Sea's Practical Guide to Software Development

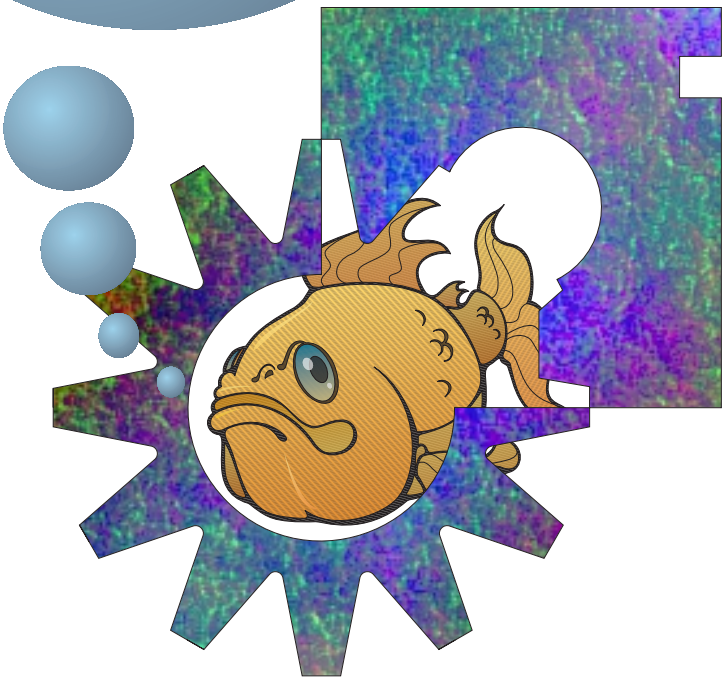

Table of Contents

Introduction

Designing the Product

Implementation

Testing and Delivery

Post-Release Activities

1

Introduction

Like many other types of products, software products are built by using sets of components (commands), which eventually comprise the product itself. The result is not a piece of machinery but rather a computer file which instructs the computer running it what to do, when, how, at what sequence etc.

As a product, it is more like a process management tool. The desired outcome of this product is a specific process definition.

When building a house, a chair, a car or similar products, the performance characteristics are often clearly visible (and I am not implying here that they are always easily achieved). One can measure the height of a wall, the thickness of a chair's back and the capacity of an engine. "Measuring" software can be just as easy, if proper tools and methods are used.

Software products have "hidden" attributes, such as background processes, functions and underlying operating system's activities to

consider. They might respond differently for different users, on different machines or when running with different versions of the operating system for which they were designed. Often development teams need to build into the software allowances for those unknown factors, for which they are not responsible and on which behavior they have little to no influence.

Background

Books about software development, methodologies and testing have roamed the market for many years. Every year a bunch of new ones is crowding the stores' shelves. The question is — why bother? What can this one give you that others don't?

First, I promise to do my best not to be boring.

Second — this is supposed to be the practical guide to software development. As such, we will try to pinpoint specific processes, requirements and outcomes of a software development effort without getting buried under a pile of paper.

If we look at most software development methodologies, you will notice one significant outcome: many reports, specifications and other very important documents are usually outdated when they are signed.

However, for the product to be successful, we need a set of objective criteria. That means we would need a certain level of documentation. The idea is to concentrate on the real task of developing a software product, while maintaining a close-loop, and efficient documentation management process.

A portion of the readers may consider some methods and views expressed in this guide unorthodox. Processes are skipped, a few are short-cut and others are emphasized. Even if you do find it hard to accept some of the methods expressed in this book, you may still gain a new perspective and expose yourself to another line of thinking.

You don't need to be big in order to "think big". Many methodologies will accommodate a large software company but will crush medium to small ones due to overhead. At the same time, many examples in the history of computing show that some fairly small software companies were very successful in developing big products. The largest companies in the world started as a bunch of graduate programmers with a brilliant idea or two.

The processes described in this book can be easily managed by a development team of any size (as long as you have the resources to actually develop the product). The basic assumption is that the smaller the team — the more "hats" each member of it needs to wear. There is no harm in one person performing more than one function, as long as any conflicts of interests are known and addressed.

Key element for the success of this process is flexibility. You may need to cater for various sizes of development teams, and phases should be implemented according to project's complexity. Applying a defense system development methodology to a home cookbook software is a bit of an overkill. "Cover everything but do not get carried away..."

The process works. It has been fine-tuned for years, and the results speak for themselves. Software products, on various platforms, that were developed while following these guidelines were free of defects, reasonably priced and always on or ahead of schedule.

What's more — the authors of this guide do not believe that error-free software is a non-achievable goal. As will be discussed in the next sections, software products can (and should) be virtually error-free (i.e. without noticeable "bugs").

Target Audience

Essentially, everyone who is involved in developing software will benefit from leafing through this guide. It covers all aspects of software development, beginning with the initial idea and ending with the product being used, maintained, enhanced and, eventually, being phased-out.

If you are already experienced with software development projects — you will find some simple and fresh ideas in this book, and probably some solutions to common problems. If you are new to the field — welcome. Read this guide and give yourself a chance to do things right the first time.

This book will not address programming issues, such as algorithms, development tools, languages or structures. There are enough publications targeting these subjects. It will concentrate on managing the process of software development. Therefore, it can accommodate anyone developing software on any platform.

Meeting the Standards

We all know that there are some standard requirements regarding software development, quality, configuration management and the like. These may be ISO, IEEE, ANSI, US MIL-STD and similar documents.

How does the methodology in this book meet those requirements?

The answer is simple: it will definitely meet the most demanding ones, for the following reasons:

- success criteria are clear
- the process is well defined

- the outcome is verified
- every step is documented properly.

One thing we have to keep in mind during the process of development (of any product), is the level of confidence we are required to provide to the customer. This means that some customers (depending on the type and criticality of the product) will trust you to develop the product properly and keep your files neat and tidy. Others will require that you continuously prove to them that you are doing so.

Running an organized and properly documented process will enable you to gain confidence in your own products. The rest will follow.

Creating Error-Free Software

As mentioned before, I strongly believe that when a software product is developed properly it can be operated without problems (i.e. with no “bugs”).

Later in this guide we will discuss different types of software errors and their effect on the user, the product and the developers. We can, however, argue that no matter what we do and how we test the product — there is always a chance that we miss something.

There are also problems that are beyond our control. The product we develop is not floating in the air but is rather relying on certain assumptions involving the hardware (the computer it is designed to run on), the firmware (the software which is saved on the computer's processors), the operating system, the compilers we use etc.

One assumption we may have to make is that the hardware (our's and the users') is in nominal condition. Another one is that all underlying software and firmware, which came with the hardware or was purchased off-the-shelf, might contain errors of some sort.

As shown in figure 1-1, we can represent our end product as an inverted pyramid. Each layer is a product that supports, creates or services the product we develop. Every problem manifests itself exponentially in the next layer and eventually we get a product which might contain not only the problems that we have built into it, but also the projected problems from the underlying layers.

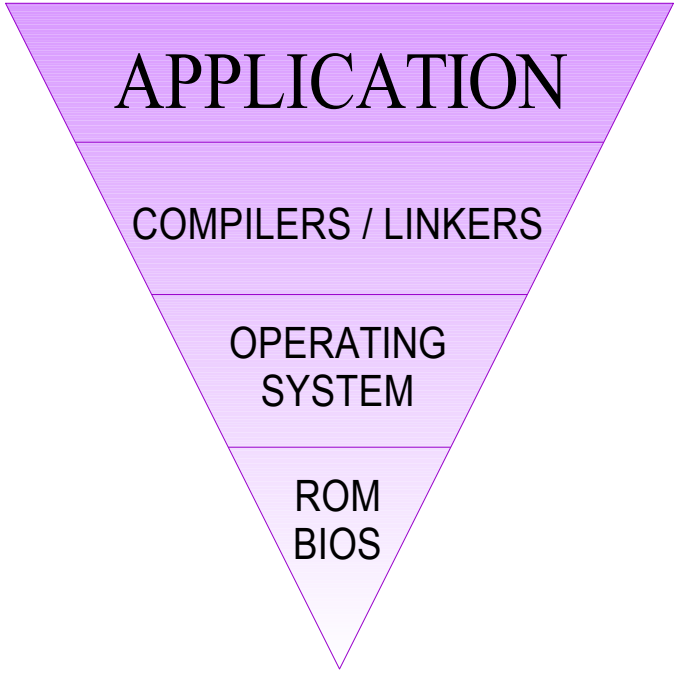

Figure 1-1: Inverted Pyramid of Errors

Allow me another assumption: so far as the end user is concerned, any problem in the product was caused by the last manufacturer in the line towards accomplishing the application (top layer), i.e. — you and me. Many users are not information technology experts, and would not appreciate excuses such as "it is a problem in the operating system."

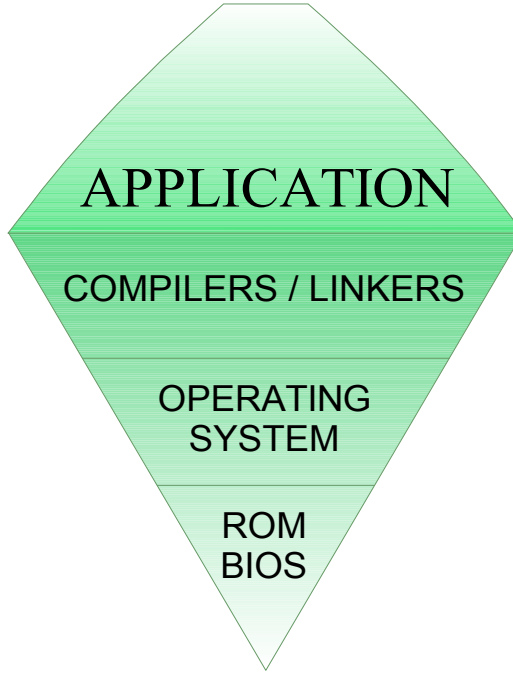

Figure 1-2: Compensating for Errors

Somehow, along the software development process, the team must assume responsibility for the whole pyramid. Figure 1-2 demonstrates how a good development and testing practice can compensate for high and low level problems. Even if we cannot debug or recompile the underlying software we may still be able to bypass its errors.

The Structure of This Guide

This guide is a source of on-line information. You are supposed to have it handy for continuous reference. To make your life simpler the guide is divided into chapters which logically follow the process of software development. Figure 1-3 illustrates the block diagram of a software development process, as will be followed in this guide.

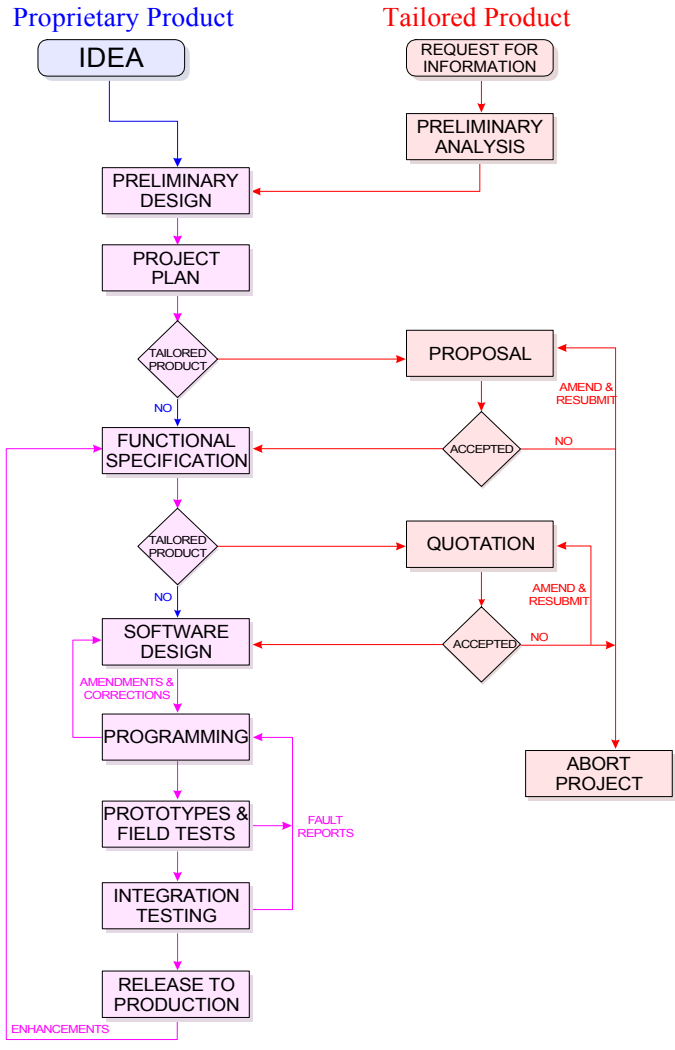

Figure 1-3: Example of Development Process Flow Chart

For each phase during the development process there is a dedicated chapter. Everything you need to know and do at that stage is detailed there. This guide deals with the “what”, and, to a certain extent, with the “how”. It avoids discussions regarding specific implementation tools, therefore it may accommodate any platform. Concepts, approaches and structures of information will be discussed, leaving your team with the actual choice of a particular tool.

