# DEBUGGING CHRONICLES

Lincoln Stoller, Ph.D.

**Chapter one — Table of Contents**



*(This is the first of a six chapter book on common mistakes in the design and implementation of computer applications and how to avoid them.)*

## INTRODUCTION

Let's face it — most custom applications are poorly designed and full of bugs. Programmers are frequently unskilled at designing and debugging their code because they are inadequately trained; and they are inadequately trained because good training is generally unavailable. The technologies that have brought us enterprise-wide, desktop databases have forced a multitude of previously shared responsibilities onto a single person: the programmer. Meanwhile, opportunities for programmer training are not commensurate with the demands of these new technologies.

The training opportunities that do exist come mainly in the form of the occasional seminar, and a plethora of books dealing with single issues relating to a given stage or aspect of the development process, most of which are either extremely technical, narrowly focused, or highly theoretical. This book attempts to fill in the gaps by presenting an integrated approach to small system development and project management, which acknowledges the interrelationships among the various phases of a project rather than treating tasks and problems as isolated occurrences. The approach is practical and problem-oriented rather than theoretical.

Most people, including many programmers, have the impression that creating software primarily requires the ability to string commands together. This notion has particular currency in the world of desktop computer systems. Students are first taught to program, and only later, if ever, taught

system design. Perhaps this is due to the rapid growth in programming tools and interface standards, or the high demand for new software. Whatever the reason, only a small minority of programmers — and usually only after countless trials and errors — know how to create a successful system from the ground up.

Despite the promise of easy-to-learn fourth generation languages, creating software is not a simple task. First there are the technical hurdles of analysis, design, project management, testing, installation, and support. Add to this the human issues of communication, cooperation, personnel management, and the ability to change one's attitude or direction as required in the different stages of a project. Handling these issues requires skills that are interdependent, skills that support and depend on each other. Just as you cannot learn to juggle one ball at a time, you cannot learn design first and communication next, then expect to add them together. You need to be effective in all of the requisite skills before creating software.

As a person who has developed ground-up systems and who now spends much of his time wading through other people's code, I know firsthand the types of problems that arise when the perfectly intelligent but insufficiently trained programmer sets about the task of designing and developing an application. And like so many others in this field, I have had to teach myself most of what I know about system design, development, and debugging.

This book is accordingly based on errors I have made along the way, then subsequently learned to recognize and rectify. It is written primarily for developers of desktop software, and secondarily for clients who work with such developers. I expect that most readers will have had some personal experience with the software development process and the problems on which this book is based. Yet it can also be used by those who are new to the process as a resource for learning about small system development. The discussion contained herein is by no means complete — I am still learning, and encourage you to fill in the gaps.

The book is organized according to the six phases of small system software development: 1) analysis, 2) design, 3) programming, 4) testing, 5) installation, and 6) support. Each chapter focuses on a pitfall or problem that commonly besets the process as the project unfolds. Some of these problems can be chronic and annoying, others catastrophic. If the effort has resulted in usable software, then these problems result in what are generally called "bugs." Although any problem can be called a bug, those technical errors usually recognized as such often originate in flaws in the development process. As I use it here, the term "bug" means any unexpected problem or error. A bug may be narrow and technical or broad and conceptual. It may be a trivial artifact or a menacing systematic error. The one thing all bugs have in common is this: each is a symptom of other problems.

Avoiding problems in software development is not something to do after you have written code any more than avoiding traffic accidents is something to do

after losing control of your car. The creation of bug-free software begins at the earliest stage of analysis, and continues throughout the software development process. The threats to this process come from many directions, and each individual's work has its own weaknesses. Some problems originate in design, others in implementation, still others in execution. Some bugs arise from errors in syntax, some from errors in logic, still others from a breakdown in communication. The emphasis here is on developing those skills and habits that result in fewer bugs. I also offer suggestions about how to find and fix bugs when they do occur. I hope that the I ideas I present seem so reasonable and attractive that you will begin to incorporate them into your own work.

## DEBUGGING CHRONICLES #2: MISUNDERSTANDING THE PROBLEM

Problem: You don't really understand the dynamics of the situation.

### The analysis phase

Good software design requires a thorough understanding of the problem that the code is intended to address. Defining the scope of the problem has more effect on the life cycle of a development project than the particular software solution. Any given solution will only be as good as your understanding of the problem, and such an understanding is the objective of the initial analysis phase.

The operational problem is easier to describe than the software solution. This is because the user (your client or manager) is already familiar with the problem, and both of you can use the same language to describe it. This is not to say that the problem doesn't involve technical terms; rather, the problem is described in the same language used to describe the current operations. In contrast, describing how software operates relies on concepts that are often foreign to the user, and that tend to obscure the original problem. This distinction between the problem and solution domains plays a central role in systems design.

Don't be misled by users who want quick, simple solutions to complex problems. When a user requests such a solution, it either means that he or she is naïve with regard to software design, or that the problem is out of control. In such cases it is all the more important for you to develop a thorough understanding of the problem before proceeding to developing a solution. Doing anything less unnecessarily compromises your value as a solution provider.

### Statics: what's the problem?

Things that are static remain the same. By the term "static problem" I mean the "slice in time" description of the problem requiring a software solution as articulated by the user. It is static in the sense that the user has already abstracted a set of crucial circumstances that he or she believes reflects important aspects of the business' operations. This expresses what the user both wants and thinks is needed. Whether or not the assessment is correct, such a description is not complete so long as it lacks the dynamic component.

There are three questions you can ask to get a better understanding of the scope of the static problem. They are:

1 ) What are the primary business functions?

That is, what are the activities that contribute value to the organization? Less centralized activities may be left out of the initial description because they are

less prominent, yet such activities may still play an essential role. It is your role to tease these out and address them in your solution.

2 ) What activities support these functions?

Who is involved in supporting the primary functions, and what must the business provide in order for them to be effective?

3 ) What do other users think about the initial description of the problem?

If possible, survey end-users to find out what they think is important. Frequently, lower level end-users have a more accurate sense of day-to-day operational problems than managers or executives.

These questions are designed to elicit a more complete description of the problem than that originally presented to you. The user's version of the static problem is an expression of what he or she thinks the problem is for the purposes of developing software, and this may not be the best or most accurate description. You must uncover the *real* problem, because solving the wrong problem will doom the project from the start. If you hear the dreaded refrain, "It may be what I asked for, but it's not what I want," then you have failed to understand the problem.

**Dynamics: how can the problem change?**

Things that are dynamic are things that change. While it is natural for a programmer to wish for specs that are static, a good designer expects otherwise. Even if the user presents a static picture of business operations, a true picture must include the changing nature of the business, the market, its objectives, and its operations.

Users will exclude from their specs the vague and uncertain forces that affect their operations not because they are unimportant, but because they don't know how to name or quantify them. It would be a mistake for you to assume that these forces don't exist or can simply be ignored: you must make yourself aware of these influences. Often what the user perceives as unquantifiable has immediate consequences from the perspective of software design.

For example, consider a business that deals exclusively with a single sales rep firm, and that presents you with a set of specifications that make no mention of a potential change in that situation. The user may think that the forces leading him or her to consider multiple sales reps is too uncertain or too complex to be factored into the system design. However, from a design

perspective, it is easy to add a field to each invoice that tracks which sales rep is involved. It would be much easier to include this flexibility into the design *before* the application is programmed than after it is already in operation. Knowing dynamic factors may enable you to design a better, more flexible application with little extra effort.

As most of us know from experience, many design decisions have multiple solutions. Choosing a particular solution requires us to make certain assumptions about the system that are not discussed in the specs. These may include assumptions about the value of flexibility, the importance of speed, or some other aspect of operations. Since it is too much to ask the user for advice in these situations, we usually make the decisions ourselves. To make the best choices in these myriad decisions we need a comprehensive understanding of the dynamics of the situation.

**Rapport and politics**

Beyond establishing an objective and agreeing on a means by which software will satisfy this objective, it is also important to develop a rapport with the user(s) and to carefully maneuver through sometimes sticky internal politics. This of course implicates a wealth of issues that are usually ignored by the developer.

Establishing rapport with co-workers is essential for the success of a team project. To some degree your rapport with others will be based on how your abilities, insights and expertise contribute to the process. But do not make the technician's mistake of assuming that your role in the project can be based on technical factors alone. If you are not playing a positive role in the lives of the people you work with, then you will be perceived as a not entirely positive force. This means that you must also apply your organization, communication, and judgment skills in a way that is appreciated. If you see this as unnecessary, then you are either marginalizing yourself, failing to understand the larger situation, or both.

Bear in mind that in addition to controlled, objective and rational forces, businesses are also driven by forces that are spontaneous, subjective and emotional. These forces are difficult for an outsider to perceive, and even harder to predict. Nonetheless, they can exert a strong and sometimes dominant influence over the course of events. While it may be too simplistic to think that all decisions are motivated by greed or fear, for example, important decisions always reflect certain forces of this nature.

Personal forces are always at play in the politics of an organization. Outside influences (such as you) are often perceived either as threats or opportunities to insiders in different circumstances. Those in similar situations often join together to strengthen their ability to effect change, and allegiances are formed between people of like mind. If your job is to automate the business, then there are probably people who view *you* as the problem. Expect personal issues to arise with these people. The days of monolithic IS departments

shielding you from the users is gone. Today, whether you like it or not, political maneuvering is part of your job.

**Golden Rules**

Failure to appreciate the subtle aspects of a design problem can result in a solution that misses the mark, and in a project that is at risk of collapse. To minimize these risks, consider the following steps:

1) Draft realistic proposals based on a thorough appreciation of the problem from the user's perspective.
2) In addition to understanding the static objectives, design your solution so that it addresses dynamic forces.
3) Recognize political forces and be proactive. These forces will remain unstated unless you ask.

## DEBUGGING CHRONICLES #3: MISUNDERSTANDING THE OBJECTIVE

Problem: You don't really understand what needs to be done.

### Part A

#### Define the objective

There are many stages in the development of computer systems, and various ways to approach a problem. Since any given problem is inevitably too big to tackle in a single step, it is your job to break the problem down and propose a solution. The trick is in understanding *how* to undertake such a breakdown effectively in this crucial stage of the project. The first step in doing so is defining the objective.

I propose considering that each project have one of three possible objectives:

1) development of a prototype;

2) development of an initial system; or

3) development of a comprehensive system.

Larger projects may break down into finer gradations with partial objectives, such as a feasibility study or an analysis of information flow. But in the context of desktop systems for small to medium-sized businesses, you'll probably be involved in projects with one of these objectives.

Each objective can be decomposed into the following five phases:

1) rough estimate

2) software design

3) refined estimate (fixed bid)

4) programming

5) installation

Note that these are sequential phases that build upon each other. The completion of each phase affords an opportunity to review the specifications and to remedy problems before proceeding to the next. Each successive phase also requires an increase in the client's commitment to the project. Each phase can be treated as a semi-independent product that can be contracted, billed, and delivered separately.

#### The development grid

For heuristic purposes we might consider software projects in terms of a development grid whose columns list alternative objectives and whose rows list the phases involved in each, as shown below. The grid presents the three small-system objectives in terms of the five project phases. A rough estimate

of the cost of each phase is shown in terms of some relative "unit" of cost, as in the sample values provided here:

| Relative Cost of Project Phases | Prototype | Initial System | Comprehensive System |
|---|---|---|---|
| Rough estimate | 0 | 2 | 6 |
| Software design | 2 | 10 | 30 |
| Refined estimate | 0 | 3 | 9 |
| Programming | 15 | 40 | 100 |
| Installation | 2 | 6 | 18 |
| Total Cost | 19 | 61 | 163 |

**Figure 3.1: the development grid.**

By adding the cost in each of the columns you can see that the three objectives have significantly different costs. A prototype is roughly one third the cost of an initial system, which is, in turn, one third the cost of a comprehensive system. When zeros appear they mean the corresponding phase either is not done, or is done at virtually no cost.

