E-GENTING SDN. BHD.

PROGRAMMING COMPETITION

2005

SPECIFICATION WRITING LECTURE NOTES

Third draft
Jonathan Searcy
17 August 2005

1 WEEK 1 – EXPRESSING BASIC REQUIREMENTS

1.1 Introduction

Welcome all.

E-Genting is running these clinics for two reasons. The first is to show you how to write effective specifications, which are an essential, but often poorly done part of the system development cycle.

The second reason is to promote our programming competition to be held at Genting Highlands on 26 November 2005.

The clinics will run for four weeks. This slide is a broad outline of the material we will cover each week.

The E-Genting Programming Competition will be held on 26 November 2005 at Genting Highlands. First prize RM5,000. Second prize RM2,500. Third prize RM 1,500.	
Week number	Material
1.	Expressing basic requirements
2.	Expressing conditional requirements
3.	Other low-level techniques
4.	Writing a complete specification

Today we are going to go over the reasons why we need to write good specifications and the techniques for expressing basic requirements. For example, if a system is to have a colour touch-screen user interface, how do we express that requirement in a specification?

Next week we will consider the techniques for expressing conditional requirements. For example, if a system is to process a request when a user presses the enter key on the keyboard, how do we put that into a specification.

In the third week we will look at other low-level techniques, such as data dictionaries, function declarations, report and screen layouts, pseudo-code and dataflow diagrams.

In the last week we will endeavour to write a complete specification from a classic ‘café napkin’ specification.

1.2 Why Do We Need Specifications?

Why do we need specifications?

- so that a system buyer knows what he can expect from a system developer;
- so that the system developer knows what he has to deliver to the system buyer;
- so that a large system can be broken down into smaller components that can be programmed in parallel and will fit together when they're finished.

As a system buyer you need to be sure that the multi-million dollar system you're going to purchase for your company is going to satisfy your company's requirements. Time and time again, I see systems being bought that fail to satisfy buyer needs. More often than not, it is because the buyer did not adequately specify what his requirements were.

As a system developer, it is critically important that you know the requirements that you have to satisfy in order to complete your project. Imagine not being able to collect the final payment because the buyer is coming up with a never-ending stream of new requirements that you didn't contemplate in your estimates. In order to complete a project, you need to know what has to be done. A specification puts finite boundaries on what has to be done.

We cannot develop large systems until we can write specifications, read specifications and comply with the requirements of specifications. Specifications of system interfaces are absolutely essential in dividing large systems into smaller sub-systems that will fit together when they are finished. You cannot create large systems in an amount of time that a customer is prepared to wait without dividing the large system into sub-systems that can be developed in parallel and that will fit together in the end. The best, and perhaps only practical way to make them fit together in the end is to specify the interfaces between the sub-systems in the beginning.

If you can't write a spec, or worse if you can't comply with a spec written by someone else, you're not going to turn out to be much of a programmer.

1.3 Specifications Become Law

- specifications get incorporated into contracts;
- contracts are private law.

So you've written your five-page specification, bundled it up with your quote and sent it off to your customer. The next thing you know, there's a purchase order coming out of the fax machine. The quote was an offer, the purchase order is an acceptance of the offer; by offer and acceptance a classic contract has been made.

You now have a legally enforceable obligation to deliver the features described in the five-page specification. If you fail to deliver, your customer can sue you for damages such as the cost of having someone else develop the system. Oops!

When you write a specification you must take the same amount of care as you would take if you were writing a law, because as likely as not, your specification will become law. If

the project goes ahead, the contract for supply of the system will in one way or another incorporate your specification, thereby drawing the specification into your legal obligations, just as effectively as, say, a change in taxation law.

So, if what we're doing when we write a specification, is actually writing law, it might be a good idea to get it right.

1.4 The Specification Writer's most Useful World

The next question might then be: how do we get it right? Let's start with the specification writer's most useful word. It is:

must

1.5 A Prototype Specification

And how do we use it? We use it to create sentences like this:

It must work.

You can use this sentence as a prototype for expressing basic requirements. All that is needed is a clearer idea of what is meant by 'it' and 'work'. 'It' is the subject of our specification, and 'work' is the function that it must perform.

1.6 Using the Prototype

For example:

The data entry system must receive user input. …….. …… It must work.

In this case the subject of the specification is 'the data entry system' and the function that it must perform is 'receive user input'.

It couldn't be easier!

Well it is easy, but only if you have a clear idea of what the system must do. If you are confused, poorly informed or unable to visualise in your own mind what the system must do, you're not going to be able to write the specification. In that case, you might need to do some more research before trying again.