2

Designing the Product

This is the initial phase of any product development process, and software products are no exception. Actually, one might say that there is even an earlier stage, which is the idea itself.

Since the line between “dreaming the product” and the beginning of outlining its design is very thin — we will combine the two and incorporate them into the design phase. We will call this phase “Preliminary Design”.

Careful design of the product at early stages is vital to its success. If we plan accurately, we will avoid delays, misunderstandings, re-testing and patchy programming. Careful design will also significantly enhance our capability to service the product during its life cycle.

The types of activities carried-out during this phase depend mainly on whether we are developing a shelf-item on our own initiative, or we are providing someone with a tailored product.

It may be a mix of the two, if we are modifying an existing product to suit specific needs. In this case, the preferred way is to explore the possibility of enhancing the shelf-item product to suit a larger variety of target customers.

Preliminary Design

This is what the project's originator comes up with. It may be an external source, such as a potential customer, or an internal one — a team member with an idea for a new / improved product.

The process is distinctively different for each case. While a proprietary product gives you full control over the rhythm of various phases of developing the product, some of the phases when dealing with an external customer are dependent on its approval and cooperation, which are not always forthcoming.

The outcome of the preliminary design process is an outline of the product's high-level functionality. It is preceded by a definition of the desired product. This definition is descriptive enough so that after reading it the reader understands what it is all about.

For example, the product definition for a banking system may be something like this:

A software system that will connect a server machine to local and remote terminals, and will centrally manage accounts, transactions and corporate's administration.

The design outline is a bullet list describing what the definition comprises of, and may look like that:

- Open accounts.
- Change accounts details.
- Close accounts.
- Deposits.
- Withdrawals.
- Interest calculations.
- Transfer between accounts.
- Transfer between systems.
- Cash-flow management.
- Human resources' management.
- Assets management.

This is, obviously, not a complete list, but it provides a good example of how the preliminary design should look like. Each of the above items will be expanded upon during the functional design phase. Some items may become sub-systems while others will be functions within a sub-system or shared among several modules.

Tailored Products

Someone approaches your company with a concept for a software product, which they reckon will improve their organization's performance. Usually it is in the form of Request for Information or a similar procedure.

Frequently, the approaching party has some kind of a preliminary design for the product. Your team members may need to have a meeting with the customer and further analyze their requirements.

Often enough, the preliminary design proposed by potential customers is not what they need. Usually they are not expert computer users, they don't know what they can expect, or even demand, as part of the delivered product. In almost every case they don't have system analysts at their disposal, to make sense of what is required, what is available, what is desired and the way to combine the three. Your representative(s) will need to walk through the customer's processes with an open mind, and define their characteristics in terms of input sources, physical layouts, connectivity, required outputs etc.

The design input (i.e. the information sources by which the product is designed) will consist of information about:

- Existing operational procedures.
- Existing computerized solution.
- Problems with the existing system.
- Expected workloads of the proposed system (quantity of users, physical networking, transactions rate etc.)

- Preferred configurations (hardware, operating system, database servers etc.).
- Users' competence and experience with computers.
- The organization's culture.

The preliminary design document is generated by the prospective customers and owned by them. If your organization generates it, you might cause the customer to lose sight of the project's objectives. The customer must be "on top of things" and have ownership over the process. Otherwise, you are opening a wide door for misunderstandings and wasted development efforts.

The preliminary design is not a legally-binding document. It is merely a baseline understanding, so that the functional specifications to follow will have some kind of an agreed contents and scope. The binding document is the functional specification (page 31), approved by both parties.

Proprietary Product

We will use this term to identify a software product which is designed on your own initiative, i.e. someone within your organization thought about it, its purpose, expected market share and preliminary design.

As opposed to tailored products, where the risk lies with the customer as well as with you (once the specification is agreed upon, the customer starts paying), the risk when developing a shelf-item is completely yours.

If you decided to develop a product that nobody is interested in buying, you have wasted your time, resources and, sometimes, a lot of money.

The preliminary design for such a product is triggered by identifying a market need for a new product or enhancements to existing ones. The market research activity is very tricky, and if you want to do it "by the book" might become quite expensive.

Commonly it involves interviewing a quantity of potential users and compilation of wish lists. Many successful products, however, evolved by "gut feeling" of the developers' marketing experts.

No one can teach you how to develop a "gut feeling". It is maybe the most creative part of developing software.

I can suggest interviewing a small sample of the expected target market. This sample has to represent a variety of users, which are expected to implement the proposed product in different ways. This may not be representative enough as a statistical tool, but it may serve in helping you hone your intuition, and in getting a few good ideas. For example: a version control tool that may be used by software developers, but will meet the needs of civil engineers, military projects and office support personnel, by adding a few extra capabilities.

You may find that for a product to be successful it is not always enough for it to be good. Try to assess (as objectively as possible) the level of enthusiasm demonstrated by potential customers. This is especially true in

case of a new product that might force people into a totally new line of thinking. Don't ask yourself whether they appear to like it, but whether they will buy it, when it is available.

Lack of enthusiasm should not necessarily convince you not to develop the product, but it might affect your marketing and advertising strategies if and when the development process has concluded.

See what the competition is doing:

- Are there competitors for this market, providing a similar product?
- How strong is their hold of the market?
- How does their product compare with your proposed design?
- How much is the customer paying for the available products?
- Is there room for another product, or is the market expected to reach a saturation point soon?

When considering enhancements to your own product, you can limit the research to include a sample of existing customers. Prepare a short questionnaire, explaining the proposed enhancements (a preliminary design outline) and asking for feedback. Provide a simple communication method (such as a fax) so you will not be bothering them too much. Consider further inducing them by proposing better deals (such as better prices for the up-coming version) to those who respond to the questionnaire.

Include an estimate of the impact on existing users should they decide to upgrade once the product is ready for production:

- Hardware requirements or operating system changes.
- Conversion of existing data (may be included as a required function in the preliminary design document).

- Training needs.
- Pricing schemes.

It may be a good idea to ask some of your customers whether they will be interested in running a field test (Beta Site) at their facility. You may also need some real-life data to conduct your in-house testing, which they can provide.

Preliminary Design Review

At the end of this research process, arrange a meeting with your leading team and reach a conclusion: do you want to invest the time, effort and money and develop or enhance the product?

The proposed product must provide at least one of the following benefits over any existing products (yours or your competitors'):

- provide additional desired functions;
- be more reliable;
- be easier to use (more "user friendly");
- perform more efficiently (for example: faster);
- accommodate a preferred platform and/or operating system;
- be significantly cheaper;
- provide a combination of the above.

When comparing the proposed product to existing ones, according to the above list, you must ask yourself (and your team): in cases where the answer for any item was "no", are the other products better? If so — what weight would the customer give to that attribute?

For example: you are exploring the feasibility of developing a product, which is to run on state-of-the-art hardware, be twice as fast as the competition, with similar function and reliability characteristics, and the estimated price will be ten times the current one. What are the chances that customers will upgrade their computer system to the new platform? How important to them is the processing speed? Would they pay ten times the price for it?

An important issue to consider is, which of the companies on your list are best fit to answer the questionnaire and to be used as beta sites. For example: a company agreed to inspect a beta version of a product, which it was using. The new version was to be enhanced dramatically compared to the older one, be simpler to use and with more functions. After a while the company came back to the software developers and said that the older version was much better than the new one. A year later, when the new version was already in the market, that company ordered ten copies of it.

What do we learn from that? You should select your candidates for beta sites carefully, and use only companies where the personnel is open minded and will accept changes at face value. Otherwise the results they come up with will have no merit.

Your leading team has to consider at this stage what will be the best configuration for the proposed system. This includes:

, Hardware (both servers and work-stations).

- Operating systems.
- Networking requirements, tools and platforms.
- Languages to be used.
- Required methods of distributions (special data transfer facilities, protocols, backup utilities etc.).
- An overview of data structures.

Keep in mind that at this stage the above considerations are preliminary, and may change during the detailed design process. These are, more than anything, design assumptions for the purpose of initial planning.

Project Planning

Why Plan?

Before making the final decision to commence development, there is another extremely important factor to consider: available resources. Do you have enough personnel to deal with it? Are they familiar with the required platforms? If not - do you want to take the time to train them, or maybe hire others who have the expertise? It is possible that your current organization cannot support the development of such a product, and it is also possible that you are not interested in diverting your core expertise to the product's field.

You should ensure that:

- you have, or can readily obtain, the necessary human resources;
- these people possess the necessary skills to perform the work properly.
- the development team have access to the necessary equipment, operating systems and network facilities for developing and testing the product.

Once you have decided that you are interested in pursuing the development or enhancement of the product, you need to plan how to do it.

The reasons for planning are many, and include:

- Better understanding of the time frame;
- Efficient allocation of resources;
- Coordination with internal and external functions;
- Visibility of progress.

The project plan is a set of tasks, allocated to specific resources, to be completed within a certain time frame. These are sometimes referred to as Milestones, which may be manifested as specific deliverables.

The plan can also be used to describe dependencies between tasks. For example: integration tests cannot be conducted before the participating subsystems passed unit tests, and those cannot commence unless the software components were written. Figure 2-1 shows an example of a project plan.

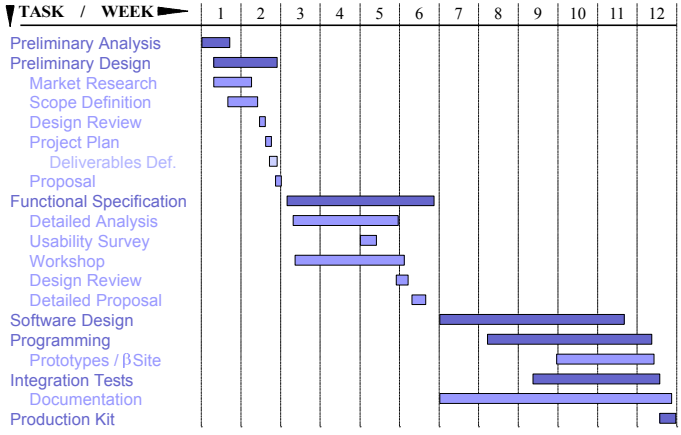

Figure 2-1: Project Plan Example

The scope, depth and selection of tools for project planning depend on the project's complexity. You will find that for projects with average breakdown levels a simple spreadsheet will suffice. Large projects might require a specific tool for project management (such as Microsoft® Project™, Primavera™ etc.).

At this point, I think we should try and define what makes a project "large". The size of a project is mainly defined by the quantity of developers needed to get the job done. A project resulting in 5000 source code modules and 250 executable files may be considered as small, if it is managed by a few developers in a homogeneous, straightforward environment.

From a project management's point-of-view, the definition of a project's size is driven by resources quantity and the diversity of the work to be done.

As a rule, you define how big the project is, in management terms, by the amount of underlying sub-groups (internal customers and suppliers) and the interaction between them. It is all a matter of the complexity of communication between the developing entities.

Project plans are live documents, which are updated continuously as the project progresses.

What to Plan

As mentioned, the level of activities breakdown depends on the complexity of the project. It also depends on your organization's hierarchy and culture. In medium to large organizations defining internal customer-supplier relationships between the groups is best. Therefore, at the project management level you will treat each task as a black box. For example: development of ABC subsystem. On the project's plan you will identify it as a single task with planned/completion dates. The specified development group delivers the subsystem as one of the project's suppliers-subcontractors.

You have probably noticed that, while discussing the project management, we still have nothing much to plan. All there is at this stage is a preliminary design document, which may consist of a single page, and some review notes. True enough. However, it is a very important stage, if not the most important of all — we are building the infrastructure of the project. If you do this part right, the rest will be easier and clearer to follow, and therefore will take less time and money.

Don't worry too much about the accuracy of the project's release to production date: at this point it is a rough estimate which is probably inaccurate. You should be fairly accurate, however, about the completion time of the next step, which is the functional specification.