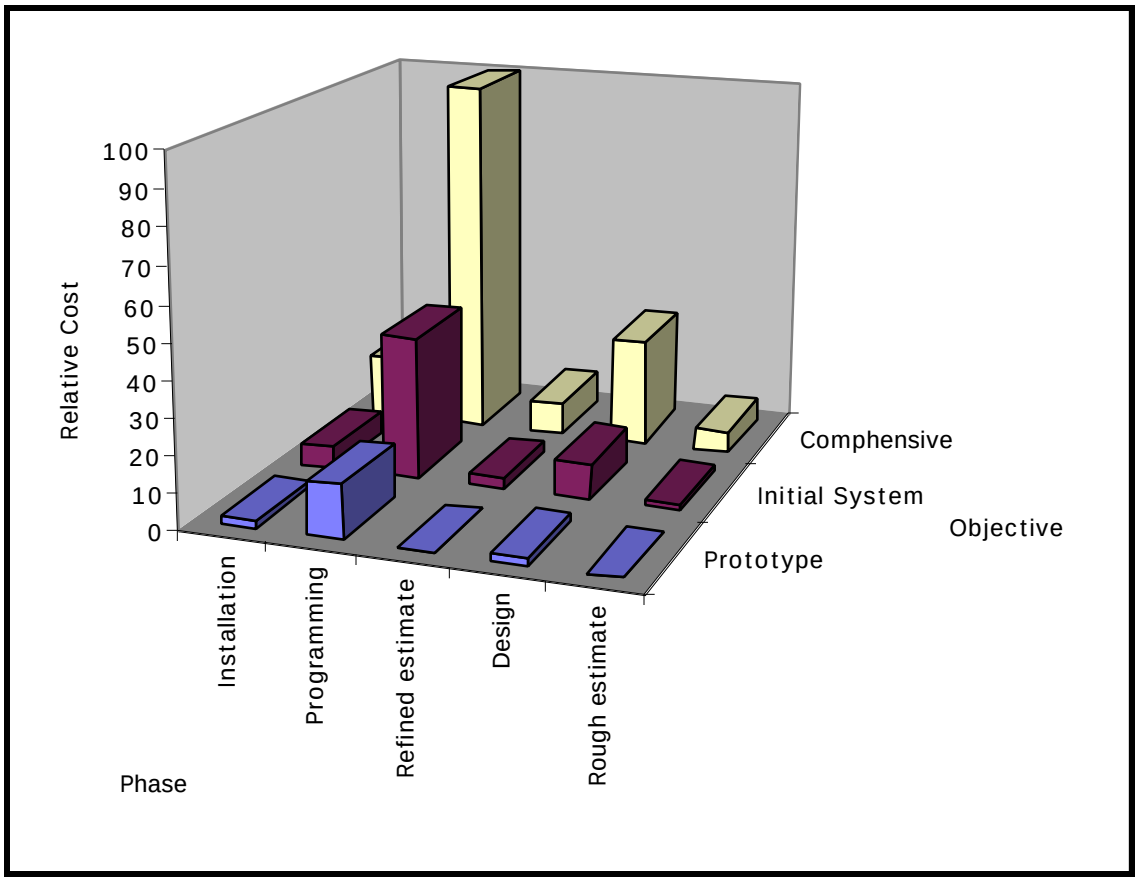


Figure 3.2: the costs associated with the development grid.

**The Objectives:**
**PROTOTYPING**

Producing a prototype means quickly creating a rough application using an untested design. The purpose of a prototype is to test the design. Prototyping is a form of systems analysis made possible by RAD environments. Prototypes are useful because they express technical design decisions in graphical and functional ways. The prototype's friendly — or at least familiar — face allows less technical people immediate access to the design process. This can simplify and streamline the process if the user participates sucessfully. However, the valuable result is the *design*, the blueprint embodied in the prototype. The prototype is not the final realization of the design.

A frequent motivation for prototyping is quicker development at lower cost. Unfortunately, this plays into the misconception that the prototype will serve as a final product — an idea that is usually dispelled as increasing amounts of time and money are spent to extend and refine the prototype.

**Initial System, or Staged Effort**
An initial system:

- is a minimal design that realizes part of a larger objective;
- is based on detailed and complete specs;
- makes use of known factors rather than testing unknown factors;
- requires a complete software development cycle.

The development of an initial system is the first step in a staged development effort. Reasons for taking this approach include:

- The client only has the resources and/or background to support a limited effort.
- Core functions need to be automated immediately.
- It is desirable to limit the number of unknowns. These can include personal and business factors like compatibility, work quality and credit risk, in addition to the usual technical issues.

Initial systems are a good choice for inexperienced clients and developers, or for projects whose modular or sequential nature makes them suitable for staged development. A defining characteristic of the initial system approach is the absence of certain components central to the complete design.

If you combine the time and cost of all of the stages in a staged development effort, it will be more expensive than a single, comprehensive effort. But because it's done in separately monitored and evaluated stages it has lower risk.

**Comprehensive System**

A comprehensive system effort is one that starts with nothing and aims to satisfy most automation objectives. Even in this case it is advantageous to limit the scope of the proposed system in order to minimize development risks. On the other hand, unlike either of the other approaches, a comprehensive system includes all core operations. Its attraction is that it combines a sequence of development efforts, allowing for more managerial control, a more tightly orchestrated team effort and a more tightly integrated final application. Also, certain problems do not lend themselves to decomposition and so cannot be implemented in stages. Some clients are not interested in a staged effort, preferring a single schedule, a single set of goals and a single, complete objective. Comprehensive systems are easier to install and support than staged systems, which continue to undergo change as new stages are completed.

The problem with a comprehensive effort, aside from its complexity, is the difficulty in controlling it. That means controlling cost and production

schedule, soliciting feedback, incorporating changes and enforcing quality control. If any of these aspects of the project is mishandled it could threaten the viability of the project. Comprehensive efforts require the most experienced players and demand a higher level of discipline and responsibility.

Part B of this article will go on to consider the five phases and how each contributes to the overall understanding of the objective.

## DEBUGGING CHRONICLES #3: MISUNDERSTANDING THE OBJECTIVE

### Part B

**The Phases:**
In this installment I consider five different phases of a project and describe how they fit into the development effort. The five phases are 1) the rough estimate, 2) the software design, 3) the refined estimate, 4) programming, and 5) installation. Each of the three objectives can be broken down into these five phases. The amount of work required in each phase differs with the objective, as does the information needed to complete each phase.

**1) Rough Estimate**
Most efforts begin with a rough estimate. In addition to providing an approximate cost, the estimate also delimits the scope of the project in terms of complexity, time, and other resources. It also affords a first opportunity for establishing a relational file design, a graphical interface, and a functional description. None of these should be considered complete or accurate, but some estimate of each will be needed.

The result of a rough estimate is a document that sketches the system's structure, functions, and costs. This document summarizes your initial understanding of, and solution to, the development problem. It should break the development effort into tasks of relatively short duration, say between a day and a week, and should be written in a language that the client can understand. Since the rough estimate is both a schematic and a preliminary cost estimate, some form of it must exist before proceeding to the next phase of the project.

I like to put enough time into my calculations to feel that the estimated cost is accurate to within 30%. For a small, 100-hour project the rough estimate may require a few hours on-site and a day of drafting an analysis. Developers specializing in projects smaller than this often supply a rough estimate at no charge. But for anything larger, a rough estimate will require a couple of days to prepare and should be compensated.

**2) Software Design**
The software design phase includes accumulating the following necessary details:

- The functional specs;
- Resource specs (hardware, software, personnel, budget);
- Data flow diagram (showing where information is drawn from and sent to);

- System structure chart (showing how the user interacts with and controls the system);
- The file structure;
- The data dictionary;
- Special algorithms.

Each objective has different design requirements, and clients differ in their ability to supply this information. The design phase requires a significant amount of effort, organization, and exactitude. One of the benefits often cited to support the prototyping approach to development is that it entails minimal design requirements. This often appeals to smaller clients who lack the resources or the experience to support a more extensive design phase. It also reflects the typical programmer's belief that the sooner he or she begins coding, the better. This, however, is rarely the case.

Inadequate design skills is probably the greatest shortcoming of desktop system developers; most developers habitually produce inadequate design specs. Many have no training in system design and don't know what it requires. This generates the following self perpetuating cycle, in which the programmer:

- Doesn't understand the design phase;
- Underestimates the importance of good design;
- Underbudgets for design;
- Generates inadequate specs;
- Suffers project delays, setbacks, revisions, and overruns;
- Completes the project without an understanding of how the design process could be improved;
- Starts a new project with an inadequate understanding of the design phase.

The truth is that complete design documents are as necessary as charts for navigating the ocean — only an inexperienced sailor would set sail without them.

If you are new to programming, you should understand that software engineering is a field that has been around for 20 or 30 years. There are books on the subject, national conferences, university courses, and professional certifications. The problem for the small system developer is that much of this material is inapplicable or overly theoretical. Most authors address a

narrowly practicing group of academics and information system professionals working within the older, mainframe paradigm.

My purpose in writing these articles is to place this type of knowledge in a context appropriate to small systems development. Once you know what you need to learn and why, you can find supporting material in many sources, a few of which I've listed at the end of this article.

**3) Refined Estimate**

I define a refined estimate as anything from a slightly revised rough estimate based on new information, to an accurate estimate that is used as the basis for a fixed bid, the latter being an estimate where you, the developer, take all the risk of error.

I have never written a fixed bid, though I've been involved in some horrendously bad proposals. In these situations the only thing estimated was the finished cost. The value was based purely on overly optimistic estimates of how many hours of presumably trouble-free programming would be involved, and if the projects had been completed — which they weren't — the actual costs would have been 5 to 10 times higher!

Ideally, a refined estimate is a detailed plan of the development project in all its material phases. The best way to learn how to draft such an estimate is to keep detailed project mangement records that will shed light on the difficult, unpredictable, and expensive aspects of project development. A good refined estimate must consider all these issues *before* the project even begins. I offer the following suggestions:

- Write down and consider every project development factor you can think of.
- Draft all recommended documents.
- Prevent the client from changing the specs. If changes are unavoidable, clearly communicate to the client what these changes will cost.
- Learn to manage the programming process, which includes managing people. This will help to prevent wasted time and cost overruns.

One should also take into consideration indistinct or unforseen tasks and situations like:

- Documentation review and revision;
- Design and code review, revision, and debugging;
- System failures, software incompatibilites, data corruption, and lost work;

- Project management and personnel problems;
- Sickness, stress, interpersonal strife.

For your own protection, it is advisable either to state clearly that such tasks and circumstances have not been included in your estimate, or to include a 'guesstimate' accounting for such variables.

**4) Programming**

Since this article is about understanding the development objective, you might assume that this objective is fairly well understood by the time you begin programming. In certain respects that is true. If you've worked according to estimates and produced adequate design specifications, then much of the application's final objective will have been defined. But your employers may have assumptions about when they will see partial results, what those results will show, and the extent to which they will be able to make suggestions and modifications. You may see the final product as the clear objective, but an employer — especially one wishing to have active oversight — will have his or her own assumptions and expectations about how the programming phase will progress. Addressing those expectations will be crucial in maintaining the employer's confidence and continued support.

