundamental question here" rather than "There is a fundamental problem here".

13. Humorously, it was asked whether they wore ties in the shower. On the negative side, this brings up issues of gender.

14. H. W. Lovelace, *Information Week*, Sept. 2, 1996.

C. Educate Your Client

In the course of this project, you may need to educate your client about your process, about the impact of computer systems, perhaps even about aspects of their own organization.

The first step in educating the client is of course to identify their needs in this area. They may be inexperienced with computers. They are more likely to be inexperienced with information management and, if so, they will probably confuse this second condition with the first.

Make your client aware that change in computer systems is both inevitable and costly. They tend to view acquiring a computer system as similar to acquiring a new copier, which is stable for many years provided it gets cleaned monthly. While you are doing your development, this can be accomplished through the use of a formal change request mechanism.

In professional settings, an information systems consultant often needs to "reform"[15] his or her client. For this course, we try to avoid projects with this requirement but can't always foresee such situations. Therefore, if you feel that you have gotten into a situation that requires major, problematic changes in the client's policies or procedures, you should inform your supervisor and the instructors. If you must "reform" the client, the following steps have been recommended: (i) lay the groundwork by helping the client's staff understand the problems with the current policies and procedures, (ii) choose the moment to make your case to make sure that responsible individuals in the client's organization can focus on the problems and solution that you present,[16] and (iii) insure the client "buys in" with appropriate follow through.

Another good approach, which treats the client's perspective in an active rather than a passive way, is to engage the client in analysis. In companies with an internal data processing organization, the technique known as "client centered development" requires that members of the client's staff actively take part in the analysis and often in the rest of the life cycle phases. For example, in the university, the Registrar's office, the Human Resources office, and others have full-time staff members who work with the computer center on information systems development.

D. Present Ideas in Ways Appropriate to the Client

While the previous subsection discusses ideas that you should communicate to your client, this subsection addresses ways to make that communication more effective.

Be careful what you call things. For example, common terminology in specification documents classifies requirements as either "functional" or "non-functional." The analyst understands this distinction but the client is more likely to understand "non-functional" as

15. The word "reform" connotes not only change on client's part but also difficulty in comprehending or achieving the change.

16. The intent is to avoid distracting immediacies which interfere with the staff's ability to listen and respond.

equivalent to "not working" and think "why are we bothering to specify things that don't work?" Valuable time and client rapport is then expended getting the discussion back on track.

Many aspects of system definition and implementation will run counter to the client's instincts and perceptions and therefore clients will sometimes balk at your suggestions for requirements. A good way to alleviate this problem is to illustrate your ideas with examples. There are two opposing schools of thought concerning examples, such as the content of the sample screens. One recommends making the example as close to the target application as possible, because that will engage the interest of the client. The other recommends a fictitious example, because that helps the client (and the team!) to focus on the interface rather than the functionality. However, if you use examples from the client's context, they must be correct. Say, for example, that you have sample screen showing Acme Corp. on Elm Street but the client knows the real address is Oak Street. The client is immediately distracted by this "error" and begins, however slightly, to loose confidence in the reliability of the project team.

Another way of communicating abstract ideas is to use metaphors. For example, in data modeling in a medical assistance organization, entities Employee, Patient, and Volunteer should be generalized into Person even though the client does not see these as connected. The issue is the necessity of connections that are not immediately visible. One metaphor that applies is plumbing: we do not shower in the kitchen sink nor wash dishes in the toilet, but we must connect those facilities. The design of the house must consider those pipes. While slightly humorous metaphors such as the above are not appropriate for some clients, they usually help catch the client's attention.

An implication of the preceding subsection is that the Feasibility Study and Requirements Specification documents can in fact be educational documents for the client's staff, giving them a perspective on their daily operations that they don't get in the daily press of business. Consequently, you must remember that you are writing educational documents: write clearly, do not assume that the reader knows things, use simple examples, *etc.*

E. Two-way Interaction

Much of the above discussion deals with things you should tell the client. But of course the most important aspect is to get the client to provide information. This is essentially just a prescription to "let the client talk". – and of course to listen carefully.

Techniques for information gathering are discussed later in section V.3.

4. Coda

Although the above discussion sometimes appeared negative, it is not intended to demean clients and users but to sensitize you to the fact that they have different motivations and perceptions than you do. Often they are working under considerable pressure. However, the impact of client factors – if they are allowed to become problems – can be exceedingly negative.

Consideration of client factors also raises an important issue of professional responsibility – an issue not for this course but for you longer career. As a professional, you have knowledge and hence power. You can use this power to do what your clients really need or to do what the clients think they want. Do not act like the tobacco companies, which sell cancer while claiming "we only sell what our customers ask for."

Exercises

1. For each of the benign common pitfalls listed in § IV.1.A (that is, all but the last two), discuss (1) indicators which help you recognize that pitfall and (2) ways by which you can correct for that perceptual problem.
2. Characterize each of the pitfalls bulleted in § IV.1 in terms of the perspective dimensions discussed later in that section.

V. Feasibility Study

The feasibility analysis is intended to determine whether it is worthwhile to engage in the proposed project, whether or not the project can be finished in two semesters, and whether the client is willing and able to support the team throughout the entire project. Although "study" and "analysis" are essentially synonymous, we use "Feasibility Study"[1] to refer to the product (the milestone document) and "feasibility analysis" to the process that produces the document.

It is difficult to distinguish between feasibility analysis and requirements analysis processes; hence we use the single word "analysis" for the overlapping processes. However, we can clearly distinguish the questions that each of the respective documents is intended to answer. The Feasibility Study asks "Should we do the project?" The Requirements Specification asks "What functionality should the system provide?" The related term "systems analysis" covers the first half of the lifecycle, from feasibility through requirements and into design.

A good evaluation begins with the problems faced, suggests several alternative solutions, weighs the benefits and disadvantages of each alternative, and recommends one alternative. In a commercial setting, several alternatives of different costs are typically recommended; then the client selects an alternative depending to fit their budget. A feasibility evaluation that begins with the proposed system and presents a "take it or leave it" decision cannot do justice to the issues.

This chapter begins by summarizing the analysis process, including examining the knowledge necessary to conduct a feasibility analysis. The second section suggests an checklist/outline for the entire document. That is followed by a section on methodologies for information gathering.

Remember that the purpose for writing this document is to determine the feasibility of a project. This means that your grade on this document is determined by the accuracy and clarity of your analysis and not by whether or not the project is continued.

1. Background

A. Analysis Process

The analysis process is an iteration of investigating, understanding, and modeling. It is no surprise that analysis requires investigation and understanding – we only need to know how to investigate and what to understand. The requirement that analysis must include modeling, on the other hand, is surprising or at least novel (hence §V.1.C and much of Chap. VII deal with models). Furthermore, the fact that the iteration includes all these steps is important.

You should aim to involve all stakeholders in the analysis process. Stakeholders include those who provide or consume information managed by the proposed system, those who

1. Note that "Feasibility Study" is capitalized, to indicate that it is a named object.

will operate and support it, and those who will use the software and interfaces of the system. Because the success or failure of the system is determined by the latter group, this approach is known as *user-centered development*.[2] A user-centered approach thus necessitates listening to the users and other stakeholders, understanding the users (both their technical and human needs), and modeling the users.

The full project process mechanisms are not yet in effect for this Milestone. The short time frame and small two-person teams do not justify the overhead of these mechanisms, which include weekly progress reports and supervisor meetings. None the less, each team will be assigned a supervisor who will make certain that the milestone is proceeding on track and provide a voice of experience.

B. Necessary Knowledge

There are three basic sources for the knowledge necessary for specifying and designing a system: facts and skills that you know outside the scope of the project,[3] information that you discover during the analysis that you perform for this project, and factors that you anticipate. Table V.1 gives an overview of this information.

	know	**discover**	**anticipate**
intellectual skills	programming abstraction		
information	ER modeling & relational design methodologies	content & use	new applications more relationships & entities
	——— domain knowledge ———		
operations	process analysis methodologies	user activities & operations	additional uses of information
	——— domain knowledge ———		
technology	options available in current technology		more networking better UI's new versions of platforms
	——— platform tools ———		
personnel	client communication skills	roles attitudes	expertise leads to new expectations

Table v.1: Necessary Knowledge

In the "know" column, the most important resources you bring with you to an in-

2. Perhaps "stakeholder-centered development" would be a more accurate designation, but "user-centered development" is the accepted term, as well as being easier to say.

3. Because the project is part of the Information Systems course and because you learn much of the applicable methodology during the project, it would be incorrect to describe these facts and skills as "things that you know before the project begins," although this statement is correct for projects later in your professional career.

formation systems project are analysis and design skills. Your computer science training has already given you the ability to cope with the complex details of programming and programming environments. You have also become skilled at rising above the swamp of details through abstraction. That ability to abstract is what sets you apart from hackers and hawkers. It is also essential for a project of this size and scope. The ability to abstract the right information model means that the data maintained by the system will be more useful, consistent, maintainable, and adaptable. The ability to abstract the process of the client's organization and the resulting computer-based applications means that the users will experience a system which has better flow from task to task, fewer errors, less hassle, and greater effectiveness. The remainder of the 'know' column is the content of the Information Systems course.

The information you gain specifically for this project is of course the heart of this and the following chapter.

Domain knowledge, which is general knowledge about your client's area of business, includes both outside knowledge and project-specific information and therefore spans the first two columns. For example, having worked in a stockroom will give you general insight into the requirements for an inventory system, even if your client keeps their inventory in ways very different from your previous employer. Some domain knowledge is more consistently applied than others – all businesses follow essentially the same accounting procedures, because of tax record-keeping rules and other legal requirements, while every business handles its inventory differently. Knowledge of platform tools is shown spanning two columns as well because, in spite of your professional background, there are always project-specific tool factors.

It is the final area, anticipation of factors to come, that is likely to be most critical for the long-term maintainability of the system. It is not enough to know that things will change, although poor system analysis and design seems to forget even that simple fact. It is also not responsible to throw up your hands and claim that you can't anticipate the future. There are a number of things that you can and should anticipate. First and foremost, once data has been collected and organized into information, that information will find new uses and new users. Almost immediately the new information will facilitate new decisions and hence cause new paths in the processes. These paths will in turn need additional information – if maintaining that information requires major additions to the data model, the change will be costly. Hence the anticipated factors should be, as much as possible, built into the specification and design of the system. Because judgment and experience are most essential here, be sure to ask advice from the course instructors, who will in particular look at the draft ER model with these factors in mind.

C. Model-Driven Process

The recognition that models were central to systems development processes resulted in a major improvements in those processes. That recognition which came toward the end of the twentieth century, held that analysis should result in models and that design and development consisted of elaborations and transformations of these models. A methodology built around this idea is known as a *model-driven process*. A major tenant of this approach

is that the initial models should be expressed in simple yet precise notations – simplicity so that clients (and even users) can understand and therefore expand and correct these models. We will return to this tenant, stated as the need for models to communicate as well as specify, often during this course. The first consequence of this tenant is that a variety of models is required to exhibit various views of the system.

With such a variety of models, it is important to have good conceptual framework that organizes the models, facilitating both the analysis process and the presentation of the results of the analysis. Although any well-thought-out framework will do, the Zachman Information Framework is particularly well-suited for the task at hand. The Zachman Information Framework Architecture is intended to support analysis and modeling at the enterprise level – that is, the broadest and most inclusive scope.[36; 30]

The Zachman Architecture begins by identifying two broad categories for organizing ideas: *interrogatives* and *roles* and using these as dimensions for organizing the analysis. There are many conventions for labeling the dimensions, but the most common uses (some variation of) "*what*", "*how*", "*where*", "*when*", and "*why*" for the interrogatives and "*specify*", "*design*", and "*build*" for the roles. A major contribution of Zachman's was the realization that roles are ordered from the abstract ("specify") to the concrete ("build"), while there is no special order among the interrogatives.

Table V.2 shows how these dimensions designate cells in a generic Zachman frame, along with another labeling of the interrogatives which focuses on the answers rather than the questions themselves. These cells may contain contain further frames in a recursive expansion of detail. The critical observation here is that an object in one cell, say the upper left or ⟨"specify", "data"⟩ cell, has aspects of all roles and all interrogatives when examined in further detail.

The items that are placed in the cells of a framework are the models and other components that describe the organization. Note that the roles generally correspond to the project lifecycle. As we progress through the lifecycle, we will populate the framework cells with components appropriate for the particular phase. For now, the use the interrogatives to organize your thoughts about the project – and be sure to ask all the questions!

	what *data*	**how** *function*	**where** *location*	**who** *people*	**when** *time*	**why** *motivation*
specify						
design						
build						

Table V.2: A Zachman Frame

2. Content of Feasibility Study

The following is a checklist and an initial outline for a Feasibility Study. Because it strives for generality, it cannot be an outline precisely suited for any individual document. Therefore the outline should be used as a starting point, adapting it to your project. Furthermore, some of the particulars overlap, creating redundancies which should be removed in writing your document.

In order to have a general picture of the Feasibility Study, Table V.3 presents a suggested table of contents for the Feasibility Study(that is, an outline of the outline). Although the checklist will not be suitable for every detail of your particular project, the more general Feasibility Study outline is highly recommended.

1. Executive Summary
2. Client Environment
   - 2.1. Identity
   - 2.2. Organizational Structure
   - 2.3. Personnel
   - 2.4. Commitment
3. System Characteristics
   - 3.1. Goals and Objectives
   - 3.2. Classes of Stakeholders/Users
   - 3.3. Scope – Other Information Systems
   - 3.4. Current System
   - 3.5. System Needs
4. Technical Environment
   - 4.1. Hardware Platform
   - 4.2. Networking
   - 4.3. Software Platform
   - 4.4. Resources during Development
   - 4.5. Client's Experience with Computers
   - 4.6. Project Knowledge and Skills
5. Evaluation of Solutions
   - 5.1. Possible Solutions
   - 5.2. Benefit-cost Analysis
   - 5.3. Best Alternative
   - 5.4. Implications of Recommended Solution
   - 5.5. Other Factors
   - 5.6. Conclusions

Table v.3: Typical Table of Contents for a Feasibility Study

Checklist and Outline for Feasibility Study

Feasibility Study 1: Executive Summary

An Executive Summary is a condensation of the important aspects of the entire document. It must cover background, important facts presented in the document, and major conclusions; operational conclusions that require further action are especially important. An Executive Summary is thus more than an Introduction although it begins like an Introduction.

An Executive Summary is typically one page long; it is never more than two pages.

Feasibility Study 2: Client Environment

Right from the beginning you must begin to understand the client – who they are and what they do. Understanding their goals will help you in the next phases of your project and possibly determine whether or not the project is truly feasible. Below is a list of points designed to assist you in describing your client.

Feasibility Study 2.1: Identity

F.S. 2.1.1. What is the name of the organization?

F.S. 2.1.2. If it is part of a larger organization, identify that parent organization (for example, part of Monroe County government).

F.S. 2.1.3. What kind of organization is it (private, not-for-profit, *etc.*)?

F.S. 2.1.4. What are its areas of business and types of operation (manufacturing, medical, *etc.*)? If the client is part of a larger organization, briefly characterize the parent but be more specific about the client.

Feasibility Study 2.2: Organizational Structure

F.S. 2.2.1. Components: If the organization has any relevant subcomponents (divisions, subsidiaries, *etc.*) describe those. In particular, units that will use or interact with the proposed system must be discussed. Also, discuss an information system department or any equivalent unit that supports the current or proposed systems.

F.S. 2.2.2. Locations:

F.S. 2.2.2.1. If the organization has several sites, identify and locate those. If there is only one site, this is a good example of where to shorten this outing.

F.S. 2.2.2.2. Map organizational structure into sites (*e.g.* GE manufactures refrigerators in its Bloomington plant).

F.S. 2.2.2.3. What is (are) the primary site(s) for this project?

F.S. 2.2.3. Future structure: What structural changes within the client organization are under consideration or under way?

Feasibility Study 2.3: Personnel

F.S. 2.3.1. Primary client: The primary client is the person who is responsible for the specification and ultimate operation of the proposed system. The primary client should have the authority to commit resources to the project: both the staff time to meet with the team during design and development and the hardware and software platforms to implement and operate the system, or the funds to purchase those platforms.

F.S. 2.3.1.1. Who is the primary client? Include address and telephone as well as name and title.

F.S. 2.3.1.2. What is her/his role in the organization? Is s/he empowered to make the necessary commitments for this project?

F.S. 2.3.1.3. Did s/he request the system? If not, who did?

F.S. 2.3.1.4. Are her/his objectives for the proposed system clear?

F.S. 2.3.1.5. If the primary client holds the authority and responsibility for ongoing operation and maintenance of the proposed system, include item 2.3 here and omit the rest of § 2.3.

F.S. 2.3.1.6. Will s/he be regularly available or is s/he likely to be away from the office for extended periods (*e.g.* while traveling)? A primary client who is unavailable is a serious problem: make sure backups are available.

F.S. 2.3.2. Primary contact: The primary contact is the person with whom the team will have contact. S/he will provide the most general information specifying the proposed system. The primary client and the primary contact are often the same person. This is an advantage. In this case, combine the two entries. It is possible that more than one person will serve as primary contact. In this case, discuss each person who serve this role individually.

F.S. 2.3.2.1. Who is the primary contact? Include address and telephone as well as name and title.

F.S. 2.3.2.2. What is her/his role in the organization? Include the contact's relationship to the primary client.

F.S. 2.3.2.3. Are her/his objectives for the proposed system clear?

F.S. 2.3.2.4. Does s/he fully appreciate the time expected of her/him as contact for this project.

F.S. 2.3.3. Information Systems staff: The unit(s) providing support for computer and information systems were itemized in 2.2 above, the staff of the unit(s) need to be specifically described. This analysis is especially important, because the greatest risk to the longterm success of your project is inadequate support.

F.S. 2.3.3.1. List the technical and information systems support staff.

F.S. 2.3.3.2. Describe each staff member's relationship to the current and proposed project.

F.S. 2.3.3.3. As respectfully as possible, discuss the ability of the staff to support the proposed project. The intent here is to identify gaps in the skill sets of the computer systems staff, and thus propose required training or hiring to fill these gaps.

Feasibility Study 2.4: Commitment

Discuss the following requirements with your client and record their responses. These requirements apply to the commitment of staff time or other resources.

The client organization must provide:

- time for meeting with team while gathering requirements
- time for reading Requirements Specification document and viewing the Prototype

- facilities for running the system

- access to facilities for project team during Testing and Installation phases

- staff time for assisting with preparing test data

- staff time for test system operation, including data entry

Discuss all the above issues with your client, then answer the following in your Feasibility Study:

F.S. 2.4.1. Is your client willing to make the necessary commitment to the project?

F.S. 2.4.2. Is your client interested/enthusiastic about the project?

F.S. 2.4.3. Are their seasonal or other time factors which distract the client during the course of the project? More generally, do you have any concerns about the client's continuity of involvement over the course of the project?

Feasibility Study 3: System Characteristics

Feasibility Study 3.1: Goals and Objectives

State the problem in terms that are high-level, broad, and flexible. A good problem statement allows a range of possible solutions. Recall the distinction between goals and objectives from §I.3. Goals are broad aspirations addressing general benefits. Objective are the measurable (not necessarily quantifiable) components of a solution meeting the goals? List goals first, then objectives.

If necessary, differentiate overall goals for the *organization* and the complete *system* from the goals for this particular *project*. Such differentiation is necessary, for example, when the project involves updating or reimplementation of an existing system. Without the system goals as a context, project goals and objectives may be difficult to interpret. For example, if the only objective is "eliminate data entry burden", the obvious yet obviously fallacious interpretation is to eliminate the system for which data is being entered. Continuing the example, if the goal of the system is registration of some sort, then the above project objective might be met by a web-based interface where participants self-register.

Feasibility Study 3.2: Classes of Stakeholders/Users

At this point, discuss the users and other stakeholders of the proposed system with respect to the goals and objectives of the system. Discussion of their computer experience and other factors related to actual feasibility are discussed later. If the primary client is the actual user, merge the respective subsections.

F.S. 3.2.1. Who exactly are the (internal) users of the system? This means identifying people[4] and positions; roles are discussed next. If the system will have external users (users from outside the client's organization, typically *via* the web

4. In a project for a large client organization, this means identifying groups rather than individuals. Such projects are almost always too large-scale for this class.

these days), the single category 'external users' typically suffices. However, demographic information is beneficial if it is available.

F.S. 3.2.2. What role(s) does each person play in the organization? (see the exercise at the end of this chapter as well.)

F.S. 3.2.3. How many people will use the system concurrently?

F.S. 3.2.4. How are they related to the primary client? To the primary contact?

It may be advantageous to split the discussion of actual users according to major roles, such as "information providers" and "information consumers". Division according to organizational units, such as sales and shipping, may also be used; however it is usually more informative to separate by roles rather than units. In any case, clearly map between roles and units.

Feasibility Study 3.3: Scope – Other Information Systems

F.S. 3.3.1. List the other information systems with which the proposed system will interact?

F.S. 3.3.2. For each such system:

F.S. 3.3.2.1. Is it manual or computer-based?

F.S. 3.3.2.2. What volume of information is involved in its interactions with the proposed system?

F.S. 3.3.2.3. What is the frequency of such interactions?

F.S. 3.3.2.4. Is the interaction regular(scheduled), automatic (triggered), or manual? If the interaction is automatic, is there verification that the interaction is in fact occurring.

F.S. 3.3.2.5. How serious are the consequences of difficulty or failure in the interaction?

Feasibility Study 3.4: Current System

Recall §I.3 of this *Guide* – the system is always the information system. Furthermore, there is almost always an existing information system, even though it may be informal or paper-based. Therefore you should investigate that system, asking the following questions.

F.S. 3.4.1. Does the organization currently achieve the overall goals and objectives of section 3.1? If so, how successfully?

F.S. 3.4.2. What are the sources of the data?

F.S. 3.4.3. How does the organization store the data - file cabinets, spreadsheet, *etc.*?

F.S. 3.4.4. What reports are prepared? How are they prepared (spreadsheet, manual, *etc.*)? Who prepares the reports.

F.S. 3.4.5. What happens to the data after it is processed? That is, how is the derived information currently used within the organization?

F.S. 3.4.6. What features and functions of the current system must be carried forward into the new system? What functions can or must be altered?

F.S. 3.4.7. Is it expected that current data be converted to the new system?

F.S. 3.4.8. If the current system is computer based, is source code and documentation available and useful? Do not report details here, only discuss whether this material be available for the Requirements Specificationand is generally understandable. For example, "C source for the project is available but it is 'spaghetti code'." is an appropriate item to report here, even if that report does not bode well for the subsequent project phases.

Feasibility Study 3.5: System Needs

The range of issues which could be covered here is immense. This *Guide* can cover only a small fraction of the possible issues. These are just suggestions to get you to understand what is going on at the client site.

F.S. 3.5.1. How will the proposed system help the client achieve the overall goals and objectives?

F.S. 3.5.2. Information

F.S. 3.5.2.1. What types of objects (material or conceptual objects) need information recorded about them? For example: Students and Courses.

F.S. 3.5.2.2. For each of the above object types, what data about are you going to store about objects of that type? Remember that, at this point, you are specifying needs, not implementation specifics. Hence your view of the data at this stage should be more abstract than a file structure. This abstract view will be expanded into the Entity-Relationship model for the Requirements Specification. File structures will be designed even later. If information from a current system is to be carried forward, just mention that fact here; you should have already discussed the current system in section 3.4.

F.S. 3.5.2.3. For each of the above object types, roughly how many instances of objects of that type will be represented in the proposed system?

F.S. 3.5.2.4. What are the sources of information for the various object types? In particular, what information is imported from other systems and what information is to be captured from non-computer sources.

F.S. 3.5.3. Operations

The preceding questions focused on the characteristics of information in the proposed system. Now we focus on the activities that generate, maintain, and use this information.

F.S. 3.5.3.1. What non-information system operations cause data to be entered into the information system. For example, stocking a warehouse generates data for

the inventory system.

F.S. 3.5.3.2. What manual data entry tasks will the proposed system entail? How and by whom will these data entry task be handled? Note that data entry may include both the keying and the validation of the data and may be done internally or externally. Internal data entry should be related to the roles identified in §3.2.

F.S. 3.5.3.3. How is information imported into and exported from the current system? How will this change with the proposed system? Consider the general nature of the import and export processes, how they occur, and how or by whom they are initiated. See also Scope, §3.3.

F.S. 3.5.3.4. What reports should be generated?

F.S. 3.5.3.5. What other manual operations will be needed on the system?

F.S. 3.5.3.6. How is information in the system distributed and otherwise accessed?

F.S. 3.5.3.7. How and by whom is system information used in decision making?

F.S. 3.5.4. Conversion

What kind of assistance will be required to move from the current system to the one implemented by the InfoSys Team? This issue is very open ended but can include technical issues like converting data from one file format to another. It almost always includes a training component.

Feasibility Study 4: Technical Environment

In a perfect "start from scratch" systems development project, many of the following issues would still be undecided and many suggestions could be made (see §F.S. 4.5.3). However, today most clients already have computing facilities with which compatibility is mandatory. If acquisition or upgrading of hardware or software is necessary or possible, do discuss the alternatives in section F.S. 4.5.3.

Feasibility Study 4.1: Hardware Platform

F.S. 4.1.1. What current computing facilities does the client have on hand? A general answer is all that is required: "PC's running Vista" or "Mixed PC and Mac environment" are answers at the proper level. Is the hardware and operating system environment likely to over the course of the project?

F.S. 4.1.2. On what platform(s) will the system ultimately run? Again, avoid excessive detail. A general idea of processor speed and dick capacity is all that is required.

F.S. 4.1.3. Are the client's machines to be used for system development or will development occur off-site, requiring a specific installation step at the end of the project?

F.S. 4.1.4. How is information distribution supported? Is there a server?

F.S. 4.1.5. If it is likely that additional computing facilities will be required, what are the general characteristics of these facilities? When will they be acquired? (The client's commitment to provide these resources was addressed in §2.4.)

F.S. 4.1.6. Is your access to the machines going to be restricted? Elaborate. (see also §2.4)

Feasibility Study 4.2: Networking

If the project will run on multiple hardware platforms, briefly describe the network connecting those platforms.

F.S. 4.2.1. Is there a local network already in place? If so, describe it. If not, does the client expect your team to help specify the network (which would have obvious implications on skills needed by the team, see §4.6).

F.S. 4.2.2. Is there an Internet connection already in place? If so, describe it. If not, does the client expect your team to help establish this connection.

F.S. 4.2.3. In the unlikely case that the client uses some proprietary wide-area network rather than the Internet, describe that network and its potential impact on the project.

F.S. 4.2.4. Is the client organization currently or potentially able to manage a distributed application?

Feasibility Study 4.3: Software Platform

The underlying assumptions for all other Computer Science classes are that a software platform with sufficient functionality is readily available and that instructional staff are very familiar with that platform. Because the appropriate platform is project specific, neither of those assumptions are valid for this project. In particular, the instructors and supervisors are not experienced in all relevant software tools, although they are aware of how to approach new software tools

F.S. 4.3.1. Which DBMS are you going to use? It may be necessary to evaluate several possibilities. On the other hand, it may be that a specific platform is mandated for compatibility reasons.

F.S. 4.3.2. Is the DBMS software available? If not, when is the client going to purchase it?

F.S. 4.3.3. What other software tools will be needed (such as C++ or Java development environments)? Again, consider a range of possibilities unless specific ones are required. Carefully evaluate how the tools interface with your DB platform and whether they *really* provide the necessary capabilities.

F.S. 4.3.4. Are the tools available? If not, what is their cost and when will the client purchase them?

F.S. 4.3.5. Is a comparable software environment available on campus (or elsewhere with easy access for all potential team members)?

Feasibility Study 4.4: Resources during Development

Section 2.4 dealt with the client's commitment to provide material and personnel resources. This section focuses more narrowly on technical resources.

F.S. 4.4.1. Are necessary paper information sources (see §V.3.D below) available?

F.S. 4.4.2. If the software platform is mandated, will documentation be *always* available?

F.S. 4.4.3. If an existing computer information system is to be upgraded or replaced, is source code and *meaningful* documentation available?

Feasibility Study 4.5: Client's Experience with Computers

F.S. 4.5.1. Does the client's staff have experience using computers? If so, what software and for what tasks?

F.S. 4.5.2. If there will be multiple users, does the staff have experience using and managing shared information? In particular, is there one person with designated systems management responsibility?

F.S. 4.5.3. Does the client have relevant technical support available, either in-house or through a consultant? Relate to §2.3.

Feasibility Study 4.6: Project Knowledge and Skills

This section is unique within the Feasibility Study because it concerns the project team, not the client or the proposed system. It should consider both the current project team, that is the two of you writing the Feasibility Study, and what is desirable in the full project team.

The knowledge and skills required to successfully complete an information systems project should be evaluated; the variety is likely to be surprising. Begin this evaluation in the context of your current team, even though it is only made up of two people. Ask both general and specific questions: Do you know about the type of work done by your client? Have you used the relevant software tools? Have you used the relevant computer or operating system? The first column of Table V.1 is a good place to start. It is more important to indicate skills that you lack rather than skills that you possess.

This evaluation will not only help determine whether the project is feasible but will assist in determining what skills are needed to complete the team. Therefore you should describe necessary knowledge or skills that the team lacks as well as those that it possesses.

Feasibility Study 5: Evaluation of Solutions

This section presents the range of possible solutions and their respective benefits and costs. It is where your team determines what they feel is the best solution and recommends whether or not the project should be continued.

Feasibility Study 5.1: Possible Solutions

F.S. 5.1.1. What are the alternative solutions to the problem introduced above? Discuss.

Alternative solutions *might* include:

- do nothing.
- a new paper system.
- changing management procedures – in the case that difficulties are not with the information but are with the way the client uses information. Do not expect the client to respond happily to this suggestion, but do not hesitate to make it.
- off-the-shelf software – for example, a commercial accounting package. Even inexpensive accounting packages contain much more accounting expertise than we can provide.
- a system developed specifically for this application – possibly several alternatives.
- a "deep" prototype – this differs from the exploratory prototype required of every project, as described in Chapter VIII. While the required prototype explores your solution to a problem and the tools required to provide that solution, a "deep" prototype explores the problem itself. A deep prototype is suitable for a client who wants to experiment with fundamentally different business practices.
- multiple systems – the project objectives can be met by several diverse components (*e.g.* some information is maintained using an off-the-shelf package and some on paper, while custom software is required for the remainder).
- partial solution – if task seems large and there are some needs that are more important than others, it may be possible to propose a system that meets the important needs while postponing addressing the others. This is also useful if some requirements are clear while others are very fuzzy.

F.S. 5.1.2. What are the alternative hardware platforms? Discuss.

F.S. 5.1.3. What are the alternative software platforms? Discuss.

Feasibility Study 5.2: Benefit-cost Analysis For each alternative, estimate:

F.S. 5.2.1. How will it benefit the client? King and Schrems [20, reprinted in 25, p. 257] provide a list of "information system benefits." While some of the items on this list are no longer a serious concern (such as the "ability to move large files of data about"), the list provides an informative categorization of benefits:

cost reduction or avoidance	CR
error reduction	ER

increased flexibility	IF
increased speed or throughput	IS
improved management planning and control	MC

F.S. 5.2.2. How much will it cost to develop the system? (Include items such as client's time during analysis, client's time for testing, equipment and software purchases...) The cost need not be a dollar amount for items such as the client's time. Although the client is not charged for your services, it is a good educational exercise to attempt to estimate your team effort and hence the development cost.

F.S. 5.2.3. What are the start-up costs? (Time for data entry, training users...)

F.S. 5.2.4. What are the operational costs? (Cost of staff, management time...)

F.S. 5.2.5. Is the client ready for the system?

Feasibility Study 5.3: Best Alternative

F.S. 5.3.1. Which solution do you recommend?

F.S. 5.3.2. If the solution has multiple components, prioritize them.