As a rule of thumb — do not commit yourself to dates which are not under your control. When designing a shelf-item product you can estimate rather confidently when things will happen: you control the resources, you know the existing workload etc.

However, when proposing a tailored product, your project's schedule has some dependencies which you do not control, but which must be referred to in the project plan (for legal purposes also), such as:

- Customer's responsiveness in general.
- Dates of documentation approval by the customer.
- The ability of the customer's organization to meet deadlines for its commitments.

For that purpose it is recommended to schedule the project as a set of relative dates, clearly identifying the dependencies between the planned activities.

When you prepare a quotation for a customer, you may have some other proposals pending. Since the schedule is based on your estimated workload in the near future, this estimate might be completely off if another proposal, to a different customer is, in the meantime, accepted.

Make sure that there are provisions in your proposal for such an event. Limit its validity to a known, reasonable date and ensure that conditions for meeting the proposed schedule are clear.

The same principle by which you have outlined the functions of the proposed product in the preliminary design — applies for outlining the project plan. Start with the high-level activities, such as:

- Preliminary design (complete);
- Preliminary design review;
- Functional specification;
- Functional specification review (internal + external);
- Functional specification approval (internal + external);
- Software design documents;
- Development;
- Documentation outline;
- Documentation concept review;
- Critical design review (internal only);
- In-house testing;
- Field testing;
- ...etc

When you have outlined the project's milestones, detail the process required to achieve them according to tasks' complexity. It is recommended to allocate specific resources for each detailed task, and to identify at least two dates: planned and actual.

Project's Deliverables

Project deliverables are the items that the final product is made of. Once you have them all, you know that the project is complete. The deliverables include internal documents and shipped items (i.e. sent to the customer/end user). A typical list of deliverables consists of:

- Preliminary design;
- Functional design;
- Software design;
- Design and review notes;
- Test plans and reports;
- Software;
- User manual;
- Training material;
- Packaging material;
- Product duplication/installation kit.

The list may vary, depending on the type of product and whether it is a shelf-item or tailored for a specific customer.

If the product is being made-to-order (tailored) then the customer's requirements must be identified. Occasionally the statement of work will

specify the deliverables to be supplied with the product as components thereof , if not — clarify this point as early as possible, for various reasons:

- It has a direct impact on project's costs.
- It might contradict your wishes regarding information and knowledge security (if the customer requires the software design documents or even the source code itself).
- It will cause delays if left undealt-with.

The list of deliverables should appear somewhere on a legal agreement between your company and the customer. It can be a part of your proposal, quotation, an appendix to the functional specification or a statement of work issued by the customer. Failure to clearly define these requirements might mean the difference between loss and profit — for your organization, satisfaction or disappointment — for the customer.

Preliminary Design Completion

As detailed above, you now should have clear definitions of:

- What the proposed product is;
- What its high-level functions are;
- A general idea of its configuration (hardware and software);
- The project's deliverables;
- The resources necessary to complete its development;
- An estimated time frame for completion of each task;
- A list of milestones and dependencies.

If the product is a proprietary one then please skip the next section and continue reading about the functional specification (page 31).

Formal Proposal

At this stage you compile a formal proposal, if the product is tailored for a specific customer. This document is legally binding, so you should be very careful with your wordings, with what you promise and for when. Secure your rights but make sure that the customer receives what he needs and wants, the best way you can.

In the proposal, identify the following items:

- The preliminary design (descriptions and outlines).
- How you would construct the system — proposed technical structure (platforms, hardware components, operating systems etc.).
- A detailed list of project's deliverables.
- Phases during development (milestones).
- Estimated time frame between milestones (highlight dependencies).
- Expected level of customer's involvement during phases.
- Estimated time frame for project's completion.
- Method of implementation (training, installation etc.).
- Support and maintenance policy.
- Costing schemes (see below).
- Proposal's validity.

When dealing with costing schemes, bear in mind that at this point you only vaguely know the scope and complexity of the proposed product. All you have established so far is that you have the resources (time, human, know-how, tools etc.) to complete the project. You will fully understand the detailed requirements (and therefore the exact cost) after approval of the functional specification and outlining the software design.

Therefore, it is recommended to quote only the price for the functional design phase in your proposal, with final product's price being submitted after the design was approved and agreed upon by both sides. Some customers may require a price for the whole job — I suggest that you only give an informal estimate, preferably not in writing, and definitely not as part of the proposal.

Functional Design

After the preliminary design is accepted by your organization (and by the customer's, if it's a tailored product) you need to detail the product's functions in a way that will enable the software engineers to construct the software product.

This document is an event-driven one, and it describes processes ("what", but usually not "how"): each high-level function in the preliminary design document is broken into its components, and the activities carried-out for each component are detailed.

The functional specification is a legally-binding document, especially when you develop a tailored product. This is what the customer wants and this is what you have agreed to do.

A properly-written specification should not give room for interpretations. This serves both sides and prevents misunderstandings and grievance. It details all aspects of operations clearly, comprehensively and accurately.

Since no one person can think of everything, it is strongly recommended that the functional specification be the result of a team effort. Running a workshop and walking-through the processes is a common way of accomplishing this task effectively.

Recommended members of this workshop group are:

- A business analyst, who is very familiar with the operation side of the product, its environment, users, business concerns and needs. This person is the workshop coordinator, and "owns" the functional specification. This may be a customer's representative.
- A software engineer, preferably the person who will lead the development team. His/her involvement at this early stage will result in a more adequate product.
- Quality assurance representative, who will assist in cross-referencing requirements, practices and risk-assessment (see page 45). This person will be then capable of coordinating testing activities and creating test plans.

Development vs Product Specification

The development specification (which is the functional specification, in our case) is a document detailing what the product will do and how it will perform. The product specification, on the other hand, describes an existing

product and may be used for marketing purposes, help-desk reference and the like.

Without much effort, these two documents can be combined. The main difference is the way you word them. While in the development specification you refer to the product in future-tense, in the product specification you state what the product does, at present.

Suitable words for development specifications are:

- Shall — **a requirement**: the user interface ***shall*** meet the standards of the Microsoft Windows Graphic User Interface.
- Should — **a recommendation/expectation**: the system ***should*** run on inferior hardware so it does not force the user to an expensive, immediate upgrade process.
- Could — **an added functionality** which enhances the product, but is not required nor recommended in the specification: the reports engine ***could*** also export files as Rich Text Format documents.
- May — **an allowance**: the backup facility ***may*** be incorporated externally, using a commercially-available tool.

On the other hand, in the corresponding product specification we will state that:

- The product has a graphic user interface which meets the standard of the Microsoft Windows operating system.
- It runs on 80386 Intel compatible hardware or a higher processor.
- Reports can be exported as text, WordPerfect and common spreadsheet formats. Additional filters may be added in the future.
- The software does not include a built-in backup facility, but a gateway for running Norton Backup is provided.

Document Structure

The best method to describe a process, in my opinion, is by using flow-charts. I strongly recommend using them extensively throughout the document.

When compiling this document make sure that it is easy for you to extract requirements from it: the final product has to be traceable to its design document, because this is the only way by which you will have confidence that every requirement has been met and each recommendation — considered.

One way of achieving this is by maintaining a parallel table (using either a word-processor, a spreadsheet or a database) in which the analysts "register" briefly each requirement as it is added to the specification. The software designers can then cross-reference their work to the functional specification.

The recommended outline of the functional/development/product specification comprises of the following sections:

- General — product description, an expanded version of the preface to the preliminary design.
- Scope — Project's scope, what it will include and what it won't.
- References — documents that govern the processes, such as military standards, statement of work, quality standards etc.
- Definitions — explanation of abbreviations and acronyms used in the document.
- Functions — high-level functions of the product, as listed in the preliminary design.

,	Performance	System level characteristics, reliability requirements, responsiveness, interface with external systems etc.
,	Architecture	Platform, hardware configuration, operating system, product structure (block diagram detailing sub-systems and modules). Figure 2-2 is an example of such a block diagram.
,	Modules	Distribution of functionality between sub systems and modules.
,	Details	Operating characteristics of each function.

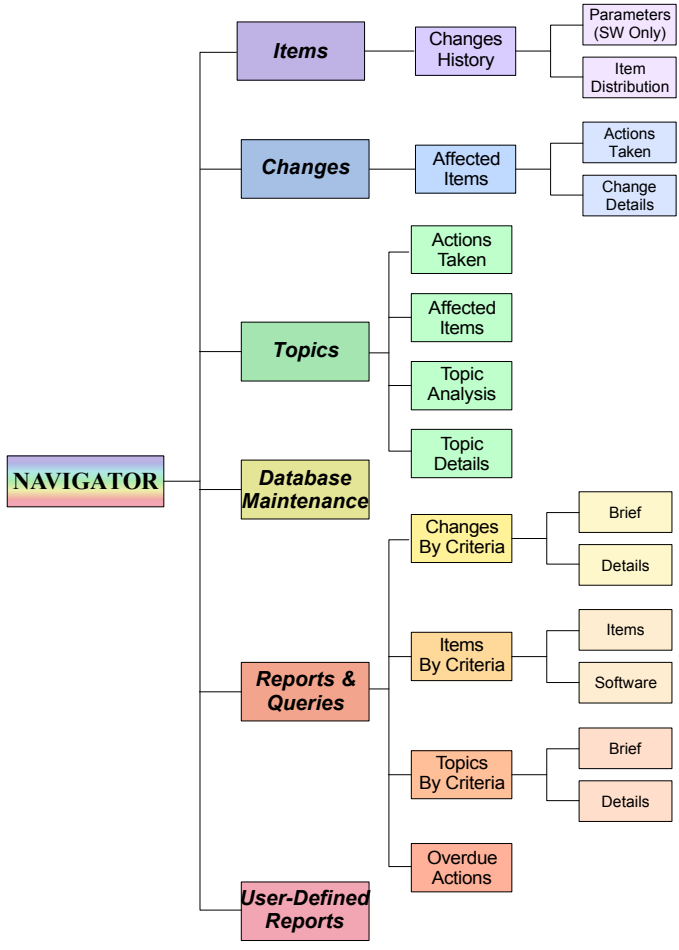

Figure 2-2: Example of a Product's Structure

A practical way of detailing the functions may be to create a template, which will be used for each function. This template is then retrieved into the document and details are inserted as appropriate.

The details for each function include:

- what is triggering the event (the function)?
- where does this function get its input from?
- what does this input consist of?
- how is this input processed?
- what are the outcomes (output) of this function?
- a flow chart.

The following is an example, describing the check-in function of a revision control software system:

Check In

Create a new archive or update an existing one with a new revision of a source file.
This action may be taken on a single file or on many files.

Triggered By	1 File(s) dragged from the Work Directory control and dropped into the Archives control 2 Check-In was selected from the menu 3 Command line parameters
Input Data	1 Revision (numeric, 0.01 to 999.99); default is last registered revision + 0.01 2 Version label / revision note (<=50 characters string); optional 3 Password (n character string); optional
Process	1 Create archive folder, if does not already exist 2 Validate data: , Revision is within the acceptable range , Version label is unique for the checked-in file 3 Verify file's lock status 4 Update reference library 5 Check in 6 Update database (see data structure) 7 Unlock file 8 Delete source file (see user configuration x...)
Outcomes	Inform user of progress and completion

Notes:

1 Prior to user input the software checks if the file was modified since it was last checked-in. In this case it allows the user to skip this/all unchanged files or to force check-in.

2 Action may be taken on one or more files. The user can specify that the parameters apply to all selected files.

3 Over-ride of revision number applies only to the first file if multiple files were selected and Apply to All was checked.