Common errors you might make at this stage are the assumptions that the employer:

- Is interested in the details of your work;
- Perceives your completion of intermediate steps as adequate progress;
- Will not make major changes in the specs if given the opportunity;
- Will feel greater commitment to the project as it progresses;
- Can be made to feel a greater incentive to pay old unpaid bills through the submission of new bills;
- Accepts the cost of any changes requested without your having to explain them;
- Appreciates the difficulty of software development and will forgive your miscalculations.

In my experience these assumptions tend not to bear themselves out. The employer's expectations will have an impact on establishing an appropriate objective, and therefore should be given serious consideration.

I consider ongoing debugging work to be part of the programming phase. The issue of good debugging skills is such a large one that I won't consider it here; managing, reviewing, and debugging software are themes I will address in future articles in this series.

For the purposes of understanding the development objective, the important aspect of the programming phase is knowing how it will progress and when, and what it will produce. In this context the process of debugging acts to ensure that the products of your labor function as intended. Clearly, if you are adept at writing code at a scheduled pace, but the code you produce functions incorrectly, then your efforts may be judged to be of no value. The error may be small and insignificant, but the client doesn't know that and, in most likelihood, neither do you. I will assume here that as you progress through the programming phase you are producing working, relatively bug free code. (If you've chosen not to debug your code as you go along — and there may be instances in which this is appropriate — then it's very important that your employer accept this method and understand the implications.)

**5) Installation**

As with the programming phase, the aspect of the installation phase that concerns us here is client expectations. Misunderstood expectations could lead you to think that your obligations end with the delivery of the application. But the client's objective has not been achieved until the application is successfully integrated and is processing information.

You may perceive the transition from completed application to integrated application as minor, but the client may perceive this process as fraught with risk. The client may be aware of difficulties that elude you, such as the training of users, or the considerable impact of a small bug from an operational point of view.

The three installation phases in which you could be expected to play an important role are:

- Providing documentation and training to end-users;
- Fixing bugs that appear in the course of actual use of the application;
- Guiding the 'data cut-over' process in which historical data are brought into the new system.

Each of these tasks requires its own set of skills. You don't need to be an expert in each, but you do need to consider each and make clear to the client what steps will be necessary and exactly what role you can be expected play. You could put down everything else you're doing and learn how to write technical documentation, how to teach, how to debug, and how to move data between the relevant file formats. On the other hand, you could simply proceed with

whatever skills you already have, and allocate aspects of the installation and support process to others. It's no crime to lack certain skills, but it is important to know your own strengths and weaknesses. Technical people are frequently poor at teaching and/or writing. Let the client know what services you can and cannot provide, and do not commit yourself to providing support that goes beyond your level of competence.

If you do plan to provide support during the installation phase, it would be wise to introduce the client to your work in these areas while the project is still in the design phase. For example, show the client user documentation from earlier projects. If you're not the person in your organization who provides on-site training, then introduce your client to the person who does. The data cut-over process should be given enough consideration to ensure that what's needed is feasible. Again, you don't have to solve every problem or succeed at every task — you just have to deliver according to the expectations that have been established.

**Golden Rules**

The three design objectives vary widely in project magnitude, commitment, and risk. By finding the alternative that best fits the client's needs you can avoid conflicts and misunderstandings. This will result in a more effective use of resources and a more successful development effort. Be sure to:

1) Consider the alternatives of designing a prototype, an initial system, or a comprehensive system;
2) Offer your client a choice. Explain the benefits of each and how each phase is a separate product;
3) Select the objective and methodology that best fits your client's resources and expectations;
4) Define the methodology yourself, even if your client helps to define the objective.

## DEBUGGING CHRONICLES #4: INADEQUATE SPECIFICATION

Problem: You accept inconsistent or incomplete functional specifications.

### Part A

### Specifications

Specifications are written or stated instructions that describe the application and the development project. Creating the application is one of the objectives of the project, but while the application is a 'thing,' the project is a 'process.' Small system programmers learn how to code an application, but most are poorly versed in the development process. The two are quite different, requiring different skills. The lack of expertise in knowing how to gather and use project specifications is the source of most problems in small development projects.

In this article I present some of the more common errors that can occur in the process of developing specifications. Then I offer a general description of the correct process of gathering and using specifications.

### Things That Go Wrong

#### #1: Allowing the Client to Manage the Project

In this scenario, the client (assuming here an individual who does not have a background in programming) believes that his or her basic concept and description is sufficient for designing the software, and that an operational understanding is all that is needed to resolve any programming problems that might arise in the course of extending the basic design. The client urges the programmer to start coding as soon as possible, and takes responsibility for providing additional directions as needed.

While the client may have a highly developed functional understanding of the business, her or she generally will lack a comprehensive structural or mechanical understanding in information management terms. The difference between this functional understanding and a structural or mechanical one is like the difference between knowing how to read a clock and how to build one.

In well-designed software, structure and function are subtly related: major structures correspond to the central functional concepts. Common functional ideas, which may be spread throughout business operations, are centralized in the structure of the application. The major functional areas appear only at the highest level. Their predominance is often limited to the user interface.

Consider accounting as an example. Business operations generally revolve around filling out and processing forms, such as invoices, purchase orders, packing slips, deposits and withdrawals. These operations are functionally central, but they all tie back into a common accounting underpinning.

Developing an effective business system requires starting with accounting, even though accounting is not the primary function.

The programmer who allows the client to design the system will implement basic functions and only later learn of major exceptions that either violate or exceed the scope of the original design. He or she will discover that major functions are not identical to major information structure. Technical issues will inevitably arise that cut the client out of the decision loop, causing cooperative interaction to break down. Attempts to remedy these problems will result in overshooting both the schedule and the budget.

**#2: Overlooking Inadequate Resources**
In this scenario, the client specifies the scope of the system and the budget. The programmer tries to prepare a matching bid either by avoiding or minimizing certain phases in the process. Or the programmer may skip necessary supporting functions because the client did not mention them.

For example, consider the developer who does not include in the estimate the cost of developing reusable procedures for managing windows, record locking, and the assignment of ID values. The lack of these features will become evident when the application is subjected to operational and data integrity requirements. Their subsequent inclusion will be costly and time consuming, far more so than if they had been included in the foundation of the original application.

**#3: Adding Functions Without Analysis ('Feature Creep')**
This occurs when the programmer includes or even encourages the client's requests or suggestions, and implements them without considering their full scope or consequences. Cost estimates are not prepared or discussed, and the value of the extra features is not determined. As a consequence, the number of defects rises, the schedule is delayed, efforts are uncoordinated, and the client is usually dissatisfied.

**#4: Stages, or Modules, Completed Without Incremental Review**
Because of time constraints or the desire to appear productive, in this scenario stages in the development process are not subjected to review and testing. If programming is in progress these stages may correspond to the completion of code segments or modules. Unseen problems inevitably accumulate. The number of unexplored paths, and hence undiscovered bugs, increases geometrically. That would mean that 100 errors in module A, when combined with 100 in module B, could generate up to 10,000 ways to trigger these errors in the code!

The small amount of time and effort saved by not reviewing each stage as it is completed leads to a much larger effort later. Apparent progress is deceptive, and estimations based on this rate of progress will probably be misleading and

inaccurate. Whatever problem is being deferred in the present will only return with greater impact later.

**#5: Poor Project Management**
In this case the programmer doesn't know what is needed and doesn't stop to ask. This is either because the client is in a hurry or wants to contain costs. Necessary ingredients and essential steps are simply left out. The programmer assumes that by giving the client a cursory warning, he or she is no longer liable for any problems that may appear. However, clients usually expect the programmer to provide critical direction and accept ultimate responsibility. Problems arising from misunderstood responsibilities result in difficult communications and confused expectations at best. At worst they result in accusations, noncompliance, and the project's collapse.

**#6: Failure to Establish Terms of Work or Payment**
This is a special case of poor project management that occurs right at the start. It arises from programmers not knowing why and how to legally protect themselves, not knowing the importance of written ground rules, and not knowing what alternatives and emergencies to plan for.

Many people prefer to avoid legal documents, and shy away from any discussion of problems and problem resolution. This stems from a variety of sources, including:

*Legal Phobia*: the dislike of legalese, lawyers, and the associated costs, as well as a studied ignorance of the issues involved.

*Over-optimism (aka the Candide Syndrome)*: the preference to remain in the domain of the 'warm and fuzzy,' where enthusiasm is high and the future looks rosy. This is frequently accompanied by the belief that discussing problems before they exist jinxes the project or introduces pessimism and distrust.

*Legal Blindness (aka the Ostrich Syndrome)*: the belief that what you don't know won't hurt you.

*Naïveté*: a misplaced trust in the good intentions of the individuals and organizations party to the effort.

What is needed is a full written disclosure of both the client's and programmer's responsibilities and obligations. This includes progress benchmarks, payment terms and access to resources, as well as an agreement about what to do when conditions are not met and how to resolve problems.

There is no substitute for good legal documents and advice. Only inexperienced business people proceed without contracts, and they often learn through difficulties they hope never to repeat.

In part B of this article I'll give my recipe for the kind of specs that can help to prevent these problems.

## DEBUGGING CHRONICLES #4: INADEQUATE SPECIFICATION

Part B

## ADEQUATE SPECIFICATIONS

What Specifications Should Accomplish
Project specifications should be:

*Comprehensive:* they should consider every functional area and how the areas interact.
*Complete:* they should include all major processes within each area.
*Detailed:* they should include every important step within each process.
*Verified:* all functions should be communicated to, understood, and approved by informed users.
*Accurate:* design and operations should be tested against known results.
*Logical:* design and operations should be analyzed for logical flaws.
*Practical:* design and operations should be considered in terms of cost, maintainability, and performance.
A separate document could conceivably be prepared for each criterion, but that would not be practical. The specs should be prepared as working documents, and should be flexible, brief, and focused.

There is no one best approach to working with specifications in all situations. What documents you prepare will depend on your specific needs. These, in turn, will be determined by:

- the problem's complexity;
- the problem's modularity;
- the participants' familiarity with the development process;
- the participants' familiarity with each other, and with a common technical language;
- the range of perspectives and responsibilities;
- what is written in the contract.

This having been said, there are some general guidelines for how to make documents most useful. At the end of this article I will describe, by way of example, a set of documents that I've found sufficient in complex design projects.

## WORKING DOCUMENTS

Documentation can be a curse when it is excessive, inaccurate, or unreadable. To make specification documents as useful as possible, they should serve several functions at once. Specification documents should be vehicles for communication and descriptions of functions and features, and should provide a basis or outline for the next stage of development. In addition, specifications can, in many cases, be reused in the development of user manuals.

Specifications need to be *working* documents. This means that they both accomplish their task of enumerating features and tasks, and play a central and vital role in the life of the project.

What goes into the specs is not a great mystery. Anyone familiar with software development will probably find the above list of what specs should accomplish to be rather obvious. The mystery lies in knowing how to prepare and use project specifications. Because it's so important to make the specs useful, we'll consider what it takes to make a successful working document.

**Comprehensibility**

Documentation should be as readable as possible. It should be able to be read or used by the software designer, the system analyst, the project manager, and even the end-user to some extent. Creating something useful to this range of readers is not an easy task, especially given the technical nature of the material. The following guidelines may help:

*Write for simplicity:* imagine that you are addressing an uninformed reader. Keep this imaginary reader in mind at all times.
*Outline carefully:* gather similar topics together, and keep your focus narrow.
*Number each paragraph:* use a hierarchical scheme whereby each level of detail has its own number. I prefer a legal numbering style, where I can always add additional levels without having to renumber everything (see Figure 1 below). I also find that this style shows each issue's prominence more immediately.
*Introduce and summarize:* motivate each section by putting it into context. Summarize what has been said at the end of each section.
*Avoid technical jargon:* if you must use technical terms, explain them carefully.
*Segregate details into sections that are clearly marked:* make it easy for readers to get an overview, or to skip sections that are not relevant.
*Spend time preparing the specs:* this may mean billing for the specs on an hourly basis. Stand firm on the necessity of preparing good specifications.
*Learn from past experiences:* writing reflects mental organization. The specs reflect the project's organization, and they will improve as your management skills improve.