Feasibility Study 5.4: Implications of Recommended Solution

Try to anticipate all the implications of the proposed system, asking how that system might change the clients' procedures and operations. Often system changes cascade – for example, the recommended solution requires new database software which requires a new operating system which requires hiring new technical staff for system support. As this example shows, the implications may flow over wide areas. None the less, it is worthwhile to characterize the general areas:

F.S. 5.4.1. Technical implications: The above example illustrates the kinds of factors you should consider.

F.S. 5.4.2. Operational and managerial implications: A change in the information systems can effect the operational policies and procedures of an organization. Eliminating the flow of paper may change the actual flow of information as well, which in turn may change how and where control is exercised. For example, in a paper-based order-processing system, suppose that an order passes from sales through accounting to shipping. After computerization, order entry in sales will automatically verify account status prior to shipping, thus eliminating direct control by the accounting department over order processing.

F.S. 5.4.3. Staffing implications: This may include obvious factors such as data entry or hidden ones such as staff realignments.

F.S. 5.4.3.1. New tasks: *e.g.* data entry.

F.S. 5.4.3.2. Reduced or eliminated tasks: *e.g.* less paper processing.

F.S. 5.4.3.3. Training.

F.S. 5.4.3.4. Eliminated roles or positions: *e.g.* file clerk.

F.S. 5.4.3.5. New roles or positions: *e.g.* technical support or system administration.

It is exceedingly important to recognize that any shared computerized information system requires a significant fraction of some person's time for system administration. These days sharing usually means client/server relationships across a LAN, but sometimes even a stand-alone PC qualifies if it is shared between multiple distinct roles. For example, a moderate sized office with a LAN and a single database server requires a half-time person for technical and administrative support. In a small business which has failed to anticipate and meet this need, the support role often falls on the owner or manager, causing the entire operation to suffer seriously.

Feasibility Study 5.5: Other Factors

F.S. 5.5.1. Are there any other factors that may affect the success of the project?

Feasibility Study 5.6: Conclusions

F.S. 5.6.1. Discuss your views about the client's suitability for the proposed project.

F.S. 5.6.2. Discuss your views about the project's feasibility.

Remember that the only right answer is a good solution to the client's problem. If that solution is pen and index cards, then you should recommend it.

3. Information Gathering Methodologies

Information gathering methodologies for the analysis phase can be broadly divided into those dealing with people and those dealing with artifacts.

Before discussing particular methodologies, it is valuable to consider issues related to the characteristics (not the content) of the information you need to gather.

quantity: Over the course of your analysis, you will deal with vast amounts of information. The trick is to gather and process the information without being swamped.

reliability: Exercise "reasonable doubt." Every important fact or conclusion should be validated.

scope: Make certain that the full breath of the proposed project has been investigated.

perspective: Seek out differing views of the proposed project. Map those different views into one common understanding. See section IV.1.

organization: Organize your data as you gather it but be flexible enough to restructure as your understanding grows.

refinement: Know when more detail must be sought out and when to step back from the details in order to obtain the "big picture."

relevance: Information must be filtered. A past supervisor once observed "As a software designer, I'm used to working through extraneous information everyday."

A. Interviews

The following comments apply to the Requirements Specification as well as the Feasibility Study.

i. Whom to interview

- Executives - for goals and objectives.
- Managers and supervisors - for objectives and operational overviews.
- Operational staff (secretaries, shop workers, *etc.*) - for operational details.
- IT staff. These may be explicit, with a job title that reflects an IT role, or implicit, the person everybody calls when their system crashes.
- Outside users (if applicable) - for special needs.

ii. How to interview

When we think of interviews, our mental picture is probably formed by TV talk shows. Those seem to be very casual and spontaneous, but they wouldn't work if the host hadn't found out a lot about the guest's most recent movie/love affair/play of the week. Similarly your team needs to prepare for interviews with the client.

Before the interview takes place, your team needs to lay a foundation. The need to schedule the interview is obvious, but when you schedule you should do more than just set a time and place: indicate the topic and expected outcomes of the meeting. Your team should consider the topic and outcomes at greater depth and draw up specific questions. Having these questions printed will help keep the meeting on track, since interview meetings invariably stray from to original agenda. The next step in preparation is to learn as much as you can before the meeting: the client's business area, the structure of the client's organization, the client's staff – if you can find the names of the client's staff, memorize them! It may be appropriate to assign particular subject areas that a team member should not only research but lead during questioning. Finally, designate roles to the team members participating in the interview; the roles include leader and scribe, described below, and possibly other interview activities.

At the beginning of an interview, it is always important to establish rapport. The obvious first step is to introduce yourself; when you do so, go beyond the military protocol of "name, rank, and serial number" to establish a connection with the interviewee. If you are in that person's office, look for clues to their interests in terms of photos or memorabilia. A connection by place is always good to establish.[5]

After the personal connection is made, another introduction is still in order – the topic of the meeting itself. The *leader* of the meeting should describe why you called the meeting

5. A sociologist interviewing residents of the capital city of Indiana once described the benefit he gained by pronouncing that city's name "In-din-ap-less," thereby establishing himself as a local boy.

and what facts or decisions you expect to gain from it. If there were previous meetings, recap what you perceive as the conclusions and decisions. Do this recap for two reasons – the first is to bring the other participants "up to speed" and the second is to confirm or correct your perceptions. Note that the meeting's leader, who is responsible for seeing that the agenda is covered, need not be the team leader.

During the heart of the meeting, when you are actually addressing the reasons the meeting was called, it pays to have one person who acts as *scribe* (or stenographer). There are always details that should not be missed, but if the person leading the meeting also is responsible for those details, then either things will be forgotten or the meeting will lose focus. If you are by yourself, bring a tape recorder if you can, of course asking permission before you use it.

At the end of the meeting, wrap up by summarizing what transpired, planning any future interactions, and thanking all participants. As soon as possible after the meeting, hold a "de-briefing", including both team members who attended and those who did not, if any. Again, there are both process and product (= information gathered) issues. Evaluate individual and collective performance: was a sufficiently detailed record kept, was the agenda adequately covered, were the clients happy with the meeting process and conclusions, *etc.* Evaluate the information: organize and summarize it, identify open questions, identify action items and assign them to team members, *etc.*

Remember that the best meetings have a two-way exchange of information. Not only must you elicit the best answers from the clients and listen carefully to those answers, but you must also keep your clients well-informed about what is happening.

B. Questions to Ask

The interrogatives that help organize a Zachman Frame (§ V.1.C) suggest questions with which to begin an interview:

- *Why* does your organization ask you to do what you do?
- *What* do you do in your job and how does that relate to the proposed project?
- *What* information do you provide, in particular information related to this project?
- *What* decisions do you make and what information do you use in making those decisions, particularly related to this project?
- *How* do you do your job? Both tools and operations should be covered.
- *When* do you do specific tasks and activities? "When" includes both time and circumstance.
- *Who* are the other people, inside and outside of the organization, that interact with you during your job?

While the above questions are good for initiating an interview, the following questions should be asked toward the end, as they help direct your further investigations:

- Are you answering for yourself or in an official capacity?
- Are my questions relevant to the problem that you have?
- Is there anything else that I should be asking you?
- Is there anyone else that I should be asking?

C. Observation

Observation is akin to interviewing in that it deals with individuals on the client's staff, but it emphasizes watching and listening to those individuals rather than talking with them. This is one occasion when you can verify that people really do what they say they do (see the discussion of normative *versus* actual behavior in §IV.2.B). It also provides a good evaluation of the workers' environment, leading to conclusions such as the need for a redundantly informative interface because the user has lots of interruptions and needs frequent reminders about the state of the system.

A step beyond passive observation is active participation in a user's job. If you can sit behind his or her desk for a while, you will get many valuable insights.

D. Documents

A large organization will have a number of documents, itemized below, which formalize their policies and procedures and explain their operations. A small organization will have some but not all of these.

- business plans – the current move toward "quality management" practices encourages organizations to set formal goals and metrics by which these goals are measured.
- training and operations manuals – organizations use these documents to help staff conduct business.
- organization charts – reading between the lines on these documents can tell about information flows in the organization.
- job descriptions – these are usually too general to provide operational details but they can be very helpful in tracking information flows and decision making responsibilities.
- forms – those documents currently used to collect information are extremely important in the analysis process.
- reports – the reports themselves tell how information is collected and summarized, knowing who uses these reports tells the implications of that information.
- current computer programs and databases – investigate not only programs which the new system will replace but those which are in any way related to the proposed system.

Remember that many of these documents are normative (§ IV.2.B) and should not be relied upon as the absolute truth about what people really do.

4. Coda

Remember that *our* goal is not not write software, not even to implement systems, but to help people solve information management problems.

Say this mantra daily and repeat it in job interviews.

Exercises

1. The section on Classes of Users, 3.2, suggests that you identify each individual internal user. At first glance, this seems to contradict the general tenor of this *Guide*, which emphasizes the need to be abstract and generic. Indeed, when comparable user information is discussed for the Requirements Specification, a more generic categorization by role is suggested. Why is it necessary to enumerate individuals for the Feasibility Study and roles for the Requirements Specification?

VI. Teams

Teams are an essential part of this course, as they are essential in real life. Your success in this course depends upon several other people and their success depends upon you. Team behavior is of course dependent upon the individuals in the team, but many team characteristics can be "engineered" too.

Team makeup in this class has many real-world aspects: the team is not of your choosing, the team is multi-cultural and multinational, the team members have varied aptitudes and motivations. However, as in the real world, teams are formed with the goal of maximizing the chance for for success.

Team organization is important. In past years, teams with good organization, whether "democratic" or "autocratic," have done well. Teams where one individual has shouldered all the tasks, teams where responsibilities have not been clearly established, and teams with poor internal communications have fared less well.

This chapter begins by describing a number of team roles, some of which are required for this course. These roles are then placed within team organization and governance. Meetings and tasks, the primary ways in which a team acts as a team, are discussed next, followed by a brief section placing your team in the context of the formal course supervision. The chapter ends with a few suggestions aimed toward making your team experience more rewarding. However, this *Guide* can only begin to help with team issues; seek out some of the many valuable sources on this subject. Von Mayrhauser [32, chapter 15] has a good discussion of small teams and personnel issues. Mynatt [23] has a good discussion of meetings.

1. Roles

The team endeavor has many continuing functions, or *roles*, which must be filled. The notion of role needs to be distinguished from that of task. The former indicates a continuing involvement, the latter a single activity. In a household, one family member may have the role of being the primary cook but each member will sometimes perform the task of chopping vegetables. A role is sometimes shared between several individuals.[1] Furthermore, one individual may have more than one role.[2]

Team roles may be formal or internal. Formal roles designate specific ongoing or recurring activities and are visible/applicable outside the team. It is essential that your team have each of these roles covered. The entire activity of team organization can be summarized as mapping formal roles to team members. The formal roles are:

1. A movie focusing on the life story of one person will have different actors playing that person as a child, as a youth, and as an adult.
2. The comedy films staring Peter Sellers were particularly humorous because Sellers appeared in so many guises.

- convener – The convener organizes team meetings, including setting the agenda. S/he is responsible also for other formal communication within the group.[3]
- assigner – Task assignment, in a way that balances responsibilities and reflects individual capabilities, is essential for project success. Task assignment requires complete knowledge of the project and full appreciation of the team members and their capabilities.
- monitor – Task assignment is an empty effort if there is not a corresponding process which monitors progress on and completion of the tasks. The monitoring role is commonly performed by the assigner but may be handled in some other way.
- client liaison – The relationship between you and your client can set the tone for how your team is treated during the various phases of the project. Therefore, your team's client contact should be a real "people person."

  Having several people independently talking with the client will cause considerable confusion. Therefore only one person should fill this role. This does not prohibit specific meetings focused on particular topics but pertains to major issues and commitments. Also, confusion is likely if the scheduling of meetings with the client is not in the hands of a single person.
- supervisor/instructor liaison – Just as in dealing with the client, it is useful to have one person coordinate communication with the supervisor and with the instructors.
- secretary – The team secretary, who maintains the log of meetings and assignments, is an important part of making team work into teamwork. This log allows the team, the Supervisors, and the Instructors to determine whether the process is working well and the product is on-track.
- librarian – With many people delivering pieces of the project, there are numerous opportunities to use the wrong piece. The librarian must insure that the pieces are where and what they are supposed to be. The librarian role is important during the Fall semester but essential during the Spring when the actual software is being developed and tested.
- guru – By the end of the Fall semester, you will probably be tired of repeated exhortations to learn your system platform. By the end of the Spring semester, you will wish you had been exhorted more strongly. Simply put, you must have someone who knows – really knows – your project platform. This is the guru's role. Beyond merely understanding the platform, the guru must also be able to train other team members in the platform and guide the platform-oriented aspects of the design.

The internal roles, so called because they are only visible within the team, tend to be less observable, transitory, and hence more difficult to describe. They often arise in the course of interviewing a client (as described in § V.3.A) or in running a meeting (§ VI.3.Abelow).

3. Although these are distinct activities, they are combined into one role in order to assure consistency in communications.

Sometimes roles are inherently in conflict. An obvious, often-reported example is the role conflict between a project manager, whose goal is to get the product out the door as quickly as possible, and a quality assurance manager, whose goal is not to release any product until it has passed exhaustive tests. It is essential to recognize that such inherent conflicts arise between roles and to prevent role conflicts from becoming personal conflicts. In spite of – perhaps because of – the opposition of their goals, the project manager and the quality assurance manager should be able to sit down with a cup of coffee or play a round of golf.

In addition to the roles within a team just discussed, there are also roles related to team supervision, These roles - supervisor, quality assurance monitor, and course instructors - are discussed in § VI.5, which follows.

2. Organization and Governance

Organization is the way team roles are mapped to individuals. Governance is the process by which decisions are made. These two topics come up together because many roles have a decision-making component. One of the first serious decisions facing the team is about its governance and organization. This is best handled by (1) understanding how organization derives from governance, (2) choosing a mode of governance, and (3) assigning roles to accomplish the organization.

The roles of convener, assigner, monitor, and the various liaisons are distinct but strongly coupled. They should be carefully coordinated. As an illustration of how teams have subdivided these roles, one common structure has the first three roles on the list assumed by a team leader. An alternative, used by some teams, has the leader doing task assigning and monitoring, but leaves all communication (convener and liaison roles) to a second person. One team found it was effective to have the convener manage calendars for the entire group ("directing people to the right place at the right time").

In a democratic team, assignment and monitoring are performed collectively; the convener role assumes more importance with such a structure. Do not be mislead into believing that a democratic organization is easier. A democratic team can be most effective, but only if *every* team member works hard at maintaining effective governance.

The roles of secretary, librarian, and guru do not have the same strong coupling as the others. They may be assigned more independently within any governance structure, which is fortunate since they (especially the guru role) do depend upon technical experience.

With respect to the details of team organization, what you do is not as important as that you do it well. Get to know your team members early, find out your strengths, and take advantage of them. Have an early meeting where each person identifies his or her skills, experience, and qualifications (this includes both technical and interpersonnel skills). Talk sincerely about everyone's objectives for the course – some are seeking only technical experience while others desire a leadership role. Be precise in your role assignments, so each person knows his or her task, but be flexible too. Role assignments may change but not to often, since each change diminishes team effectiveness and cohesion.

3. Meetings

One consequence of working in teams is the dependence on meetings. A meeting is a lot more than just a get-together at Nick's or an evening barbecue. If your team members invest in meetings, the team as a whole will realize valuable returns. Mynatt has a brief but worthwhile section on meetings [23, § 1.7.3], including a "meeting diagnosis checksheet" which you can use to improve the effectiveness of your team.

This section begins with a general discussion of successful meetings and roles in such meetings. Then, recognizing that meetings need not have the same format, we discuss one particular (and typically unfamiliar) form, the "brainstorming" session. Finally, a few specific suggestions are given.

A. Running a meeting

A successful meeting has a goal, an agenda, and a focus. Begin the meeting by stating the goal; many meetings are doomed from the outset by improperly set or inadequately communicated expectations. The agenda items are the objectives of the meeting; as such, they must be specific, achievable, and measurable. Starting the meeting by making sure the agenda items target the goal and meet the three criteria. This is not to say that the meeting should begin by setting its agenda. Quite the contrary – the agenda should be set and announced sufficiently in advance that all participants have time to prepare. And, those participants should prepare! Following the review of the agenda, it is usually valuable to review past meetings and current status. For Information Systems project teams, this is principally handled by monitoring the status of past task assignments.

Maintaining focus during the meeting is essential but not always easy. There are a number of tactics to help this. One is to designate the specific role of focus monitor. Another is to have some object which a participant can raise or a gesture that can be made[4] in order to alert the group that it has strayed off-target. While this latter suggestion seems "hokey," the alternative may be much wasted time and effort.

Roles during a meeting may be designated by a formal process - indeed designation of certain roles is highly recommended - but the are informal in the sense that they are transitory and not reported in the log. The roles of leader and scribe have already be described in §V.3.A, How to interview, which gives some suggestions on conducting a meeting (albeit a particular kind of meeting).

Another meeting role is that of facilitator. The facilitator(s) creates and maintains a situation in which everyone can freely participate. This involves both attention to feelings, drawing in those who are "put off" or uninvolved, and to ideas, asking participants to clarify as necessary. One specific suggestions: ask questions that require open-end answers, not just a "yes" or "no" or simple fact.

Certain meetings benefit from a "devil's advocate". Originally this term referred to a Roman Catholic official whose duty was to examine critically the evidence during the proceedings by which sainthood is determined. The meaning was thus extended to the

4. No, not that one. Pick something more polite.

role that critically examines information and champions the alternate cause, for the sake of further discussion and closer study. Thus for certain meetings it is valuable to assign the role of devil's advocate; however, this role may not be suitable for meeting in which clients or other outsiders participate and it is completely antithetical to brainstorming.

The goal of most meetings is not words but actions. Here is where the formal role of assigner comes into play. Whether the assigner role is filled by one person or collectively, the first step is to identify those who are able and are willing to take on a task. If no one steps forward to pick up a task, it still must be assigned. Here is the point where a democratic structure comes under greatest stress. All tasks must be assigned! The Team Log is mandated specifically to make sure this happens.

In addition to the procedural suggestions given above, be sure to remember that the meeting is made up of people. At the beginning, a bit of "small talk" is natural and beneficial. At the end of the meeting, remember to give each person their say. This may be a specific round where each person speaks. Listen for any hidden concerns and any surprising ideas.

B. Brainstorming

Brainstorming is more than just thinking hard – it is a deliberate attempt to quickly discover a broad range of alternatives and generate creative solutions. The key notion, which the entire group must completely appreciate, is that that brainstorming involves a period when the "censor" is turned off. Ideas should flow freely, no matter how whacky.

A brainstorming session has three phases. First there is a preparatory phase where the problem is presented. The more specific the problem addressed, the more productive the brainstorming. During the next phase, the brainstorming proper, members present solutions without examining or criticizing them. Effective brainstormers sometimes offer "off the wall" suggestions merely to break others out of constrained points of view. During brainstorming, suggestions are recorded – in a notebook or on an available blackboard – for later careful examination. After an appropriate time, perhaps determined by the clock and or perhaps determined when members run out of fresh ideas, the brainstorming phase ends with one final pass around the group to get any final ideas. The final phase is where the ideas are evaluated. Begin evaluation by clarifying and classifying the items (see VI.3.C below). For each item, discuss its pros and cons. You should now have some valuable, creative options.

Actual brainstorming sessions rarely fit the above scenario perfectly, but it serves to emphasize the the idea of a meeting with several distinctly structured phases. This approach, with a phase noncritically accepting all ideas, followed by a phase that examines those ideas for their merit, works especially well during design activities, where it is often iterated.

C. Meeting Techniques

i. Agenda and schedule

The main purpose of an agenda is the insure that all relevant topics are covered during

the meeting and, hopefully, each receives the attention that it deserves. An agenda may also help keep a meeting "on topic" in that participants may defer discussion less relevant to the current topic if they know that a topic that is more relevant is scheduled later on.

Besides merely listing topics, an agenda may schedule them as well, with time limits so that one topic doesn't "hog" all the discussion. An agenda may also explicitly organize the various phases of a meeting, such as brainstorming, break-out into small groups, *etc.*

ii. Organizing ideas

A large collection of ideas, such as might result from a number of client interviews or might arise from a brainstorming session, may be organized by the following technique: Find a classroom or other meeting space with a white board or other smooth surface. Write each idea on a Post-it note. Place the Post-it notes on the board, near other similar ideas, continuing moving notes as clusters of ideas seem to form. As clusters coalesce, give each cluster a title and replace it by a single note.

This technique typically works better with a vertical surface and Post-it notes than with a table and 3×5 cards. Standing up seems to encourage parallel interaction and full participation of all group members.

iii. Tradeoffs

The analysis and design process is characterized by a recurring evaluation of tradeoffs. It is rarely the case that there is a uniquely *right* solution (and a seriously-posed suggestion is rarely totally wrong). You will be weighing a large number of alternatives, each of which has some benefits and some disadvantages.

The first step that the team should take to handle tradeoffs is to clearly understand the various dimensions along which the alternatives differ. For example, alternative implementations may differ in usability, efficiency, maintainability, verifiability, and other factors. The dimensions often overlap – at least recognize when they do. Think back to your linear algebra class and remember the concept of "basis"; try to find a basis for the dimensions of your alternatives. Attach a weighting to each dimension – numerical values are of course best for this but may be hard to set, so you can use a scale something like "critical," "important," or "secondary."

After you know the alternatives and the dimensions by which they differ, make a table with the alternatives along one side and the dimensions the other. Fill in *all* the entries in the table. If you can't fill in some entry, then you don't know enough about that alternative (or the dimension is not well-defined) and you should seek to understand more. At this point, you must "invest in further information," which may require a bit of prototyping, a return to the client, or a trip to the library. It often happens that some of the alternatives will prune themselves as they are obviously infeasible.

Once the table has been filled (except, of course, for already discarded alternatives), look for and eliminate alternatives that rank low on critical or important dimensions. If all the alternatives have serious drawbacks, then of course you must either re-evaluate the dimensions to decide what is most important or return to finding new alternatives.

Otherwise, since you have discarded alternatives with serious drawbacks, you can simply choose among the rest by maximum utility. This tactic is different in two ways from computing a weighted score for each alternative and choosing the highest scoring. First, it is often easier, since arriving at precise weightings and scores is time-consuming. Second, by giving "veto power" to important dimensions, it avoids picking an alternative with one serious deficiency but high marks in other areas.

Note that the suggestion of completing the table is the real heart of the matter; evaluating alternatives with a table is rather natural but filling the table completely is a discipline.

iv. Consensus

It should go without saying that everyone must be willing to truly seek consensus, but sadly this point does need to be stressed. Everyone must understand – really understand and not just pay lip service to – the fact that they benefit more from the team's success than from the momentary thrill of "winning a point" during the meeting. Everyone must accept the strengths and abilities others bring to the project. Everyone must listen fully, with open mind as well as ears; it is not always the best programmer who comes up with a clever data structure or the best writer who phrases an idea most clearly. Everyone must, on the other hand, take responsibility for their statements, avoiding contention and distraction.

The primary precondition for consensus is that the issue be well understood. Ideas previously discussed about goals and objectives, meeting processes, and focus and facilitator roles apply here. Review the meanings of all significant words – recall that "misunderstanding" is also a synonym for lack of consensus. If necessary, use the specific brainstorming and tradeoff evaluation strategies discussed above.

It may happen that consensus cannot be reached. That's OK. But stop and understand why consensus was not achieved. At least get consensus on the cause of the lack of consensus. Hopefully this will point to further investigation that will clarify the matter. Or it may point to the need to ask for help from the Supervisor or the Instructors.

4. Tasks and Planning

The benefit of working as a team is that several individuals can be making progress on different objectives – that is, they are doing separate tasks in parallel. But inexperienced teams often get themselves in trouble by assigning tasks that are too large and not coordinating the execution of these tasks. Thus the team members arrive at the time when they had expected every task to be completed, only to discover that some tasks aren't done or some are nominally done but do not meet general expectations. The way to avoid this, of course, is to break down tasks into smaller pieces and to review the progress at the end of each piece. Even if a task cannot be broken down in this way, there should be regular meetings to review progress. For example, if a task takes three weeks, it should have two intermediate reviews at weekly meetings.

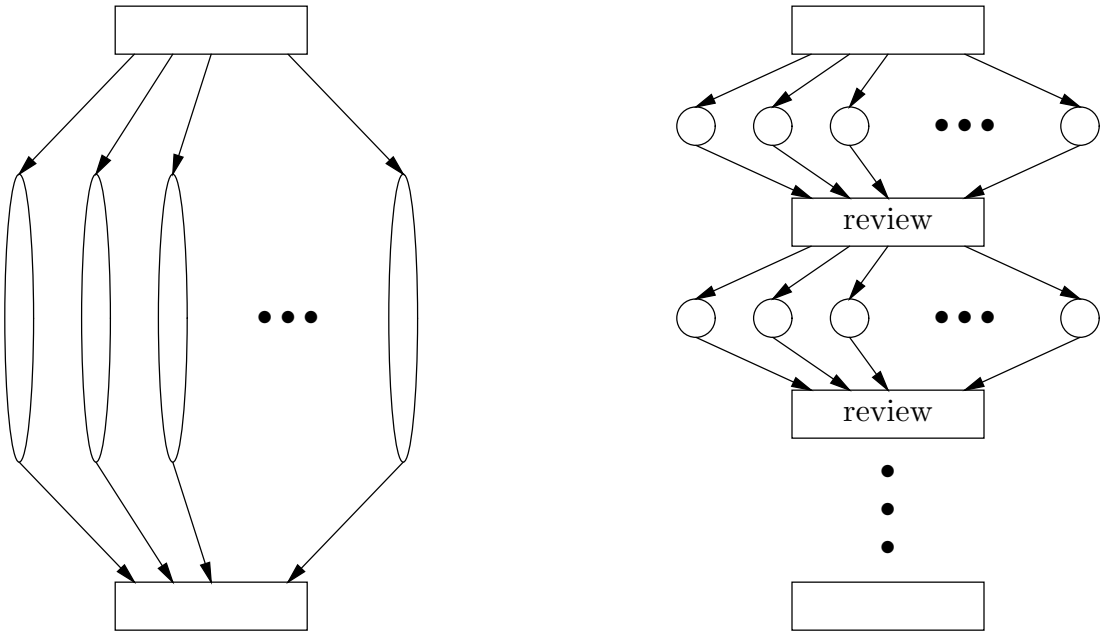


Figure VI.1: Bad (left) and Good (right) Modes for Working in Parallel

Figure VI.1, on the left side, shows a bad way to do parallel tasks, with no intermediate review, and a good way, with smaller tasks and regularly scheduled reviews.

Dividing and scheduling tasks, as in the right side of Figure VI.1, is the planning of the project. Planning includes: identifying tasks, subdividing tasks, estimating tasks, sequencing tasks and scheduling tasks. Note the difference between these last two: the sequence gives the order in which tasks must occur while the schedule gives the time during which those tasks should be accomplished. Because task A must be completed before task B begins does not mean that B must be started immediately after A is finished.

The further you go in the project, the more important the plan becomes. Therefore you should begin planning – and working to the plan – early in the project so that you will have experience when planning becomes vital.

On the other hand, do not get trapped by a plan. When you come across something that you can't handle, ask for help. If it does not seem possible to make a deadline, inform your team members as quickly as possible so that things may be adjusted. "I'm almost done" is a dangerous statement when it covers up a difficulty.

5. Supervision and Management

Throughout your professional career you will be subject to some form of management.[5] Similarly, your project team will be managed.

While other aspects of this project attempt to maintain the fiction that this is a professional endeavor and not an academic class, the supervisory aspect sometimes cannot ignore the academic nature of the project.

5. If you are an independent consultant, you will be self-managed and discover that is the severest form.

This section first discusses the various management roles and then explicitly reiterates (from the *Course Packet*[26]) certain things that your team must do related to project management.

A. The Supervisor

The team Supervisors have experience with the software process in the context of this course, industrial employment, or both. They have three distinct but overlapping functions:

- communication between the team and "upper management". The Team Log (see § VI.5.D below) is an essential channel for this communication. Each week the Supervisor will review the log with the team and then with the Instructors. In addition, questions concerning both product and process manners frequently require answers from the Instructors; the Supervisors are an excellent channel for getting these questions answered.
- advice and guidance on process matters. Typical concerns include team organization and governance. schedule planning, and inadequate task completion. Because the Supervisors and Instructors discuss team behavior on a weekly basis, there is a wealth of experience to help your team on these matters.
- advice and guidance on product matters. They may be of help in doing some magic trick with a form or may critique a draft of a document. In particular, Supervisors are resources for further information. They may not know the answer to your question, but they can provide assistance in obtaining the answer.

Remember the adage "Bad news is like a dead fish, it only gets worse as it ages." If problems arise, inform your Supervisor at the first opportunity.

B. The Quality Assurance Monitor

The Quality Assurance Monitor (QAM) is part of the management structure but does not have direct supervisory responsibility. While Supervisors are directly involved in defining and managing the process, QAM are concerned that the process is followed and the outcome meets specifications. The very title "Monitor" describes the activities associated with the role: namely "to watch, observe, or check esp. for a special purpose".[34]

The QAM is analogous to a business auditor, insuring that the principles of accounting are being followed and that the books balance. Just as an auditor is typically an "external auditor" or "independent auditor", QAM's (or their equivalent) are often independent of the team and sometimes external to the company. External QAM consultants are usually hired when there are legal (statutory or liability) requirements relating to software/systems quality.

A QAM's responsibilities "ramp up" over several project milestones. The first responsibility is a nominal review of the Quality Assurance Plan – only nominal monitoring is required because the QAP is mostly provided by a template – and the Project Plan, to insure that adequate time is alloted for testing. Next, the Test Plan is thoroughly reviewed

by the QAM, to insure that the plan is adequate for the project requirements. During the Testing phase itself, the QAM will monitor the testing, both by watching or participating in tests and by reviewing test outcomes. Finally, the QAM will participate in acceptance testing.

C. Course Instructors

Course Instructors are the "upper management." They are also most aware of and most responsible for the academic aspect of the project. The Instructors make final decisions about which projects to pursue, staffing, schedule deviations, team governance and discipline.

In these projects, as in the professional world, there is an occasional and unfortunate need to remove someone from a particular project or even "fire" them. The *Course Packet*[26] describes how "firing" occurs within a course context.

D. Mandates Concerning Reporting and Communication

While most of this content of this guide is intended to be broadly applicable to any information system or software project, there are a few specific course-related requirements that we repeat here.

The weekly Team Log is the primary formal mechanism by which the Supervisors and Instructors monitor team processes. The contents of this log, as mandated in the *Course Packet*, are:

> The team must maintain a Team Log for the life of the project. The Team Log must record all important team decisions, task assignments (such as 'Joe agreed to provide a debugged copy of module A by Feb 15'), task completions and other task status facts, and design or process changes. All this information must be recorded in a timely manner. The Log should also record role assignments, discussed below, and may also include notes from team meetings.
>
> The Team Log must be a "write-once" document. Originally the Team Log was a permanently bound volume (hence we still refer to the "Log Book"), but now it is more common to use electronic media. But even though the log is kept in a computer file, it is important that it retain its "write-once" nature. When the log was kept on paper, the "write-once" characteristic was enforced simply by keeping the log in ink, not allowing erasures. In a computer file, it can be enforced by using a file that is append-only. The reason for these constraints is that we wish to require that changes in the log be explicit. Just as a bank reflects an error in a deposit by adding a new journal entry rather than changing the original deposit, a correction or change in a log requires noting the incorrect statement and presenting the correct one.
>
> In order to insure the "write once" characteristic, each team is required to email their log entry to their Supervisor each week. It is required that the team have one formal meeting a week during which process matters are dealt with – in particular, tasks are assigned and task status is reviewed. These process matters are the heart of the log entry.
>
> You are asked to use a template for your log entry. This is easier for you because it indicates exactly what is required. This is easier for the Supervisors and Instructors because we

know exactly what to look for. Templates are available in the files `team-log-template.*` in the course's NFS repository. The team secretary should simply edit this file and submit it by email each week. In fact, it is a bit easier to start with a copy of last week's log entry, since much of it carries forward.