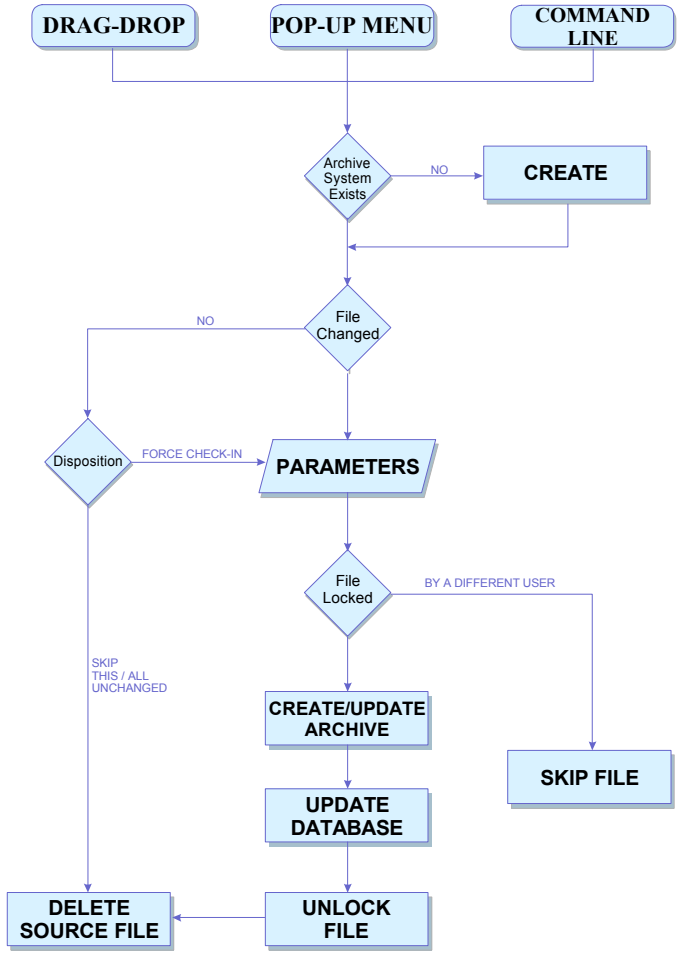

Figure 2-3: Example of a Function Flow-Chart

As you have probably noticed, I preferred to use the product specification method of wording, by which a present situation is described.

In addition, the Input Source item (which appears on the recommended bulleted list) was skipped for this function. This is because (1) it is considered bad practice to state the obvious in specifications, and (2) it demonstrates that we don't treat anything as a rigid requirement, as far as layouts, templates and such go (some specification practices require that you identify each item, and if it doesn't apply you shall insert "N/A" or a similar notification, to show that the item was not forgotten. We try to be practical in this guide).

This function, as detailed, is still open to interpretations, but on the technical side — not the functional. All process parameters are clear. However, there is no description of, for example, the archiving method being used (reverse / forward delta, compression etc.) or what the feedback to the user consists of.

At the higher level (system performance, as supposedly detailed previously in the specification) such things should be identified either by specifying a required method of implementation, or by quantifying characteristics, such as the number of failures allowed during normal operation. In a way, the latter complicates things significantly, because even though you are not required to use a given process (which you might disagree with) the responsibility remains with you to meet the reliability requirements.

Usability

For the product to be accepted by the users, its mode of operation has to be consistent, and to support their methods of running their business.

Some functions may be accessed via the third level of a drop-down menu, while some others, which are more frequently used, may need to be activated faster, with a short-cut of a single keystroke or a click on a command button.

For example, in a telephone marketing support system, you would not expect the user to type-in the name of the desired product, but rather select it from a list. Further more — even this might not be fast enough if there are many items in the database. To speed things up you will also allow the user to start typing the product's name (or catalogue number) and the list will be filtered on-the-fly.

The designers can tackle this issue by rating each function according to its frequency of use, thus, at a glance, the developers identify requirements for shortcut-keys, desired menu structure etc.

The product may conform to the requirements, but if the usability aspect wasn't taken into consideration, nobody will use it.

A typical example involves the "once-removed" user — the person receiving the service rather than the person at the keyboard: in a banking system, the client approaches the counter and asks at the teller for the account's current balance. The software is clever and knows its math, so it opens the account's history file, scans the transactions, adds and subtracts (as appropriate) and (after 35 minutes) prints the balance in a very nice typeface. As a user ("once-removed") of a banking system, would you think its performance is acceptable?

Usability considerations have to be addressed during the functional specification phase even though they infiltrate, in a way, the domain of the software design.

The function, in the above example, at the preliminary design stage, will state that "the system will provide up-to-date accounts' data", but the functional specification (as a note in the Account Balance function, maybe) will cite that "processing of this function will not take longer than five seconds under nominal operating conditions." (The "nominal operating conditions" are defined by the system's high level requirements, which are to support nn terminals, nn accounts, transactions rate etc.)

The product has to follow the specified user interface commonly accepted as standard. The user expects a certain behavior as a response to an action (when pressing Enter, for example). Failure of the software to respond in the expected manner will result in a confused user.

This aspect needs to be considered twice during the design phase: when specifying the operating system for which the product is designed (functional specification), and when designing the software itself (software design document).

Selection of the operating system is critical. It is true — graphical user interface is delightful, but it is not always suitable for the required product. If we take a typical DOS application and compare it to a typical Windows one, for example, we see that the DOS application is structured serially (such as menu-driven) while the Windows environment allows the user to jump from one task to the other (event-driven). Do you want the users to have such a freedom? Would it confuse the typical users if they click by mistake on the wrong window? Is it required for them to be able to jump all over the place?

What's more: how do the users use the product? Is their main input source the keyboard? If so — would it be convenient for them to take their hands off the keyboard and navigate with the mouse? Would they know what to do if faced with mechanical problems, such as their mouse getting dirty (which it often does) and not responding?

On the other hand, some desired usability features, such as drag and drop, might not be available in a text-based operating system.

It is expected, that the initial enhancements to a product after it is released relate to its usability. Analyze the usage of any existing and

proposed system at the functional design stage, so that these enhancements can be easily slotted in afterwards, without the need for any major changes.

Functional Design Review

When the document is complete to the workshop's members satisfaction, it is best for it to be reviewed by a broader audience. This may include the team of programmers, marketing, support etc., at your discretion.

A practical method of reviewing such a document is to circulate copies of it to the applicable functions within your organization and then assemble them in one place and have them "bomb" the workshop team with queries, uncertainties and concerns.

This process, called the critical design review, is especially important if the project deals with a tailored product, because once approved by the customer it might be rather difficult to change.

Upon approval of the functional design of a tailored product within your organization, you need to send it to the customer, thus fulfilling the first phase, as proposed (see page 30). Now, hopefully, you have all the pieces which were missing from the project plan and cost components: you know the complexity of the product, the exact resources needed and the time frame. What you don't know is when (and if) you may commence development: the ball is now in the customer's court.

After you have sent the functional specification to the customer for review and received an approval, you should send the customer a quotation for the completion of the project, as designed. It is undesirable to send the final quotation earlier than that, because the customer may not approve the functional specification as is, and the changes may cause changes in the quotation as well.

Quality Assurance

At this stage, the extent of quality assurance (QA) involvement in the project should be determined. As mentioned before (page 31), a quality assurance representative participates in the functional specification workshop. This person will help the group in determining QA requirements as agreed with the customer, or as expected from this type of a proprietary product, including:

- Level of documentation.
- Documentation review process.
- Configuration management.
- Scope of testing activities.
- Field testing requirements.
- Adherence to specified standards, such as ISO, IEEE or military.

Quality Assurance Plan

The quality assurance representative plans the desired approach for the project from the QA perspective. Whether the QA plan is formalized and documented or not — a clear idea of the approach must be established.

> It is a good practice, and a quality-standards' requirement, that the quality plan is documented. For small projects, however, it should be sufficient to include QA activities in the project management plan, which is essential.

The level of documentation can vary, according to the type of product and its intended use. Since technical writing can be a costly exercise you may find that customers who need a database application for managing sales activities, for example, will be happy with an on-line help file and would wave requirements for printed user manuals, administrators' handbooks and maintenance manuals. An anti-aircraft missile control software, on the other hand, will need to be accompanied by all of those publications — and more.

The review process requirements depend on the level of confidence shared between your company and the customer. The customer may request that each document is reviewed by its delegate prior to implementation, that only documents affecting the functionality of the proposed system are reviewed or none at all.

I would recommend that, in any case, the customer will formally review and approve the functional specification and any functional changes that may be proposed during the product's life cycle. No reason for you to take the full responsibility for the system's functionality and then find out that you have misunderstood a sub-system's concept of use.

Documents for proprietary products, of course, are reviewed within your organization.

Implementing a minimalistic approach, by which you demand as little as possible from yourself, may be the cheapest way to go — in the short run. In the end, however, you will probably find that you have wasted your time and money for nothing because you will have a product that no-one will buy. The customer expects a certain minimum even from a shelf-item product, intended to be sold in shopping centers stores: user manuals, attractive packaging, easy installation procedure etc.

Consider your product's market: who will be using the product and for what purpose? In many cases you have to adopt the requirements imposed on your intended customer by others, because its organization would not be able to use a product (good as it may be) if it doesn't meet those requirements (for example: a software product for managing QA activities that doesn't provide what the quality standards demand)!

The scope of testing and the processes for field tests are also determined by the type of product, its criticality and intended use. Proprietary products

are more heavily tested in-house and advanced pre-production systems should be supplied to a representative sample of potential users.

Tailored products, however, undergo only preliminary, functional testing in-house and will be supplied to the customer at the earliest-possible stage. A close-loop fault management system (page 95) Should in affect and corrected versions should be supplied continuously so that testing may resume on the latest release.

Configuration Management

The day-to-day tasks of controlling the product's configuration will be discussed during the product's implementation phase, on page 67. Here the subject will be approached from the management and planning viewpoints.

All products undergo changes during their life-cycle. The purpose of configuration management (CM) activities is to identify the product at each stage. This stage may be a formal milestone (such as version 2.31 of the software product) or a specific point in time. It is basically a series of baselines, used as a jumping-board for the next step.

Managing product configuration provide the management and customers (of a tailored product) with visibility of the project, and the development team with the confidence that they know what they are working on, whether

the correct version is being modified and the steps to be taken to achieve a certain result, as well as the ability to "go back" if something goes wrong.

A correct definition of the scope of CM activities will guarantee product success. A well-known software company used to manage the configuration of their software product using state-of-the-art tools. All source code modules were under tight control and all records were perfectly straight (they even passed external audits).

However, there was a need to modify an older version to satisfy a customer's requirement and the system didn't compile! It so happened, that two years previously the operating system was upgraded.

On the other hand — to treat the operating system as another bunch of source files might clog the version control system (operating systems can be large, and increase in size with each new version). So, the question is: what to control and to what extent?

The solution is to control anything that affects the outcome of the product, but to go about it in a reasonable manner. From bottom to top, the list includes:

- Hardware configuration: machine type, make, CPU, memory, integrated peripherals.
- Operating system: type/name, version.
- Compilers and linkers: name, version.
- Source-code files: the files themselves — for each version. This includes project's documentation, user guides, test plans etc.

Assuming that the original distribution media is stored in a safe place and is write protected, the first three items may be simply listed in a text file, which is stored with the source files as another controlled file. It is best to include this information in the version description document (VDD, see page 88), generated for each product's version.

Any file that you can and may modify within the system (including files which form part of the operating system, such as a registry database) might need to be treated as a source file, since its state is not as supplied by the software vendor on the distribution media.

Imagine that you take a blank machine, as specified, with a formatted hard disk. Everything you need to rebuild any version of the product has to be available to you for installation.

Since we are aiming to be result-oriented, it is expected that a successful configuration management system will be able to:

- Rebuild successfully any version that was ever registered, so that the outcome, when compared with the original release, is found to be identical.
- List files that were added or modified between versions, between dates, by a specific person etc.
- Compare differences between files and product's versions.
- Identify the functional differences (unlike the previous item, which details the physical ones).
- List files and generations which participate in a specific product version (engineering tree).
- Ensure complete traceability between the on-line system, the tested one and the one stored under the version control system.
- Provide visibility of inter-relationships between sub-systems and modules (product tree).

As mentioned before, project's documentation, specifications and test plans are also under configuration control. That means, that the CM activities for a product start when the preliminary design is **about to be written**, because it has to be registered (and possibly allocated a unique identification).

The product, at a given point in time, may be compatible with one version of the functional specification, while, at a different point — with another (an example to that is an added functionality or a sub-system).

I have noticed that, while product trees (see figure 3-1, page 65) are broadly utilized when hardware is developed, software development groups often reject them as an overkill. In my opinion this is a dangerous approach, especially when dealing with several developers working on a medium-to-large size project.

When not using a product tree, people have to rely on their memory in regards to inter-dependencies of modules and the affects of changes on the various modules, because there is no document describing in a clear manner where the modules are used, which other module includes or calls them etc.

Meticulous developers will conduct a search through the system before they change anything, which will take up much of their time, and we all know that time is money. A "Where Used" query or a quick look at the product tree would have solved the problem easily enough. Besides — even if people's memory is good (which it isn't) — what would you do if they

decide to “move on”, and the new team members don’t have the faintest idea about the structure of the modules? How long will it take for them to learn it, and what parts will be missed after all?