4. Client Area
    4.1 Demographics screen
        4.1.1 Billing Address
            4.1.1.1 Checking for unique addresses
            4.1.1.2 Modifying existing entries
        4.1.2. Shipping Address
            4.1.2.1 Checking for unique shipping addresses
            4.1.2.2 Storing different possible shipping addresses
                4.1.2.3 Marking a primary shipping address
    4.2 Contacts screen

etc.

Figure 1: Example of legal numbering in functional specifications.

**Reusability**

Some of the required documents are prepared in sequence; others are prepared simultaneously. You should aim to reuse the outlines on which the documents are based wherever possible. For example, the initial estimate should be closely related to the initial design documents. The initial design documents should be used as the basis for the organization of the detailed design documents. The system structure charts, which provide a map of how the interface works, should be the basis of user manuals.

The project evolves through a series of related stages, and the documents should evolve in a similar fashion. Just as the results of the previous stage feed into the next stage, the documents should reflect this process. This means that the structure of the documents, as well as some of the contents, reappear in subsequent stages with additional details.

This reuse of material is beneficial in two ways: it both reduces the need to reorganize your ideas at each stage, and it provides a common set of ideas that make reading and discussing the issues easier.

When you begin the process of specification, you should create an outline of how the documents will be related. This is a sort of meta-outline, or an outline of outlines. This will make the transitions from one document to the next easier. I'll show how you can do this in the next section, where I suggest a particular set of documents.

**Brevity and Clarity**

Be brief. Simple, clearly stated ideas are your friends, complicated expression your enemy. Just remember, the specs must be read, and therefore must be readable.

**Modularity**

Modularity works hand-in-hand with brevity. Chopping a large document into small sections makes it easier for your readers to get through the material. It also helps readers to break the problem down in their own minds. Modularity makes for choppy reading, but that's o.k. since you're writing documentation, not the great American novel. Modularity makes for better reference material. And if the specifications are to become working documents, they will need to be referred to frequently.

**The Documents Used for Project Specification**

In a standard situation the designer and user begin by deciding on an objective, such as one of the three objectives listed in article #3. Documentation for each phase of the project is prepared and discussed before work begins on that phase. The documentation could be used directly, as for

the purposes of scheduling the project, or could provide a point from which the schedule is developed. Each document would act as reference material for resolving problems that arise in completing that phase.

In the case of designing an initial system, as described in a previous article, the following specification documents could be used:

**Rough Estimate Phase**

**Work for Hire Contract** (for independent contractors)

The contract should describe the project in general terms. It should present each party's responsibilities and obligations. This includes the clarification of payment terms and product ownership, among other issues. I believe that a contract's primary objective is establishing communication and precedents for handling difficult situations. However, it is also a legal document and, as such, should be reviewed by an attorney.

**Rough Cost and Time Estimate**

This document would be written and approved by both parties. It should break the project down into estimable steps, provide an explanation of what each step is, and estimate cost and time for completion. It should also summarize the major system requirements.

Design Phase

The design phase roughly breaks into a functional description stage and an application design stage. In the first stage you solicite from the client what they want the system to do. This is accomplished using the functional specs, system structure, and screen design documents.

**Functional Specs**

This core document lists everything that the system is to do. The document should be written in the end-user's language. Its organization should reflect the operation of the system. Functional specs should be based on the same organization as the rough cost estimate. It provides a checklist of all functions to ensure that the system design is complete, a checklist for creating Acceptance Test Procedures (see below), and a working document for analyzing the system.

**System Structure**

This describes how the system is used. Specifically, it provides a map of each screen and the menus and functions associated with it. It details what can be done on each screen, and how each screen is related to every other. This document can be a combination of both maps and descriptions. Its purpose is both to elicit suggestions from the users as to what constitutes an optimal interface, and to confirm that the designer is providing all the needed functionality.

**Screen Design**

This is a set of designs and descriptions of the screens that appear in the System Structure document. The purpose of the document is to provide another means for the designer to explore the function of the system with the end-user without resorting to technical language. Designing screen

prototypes, either on paper or on-screen, is an effective means of running through the application before it's programmed.
In the second stage you design an application that meets these requirements.

**Data Flow**
This document centers around a diagram, the Data Flow Diagram (DFD), that illustrates what data enter the system, where they enter, where they go, and what they are used for. The purpose of the DFD is to provide a common reference enabling the designer and the users to discuss the system. The main purpose of this is to educate the designer about all facets of the system, and to facilitate the task of modularizing the system. The DFD plays a role in both the functional and the structural stages of the design.

The DFD identifies user-defined information stores. It details the entities and functions familiar to the user in the user's language. The DFD should not present the programmer's vision of the application or the application's file structure. These user and programmer's views are linked, but they will differ in content and structure.

**Structural Specs**
This is a companion document to the Functional Specs and should have roughly the same outline. This document should discuss how each item in the functional specs is going to be implemented. It connects the user's vision of the system with the programmer's vision of the application. It provides a blueprint telling the programmer precisely how to interpret the functional specs.

**File Structure & Data Dictionary**
This is the structure map of files and fields in the relational database. It must display the relationships and the dependencies of one file on another. It can also list the data integrity requirement for all data items, if these are not already included in the structural specifications. The data dictionary explains each field's purpose, type, and additional properties. These documents function as working documents for the programming phase.

**Refined Estimate Phase**

**Refined Cost and Time Estimate**
This can be an extension of the Rough Cost and Time Estimate prepared earlier. It should account for all the new items that have been included since the initial rough estimate. If certain options are being considered tentatively, then these should be handled in a separate section. If there are known uncertainties, these should also be given their own section.

The refined estimate draws on all the work done in the design phase. Its purpose is to provide a 'reality check' for the client to ensure that the project as currently envisioned is still within budget and on schedule. If this is not the case, then the Refined Cost and Time Estimate provides a working

document for reshaping the proposed system so that it lies within these constraints.

## Programming Phase

### Project Schedule

There are many ways to manage progress and scheduling. Scheduling means enumerating each task, deciding who is going to do it, how long it should take, what other work is required in order to complete the project, and what other tasks depend on this task. Because this level of scheduling is so specific, it is also prone to being inaccurate. However, it serves as a working document that helps the various parties to formulate their needs and expectations, providing a common reference that puts the various tasks in perspective. It also gives the programmer an accurate idea of the resources needed to complete the project. The process of preparing the schedule is more useful than the finished schedule.

### Progress Reports

Even if the application is being designed and programmed by the same person, that person should prepare quasi-regular progress reports. These can be detailed or general, colloquial or technical. Progress reports provide some record of what was done, why, and by whom. But their purpose is to coordinate ongoing activities, and in this respect their form is determined by what the workers and managers need.

### Acceptance Test Procedures

The best way to end a development effort is to meet a specific goal defined at the beginning of the project. That goal is embodied in the Acceptance Test Procedures (ATPs), which should read like a punch list in a construction project. They should include specific tests covering each of the functional specifications. Since we cannot consider all configurations of the data, nor specify all means of operating the application, the ATPs must limit themselves to a specific example, or several examples, of each operation. When the developed application satisfies the ATPs, it is considered finished.

The purpose of the ATPs is to segregate the unpredictable issues of debugging and last minute requests. However you decide to deal with these, they should be handled after the programming phase is complete. Meeting the ATPs' specifications marks the end of the development phase and the beginning of the installation phase.

**Cutover Plan**

'Cutover' is the process of carrying over existing data into the new application. This information can be brought in by simple file transfer, through manual re-entry, or by writing special import/export routines. You'll need to consider whether there will be changes in the type or format of certain items. For example, text items may need to be converted to date format, numeric data converted to string format, or any of a multitude of other possibilities.

The purpose of the cutover plan is to describe:
- the data that will be taken from existing data files;
- the method used to transfer this information;
- the source of other data necessary for the new application;
- the magnitude of the cutover process and the resources it will require.

**Installation and training**

**Training Syllabus**

This document contains a list of which personnel must learn to do what procedures in order to operate the system. It contains a proposal of what it will cost and how long it will take to train these users. And finally, it contains a course outline for student and trainer. The purpose of this document is to coordinate the training effort. Using the training syllabus, the design team can check that users will be adequately informed about how the system is to be supported.

**Administrator's Manual and End-User's Manual**

We've all seen these manuals, so they are less of a mystery than the project development documents. Although it is not easy to create good, well-organized and clearly written manuals, they can be based on the documents prepared in the design phase, particularly the functional specifications and the system structure documents.

Summary

While most programmers tend to think of writing an application simply in terms of creating a finished product, the product itself is the outcome of an entire process that encompasses much more than writing code. The development process calls for good management, organization, and communication skills. These skills must be mobilized in the creation of project specifications that serve to guide the project to completion. Good, clearly documented specifications — as described above and outlined in Figure 2 below — help the programmer to avoid common pitfalls, and provide the foundation for a successful project.

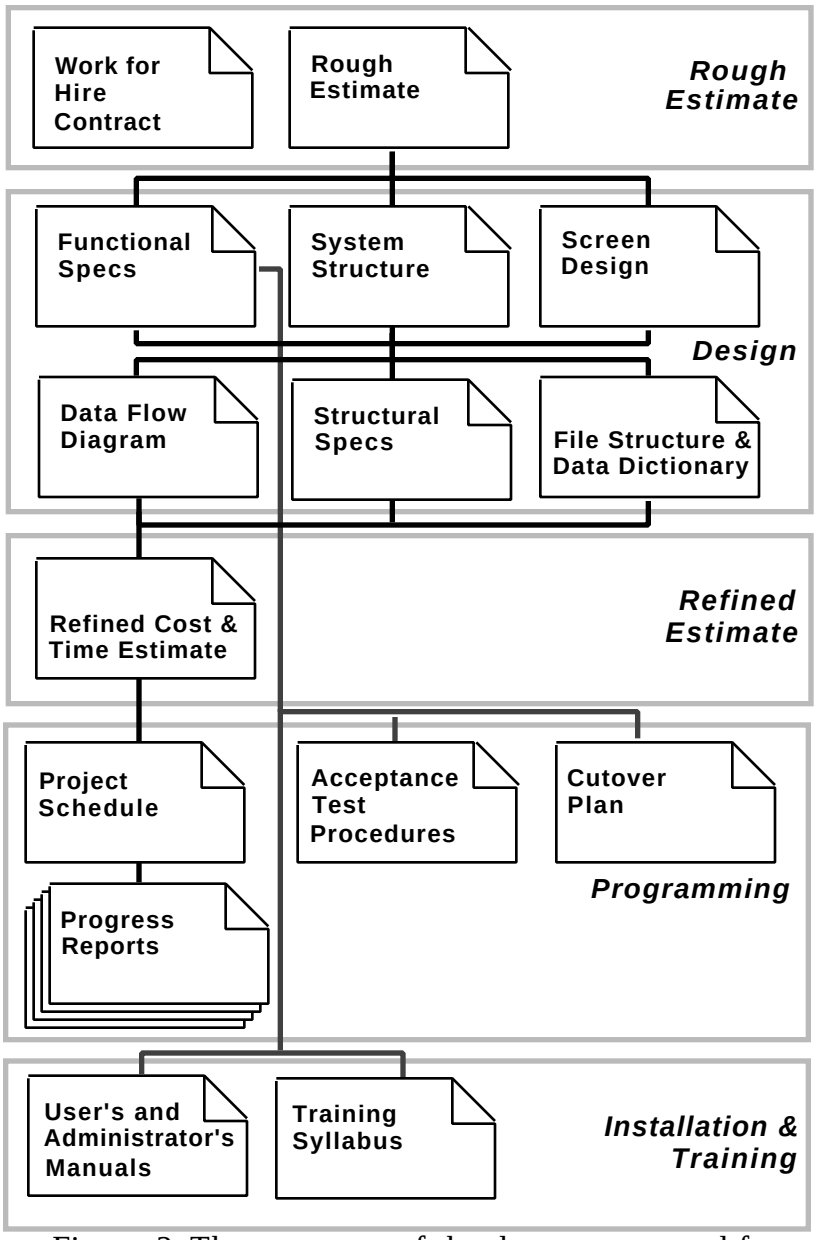


Figure 2: The sequence of the documents used for project specification.