Most of the fields in the template are self-evident, but some deserve a few words of explanation:

- `Recording Date` The date on which the secretary edited the log entry.
- `Due Date` The date given in this packet on which the milestone is due (see the Project Schedule on the following page).
- `Group Target Date` The team should set their own internal due date a few days before the milestone is to be turned in. Record that date here.
- `Deliverables Due` A list of those items to be turned in.

In certain circumstances (*e.g.* when working on the Prototype and Preliminary Design), you may need to have two "Current Milestone" entries.

At each meeting when new task assignments are made, those assignments should be logged where indicated. The next week, these assignments should be move down to the next area, as carry-forward assignments. Tasks remain as carry-forward until after they have been marked "completed" in the status field. A completed task is then removed from the next log entry.

To reiterate, the purpose of the log is to record in a consistent manner and place: what tasks must be done, when they must be begun and finished, who is responsible for accomplishing them, and where they stand with respect to their completion.

In conjunction with the Team Log reporting mechanism, certain roles are also required:

- secretary
- liaisons
- librarian

6. Etiquette *et cetera*

If you've ever worked in a group situation before, you know that there are sometimes people who don't get along with everyone else and sometimes those who don't do their share. The team Supervisor and the monitoring mechanisms are there to help you with this situation. The course instructors are always available to help you sort out such problems.

It should be obvious, but you have to work with your teammates for the year. An investment in team compatibility will be more than adequately repaid. Start early with team lunches or pizza after the formal team meetings. Games or other group activities build team cohesion. A snappy title or logo helps form team identity.

Think of your teammates as your intellectual roommates. You don't like a roommate who leaves dirty socks around, so clean up when you edit files and test programs. You don't like a roommate who hides important messages under the sofa cushions, so be responsible with communications, reading and replying to email promptly. You don't like a roommate who leaves you only cold water for a shower, so don't act as if you are the only one on the

team. And most of all, you don't like a roommate whose idea of helping cook is to get a beer out of the refrigerator, so make sure you do your share.

While we are thinking of metaphors of living space, think about another space issue. If your team were in an industrial setting, team members would have office space in physical proximity with each other. This proximity facilitates technical and social communication between team members. Consequently, try to plan times common work times as well as common meeting times. Note that "working nearby" is not the same as "working together," the latter implies collective effort on a common task while the former is individual effort on separate (but possibly related) tasks. This advice applies to all milestones. If the milestone deliverable is a document, this facilitates common format and smooth reading. If the milestone is code, this helps the team guru teach other team members the subtile points of the tools.

Even though team interaction may seem to be mostly a cause of anxiety at this point in the project, there is an upside. When you get to know your teammates, you may find that you mesh well together and working in a team makes the project more interesting and exciting. Teams in the past have even designed logos and t-shirts in order to herald their team image.

Exercises

1. Section VI.1 mentions that certain roles have decision-making components. What project decisions are associated with the various roles? What decisions are associated with no role?

VII. Requirements Specification

This chapter begins with sections placing the Requirements Specification in the contexts of the client's needs and of the waterfall lifecycle. It then sketches the contents of the Requirements Specification and describes two methodologies important for this milestone: Data Flow and Entity-Relationship modeling. The remainder of the chapter describes the content in greater detail.

1. Purpose of Requirements Specification

The purpose of the requirements specifications is to analyze the client's needs for the project, to specify those needs in great detail, and to lay the conceptual foundation for the design of the project. At this stage, the project team must completely understand the client's needs and wants – unfortunately these are often not the same.

If the project team develops a complete and correct requirements specification, subsequent phases will take substantially less time and effort! If there are errors or omissions in the specifications, the team will need to correct the resulting defects during later stages, at many times the effort it would have taken to do the analysis right the first time. Furthermore, a good specification is the foundation for maintaining a quality product long after the initial development is complete.

During the analysis phase[1], the project team will need to visit the client several times, confirming what the team understands and clarifying details as that understanding gets deeper. The team, because of its expertise and perspective, is likely to perceive issues that the client does not. If the team waits for the client to independently perceive these issues, that will happen at a later stage when changes are difficult to make.

For example, a team doing a billing system for a law firm might initially sit down with the client and go over the data to be stored and the reports to be generated. When the project team meets later on, they will discuss how they understand the needs of the client. Perhaps the team members cannot agree on a consistent interpretation of the client's goals. The team then returns to the firm to clear up any inconsistencies or ambiguities that they might perceive. The project team, after meeting with the client a second time, realizes that there are several valuable reports which the client has not proposed.

A more difficult situation arises when the client perceives needs which are different from the real needs. This often occurs because clients tend to think in terms of normal behavior, not the exceptions. For example, the lawyer thinks of each defendant as the party who is responsible for paying the lawyer's fees, handling exceptional cases with marginal notes or just by memory. But the computerized database will need to retain separate defendant and billing identities, even though these are the same most of the time. Because clients tend to think in terms of normal behavior only, it is <u>essential</u> that teams consider exceptional behavior too.

1. Recall that analysis began during the Feasibility Study, so this phase is already underway.

A still more difficult situation is when the needs go beyond the boundaries of the proposed system. In our example, the team may discover that it is really the client's complete billing practices that need to be revamped. If your team runs into a problem of this sort, seek advice from your supervisor or the course instructors.

Continuing the example, the project team returns to the law firm for a third time to get approval (or disapproval) to include the reports as a part of the project and clarify any remaining issues with the client. Finally the project team writes up the Requirement Specifications and gives it to the client for sign off.

The team must have complete understanding of what the client wants. The questions what, how, where, when, who, and why should be asked at each stage when appropriate. The level of detail should include such items as: what fields need to be stored in the database, how many users will be using this at one time, what exactly should the reports look like, which existing forms the screen layout should mimic, whether the tab key should be used, whether the arrow keys should work in a certain way, whether the user can abort changes, *etc.*

As is evident, many small details need to be thought out before any kind of planning for programming is done.

2. Scope of the Requirements Specification

The purpose of this section is to place the Requirement Specifications (process and document) into the waterfall lifecycle, showing how this milestone interacts with those preceding and following.

Preceding the Requirements Specification is the Feasibility Study. The Feasibility Study gives a brief overview of the project and its feasibility given the capabilities and resources of the client. The Feasibility Study only outlines the clients needs for the project; the Requirements Specificationfills in the details of that outline. The Requirements Specification therefore builds on the Feasibility Study, so the material in the Feasibility Study carries forward, sometimes verbatim. It is of course necessary to catch misunderstandings and correct mistakes in the Feasibility Study, so work on the Requirements Specification should begin by discussing the Feasibility Study with the client.

Following the Requirements Specification is the Preliminary Design. The Preliminary Design takes the Requirements Specification and starts planning how the project will be implemented. Usually this means splitting the project into components, with an eye toward writing pseudo-code for code modules and equivalent designs for other components. Therefore the Requirements Specification must describe the project in sufficient detail that, during the design phase, the project may be broken up into modules without the need for extensive further analysis.

While the intention is to gather all the requirements in this phase, in reality this is not possible. Industrial colleagues who specialize in such matters estimate that 80 percent of the specifications can be captured during the initial requirements analysis. Thus

there will always be details left out of the Requirements Specification, requiring further analysis during the design phase. What is important is that the Requirements Specification omit nothing that affects the *structure* of the design. Finally, if the some aspect of the specification is simple and the subsequent design easy and obvious, don't spend time and effort on it.

3. Major Components of the Requirements Specification

The heart of the Requirements Specification is the itemization of *Functionality* and *Qualitative Requirements* (FR and QR respectively). FR are those specific operations, activities, and information contents that are to be supported by the new system. FR are always measurable, in the sense that it is always possible to determine whether or not a particular functionality requirement has been achieved. QR, on the other hand, are general properties that the system should have. QR are often difficult to specify precisely; reflecting this, the standard example of a qualitative requirement is "user friendly." The terms "functional" and "non-functional" are often used instead of "functionality" and "qualitative." However, as noted in § IV.3.D, Presentation of Ideas, this choice of vocabulary often puzzles clients and should be avoided.

Modeling is essential to the analysis process and a Requirements Specification document should include various specification models. Modeling is done to understand a problem and models are used to present that understanding, helping not only the project team but the client as well. Sommerville [29, introduction for Chapter 6] gives further motivation to the use of models and itemizes a variety of ways models are used in requirements analysis.

There may be a number of different models of the same problem, each from a different perspective, each focusing on some aspect of a problem. Indeed, every one of the eighteen cells of Zachman frame, introduced in Section V.1.C, has at least one model. Table VII.1 below gives some of the common specification-row models in a Zachman frame. In information systems projects of limited scope, as in this course, the commonly occurring models are Entity-Relationship and Data Flow; these are discussed later in this chapter. The other models in Table VII.1 are not just overlooked; instead, they are typically simple enough to be covered without full-fledged formal models. The relevant aspects process flow model are subsumed into the data flow model. The logistics network, the master schedule[2], and organization chart each require only a paragraph or two. Those business rules essential to the information system are captured in conjunction with the information model (although not part of that modeling formalism).

A model must not be restricted to the probable. If a model omits or limits something because it is unlikely, the resulting implementation will not be able to accommodate that rare circumstance when it occurs; a corollary of Murphy's law states that "the unlikely happens." On the other hand, it is always appropriate and sometimes valuable to note

2. A master schedule includes such things as long range business cycle specifications. For example, manufacturing of Christmas decorations must be completed by early summer in order to move them through distribution channels. The trees they will adorn cannot be cut so early.

	what *data*	**how** *function*	**where** *location*	**who** *people*	**when** *time*	**why** *motivation*
specify	information model (ER)	process flow model data flow model	logistics network	organization chart	master schedule	business plan business rules
design						
build						

Table VII.1: Specification Models in a Zachman Frame

that certain circumstances are "unlikely" or "rarely occur" in the textual discussion accompanying the model(s). A model should also be comprehensive, including aspects that cannot be implemented in the current project but are part of the larger context. Such future aspects must of course be clearly marked.

As noted earlier, there is considerable "carry forward" of material from the Feasibility Study to the Requirements Specification. Certain items, such as identification of the client, can probably be copied verbatim; other items, such as the description of interactions with other systems, should start with the Feasibility Study material and augment it.

4. Data Flow Methodology

Data flow models are essential for understanding and documenting the proposed system. Data flow models are based on simple concepts and use minimal notations, but they come in several flavors whose subtle variations can be confusing. Given their importance to the Information Systems course and the projects, Data flow modeling receives too little attention in lecture. However, you should study the material on your own. We will comment about further sources of information after a brief overview of data flow modeling.

Data flow models are represented by Data Flow Diagrams (DFD's); in fact they are so closely liked we use the abbreviation "DFD" for both. DFD's are just what they sound like: they show the flow of data through a system (or subsystem). DFD's (and other flow diagrams) came before computers, inspiring "flow charts" used to analyze and design programs. The flow is different, however. A "flow chart" shows flow of *control*, not data. Whereas many different data flows can occur in parallel in a data flow model, there is only one locus on control that flows around a control flow chart. Whereas the items flowing in a DFD can be identified and understood outside of that model, the singular control of a control flow chart is only meaningful to programmers and only in the context of control flow models. Of course a datum sometimes takes on aspects of control.

DFD's are constructed with very few components: sources and sinks where data enters and leaves the system, respectively, processes which operate upon the data, files (or databases) which store the data, and the flow of the data itself. A traditional notation for these components is shown in Figure VII.1. However, many different variations are used.

The most common variation from our notation uses a circle rather than a rounded box for processes. There is no fundamental reason for preferring on graphical notation over another - use what is convenient with the tools at hand. But use your notation consistently and provide a key for the reader (as with Figure VII.1).

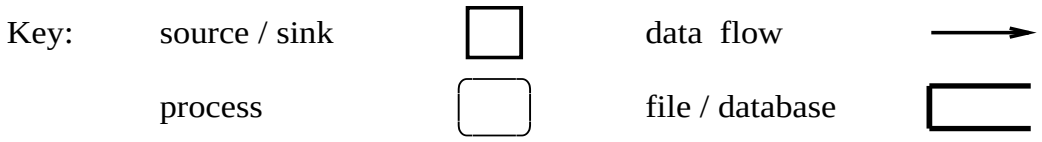


Figure VII.1: Data Flow Notation

There are several types of DFD's, varying along three dimensions:

current versus proposed: Because most information system development refines or replaces an existing system, system specifications should describe both the way things are now (the current system) and the proposed system that will replace it.

physical versus conceptual: Sometimes information is recorded on and closely associated with particular pieces of paper (or other media), as is the case with purchase orders or class rosters. Therefore one easy way to describe the information processing is to track the physical flow of these pieces of paper. When the information flow is described more abstractly, particularly when the the flow is between files and programs in a computer system, that is a conceptual data flow. A physical DFD also should capture the relevant aspects of process flow (row 1, column 2 of Table VII.1).

Although a conceptual DFD is usually a refinement of the corresponding physical DFD, this is not always the case. An example where the conceptual DFD is not a refinement of the physical DFD occurs when the physical DFD shows a piece of paper being handed along a chain of several individuals who do not process the information but merely carry the paper. The corresponding conceptual DFD would show this delivery process only as a single arc.

levels of detail: Because processes often have subprocesses and hence informations flows within them, the set of DFD's describing a system often spans several levels of detail.

Not all of the combinations of the above dimensions are used in describing the requirements for an actual system. Typically, a requirements specification should have (1) the current physical flow, (2) the proposed physical flow or a context diagram (explained below), and (3) several levels for the conceptual flow of the proposed system.

We now develop an example of data flow that occurs with bookkeeping in the Dewey Cheetham and Howe law firm. The current system is paper-based. Law firms track information about work performed for their clients, which is in turn used to generate bills sent to clients. When an attorney works on a case or talks to a client, she writes down the time she spends for that client on a time sheet. At the end of every day, this time sheet is given to the attorney's secretary who records on each client's log a running total and

other data. At the end of each month, the running totals are passed to the accountant who makes entries in the formal books and sends out a bill to the clients. The data flows from the attorney to the secretary, then to the accountant, then to the client, as shown in Figure VII.2. It is important to note that this process would be the same if everything was kept in a computer; with the exception that the data would not be physically flowing on pieces of paper, but just viewed and represented differently in the various stages.

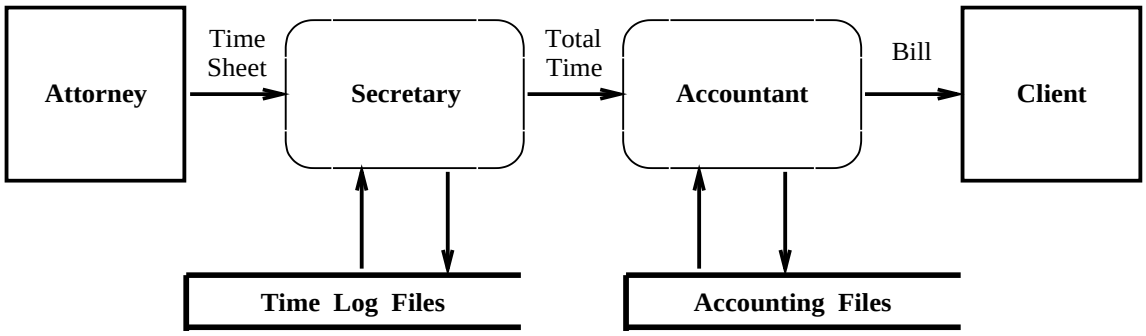


Figure VII.2: Current Physical Data Flow Diagram

The above description of the law firm use the word "client" in a very different way than the rest of this document. The problem is that our client (in the example, the law firm) may itself have clients. Such word use problems arise frequently enough merit attention: we may have an inventory system for computer systems, we may have relations between relations such as father and son. There is no easy solution other than paying attention; mentally attach a red flag to these words. However, in this particular case, there is an easy solution for the Requirements Specification: consistently use "Dewey Cheetham and Howe" or the abbreviation "DCH" for our client and reserve the word "client" for the clients of DCH.

Continuing the law firm example, we have a very trivial physical flow for the proposed system. As Figure VII.3 shows, the only flows are those into and out of the computer system. Because physical DFD's of proposed systems are often similarly trivial, they are commonly extended with a certain amount of conceptual information showing the major subsystems. This is called the context DFD. The physical DFD is often omitted if a context DFD is present. Because the proposed physical DFD of Figure VII.3 is so trivial, its only value is as a contrast to the current DFD of Figure VII.2 – the Account process of the current DFD is transformed into the Bookkeeper agent in the proposed DFD.

Warning: the analysis behind our law firm example is seriously deficient. The example is intended to illustrate DFD concepts and notations, not good analysis methodology. We will catch some of the oversights in section 6.4 below. But a serious requirements study would delve more deeply into internal issues, such as rate setting mechanisms, and explore beyond the purported boundaries of the system, investigating the articulation with the accounting and legal records functions.

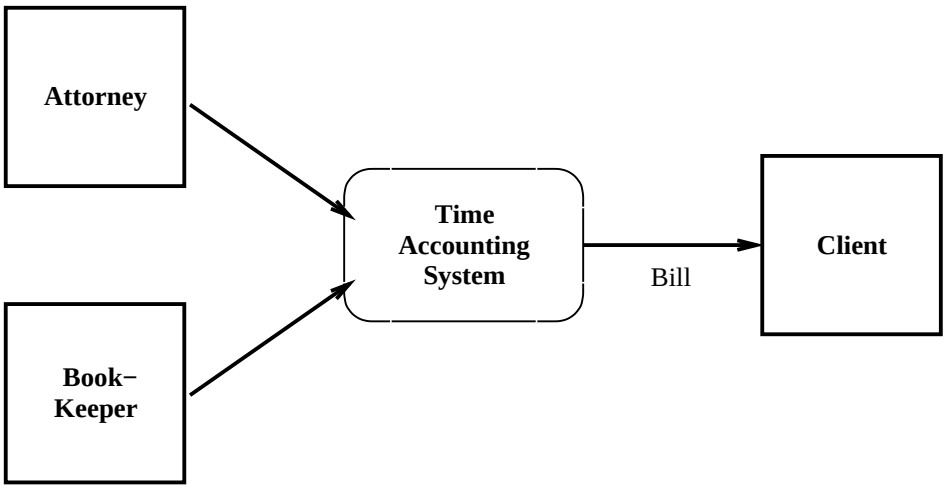


Figure VII.3: Physical Data Flow for Proposed System

The context DFD for our example, given in Figure VII.4, illustrates the relatively common situation where each user role is a source connected to a subprocess for that role, with few other interconnections besides a central database. Note, however, that Figures VII.3 and VII.4 distinguish user roles, not actual people. An independent lawyer could act as her own bookkeeper in accordance with these diagrams.

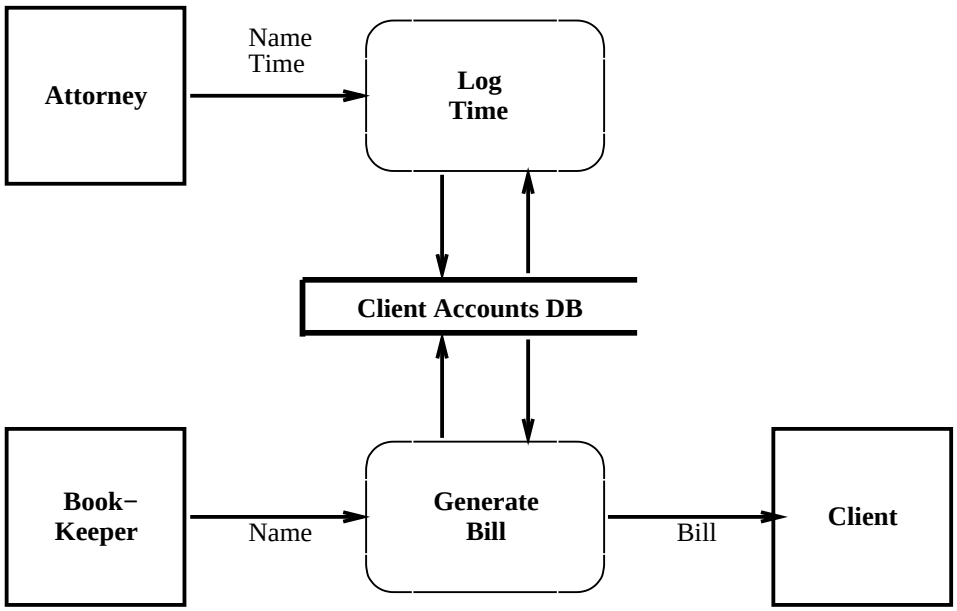


Figure VII.4: Context Data Flow Diagram

Figure VII.4 clearly distinguishes flow, such as Name and Time from an attorney into the time logging process,[3] from information sharing between two processes, such as the

3. Note how "time logging" distinguishes the activity performed by the "Log Time" process from the

Accounts Database accessed by both the logging and the billing processes. It is common to have two flows in opposite directions (graphically, parallel arrows with heads at different ends). It is possible to have a two way flow (a single line with arrow heads at each end), however a two way flow without a mediating database is suspect[4] and needs to be carefully validated. The concern with a two way flow is timing – it implies synchronization or coordination between the processes. Because a database is persistent, data stored there is available to any pertinent process at any time.

The step from the context DFD to the level 0 DFD is one of changing the focus from the system's relation to the outside world to the the internal structure of the system. At its simplest this is merely removing sources and sinks.

Mynatt [23, page 57] recommends a top level DFD as shown on the left of Figure VII.5, with a single input stream that flows into a process that classifies that input and directs it to the process that handles it. Nowadays this is almost certainly **wrong**. Her suggestion is correct only if the "classify & direct" activity is implemented by you. With a menu-and-form system, it is much more common that the classification is done directly by the user through the menu selection, as shown on the right of that same figure. Even command-line systems are likely to have different procedures/parameters through which the user evokes different operations.

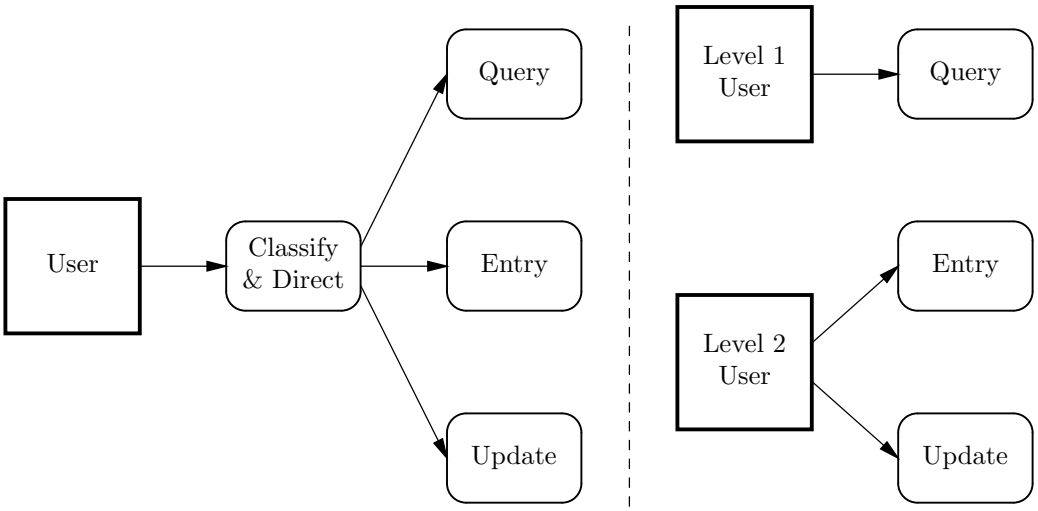


Figure VII.5: Variations to Input Classification

Data flow diagrams deal with more than just flow in and out. They deal with the processes that the data will undergo. A process is usually a transformation or validation of the data at certain points. The first process (1 in Figure VII.6) in our example starts when the attorney enters the client's name and the hours spent. The next process occurs

process itself. Try to keep these distinctions clear and explicit in this way.

4. Yet another aside on word use: "suspect" is used here rather than "erroneous" to indicate the strong possibility but not certainty of an error.

when the bookkeeper causes the system to generate a bill for a particular client, entering the client's name. The bill generation step involves two subprocesses, one for time records and one for financial matters (2 and 3 in Figure VII.6), and the flow between these processes.

Data flows have descriptive names indicating[5] what is being passed. The name is then listed in a data dictionary with a description of each item in the data flow. For example, Time_Sheet_Info would include beginning and ending hours spent, client name, and description of tasks performed for the client.

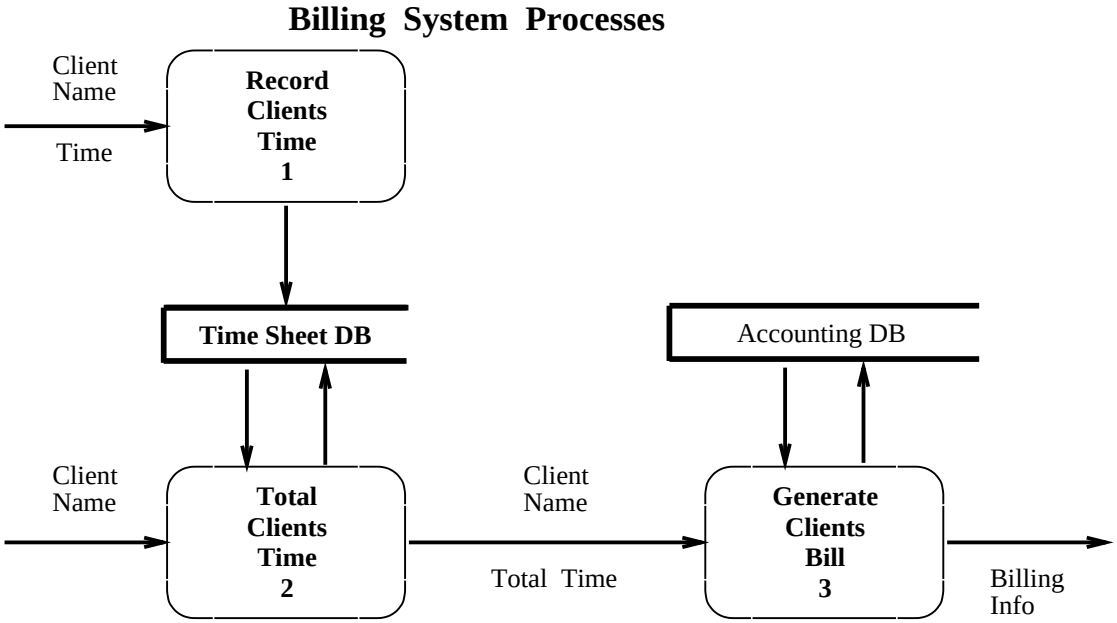


Figure VII.6: Level 0 Data Flow Diagram

Processes are different from sources and sinks in that processes have both in and out data flows (and corresponding arrows in the diagram) while sources and sinks typically have only flow out or in respectively. Sources and sinks can never be decomposed; that is, we can never look inside them while it is natural to look inside and thus decompose processes.

If processes are large, then they should be broken into subprocesses, following the same notation. When a process is decomposed in this manner, it gets its own digram representing its subprocesses and internal data flows. A dotted decimal notation is used to indicate the process hierarchy. For example, process 2 in the top level conceptual diagram (Figure VII.6) is shown broken into subprocesses 2.1 and 2.2 in a diagram at the next level (Figure VII.7). Subprocess 2.1 is further decomposed in a level 3 diagram, with subprocesses 2.1.1, 2.1.2, and 2.1.3[6] (Figure VII.8). Flow into and out of a process should appear in the detail diagrams indicating its subprocesses. For example, the flow from process 2 to process 3

5. An earlier draft of this *Guide* used "representing" rather than "indicating"; why was this word changed.
6. This is *but one* level 2 diagram, the only one that we show out of the many necessary.

in Figure VII.6, labeled "Total_Time", appears in the detail diagram for process 2, Figure VII.7. There the arrow labeled "Total_Time" appears as flow out of subprocess 2.2 without flowing into anything. This indicates that the flow exits the parent process. Similarly, the arrow labeled "Client_Name" in Figure VII.7 indicates a flow into process 2 because it does not come from a source or other process.

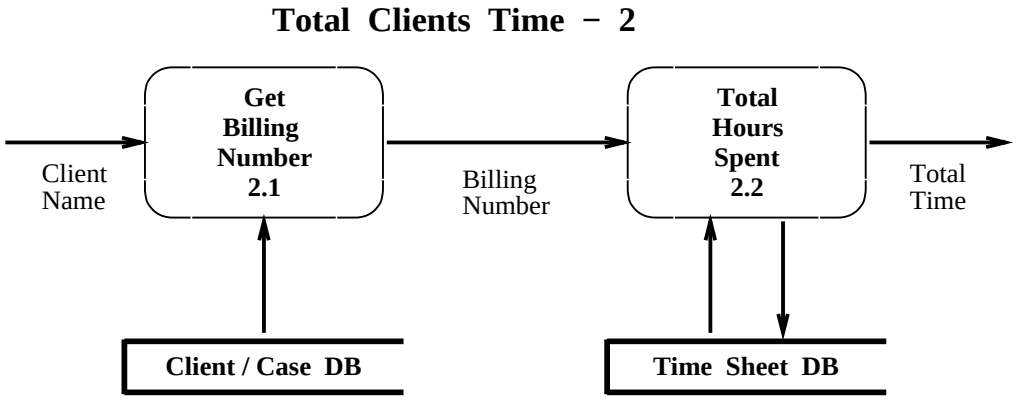


Figure VII.7: Level 1 Data Flow Diagram

Another part of the data flow diagram indicates the databases. In our convention, databases are shown as just open-ended boxes[7] without attributes or even relational structure. In the law firm example, the collection of time sheets from the attorney is a database, the collection of client's folder is a database, and the accountant's records are a database. In addition, note that the only flow out of process 1 in the level 0 diagram for our example (Figure VII.6) is to the Time Sheet database; that is, the (implicit) flow from process 1 to process 2 is through the database. This mediating role of databases is common and therefore databases should be inherited and shown where relevant in subprocesses (in contrast with sources and sinks, which are commonly omitted from subprocess diagrams).

Because our analysis is incomplete, there could be more databases in this system.

Textbooks covering DFD's present interesting variations to the methodology. Mynatt [23, §2.2] illustrates, with an example of a factory baking muffins, that data flow is only a special case of more general flows. This is relevant because sometimes the system context requires documenting flows of other material objects. Davis [14, Module D] gives a simple but clear description of DFD's. He makes valuable suggestions, such as generally drawing the flow right and down, in keeping with English reading style. The book derived from Andersen Consulting's training program[8] [16, pp. 227-240] begins with a somewhat larger example and introduces some variations in notation. In particular, the have a notation for marking objects which appear several times in DFD's (these are typically files) so the reader knows to look for the same object elsewhere. Pressman[25] gives a number of extensions to the basic DFD notation for data flow modeling (Chapter 7) and later shows how data

7. Other notation uses squares, but this risks confusing databases with data sources.
8. Andersen Consulting is now named Accenture.

flow models can be transformed into system designs (Chapter 11). Von Mayrhauser[32, Chapter 3] views data flow modeling as the heart of functionality specification. Figure VII.8 requires some intelligence on the readers part in the labels applied to certain flows, in that shortened names ("Name" instead of "Client Name") and abbreviations ("Billing No." instead of "Billing Number") are used to reduce clutter on the diagram.

One area not well-covered by traditional textbooks is the impact of DFDs on user interface design. The goal is to automate the flow between processing steps. For example, the level 2 diagram from the law firm, given in Figure VII.8, indicates that flow out of process 2.1.1 depends upon whether or not the client name was found in the database. The user should not need to direct this flow and in fact the user should have no control over screen flow (beyond the option of re-trying with a corrected name). This use of data flow to indicate screen flow is particularly relevant at lower levels of the DFD where the flow is more dependent on data values (and hence has begun to look a bit like traditional control flow).