The conclusion: a successful configuration management system will do any task listed above without being dependant on the development team’s members.

The configuration management plan, which is sometimes obligatory to document, needs to address your conclusions on how to carry out the CM activities for this project. A possible outline of this plan may be[1]:

- Introduction — a description of the plan, its objectives and scope. It will include definition of terms, as applicable.
- Management — includes the CM hierarchy, functions, responsibilities, applicable policies, directives, constraints and their affect on the CM plan.
- Activities — functions and tasks required to achieve the CM including Configuration Items identification, change request processes, version control, documentation management.
- Schedule — interaction between the CM activities and the project plan.
- Resources — personnel, tools, techniques, training needs.
- Plan Maintenance — who is responsible for maintaining the CM plan, how frequently, change processing, plan distribution.

[1] ANSI-IEEE Standard 828-1990

Software Design

The functional specification (in the example on page 37) doesn't state how the "Work Directory" control should look like, its size or its type. It also doesn't specify how exactly an input form is laid-out. These items, and more, are detailed in the software design document.

It was said, that the functional specification is an "event-driven" document: the function is triggered, input is provided and a process commences. By the same token we can say that the software design document is "object-driven".

In this document we detail the physical components of the product. We follow the system's architecture as detailed in the functional specification (figure 2-2, page 35) and, for each sub-system and module (or "software unit") the objects are defined.

As I wrote in the introduction — I have no intention of being involved in the actual selection of tools. However, while I don't consider the layout of the functional specification too important, allow me to suggest that for the software design document(s) a word processor is not quite suitable.

Many developers skip, or at least shortcut this phase, to their demise. The reason why it is important is that at this stage you engineer the actual structure of the various components, and find most of the "glitches" in the

structural logic. Large organizations may have personnel whose only job is doing just that.

Let's consider the structure of the "document". If we look at the system's architecture (page 35) the hierarchy is clearly visible. Bear in mind that the level of details in the software design far exceeds the block diagram which represents the system's architecture. The hierarchy according to the scope of the software design include the followings:

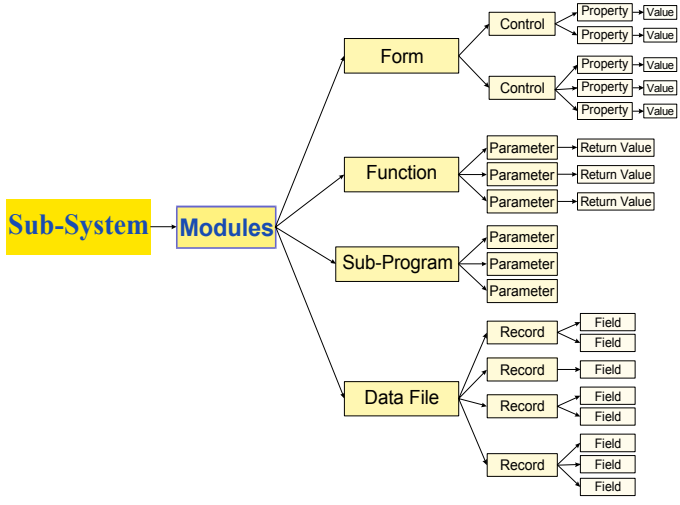

Figure 2-4 - System Components

Screens, forms and controls are listed and ordered. For each type of control the applicable attributes and properties are defined. In addition we also identify hidden objects, which participate in the processes but are not

necessarily visible to the user, such as functions and sub-programs. Input parameters are documented, as well as functions' return values and data-types.

For example, properties of a list control may include:

- Height — quantity of items visible at once;
- Width — the ability to see the full contents of a given field or value;
- Sorting — should the list be sorted? According to what rules?;
- Selection — can users select only one item at a time, or may they select several items and perform a single action on the lot?.

Data files are documented, including:

- The file's location — application directory, data directory etc.;
- File type — sequential, random, binary etc.;
- Record structures — detailing for each field its name, data type, size, bitmaps etc.;
- Scope.

As you see, a word-processing document might not be the best implementation for such a document. A relational database, on the other hand, may also provide you with some additional values, such as reminders of properties of specific controls, a simple to use engine for reports and queries, search facilities and the like. Figure 2-5 demonstrates the use of a relational database for editing software design documents.

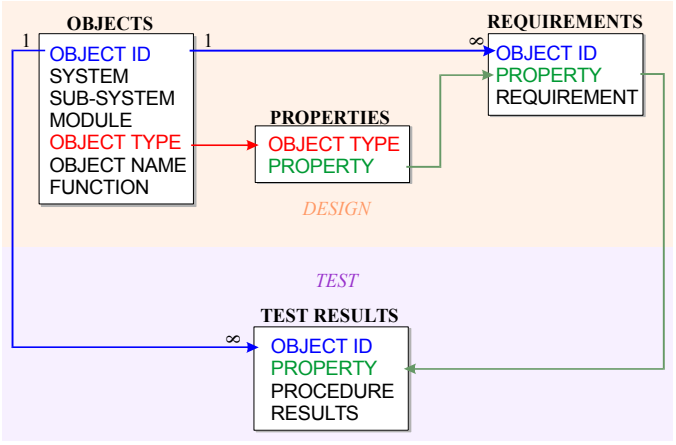

Figure 2-5: Software Design Document Structure

Depending on the programming language used by the development team, on the tool selected for creating the software design document, and on the team's resourcefulness, parts of the code may even be generated automatically for you, thus providing complete traceability between the design and the actual product!

Maintenance of this document is a never-ending, ongoing task. Properties are added frequently, object definitions change because the performance of the system as planned is not good enough and, most important, considerations are added due to fault reports.

Test Plans

The quality assurance team should work on the product test plans in parallel with the design activities.

Test plans are also divided into levels, similar to the design documents: the functional and the detailed plans are driven by the requirements stated in the functional/development specification and the software design documents, respectively.

Each requirement is parsed to detail where and when it should be tested, as appropriate, how it is tested, the expected results and pass/fail criteria.

Many test teams grade the criticality of functions in the test plan, thus identifying the disposition of faulty products (whether to stop testing, fix and resume, just report the fault, accept as is etc.). In my opinion this is not a good practice, for several reasons:

- Every requirement must be met.
- It is possible that as a result of the fault report the requirement will be changed (according to the change-control procedure).
- Grading defects and negotiating them is a management function, which is usually carried-out during Material Review Board (MRB) sessions.
- The responsibility of the test team should be limited to comparing results with requirements.

As shown in the example in figure 2-5, using a relational database for designing the software can accommodate the testing facility as well. This ensures a close-loop process by which new and modified requirements are

immediately apparent to the test team, who then design the test process for that function.

Failure Mode and Effect Analysis

One of the main objectives of the software design phase is the FMEA process, which is sometimes being referred to as "risk assessment".

When designing the forms, functions and controls, the designer must keep asking himself / herself: "what can go wrong during this process?" This may happen, for instance, during the execution of a function (disk is write protected), or because of an "uncooperative" user who pressed the wrong key at the wrong time (yes, we should try and prevent that, too...).

If you design the software using a supportive tool, such as suggested, some of the thinking processes may be done for you. The tool can provide a list of possible attributes and properties for each type of object, so that the designer may consider its aspects and act accordingly.

For example: the property "Minimum Value" of a text box control. Typically, especially when any string may be inserted (as a name, for instance) there is no "value" to consider. However, if the software needs to perform calculations based on the information provided in this field it might

crash if a wrong value was entered (try to divide five oranges by n children, where n is zero!)

While planning the test cases for each function of the product, the FMEA process is the main driver. The test team have to be imaginative and "mean", since the objective is to find the problems in any possible way.

Material Review Board

The Material Review Board (MRB) is a forum which deals with discrepancies, faults, and change requests. It serves as a link between the designers, the testers and the programmers.

Related documents and reports are submitted to the MRB, which has the power to decide on items' disposition.

When the design (or parts of the design) is owned by an external party, such as the customer in the case of tailored products, the decision is made in conjunction with contractual requirements.

The MRB members are:

- The quality assurance manager (chairperson)
- The product manager / principal analyst.
- The test team manager
- The programmers team leader
- A representative of the customer, if applicable.

As mentioned before (Test Plans, page 57), only the MRB should make decisions regarding the acceptability of faults and approval of change requests.

To simplify the MRB procedures and to improve its response time, it is recommended that the fault management system is used for initiating it, by means of data-sharing, electronic mail messages and the like. The committee should meet at regular intervals to discuss strategies and high-impact subjects, while the day-to-day issues are dealt-with by networking.

3

Implementation

This phase is technical and consists mainly of translating the design documents to a language which is readable by a computer.

Like every translation job, the source language data must be available to the translators. In our case — the functional specification and the software design document(s) are the input to the programmers.

Design Review

Documents which are incomplete, ambiguous or missing, result in an undesired product. The programmers then tend to add their own interpretations to the requirements submitted to them, and the product will need to be streamlined after it was compiled. This often leads to a patchy job and wasted time.

The goal should be to do it right the first time. Therefore, before writing any code, the development team reviews the design documents.

During the review process, the design documents are examined from a technical point-of-view, to assess the suitability of the proposed product and its capability to meet the functional requirements.

The relationships between the requirements and the proposed implementation are verified and mapped to ensure that nothing was overlooked.

Design Changes During Development

It sometimes happens, that a design which looked great on paper is not feasible or efficient when it is carried out. Changing the design documents is acceptable, as long as:

- A close-loop is maintained between the programming team, the designers and the testing team.
- The product is compatible with the specification.
- Changes are implemented in a controlled manner.
- Functional changes to a tailored product are reviewed and approved by the customer.

Programming

Your organization should have defined ways of coding software and building products ("programming practices"). Preferably, these definitions are documented in procedures and workmanship standards. We are not going to get into the details of how to write code, mainly because this is really up to you. What's more — those details probably vary with the tools used, languages, and even the type of product.

The main issue that should concern you is the maintainability of the product throughout its life-cycle. This may be a one-time product (for example: a utility to convert data from a previous version) or a product with a life-expectancy of twenty years (such as a shipping company database).

During the life-cycle of the product, any reasonably-skilled programmer should be able to look at the product's package (specifications and source files), and understand the processes being carried-out in a timely manner (i.e. not spending five days changing the position of a field on the screen).

For this purpose, consider all aspects of the product, the applications needed to build it and the support tools proposed to be used. Think in terms of what will be their status up to the date the product you developed retires.

Some operating systems, for example, are limited to a date range. Also, will you use a compiler which is not being further developed? Do you

expect the version control tool you are using to survive the quantity of revisions you intend to make? Would it still be supported in, say, ten years time?

Consider the structure of data; organize it in a clear, straightforward manner. If you are relying on network servers — do you anticipate any changes to drive mapping in the future? Does the project depend on a specific drive mapping (such as search paths in the build command files, included files' location etc.)?

Documenting the Code

Basically, there are two matters here: parallel update of project's documentation and commenting within the source files.

As stressed before — keeping the design documents and the actual product compatible is essential for future maintenance. In large projects it is a good idea to have a search mechanism to find common functions and sub-programs, which the programmers can refer to and use in their modules. Thus, whenever a function is created, the creator immediately registers it in this indexing tool. Such global functions may be dealing with security, date/time manipulation, string processing etc. (This may be a part of the software design document, if implemented as a relational database, as suggested on page 55).

Creating a product tree may also be beneficial, and sometimes essential. This tree can be graphically presented, using a flow-charting tool, or it can be what is sometimes called “indented product tree”. The product tree illustrates the relationships between the product’s units. In the case of software — source files. The following example shows the relationships within a product-tree branch:

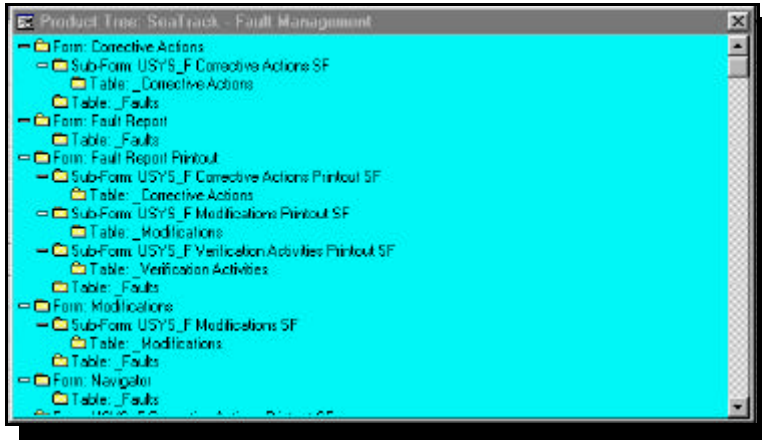

Figure 3-1: Product Tree Section

With a little imagination, and maybe some programming, you can create a utility that will scan source files for keywords such as “include” and generate the tree automatically for you. This is a superb cross-examination tool because it identifies loops which might cause problems.