## DEBUGGING CHRONICLES #5: FAILURE TO CONSIDER THE NEED FOR FLEXIBILITY
## PART A

Problem: You don't recognize the variability of certain elements, so you design an application that is rigid and difficult to support.

Adding flexibility simply means creating an application whose performance meets a range of current and future needs. The words *range* and *future* distinguish this effort from a basic analysis aimed simply at meeting current needs. The distinction may seem subtle, but it has deep implications. Analyzing the need for flexibility requires a complete understanding of the client's objectives. Designing flexibility into the application hinges on such an understanding, and on careful consideration of the technical issues involved.

### I. The Role of Code

This chapter deals with the interaction between code and concepts. It assumes an understanding of basic database programming concepts. My focus is on the design opportunities and limitations that are inherent in certain approaches to coding, not on what or how to code. To make an analogy, this chapter relates to the consequences of using different building materials as opposed to a discussion of how materials are to be used.

To be computer language nonspecific we need to define some basic terms.

*File:*
A file is a template. It defines the data that make up a record. It is also referred to as a table. We say that records are "stored in" files. Each file is unique in the sense that there is some way, either by name or number, to distinguish them. A file is a repository for data, but aside from its structure, it is not considered to be data itself. The file structure of a database is changed infrequently and only after careful consideration.

*Record:*
A record is a collection of data items that match the template of the file to which they are related. Records, also referred to as rows, are always stored as a whole. For the most part, all the data in a relational database are stored as records. Records are created and deleted as needed in the course of the normal operations.

*Field:*
The term 'field', also know as a 'column', can either refer to a datum specified as part of a file or to an actual instance of this type of information stored in a record. For example, we can say that the first field of some file is the ID number field. This implies that the first field of every record in that file contains an integer value used to uniquely identify the record.

*Database:*

A database (DB) is a collection of files and constituent records along with any related information. A DB can be a diffuse and ever-changing collection of unrelated files, or it can be a fixed set of tightly related files.

*List Screen:*
This is an on-screen or printed display of data in row and column format. Each row has the same structure. The number of columns is usually small, a dozen or so, while the number of rows is limited only by the amount of information stored in the system.

*Entry Screen:*
This type of screen allows the user to enter new data or to modify existing data. The simplest entry screen allows the user to enter the details for a single record. However, it is more common for the data that's entered to be stored in more than one record. These logically related records may be stored in one, or in several related files.

## II. Examples of Flexibility

It is important to understand the strengths and weaknesses of various coding strategies during the design process. Even so, ignoring the theoretical side of things is all too common. It is even considered a virtue by programmers who pride themselves on being "hackers." Hacking is a brute force approach to programming that some programmers honor as a kind of machismo.

To illustrate the consequences of code structure I've taken three different areas of database design — file, menu, and code structure — and constructed the following examples:

### File structure

Students are admonished to design databases that satisfy the requirements of "data normalization." This important concept, described in the Methods of Flexibility section later in this chapter, roughly means that independent but logically related data are managed as separate entities by the database (i.e., are records of related files). In real systems the factoring of data into related components may be imperfect. Or the factoring that may be true now, will become less true later, or vice versa. Consider the following two examples:

#### Example #1.

Each contact in the database may initially have their own address. To support this you might design a file that identifies the contact and, in the same file, stores their address. It may later appear that some contacts have several addresses. In addition, some contacts may have the same address. In this circumstance the address and the contact are no longer identical. Treating them as identical results in duplication of contacts for those who have several addresses, and addresses for contacts who share one address.

While it may be simpler to implement a one contact, one address structure within the same file, this will lead to difficulty if the multiplicity changes. At that point you'll be faced with the option of implementing cumbersome workarounds or changing the file structure. The first is a short-term expedient that becomes difficult to maintain. The second involves extensive reworking of data, code, and interface.

An alternative is two separate files: one for contacts, and one for addresses. This involves considerable extra work to keep the related information linked, and to manage multi-user access. But once developed it can serve a broad range of purposes within the project, and can be reused in other projects.

### Example #2.

Functional specifications may indicate that every client handled by the system has an account balance indicating how much they owe. To support this you may create an account balance field in a client file.

In the future, clients may enter into additional arrangements. They may sell goods back on consignment, or they may enter into cooperative marketing arrangements. These changes could be incorporated by adding new account fields as requested. This would mean that any accounting functions that scan and total accounts would have to be modified to include each new case. This would involve changes in many locations in code that might have been written to other specs or by other people. Changes of this sort are always an extensive source of new bugs, and require careful retesting of large parts of the application.

Alternatively, an account entity could be added to the file structure in the beginning. Adding new types of accounts would then be a matter of entering new records or new types of records. These separate records could be linked to the client, and new links could be added as necessary. If the related accounts are handled independently from the start, then the impact on the accounting modules of adding new records or types is minimized.

## Menus

Applications often use a menu structure where one menu is present for list screens and another for entry screens. This is adequate in situations where each file is managed similarly and independently. These menus, and the code behind them, are also reusable as long as the data they control are of relatively similar format.

### Example #3.

You design a menu bar that is used for all of your list screens. This menu bar has the items Search, Sort, and Print. The code is of a centralized design. You indicate to the menu bar which file you want it to act upon.

The search item allows the user to search each field of the indicated file for a particular value. The sort item allows one to sort the display on the basis of the values in any of the fields. And the report item allows one to print the list in either a summary or detailed form.

The benefit of the centralized code is that all the functions that support the listing of records are quickly implemented by specifying a new file and some associated parameters. Also, a new function can be added to all cases by making a change within the central code. This is an efficient implementation as long as it satisfies the specifications.

Now suppose that a special search is needed for a particular file. Suppose the user needs to be able to locate all clients who have purchased sales items in the last quarter. This search is only needed for the client's file, and nothing like it is needed for any other file.

The problem is that this is a computed search that depends on multiple parameters. It cannot be accomplished by the centralized code. To satisfy the requirement you will have to rewrite or replace the menu handling procedure.

If you choose to rewrite the procedure, then after you've implemented several "patches" to handle exceptional cases the centralized code becomes a hodge-podge of special cases and requirements. If you choose to replace the procedure in certain places, then it no longer acts as, nor offers the benefits of, a centralized procedure.

In contrast, if you develop code for handling menus that is intrinsically flexible, then at least the central parts, structures, or methods can be reused without sacrificing the whole. In OOP languages this can be done with objects and methods. In non-OOP languages it can be done by defining code structures, processes, and tools that are specific to menu handling, but general enough to support broad reuse. I'll give specific examples in the methods section.

### Code

Code includes everything from high-level preprogrammed functions to low level operating system calls. Coding is the most time-consuming part of programming, and the area in which reuse has its most immediate benefits.

### Example #4.

Suppose the system has a function that facilitates the writing of letters to contacts. Since each contact can have one or more addresses, you have decided to implement a scheme in which the various contact addresses are displayed once the contact has been chosen.

Displaying a small selection of related items is clearly something that will be of general use throughout the database. Some tools may already be available

to facilitate such a display. However, let's say that the tools are not exactly what you want and you have decided to program the selection feature yourself. You could create a special layout to list addresses. At the top of the list you will identify the contact, and the list will have columns giving the first line of the street address, the city and the state. The system will display the addresses to the user, who will be able to choose an item by clicking the mouse on that item.

This requirement is sufficiently common that it should be implemented in a general fashion. For instance, you could write a function or procedure that handles a nonspecific set of choices through the use of a set of variables. The variables are displayed in some sort of list, and the one that is chosen is indicated in some fashion (like being highlighted) once it has been selected.

All microcomputer operating systems now provide users with sets of reusable calls to handle basic operations. The more advanced languages add more specific and elaborate tools. No matter how far the available tools go, however, chances are that you will need to create some of your own for your application.

The hard part is recognizing when a particular need is, or will become, a general need. And once it is recognized as something that should be designed for reuse, it is usually difficult to determine just how much flexibility it should support. Too little flexibility will require the code to be re-written. Too much flexibility makes it unwieldy and prone to error.

Designing reusable code is not easy, nor is it always called for. What you need to realize is that tool making is part of writing code. Common tools can be bought in the form of operating systems, programming languages, and language extensions. But even on a project-specific level, you still need to distinguish between code that can be consolidated by developing your own custom tool set, and code that is designed for a single use.

**DEBUGGING CHRONICLES #5: FAILURE TO CONSIDER THE NEED FOR FLEXIBILITY**
**PART B**

## III. Adding flexibility

### Analysis & Design

The need for flexibility arises from handling exceptions. In order to determine the need for and value of flexibility, you must have a complete understanding of exceptions.

Don't cut corners in your analysis of the proposed system! Whatever time may be saved in skipping seemingly minor issues could easily be lost many times over when the absence of an essential piece is noted later on. Cutting costs should not be an object at the analysis stage.

Imagine trying to save time in planning a car trip by using maps that did not show sufficient detail. If you equate the development project with the trip, and the analysis with the examination of the maps, then the need and the value of flexibility lies in considering the details of what connecting roads exist, what roads are under construction, and what opportunities might lie along the way.

As the system developer you must insist on a clear understanding of where you are going, how to get there, and what alternatives should be considered. If the client or supervisor fails to engage in, support, or pay for an adequate design effort, this should raise a red flag. It implies a basic disagreement or misunderstanding of their own best interests. If you are responsible for the success of your work, then you should take the initiative to resolve this problem.

Initiative is certainly needed to flesh out details. You may have to be insistent with clients about detailing the way in which exceptional circumstances are handled in their business. How these exceptions are handled can have major implications for the system you are developing. Be aware that various factors can work against your getting the information you need in a timely manner, such as the following:

- Exceptions are often handled piecemeal. Their processing is often difficult to describe, so they tend to be downplayed or marginalized.
- Users may equate your need to learn about special circumstances with their ability to control them. If they are not authorized to decide what to do in certain circumstances, they may feel they can't describe how such matters should be handled.
- You, or more likely the users being interviewed, believe the computer system is modeling only standard procedures, and consequently that

exceptions need not be explored. In fact, users may even have been instructed not to bother discussing exceptions by managers wishing them to remain focused.

- Time pressures, either yours or theirs, limit the discussion of exceptions.
- Discussions about unusual circumstances may appear less valuable than those concerning usual circumstances. If someone seems to feel this way, then remind them that the value of a process does not depend on how frequently it is done, but on how it contributes to the overall effort. In competitive environments success is often determined by small differences.
- People in insecure positions, or in situations threatened by departmental politics, often feel the need to retain control. They may be anything from reticent to hostile about giving a full description of how they do their job.