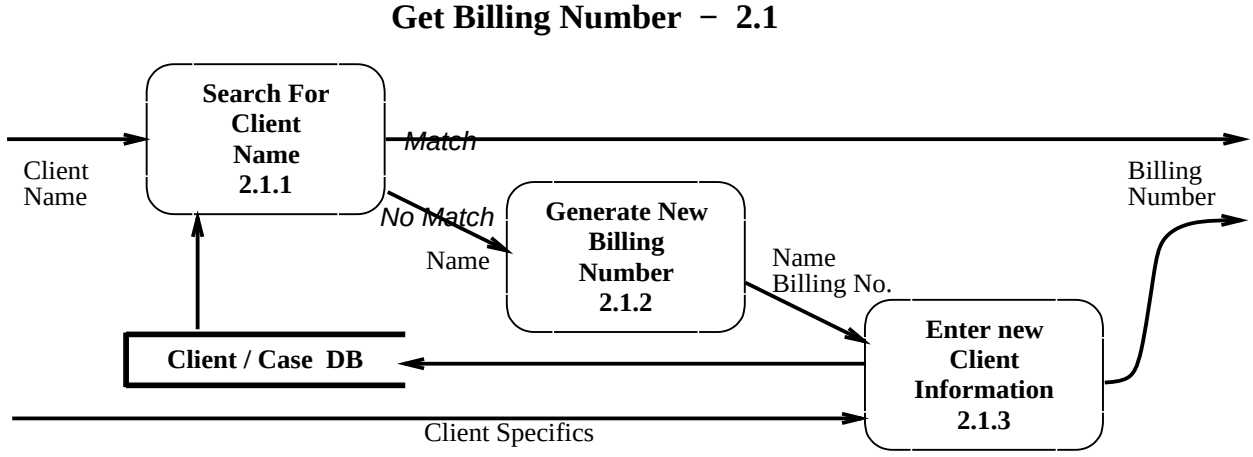


Figure VII.8: Level 2 Data Flow Diagram

In addition to flow of physical objects or information, there are often control flow *signals* – flow which activates or modifies the behavior of the receiving process. Such signals occur in Figure VII.8 as and . The names indicate that these are complementary signals, in that each search sends only one of these signals. There is no graphic convention to indicate this, only the choice of the names. There is a temptation to use decision boxes (often drawn as diamonds) to implement such branches; but this use indeed reflects implementation and the temptation should be avoided.

5. Information Modeling

The *information model* is the heart of any information system project and certainly of this class project. Indeed, the dominant requirement for for an information system is simply "maintain information as described in the attached information model." Thus this model is the most essential (and most scrutinized) part of the entire project!

It is essential to keep information modeling in its proper place in the project lifecycle – equivalently, in the proper cell of the Zachman framework's "What" column. The information model, expressed in the graphic Entity-Relationship (ER) notation, is an essential part of the analysis phase – Zachman's "specify" row. That model will be directly transformed, during the design phase, into relation schemata for the project database (and will indirectly be the key point for designing screens and data manipulations). During implementation, the table designs are translated into an SQL script that implements the database; with the right tool, this translation is largely automatic.

There is a tendency to rush into database design during the analysis phase; it must be resisted! This tendency is exacerbated by the fact that ER diagrams are often used to represent database designs rather than conceptual models, a fact that is evident when a diagram is called a "data model", indicating a focus on representation rather than content.[9]

This section first discusses the ER notation. It then discuss metadata and metadata modeling, the most serious stumbling block in information modeling and hence in database design. The metadata discussion is more extensive than most material in this *Guide* because there is no text that covers this material.

A. Entity-Relationship Notation

The ER notation is the globally accepted representation of information models. It occupies a rare "sweet spot" among among modeling methodologies, in that information models have precise, formalizable interpretations and yet can be communicated with project stakeholders. The communication aspect is vitally important. Throughout a project, especially during the analysis and specification stage, communication with clients and other stakeholders is the source of the most, and the most serious, errors. Hence the "communicational" aspect of information modeling will be a major focus of this *Guide* and this course; the correctness and appropriateness of the conceptual model being the other major focus.

This *Guide* will briefly introduce ER modeling; it will not explain that methodology in depth because this material is covered very well elsewhere. In particular, you should carefully read the lecture notes on information modeling. The book by Batini, Ceri, and Navathe[3] is an excellent source – it stresses conceptual and communicational aspects. Many database texts discuss ER notation, but often merely use this notation as a medium for table design. Using the information model to do table design is flat out wrong! It miss two opportunities: the first being the complete gathering of information needs and the second the full optimization of table design.

9. Indicators that an ER model is really just a database design include: the only relationships are binary ones, attribute lists include artificial or generated ID fields, multivalued attributes have been "normalized" to separate entities, entities exist that only contain reference values (such as `State` only connected to address fields), and entities manifesting subtypes. These same indicators, if they appear, warn that you have strayed from conceptual information modeling.

Do not be misled by the fact that we do not repeat here information on information modeling which is available elsewhere. The information model is essential to the Requirements Specification.

An ER model is composed of *Entities* and *Relationships*. An Entity abstracts a class of objects, which may be physical objects such a people or chairs or may be classificatory objects such as legal cases. Entities are connected to (or "participate in") Relationships. Both Entities and Relationships may have Attributes, names to which values may be attached. The diagrammatic notation for these three major pieces is shown in Figure 9; the connection from an Entity to a Relationship is diagrammed by simply drawing a line.

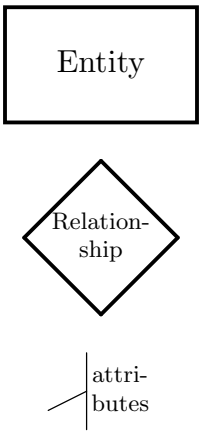


Figure 9: ER Notation

Following our own advice that an information model is a good way to specify information content, the ER diagram in Figure VII.10 expands upon some of the information that your team must include in its Requirements Specification. Figure VII.10 is therefore to be viewed from two perspectives, both as an example of the ER notation and as a description of information that you must gather during your analysis. In the following paragraphs, we distinguish the two perspectives with the terms "notation" and "content" respectively.

From the content perspective, Figure VII.10 shows how staff members in the client's organization (labeled "Person") ultimately relate to the contents of the information system (labeled "Information Component"). A person holds a position and through that position has one or more roles. From the notation side, the diagram may be interpreted so that one position has several roles; a more precise notation, which indicates when a relationship is singular or multiple, is covered the lecture notes. The singular/plural ambiguity also emphasizes that an entity represents a set of objects in the real world, not just one object. Returning to content, a role (that is, a person acting within that role) can perform an operation or apply an operation to some information sources or sinks. An operation also may act on one or more information components; note that the diagram shows that the information sources and sinks of the information are visible to the role-holder but the components are behind the scene.

The diagram also shows the way in which entity classes are generalized (grouped together) into larger abstractions using the "box in box" notation. The placement of the boxes labeled "Role", "Operation" and "Information Component" inside of the box labeled "Resource" indicates that entities corresponding to the first three boxes are generalized or abstracted into the fourth. That is, an object which is a Role entity, or an Operation entity, or an Information Component entity is also a Resource entity.

There are several subtypes not shown in Figure VII.10. First, Information Components are parts of an ER model themselves and hence could be an Entity, Relationship, or

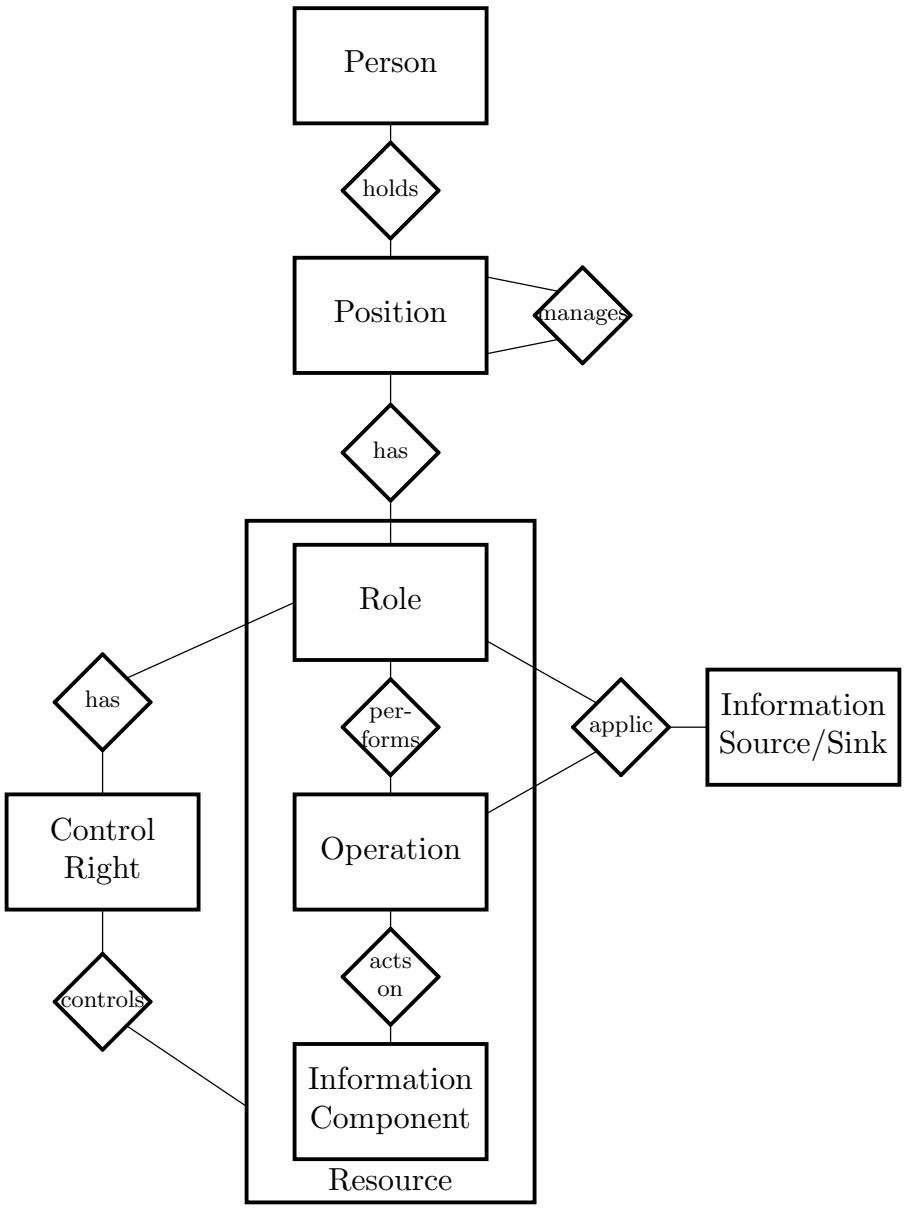


Figure VII.10: People and Information ER Model

Attribute.[10] Second, there are a variety of subtypes of Operation: Enter, Update, Correct, *etc.* The complete list of operations will also appear when Functionality Requirements are being discussed.

Figure VII.10 also illustrates a danger with modeling: we might be tempted to interpret some incidental aspect of the model as having significance in the real world. In our example, the fact that the relationships labeled "performs" and "acts on" are drawn within the

10. Thus we apply the computer scientist's technique of "reflection," developing an ER model that describes the general structure of ER models. When you get out in industry, do not try to explain this to the guy in the next cubicle with an MBA.

Resources entity is such an incidental aspect; those two relationships could equally well be drawn outside of the Resources entity.

In order to make our example complete, Figure VII.11 presents, without comment, the information model for the law office. If the example project were real, this information model would be shown to the stakeholders and example instances from the client's organization would be placed into the model. Working jointly with stakeholders to populate an information model with instances both helps the stakeholders understand the ER notation and helps the team correct and validate its model.

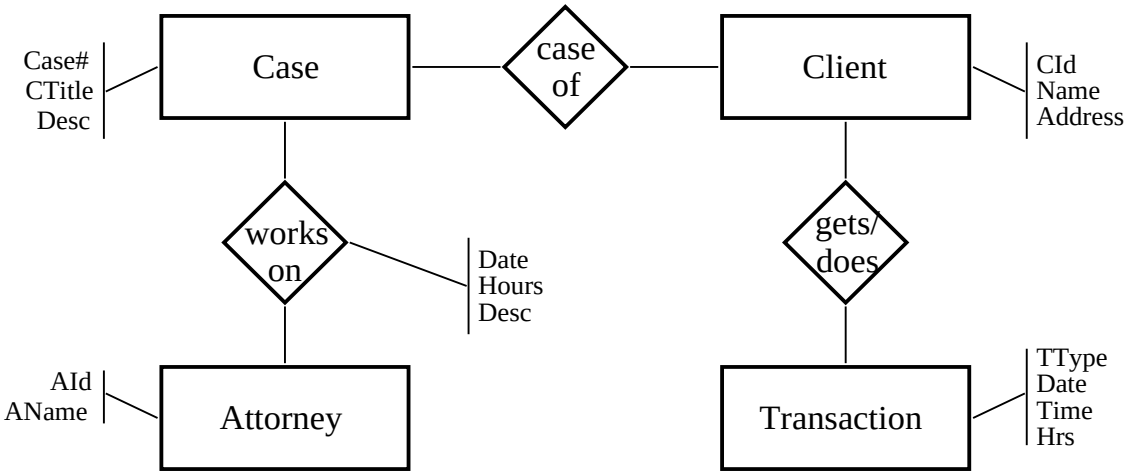


Figure VII.11: Entity-Relationship Model for Proposed System

There is a growing collection of discipline specific standard models (also known as "model patterns") and terminology standards (also known as "ontologies"). Such standards are good starting points for modeling and are valuable additional perspectives. By placing your models alongside others addressing similar problems, you will show your client the seriousness and relevance of this modeling. These standards can of course be found on the web; www.cs.utexas.edu is a good source for ontologies.

B. Metadata and Metadata Modeling

In a recent paper in the *Communications of the ACM*, Shankaranarayan and Even observe that "Metadata requirements are complex and difficult to capture, implementation is demanding, and the end result is rarely satisfactory ..."[28] Much of this complexity and difficulty can be ameliorated by understanding the nature of various kinds of metadata and by modeling metadata in a way which is easy to communicate yet natural to implement.

Broadly speaking, metadata is "data about data". The spectrum of meaning is indeed broad, with many uses of the term "metadata", differing in both obvious and subtle ways. Metadata exists at each of the three levels in the traditional ANSI/SPARC external/conceptual/internal (or "implementation") hierarchy[8], but has clearly distinct characteristics at the different levels. At the top, at the external level, metadata is just

data that has special meanings and uses. In the realm of document management and information retrieval, the documents are data and the documents' creators, publishers, *etc* are external metadata. Indeed, the most well known metadata activity, the Dublin Core Metadata Initiative[15], is specifying such external metadata.[11] At the bottom, at the internal level, metadata defines representation schema. In particular, relational table and attribute names are internal metadata. In between, at the conceptual level, metadata semantically characterizes other data, often as semantic rather than representational schema. The Dublin Core has fixed conceptual metadata, with "Creator", "Publisher", *etc.* itemizing the categories alluded to above. As one might expect, conceptual modeling should focus on conceptual metadata because conceptual metadata greatly influences internal metadata during development and external metadata during use. Figure VII.12 illustrates this situation.

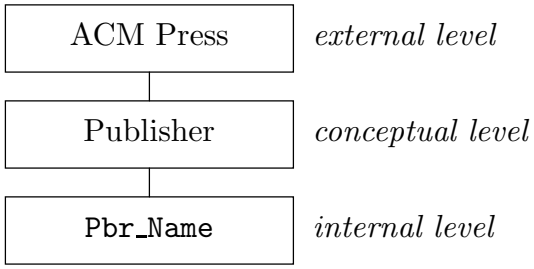


Figure VII.12: Metadata at various abstraction levels

The prefix "meta" indicates abstraction in some manner. In the case of relational databases, the abstraction is from data values to relation and attribute schema – a notion which is recognized as brittle when dealing with heterogeneous representations of semantically similar data (for example, [12; 21]). For the Dublin Core, the abstraction is to classifications of documents. While in any particular context, the meaning of "meta" is fixed and generally understood, overall, "meta" is relative. Evidence of this relativity is that the phrase "meta-meta-data" is perceived as meaningful.

In the days when most information systems were management information systems, metadata was not recognized as a critical issue because information requirements were fixed by accounting standards and established business practices. This allowed metadata to be fixed as relation schema, since the semantic and syntactic notions of "metadata" coincided. As inter-operation of information systems increased and as systems grew in other realms where metadata was wildly in flux, the significance of the metadata issue increased. In particular, scientific research necessarily generates new metadata (conceptual as well as external) as it progresses.

The rise of XML and the semantic web have given further prominence to metadata. Originally, XML metadata was understood structurally and encoded in special tags. But

11. Although topics such as the Dublin Core and the semantic web (discussed below) are tangential to this course project, the represent new trends in information systems. In particular, you should be able to respond if these come up during a job interview.

that understanding has grown, as the term "semantic web" indeed reflects. Thus the Resource Description Framework (RDF)[33] has been developed to represent and manipulate data about data. An essential characteristic of RDF is that is does not separate regular data and metadata, allowing the uniform representation and manipulation of metadata at all levels. The distinction between data and metadata appropriately is left to "the eye of the beholder." Consequently, perspicacious modeling of metadata is essential to wide-spread use of RDF.

To this end, we now introduce an extension to the ER notation to capture conceptual metadata (external metadata thus populates instances of such models). Adding any additional notation to the ER palate requires serious justification, because excessive notation can interfere with the essential communicational aspect of information models. However, this notation is entirely directed toward communication and analysis, based on the serious difficulties encountered when trying to understand metadata. The extensions provide no additional capabilities, in terms of what can be modeled.

The suggested extension to the ER notation will be given by an example. Unfortunately, the metadata needs of a small legal firm are not sufficiently rich to provide a challenging example. So we will switch from the legal for to a forensic laboratory.[12] The laboratory runs a variety of assays: toxicology of blood and tissue, ballistics, *etc.* Each type of test has a particular set of properties which apply to all assays runs of that type. These properties include inputs and readings, required steps, specific operating instructions, *etc.* Each run of a assay derives properties from its assay type, for example the need for calibration and a scale for those calibrations; but each run has its own unique calibration settings. Thus entity Assay_Type has meta-information about entity Assay_Run, as indicated in Figure VII.13, which shows the conventional way to model this situation.

Figure VII.13: Metadata, hidden in conventional notation

The suggested alternative notation is shown in Figure VII.14: simply replace the diamond and accompanying lines with a heavy double arrow, called the "meta-arrow" in the following. The head of the arrow points to the meta-entity, paralleling the heavy single arrow sometimes used for Is_A. The suggested vocalization for this figure is "Assay_Run obtains metadata from Assay_Type."

If the set of assay types were immutable, this situation could be modeled with conventional entity subtypes.[13] However, in reality the collection of assay types will change, so a fixed hierarchy would soon be out of date. A different extension to conventional notation

12. Keeping with the trend in TV programming, this is *CSI-InfoSys* or perhaps *Crossing the Jordan River.*
13. Observe how the informal use of "type" melds into the formal use.

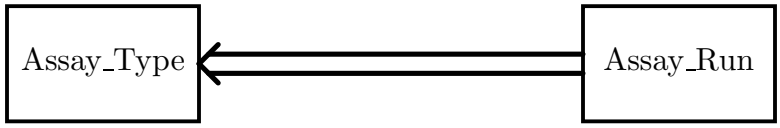


Figure VII.14: Metadata, explicit in proposed notation

would be to indicate open-ended collections of entity subtypes (for example, by simply adding "$\cdots$" to the vocabulary of ER subtype notations). But this does not cover cases where meta-information has a significant internal structure.

Such a situation arises with the addition of entities Instrument, Sample, Measurement and their corresponding meta-types to the running example. The scenario is that one Assay_Run assays a particular Sample, taking one or more Measurements with one or more Instruments. The meta side of this model is essentially a procedures manual for the laboratory, indicating which Assay_Type is appropriate for which Samp_Type, *etc.* One base entity which does not have a corresponding meta component is Technician; Technicians perform Assay_Runs. In the proposed notation, metastructure is expressed by enclosing the four base entities in a dotted box, enclosing the four corresponding meta-entities in another box, and drawing a meta-arrow from the first dotted box to the second. For unambiguous interpretation, it is essential that the dotted boxes be the same size and base entities and their corresponding meta entities be placed in the some location within the respective dotted boxes. Figure VII.15 illustrates this placement.

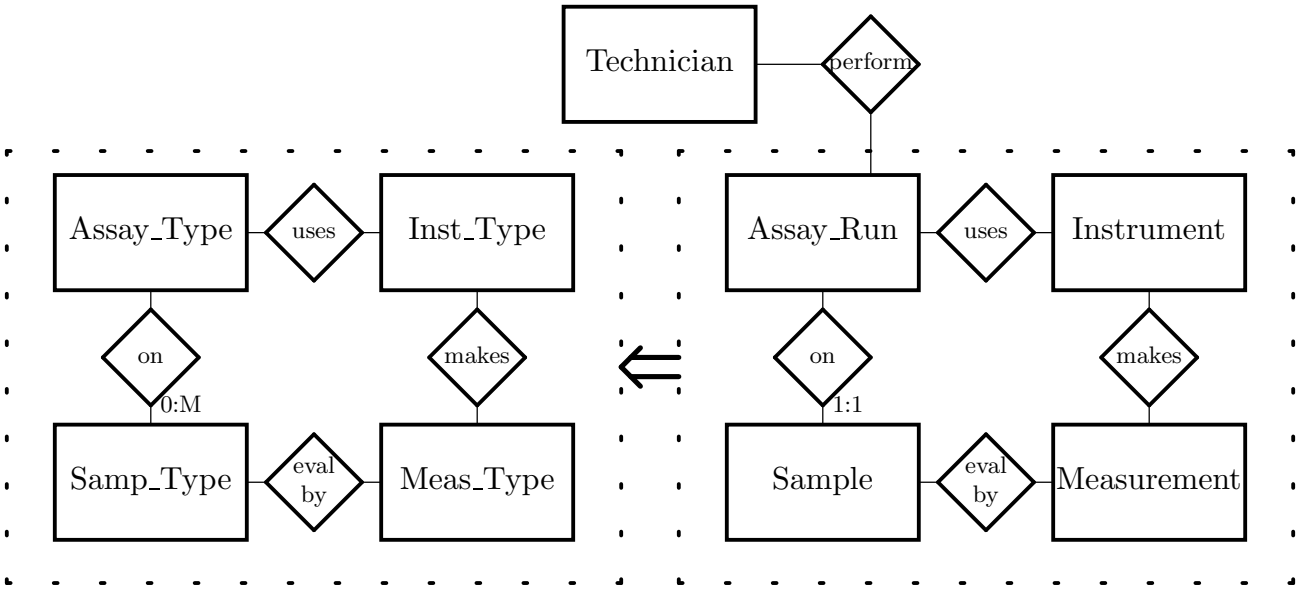


Figure VII.15: Metadata with Explicit Structural Constraints

The first benefit of this notation is visual. Replacing the meta-arrow abbreviation with conventional ER notation (similar to the rewriting of Figure VII.14 into Figure VII.13) would require four of_Type relationships, which would clutter the image, especially if the

eight entities are part of a larger model. Also, it is visually apparent that the "Type" information (*i.e.* the metadata) provides a template for a Run of a particular assay. Figure VII.15 also illustrates that distinct regions are required for the base component and the meta component because relationships and constraints – in this case participation constraints – differ for the base and meta cases.

It might make sense to show only one dashed box in the global figure, having a separate "drill down" figure that has two dashed boxes, as above, and indicates the distinct participations and attributes of the base and meta entities. Because standard ER notation is inherently flat (as opposed to, say, data flow diagrams, where the expansion of a process into its sub-processes allows a deep hierarchical structure), any opportunity to have structural decomposition should be seriously considered.

A further significant benefit accrues from interpreting the meta-arrow to constrain relationships. We would certainly expect that a particular Assay_Run instance uses only Instrument(s) of Inst_Type as specified by a uses relationship with the corresponding Assay_Type. For example, if a particular instance of Assay_Type requires a voltmeter, then the Instrument instance for run of that type should indeed be a voltmeter. Unfortunately, traditional ER notation has no way to express such constraints. However, the meta-arrow may be interpreted to explicitly require that instances of base entities be in exactly the relationships as their corresponding meta-instances, of course restricting this constraint to relationships within the dotted boxes.

6. Scenarios and Use Cases

A scenario is "a sequence of events, especially when imagined; an account or synopsis of a projected course of action or events."[14] Imagining various uses of the proposed system, that is constructing scenarios, is often a good aid while analyzing the system. It is even better at communicating the results of that analysis. This *Guide* will only mention the benefits and risks of scenarios, with only pointers to descriptions of how they are developed and used.

Scenarios, called "use cases", play a particularly important role in methodologies proposed around UML, the "Unified Modeling Language"[7]. Thus most of the literature is found under rubric of "use cases"[27; 25, §11.2.4].

Because they are typically just stories in plain text, scenarios are typically easier to create than other, more formal models. Because they are just stories, scenarios may lack precision or overlook important assumptions. Because some stories are more engaging than others, some areas are well covered while others are ignored. To avoid such problems, scenarios or user cases should only be used following a careful inventory of all users, all roles of those users, and all actions performed by those users. Thus a correct process that uses only scenarios is, in total, at least as difficult as any other methodology.

14. After an online copy of Webster's 7th dictionary.

7. A Typical Requirements Specification Document

The following is an outline for a *typical* Requirements Specification. The format and contents depend upon the project; your Requirements Specification document must first and foremost reflect your particular project. Most projects will have similar content, with a few idiosyncrasies which should be treated with special care. As with the Feasibility Study, we first given an outline of the outline.

0. Executive Summary
1. Introduction
2. Background of Client
3. Goals of the Project
4. Environment
   - 4.1. Users and Roles
   - 4.2. Hardware and Software Platforms
   - 4.3. Interaction with Other Information Systems
   - 4.4. Impact on Operations
5. Current System
   - 5.1. Current Entity-Relationship Model
   - 5.2. Current Data Flow Diagrams
   - 5.3. Other Supplementary Material
6. Proposed System
   - 6.1. Overview
   - 6.2. Proposed Entity-Relationship Model
   - 6.3. Data Flow Diagrams
   - 6.4. Interaction of ER and DF Models
   - 6.5. Functionality Requirements
   - 6.6. Maintenance Requirements
   - 6.7. Qualitative Requirements
   - 6.8. User Interface
   - 6.9. Implementation/Installation
   - 6.10. Other Considerations
   - 6.11. Limitations
7. Appendices
8. Client Comments

The remainder of this chapter discusses each section of the Requirements Specification document. This outline is less detailed – that is, it is less of a checklist – than the outline for the Feasibility Study. This is not because the document is expected to less detailed – quite the contrary. However, by this point in the lifecycle, things are getting specific to the project and it is hard to frame general rules that fit all projects. Furthermore, you are now expected to have some experience in framing details. It may be worthwhile, both as a vehicle for understanding the Requirements Specification and in preparation for further investigation, to rephrase the outline in greater detail, using the Feasibility Study as a model for the level of detail.

The Requirements Specification outline also introduces the law firm example. Note that in the law firm example "client" indicates those people served by the law firm and not the law firm itself (our client for the information systems project).

Requirements Specification 0: Executive Summary

The Executive Summary is a one or two page synopsis of the most important points of the document. Any recommendations made or conclusions draw in the document must be summarized. The Executive Summary should cover those most important matters that any reader should take away from the document. In fact, the Executive Summary is often distributed much more widely than the entire document - to people who need to know the gist but not the details.

Requirements Specification 1: Introduction

The Introduction typically introduces both the subject matter and the document itself, usually in that order. Some material may be borrowed from the Executive Summary. While the Executive Summary indeed summarizes points, the Introduction may only note the that points are covered. For example, the Introduction might say "Five recommendations are made in Section 6." while the Executive Summary would say "The following five steps are recommended: . . ." (ending with a phrase or sentence about each recommendation).

The Introduction to a document such as a Requirements Specification should also briefly overview how the analysis was conducted.

Requirements Specification 2: Background of Client

This section describes what the client does, both in a macrospective sense and in day-to-day affairs, and describes how the project will fit into the day-to-day workings of the client.

The project members can they tailor the project to the client's needs only if they understand how their project fits into the client's daily operations. It is especially important that the project team understands the client's objectives and operations, since the client is likely to ignore certain facets that are "common knowledge" to the client.

In the law firm example, a very brief description of what the firm does and how it is structured might be:

> The proposed project is for the accounting department of law firm of Dewey Cheetham and Howe. Dewey Cheetham and Howe is a professional corporation providing a variety of legal services to its clients from offices in Indianapolis, Bloomington, and South Bend.

A description of the problems faced by the client often helps. For example, the statement:

> The accounting department has recently been plagued by by a variety of complaints from the clients of Dewey Cheetham and Howe. The complaints commonly were that services were billed to the wrong clients and that clients were charged the wrong rate for the services.

Note that this provides a general view. Specifics for users and uses are given in the next sections.

Requirements Specification 3: Goals of the Project

This section will describe, succinctly but precisely, what utility arises from the project's implementation. Recall the specific meanings of 'goal" and "objective" as discussed in the section on Terminology (§ I.3). Particular objectives, such as adding items to a database or generating reports, are fairly easy to specify but should be covered later – in fact the functionality requirements are essentially the project's objectives. Broader, more abstract goals are more difficult to describe. Qualitative requirements, such as ease of use, simplicity, and logical flow, are sometimes closer to goals and may be summarized here. Goals (and

later objectives) are often best presented as bulleted lists. For the law firm, the primary goal is

- store and retrieve information concerning billing at Dewey Cheetham and Howe.

while the corresponding objective (discussed in § 6.5, Functionality Requirements) is

- support and maintain the information described in the Entity-Relationship model by providing the following operations on data: enter, update, correct, delete, access, summarize, use, control, backup/restore, and warehouse.

Requirements Specification 4: Environment

Requirements Specification 4.1: Users and Roles

The users of the system have already been introduced in section 2.3 of the Feasibility Study; copy that section as the starting point of this subsection. Then go into more depth, particularly in terms of the users' motivation, skill level, frequency of use, and other factors. Try to move from users as particular individuals to the roles through which the respective individuals interact with the system. Particularly in small organizations an individual may have multiple roles: for example, the sales person also has human resources responsibilities.[15] Different roles have different uses of information and information systems. In our example, the sales role requires access to customer and order information while the HR roles accesses personnel records. If these two roles are not distinguished, combining customer and personnel subsystems, it will be hard for the organization to hire a separate HR staff as it grows.

Discuss the users' working environment as well; the degree to which the users will be able to concentrate on the proposed system, as opposed to dealing with it in the midst of a flood of distractions, affects interface design.

In addition to describing the users individually, it is important to describe the collective mode of use. A single-user program will be used differently than a system that can be accessed by many users simultaneously. For example, records in a database need to be locked in a multiple user system. A system that is used by several users but in a one-at-a-time mode doesn't need concurrency control but it usually needs certain protection mechanisms (not necessarily for security but just to prevent unanticipated interference). For any system with multiple users, the way that system is managed is critically significant.

Requirements Specification 4.2: Hardware and Software Platforms

This section, out of the entire document, is the easiest to conceive. Therefore, only a few words of advice: describe current and proposed configurations, specify computers and networks in a concise tabular or outline format, describe the purpose of any unusual software. The development environment, including toolkits and utilities, is also covered here.

15. The term "human resources" (HR) includes hiring, job benefits, employee tracking, *etc*

Implications of the platforms will be important. For example, when using an application known to crash at unpredictable times under the client's version of Microsoft Windows, frequent checkpoints and backups are in order.

Requirements Specification 4.3: Interaction with Other Information Systems

Again the Feasibility Study is a good starting point but a poor stopping point. It will be very unusual for you to find all interactions during the feasibility investigation. Certainly lots of interface details must be added.

Note that this section focuses on information systems; networks, communication software, web tools, *etc.* should be covered in the previous section. Occasionally there are information-centered interactions with systems which are not primarily information systems; such interactions do belong in this section. For example, an information system might record usage and performance data on a computer network; in this case the network is both a platform and a peer for the information system.

Requirements Specification 4.4: Impact on Operations

The section attempts to anticipate the effects of the proposed system on its external environment, as opposed to the impact of the environment on the system. One factor which can always be anticipated is the time and effort to be spent by the client's staff to operate the new system.