As shown in figure 3-1, the table "_Faults" is called by more than one form, which is possible. However, what do you think will happen if the form "Fault Report Printout" will be referenced by the sub-form "Corrective Actions Printout"?

A great advantage this product tree gives the developers is the ability to easily identify which modules are affected by a change. Modifying a parent module will have no effect on its children modules. However, it will have an impact on its parents and their parents up to the top of the tree.

As for commenting the source files themselves: opinions regarding this matter range from "never comment code" to "a source file is like a book and should be self-contained". There are pros and cons to built-in comments.

Disadvantages may be:

- Files might become huge.
- Source file editors are usually inadequate for writing documentation (no tables, columns, bullets etc.)
- The reading material buries the really important stuff — the code.

On the other hand, advantages may include:

- Independent, self contained structure.
- Every bit of code is fully explained on-line, so the programmer doesn't need to have on his/her desk an additional pile of books.

Assuming that the product is written in an easily-readable language (third generation and up), and laid-out in a convenient manner (indentations, variable names etc.), I would recommend that comments are

made only to commands and sequences which are not readily understood by an average programmer, or ones which divert from common, expected programming practices (for example: an unorthodox approach to an algorithm).

Configuration Management

Managing the configuration of source code in a large project is not an easy task. Its importance, though, is immense: it provides a set of stable baselines, identifiable changes scope and confidence to the developers, the test team and the customers.

The configuration management activities in a software project consist of:

- Registration and indexing of project documentation.
- Documentation distribution management.
- Change control.
- Source code control (including help files and user manuals).
- Product's version control (release management).
- Configuration status accounting.
- Configuration audits.

Documentation Control

By using a spreadsheet (or, better yet, a database), documents can be registered, indexed and searched-for by all members of the team, especially if this register is accessible on the network.

This registry can be used even better, if the documents can be viewed directly from the registered record (using an OLE attachment, for example). It provides the project team with a centralized information system, comprising of everything they need to know.

Registering intended documents is best, even before they are actually written, to avoid duplication.

A common requirement of quality standards is that a distribution list is maintained for each document. This list may be added to the electronic registry without much effort.

If you are using a relational database to manage documentation, then the distribution list can be an underlying table. In it you should identify recipients of documents and the purpose for which the documents were distributed (such as “Review” or “Controlled Copy”). It may be beneficial to also register the date, copy number and document’s revision.

Change Control

Software development projects sometimes refer to changes as modifications to the software. This is definitely not so!

Let us, first, define what "change" means: if you make cars, for example, and you have decided to change the shape of the roof — you don't take the existing roof and change it: you edit the drawings describing the roof. On the same note, if you have decided to change the seat (which you buy elsewhere and fit into the car) you change the Bill of Material and the purchasing specifications. In Short — you change the governing documents and then make the product compatible with them.

When we want to change software, therefore, we change its specification, whether it's the functional specification or the software design document.

Do not confuse correction of problems or errors ("bugs") with a change request. Those are triggered by fault / test reports while a change request is a prompt to enhance the product or modify its functionality.

A failure of the product to meet its documented requirements does not constitute a change request (although it may result in one, for instance, if the requirement can not be technologically met, and a change in requirements is needed), but rather a fault that needs to be rectified, i.e. to make the product compatible with the current requirements.

If, however, the test team found that the product functions as specified, but this function is unsatisfactory — a change request may be in order, because they want the specification to be changed.

Without getting into the "how" of the change control process, which will differ from one organization to the other, bear in mind that:

- Change requests have to be documented.
- They need to be reviewed by the appropriate authority.
- They need to be implemented in a controlled, reversible manner.
- Their implementation must be verified and approved.

In medium to large projects the process of review and approval of changes is handled by a Change Control Board (CCB). This committee is appointed by the project manager, who usually chairs it, and includes representatives of all functions in the project: engineering, programming, testing, purchasing etc. This enables the CCB to assess change requests, approve or disapprove them quickly and efficiently.

Several aspects of the change request are examined by the CCB:

- The CCB has the authority to approve the change request (sometimes the prerogative is with the customer only).
- The change is beneficial.
- Its consequences were evaluated and found acceptable.

In smaller companies it is still a good idea to try to follow the same procedures, even if the CCB is manned by one person, and even though it is more difficult for one person to think of everything.

Source Control

During the development process, the programmers continuously check files in and out of the revision control system. “Check-in” means to insert a new revision of a source file **into** an archive. You “check-out” a specific revision of a file **from** an archive, which may contain many revisions of that file.

If your organization is using the same archive structure for development and for managing actual releases, you end-up with a huge system, which is slow, as well as hindering your capability to distinguish between the interim activities (changes during development) and the important information (on-line products).

For that reason it is recommended to utilize two separate version control structures: one for development and one for release. The development version control consists of all changes to files, while the release structure contains only the files and revisions applicable to the various formal versions, current and previously released.

One might ask: “so what do we gain if we keep interim revisions of a file that occurred between the previous release and the one before that?”

The answer is “Nothing!” — there is no advantage in doing that. The file may have been modified, and modified again, even with changes

contradicting the previous ones. It simply isn't significant as far as the released product is concerned.

The best method, as I see it, is to carry out changes using the Long Transaction method, illustrated in figure 3-2. A snapshot of a formally-tested and released product is provided to the development team, which use this set of files as the baseline for the change. The development team may then create a version control structure for managing the change implementation, until it is ready to migrate to the release environment as a new baseline.

At this point the development version control system is dumped as a report to document the change's evolution — and discarded.

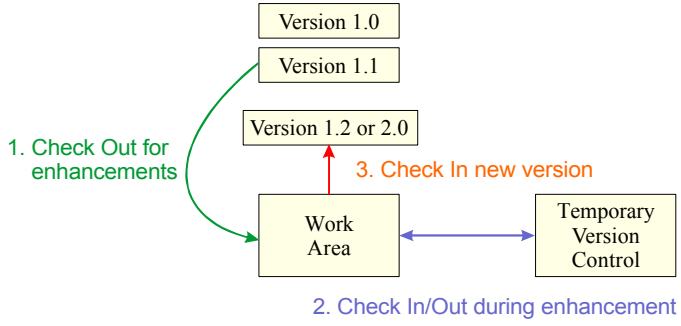

Figure 3-2 : Implementation of Long Transactions

Configuration Status Accounting

The purpose of Configuration Status Accounting (CSA) is to provide the project's team with visibility on components' status at any given point during the development process.

Usually linked to the version control system, this engine enables users to generate reports and queries which detail configuration statuses, pending change requests and the like.

The CSA service should be provided at any level of project's components: documents, source-code, modules, sub-systems and system. Most version control tools are equipped with such a facility, which should be placed on the network for everyone's use.

Configuration Audits

Two types of configuration audits are applicable for software products:

- Functional: the system is verified against the required structure.
- Physical: the released product is verified against the approved configuration.

Functional configuration audits are conducted during development and testing. The physical configuration audit can take place anytime after the product was formally released to testing.

During the audit it is verified that the files installed in the test environment or in the production area are identical to the ones approved for release.

In critical systems it is sometimes required that the system's configuration is verified prior to running it. This is usually done by comparing the files with a read-only reference system or a data table containing information about files and their properties, such as Cyclic Redundancy Check (CRC) values.

During the audits the following is verified:

- All files that should exist — do;
- Unnecessary files are not present;
- The system is stored in the correct location;
- Files properties / contents are correct.

User Documentation

User documentation is not limited to the manual(s), supplied with the product as reference for the end user. It also includes in-house training material, production kits and support information, which are not always conveyed to the customer.

The process of documenting the installation and usage of the product is managed in parallel with the actual source-code development. A close-loop communication is required between the programming team and the technical writers, who document the product.

For each unit that was approved after preliminary tests (i.e. its configuration and function are as specified, though it might still contain errors), the programmers communicate to the technical writers the fundamentals of its usage and some "hints, tips and tricks", which are not readily visible. These may include short-cut keys, secondary mouse button functions etc. The technical writers are also briefed on the actions being carried-out by the unit "behind the scene", so that they gain better understanding of the integration of this unit within the overall product's function.

Messages Library

Another aspect of the unit's function to be discussed with the technical writers is troubleshooting.

The programmers identify areas in which the unit might fail to perform its function. The reference here is not to errors within the unit, because we assume that by the time we finish testing the product, it contains none that will affect its function. Some failure modes, however, may be caused by a

system external to the one we develop, such as the operating system or a peripheral equipment failure.

A “failure” of the unit doesn’t mean that the user had lost control over the application (“crash”); an error message or a warning generated by the application itself sometimes indicates that it cannot perform its function as intended, and the user should be able to identify (and rectify, if possible) the cause.

Programmers should list all the messages that may be generated by the unit, their triggers and meaning, if not self-explanatory. For each listed message a detailed procedure for troubleshooting is provided, as applicable (let’s not get carried-away here: for a message like “printer is off-line” a “procedure” might be an overkill).

Depending on the size of the project, a spreadsheet or a similar tool can be a good and organized way to document these messages.

Bear in mind that this is a useful reference for the help-desk, after the product has been released, because when users call with a problem, in many cases they have a message on their screens.

Documentation Review

The technical writers accumulate the data for the software units and create the draft manuals. These documents are then submitted to the development team for review.

During the review the following aspects of the documentation are examined:

- Documentation scope (i.e. providing the users with what they need to know, but without revealing techniques and trade-secrets).
- Information correctness.
- Adequacy of embedded examples and illustrations.
- The readability of the documents (layout, fonts, language used, grammar).

As for the "language used" item: keeping in mind the audience for these publications, the language should be simple and explanations — detailed. It is considered to be a good idea to "test" the suitability of the documentation on potential users which are not yet familiar with the product. For that purpose, a draft documentation package is supplied with the Beta version of the product (see page 98).

These publications may appear in different forms and shapes, as applicable for the product, its intended usage environments and, possibly, specified requirements.

Shelf-item products are expected to be accompanied by:

- A presentation — for marketing purposes.

- An Internet document (HTML, JAVA etc.) for visibility on the World-Wide Web, if your organization maintains such a site.
- A pre-formatted document, which may be supplied electronically or physically, as a book.
- An operating-system's standard help file, when appropriate.

On-Line Help Files

Typically, when using a help file, topics are triggered according to the control active at the time ("context-sensitive help"). This requires some interactivity between the programmers, who create the product, and the technical writers, who compile the help file, because the software objects are referenced to specific topics within the help files, and compatibility of labels must be managed and controlled.

At an early stage of development, just after the units' structure is identified, the documentation team generates a list of proposed help-file references. It is best that these references (topic labels) follow the hierarchy of the product itself. This list describes the interface between the product and its on-line context-sensitive help topics.

In-Process Quality Control

Quality control during the implementation phase is handled mainly by the programmers. In small organizations the quality control representative is the development team leader, who coordinates the related activities.

As mentioned before, the development process is a translation of the design into a working product. The quality control process ensures that the translation is correct and according to acceptable practices.

During the process, the developers should verify that the product conforms to the *functional* and *physical* requirements, while quality control is concerned with the *practices*: how the development process is managed, controlled and carried out.

Aspects which are monitored by quality functions during the development process include:

- Code structure and naming conventions are in accordance with the organization's guidelines.
- Configuration is managed as planned.
- The technical documentation is adequate and compatible with the actual product.
- Data is secured (see page 80).
- Software units tests are carried-out by the developers and are comprehensive and properly documented.
- Project status is visible (product trees, project plans etc.).
- Test plans and scenarios are developed in parallel with product development and maintained compatible.

To achieve the desired level of control, quality control personnel participate in the development team's review meetings. This way they always know the project's status and can plan audits and testing activities.

Securing Data

Project data has to be protected for various reasons, and while protecting it, we should make sure that access to authorized personnel, both internal and external, is granted with minimum hassle, or else people might refer to their memory instead of to the applicable, inaccessible data source.

Data needs to be secured for many reasons, among which are:

- The data documents the company's knowledge, which makes the company special and keeps it in business.
- Methods of licensing and end-product protection may be rendered useless if known by customers.
- Changes must be controlled, therefore we need to know who made them, when and to what extent.
- There must be a way to recover from disasters, such as hard-disk failures.
- We are responsible for the integrity of products distributed, so it must be ensured.