In probing for areas where flexibility could be valuable, ask the following kinds of questions:

- Is this the way processes are always handled? Are there any exceptions?
- Is this information always supplied? Are there any exceptions?
- Is the information that is supplied always taken from the same sources? Are there any exceptions?
- What information is modified after it is recorded, particularly after a long period of time has elapsed?
- Would certain operations now performed under pressure be handled with more flexibility if that pressure were removed? (For example, the time pressure to deliver goods, or to supply interdepartmental information.)
- Do any people other than those normally responsible for data entry ever access certain areas?
- What changes are being considered?
- What changes from outside the business might impact what is done or how things are handled?

The issue of flexibility should be addressed, either explicitly or implicitly, in every section of your analysis. Areas in the system where flexibility is needed, might be needed, or can be sacrificed should all be noted.

## Specifications

The specs express the user's needs in programmer's terms, and play a dual role in speaking to both users and programmers. As such, both user and programmer should be made aware of the flexibility intended and the means by which it is incorporated. This is important because it may be the first time users have seen a full description of how they handle exceptions.

The discussion of flexibility directed at programmers is aimed at helping them to coordinate actions performed in different areas, and to know the relative importance of related tasks. Since the specifications don't elucidate

every programming detail, programmers need to know where flexibility is important so they can emphasize it in the areas where they must exercise their own discretion. Conversely, programmers should know when they can code rigidly in areas where stability or performance, rather than flexibility, is warranted.

The specs should detail not only the current needs, but also the extent to which those needs might change, and how they might change. While features that are too expensive or of little consequence can be omitted from the specs, features that are almost worth implementing should be mentioned. This is important if the specs are to continue to play a role as a design document and not become simply a programming blueprint. The specs should function as a working document in the document scheme, as described in a previous installment of this series.

If flexibility leads to relations between functions in different areas, then the specs should clarify the relationships. For example, if cash receipts are normally handled by the accounting department but the system will also support cash receipts handled through the sales part of the program, then the specs should point out that these two areas are handling the same process. Failure to do this may lead to duplication, inconsistencies, and excessive programming efforts in relation to the importance of the function being supported.

## Consequences of Inflexibility

Perhaps more compelling than the benefits of a design that includes flexibility are the costs of a design that lacks it. Consider the problems that can arise when attempting to implement changes to a rigidly defined application:

Limited range of affordable changes:
There are limitations to the number of changes that can be made to data entry rules. Changes to the interface will become more difficult. There may be limitations on the kind of new reports or queries that can be added.

Maintenance costs increase:
Rules, functions and processes are harder to modify. Mistakes may be more costly and repairs may be less secure. When an application is coded rigidly, modifying it is more involved.

Changes to the underlying data structure may become prohibitively expensive:
Modifying the structure to include new types of items, as discussed below, might be prohibitively expensive if not done when the database is first designed. This may include adding new interdependencies between existing data.

Changes of this sort are quite common, both in development and as additional features are added. If you are not sure of exactly what the system is supposed to do, then adding flexibility is a kind of insurance.

## IV. Methods of Flexibility

The single most effective set of rules to follow in order to enhance a system's flexibility are those of data normalization. The most immediately accessible is the "Boyce-Codd normal form," which encompasses the following three directives:

1- Atomicity
The data stored in any field should be drawn from a single domain of simple (i.e., not compound) values. A given field should not store different types of information, or a concatenation of independent values.

2- Modularity
Data structures should be composed of logically independent files. That is, related but independent information should be stored in separate, related files.

3- Nonduplication
The data values stored in one file should not be derivable from each other. More precisely, to the extent that data are functionally related, the determining factors are stored only once.

These rules, which I've restated in the vernacular but which also have precise mathematical definitions, put the familiar concepts of atomicity, modularity, and nonduplication on a rigorous footing. Violating these concepts is the easiest path toward a rigid, nonextendable design. If it were only this simple, the Boyce-Codd normal form could be engraved in stone. Unfortunately such an approach fails in three respects:

1- Correct normalization is the first victim in the search for improved performance.

2- Normalization does not determine a unique "best" data structure. It should not be used to distinguish between alternative well-normalized structures each of which may lend itself to supporting different functions and extensions.

3- Normalization has little effect on the user interface or the manner in which data are extracted from the database, two areas that play a large role in flexible design.

We'll limit ourselves here to considering methods of supporting flexibility that go beyond normalization. The benefits of violating database normalization will be considered when we take up the topic of improving performance in the programming phase.

## Making Reporting Easy

Reporting is probably the single most common area where changes and extensions are requested. As such, it is important that the interface to the reporting functions be designed so that it will support users with various levels of expertise and a variety of purposes.

In a small application it makes sense to consolidate all reporting functions in a single area. But this becomes increasingly cumbersome as more reports and different types of reports are added. At some point a new area may be needed, and then the whole interface through which reports are accessed will need to be modified. Furthermore, it is easy to imagine that such an area originally could have been written by a programmer who is no longer working on the project, perhaps in code that is difficult to follow, with little or no documentation of his or her work. At that point what could have been a simple task turns into an arduous exercise in reverse engineering.

A much better approach is to segregate the interface so that it provides users with access to reporting in accordance with the tasks they perform. If the code is explicitly written so that new reports can be easily added, then it will not be a problem if the original programmer is no longer available when a new report is needed. The point is that if extendibility is included as an objective in the specifications, then conventions will already be in place when new reports need to be added.

## Customizing Data Entry

The great advance of the graphical user interface was to make it easy for the user to understand what the computer is doing, and easy for the user to control the computer. This does not mean, however, that there exists one interface that is best for all users. On the contrary, graphical interfaces can be designed either to limit the user or to provide them with far greater complexity than would otherwise be possible.

A common request, especially in complex systems, is for different data entry screens designed for users of varying levels of skill and responsibility. For example, the order entry screen for a sales clerk in a point-of-sale system must be simple, fault tolerant, and responsive. The same system may also support the entry of large wholesale orders for customers granted special considerations. Satisfying both situations would require two data entry screens, one simple and streamlined, and the other detailed and more complex.

Designing an application with various entry screens for much the same data requires some special planning. If you are going to impose similar data validation requirements in each case, then you should place the validation code in modules that can be accessed from the different areas. If each area needs to have slightly different levels of discipline, then this needs to be built

into the code from the start. What you want to avoid is installing code in different areas that performs many of the same functions. This is usually a mistake because it makes it much more difficult to modify and debug.

## Combining Files

The second of the Boyce-Codd directives tells us we should create separate repositories for logically independent data. However, it does not tell us how broadly or narrowly to define independence. The very concept of “related but independent” is somewhat of a contradiction in terms. It is common for a system to handle several related types of data which share many fields and processes, but which are nonetheless separate types of data.

Consider, for example, different types of sales contracts. In one case the price and terms of delivery of some commodity is fixed beforehand. In another type of contract the price is specified at market value at the time of delivery. A third specifies a minimum price subject to conditions. These different types involve substantially different quantities and types of data, but they are processed in a similar fashion. In addition, reports drawing on contract information may draw upon contracts of different types.

Other examples in a similar vein include types of inventory (such as components versus assemblies), or types of accounts (such as receivable versus general ledger accounts.)

In cases such as these it is a good idea to store all of the related types of data in the same file. Consolidating such items within a single file has some immediate consequences:

- The file created to store all the related items must be “overloaded” with fields in order to support all of the items being stored. This means there is a measure of inefficiency since some of the fields will remain unused.
- A field (or fields) must be added to the file to store a code value that identifies each type of item. This is an internal system code that must take one of several predefined, allowed values.
- Conventions must be defined to distinguish undefined values from unknown values (i.e., some solution to well known “missing value” problem.)
- Multiple entry screens may be needed, one for each type of item being handled. On the other hand, if the items are sufficiently similar they may be handled through a single screen.
- Displaying or printing lists requires screening combined selections to filter out the types of items that are unwanted. This can be quite a burden, especially considering the confusion that could arise if this filtering should ever be missed!

Storing related types in a single file generally enhances flexibility in the following ways:

- It is easy to add, modify or remove processes and screens for handling other types of data.
- There is an immediate reuse of code since all types of items will be handled through the same (or similar) lists and entry screens, and through calls to the same controlling processes.
- By consolidating related data and related actions in a single area it is easier to modularize the application. One of the most important consequences of modularization is that it allows areas of the application to evolve independently.

In enumerating these strengths this list also provides guidelines that can be used to determine the value of a consolidated data structure. Consider storing related types of data in a single file when:

- You may need to change the definition of the item types, either by equating types or generating new types.
- The code is complex or extensive. In this case code reuse would lead to substantial savings.
- The types of items that are being combined define a functionally contained unit to the extent that the combined data can be associated with a quasi-independent module.

## V. Summary

Taking the time to design flexible code may add 10 to 30% to the initial cost of the project. It depends on the project, the kind of code you normally write, and the degree of flexibility you want to achieve.

Your objective should be to identify the most likely future needs, evaluate their benefit, and determine a means for including them in the project. If you are working from complete specs and in the habit of writing code that is maintainable to begin with, then adding a degree of core flexibility should be easy to justify. The initial extra expense can save time, money and unnecessary frustration in the long run.

**DEBUGGING CHRONICLES #6: PROGRAMMER INEXPERIENCE**

Problem: Assuming that you will learn as you go along generally means that you do not appreciate the risks involved in such an approach.

## Part A

### I. Introduction

Just as a person with a hammer tends to see every problem as a nail, so a person with specialized skills in computer programming may see application development as purely a programming problem. This is a common state of affairs since most developers of desktop systems are self-taught, and since most small systems efforts minimize the problems of analysis and design. This frequently results in a lack of awareness about the true scope of system development. In addition, developers are often encouraged, or even required, to retain this myopia by managers and clients who have bought the vendor's promise of easy computer solutions.

This is the situation I refer to as programmer inexperience. It does not necessarily imply a lack of skill as a programmer, but rather a lack of skill in the broader realm of application development.

Here I consider two ways that such a lack of skill can affect a project. The first entails failing to perform adequate analysis; the second involves trying to skip the design process by relying instead on a process of successive revisions. I'll call this "design by trial and error." You may recognize these symptoms in projects of your own, or you many recognize their effects in the difficulties you experience extending your efforts to the creation of larger systems.

### II. Failure to Perform Adequate Analysis

The failure to perform adequate analysis is usually rooted in miscommunication and misconception. Misconception enters through a developer's belief, usually erroneous, that he/she fully understands the situation that the client has described. This is often compounded by the client's difficulty in effectively communicating all that is needed from the system.

These failings and the problems they engender can be circumvented by exploring your understanding and testing your design ideas. Such exploration and testing constitutes "analysis." Performing analysis in a thorough and disciplined manner is as important to a successful development effort as writing good code, yet many projects proceed without developers subjecting their thinking to exploration and testing.

I believe that the underlying cause of inadequate analysis is the developer's failure to recognize and assume responsibility for their role as consultant and

advisor. Developers often perceive their role as simply that of translating the designs of their client or manager into software. Consequently they cast what they hear into operational terms that they understand. They either ignore what they don't understand, or expect it to be resolved at a later stage.

The key to correcting this failure is in realizing that as a developer, you have no choice but to accept the role of consultant and advisor. The bottom line is that if something goes wrong and the project suffers setbacks or outright failure, **you** will be held responsible. Developers can either learn this lesson through experience, reflection, or some combination thereof. If you are not sure about how much responsibility you should accept, or about whether you should expand your role beyond that of programmer, consider some hypothetical examples of the risks and opportunities that may await you:

### Failing to act as an advisor

#### Situation 1: No Budget

The client assumes a low cost and is not led to believe otherwise because there is no budget.

#### Consequence

The client stops paying when costs escalate. You blame the client for adding features without being willing to pay for them. Development stops. The client feels misled and cheated.