Requirements Specification 5: Current System

If there is a current system,[16] this is the place to describe what it does and how it does it. The current system does not have to be a computer system. It could be a system of people passing paper and manually looking up information. It could be a computer-based system on an obsolete platform. Or it could merely one that needs to be enhanced.

This analysis phase describes the what and how for the procedures of the current system. If this is a manual system, then much time could be spent interviewing people, analyzing what they do and what decisions they make. If a computer-based system is in place, then documentation might be the best way to discover what is going on. If documentation is unavailable or inadequate, then analyzing the source code might be useful. If possible, use the current system and observe its functionality and user interface characteristics.

Briefly describe the sources of your knowledge about the existing system. In particular, were you able to obtain the original specification and design documents, to the extent that these ever existed. Of course you should describe what was in these documents. For example, could you obtain an original Entity-Relationship model or did you infer that model from the schema. Note that the following section covers the content of that model, emphasizing that you should at this point discussed how you learned about the current system rather than what you learned.

16. Recall from §I.3 that, in our context, "system" always means the information system.

Requirements Specification 5.1: Current Entity-Relationship Model

The Entity-Relationship (ER) model[17] is a cornerstone of the Requirements Specification. Most of the effort should go toward modeling for the new system, but modeling the old one is an excellent way to start. Section VII.5 above gives some information about ER modeling, but the class lecture notes are the primary source.

Typically, there was no information formal model prepared for the current system and the "back-of-an-envelope" informal models disappeared long ago. In this case, modeling involves hypothesizing the abstraction behind the implementation. This is difficult because these systems usually grew up over time without any abstraction or data design.

With a computer-based system, files or tables usually turn into the entities. Relationships may be more difficult to discover, requiring analysis of the code and workings of the programs.

With a manual system, discovering entities and relationships may be a little harder. Usually entities are persistent data, such as a telephone log or a supplier list. However entities can come from more transitory things, such as appointment schedules. It may also be appropriate to discuss the representation of this information, especially in cases where all or part of that existing information must be carried forward.

Requirements Specification 5.2: Current Data Flow Diagrams

Modeling the current flows of the client organization – both data and relevant process (or "physical") flows – is an excellent way to understand that organization's business practices. In particular, all relevant paper documents and paperwork should appear in this model.

Requirements Specification 5.3: Other Supplementary Material

Documentation on file structures, forms, and code sources from which the current ER model was derived should be presented at this point or given an an appendix. In fact, this section is usually tied to an appendix containing important information that is already on paper, often in graphical format. This section should include a summary of that appendix.

File structure information includes field types and lengths as well as primary and secondary keys. File structure is often recorded using a chart with a separate page for each file or table (essentially the same presentation will be used in subsequent milestone documents when the new file structures are designed). Each field appears on the chart giving the field type (integer, character, boolean, *etc.*) and size where relevant (14 characters, 9 digits, *etc.*). Fields which are the primary key(s) should appear on top and possibly be highlighted. Secondary keys should also be noted.

17. After the polemic about terminology in § VII.5, you may wonder why "ER model" is used rather than "information model." The answer is that the Requirements Specification is to be read by the client and the specificity of the representation-based name reduces confusion; the client will immediately recognize "the pictures with the boxes and diamonds."

Include copies of forms if there is an existing system. These forms could be reports printed out by a computer system or forms that people write on and pass along.

Requirements Specification 6: Proposed System

The bulk of the work should really be spent analyzing the proposed system, not describing the current one. Working as a team is especially important for this phase because multiple perceptions are essential. Teamwork will help bring out questions and comments that one person alone usually will not think about. Also the entire team must collectively be convinced of the ideas and concepts of how the project should proceed.

The proposed system should be described in several ways to clear any ambiguities. First in English, then in several model formats like DFD's and ER diagrams. If the need should arise, use other diagrammatic presentation such as decision charts.

Requirements Specification 6.1: Overview

This section should broadly describe what the system will be doing and how it will be doing it. It is a careful reprise of the goals and objectives, with emphasis on operation aspects.

Overall this is a comprehensive section which should take several pages; but much of the material can (and should) be carried forward from the Feasibility Study, after careful review and appropriate adjustments of course.

Requirements Specification 6.2: Proposed Entity-Relationship Model

The ER model should be given here. It will probably take several pages, including (1) a simplified diagram for general view including client communications, (2) a global view with relationship constraints, and (3) detailed views of each entity. The ER model reappears later in the Requirements Specification when the Functionality Requirements are described.

Requirements Specification 6.3: Data Flow Diagrams

Whereas the DF model describing the current system typically concentrates on the physical flow, the model for the new system says more about the conceptual flow. As seen in the example in Section VII.4, the transition from current to proposed physical flow often makes very clear what is to be computerized. It also explicitly identifies other systems with which the new system will interact.

The question "How many levels should a DFD have?" is often asked. The answer, of course, is that it depends on the complexity of the problem. There should be as many levels in the DFD as needed to remove any ambiguity, but no more; the advice at the end of Section VII.2 applies here!

Requirements Specification 6.4: Interaction of ER and DF Models

The ER and DF models are integrated and mutually reinforcing. The team should carefully cross-check the two models for consistency and completeness. For each entity or relationship in the ER model, ask whether the DF model has processes that enter, update, query, and otherwise maintain the information associated with that entity or relationship. For each process in the DFD, ask whether the ER model has entities that contain the information manipulated by that process and ask whether any new relationships are implied by the process. For each output (report, screen, *etc.*), ask whether the ER model represents sufficient information to produce that output.

For example, examination of the ER model for the law firm immediately shows deficiencies in the DFD. There are two entities, Attorney and Case, with no supporting processes in the DFDs. This indicates the necessity of additional level 2 analysis comparable to Figure VII.8. This addition then percolates throughout the rest of the DF model. Consequences might include in a new role, "Clerical", in the physical and context DFDs.

The cross-check between ER and DFD models is not necessarily a section in the final Requirements Specification document; rather, it is a process necessary in the preparation of that document.

Note that it is not expected that the ERM model processes or the DFD model information. Instead, each should support the other.

Requirements Specification 6.5: Functionality Requirements

The Functionality Requirements are the heart of the Requirements Specification. The functionality requirements should itemize, comprehensively and in great detail, how the proposed system will meet the client needs – information, operations, processes, *etc.* Every item should correspond to some operational aspect of the system, so that at the end of implementation you can return to the functionality requirements and say "Yes, our system allows the user to do that." or "Yes, our system supports that." Note that there are two other categories of requirements covered in subsequent subsections: Maintenance Requirements involve operations intended to support the running of the system and Qualitative Requirements specify non-measurable characteristics, such as 'easy to use'.

The first functionality requirement for an information systems project – first both in the Requirements Specification document and in importance to the project – is the requirement that the operations of the system support the information in the ER model. This might be specified as:

i. record, in relational databases, information described in the Entity-Relationship model.

ii. support and maintain the information described in the Entity-Relationship model by providing the following operations on data: enter, update, correct, delete, access, summarize, use, control, backup/restore, and warehouse.

Of the operations in item *ii*, only two require explanations: "use" indicates that information may be absorbed into other applications and "control" is over the permissions for accessing and manipulating the information. Every operation in *ii* should be associated with user roles, not positions or individuals, in accordance with Figure VII.10. Also in accordance with Figure VII.10, operations should be linked to the information items that they manipulate.

As always, this material is often best presented with itemized lists or tables. We illustrate a table within a list with an example from the law firm system Requirements Specification which specifies required reports:

xii. Produce the reports specified in the following table:

report title	*data source(s)*	*select cond.*	*group & sort cond.*	*other*
productivity	time log ($\bowtie$ all)	all	by attorney (AName)	SUM(Transaction.Hrs)
major clients	Client $\bowtie$ Transaction	SUM(HRS) $>$ 50	by Name by TType	
		etc		

Expect the Functionality Requirements section to be many pages long. This section should clear up any discrepancies between the team and the client, remembering both that the clients wishes are primary and that you may understand certain implications that the client does not. This is the last chance that the client has to substantially[18] change the objectives or behavior of the system. The more detail you write, the more you have thought about the topic, and the fewer questions and ambiguities are left open. Remember, the Preliminary Design is next!

Requirements Specification 6.6: Maintenance Requirements

Whereas Functionality Requirements specify those operational aspects of the system that satisfy the client's external needs, the Maintenance Requirements deal with aspects that arise because of the presence of the system. In our law firm example, the need to track hours and payments exists independent of the system and thus requirements directed toward meeting this need provide functionality. However, doing backups or updating access permissions are operations that only exist because the system exists and thus requirements mandating these operations are for maintenance.

Beyond this distinction, Functionality Requirements and Maintenance Requirements are treated very much the same. In fact, they are often included in one single list.

The major reason for separating these two requirement categories is that maintenance operations are commonly provided by the toolkits these days. This the client still must

18. Minor changes, of course, will occur up until the very end. But these should be restricted to things like field placement in a report or button color on a screen. To repeat, later changes in specification should not affect the structure of the design.

do backups and monitor security but using operations provided by the software vendors.[19] In such cases, job is therefore to select a platform that does support the maintenance operations.

When dealing with these maintenance matters (especially the "hot button" issue of security, it is valuable to convey to the client the important distinction between mechanism and policy. For example, the classification of various users in six security levels is a matter of policy while the ability to support multiple levels with difference access rights is depends upon mechanism. Whenever possible, the software should implement the mechanism and the client implement the policy through the use of the mechanism.

Requirements Specification 6.7: Qualitative Requirements

Qualitative Requirements are requirements which do not directly correspond to operations or actions. There are often are not easy to check or to describe. Common examples are 'will be easy to use', 'will be understandable for a first time user', 'will have a logical layout and menu', *etc.*

Each QR should be described in great detail to make sure that the client fully agrees with the need that is stated.

Identifying QR has implications for project staffing and planning. Since QR typically cross-cut the entire project, each such requirement should have a person (or group) specifically identified to ensure that requirement is given proper attention during design and implementation. Security, and important QR these days, provides an example: because effective security vitally depends upon integrity of design and uniformity of implementation, it requires a different perspective that one attached to any particular FR. Thus it helps to cluster QR so that like clusters may be assigned to those responsible.

Requirements Specification 6.8: User Interface

This section of the Requirements Specification should emphasize characteristics, not context, of the user interface.

The user interface is of course the screens and menus through which the user interacts with the system. This is the section to describe in great detail what the interface will look like and how it will act. The first and best way to describe it is with a paper example of the layout. This way there are no ambiguities about fields and menu items – both their logical content and their placement and presentation. Corresponding to each screen should be an explanation of what keystrokes invoke which actions.[20] For example, with a menu, is it necessary to press return after pressing the menu selection? Or, when there are several fields to be entered, how does a user move from field to field – by tab, by return, or by arrow key? Keyboard shortcuts, commonly called "hot keys", should be explained. This

19. Thus such matters as backup and security are Functionality Requirements for the developers of the database or other software platforms.

20. If a standard set of conventions always applies (as should be the case with things like help), the full details should not be repeated for each screen.

includes both standard bindings, like F1 for the help button, and general conventions, such as using the first letter of a menu title as a hot key.

To repeat, it is not necessary, or even wise, to attempt to design the complete layout of the user interface at this point. The specifications are supposed to be for the general characteristics and features of the interface. However, you should define classes of transactions (screens) which will do similar or related operations and therefore should have similar appearance. While not part of the Requirements Specification document proper, it is time to for the team "guru" to begin investigating how to achieve desired user interface characteristics with the project tools.

It is important that the client sees this section and checks it carefully, since it defines the part with which the client's staff will deal on a daily basis.

Requirements Specification 6.9: Implementation/Installation

There is a wide variety of issues related to the transition to the new system. Some of the more common issues are:

- data conversion: Transferring data from the old to the new system is common, entailing the transformation from the old representation to the new one. These transformations are performed directly by the team (typically using a mixture of SQL and scripting languages) and are thus free of robustness constraints that make implementation of other modules difficult. The difficulty with data conversion is almost always "dirty data." Dirty data may include syntactically invalid values, such as letters in a phone number field, or incorrect values, such as an account number for a non-existent account – note that insertion either of these values would fail in a database platform properly configured with field format and referential integrity constraints.

  It is not the team's responsibility to clean the data; only the client should do that. However, the team can and should report to the client what data values appear erroneous. Therefore it is important that the Requirements Specification explicitly lay out the data conversion tasks for the team and for the client, along with a general timetable when these tasks should be completed.[21]

- training: The specification here should itemize what training the team will provide and when the client will make staff available for that training. If the law firm does a substantial amount of tex preparation, for example, the team should not expect to do training immediately before the April 15 filing deadline.

  Note that different roles may require different training – for example system administrators and end users.

- concurrent operation: Concurrent operation means that the old and new systems are run in parallel for a specified period, combining aspects of system validation and transition. This is a common requirement in large commercially installed systems. In this

21. The concern here is to make certain that the client has completed the data cleansing well in advance of the transition date.

course, however, you should consult with the instructors before agreeing to such a requirement.

Requirements Specification 6.10: Other Considerations

This optional section can explain facts that do not fit in elsewhere, as might be of interest and relevance to a new project. For example, a system might have special concerns for confidentiality.

Requirements Specification 6.11: Limitations

Sometimes it is as important to indicate what a system will not do as what it will do. In commercial practice, this serves to protect software contractors from continuing requests by clients who expect the system to solve their every problem.

A important limitation is when the identified needs are too large to be met within the current project. In such a case, this section should clearly indicate those aspects of the functionality requirements that can be accommodated within this project and those aspects that must be postponed until later phases, perhaps even scheduling several phases.

Requirements Specification 7: Appendices

This section is for forms or diagrams which don't neatly fit into the body of the document. If they client is currently using a computerized information system, then any copies of forms, descriptions of file structures, manuals, *etc.* should be placed here.

Requirements Specification 8: Client Comments

This section is a way to force the client to seriously consider the needs and solutions you propose. It should help the client evaluate the system.

The effort you put into this section of course depends upon your client. Client staff who understand both the business operations of their organization and how computer systems will impact their operations require little effort on your part. Client staff who do not understand the breadth and depth of their operations and who are computer novices will need a great deal of guidance.

8. Coda – Specification, not Implementation

In order to develop a good information model, you should understand and keep focused on the distinction between the modeling you are doing at this phase and the design and implementation you will be doing later. In most cases, the entities turn into tables of the database and attributes will turn into fields. But that does not mean that attributes are the same as fields or that entities are only records. Developing a database design rather than a more abstract information model at this point has two dangers: it limits the ways of thinking about the information necessary for the system and it may yield a less efficient implementation by locking in certain implementation decisions.

VIII. Prototypes and Platforms

1. Purpose of Prototyping

Prototyping serves two purposes – exploration of user needs and exploration of implementation characteristics. While these two distinct goals are commonly pursued within one prototyping effort, it is essential to clearly distinguish them. One goal is to determine whether an approach is desirable; the other, whether it is feasible. A particular project may concentrate on one of these goals: programmers, working in a familiar environment, prototype an information systems project to understand the client's needs and wishes; aeronautical engineers prototype an aircraft to investigate the characteristics of new airframe material. In this Information Systems class the prototype has *both* goals but the second (implementation characteristics) takes on special importance and character.

Industrial information system prototypes are built in the recognition that clients have great difficulty comprehending the abstractions that we, as systems professionals, use to describe our products. In this case, the prototype is a mock-up of the proposed system, giving the client a comprehendible picture of the system's capabilities and deficiencies.

In addition to gaining information about the functionality of the proposed system, a prototype will provide much feedback about the user interface itself. There is a growing body of material on human-computer interaction, but little of that has been codified in a way which can help our systems design process. But actually sitting down at a system, even a mere mock-up, provides a quick diagnosis for misplaced buttons and counterintuitive command names.

In contrast to an industrial setting, where it is assumed that a substantial majority of team members are well-versed in the development platform,[1] teams in the Information Systems class often face projects with little or no experience in the platform with which they will be working. Therefore this class refocuses the second goal from the general to the very particular: to provide teams experience with their platforms.

Focusing the goal to platform exploration changes the way prototyping as done. When the only reason for prototyping is to have a mock-up to show the client, prototype can be a "surface" product. The client would be told, for example, to search for John Jones because the information about Jones was hard coded into a few print statements. When the prototype is also the vehicle for learning the platform, the prototype needs to be "deep", exploring features of the platform which the client never directly sees.

Another way in which a platform-exploration prototype differs from a user-needs one involves exception handling. Other than perhaps displaying one sample error message, a user-needs prototype would assume that the user's input is correct, or at least not bother to verify it. But making the real system "industrial strength" involves a great deal of error checking and exception handling. Unfortunately the platforms used for class projects are

1. In these settings, the introduction of new tools or methodologies are projects in their own right, identified, managed, and budgeted specifically for staff development.

often awkward when it comes to implementing exception handling, so you are required as part of your prototype to implement at least some error routines.

One area of error checking which is particularly important is "referential integrity" checking. This occurs when the system checks that a value in one field which "refers" to information elsewhere in the database actually matches some value stored there.[2] For example, the procedure RECORD_CLIENTS_TIME used by an attorney to enter service time information will verify that the client is valid. Because this kind of checking is so important for insuring database integrity, your prototype is required to do at least one referential integrity check. Database tools are getting better at providing support for input validation and referential integrity. With such tools the error checking requirements for the Prototype are easy to meet; the essence of the requirement in this case is that you learn what error checking features your tool provides – and what you will need to implement for yourselves.

2. Platforms

If your team is exceedingly fortunate, several of the members will be experienced with the tools you will be using. If your team is typical, however, you will need to learn those tools. The best way to quickly learn about a new environment is to ask those who already know it well. Look for newsgroups or web sites that specialize in your tool.[3].

There may also be on-line archives (code libraries) that are good sources of examples.

A. Form Development Toolkits

This section provides suggestions for working with modern development platforms that provide "drag and drop" form implementation. These platforms require an approach to design and implementation that is vastly different than the one used for traditional imperative languages. The discussion in this section will center on Microsoft Access because it has been the most frequently used platform in recent years. However Access is only representative – as do many web development toolkits fit the same general pattern, (however, there are other quite distinct aspects of web development which will be covered later).

The dominant factor governing design and implementation in Access is that the application implementation is necessarily diffuse. An Access application is built up by constructing forms using menu choices, populating those forms with controls selected from palettes provided by the toolkit, and attaching actions to these controls. Thus the forms reflect the overall structure (a mostly treelike structure) of the application; Access implements that structure within its jurisdiction; and the programmer only controls the leaves Although this is structurally the antithesis of the monolithic program, it is no more reflective of good modularity. The single huge program fails to reflect the structure of its components[4]; an Access "program" is so fragmented that its significant components are unrecognizable.

2. A constraint that enforces referential integrity is known as a *foreign key constraint* because the value referred to should always be a key to a different (hence "foreign") table.

3. For example, `comp.databases.ms-access` and `www.acc-technology.com`

4. Or, even worse, its structure is so ill-defined that it cannot be separated into components.

The consequence of this situation is that a platform that at first appears to obviate the need for design in fact makes design even more essential. It is very easy to create a simple Access application – providing that ease was a major objective of the software and that objective has been well-achieved. Creation of tables and forms, specifying content and layout, and choice of actions bound to controls can all be done merely with a mouse; the only mandatory typing is to provide text for names and labels. Thus the *decór* of a system can be developed without an an application architecture.

Finally, a few implementation suggestions.

- Facilitate configuration management by defining a core module, what Access calls a "database," that has links to modules developed separately by each team member.
- In order to retain complete control over when tables are updated, do not bind form controls to table fields. Instead write two procedures for each form, one that loads the form from the database and one that stores the form into the database. Among other advantages, this avoids error messages that occur when Access tries to write a bound form with incomplete identifiers.
- The "Documenter" tool in Access (available on UCS machines) [5]) provides a good view of the details of your database. To reach the Documenter, select "Tools" from the menu bar and then select "Analyze." After running the Documenter on your system components, select "Save As Table" from the "File" menu bar item; the table this produces can then be used as input to your own reports.

B. Web Interfaces

There is a spectrum of technologies to deliver information systems via the web; where your project falls along this spectrum greatly influences its design and implementation. The entire spectrum shares the same basic architecture, the "client-server" model, varying most significantly according to what is "client-side" and what is done "server-side."

One extreme of this spectrum, the client software (*i.e.* the user's browser) plays a very passive role, only displaying each page sent from the server and transmitting back to the server the very same page with the user's keystrokes included. The typical platform for this approach is Perl-CGI,[6] where Perl scripts on the server scan returned pages and format outgoing ones.

Perl-CGI technology has two essential characteristics with serious implications on design: (i) interaction events are "heavyweight" and (ii) server scripts are stateless. Characteristic (i) arises because the page is shipped back to the server on every event and the reformatted version is then returned, incurring communication delays that are relatively large. Hence, client-side, try to avoid pages where the user enters (or otherwise specifies) only one value or parameter. Characteristic (ii) arises because the first thing that happens

5. Even if you are not developing for this version of Access, you may use the Documenter after converting your objects to Access. Note that you *cannot* convert back, so don't do any development here if your final target is not Access!

6. CGI stands for "Common Gateway Interface".

when a page is returned to the server is that a Perl script is evoked with *only* the returned page as a parameter.[7] The page must therefore contain sufficient information that the script can determine who returned it, what they were doing, *etc.* In other words, the Perl scripts must encode state information in the outgoing pages and extract that same information from returned pages – fortunately the CGI toolkit helps.

The other end of the client-server spectrum does not have such a well-defined extreme, in that server must necessarily do a substantial amount of the processing, including all database activities. The typical execution environment for this approach uses Java on the client side, with any number of toolkits facilitating implementation. Therefore the design considerations are essentially the same as with the other form development toolkits covered above.

One vital consideration when doing web-based interaction is that there are a host of different client platforms – a consideration that has immediate consequences during the prototyping. This consideration is especially important when Java is used, since (in spite of standards) the Java execution environment is not the same between Netscape and Explorer and even varies some between different versions of these products. Hence the prototyping effort must try out a variety of browsers.

3. Coda

A student from a previous Information Systems class left us the warning:

> Believing "We don't need to know the tool now; we can deal with that during implementation." is a major invitation to disaster.

7. In this case, the operating system provides no context or state information to the script, hence the term "stateless".

IX. Design

1. Motivation

The design phase of the software life cycle is where developers identify the structures that will satisfy the Requirements Specification. The final design document should include every detail of a solution. A Requirements Specification tells *what* is needed for a system; a design describes *how* the system meets those needs.

The design process gives developers the chance to work out careful solutions on paper before committing themselves to an implementation. If the design document is not sufficiently detailed, decisions about how to meet given requirements are pushed back into the implementation phase. This provides less opportunity to review a solution and often leads to errors that are very difficult to detect and costly to fix. In particular, a decision made during design should hopefully be focused at one place, while deferring that decision into implementation may cause that decision to be dispersed into many places, with inconsistency a likely result. For a simple example, the type of a parameter can be specified once during design, when the component interfaces are defined. If delayed until implementation, on the other hand, the decision on parameter type must be made independently for each component. Fortunately the compiler will often catch type inconsistencies; we are not so luck with situations that are less syntactic. See IX.1.B below.

On the other hand, design decisions are vitally dependent on the software tools. A design that is well-suited for Access on a PC can be a disaster for a web-based interface to a central database server. Paradoxically,[1] tools that facilitate the construction of simple applications seem to complicate the design of complex ones. That is because the implementation of the functionality gets parceled out to a myriad of little actions hooked to screens and menus and thus overall coherence is lost.

> You cannot design without understanding the software tools. Learn the tools before design.

A design document is sufficiently complete when the description of any component has enough detail so that component may be implemented without reference to other components. Global standards and declarations of course can and should be consulted during the coding. The design activity also looks forward to the final project documentation: good documentation comes from good design.

A. Difference between Preliminary and Detailed Designs

When working out a design, developers often iterate through the design process several times. They conceive of different design alternatives and work out the implications of those alternatives. Creating two separate documents, a preliminary design and a detailed design, makes this iteration more explicit. Dividing the process in this way provides an opportunity for a possible solution to be examined closely by reviewers not on the design team before the team spends an inordinate amount of effort fleshing out a problematic design.

1. Perhaps this is the origin of the name of a once-popular database product, Paradox.

The primary difference between the Preliminary and Detailed Designs is the amount of detail used to describe a system's components. In a Preliminary Design, it is sufficient to describe what each component does without saying exactly how it does it. For example, the Preliminary Design might specify a module `Sort` that, given an unsorted list of names, outputs an alphabetically sorted list, but not specify the exact algorithm `Sort` uses to accomplish this. The details from the Preliminary Design can easily be transferred to the module headers for coding. The detailed design must specify, with pseudocode or the equivalent, precisely how each component performs its functions. With the `Sort` example, the Detailed Design could show exactly how `Sort` sorts a list, perhaps with pseudocode for a merge sort algorithm, or could merely be a citation to a standard text which gives the heap sort algorithm.

If you want to do lots of unnecessary work, start over from scratch for the Detailed Design. However, if you want to avoid busywork, plan your design document so that it expands from the Preliminary Design to the Detailed Design without major restructuring.

B. Consistency

We have already discussed consistency in various contexts. We now return to focus on this topic (or rather its opposite, inconsistency) because it is so crucial. After blatant miscommunication between client and development team,[2] inconsistency is the most common cause of system failure.

In order for inconsistency to arise, some aspect must be repeated in the system. Depending upon what is repeated, several kinds of inconsistency are possible:

- syntactic, arising from repeated variable use – An example is inconsistent variable typing. Most modern compilers check compilation units for type inconsistencies, but it still occurs when linking external modules.

- data structure, arising from repletion of programmatic manipulation of one data structure – For example, different pieces of code may assume different orders for items on a linked list.

- informational, from repeated use of files, relations, or other data stores – For example, one component assumes `Employee.address` refers to home address while another assume it refers to work address.

- operational, from repetition of operations or system functions – An example occurs when "Delete" is a separate operation in some interface components, while others include it under "Update."

- user interface, from virtually any repetition of window "widgets" – For example, non-updatable fields are colored blue in some screens while they are gray in others.

2. Even miscommunication is often a form of inconsistency – of vocabulary.

2. Major Components of the Design

The difference between preliminary and detailed designs becomes apparent when considering a Zachman framework. The components constructed by "designers" are essentially those of the Preliminary Design, while those of "builders" are of the Detailed Design.

	what *data*	**how** *function*	**where** *location*	**who** *people*	**when** *time*	**why** *motivation*
specify	information model (ER)	process flow model data flow model	logistics network	organization chart	master schedule	business plan business rules
design	relation schema	HIPO	distributed system architecture	human interface architecture	processing schedule	formalized business rules
build	table layout	component specifications		interface details		knowledge representation

Table IX.1: Specification and Design Components in a Zachman Frame

3. Layout of the Design Documents

A design document is a reference work for implementation and maintenance. As such, its layout is extremely important and one should plan it out very carefully. It is amusing to think what would happen if one were careless with the layout of other kinds of reference works, say a dictionary. A poor layout is a hindrance to system development and maintenance since it is an impediment to accessing needed information.

A. General Guidelines

In general, strive for completeness and ease of use. Include every detail that might be of importance in refining the design, implementing the system, or maintaining the finished system. Include text arguing in favor of design decisions. Also provide several mechanisms for finding design components, *e.g.* indexing and bidirectional cross-referencing.

i. Defending design decisions

Providing the rationale for design decisions is an important part of the design task. Since a design is likely to go through several revisions before it is completed, explanations prevent the revisers from exploring blind alleys that previous designers have already already investigated. In addition, these arguments make explicit the assumptions upon which the design is based and help readers to see errors in the design more quickly[32, page 346].

Statements of design rationale can be placed wherever the level of detail is most appropriate. For example, a component description might include the reasons for choosing one certain algorithm over others, while an introductory section describing the user interface

as a whole might defend a general form used throughout the system (*e.g.* always using menu selections as opposed to typed commands).

ii. Methods for making information easier to access

There are several ways to make parts of the design easier to access. These include indexing, bidirectional cross-referencing, ordering, and redundancy of information. Note that most word processing software can generate indices, tables of contents, and the like. Use these tools wherever possible, otherwise your cross-references will never be correct.

An index lists various components of the design along with their corresponding locations. It is a unidirectional reference insofar as the components referenced do not refer back to the index. There may be multiple indices, each indexing a different collection of topics. An example index is a list of global variables along with, for each variable, a list of the components that use the variable. This assumes that the component descriptions do not list the global variables used in the respective components. If there were such a list for each component, referencing between the global variables and the component descriptions would be bidirectional. Index listings are usually alphabetically ordered – the one exception is the table of contents.

Bidirectional cross-referencing allows various interrelated components to refer to one another. Unlike indexing, it allows one to go back and forth between referenced items. As an example, every component description could list all tables that the component uses and each table description could list all the components that access it.

Finally, to make information easier to find, the same information can be included in the document in different forms. For example, a hierarchy chart of the components repeats, albeit in a different format, the same information that is in the bidirectional cross-referencing between components. (*i.e.* every component description lists of all the other components that calling or called-by that given component). The hierarchy chart would show at least as much information as the bidirectional cross-referencing. However, if the hierarchy chart were large and components were numerous, including the bidirectional cross-referencing could be very helpful to a reader, taking away the necessity of flipping back and forth between component descriptions and a complex hierarchy chart. Of course this deliberate redundancy requires some extra effort checking for internal consistency.

B. A Sample Design Outline

We now outline a possible design document, with a short description for each section. This outline is merely an example to ensure that you think about the design documents adequately. The outline is suitable for either the Preliminary Design or Detailed Design, which means that it is necessarily general in places. It is merely an example and is not, repeat not, utterly complete and applicable to every situation. Above all, it should not stifle creativity when coming up with a layout for a new design document. This suggested outline borrows from [32, page 346].

Design Documents 1: Executive Summary

The executive summary is a concise description of the project for nontechnical people. It should describe the project in terms understandable to a broad audience and explain the purpose of the system and the design document.

Design Documents 2: Introduction

The introduction, as opposed to the executive summary, addresses a technical audience whose expertise is assumed to be high enough to understand the design in its entirety. It discusses such aspects of the system as the scope of the solution, the major functions of the system, hardware constraints, the operating environment, *etc.* It also discusses aspects of the design document, such as its relationship to other documents like the Requirements Specification and discusses the organization of the rest of the design document.

Design Documents 3: Requirements

The requirements section describes all the functional and qualitative requirements for the system. This includes all the quality expectations for the system, *e.g.*, requirements for the system's functionality, reliability, performance, *etc.* The information here comes from the Requirements Specification, of course updated by any information acquired since the team completed the Requirements Specification. The inclusion of this section is an example of how developers carry previous documentation forward to subsequent documentation. Finally, there should a correlation indicating how the given design meets its requirements. One good way to accomplish this for functional requirements is to use a table with the functional requirements along one axis and the components along the other. General descriptions of approaches to fulfilling qualitative requirements are sufficient.

Design Documents 4: Specifications

The specifications section is also carried forward in large part from the Requirements Specification, supplemented by the latest changes. It includes details of *what* the system should do, but not *how* it should do it. Examples of the types of items one should include are Entity-Relationship diagrams and Data Flow Diagrams of the proposed system.

Design Documents 5: Data Design

The data design section describes the data structures to be used by the system. In Information Systems, this is usually a relational database documented with relational schema, but other types of designs are possible (such as a hierarchy chart for a hierarchical database or class and object diagrams for an object-oriented design). Specifications for relations should include all pertinent aspects: keys, indices, functional dependencies, derivation rules for derived attributes, and other constraints.

As with all other aspects of the design, the data design should include design rationale and cross references into both the specifications and design documents. For example, relational schema should be related to the entities and relationships in the entity-relationship diagrams. Finally, text arguing for why the team made given design decisions may be very helpful to understanding the design.