It all comes down to four basic items: access control, backup, virus protection and disposal.

Access Control

Depending on the size of the project, and the organization as a whole, some procedures to control access have to be in place.

When dealing with local or wide networks (LAN/WAN), access is defined to specific users or groups, in specific areas (servers, directories etc.). Privileges are not very visible from the users' point of view and might change without them noticing it.

A real-life example: a server was upgraded and users which had Read-Only privileges in a specific data area suddenly have Read-Write access. In both cases they see the same thing. The only difference is when they try to write into that area.

In addition, don't forget that there are people in the organization who have unlimited access to data — the administrators. That doesn't necessarily mean that they should be able to read everything, from the security point-of-view.

All these examples mean that provisions to further secure confidential material should be maintained. Protecting or encrypting files with passwords is a good way to secure them, but when you do that, make sure that the password is retrievable somehow, stored in a safe place other than a person's memory cells.

Backup

Hopefully, this subject seems self evident to you, but you will be amazed to know how many people there are, who don't bother to backup their data.

Data files should be backed-up frequently. If you are using a version control system for your files — be sure to backup the archives and not only the files currently being worked on.

It is recommended to maintain a backup cycle, as opposed to overwriting the previous backup.

Take into consideration that the media you use for backup purposes (diskettes, tapes etc.) has a limited life span, and use fresh media occasionally.

If you are backing-up files every now and then, consider this:

- Have you ever tried to restore files from your backup sets, to make sure they work?
- Do you somehow protect these sets against unwarranted extraction of information (remember that this is a compact, convenient package containing all/most of your intellectual property!)?
- Does their storage facility protect the media against fire and other common hazards?
- Is this storage facility "off-site", or will it be destroyed (in case of...) with the original data it is supposed to preserve?

Virus Protection

Computer viruses are, unfortunately, an inseparable part of today's information technology. They consist of small software units which cleverly attach themselves to other, legitimate files and interfere, one way or another, with computers and software operations.

Computer virus concepts are created and distributed every day all over the world, so you are probably aware of their existence and the damages they can inflict on computerized systems.

As a software product producer you have no wish to become a distributor of such viruses, voluntarily or not. Therefore, you have to do your best to verify that your organization, especially the development part of it, is not "infected" by them.

Anti-virus software packages are widely available (probably since five minutes after the first virus was created), but, as with any disease — prevention is the best way to go; be careful what software you bring into your organization and make sure that all operating systems, compilers and support software used are maintained identical to the original, as supplied (which means: "traceability").

Never communicate with the Internet, Bulletin Boards (BBS) or similar services using a computer which is connected to the development network or is itself being used to compile products.

The integrity of your product is extremely important: imagine what the state of your business will be if it has a reputation for distributing infected software! Your customers may accept a few “bugs” as an understandable imperfection — but never a virus.

Disposal

Your organization’s data is valuable even after you don’t need it anymore: when you revise a product’s specification, for example, the previous revision still contains a significant amount of knowledge. It is quite possible that you don’t wish to communicate this information to just anyone.

This is why you should possess the means to dispose of data appropriately. It is true not only for specifications, but also for correspondence, budget plans and other confidential material.

Depending on the volume and type of media to be disposed-of, material can be destroyed in-house (by paper shredding machines, for example) or sent out in a secured manner to companies which do this for a living.

4

Testing and Delivery

Formal tests begin when the product's functional entities are mature enough to be verified against the requirements.

The development team should aim at delivering the product to formal testing at an early stage, so that faults can be rectified as close as possible to the product's basics. The product, for our purpose, is a sub-system, system or a composition of several modules which satisfy specified functions.

Remember: "It costs $1 to correct a problem during development, $10 during testing and $100 after the product was released".

Traceability

One of the most important things to monitor during the product's life-cycle is its identity at each stage and its relationship to ones in other stages.

There is no point investing all this time and money in testing the product if what is eventually installed for the users is not identical to what was tested.

Even though we've reached the formal testing phase at this point, we may assume that the programmers conducted unit tests on whatever they have created and the team leader or a development test team invested many hours of preliminary tests.

As discussed in the introduction, software products are not that visible to the user: unlike a mechanical device, where features are visible, the differences between two versions of a software product are sometimes not immediately apparent.

This is where the version / revision control tool fits into the picture. If the entire development process is built around this tool, and the version control system serves as the hub, data concentrator and project's IT source, we know what we have, when, and how it relates to what we had yesterday.

As illustrated in figure 4-4, data is gathered in the version control system and it is the only source when data is required. This ensures complete traceability between the developed product, the tested versions, the approved release and the installed software.

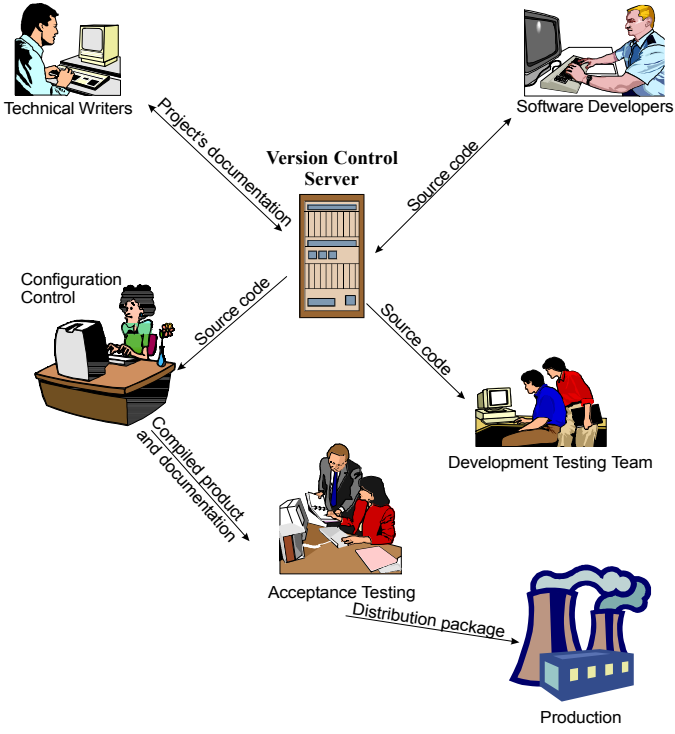

Figure 4-4: Release Management

Traceability is the maintenance of a direct relationship between the specifications, the developed product, the tested system and the product which is released.

The importance of traceability can not be stressed enough. In this competitive world, the difference between well-documented processes and badly managed ones, is the difference between success and failure.

You can not afford to repeat trials and mistakes blindly, while your competition is saturating the market, and the only way to avoid repeating mistakes is by documenting them, by making sure that whatever is tested is what was supposed to be tested and by releasing "out there" only what was fully defined, tested and approved.

Traceability will save you time and make your products the best you can make them, thus giving you the best shot at success.

Submission for Testing

Modules submitted for formal testing, whether they constitute the whole product or just a sub-system, are accompanied by the documentation package which describes them.

As we try to simplify the process, the documentation package is stored, along with the source files, in the version control system, where all the components are identified with a common label.

Version Description Document

The package itself is described in the release notes, which will be referred to as Version Description Document (VDD). The VDD is a brief description of the product as-so-far, at two levels: technical and functional. It links to the product and the documentation package by the version control system label.

The technical section of the VDD describes the differences between the submitted release and the previously-approved one in terms of physical changes to files and documents. It lists the modified components and the changes made. For example:

```
,    1234.c     added GetFileDate() function
,    abcd.bas   string inputs exclude extended ASCII codes.
```

This section also described the characteristics of the required hardware, operating system, tools needed to build the product etc.

The functional section of the VDD details faults corrected and enhancements made to the submitted product, as compared with the previously-approved version. Examples:

- FR 1234 Screen blinks when scrolling down and pressing Shift at the same time.
- EN 1 The software automatically retries to print every five seconds if printer times-out.

> Since we refer in the Version Description Document to the "previously approved version", the scope of the first submission's VDD is limited to its header and some of its technical section, because no previous versions exist.

A proposed structure of the VDD may look like this:

Header

- Project Name or identification.
- System / sub-system or application submitted.
- Description of the submitted product.
- Product's version.
- Submission date.
- Submitting authority.
- Identification in the version control system (class / version label).

Functional

- Governing change requests or fault reports.
- Functional description of changes and enhancements made.
- Compatibility with currently-released system(s).
- Functional dependencies, such as: requires upgraded operating system, more memory, larger storage capacity etc.
- Changes to installation procedures.
- Changes to user documentation and training material.

Technical

- Affected file(s) / module(s); change description; revision / generation of this module within the version control system's repository; unit's test results, date and programmer's name.
- Hardware configuration.
- Operating system requirements.
- Compilers and linkers used and their versions.
- Support tools (converters, pre-compilers, build generators etc.) and their versions.

While the technical section of the VDD may be confidential, you may wish to distribute the functional section with the upgraded product. Therefore, you want it to be separated from the technical section, and to exclude any information you don't wish to propagate to your customers.

Revisions Designation

These uniquely identify the software product, its release status and level of change, compared with the previous version. A smart system of designating revision identifications will enable you to visualize the product's evolution and support users' expectations and decision-making process.

The revision designation should appear in all the items which constitute the released product, for example: manuals, diskette labels, help files, packaging, and of course within the software itself. The user must be able to determine which version is being used.

We identify three classes of change in software products. A three-dimensions array could easily identify each class, for example:

- Version 3.2.2 to 3.2.3: class III change.
- Version 3.2.2 to 3.3.0: class II change.
- Version 3.2.2 to 4.0.0: class I change.

Here are some guidelines to help you decide which change falls into which category, and constitutes which class:

Class I

- User interface concept.
- System architecture.
- Compatibility with hardware or operating system.
- Compatibility with previous versions.

Class II

- Added, modified or removed functions.
- Error handling.
- Documentation methodology.
- Changes to the user interface, such as objects' layout, navigation and other non-conceptual modifications.

Class III

- Error corrections
- Functions optimizations
- Changes and enhancements not included in the above classes
- etc.

A product which contains several changes, belonging to different categories, is designated with the higher level, for example: modifications

making up a class II change and some that constitute a class III change will increment the version identification as class II.

In some cases, where many changes of the same class were made, you may wish to combine them into a higher level, i.e. many class III changes constitute a class II revision.

Installation Kit

Before the product is submitted for testing, a decision should be made as to how the product will be prepared in the future, for distribution to customers.

Since the test team needs to test the complete product's life-cycle events, it must have access to the installation kit, as close as possible to its final configuration. This kit should accommodate events such as:

- New installation;
- Upgrade of an existing version;
- Conversion of data, as necessary;
- Maintenance and related utilities;
- Uninstall procedures and/or functions;
- Other functions, such as license verification and the like.

It is best if the test team use this kit to install the product on their test bed, rather than obtaining it directly from the development environment. In this case the tests are conducted on the same product which will be installed for the end-users.

Test Bed

Just to state the obvious, the product has to be tested in an environment similar to the one it will be used on.

For a tailored product it should be the exact hardware configuration and operating system as your customer uses in his facilities. Otherwise — a typical configuration should be set-up for the test bed.

This is especially true for shelf-items; never use state-of-the-art equipment for your test bed. Software developers tend to do this, therefore missing-out on visualizing product's performance from the users' perspective. Actions which are carried-out within a reasonable time on the test bed might be unacceptably slow on an average machine, or might not function at all.

You will notice, that on software packaging envelopes there is usually a section describing the system's requirements. Sometimes you find, that requirements are reduced to avoid frightening users or forcing them into an expensive upgrade, but when you actually use the product on a system as described it barely performs as you would expect or accept.

Fault Management

In order to have a close-loop failure analysis system, you need to establish a system for managing faults. This system, and the related procedures, must be in place for the testing process to be successful. It will enable effective communication between the test team and, in the future, between support functions and the developers.

Definition of a Fault

A fault is a failure of the product to meet specified or expected requirements. By this definition, it is not limited to deviations from the functional specification or the software design document.

A fault may be:

- An error in the product, causing the product to fail to meet one or more of the specified requirements;
- Wrong interpretation of the specification;
- A query, sometimes referred-to as an “issue”;
- A change request, designed to enhance the product;
- A suggestion.

Fault Classification

Generally speaking, in software products, we have four types of faults:

- Functionality: the product doesn't meet the requirements specified in the functional or product specification.
- Construction: the product is not compatible with the software design document.
- Programming error: a "bug" or lack of compensation for external possible faults, with the result of the product attempting to perform as specified but failing.
- Cosmetic: that includes screen layouts, grammar and spelling mistakes.