#### Situation 2: Incremental Development Without An Overall Design

The client assumes that you will develop an application incrementally, starting with the most basic functions and adding details and other related functions. You have no reason to discourage this since there has never been any comprehensive analysis. You might tell yourself that such an analysis could discourage the client and put the whole project in jeopardy. Besides, you might reason, it would cost more than the client would be willing to pay.

#### Consequence

The initial software is inadequate and buggy, and the changes required take more time than anyone expected. Knowing that trouble is brewing, you present the client with a large bill, explaining that you have only billed them for half the time you have spent, which is true.

The client is shocked by the bill and offers to pay it only if you agree to complete the remainder of the project at fixed cost. Not knowing exactly what they want, you refuse to make such a commitment. The client does not pay, and instead sues you for breach of contract.

#### Situation 3: Attempting to Install a Prototype

The executives in your small company want to implement a comprehensive system and call on you, as their IS person, to design and implement it. You proceed to create prototypes built to satisfy their continuing stream of requests. You expect to incorporate all requests incrementally until you have a finished system.

Consequence

After 6 months, management becomes anxious and wants to know when the system will go on-line. You respond that you're still adding features and you expect to be finished in another 6 months. Twelve months after starting the project you present an incomplete application in which minor bugs appear. You tell management that you couldn't implement all the requested features because the requirements kept changing. After another 3 months the same thing occurs.

Management decides to put the system "on hold" and charges you with the task of finding a preexisting system. You locate several partial solutions and management decides to implement one, over your recommendation. The new system's vendor supplies technical support and training. You're demoted to a technician's role.

## Acting in the role of an advisor/consultant

Situation 1: Working With a Budget

The client assumes a low cost. You tell them that the cost can only be assured through a fixed bid, which requires a full specification. You explain that any changes to the initial specs will result in additional costs. You elaborate the benefits of the specification process in assuring a functional, maintainable, and well-integrated application. The client accepts the proposal and the extra cost of the initial specifications. This adds 30% to the initial cost of the system.

Consequence

The specification process helps to build the client's trust in your abilities, and you respect the client's ability to learn quickly and make decisions. The initial design work introduces substantial revisions and an expansion of the original proposal. Other changes are implemented in the business operations to simplify the system and to allow it to play a more valuable role. The application is completed with minor cost overruns, but since the client has been involved throughout, the overruns do not jeopardize the project.

Situation 2: Incremental Development With Staged Specification

The client wants an application developed incrementally, starting with the most basic functions and adding additional details and functions later. You offer incremental implementation only on the condition that each phase is

specified in advance, estimates are non-binding, and you don't provide guarantees as to the cost or feasibility of the unspecified subsequent phases.

Consequence

The client accepts your proposal. Since they pay for your design expertise, they take your advice seriously. You suggest that certain functions be handled in the early phases, and that others be deferred until later. This results in some reorganization that allows the new software to be used more effectively. Because of cost concerns the client elects to halt development after the second revision. You remain in their employment to install and maintain the application and develop future plans. Eighteen months later, with the initial system working to their satisfaction, they proceed to a third revision.

Situation 3: A Careful Design Effort

The executives in your small company want to implement a comprehensive system and call on you, as their IS person, to design and implement it. In response to their disorganized stream of requests you begin a systematic design process. It turns out that the executives don't have a clue as to how the users should or do perform their jobs. You spend a lot of time interviewing users and managers in the manufacturing, sales, and accounting departments.

Consequence

Your design is completed in 3 months, and the basic data and application structure is in place when the executives call you in for review after 6 months. They are surprised that the system isn't in place, but it is obvious to them that you know the company's operations better than they do. Appreciating your careful effort and not wanting to appear foolish, they accept your projected completion date and plan for staged installation. They congratulate each other on their foresight in picking the right time for automation.

The system is completed and implemented to the satisfaction of the users. Clearly pleased with your work the executives offer you the choice of continuing in your present capacity or accepting a 30% pay raise that accompanies a promotion to a non-technical management position.

## III. Designing by Trial and Error

You can't entirely blame the person with a hammer for looking at all problems as nails if, in their experience, all problems previously have been nails. Many programmers have only handled programming tasks, tasks that required straightforward programming. The initial mistakes of programmers new to system design are explained by the hammer metaphor. They are not to blame for all their mistakes.

When a programmer is hired as a system architect, the blame for the resulting problems should be shared by those responsible for the error. I think that some of the blame falls upon sources from whom we normally take reliable counsel. Naturally we would blame the programmer who claims to be competent in designing database systems when in fact he or she is not. The client / user might also share some of the responsiblity for hiring the wrong person for the job or, to put it another way, of not understanding the qualifications required. In addition, schools that produce programmers are responsible for adhering to outdated curricula that claim to teach students modern system design. I have yet to hear of an adequate academic program. Software vendors are to blame for misleading users and service providers. Vendors strip every hard-earned lesson we've learned about the requirements of software development out of their promotion and training in an attempt to convince everyone that they're selling the "silver bullet" for successful development.

Having distributed blame we may now consider various solutions. In the best of worlds all parties could be educated to see and correct their actions. However, in the real world schools and vendors both have a vested interest in their products. As long as the economics that affect their decisions remains the same, change will be seen to bring more risk than opportunity. That leaves the client and programmer.

The first thing to be said to inexperienced clients and programmers is that they should consider the advice given by academic authorities and software vendors as incomplete and self-serving. The best way to make an initial judgment about a programmer's ability is on the basis of his or her past work in system development.

The best way for a programmer to become competent at system development is through practice, participation, and self-education. On-the-job training is not adequate by itself. It is too narrow and too short-term.

Debugging Chronicles #6: Programmer Inexperience
Part B

## IV. Fallacies of System Development

How can we convince the hammer-holder not to use a hammer to tighten a nut? How can we convince the programmer that programming per se cannot solve all problems in system development? One way is to list the common misconceptions that programmers have about system development and explain why they are wrong:

### Common Fallacies of Inexperienced Programmers

1- Clients understand their own needs.
2- Powerful people are knowledgeable.
3- Designs can evolve incrementally.
4- The programmer will find wide support in the organization.
5- The programmer's skills will be appreciated.
6- The programmer understands what the user wants.
7- The programmer understands what the user needs.
8- The programmer knows the best tools for the job.
9- The programmer's work will prove itself.
10- The client wants to collaborate with the programmer.
11- The programmer is not responsible for the system's ultimate success.

Some of the attitudes listed above may not be false in your particular environment. But very often they prove fallacious, particularly when taken for granted. While these attitudes seem most common among inexperienced programmers, they could also be shared by more seasoned individuals who have led sheltered working lives. Note also that if you are in the role of the user or the client and you blindly accept these beliefs with respect to your project and your programmers, then you may be contributing to present and future problems. In the best of worlds these attitudes might all be true! However, none should be assumed. If any are true, however, it is because the client, the users, and the design team have worked to make it so.

### Fallacy 1 - Clients understand their own needs:

The system you are asked to develop is self-consistent and consistent with the client's (or user's) needs.

### Fact:

Computer systems are rarely successful if they are designed to directly automate a non-computerized system. Clients who ask for a system based on their previous non-automated model usually don't know what they are doing.

The other extreme is the client who plans to completely change the operations of their business to suit their concept of a comprehensive system.

This is the “send me to the moon” approach, named for the client who requests that the software support the most complex, conditional operations at the touch of a button. This is basically a feature request that is both impractical and unaffordable. Such requests are common when they come from clients unfamiliar with the development process.

Only in the case of an experienced client — one who has successfully participated in system design project — can a programmer begin to assume that the client understands their own needs. Even then, it remains the system designer’s job to ensure that the client’s needs are being met.

### Fallacy 2 - powerful people are knowledgeable:

The higher the level of the person providing direction, the more well-informed will be their understanding of the system.

### Fact

As a rule, the people who best understand operations are those who actually perform them, or who are directly responsible for their being done correctly. The higher you go in the chain of command, the less informed the individuals will be when it comes to operations. Executive management may be both out of touch and misinformed about what goes on at lower levels. This derives in part from being distanced from on-the-ground operations, and in part from the tendency of subordinates to retain their power by withholding information from those above them.

System design can become stuck in situations where the people with greater authority are uninformed: the ones with the power to make decisions don’t know, and the ones who know don’t have the power to make decisions. In these situations the client must appoint someone with both knowledge and authority to act as their representative.

If the client is already experienced with computer systems and system design, then such a representative will exist in the organization. They might be the system administrator, the technical expert, the office manager, the bookkeeper, or the CFO. As a programmer you will recognize this person when you meet them because they will speak both your language and the language of the users. If such a person does not exist, then it’s your job to see that such a position is created.

### Fallacy 3- Designs can evolve incrementally:

Changes can be made to an evolving system through a process of incremental modifications.

### Fact

Designing by incremental modifications means introducing elements that were not in the original plan. The incremental approach is most suited to the

hacker and the tinkerer. The “tinkering” approach to system design follows from the fact that many who are attracted to computing have a knack for tinkering.

Our society applauds the teenager who hacks her way into the Defense Department’s computer system, calling her a “computer wizard.” But hacking is really just a form of trouble-shooting, and in no way represents general computer skill. Hacking is best reserved for systems that are not working as they’re designed to, or more commonly, systems that were never designed correctly in the first place!

In contrast, we want to design systems correctly, and create systems that work as they are designed. Ideally there would be no need for trouble-shooting. In an ideal system the hacking approach would be equivalent to beating down a door when all one really needed to do was turn the knob.

Designing by incremental modification reflects a process in which someone isn’t seeing the forest for the trees. That someone may be the programmer, the client, or a person responsible for another crucial aspect of the system.

Let’s consider some basic reasons why the incremental approach is flawed. All systems are built on some model of how information is structured.

Information structures can be represented as one or more trees, whose basic concepts are the tree trunks and whose subsidiary concepts are the limbs and branches. The model you choose determines the sequence in which one concept, and hence information, builds on other concepts.

The facility with which you can change an application is related to how far up this tree your changes are made. Changing or adding to the high-level information, information that uses pre-existing concepts, is relatively straightforward. Changes to lower level concepts, upon which the already coded application depends, is much harder.

You need a complete picture of the system in order to explore its logical structure, and even with a complete picture you must still make a special effort to analyze its dependencies in this manner. You cannot perform this analysis on a system that evolves without any larger logical context. Consequently you have no way of knowing what changes can be made incrementally and which will require extensive revision.

In many cases where evolution is attempted by inexperienced designers the design begins with one of several fundamental areas and progresses to add or link together other, sometimes altogether new basic concepts. This is especially likely in a methodology that is based on the automation of forms, and proceeds by considering successive underlying levels of information processing.

Unless system development is carefully planned to proceed from the bottom up, chances are that it cannot be evolved through a series of incremental modifications.

**Fallacy 4** - The programmer will find wide support in the organization: Others in the organization will support you in your work.

### Fact

As mentioned in Debugging Chronicles #2, the support you will receive is proportional to how positive a role your work is perceived or expected to play in the lives of others.

In a climate of corporate downsizing everyone may be trying to protect their own turf. This may even apply to growing organizations with the resources to develop new systems. The system may be intended to relieve overburdened personnel, or to consolidate (i.e. remove people from) existing departments. It would probably help your position to explain that the system will make people's work easier and their efforts more successful, if this is indeed true.

Beware of situations where the development effort is overseen by, or contingent upon, the approval of more than one person. These are unstable situations where strong political undercurrents can develop. One thing that is certain is that a new system heralds change, and change is generally viewed with suspicion even when it is a consequence of successful growth.