Design Documents 6: Architecture

The architecture section details the specific pieces of the system, how these pieces interacts, and how they meet the system's requirements. It is sometimes called the "functional design" in that it lays out the design and function of system's components. In Information Systems, functional aspects of a system's design are usually documented with component hierarchy charts, module headers, and, in the Detailed Design, pseudocode descriptions of modules. The headers document the component interfaces, as do the hierarchy charts to a certain degree. Other types of charts and diagrams, *e.g.*, decision tables, state transition diagrams, process diagrams, *etc*, may be useful, too. The description for each component should relate that component to other aspects of the system. For example, module headers should describe which relations a given module accesses. Again, this is part of the larger issue of cross-referencing elements of a design document. As always, explaining why the team made given design decisions is very helpful in understanding the functional aspects of a system.

For platforms for which the user interface forms are primary means for structuring the application, such as Microsoft Access, the Functional Design section should follow the User Interface section.

Design Documents 7: User Interface

While it is possible to document the user interface as part of the data and functional aspects of the system noted above, it is generally wise to separate the description of the user interface in its own section. In environments such as Microsoft Access, where the user interface dominates the modularity, this section should precede the Architecture section. This section provides additional information to guide the implementation of the user interface. It should describe the general principles and formats of the user interface, document screen layouts, and of course provide design rationale. If menu organization is sufficiently decoupled from component structure (as it should be, see § IX.4.A below), the menu hierarchy could be presented at this point.

Design Documents 8: Exception Handling

The exception handling section should describe how the system handles errors, particularly catastrophic errors that might force the system to stop functioning. For example, one might describe what the system does when an error occurs while reading a database file or describe what the procedure is to restore the system after a power failure. This section is separate from the functional design section because the system will probably deal with errors, like a power failure, that the functional design cannot address directly. For example, the only way one could prepare for a power failure in the functional design, as opposed to having some kind of hardware backup power supply, would be to save data frequently and to provide mechanisms for data recovery after a power failure. Hopefully the software platform will bear most of the burden for handling serious errors. Still, you should document this fact.

Design Documents 9: Coding Standards and Conventions

The two most important standards concern modularity and documentation. Modularity standards include use of parameters and global variables. Internal documentation is best addressed with a module header template, which is often also used in the design phase, as discussed in the following section. Standards for screen layout could also be specified here rather than in the user interface chapter.

In a professional programming environment, this section would typically not be part of an individual design document, since the standards would be established globally for the shop. In any case, process standards should be established well before this point.

The material in this chapter is very specific to the particular tools required for your project.

Design Documents 10: Test Plan

Chapter X, Testing, details information important in constructing a good test plan. This section should address all issues relating to verification and validation of the design and implementation.

Design Documents 11: Glossary

As with the Requirements Specification, a glossary might aid users of the document in understanding unfamiliar terminology.

Design Documents 12: Bibliography

One should note all the references one uses directly in preparing the document just as with an academic paper.

C. The Project Dictionary Style of Documentation

One method for managing all the elements of a design, and the elements of a Requirements Specification as well, is to document them in a project dictionary.[3] A project dictionary is a collection of detailed descriptions of each element that comprises a project. For example, a project dictionary would have an entry for every component included in the design. Each entry includes all the data that the design team believes is pertinent for a given element. Thus, a project dictionary entry for a module might include the name of the module, any aliases used to refer to the module, all the other modules that call the module, all the other modules that the module calls, who wrote the pseudocode for the module, whether the module has been implemented, *etc.*

There should be a form or template for each type of element used to describe the project. For example, a template for relations would have places for the relation name and attributes. It would also include attribute details (domain, size, *etc*), functional dependencies, and constraints. Finally it would have design rationale, indicating how the relation

3. The project dictionary style of documentation is discussed at length in Whitten *et al.*[35], especially chapters 10 and 11.

was derived from the ER model. The team then accomplishes the relational design primarily by completing a copy of this template for each relation in the target system.

You must decide which types of elements are worthy of this high level of detail. Other relevant elements might be processes of the data flow diagrams, entities of the entity-relationship diagrams, components, screen forms, and menus. A template-based design methodology helps ensure completeness and uniformity of the design document. Another advantage of the template or form approach is that most word processors have tools that support the automatic generation of indices and cross-references from such forms. Using these tools greatly enhances the reliability of the indices.

Note that using project dictionary entries does not preclude other types of documentation. For example, one might include a project dictionary entry for each process of a data flow diagram and the data flow diagram itself. In a well-integrated CASE system, the two should be linked such that choosing a process oval on a diagram brings up its corresponding project dictionary entry. Many types of diagrams complement the information in project dictionary entries.

The following is an example of a project dictionary entry for a module used in the law firm billing system:

NAME: Client Billing Screen

ALIASES (including implementation names):

DERIVED FROM DATA FLOW MODULE: 2.1.1

ENGLISH DESCRIPTION: enter & validate client name, provide button to invoke Prepare_Bill

CALLS: Help_Message, Error_Message, Prepare_Bill

CALLED BY: Main

PRECONDITION: None

POSTCONDITION: None

RELATIONS USED: L_Client

ASSOCIATED SCREENS: Main, PickOneClient, DisplayBill

CODED BY: Kurt Goedel

CODE HISTORY:
completion due: Feb. 22, 1994

TESTED BY: A.M. Turing

TEST HISTORY:
testing begins: Feb 22., 1994

BUTTONS (with shortcut keys):

Help (F1)	/*standard*/
Exit (ESC)	/*standard*/
Bill (ALT-B)	labeled 'Bill'

FIELDS:

CNameLast	labeled 'Last Name'
CNameFirst	labeled 'First Name'

PSEUDOCODE DESCRIPTION:

Event:	**Action:**
Help	Help_Message(02110)
Exit	GoToScreen Main

Leave CName	if CName<>NULL AND CName NotIn L_Client.name then
	Error_Message('client name not in L_Client table')
	Cancel Event
	endif

Bill	if CName=NULL then
	Error_Message('you must enter a client name before billing')
	Cancel Event
	endif
	SELECT INTO TempCL FROM L_Client
	WHERE L_Client.LastName=CNameLast
	AND L_Client.Firstname=CNameFirst
	if Count(TempCL) = 0 then
	Error_Message('no clients match these names')
	Cancel Event
	Note: this should never happen
	else if Count(TempCL) > 1 then
	GoToScreen PickOneClient
	endif
	PrepareBill
	GoToScreen DisplayBill

NOTES:

Note that the designers had a convention that only those preconditions and postconditions directly predictable from the module, and not those of the submodules, were included in the precondition and postcondition areas. They used these areas for informal notes on the expected inputs and outputs of modules and did *not* use these areas for any kind of formal verification.

The entry encapsulates much information about the module. One can tell, for example, that this module uses one relation from the system's database. Examples of other facts that the designers chose not to include with this entry, but could have included, are which requirements this module helps to fulfill and which diagrams (*e.g.*, hierarchy charts) help to document this module. This does not mean that the information is completely lacking in the design document, but only that the designers decided not to include it here. They could have included elsewhere in the design document, for example, a table showing all the requirements and the components that fulfill them. A notion of the algorithm's performance (*e.g.*, noting whether it is O(n) or O(n!)) is not relevant for this milestone but should be included in the NOTES: item, if relevant.

The following is a sample from the same system of an entry for a relation (in this case,

a relation implementing a many to many relationship):

NAME: Attorney works on Case

ALIASES (including implementation names): WORKSON.DBF

ATTRIBUTES:

impl. name	**type/ size**	**local key**	**foreign key**	**remarks**
AId	integer	√	Attorney.AId	
Case#	integer	√	Case.Case#	
Date	date	√		
Time	hh:mm	√		clock time of action
Minutes	integer			time spent on client
Desc	memo			

RELATED ENTITIES: Attorney, Case

USED BY:

APPLICABLE FUNCTIONAL DEPENDENCIES:

NORMAL FORM: BCNF

DERIVED FROM: works_on relationship

It is immediately apparent that the relation entry differs greatly from the above entry for a component. Again, there is information that the designers could have included but did not. For example, the designers could have included the names of other diagrams that documented the relation, but chose not to do so. Also, the designers chose to include specifications for the data elements, such as Pages, instead of having separate project dictionary entries for them and chose simply to show local and foreign keys in the table rather than noting them separately.

4. System Decomposition

Effective design of systems of any substantial size requires decomposition the problem into smaller, more readily solvable subproblems. With data, relations are decomposed into smaller relations that have desirable normal forms. With procedures, large, difficult to comprehend components are decomposed into smaller, more easily understood components.

A. Difference between a Component Hierarchy and a Menu Hierarchy

It is important to distinguish the user interface menu hierarchy from the software component decomposition hierarchy. The former indicates only the steps users go though to access or manipulate information. The latter indicates the subdivision into functional areas.

In our law firm example, assume that the top level decomposition has one major module for most functionality and a few small modules for system maintenance, as seen in Figure IX.1. However, this is not a good menu organization to access this functionality (for reasons explained later in Section IX.5). A much better menu structure puts the

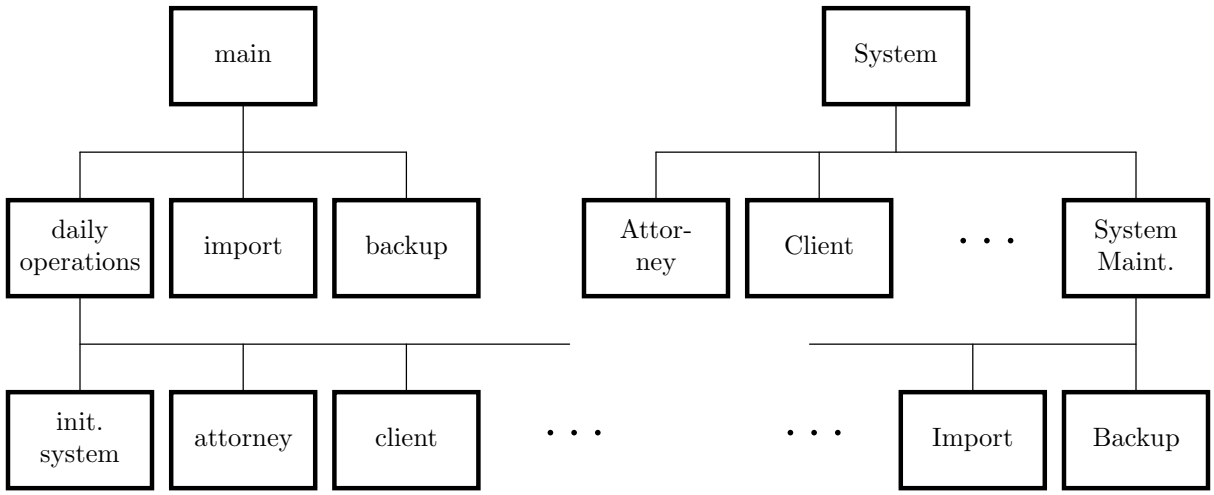


Figure IX.1: Module and Menu Hierarchies

major functionality at top-level along with another item to access the occasional system maintenance functions.

From the perspective of functionality, a sequence of menu choices is often only a technique to provide multiple parameters to some executable procedure. The order in which these parameters are picked does not affect the functionality, but this order defines the structure of the menu hierarchy.

One convenient "rule of thumb" which gets the design started right is to initially assume a complete and distinct split between user interface and data manipulation, even though most application generator tools cause the eventual retraction of this assumption. In this hypothetical decomposition, any information exchange to or from the user is the province of the interface screens, while any information access or manipulation is handled in the code modules. Carried through to the end, this results in a procedural decomposition that has only one circle for all of the user interface and a menu hierarchy that has procedure calls at all its leaves. Another advantage of separating the user interface from the module design is that this points out where the same functionality may be accessed by various paths.

A common error, in Information Systems and elsewhere, is to think that the user interface menu hierarchy bears a one-to-one correspondence with the system components. However, this is almost always incorrect. A menu hierarchy will not show "overhead" procedures, such as procedures that initialize a component or procedures that prepare the component to exit. For example, a module `Daily_Operations` might have submodules `Initialize System` and `Prepare System for Exit` which do not appear on the menu that `Daily_Operations` manages. More generally, the menu hierarchy usually does not go into enough detail about how given procedures divide into sub-procedures. For example, the option `Print Report X` may appear on a menu. Where this would be a leaf in a menu hierarchy, several modules may be needed to implement this functionality, such as `Get Report X Data`, `Calculate Totals`, and `Send Report X to Printer`, all of which

should be shown in the component hierarchy.

In spite of the above advice, software tools may impose serious constraints on the component architecture. For example, Microsoft Access considers the Form to be the top-level components but provides no mechanism to organize the set of Forms.

B. The Relationship between Functional Decomposition and Data Modeling

Almost all of a system document's diagrams describe either the system's information model or the functional model. In Information Systems, it is typical (and typically recommended) to first develop a information model and then focus on procedures that manipulate that information model.

It is interesting that either the system's information model or functional model can provide the underlying organization for the menu hierarchy design. The two hierarchies distinct in Figure IX.2 arise in this way.

C. Designing Modules

A module is an independent unit of code, of the right size that it can be understood, programmed, and maintained. The decomposition of modules into submodules helps establish this characteristic.

There is also an interpretation of "module" as merely a file unit in code is stored and manipulated. These two notions – concept and container – are linked but only weakly so. It is often advantageous to keep a top-level module and all its submodules in a single file (or whatever the relevant container is called). However, to call a jumble of code components a "module" merely because they are in one file is a misuse of the idea.

i. Cohesion and coupling

The standard recipe for module design identifies two major dimensions along which module structure varies. The first, cohesion, measures how much a single module belongs together, in terms of common functionality, shared data, *etc.* The second, coupling, measures how much the collection of modules interacts. The obvious objectives for architectural design are the maximization of cohesion and minimization of coupling.

ii. Design for testability

Although this is highly dependent on the platform and development environment, the general notion is to keep in mind the need to "instrument" your code while doing the design. For example, you may specify a collection of procedures which logs the actions of your software as it proceeds from step to step. More on this subject may be found at the end of Chapter X, Testing.

iii. The size of modules

There are several rules of thumb about the proper size of modules. It is common to hear that a module should not exceed a page of pseudocode. There are also recommendations that a module should not be shorter than ten lines, but that recommendation is

questionable. The most important consideration about size of modules is that they are small enough to be easily understood and highly cohesive.

As a rule of thumb, modules (and components in general) should be decomposed until a programmer can understand exactly how to implement each module with the given information model and the data representation's primitive operations, *e.g.*, with relations and SQL statements or, in an object-oriented system, with objects and the operations defined for those objects.

As noted above, the minimum length suggestion is not always a good one. In fact, it is better to have modules that are too small rather than modules that are too large. First, each line of pseudocode typically expands into several lines of source code, so a module described with less than ten lines of pseudocode may expand to over ten lines. Second, even if one implements some modules with less than ten lines of code, the prevalence of optimizing compilers helps to mitigate any effect they might have on program efficiency. On the other hand, if modules are too large, errors are harder to detect and it is harder for one to grasp their functioning conceptually.

Given a small module, there is still an issue with writing pseudocode, If a module is both small and trivial, so that its implementation is immediately understood just based on the modules description and external behavior, it is a waste of the writer's and the reader's time to lay out pseudocode. Thus there are occasions when a small module does not merit pseudocode. Before you decide to omit the pseudocode for some module, however, think about whether there are errors or exceptions to be handled, whether there a parameters which might be passed or might need default values, whether special cases need handling, and other subtleties.

5. User Interface

There is a huge range of issues that impact user interface (UI) design, concerning everything from the software platform to the "human platform". Since there are a number of good texts in this area, we will only touch on a few matters that are especially important or illustrative of underlying principles and considerations.

User interface design must always keep the users in mind. This includes not only overt preferences, but habits and unconscious behaviors.

The foremost characteristic of users is their diversity, not only among the various users but among the tasks and actions of individual users. Users have more or less knowledge about the application domain, more or less experience using computer systems, and more or less self-confidence in trying new technology. A particular task may require thoughtful use of the software (the term "mindful" is often used, and "deliberate choice" is a good characterization) or it may be rote or repetitive. Programming a VCR is typically mindful, in that it involves deliberate choices. A supermarket scanner, on the other hand, is very repetitive.

A. Appearance and Layout

- Illustrative of the impact of human physiology, the human eye cannot simultaneously focus on certain color combinations, so a user interface should not juxtapose red and green. There are a large number of design guidelines to accommodate people who are color blind or otherwise visually impaired.
- Illustrative of the cognitive impact of layout, use white space roughly every five items.
- Illustrative of the impact of appearance on navigation and operation, well designed data editing forms often highlight changes in color. The convention of reporting errors in red is of course very widely used.
- Illustrative of how the platform can complicate user interface design, some software adjusts form size based on monitor resolution while keeping field size fixed, thus sometimes hiding fields.

B. Navigation and Operations

Navigation may be form to form, data item to data item, button to button, or mode to mode – that is, navigation between modes of operation. For example, it is common to have distinct *search* and *edit* modes. Two (or more) modes may be available on the same form and a change in mode often changes the available operations. A mode (or more correctly, a change in the mode) is sometimes confused with an operation. Part of this confusion certainly arises because the interface implementing these mode will have `Search` and `Edit` buttons for changing modes and might a `Save` button as well. `Save` is primarily a data manipulation operation but it also changes from *edit* to *search* mode.

Navigation between forms or fields is typically more easy to automate; navigation between modes is more mindful. Mode confusion is also more problematic, therefore a polite interface will indicate the current mode. One way to achieve this, continuing the above example, would have a single button whose label and associated functionality changes from `Find` to `Save` when the mode changes from *search* to *edit*.

Pay attention to how a user interface facilitates or impedes navigation. A simple example of using hierarchy organization to facilitate navigation was seen in the module verses menu example of Figure IX.1. The assumption was that users almost always perform Daily Operations and thus they are taken directly to the choices concerning those operations, while access of infrequent operations takes extra steps.

In our law firm example, there are a number of entities – attorney, client, case, and transaction – that are maintained by the secretary with essentially the same operations – find, enter, update, and delete. There are thus two obvious menu hierarchies: first choose an entity and then choose an operation, or first choose an operation and then an entity (see Figure IX.2). Which parameter is chosen first obviously has no impact on the system functionality, but this choice produces user interfaces with different "feel".

It is likely that the user will perform several operations on one entity involving several instances of that entity type. Thus it is convenient to have the secretary pick `Client` once

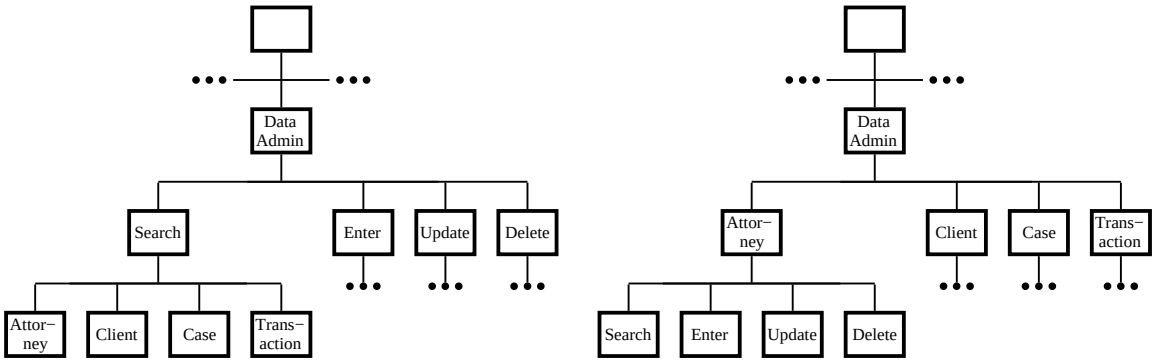


Figure IX.2: Two Menu Hierarchies with Equivalent Functionality

before engaging in a sequence of searches and data manipulations on client information. In general, mode changes and mode-related operations should be low in the choice hierarchy, on forms where mode changes effect behavior concerning the relevant entities.

A few more specific recommendations:

- Users typically perform better when given a linear flow of tasks. Therefore, wherever possible, navigation should be automatic based on task step (this notion was discussed at the end of the discussion of dataflow diagrams, section 6.3 of the Requirements Specification document), as this gives the appearance of a linear flow.
- Make common things easier and more accessible. Figure IX.2 shows two way to organize access to the same functionality and the associated text describes why the right hierarchy is preferable.
- Your software tool will make many assumptions as it builds your application. Be aware of those assumptions and change the default settings as appropriate. For example, in many tools it is possible to specify an appropriate "tab order", which is the order in which fields as visited as the user successively hits the tab key.

Exercises

1. Section 7 of the suggested design document only covered interfaces with human users, not interfaces with other information systems. Why is the later material not covered in that section?
2. Some of the best user interface designs appear in game programs, as one would expect with products whose entire success depends upon appealing to end users. The most ubiquitous of these are solitaire games, which are included in almost every system distribution. Pick a standard solitaire and play the game, imagining yourself as a novice user just becoming familiar with that interface. What navigation steps might a novice user have difficulty finding? What operations might be confusing? Is the visual layout understandable?

X. Implementation and Testing

Although commonly listed as separate phases in the lifecycle, implementation and testing are thoroughly intertwined. In particular, looking at implementation as a series of subphases from coding through installation reveals different kinds of testing appropriate for the respective phases (more in § X.2.C below).

Making testing a separate lifecycle phase gives it significantly greater prominence. Unfortunately, although testing is an essential part of any software project, it is often neglected or relegated to lower priority than design and coding issues. Especially in the context of the Information Systems course, we attempt to insure that testing remains a high priority. Whether testing is considered a distinct lifecycle phase or is integrated with coding and installation, every module of code needs to be extensively tested with all kinds of test data and by different types of users in order to make sure the product delivered conforms to the requirements in a trouble-free manner.

This chapter first discusses process issues that arise during coding. Then the remaining sections describe in detail some of the important factors of software testing. The second section (the first focusing on testing) discusses the concepts behind testing, particularly the way various kinds of testing fit into the overall project. The third section describes the Test Plan and the Test Log, which together document all testing. Methodologies for generating tests are covered in the fourth section. The fifth section describes the roles different individuals play in the testing process. Testing considerations that don't fit nicely into the above categories are covered in the final section.

1. Process Issues during Coding

Many important coding issues are beyond the scope of this *Guide* because they are specific to a particular tool or toolkit. Therefore this section will discuss process considerations that are significant to any team coding activity.

The paramount process issue is indicated by what is missing in this section – the toolkit. The importance of tool experience during design was covered in Chapter IX; the need for such experience during coding is absolute. In a commercial setting, it is expected that the team include developers with substantial tool experience. In this course, we may not have that luxury. Hence we have made tool experience a major goal of the Prototype Milestone. While that experience should be adequate for design, it is probably not adequate for coding. Therefore project planning should include considerations of thoroughly learning the relevant tools.

Planning and scheduling for coding tasks is based upon the modularity developed during the design phase. Dependency among these modules is of course the first thing considered when scheduling. Unlike scheduling physical construction (*e.g.* the basement of a house must be completed before framing can begin), there is no concrete reason why coding of one module must wait upon another's completion. However, integration and testing considerations (discussed in § X.4.A) are likely to impact the schedule. Because

the coding schedule should indicate not only "when" but "by whom", knowledge and skills of the team members are important in the scheduling.

Another implication of limited tool experience is limited accuracy in predicting coding time for specific tasks. Your plan should address this with specific schedule review points (say, after one, two and four weeks of coding/unit testing). At each such review, the team should attempt to discern why certain coding tasks are behind schedule[1] and reallocate or reprioritize tasks.

Because of the the interdependence of modules, it is important to know the current status of each module, especially for testing. Vast amounts of frustration are attributable to failing tests not because of errors in the module supposedly being tested but because of incorrect status in associated modules. Because of multiple people changing code, there is a danger of concurrent update problems. Because of the likelihood of shifting hardware platforms, there is a danger that the current version will be lost entirely. Every year there are one or two teams who arrive in class with very sad faces, having lost weeks of work in a system crash.

The answer to all these problems is to adopt good Change Control and Configuration Management (CCCM or C^3M) procedures. Note that simply using a CCCM tool is not sufficient (and using a version control tool, which is lighter weight and handles change but not configuration is even worse). Your team must adopt and follow well-defined CCCM process steps. These included *controlled* check-out and check-in, documentation of variances in specification and design, and coordination with testing. The major responsibility of the team Librarian is to oversee CCCM.

2. The Concepts behind Testing

Testing has been defined as "[T]he process of exercising or evaluating a system by manual or automatic means to verify that it satisfies specified requirements or to identify differences between expected and actual results"[19]. Testing is a part of another major software engineering issue, *Verification and Validation*,[2], which are in turn the culmination of *Quality Assurance*.

Testing is also one aspect of software measurement, an important aspects of software projects beyond the scope of this *Guide*. However, remembering this fact helps put us in the correct frame of mind for testing. Pressman has a good overview of software project metrics[25, chaper 4]; material on metrics is often classified under software cost estimation (*e.g.* [6]) even though it has applications much broader than cost estimation.

A. Goals for Testing

As always, it is valuable to understand the reasons why before considering an activity. In your previous programming classes, your testing only had one question to answer: "Does this program produce what the prof expects?" However, this goal ("Correctness" in the

1. Sadly, it is almost always the case that tasks are behind. A task ahead of schedule is a cause for great celebration and even greater amazement.
2. Testing is in fact part of Verification only, Validation is concerned with the adequacy of the requirements.

following list) is only one of many. Because each different goal is reflected in different characteristics in the software, each requires different means and methods of testing.

The following are software quality characteristics, after McCall *et al.*[22]; these characteristics translate into goals in obvious ways. Those characteristics that are the primary focus of testing (at least during implementation) are marked with •. However, in specific contexts, any of these characteristics could be tested.

- *Utility/Correctness*: the extent to which the system satisfies specifications; the presence of function.
- *Reliability*: the precision with which the system performs as expected; the absence of failure.
- *Security*: the extent to which access (to system and included information) is controlled.
- *Usability*: the effort required to learn, operate, and interpret behavior of the system.
  - *Efficiency*: the amount of computing resources required.
  - *Maintainability*: the effort required to locate and fix errors in the system.
  - *Modifiability*: the effort required to modify the system.
  - *Testability*: the effort required to test the software and the system.
  - *Interoperability*: the effort required to couple to other systems.
  - *Portability*: the effort required to transfer the system to different environment.
  - *Reusability*: the extent to which software (and possibly other system components) can be used in other applications.

Testing should not be confused with debugging Testing answers the question "what are the failures?" while debugging answers "why do the failures occur?" Debugging is necessary only when one or more of the tests comes up with incorrect results.

B. Overview of Testing

All testing involves the following seven steps(after [23]):

1. Select the program objects to be tested and the aspects to be measured for each object.
2. Decide how each measurement of each object will be tested, creating a *test case*. There may be more than one test case for a given object and measurement.
3. Develop the mechanisms and test infrastructure for running the tests.
4. Develop the data and specific test steps for each test case, known as the *test instances*. There may be multiple tests instances for one test case.
5. Evaluate the set of test cases, ensuring that all relevant aspects of the system will be measured. On small projects, this is often informal and included within the selection and development of test cases. on larger projects, it is a formal and distinct step.

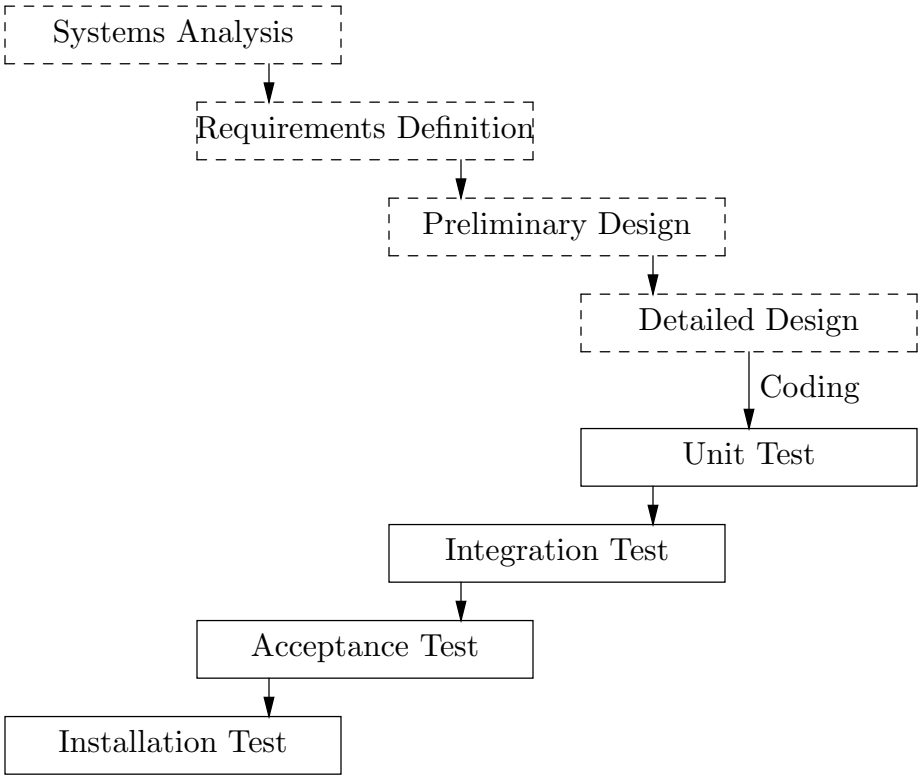


Figure x.1: Testing Strategies Mirror Life Cycle

6. Determine what the expected or correct result of each test case should be and create the *test oracle* that compares the actual and expected results. The oracle may be computerized, manual, or mixed.

7. Execute the test cases. When several tests or test instances are regularly exercised together, they form a *test suite*.

8. Compare the results of the test to those of the test oracle.

The rest of the sections in this chapter would discuss in detail these above steps.

C. Classification of Testing

Testing can be classified from many different points of view. The following two sections deal with the two main ways of treating the concept of testing, first from the systems development process point of view and then from the methodologies point of view.

i. Stages in testing: testing strategies

The structure of project testing mirrors the waterfall lifecycle, as is shown in Figure X.1, with four phases of testing. The first two of the four phases, unit testing and integration testing, emphasize testing the code to make sure that all the functional requirements are met; collectively they are sometimes called "program testing." The latter two apply testing to the system as a whole, of which the programs are only a part, and are thus sometimes

called system testing. Adding a Validation step mirroring the Feasibility Study completes the picture of the life cycle (note that Requirements Analysis has been broken down into two sub-phases).

- *Unit Testing* is that testing done to individual modules after they have been coded. Unit testing is commonly performed by the programmer who developed the module. Strict guidelines are therefore required to avoid being trapped by the programmer's preconceptions about his/her code. Unit testing is accomplished by running a module independent of other modules. Also there are several ways of testing modules without actual execution: program reviews, code walkthroughs, and formal proofs of correctness.

  It is for unit testing that test infrastructure must be constructed. Test infrastructure is necessary both to provide context for the component being testing and to simulate the execution of procedures called by the tested component. Section X.4.A discusses test infrastructure.

  Formal verification, often called "program proving," is applicable only at the unit testing phase, for only there are specifications tight enough to allow proofs (and in fact only a few of those specifications so tight – we are a long way from user interface specifications that allow verification). Formal verification is a useful tool in the testing toolkit, but is not adequate alone. Actually running the code is necessary to check for errors caused by the system/platform environment and by other factors not taken into account in formal proofs. Donald Knuth expressed this idea with his quip "[b]eware of the bugs in the above code - I have only proved it correct, not tried it out."

  For unit testing, test data to must generated to cover all aspects of the program being tested. Some test data generation can be done with the assistance of (usually expensive) software (see Glass Box Testing below), other test generation is entirely manual (see Black Box Testing). However this kind of test instance generation itself is a hard problem, so in most cases some compromise needs to be reached. Section X.4 discusses test instance definition.

  Note that the "unit" of unit testing is conceptual and not syntactic. Abstract data types provide obvious examples of this; the unit for a stack ADT is all the associated procedures, not the procedures `push`, `pop`, and `empty` in isolation.