If you attempt to grade the criticality of these types you are heading into trouble. Some developers tend to think of a cosmetic problem as a minor issue, even though it isn't. It's true that it would not affect the ability of the product to perform its duties, but from the users' point-of-view this is a shameful error, easily pinpointed (for example - using spell checking) and simply corrected.

Failing to fix cosmetic problems may be regarded as worse than a small "bug", which users treat with a certain level of understanding. Some users may even see it, quite rightfully, as indication of the level of thoroughness used when testing the product, and wonder what else wasn't checked.

Managing Faults

Many tools are available for managing faults, and you can easily create a tool like that for yourself. Depending on the quantity of developers-users, it can be a spreadsheet, a self-made database or a purchased application.

If you are using one of the full-featured automated test tools, available on the market for a small fortune, it may be possible for it to directly feed a fault tracking mechanism.

When you select a tool for managing faults, consider the following:

- Built-in features: are they suitable to your needs?
- Flexibility: your ability to customize it and control its structure.
- Connectivity: what are the methods to connect to the database (dedicated application, Internet, dial-up network etc.)?
- Up-sizing capabilities: ability to transfer data to a more expanded system, such as an SQL server, if needed in the future.
- Security features: to what resolution can you control users' access (tables, fields)?
- User friendliness. if it is not easy to use — people will refrain from using it.
- Reliability. that can be assessed by your experience with this tool, a similar environment or by obtaining a track-record from other users of this tool.
- Price: setup, cost per user, training and learning curve, the cost of maintenance.

Testing the Product

Products are tested throughout their life-cycle by their users. The purpose of testing prior to releasing the product to the market, is to minimize the impact of errors and misinterpretations on the end users.

Assuming that the software was developed as detailed here, supported by the appropriate level of documentation, environment and tools, as suggested, the product should ship out free of defects.

However, there are always things which your development team (and even the customers) didn't think of or didn't anticipate, concerning the day-to-day operation of the product.

It would have been desirable for the test team to comprise the sharpest, most capable developers, but then who would do the actual programming? Some hold the view that testers must not be programmers at all, because they should examine the product from the business' point of view. In any case, select for your test team the most imaginative and detail-oriented personnel, whether they are trained programmers or business analysts.

To assist you in eliminating as many problems as possible before the product is formally released (remember the $1-$10-$100 principle?), two testing phases are identified:

In-House Testing

These tests are conducted by the project's group. If your organization is large enough, you probably have a trained test team at your disposal. However, considering the fact that each and every requirement is stated and documented, testing can be conducted by members of the development team, even if they are not formally qualified as testers.

> The quality standards require that testing activity shall be performed by personnel independent of the development team. I maintain that, if you have clear-cut definitions of the product, anyone who can understand them and compare them to actual findings, can do the tests.

In-house testing (sometimes referred-to as "Alpha Site") should concentrate on comparing between the submitted product and the specified requirements.

To prepare a test plan, requirements are extracted during the functional specification stage (page 31). In addition, if the software design documents are laid-out conveniently (see page 53), the test team can "connect" to them and simply test each requirement, property and attribute in a sequential manner.

Ideally, the specifications and all other types of project's documentation are available to the testers on the version control system, so that they can extract from it whatever they need.

End-user documentation, which is part of the package to be shipped to future customers is also verified against the product, by using it during tests as the users would, during normal operation.

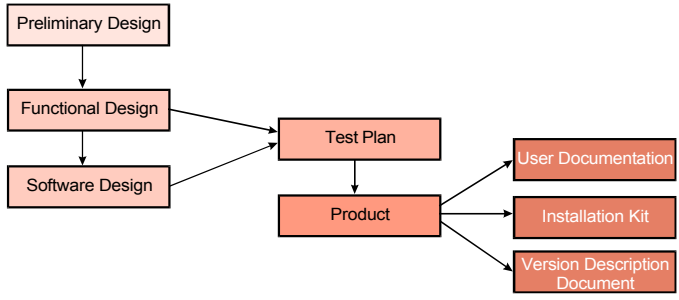

Figure 4-5: Derivation of Requirements

Field Tests

At the end of the Alpha testing phase you should have a product, or a part thereof, which fully conforms to the specified requirements and design. What you do not know is whether the design is what it should be, whether the functions are adequate and whether the implementation is consistent with users' expectations.

For that purpose we need to “burn-in” the product in an environment that is typical to that type of product’s use and represents its operating characteristics (“Beta Site”).

When developing a product for a specific customer, it is recommended to give the customer a working version as soon as possible. It does not have to be the complete system if the product is modular. The sooner you get feedbacks — the better.

When selecting a Beta site for a shelf-item you should consider:

- Typicality: does this site represent the average environment for this product in regards to: workflow, quantity of users, methods of use etc.
- Hardware availability.
- Commitment: the willingness of the customer’s organization to study your product, run with it and report outcomes on a regular basis.
- Accessibility: if you need to send someone from time to time to provide support — how much would it cost you?
- Reliability: your business relationships with that customer.

The product you install for testing at customers’ facilities is not formally released, and should be identified as such, for example: version 3.2 ß3. The shipped package should include as much of the final product as available at that time, even if it is not completed (user manuals, installation guides, on-line help etc.).

You may wish to consider a formal agreement between your organization and the customer’s. It may detail what is being done, to what scope and responsibilities during the process — yours towards the

customer, and the customer's towards you. Typically, this agreement also indicates what benefits the customer gains by helping you.

Keep in close touch with the customer during field testing. In parallel to these tests you continue to enhance and test the product, prepare marketing material and update documentation, therefore you need to know of any problem or issue as soon as it is raised.

One very efficient way of achieving this is by joining the customer into your organization's fault management system, either via the Internet, dial-up networking or a similar environment. This way your development team is constantly updated, and the customers feel that their comments are valued.

Post-Release Activities

Once you have tested and approved the product, including support tools, installation kits and documentation, you are ready to make it visible to the outside world, distribute it and support it.

Marketing

The fact that you have a good product is not enough for it to be a successful one. This is when your organization's marketing functions start sweating and earning their living.

As mentioned before, the Marketing Department is involved in products' development at the early stage of preliminary design (page 14), mainly for the purpose of conducting market research, so that they are aware of the product, its usage, function and intended audience.

I am not going to tell you how to market your product, because (a) I am not a marketing specialist and (b) the marketing strategy may be completely different, according to the type of product and the target market.

However, I have found that regardless of the above parameters, some provisions are expected to be in place for the marketing activity to be fruitful.

These items, which might make up a small project by themselves, are expected to adjust to today's technology and connectivity:

- Internet site: a page (or a more complex structure) describing the product. This is an executive brief outlining the purpose of the product, supported platform, features and benefits ("what").
- Presentation: a slide show, explaining the operational concept of the product ("how"). This is expected to be downloadable from the Internet, FTP or a similar facility.
- Physical material: brochures, fliers and hand-outs to give to customers you actually meet.
- Ordering facilities: such as e-mail, fax, credit-card etc.
- A database: or another tool which will enable you to register customers and their details.

Changes to the Product

Most products will undergo changes after they are released for production. These changes are mostly triggered by customers' comments and new technologies.

When the decision is made to enhance and/or fix the product, we are basically getting into a new project, which will typically materialize as a preliminary design document. That may lead to an amended functional/product specification, a software design document etc.

One of the major reasons for which you need to maintain a customers list, as suggested in Marketing (page 103), is to enable you to contact them if an enhanced product is available. Thus you provide good service while opening an opportunity to further your business.

> Provide your customers with an easy, straightforward method of reporting problems, asking questions and making suggestions. They are helping you in specifying the characteristics of your future products and they don't even charge you for it!

Stable Baseline

Before commencement of changes to the product be sure to know what you are working with; if the change is prompted by customers'

request, you have to know what version they are using. It is possible that the version they are using is not the current one, therefore the change request may not be valid or applicable.

When you are migrating the product to a different operating system, platform or to an enhanced development environment (such as an upgraded compiler), perform the migration first, to achieve the same results under the new system, and only then enhance the product itself.

If you enhance the product first and then migrate it, then:

- You wouldn't be sure that previously existing functions still perform as they should.
- If they perform differently, it will be harder to say whether it is because of the migration process or due to the enhancement.
- You will have more to verify during and after the migration.

The same goes for the different classes of change (page 92). Figure 5-1 describes the evolution of enhancements activities. It describes a product which is to undergo changes of all classes, as well as migration to a new operating system.

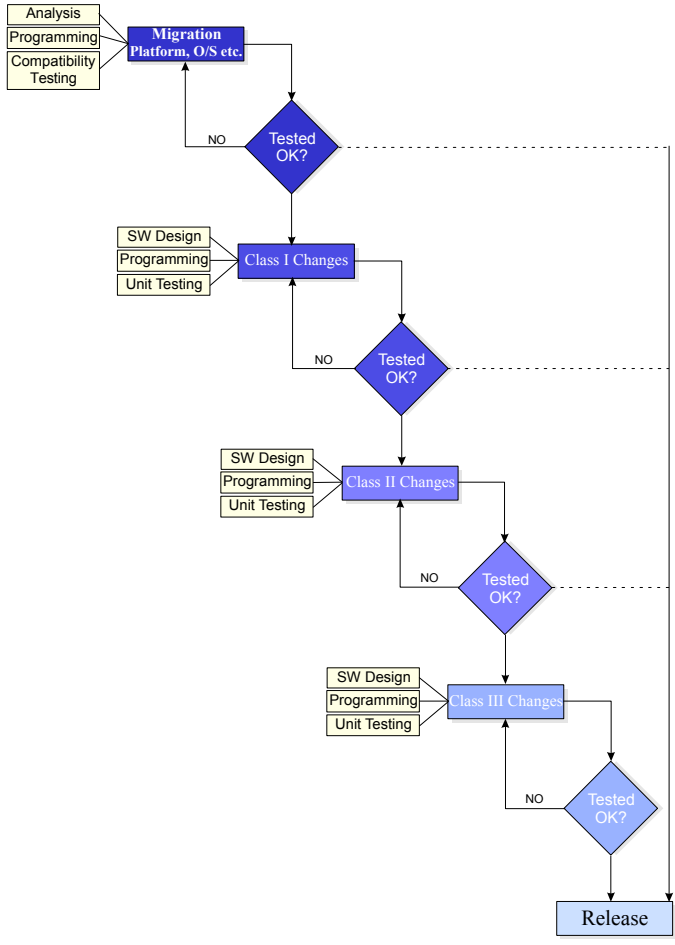

Figure 5-1: Evolution of Changes and Enhancements

High-level modifications should precede the low-level ones, because:

- They may invalidate lower-class change needs.
- They might entail some other lower-class changes.
- Their impact on the product is greater, and until the product is stabilized, “surgical” changes such as class III ones are meaningless.

Needless to say — all files required to materialize the change should be retrieved from the version control system.

Product Recall

You are probably amazed that the subject is even mentioned here. This is usually a consideration only if you manufacture health-affecting or safety products.

Some software might endanger people if it malfunctions, such as a weapon control system or even machinery and traffic control. You might be bound by a contract to recall software which contains a certain level of errors.

However, you may wish to recall your product even if there is no contractual bind or a safety hazard involved, simply because the problem might affect the function of the product or its reliability, and you value your reputation enough to do that.

When considering this option, you would usually assess the criticality of the faulty component to the overall system, the risk involved in this

component not functioning as specified, and the likelihood of occurrence. If all of them are evaluated as high, you simply have no choice - the customers will make you recall the product anyway, so you might as well do it on your own terms.

As opposed to food and drugs, software is not actually "recalled". It is, rather, updated. The difference is, that when you decide to "recall" your product, the customer is not expected to pay for the upgraded one.

Updates and Upgrades Propagation

Some companies have developed very efficient methods to update software products on-line, through the Internet, BBS or similar carriers. By doing so they are providing an excellent service to their customers, without paying the world for it. All they do is make sure that the updated version is available on their site. The customers are paying for the call.

You can enhance this method further by including e-mail addresses in your customers' database, thus enabling you to notify them of updates and maybe even providing a link for downloads.

When you send customers an updated product, be sure to include information about:

- How the updated product differs from what they are using;
- How it will affect their existing data;

- Installation instructions;
- Changes in operations.

For that purpose the VDD (page 88) is divided into sections: you can simply add the functional section of the release notes to the update package.