Ideally there is support from all quarters, but in reality there are usually varying degrees of support from different sectors. If support was completely lacking the project would never have been started in the first place, but just because a project is started does not imply that it has the support necessary for its success. As system designer one of your first tasks is to clarify the lines of influence, to determine who are your supporters and advocates, and to convert or isolate those whose influence is negative. If you can't line up your supporters in any official capacity, then you should at least determine who your friends are.

**Fallacy 5** - Your skills will be appreciated:

The client will understand and the fruits of your labor. And in those cases where they don't, then they will at least appreciate your efforts.

### Fact

The ideal, experienced client will work with you like a senior partner. More commonly, however, the client will not have much previous experience in computer systems. Some of the confidence they may have in you will be based on unjustified expectations. They may think that you are a genius because you can program and sprinkle technical terms throughout every discussion. They may expect that you will be able to fix everything and that their troubles, whatever those may be, are now over.

The extent to which the client-designer relationship is based on unjustified expectation is the extent to which the relationship will probably suffer disappointment. This applies to both parties, of course, but as the person

responsible for the project it is your job to ensure that the expectations necessary for the project are reasonable.

In this regard it is important that you clarify the responsibilities of all parties. This includes specifying the resources each party must provide, how responsive they must be, and how the project will be impacted when these needs are not met. Some of these issues are contractual, such as billing, payment, and ownership rights, but many issues are of a more procedural nature.

Consider the following representative examples, which are not meant to provide a complete list:

a - Client must provide expertise

The client must be able to promptly resolve questions regarding business operations, or else development will be delayed.

b - Client must approve specifications

The client must review and approve specifications for a given area before the programming on that area begins. In this manner, you or the design team, if there is one, cannot be considered at fault for programming features that are incorrectly specified.

c - Client must make a financial commitment

In the case where you work as an outside contractor, you rely on the project for a continuous flow of work according to the agreed upon schedule. If the project stalls or is delayed for reasons beyond your control, then you may have to lower your commitment to the project in order to secure other work.

d - Your expertise is limited.

Your expertise is limited to those systems that you are acquainted with. On many issues you may have no firsthand experience at all. Extra efforts must be made to test the effects of unusual circumstances. This may mean calls to technical support, or writing special programs to test the inter-operability of special hardware.

e - Your responsibility is limited.

You cannot accept responsibility for things over which you have no control. If there are circumstances that might negatively impact on your ability to participate, then you should make these clear. Similarly, you should determine if there are contingencies that could change the client's role in the project.

**Fallacy 6** - You understand what the user wants:
Your conception of value is consistent with that of the client and the users.

### Fact

Programmers do not usually have the same values as users with regard to the work that needs to be done. Where a programmer will value more flexible options, a user will often prefer less flexibility. Where a programmer prefers to code things concisely and efficiently, a user could not care less; to the user, efficiency may just mean getting the job done and going home as quickly as possible.

When it comes to the concerns of a particular user, the inexperienced designer tends to approach his or her problem as one of data management. Such a designer may ask, "How can this user be given the most power with the least effort?" Many users, on the other hand, prefer to have as little power as possible, since less power means less distraction, less responsibility, and less chance of error.

Users are often mistaken when they envision the system handling their most common actions, those they perform 95% of the time. If they realized that the system will be used to handle **all** of their work, then they might appreciate the need for power and flexibility.

In many cases the programmer really does understand the user's situation — the programmer does have the clearest idea of how the system should best operate. But this assumes that the user is free to make use of the system, and that the system is to be designed to best satisfy the user. The trouble is that both of these are unreachable ideals to some extent.

The user is not free to make full use of the system. The system will probably introduce unpredictable changes in work patterns and preferences. The user will react to your design in the context of the current environment, which contains complication, duplication, time-pressures, oversight, and other annoyances. You cannot expect users to respond to your design from the unknown perspective of how they will be working in the future.

Systems cannot always be designed to best satisfy the user. There are many constraints that limit what the system can do. These include time and cost restrictions, restrictions on complexity and requirements of data verification and user oversight. In addition, there are many "good" ideas which are either too expensive or not sufficiently compelling.

An example is the User Interface Guidelines, published by Apple Computer. Many programmers consider these guidelines to be rules. They fail to appreciate that the guidelines were developed for general purpose products sold to retail markets for the benefit of the consortium of Apple Computer-related hardware and software vendors. In the context of custom system development, these guidelines change from being "rules" to "suggestions."

Whatever the origin of the differences, the programmer must be fully informed as to the full range of the user's needs. When the programmer's vision of the system diverges from that held by the user, then the programmer must sell his or her approach. Sometimes the most concise and efficient code is not what is most valuable to an organization at the current time. Ultimately the client calls the shots.

**Fallacy 7**- The programmer understands what the user needs:

Your understanding of the proposed system's technical requirements accurately describes what the client needs.

**Fact**

The fallacy here is not that you don't understand the system you are designing better than the client — you probably do — but that the system you are designing is likely to be the wrong system! It is fair to assume that the client, or the people involved in performing the actual work, will always know their operations better than you will. They may not explain them clearly, and they many not approach their tasks in a manner that makes the most sense from a computer design perspective, but helping them to accomplish their work remains one of your primary objectives.

For example, consider two systems designed to do the same job. The first is a sophisticated, efficient and robust computer system that eliminates redundancy and supplies users with just the information they need. The second is a manual, paper-based system using hand calculators and triplicate forms. The computer system may have been designed to replace the manual system, but if the users reject it, then it is a failure. The manual system, in spite of its inefficiency, can keep the business afloat and may even have advantages of flexibility and low cost.

The objective of system design is to minister to the user's needs. Remember that the users are part of the system: they are the agents that make the system work. Whatever their limitations, they generally cannot be replaced with more efficient components.

**Fallacy 8** - The programmer knows the best tools for the job.

You will have fewer problems if you know and use the best tools for the job.

**Fact**

In this case "best" is a relative term. It depends on what you're trying to accomplish. If your objective is not considered valuable to begin with, then doing it better will not add value. The statement also implies that the tool is partially responsible for doing the work. To some extent the tool deserves credit, but if you are an inexperienced user of a powerful new tool, then you'll probably produce work of relatively poor quality.

Every project has different needs and involves different people. The tools appropriate for a presentation to an IS department will be very different from those for a report to a non-programmer. The former group may be fluent with structure charts and entity-relation diagrams, which will do more to confuse than edify if shown to the non-programmer.

In certain areas the quality of the tools makes a great difference. You should have those tools that you know you need, and you should learn to use tools that will improve the speed and quality of your work. But this is quite different from attempting to incorporate every new bell and whistle that comes down the pike.

In my experience programmers display an excessive reliance on their tools, whether those tools are faster hardware, extensions to their programming environment, or a new Computer Aided Software Engineering (CASE) system. In most of my projects I look for reliability and completeness in programming tools and simplicity and the integration of documentation and presentation capabilities in design tools.

I will further discuss tools in Debugging Chronicles #2.7: “Failure to Use Appropriate Tools”, and #3.1: “Failure to Use Available Tools”.

Before choosing your tools you should break the project into its phases and identify the participants and the objectives in each. Choose your tools accordingly.

### **Fallacy 9** - The programmer’s work will prove itself.

The more work you do on a project, the happier and more committed the client will become.

### Fact

This fallacy is based on two assumptions. The first is that the client’s main concern is the pace of the effort. The second is that the client’s commitment will greatly increase once they see the wonderful things you have accomplished. This is the perspective of a programmer whose success in the past has been in a carefully prescribed and monitored environment.

By now you are probably aware of just how shortsighted these assumptions are. I’ve witnessed projects where the more uncertain the client became, the harder the programmer worked. It reminded me of a painfully bad, amateur comedian who won’t stop trying. In just the same way these programmers never realized that **they** were the problem and their misdirected efforts only served to hasten the demise of the project.

A client may be shocked by the cost of your work and dismayed at the slow progress, but this does not mean that working harder or offering price breaks is the solution. The real threats to a project are rarely technical. They more likely stem from confusion, mistrust, fear, uncertainty, and doubt.

The Achilles' heel of most technical people — and programmers fall squarely in this group — is their lack of skills in listening, understanding other perspectives, and making themselves understood. These are not necessarily fatal flaws, but they can quickly become so when overlooked or denied, especially in a crisis.

If a project is in difficulty, and even before there are signs of difficulty, the system designer must be working to open lines of communication. This means presenting progress reports, soliciting feedback, listening, and trying to understand and address all potential problems. System design and implementation is largely a service business, and in service businesses the client's satisfaction is paramount.

**Fallacy 10** - The client wants to collaborate with the programmer:

The client is interested in important technical issues and is interested in helping you to decide among alternatives.

**Fact**

The client most certainly is **not** interested in technical issues. The technical issues are the annoyance that you are hired to remove. A client trying to manage a business is no more interested in your discussion of system development than you would be interested in a plumber's discussion of a valve fitting in the event that your pipes were frozen.

Your job is to translate technical issues into familiar and relevant terminology, to give clients a good reason to work with you and to make their choices as clear, easy and direct as possible.

**Fallacy 11** - The programmer is not responsible for the system's ultimate success.

You are not responsible for the success of a system once you've programmed it to satisfy the design requirements.

**Fact**

As a system designer you are hired to provide a solution, not a piece of software. The software is just the mechanism used to effect the solution. A system designer must be an advocate for successful implementation. As an outsider, and especially an outsider with the power to modify the application, you can be sure that if anything major goes wrong it will be in the best interests of the others involved to assign you the lion's share of the blame. You may think that this is unfair, and from an objective point of view it may be, but your job does not offer the protection of such objectivity.

One of the frequent sources of misunderstanding and difficulty in systems designed by inexperienced designers stems from their failure to fully appreciate the magnitude of their responsibility. It is natural for an

inexperienced person to want to avoid responsibility, especially in those areas where they are least experienced. Unfortunately, as far as the development of small systems goes the client is often equally inexperienced and even more unqualified to accept responsibility.

This is one of the reasons why people new to the field can least afford to cut corners or offer "low-ball" cost and time estimates. When a neophyte prices a job they should understand that they must estimate the cost of a **successful** project. Their estimate must be based on the extra time and effort it will take for them to get it done right. A lower estimate should not be coupled with the project's higher chances of failure.

This is not to say that every first project should succeed, or that programmers should know all that they will need to know before they try their hand at system development. On the other hand, if you are not ready to accept the full responsibility for the design, development and installation of your system, then you should do yourself and your client a favor by not accepting the job — at least not without training or assistance. In spite of this admonition most small system developers seem to learn the trade by playing a leading role in one or more complete disasters.

## V. Summary

While I invented the scenarios in section II, I built them from real reports and personal experiences. The implication that simply insisting on specifications early on will save a project from failure may be exaggerated, though not completely far-fetched.

The real implication is that if you are so irresponsible as to skip specifications, then you will probably fail in other areas where your input and action will be needed. These are the Fallacies of System Development listed in section IV.

If you take responsibility to protect the project by insisting on specifications, then you will probably also protect the project by writing a contract, following a budget and schedule, and keeping the client informed of your progress.

These are some of the other steps needed to ensure a project's success, and they will be dealt with later in this series. One thing all these steps share is that the developer takes responsibility for the success of the project. The developer accepts a central role in mitigating risk.

**Recommended Books on the Software Development Process**

1) Eckols, S., 1983. *How to Design and Develop Business Systems*. Mike Murach & Assoc.

2) Humphrey, W., 1995. *A Discipline for Software Engineering.* Addison Wesley.

3) Ledgard, H., 1987. *Professional Software*, volumes I and II. Addison-Wesley.

4) Rakos, J., 1990. *Software Project Management for Small and Medium Sized Businesses*. Prentice-Hall.

5) Weinberg, G., 1993. *Quality Software Management*, volumes I, II and III. Dorset House.