- *Integration Testing* exercises the modules as they are combined into the complete system with all the modules running together, in order to insure that they work correctly in harmony. This should be done only after the unit testing is complete, so any fault discovered at this phase is known to be due to an integration problem, *i.e.* due to the interaction of two or more modules. Problems discovered at this phase are usually due to faulty modular design and/or specification. For example, say that the specifications for some module are vague and ambiguous. A programmer reading these specification, intending to use that module externally, might assume one particular behavior from that module. On the other hand, the programmer who actually implements that module might interpret the specifications differently, resulting in behavior different from

that assumed by the first programmer. An erroneous execution of the first programmer's code then occurs, even though the component programs appeared to pass unit tests correctly.

There are many different approaches to integration testing. However, most of them involve viewing the entire system as a hierarchy of modules, consistent with the modular design. Based on this architectural view of the system, the different approaches to integration testing are: [24]

- ◇ *Bottom-up Approach*: The first step of this method is accomplished during unit testing, when each module at the lowest level of the system hierarchy is tested individually. Then, the next modules to be tested are those at the next higher level of the module hierarchy – those that call the previously tested modules. This approach is followed repeatedly until all modules are included in the testing. This method is particularly useful when many of the low-level modules are general purpose utility routines.

- ◇ *Top-down Approach*: In this method, the top level code, often one controlling module, is tested by itself. Then all modules called by the tested modules are combined and tested as a larger unit. This approach is applied repeatedly until all the modules are incorporated.

- ◇ *Modified Top-down Testing*: Sometimes instead of incorporating an entire level at a time, as in top-down testing, a modified top-down approach is applied, which adds one or a few modules, rather than an entire level, at each integration phase. If a good test infrastructure is in place, this approach provides more control than the simple top-down strategy.

- ◇ *Sandwich Testing*: This is a combination of the top-down and bottom-up strategies. The system is viewed as having three layers, just as a sandwich does: a target layer in the middle, the levels above the target, and the levels below the target. A top-down approach is used in the top layer and a bottom-up approach is used in the lower layer. The testing converges at the target layer, which is chosen on the basis of system characteristics and the structure of the module hierarchy. [24]

  This mixed strategy is advantageous because lowest level routines, especially those that manipulate storages structures, are clearly best tested bottom-up. Storage manipulation procedures often conform to (formal or informal) abstract data types and thus are very precise, reasonably concise specifications.

- ◇ *Big-bang Testing*: This is the brute-force method of integration testing, combining all the component modules together at once and testing for correctness. Its advantage is simplicity. Its disadvantage is the great difficulty in finding faults when errors do occur.

- ◇ *Vertical Component Testing*: A refinement of big bang testing, this approach integrates and tests a top-to-bottom slice of the system, commonly a specific functionality or cluster of related functionalities.

- *Acceptance Testing* is the next major step after the two phases of program testing. It involves testing the whole system, of which the programs are only a part. Many different sources of errors are taken into consideration, errors which are introduced into the system not only during the coding, but during design, testing itself, general use and maintenance. The primary categories of acceptance testing are:

  ◇ *Function Testing*: This step of testing ignores system structure and focuses on functionality. This is done by knowing only the functional requirements of the system and making sure that all of them are satisfied.

  ◇ *Usability Testing*: This step involves the client in testing, evaluating whether the system is useful and suitable for the client's operation. Even if the system meets all the functional requirements, such factors as confusing information presentation, discordant interface flow, or poor documentation could cause an otherwise correct software project to turn out to be unacceptable.

  ◇ *Performance Testing*: This step examines how well the system performs, that is, the manner in which it does its computations.

  Many different kinds of tests can be performed here: - like stress tests, volume tests, configuration tests, compatibility tests, security tests, timing tests, environmental tests, quality tests, recovery tests, maintenance tests, documentation tests.[24] Depending upon the particular project, some or all of these performance tests are applicable. Some of those applicable to the Information Systems course are discussed in a later section(§X.6).

- *Installation Testing* differs from acceptance testing in that it evaluates the system in its final execution environment. This phase involves installing the system at the customer's site and making sure that the system still adheres to all its requirements. A separate installation testing phase is not needed if the acceptance testing is done at the customer's site.

- *Regression Testing* is strategic but does not fit in any particular phase; instead it belongs in every phase. The purpose of regression testing is to make certain that parts of the system that have achieved validation retain that validation as the software evolves. A major concern is side-effects – even though a particular module has not been changed, its behavior and therefore its correctness may be impacted by changes to other modules.

  In principle, all tests should be repeated after every code change – obviously quite a chore. This chore can be lightened by remembering that the concern is the impact of side effects. For that reason, as a part of the Test Plan, dependencies between parts of the system should be identified and a subset of the entire test data should be used to check that change to one component has not corrupted other parts of the system.

  Regression testing should be integrated with the change CCCM aspects of the project.

ii. Testing tactics

Tests can also be classified according to the methodology used to design them. While the classification of the previous section dealt with stages of testing, reflecting the sequence of project phases, the present section classifies tests according to the methods employeed, particularly the basis from which the test instances are defined. These methods pertain to developing the both unit and integration tests, so the two methods of classification are somewhat independent, but not completely so. See Pressman for further discussion of testing tactics[25, Chapter 18].

Definitions of various test tactics are given here; suggestions for constructing corresponding test instances are given in sections X.4.C and X.4.D.

- *Black Box Testing*: Black box testing creates test based only on the functionality requirements for the software. The internals of the software (that is, the programs themselves) are not visible to help design the test cases, as if the software were contained in a black (that is, opaque) box. Using only knowledge of the functional requirements, test cases are designed to find errors including the following categories (see also [25, §18.5]).

  ◇ incorrect or missing functions,

  ◇ interface errors,

  ◇ errors in data structures or external database access,

  ◇ performance errors,

  ◇ initialization and termination errors.

  Methodologies for black box testing are discussed in section X.4.C.

  Black box tests are applicable to unit tests as well as the later stages of testing.

- *Glass Box Testing*:[3] Glass box testing is a test case design method that uses the actual program text to derive test instances. It is also a way to evaluate test cases.

  Using the glass box test design method, it is possible to derive test cases that:

  ◇ execute all independent paths at least once,

  ◇ exercise all logical decisions on their true and false sides,

  ◇ execute all loops at their boundaries and within their operational bounds,

  ◇ exercise internal data structures to assure their validity, and

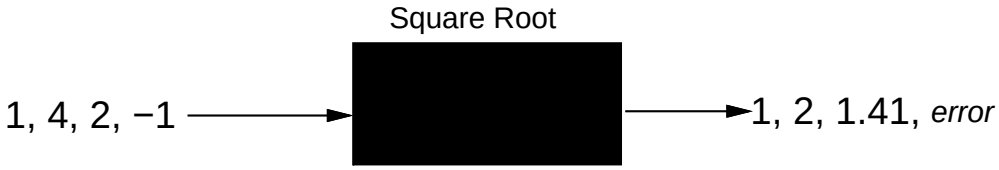
  ◇ follow all program data flows (the dependence of a variable from place of assignment to place of use).

3. This is also known as "white box testing", placing it in opposition to black box testing. However, the term "glass box" is more meaningful, suggesting translucence rather than lack of color.

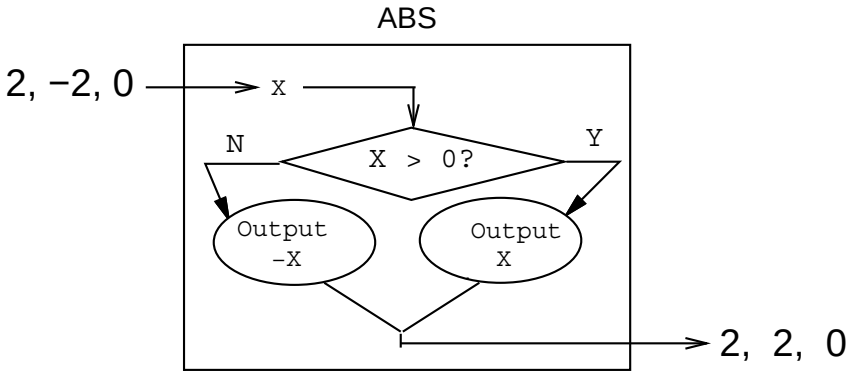
Basis path testing and loop testing [25, §18.3 and 18.4] are special types of glass box testing which deal with some of the above cases. Glass box test design is discussed in terms of a detailed example in section X.4.D.

Because most glass box testing deals with the flows of control or data within one module, it is most appropriate for unit testing.

- *Mixed Testing*: Sometimes being blind to the internals causes testing to be excessively complicated and makes finding appropriate test cases difficult. In these cases, having some knowledge about the code helps. However, for mixed testing the dominant factor in defining test cases is still the specifications. Another kind of mixing occurs when black box methodology is used to design tests and glass box methodology is used to evaluate them; this is often a very advantageous partnership.



A BLACK BOX TEST



A GLASS BOX TEST

Figure x.2:

Figure X.2 illustrates the difference between glass box and black box tests. In each case, the scaffolding feeds a sequence of values to the object being tested and observes the values. In the top of the figure, the procedure "Square Root" is being tested with a black box test. The specifications for Square Root state that the result is a real number that, when multiplied by itself, equals the original input. The test driver would then square

each output and compare it with the corresponding input.[4] Specification and testing in this example requires knowledge of error tolerance, since 1.41×1.41 is 1.9981 rather that 2, and of unacceptable inputs, since complex roots are not allowed. In the bottom, the function "ABS" is subject to glass-box testing. There is a decision box to check whether or not X is greater than 0. We have to make sure that both exits of the decision are followed at sometimes during the test, so both positive and negative values need to be fed into the procedure.

3. Test Documentation

As is true with other phases of an engineered software process, the testing phase needs to have a well laid out documentation procedure. The planing and performing of every test needs to be well documented.

The first important step in a testing procedure is to lay down the plan for the testing in a systematic manner. The main objectives of test planning, as enumerated by Pfleeger[24], are to:

1. Guide the management of testing.
2. Guide the technical effort required during testing.
3. Establish planning and scheduling of tests, including specification of equipment needed, organizational requirements, test methods, anticipated outcomes[5], and user orientation.
4. Describe the nature and extent of each test.
5. Explain the way in which these tests will lead to a complete evaluation of the system function and performance.
6. Document specific test instances: inputs, procedures, and expected outputs.

Because the scopes of these objectives vary radically, from broad to highly specific, several documents concerning test planning may be considered. Items 1 and 2 and a major part of 3 are high level and would be governed by organization-wide standards in a professional setting. Accordingly, these items are largely incorporated in the Quality Assurance Plan for the purposes of the Information Systems course. Much of the Quality Assurance Plan is provided for you, both to simulate an industrial setting and to give you, for once, a breather.

This leaves items 4 through 6, plus the remaining aspects of 3, to comprise the *Test Plan*. The Test Plan is the central document upon which the whole testing phase is based. This document is created prior to the commencement of testing, and is included in (or concomitant with) the Detailed Design document(§IX). The test instance details are not included in any milestone document prior to testing, but are essential for methodical

4. This is a fortunate instance where the test oracle can be automated.
5. "Outcomes" indicates the consequences of a test succeeding or failing and is not to be confused with "outputs," that is specific sets of values.

testing. In this section, we first concentrate on the Test Plan and then give some details on the documentation done during and after testing.

A. The Test Plan

The Test Plan should encompass everything dealing with testing, including testing tasks and schedules, test cases, and test instances.

The Test Plan must specify schedules and deadlines for all testing activities. This part can be very much like the project plan, determining the deadlines for the larger steps first (for example, "unit tests completed for all modules") and then later refined to specify activities on lower levels (for example, scheduling individual unit tests). See [32] for additional information.

The Test Plan must also clearly mention how and by whom the tests are to be performed. So it directly enforces a task allocation. This is important because the black box tests on modules developed by one team member should be done by another member, so the tester does not have substantial knowledge of the coding details for that particular module.

On the implementation side, the Test Plan should include unit testing with glass and black box tests, integration testing for combining all the modules together, acceptance and installation tests. There are different formats for describing the tests, but representations commonly followed the information systems context use a set of tables, with the following information[6]:

- Itemization of the test classes. For black box tests, these are tables of cases from the requirements distinguished by different input types and by the validity and invalidity of the inputs. For glass box test, the cases are related to the syntactic structure of the program according to the relevant glass box material; presentation of glass box classes may be tabular or diagrammatic.
- List of test cases - this is essentially a table with four columns, (i) test case identification, (ii) input, (iii) expected outcome, (iv) classes covered (from the previous itemization).

A subsequent section (§X.4.D) gives an example of how to build these tables. The example covers unit black box and glass box tests.

The acceptance and installation tests are to be documented separately. Because the objectives of these tests are greater in scope but less exhaustive, test specifications stress comprehensiveness rather than detail. For example, some integration tests – those depending upon the system environment – must be repeated. The acceptance plan should not repeat the details of those tests but should identify those system dependencies and thus the tests to be repeated. In another example, it is common in commercial settings (thankfully, less common in this class) to require a concurrent operation test, where the new and old systems are run in parallel and the results of the new system are validated against the old. In this case, there is no possibility to specify details in advance.

6. These tables are clearly and concisely described in Chapter 7 of Mynatt[23]

Development tools themselves must be tested, especially in an industrial setting where consequences of system failure are high. For example, the compiler used to develop code for a medical device controller should be validated. The rationale is that a single error in a tool can induce numerous flaws in the final product. Tool testing can be done by an external source – in our example, a particular compiler might be purchased because it was already validated. Tool validation corresponds to component testing in manufacturing QA, with the advantage that a tool need be validated only once however many times it is used.

Moreover, tool validation may be performed by a national standards body, a separate contractor, or some other external organization. For Information Systems, most tools (*e.g.* application generators) are widely used, receiving a kind of marketplace validation. However, any tools developed by your team to support the project should be thoroughly tested.

An outline for an example Test Plan, adopted from Pressman [25, §19.3.4], looks like:

1. Scope of Testing
2. Test Plan overview
   - 2.1. Test phases
   - 2.2. Schedule
   - 2.3. Overhead software
   - 2.4. Environment and resources
3. Test Procedure
   - 3.1. For each test phase α, include:
     - 3.1.1. Description of test phase α
       - A. Order of integration, where relevant
       - B. Purpose and modules to be tested
       - C. Special tools or techniques
       - D. Test infrastructure software
       - E. For each test case β:
         - a. Methodology for test case β
         - b. Data for test case β
     - 3.1.2. Expected results for test phase α, for each case β as in (1.v).
4. References
5. Appendices

B. The Test Log

The ultimate goal of test execution is to produce complete and accurate test results, not merely to "get through" the tests. The recording of test results, an important part of this process.

The information collected during the actual execution of the tests is known as the *Test Log*. The Test Log should conform to the Test Plan. The two together provide the documentation for the Testing milestone. While the Test Plan specifies what is to be done, the Test Log is evidence that it was done (hopefully successfully). The Test Log

is the collection of results from the tests, along with what tests were performed, when they were performed, the results obtained from them, and whether or not these results conformed to the expected results. Note that all test results are to be included in the Log – unsuccessful as well as successful. The Test Log therefore supports debugging as well as verification/QA.

4. Development and Evaluation of Test Instances

This section deals with test design and creation. Developing tests requires a thorough knowledge of the requirements, the platform on which the test is to be performed, and sometimes the actual code for the software. Furthermore, knowledge of the theory as well as the practice is required. Once the tests are defined, actually executing of the tests, it is necessary to see how useful the tests will be for actually verifying the project.

Examples in this section illustrate tests which are applicable to most Information Systems projects, along with some discussion of the general principles illustrated. They based on a short 'toy' program[7]. The benefits of using a toy program are conciseness and comprehensibility. The serious disadvantage is lack of motivation, in that the testing techniques we illustrate are only really valuable with programs much larger than our toy.

The specification for the code is:

> The procedure PrintMean has two inputs, an array of integers and an integer whose value is the largest valid subscript for the array. PrintMean should compute the mean of the integers in the array with values between 0 and 100, inclusive, and output the mean truncated to the next lowest integer.

Not only is the example program short, the test instances themselves are small. In fact, for the various glass box tests, they are *minimal* in the sense that the values cover the related condition (path, data flow, *etc.*) but do no more.

A. Test Infrastructure

Just as with the testing of physical objects[8], the testing of software requires the development of special facilities. These facilities are classified according to their module-hierarchy relationship to the tested component. Infrastructure above the tested component is called *scaffolding* (or *test harness* or *driver program*) while infrastructure below are called *stubs*. Figure X.3 illustrates this relationship. Unit testing of (of non-trivial systems) requires both scaffolds and stubs. During integration, stubs are used in top-down testing while scaffolds are necessary for bottom-up testing.

- *Scaffolding* is software written expressly to subject other software to testing. Typically there is a single scaffold for the module being tested. A scaffold is simply a module that calls a procedure being tested, supplying it with values, logging the returned result, and sometimes actually verifying the result (as in our discussion of Square Root above). The scaffold obviously should call the module being tested, logging all of the module's

7. The example is adapted from one given in Mynatt [23]

8. When *Consumer Reports* tests shoes, for example, they build machines which simulate the wear of thousands of miles of walking.

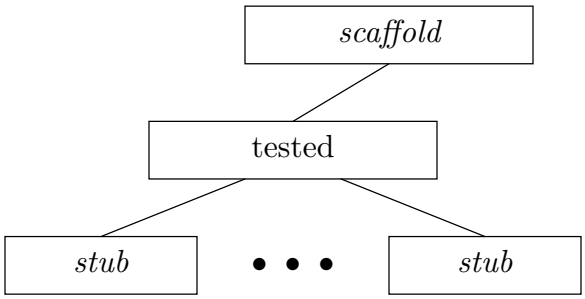


Figure x.3: Test Infrastructure

input and output. If the module is a true function (in the mathematical sense), the logging is relatively straight-forward. If the module has internal state or uses global data structures, the log must record these as best it can.

A scaffold may merely supply a value or values or it may pick one of a number of operations. The values input by the scaffold can come from many sources; a file of test data, a human operator, or random generation are the most common.

- *Stubs* are used to replace procedures called from a high level module. They are sometimes used because the actual procedures have not yet been written and sometimes used because the real procedures do not allow the tester sufficient control. So stubs are, in some sense, "dummy" procedures providing restricted functionality. In virtually every case, the stub should write a message to a log file, indicating that it had been called and displaying its actual parameters and whatever else of the environment is available and relevant.

  If the stub is a function, there are a variety of things that could be done to determine the return value. Consider, for example, a function `price` that takes as input the name of an object and returns the object's price. The simplest implementation of `price` just returns a plausible value, say $10.00. A more elaborate stub would have an array that contains object–amount pairs for a small number of objects; expanded versions of this technique are often used to stub database activities. Another possible implementation of `price` would display the object's name to the screen and ask a human operator to supply a value.

  Stubs are generally not useful for procedures that do substantial transformations, such as data structure operations. That is, it is easier just to implement, for example, the stack operations `push` and `pop` than it is to stub them. More significantly, it is better to treat `push` and `pop` as part of an abstract data type package that is tested and even formally verified as a unit, in bottom-up fashion.

B. Test Implementation/Execution

This section discusses various types of test instances and how they are used; methodologies for constructing them are described in subsequent sections. Remember that the most important part of testing execution is a thorough and accurate logging of test results,

as discussed above.

- *Sample relation instances* are created by populating the tables of the information system with plausible tuples. Almost always these come from the specifications and generally (but not solely) pertain to integration testing and beyond. The best way to develop these tuples is to look at the ER model and hypothesize instances. However, it is important to begin with the ER model and not just the relational schema because it is too easy to overlook relationships when looking only at tables – much of the testing against the sample data instances will involve querying or manipulating relationships.

  It is usually a good idea to keep master copies of sample data and restore the relevant tables before each test session (or more frequently, as needed). This insures that testing is repeatable. To see the difficulty, assume that one test operation is "insert 'Jones' into the L_Client relation." If L_Client were not reinitialized, then this test would have different behavior from the first to the second time, since the first would succeed while the second would produce a duplicate-entry error.

- *Test data files* are values used to run test programs. They are different from the data instances described above because the data instances are static and are the initial values of the test suites. Test data, on the other hand, correspond to operations, including operations that manipulate files and relations. The entry corresponding to the example in the preceding paragraph is

  ```
  INS; L_Client; name="Jones, Robert",...
  ```

  The use of test data files is often overseen by driver programs (which also implement the test oracles).

  A major benefit of using test data files and drivers is that this approach means test are repeatable (see 'Regression testing' in §X.6).

- A *script* is the instructions a human operator follows to exercise a test suite. A script should be detailed enough that the operator does not need to make any substantial choices while exercising the tests. The degree to which the test operator can automatically pick test values differs widely with the circumstances. In case the test field is known to be a numeric month code, the instruction "enter value out of range" is adequate. In other cases, the more explicit "enter value outside of range 1 to 12" is necessary. If complete repeatability is desired, an explicit instruction such as "enter -1" should be used.

  A good script execution methodology is to convert the script into a checklist, marking each test as it is performed and noting instances that do not conform to expectations. This checklist should include expected results (space permitting) and a place to note results that deviate from the expected.

C. Black Box Test Definition

As discussed above, black box tests are entirely base on specifications. Accordingly, we now give some suggestions for creating test instances based on specifications.

- *Equivalence partitioning* views the specifications as dividing the input space into *classes* and picks test input values that are representative of the various classes. The first division is always into *valid* and *invalid* classes, and the valid class may be further subdivided. For example, in a medical system, the value 1800 for year-of-birth is invalid, 1994 indicates the class "child", and 1930 indicates "elderly." A class is typically a *set* or *range* of values.

  Recommendations for using equivalence partitioning in test development include:

  ◇ ask "what causes [or changes] the condition?"

  ◇ design tests to cover classes independently.

  ◇ if a class is a set, try to express its membership condition formally.

  ◇ if a class is a numeric range, use boundary value analysis, covered below.

  ◇ if the class is defined by complex conditions (*e.g.* regular expressions), treat the class as an intersection or union of simple cases.

- *Boundary value analysis*[9] looks particularly at the limits of range classes. Recommendations in this case include:

  ◇ test each endpoint separately, to the degree possible.

  ◇ test just above and just below range endpoints.

  ◇ test on the endpoints.

  ◇ test at some distance from the endpoints.

  ◇ a number-of-items specification is a range definition. For example, if the specification reads "the program accepts up to 100 input values," test cases should be developed with 99, 100, and 101 inputs.

  ◇ attempt to generate *output* at or near boundaries too.

  ◇ for multidimensional ranges, attempt an orthogonal characterization of the class and test accordingly.

  These recommendations give rise to the following test instances. The input "22,23,24;2" means that the first parameter of PrintMean contains the values 22, 23, and 24 and the second parameter is 2.

  Black Box Test

class identifier	condition(s)
a	middle of acceptable range for array values
b	lower boundary of acceptable range for array values
c	upper boundary of acceptable range for array values

9. This term is borrowed from numerical analysis, where it has a totally different meaning. If this overloading causes difficulty, "decision point analysis" is also used.

test case identifier	input	expected output	classes
B.1	21,22,23;2	22	a
B.2	-1,0,1,2;3	1	b
B.3	98,99,100,101;4	99	c

- *Menus* and other command-choice mechanisms have the following heuristics for for test development:
  - ◇ try both blatant and subtle incorrect forms.
  - ◇ try input that is correct except for order.
  - ◇ try partial correct commands.
  - ◇ try null entries.
  - ◇ try over-constrained commands or too much qualifying data.
  - ◇ try interchanged, mismatched, and wrong delimiters.
  - ◇ try interrupts, reset, other unusual timing problems.

D. Glass Box Test Definition

Glass box methodology points out where in the program structure test should arise but does not indicate which input values to use. For this reason, glass box methodology is sometimes used to determine whether existing tests are adequate.

All discussion in this subsection is in terms of our toy example. A C program implementing the specification (and a bit more, as the program designer recognized that the mean is undefined if there were no valid inputs) follows. The numbers in comments label statements or statement blocks, known as control points in the Nassi-Shneiderman methodology. The code is followed by the Nassi-Shneiderman chart, by the control flow diagram, and by tables defining some of the glass box and black box tests for the code.

```
void PrintMean ( A, AMaxIndex )
 int A[], AMaxIndex ;
{
    int NumOfScores, NumInRange, Mean, i, Sum;

    Sum = 0; NumInRange = 0; i = 0;                          /* 1 */

    while ( i < AMaxIndex ) {                                /* 2 */
        if (0 <= A[i] && A[i] <= 100) {                      /* 3 */
            Sum = Sum + A[i];                                /* 4 */
            NumInRange = NumInRange + 1;
        }
        i = i + 1;                                           /* 5 */
    }
```

```
    if( NumInRange > 0 ) {                              /* 6 */
        Mean = Sum / NumInRange;                        /* 7 */
        printf("The mean is %d\n", Mean);
    }
    else printf("No values were in range.\n");          /* 8 */
}
```

Figure X.4 shows the *Nassi-Shneiderman chart* for the above code. Nassi-Shneidermann charts provide a standard dissection of program structure, which is useful in constructing the control flow diagram as well. Nassi-Shneiderman charts are composed of rectangles subdivided vertically to indicate successive blocks and horizontally to indicate alternatives. The whole procedure is represented by the outermost rectangle. Each rectangle is a *control point*. Decision rectangles (corresponding to `if` statements) have diagonal lines at the end labeling the alternatives. Both the true and false sides of the decisions should be assigned control points, even though a particular side might be void. In our example, this implies a control point corresponding to the false exit from statement 3, which we label "4*" since it is paired with control point 4.

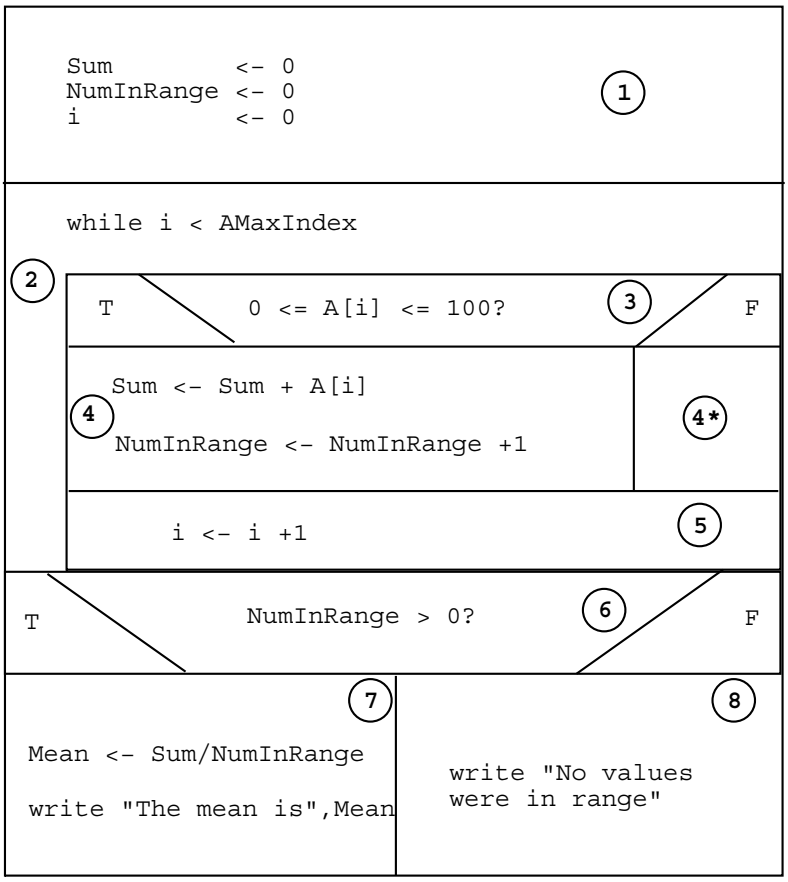


Figure X.4: Nassi-Shneiderman Chart for Example

The criterion for test coverage for Nassi-Shneidermann charts is that the test must (i)

exercise each control point at least once and (ii) execute and omit loops, where possible. The condition `NumInRange > 0` in control point 6 is a good illustration of the fact that the methodology does not directly provide test instance values; we must understand the code and understand that this condition is made false only when the array has no values in range. The minimal test instance suggested by the chart in Figure X.4 is:

Glass Box Tests Developed from Nassi-Shneiderman Chart

test case identifier	input	expected output	classes
NS.1	*empty*;-1	"no values were in range"	1,2,6,8
NS.2	1,-1;1	1	1,2,3,4,4*,5,6,7

Figure X.5 gives a *Control Flow Diagram* for the above example – another way of representing program structure. In this diagram, unconditional control points are represented using circles and decision points (control points with branches) using diamonds. There is no way of representing the special structures for while/for loops in control flow diagrams; these must be converted to if statements and branches.

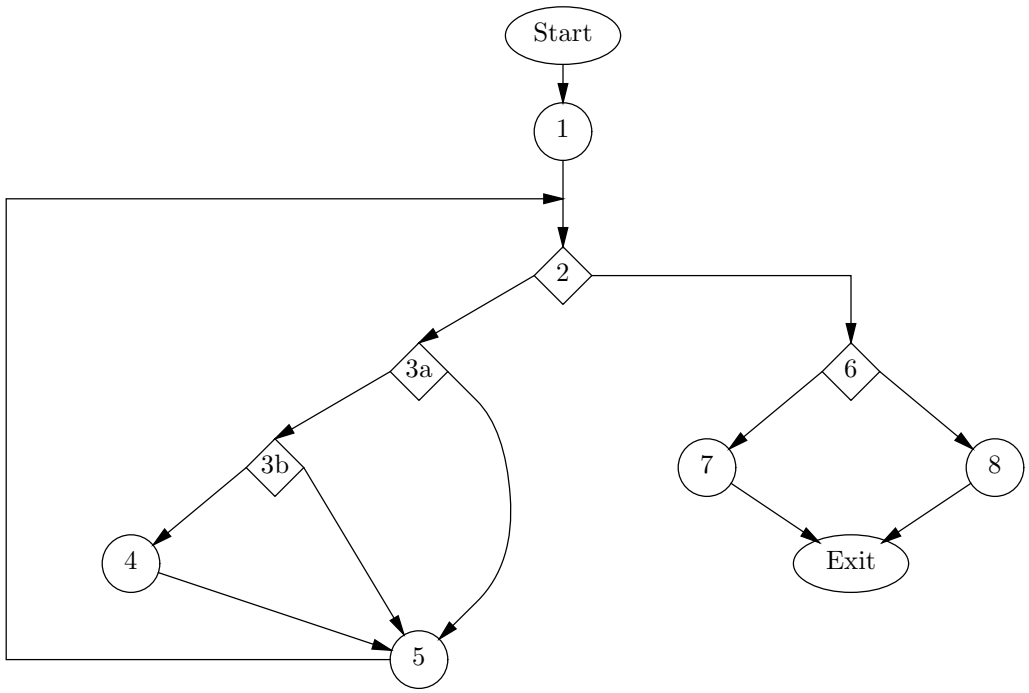


Figure x.5: Control Flow Graph for Example

These two diagraming methodologies suggest different aspects of designing test instances. The Nassi-Shneiderman methodology, with an explicit notation for control loop structures, suggests that one instance for our example be an empty array. This is a consequence of the loop testing rule requiring one test instance to have no iterations of the loop. Tests based on control flow diagrams, defined in terms of the presence of certain nodes on a path but not on the absence of nodes, would not naturally suggest an empty array.

On the other hand, the control flow diagram methodology lays out program structure in a more visually distinct way. The standard control flow methodology separates each individual test in a conditional into a distinct node. In our example, `0 <= A[i]` and `A[i] <= 100` are distinct nodes; we label the first "3a" and the second "3b". Furthermore, there is a distinction between "node coverage" and "edge coverage." As the terminology indicates, a test case with node coverage includes control passing through every node at least once. Edge coverage, as expected, traverses each edge at least once. In the first of the following two test instance tables, control passes through 3b does not follow the edge 3→5. The second includes an execution following this edge. Note that the tables omit itemizing under "classes" those nodes or edges that must always be covered.

Glass Box Tests using Node Coverage Criteria

test case identifier	input	expected output	classes
NC.1	1;0	1	3a,3b,4,5,7
NC.2	-1;0	"No values were in range"	3a,5,8

Glass Box Tests using Edge Coverage Criteria

test case identifier	input	expected output	classes
EC.1	1;0	1	3a→3b,3b→4,6→7
EC.2	-1,101;1	"No values were in range"	3a→5,3a→3b,3b→5,6→8

A test coverage table for our example follows. It concerns only glass box edge cover. In this table, "EC.1.2" means the second array value of test instance EC.1.

Test Coverage Analysis Control Flow Graph Edges			
edge			instances covering
3	⟶	3a	EC.*all*
2	⟶	6	EC.*all*
3a	⟶	3b	EC.1.1, EC.2.2
3a	⟶	5	EC.2.1
3b	⟶	4	EC.1.1
3b	⟶	5	EC.2.2
6	⟶	7	EC.1
6	⟶	8	EC.2

A final graph-based test definition criterion is *path coverage*. A path is any connected sequence of edges in the control flow graph, all edges going forward of course. Any graph with a cycle (a directed, connected sequence of edges that begin and end at the same node) has an infinite number of paths, since adding a traversal to a path creates a new path. Note that the program itself may or may not have an infinite number of execution paths; we are thinking only of the control flow graph. Thus it is possible to test all paths in only the most trivial programs. Therefore path coverage depends only upon the *irreducible paths*. A path is reducible if it contains a cycle and deleting that cycle would reduce the total

set of edges included in the path. From the perspective of the program, this definition means that a repeated traversal through a loop can be omitted if it repeats the same path inside as well. In our example, a path containing only the two cycles 2→3a,3a→5,5→2 and 2→3a,3a→3b,3b→5,5→2 is still irreducible because one cycle contains edge 3a→5 and the other 3b→5. The notion of path coverage therefore requires that each irreducible path be covered by some instance of the test. Paths that are reorderings of each other are considered equivalent and depending upon the circumstances, are typically both considered covered by a test which exercises some member of the equivalence class.

While the set of irreducible paths is finite, the set is likely to be exponentially larger than the graph size. In our example, there are 8 (*i.e.* 2^3) irreducible, non-equivalent paths through the loop beginning with control point 2.

Another kind of structure that can be derived from a program module is the *data flow graph*. These share some characteristics, as well as most of their name, with the DFD's of Chapter VII, but data flow graphs are much lower level that DFD's. Moreover, data flow graphs follow from the programs, while DFD's are precursors to the programs. A data flow graph uses the same nodes as a control flow graph, although decision points are not distinguished. There is an arc from each node $\mathcal{A}$ to $\mathcal{B}$ if, during some possible execution (possible in a graph theoretic sense, on the control flow graph), a value is assigned to a variable at $\mathcal{A}$ and that value is used at $\mathcal{B}$.

Figure X.6 illustrates the data flow graph derived from our example. Edges are labeled by the variables that flow along them; sometimes the same label applies to two nearby edges. Isolated nodes have been omitted. Note that 1→7 is not achievable during any actual computation because the `T` branch through conditional 6 can be followed only if NumInRange has been incremented at least once in block 4. But 1→7 is still a possibility in the graph[10]

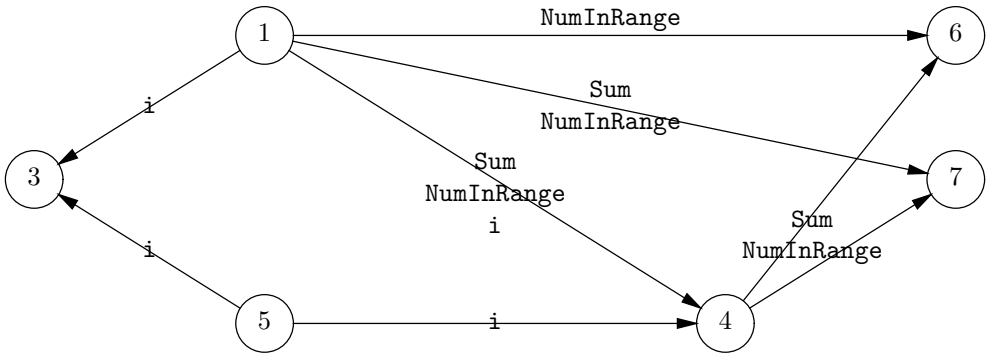


Figure x.6: Data Flow Graph

Coverage criteria are both a method of designing tests and a method of assessing test adequacy. To the latter end, the Test Plan should document how test instances provide the

10. The undecidability of many program properties, deciding whether or not a path is actually ever taken among those, forces us to be inclusive in constructing edges.

relevant cover. This corresponds to the rationale included in the design documents. For glass box testing, the Test Plan should contain two tables for every module, one describing the control points, possible outcomes, and the test cases that cover the control point, and the other actually showing the test cases with the test values, expected outcome, and space for the actual outcome.

5. Roles during Testing

Team members, supervisors, and the quality assurance monitors each have specific tasks in testing and assuring the quality of the software product developed by the project. This section will describe the roles played by these people in the testing phase, especially with respect to the Information Systems type of projects.

- *Team members* have primary responsibility for the different stages of testing. Every team member should design the glass box tests for the modules that s/he writes. Team members should be designated to write black box unit and integration tests for modules written by another programmer. The execution of integration tests can be divided among several team members, but an alternative is to have just one person serve as *integration tester*.

  Two other team roles have special significance during testing:

  ◇ *Secretary*: Collecting the Test Log is one of the secretary's most important logging responsibilities. It is possible for a team to appoint a separate *test recorder*, but the secretary already knows the discipline for logging.

  ◇ *Librarian*: As modules pass unit testing, they are considered *certified*. Certified modules given to the librarian, who keeps them in a special repository that others can depend upon when exercising their code.

- *Supervisors* of the team are not directly involved in the testing procedure, but they should give guidance in designing the tests, performing the tests, and making sure that the test documents are being consistently prepared. The team may also demonstrate the running programs to the supervisors for his/her comments regarding the user interface, loopholes in the testing, *etc.*

- *Quality Assurance Monitors (QAMs)* are assigned to teams to make sure that both product and process conform to the project's requirements. Supervisors and QAMs both have monitoring responsibilities. Supervisors monitor the process, being concerned with of task allocation and completion. QAMs monitor testing only, being concerned that the Test Plan is adequate, that the tests are executed according to the plan, and that results are recorded properly.

  After the testing phase is completed, QAMs perform, either actively or passively, the acceptance test before the product is installed or before the manuals are finalized. The team is supposed to give at least one full fledged live demonstration of the software to the QAM.

6. Other Important Topics

Other topics to be considered are :

- *Platform factors*: Most of the testing techniques mentioned in this chapter have no relationship whatsoever with the platform on which the tests actually are to be performed. However, for the successful execution of the tests, it needs to be established that the designed tests can be performed given the constraints of the platform. For example, a particular database package might have a constraint on the maximum number of records in a table. Any test that needs more than that maximum number of tuples is bound to fail, although there would not be anything wrong with the code. So, before the tests are actually performed and the inferences made based on the outcomes of the tests, test designers should ascertain that constraints of the platform itself will not cause the tests to fail.

- *Network testing*: Because of the proliferation of Local Area Networks (LANs), many projects now have multiple personal computers connected on a LAN, so that all the machines share common storage and information. This facilitates accessing the same data from multiple workstations, but at the same time brings in the problems of concurrency control and locking. Fortunately modern database packages have locking and concurrency control mechanisms, but making them work correctly is sometimes difficult, often due to configuration problems. In these cases, it is necessary to ascertain correctness of the system under network access. This is termed as Network Testing.

  Network tests must be defined to consider all conditions. But defining these tests may be far easier than actually running them. For example, a test specification may read "test the effect of two users simultaneously saving the same record" (evaluating what is known as a *race condition*). This means true simultaneity, not just close timing. But it is very difficult to deliberately cause truly simultaneous actions – there is always a difference in the delay. Of course the same perversity which makes it difficult to test true simultaneity will insure that the worst timing behavior occurs in real operation.

- *Stress testing*: Testing performed to evaluate system behavior under extreme conditions is known as stress testing, There are many load factors that produce stress, that is, the possibility of poor performance:

  ◇ Size of data sets: Large files or relations might change the performance of a database software system drastically, especially if the access of data involves slow storage devices. With PC application packages, it is fairly common that a nice looking system, with good compliance to functional requirements, fails utterly when the relations are filled with the real volume of data. It is common to prepare for stress testing by random generation of large test data files.

  If a system does exhibit problems when stress tested with large data sets, there may be ways to correct the problem. Different speedup techniques, both hardware based (disk cache, *etc.*) and database based (indexing, *etc.*) can improve performance.

  ◇ Concurrent access load: A large number of processes concurrently accessing tables

introduces another kind of stress on the system. In an environment with a LAN and a PC server, this constraint sometimes turns out to be of big importance, because unlike the mainframes, the personal computers are normally not built for concurrent processes.

Tests for adequate primary memory capacity, recovery after a crash, *etc.* should also be considered for stress testing.

- *Design for testability*: This idea began as a formal topic in VLSI hardware design (at the point it when it became impossible to test a circuit by inserting a probe) but is migrating to software. It stipulates the obvious, that the code is to be designed in such a way that tests on the code can be performed easily. For an immediate example, output statements can be placed in different parts of the code systematically so that the flows of control and values may be traced. Hopefully the software platform has 'compiler switches' that allow automatic deletion of these output statements after the tests are completed and the code is completely validated. On a higher level, the best design for testability is good modularity – when specifying a module, one question that should be asked is "how easy is it to test this module?" Specifying modules by pre- and post-conditions is very helpful in this regard because a print trace can easily record those conditions.

- *Forms testing*: Forms that are generated by certain user-interface tools have two different types of components which require testing. First is the traditional imperative code which is bound to events triggered from the form. Such code is tested by traditional glass- and black-box techniques, with the only difficulty being the fragmentation of the code into a myriad of snippets. Second is the parameter settings associated with the components that populate the form. The testing that is done here is to to make certain that the correct settings are used. For example, when the tool supports requiring that entered data must conform to pattern settings, the form components with such setting should be tested to verify that the correct settings have been used. Such settings are easy to mistype – say "`dd-d-dd`" had been set to validate a month-day-year field, then only single digit dates would be permitted.[11]

  Regression testing for form settings is a bit easier than for code. The assumption is that, unless the form has been intentionally modified, the setting will not change in arbitrary ways. However, tools sometimes will clear settings, in a misguided attempt to be "helpful." Therefore, regression testing should include a few cases to make certain that settings have not disappeared from forms. Note that this minimalist testing works because settings for one form component do not side-effect other components (obviously excepting event settings that invoke code). Such minimalist regression testing does not work for code because it is the interaction that is being retested.

11. This is not a realistic example, since most tools have built-in date handling with preset formats.

XI. Documentation

1. Introduction

Documentation is as essential to a software project as any screen or module. How many times have you had to hunt through a User's Manual for hours trying to find out how to do something with a piece of software? How often have you needed to modify a program you had written, only to spend hours re-analyzing your own undocumented code? A piece of software can be brilliantly designed, but if the user can't figure out how to use it, then it is worthless to them. A piece of software can be brilliantly written, but if another programmer cannot understand and modify it, it will may re-written from scratch because re-writing takes less time than understanding undocumented code. In these respects, a piece of software is only as good as its documentation.

The process of preparing a document follows the typical lifecycle model. The stages may have different names, but they are all there and all important. Specification is called "conceptualization;" design is "organization" or "outlining;" implementation is of course called "writing", and testing is "review" or "editing."

The specifications for documentation have more to do with the readers than with the material itself. Remember that communication involves both a sender and a receiver and that transmitted material is worthless if the receiver cannot process the information. Once the audience has been characterized, the content of the documentation is simply a response to the question "what information do they need and how will they access it."

The outlining of a document defines both its form and its content. Just as with software design, a document outline should be thoroughly reviewed. Just as code is tested, so the manuals should also be tested by a variety of users and programmers.

The rest of this chapter will discuss, in turn, each of these phases of the documentation process. Most of the comments will focus on one component of the documentation, but you should recognize that many suggestions apply more broadly.

2. Document Specifications

A. Users of the documentation

The first issue in designing a document is identifying the target audience. Since documentation in Information Systems will be written by the same team that designed the software, it is relatively easy to produce a document that correctly describes the software. However, because the team is so familiar with the system's requirements and design, it becomes more difficult to write the manuals with a novice in mind.

Issues to examine in tailoring the User's Manual to the users include:

- What is their motivation level?
- What is their skill level?

- How regularly will they be using the software?
- Are they familiar with the problem domain?

For the user's documentation, the first design decision is whether to produce a tutorial or a reference. If the client's staff are novices, are poorly motivated, or are infrequent users, then obviously the User's Manual should be more of a tutorial which guides them through the use of the system. If the staff has experience using computers and understands the previous system, then the User's Manual can be more of a reference. Table 1 indicates some of the major differences between a reference and a tutorial.

Tutorial	**Reference**
Assumes the user knows nothing about the system.	Assumes the user is familiar with the system.
Organized for learning. Starts with simplest concepts first and defines all terms before they are used the first time.	Organized for finding information.
Contains step-by-step instructions on how to perform a sequence of tasks.	Contains step-by-step instructions on how to perform a task.
Explains why it is necessary to perform an action (although not in a lot of detail) and how to perform it.	Just explains how to perform an action and assumes the user knows why it is necessary.

Table 1: Differences/Similarities between a Tutorial and a Reference Manual

Of course the best solution is to provide both a tutorial and a reference. But do not attempt both in a single document, the result will satisfy neither audience.

B. Uses of the documentation

When writing a User's Manual, remember that people don't like to read a manual, especially a thick one. Users typically would rather get on the system and see what they can do, referring to the manual when they get stuck. Therefore the manuals should be written in short chapters, addressing individual tasks, such as "How to Enter Data", "How to Do a Query" or "How to Print a Report." These chapters should generally be self-contained.

An alternative, known as minimal manual style, focuses on the broad skills needed to perform a task and assumes that the reader can make inferences, such as doing data entry in one part of the program is similar to the data entry they've already learned to do in another part of the program.

Most importantly, a single document cannot adequately serve all needs. It is often much easier to write several documents - small, targeted ones - than one massive manual.

C. Navigation through the documentation

Software documentation is often plagued with the problem that "you can't ask the question until you know the answer". For example, if you want to find out how to export ASCII files using Paradox, don't look in the User's Manual's index under "import/export ASCII files" or "read/write ASCII files", but "text streams". It took two frustrating days for a programmer to find this information. There are several ways of guiding the user to the information. These should include:

- Index. This should list several synonyms for an entry. For example, MS DOS users are used to "deleting" files, while Unix users "remove" files. Both keywords should be listed in in the index. In the above Paradox example, the index should say "import/export ASCII files, see text streams" and "read/write ASCII files, see text streams", in addition to just having an entry "text streams".
- Table of contents. This includes chapter, section, and subsection headings and the pages on which these can be found.
- Links in the text. The text should guide the user to related subject areas. For example, to find commands related to the Unix "mv" command (which moves files), type "man mv" and look at the bottom of the on-line manual page. There it says:

  SEE ALSO: cp(1), ln(1V), chmod(2V), rename(2V)

  pointing you to the related functions which copy, link, change access permissions, and rename files. Users are increasingly familiar with following links because of the ubiquitous web and on-line help.

D. Format Design

The general way in which your document is presented and the parameters which effect page layout must be understood and specified.

A document's format is its user interface. Inconsistencies in document layout and structure are every bit as distracting as inconsistencies in screens or menu traversal. Therefore, establish format standards before you begin to write.

The document preparation tools you are using may enable you to defer decisions about formating parameters, but the general nature of the formating must be decided before writing begins (see section II.3.B). As with programming proper, make sure that at least one member of your team knows the document preparation tools thoroughly. If the tools have a macro processor, define macros for all the major units of the document or use an "off the shelf" package that does so. If a macro processor is unavailable, lay out templates for chapters and sections.

3. Outline: Structure and Content

Most of the content details for a manual are of course system-specific, but the general topics are more or less the same. We first discuss one topic which spans all types of documentation and then discuss three different pieces of the documentation suite: on-line user's documentation, the User's Manual, and programmer's documentation.

A. The Running Example

It is usually a good idea to have a single example, or small set of examples, shared throughout all the documentation. The example(s) must therefore be designed in advance, especially in a situation where the writing is subdivided into smaller tasks.

Your data model has been important in all the other project stages and remains so here. Example development should begin with a very well-thought-out, varied data instances. Then develop some natural scenarios for these instances.

The examples, although hypothetical, should be well-motivated. However, trying to match the client's data too closely can raise distracting question about accuracy ("But [attorney] Smith doesn't work on liability cases ..."). A certain amount of whimsey can enliven the documents, but too much can create a bad impression, depending upon the clients. Data instances for Dracula and Frankenstein are fine for undergraduates but not for undertakers.

B. On-line User's Documentation

User's documentation should include on-line material and not just paper documentation. On-line documentation could include:

- The *README file* includes general advice for navigation through the system and through the documentation. It is also the place for last-minute information that didn't make it into the User's Manual. The fact that this *ad hoc* notion is so ubiquitous is a testimonial to its effectiveness.
- *Help* is the active link between a software system and its documentation. Help text should be brief and operational but with links to further information.
- *Error messages* are often the place where a user's quest for additional information begins – against the user's will and with substantial stress. Therefore error messages should be polite.

  Error messages should also be explicit and directive. For example, if an input data value is invalid, it is not sufficient to say "data out of range" because that gives no indication of the correct value. Instead an error message should be "a number between 0-1000 is expected." In cases more complex than a simple range check, the error message should direct the user to sources of further explanation.
- A *menu hierarchy* guides a user navigating through the menu options.
- The *On-line Manuals* are copies of the User's Manual and possibly Programmer's Manual, formatted to be read on the screen. Hopefully your document preparation tools provide for on-line as well as paper versions (see section II.3.B). It is valuable for the documentation tool to provide on-line searching for keywords. There are several ways in which tools can assist. The text file for a manual may be directly readable, as with WordPerfect or MS Word. The output may be retargetable to paper or display, as with Unix's *man* environment. Or one tool may be able to adapt source from another product into its representation, as with emacs' "texinfo."

- A *sample application* gives users examples of how to develop and use the various system components. This should of course be coordinated with the running example. Many PC database applications now do a reasonable job providing such sample. In most cases, Information Systems projects do not have a need for this form of documentation.

Keep in mind that the User's Manual may become lost or misplaced and that the on-line documentation will, in this event, substitute for it.

C. User's Manual

The User's Manual is the most difficult documentation to write because of the different types of users, as discussed above. Irrespective of the way in which it is presented, the User's Manual for Information Systems should contain:

General

- *Table of contents*, which includes chapter, section, and subsection headings and the pages on which these can be found.
- *Introduction.* Remember that this introduces both the topic and the manual's approach to that topic.
- *Quickstart summary*, which describes how to perform the basic operations and enables a first-time user to be productive with the system without getting bogged down in a lot of details.
- A *user-level data model*, including ER diagrams (typically a simplified version) and an explanation of those diagrams, along with points to the forms or operations that maintain the information. It is usually clear how entities are maintained (especially for user interfaces following the suggestions of section IX.4.A, since each major entity will have an associated form), but relationships are often only maintained with one of the associated entities and that connection should be explicitly noted. In our example, the case_of relationship could be maintained in conjunction with Case, Client or both; the documentation should tell the user which.
- *Overview of operations*, which explains what the system can and can't do. If there are items of functionality which one would naturally expect from the system but which were omitted for some reason, they should be mentioned. It is frustrating trying to get software to perform some particular action, only to discover that the system does not implement that action. If the standard mode of operation is to first select an entity and then to select operations on that entity (again, as discussed in section IX.4.A), that should be clearly indicated. This section might also show the general form of menus and other screens.

Operations

The following discussion applies to operations in your system are not part of a standard user interface provided by the platform. For operations provided by the platform, give an example or brief discussion and refer the reader to the appropriate source; this holds

both for complex operations such as query formulation and for simple operations such as built-in drop-down lists.

If an operation requires several steps, be sure to tell the user about all of them. Even though a step may seem blatantly obvious to you, a user who is uncertain will follow your directions exactly and thus not insert that step. First explain the sequence in general and then explain how to each step. Furthermore, there may be more than one way to do a step – make sure to keep these clear. For example, using a drop-down list has two steps, opening the list and selecting an item; each step may be performed with either the keyboard or the mouse.[1]

If an operation has conditions, clearly identify those conditions and explain how they are evaluated. If it is the user's choice, be explicit about the consequences of the various alternatives. Also note that "else" is computerese; the proper word for most people is "otherwise".

The following are specific operations which all documentation certainly should include; this is not an exhaustive list!

- *Navigation*, explaining how to move from form to form and command to command. This might be presented with the menu/form hierarchy.
- *Form control*. Often closely related to navigation, this includes operations that change the state or "mode" of a form (from `View` to `Edit`, for example), affect the contents or presentation of subforms, or do global operations on the form's data (`Save`, for example).
- *How to enter data*. Show example screens using realistic data and give descriptions of how to do data entry. Explain how to correct mis-typed data. Can the user correct an entry while still in the "Add Data" mode or is it necessary to go to the "Modify Data" function?
- *How to delete and modify data*. Show example screens using realistic data and give descriptions of how to delete and modify data entered in the database.
- *How to do queries*. Show examples of how to do queries, indicating operations using highlighted fields, pull-down or pop-up menus, and other available methods. Give sample entries for data fields. A text description of how to perform the query should go along with the example screen. Also describe expected errors and how to correct them.
- *How to print reports*. Describe how to print reports, with realistic examples.

<u>System Maintenance</u>

- *Installation*. Describe how to install the new system and how to convert from the old system to the new system.

1. As noted in the previous paragraph, this particular operation would typically be covered in the platform documentation and not yours.

- *Archiving databases.* Describe how to archive and restore databases. Also encourage the client to have their database backed up on a regular, frequent schedule. Describe any special end-of-year activities, archiving annual data and reinitializing tables.
- *Other maintenance.* Describe other system maintenance tasks, such as adding new users to the system.
- *Trouble Shooting.*

  Accessing Documentation Information

- *Glossary.* A glossary should be included if the system uses words which have a specialized meaning or a meaning different from everyday usage.
- *Reference Card.* This is a one-page reference describing hot keys, help keys, and error messages.
- *Index.* If your User's Manual does not include an index, the table of contents should be detailed enough so that the user can quickly and easily find the information he or she needs without having to search through the entire manual. When building the index for a tutorial, it is especially important to make sure that all terms are defined and that the concepts necessary to understand them are discussed.
- *Cross references.* See section XI.2.C.

D. Programmers' Documentation

The first goal here is to enable the reader to find his or her way around a large (and disjointed) piece of software.

In writing the Programmer's Manual, assume that the programmer knows the database language used by the system. Even if they do not, there are other manuals available for teaching them the database language. But you probably know which of those manuals are better; recommend good sources for learning the database systems (including non-traditional ones such as news groups).

A good general approach to writing the Programmer's Manual is to put yourself in the place of a programmer who has never seen the system before and is called in to fix a bug or make a modification to the software. Consider what would make this job easier, avoiding the need to wade through the source code to understand the program.

The Programmer's Manual should contain at least:

- *Table of contents*, including chapter headings, sections and subsections and the pages on which they can be found.
- An *introduction* which gives an explanation of the system and the general approach taken in designing it.
- *Entity-Relationship model* with diagrams and explanation, The ER model has been featured in every document since the Requirements Specification. It is just as essential to understanding the completed system as it was to understanding the design task. The

ER model enables a programmer who has never seen the system before to understand its overall information structure.

- *Relational schema*, describing how the relations implement the ER model and giving any additional "helper" tables. The "design rationale" that describes how and why the relational design varies from the one generated from the ER model by the "turn-the-crank" procedure.

- *Module hierarchy*, with hierarchy diagram(s) and descriptions of modules. The hierarchy will enable the programmer to easily understand the overall structure of the code, how the modules interface with each other, and how they relate to the design of the system.

- *Source code* description for each module. Each module should have a header which describes the module, its inputs and outputs, and should have in-line comments of the code explaining what a section or line of code is doing. Since headers and comments were included while writing the code, this section has already been written. A sample header is shown below.

```
************************************************************

            DATA REPORTALL MODULE

  Tables Accessed....Attorney,works\_on,Case
  Parameters.........Records retrieved from ViewDataQuery
  Global variables...
  Local variables....
  Input..............
  Output.............Screen display of information requested
  Tables Altered.....
  Side effects.......
  Called by..........ViewDataQuery module
  Modules called.....SelectAttorney

  Description
    The Data Reportall module will enable the user to view
    Case data for one Attorney, including hours worked
    in table format, sorted by CId and date and
    Save/Print that data.

************************************************************
```

Whether actual source code is included in the paper document is an open issue. Three inches of code listing with no organization is only an environmental crime. It is most important that the code be organized and that sections be easy to find and to relate to one another.

- *Implementation Map.* Describe how modules, forms, tables, and other artifacts are represented within the file system of your client's computer. This could be, for example, a list of all the modules with the names of the files containing the source and object code. This is very platform-specific, but many platforms have valuable tools to assist in the creation of such a map.
- *Likely Modifications.* While it is impossible to anticipate all the changes that might be made to a system, likely ones can be anticipated. A brief description of how to make these changes would be helpful. In some cases, specific changes are known ahead of time. For example, the client may desire functionality that you didn't have time to implement. Documentation for the database tool will probably describe such issues as adding new fields or tables, so handle those topics by referring to that documentation.
- *Index.* The index should include variable and module names and the page numbers on which they can be found. Synonyms and industry-standard keywords should be used in the index. For example, rather than just an entry under "text streams", like the Paradox manual has, have ones which say "import/export ASCII files, see text streams" and "read/write ASCII text files, see text streams". The index differs from the implementation map in that the index organizes by concept, while the may organizes syntactic objects.

4. Writing the Documentation

A. Writing Style

No one (but the sorriest geeks) reads computer manuals for the fun of it. We read manuals because we need specific information. Therefore manuals should be written to help the reader focus on the relevant information. The following list extends the general suggestions of section II with suggestions specific to manuals.

- Address the reader directly. This means using second person.
- Use active rather than passive voice. This tends to produce shorter sentences as an added benefit.

  Instead of: The computer will show you the next screen when the F1 key is pressed.

  Or: You can see the next screen by pressing the F1 key.

  Say: To see the next screen, press F1.
- Write task-oriented sentences which focus on the steps to be done by the user.

  Instead of: The computer must have a backup disk in Drive 2 to make backups.

  Or: Put a backup disk in Drive 2 before making backups.

  Say: To save a backup copy of a document, first put a backup disk into Drive 2.
- *When results may be catastrophic, describe the result first, followed by the action.*

  Instead of: Press F1 to delete all records from the database.

Say: To delete all records from the database, press F1.

This way a user won't read the first part of the sentence, press F1, and face the horrible discovery that the database is gone.

- Use a standard, consistent method for giving instructions.
- Use short sections with titles, rather than long chapters. Chapter titles should address a task to be performed, such as "How to Enter Data", "How to Do a Query" and "How to Print a Report". Use different font styles to help emphasize sections and make them easy to find.
- Use pictures, diagrams, and tables when possible. For example, when describing how to do data entry or make a query, also show a picture of the screen with sample data.

  Documentation is often the reader's only source of information on the particular subject or product, hence it must be thorough and complete.

  Documentation is often used to answer a specific question, hence it should facilitate finding a specific piece of information ("navigation").

  Documentation must bridge from general specifications to particulars of implementation and operation, hence it must make abstract concepts concrete and make concrete facts fit generalized concepts.

  Documentation can be presented in many forms: online via html, MS Help files, or just plain text and on paper as reference manuals, tutorials, quick reference guides, *etc.* It is important to choose the right medium and even more important to write to fit the medium.

B. Technical Hints

See the subsection of the same name in the general discussion on writing, section II.3.B.

- Target documentation for on-line access as well as a paper copy. To see comparable tools, learn how the Unix "man" pages or emacs' "texinfo" work.

5. Documentation Quality Assurance

Give the User's Manual to one of your users to see if they can use the system based upon what the manual says. There may be some sections of the manual which could be written more clearly. There may be vital facts omitted, facts that you know well but the user doesn't know at all.

XII. References

1 Scott Adams. Dilbert. cartoon. © United Feature Syndicate, Inc., also at http://www.unitedmedia.com/comics/dilbert.

2 Victor R. Basili and Richard W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12):1278–1296, Dec. 1987.

3 Carlo Batini, Stefano Ceri, and Shamkant B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings, 1992.

4 Bruce I. Blum. *Software Engineering : A Holistic View*. Oxford University Press, 1992.

5 Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

6 Barry W. Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford J. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifer, and Bert Steece. *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000.

7 Grady Booch, James Rumbaugh, and Ivar Jacobsen. *The Unified Modeling Language User Guide*. Addison-Wesley Publishing Company, 1999.

8 Michael L. Brodie and Joachim W. Schmidt. Final report of the ansi/x3/sparc dbs-sg relational database task group. *SIGMOD Record*, 12(4):i–62, 1982.

9 Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, pages 10–19, April 1987.

10 Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1995 (first edition, 1975). excerpted in *Datamation*, Dec. 1974.

11 Lewis Carroll (Charles L. Dodgson). *Through the Looking Glass (And What Alice Found There)*. 1871.

12 Weidong Chen, Michael Kifer, and David Scott Warren. HILOG: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

13 L. A. Clarke, J. Hassell, and D. J. Richardson. A close look at domain testing. *IEEE Transactions on Software Engineering*, 8(4):380–390, July 1982.

14 William S. Davis. *Tools and Techniques for Structured Systems Analysis and Design*. Addison-Wesley, 1983. Although somewhat old by computer science stanards, this little book does a nice job giving well targetted discussions of specific tools and techniques.

15 Dublin core metadata initiative.

16 Per O. Flaaten, Donald J. McCubbrey, P. Declan O'Riordan, and Keith Burgess. *Foundations of Business Systems*. Dryden, 1989. Andersen Counsulting Arthur Andersen & Co.

17 P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, Oct. 1988.

18 W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

19 IEEE Standard 729-1983. *IEEE Standard Glossary of Software Engineering Terms*. New York: IEEE Society Press, 1983.

20 J. King and E. Schrems. Cost benefit analysis in information systems development and operation. *ACM Computing Surveys*, 10(1):19–34, March 1978.

21 W. Litwin, A. Abdellatif, A. Zeroual, and B. Nicolas. MSQL: A multidatabase language. *Information Sciences*, 49:59–101, 1989.

22 J. A. McCall, P. G. Richards, and G. F. Walters. Factors in software quality. Technical Report 77CIS02, Rome Air Development Center, 1977.

23 Barbee T. Mynatt. *Software Engineering with Student Project Guidance*. Prentice Hall, 1990.

24 Shari Lawrence Pfleeger. *Software Engineering : The Production of Quality Software*. New York : McMillan, 1987.

25 Roger S. Pressman. *Software Engineering : A Practitioner's Approach*. McGraw-Hill, Inc.

26 Edward L. Robertson. P465-6/P565-6 Course Packet. Course syllabus and supporting material available at local copy shop.

27 Geri Schneider and Jason P. Winters. *Applying Use Cases*. Addison-Wesley Publishing Company, 1998.

28 Ganesah Shankaranarayan and Adir Even. The metadata enigma. *Comm. of the ACM*, 49(2):88–94, February 2005.

29 Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, fifth edition, 1996.

30 John F. Sowa and John A. Zachman. Extending and formalizing the framework for information systems architecture. *IBM Systems Journal*, 31(3), 1992. IBM Publication G321-5488.

31 W. Strunk and White E. B. *The Elements of Style*. Allyn and Bacon, fourth edition, 1999.

32 Anneliese von Mayrhauser. *Software Engineering Methods and Management*. Academic Press, Inc., 1990.

33 W3C. Resource Description Framework (RDF). `www.w3.org/RDF/`.

34 Webster's dictionairy, 7^{th} edition. online version.

35 Jeffrey L. Whitten, Lonnie D. Bentley, and Victor M. Barlow. *Systems Analysis and Design Methods*. Irwin, 1989.

36 John A. Zachman. A framework for information systems architecture. *IBM Systems Journal*, 26(3), 1987. IBM Publication G321-5298.