1.7 Subject Confusion

The biggest problem with expressing basic requirements is what I call 'subject confusion'. As a rule, the subject of each sentence should be the subject of the specification.

Here are some examples.

Confusing	To-the-point
The user must enter a customer identifier into the system.	The system must receive a customer identifier from the user.
The result must be displayed by the compute button, which must be flashed at 1Hz.	The compute button must display the result. The result must flash at 1Hz.

In the first case, the specification is specifying what the user must do, not what the system must do. It only implies what the system should do when the user enters a customer identifier, and in that respect it's a weak requirement. If it really mattered, a smart lawyer might be able to argue that the statement didn't specify any requirement for the system at all. As a rule, it's always better to specify what the system must do, not what the user must do.

In the second case, the writer has used a passive construct to specify that the compute button must display a result, which is not particularly good writing style in itself, but in this case it has led to confusion about whether the result must be flashed or the compute button must be flashed. I guess the programmer could toss a coin to decide.

As a general rule, compound sentences compromise clarity. In a specification, compound sentences are sentences that are intended to define two distinct requirements by the use of various joining words such as 'which'. They are very easy to get wrong.

In this case, the subject of the first requirement is the function of the compute button. If you make the subject of the sentence the same as the subject of the requirement, your specification will be easier to write, easier to read and more authoritative.

The subject of the second requirement, to flash at 1Hz, was, I believe, the result. If you use a second sentence for the second requirement, it becomes very hard to misinterpret what was meant.

1.8 Avoid Abstraction

This example, taken from RFC 2246, The TLS Protocol, seems to wander around the subject instead of getting to the point.

Abstract	To-the-point
In addition, a construction is required to do expansion of secrets into blocks of data for the purposes of key generation or validation. This pseudo-random function (PRF) takes as input a secret, a seed, and an identifying label and produces an output of arbitrary length. RFC 2246, TLS.	A TLS protocol driver must use a pseudo-random function (PRF) to expand secrets into encryption and authentication keys. The PRF accepts a secret, a seed and an identifying label and produces an output of arbitrary length.

Once we figure out what the authors are talking about, we can translate it into the requirement on the right.

In my opinion, a protocol specification should specify the functions of the protocol drivers. The protocol drivers are the processes that translate application-level messages into lower-level data transfers. I think the subjects of the TLS specification should have been the client and server protocol drivers. By wandering around the subjects in the above excerpt, the authors of RFC 2246 made the specification more abstract, harder to read and less authoritative, though in virtually every other respect it is a fine specification.

By making sure your specifications follow the 'it must work' paradigm, you will make your specifications clearer, easier to read and more authoritative.

1.9 Exercises

Translate the following statements of requirements into the 'it must work' form. These exercises are all based on real statements of requirements received by the R & D Department from user departments at Genting Highlands.

1. To include the 'available balance' from the Customer Status Report into the Cash Cheque form.
2. Require Expiry Date field, auto-expire of temporary employee card after one week to force staff to proactively replace their card (i.e. convert back to permanent employee card).
3. With the increasing number of terminals, the Terminal Status Display is unable to show all terminals. Require scroll feature to view all terminals.
4. To add Table Opened, Patron per Table (with breakdown of time segments) into output sort keys of the Table Status Report.
5. Enable to see sub-total value by location and cashier user id for particular date in point-of-sale End-of-Shift Report.

6. To list the profit centre of adjustments in the location column of the Customer Transaction Statements.
7. To indicate the user id who created and last update each Customer Group Maintenance group.
8. To allow the user to select an access power class and list all the users maintained under that power class.
9. Report on customer balance adjustment command usage, especially analysis by user and profit centre.
10. To add gender field to the XML customer data block.

2 WEEK 2 – SPECIFYING CONDITIONAL REQUIREMENTS

2.1 Preamble

Last week we reviewed the fundamentals of expressing basic requirements in specifications and introduced the 'it must work' paradigm.

Simple specifications can sometimes be written with phrases no more complicated than simple adaptations of 'it must work', but usually more complicated phrases are needed.

This week I will introduce the specification writer's second most useful word.

2.2 The Specification Writer's second most Useful World

If

The specification writer's second most useful word is 'if'.

2.3 A Prototype Conditional Requirement

And this is how you use it:

If *X*, it must *Y*.

All that is needed is a better idea of what is meant by 'X', 'it' and 'Y'.

X is the condition, 'it' is the subject of the specification, and Y is what the subject must do when the condition is satisfied.

2.4 Applying the Prototype

For example:

If a user presses the enter key,

If ……X ……

the shell must process the command.

…it … must ……Y……

In this case, the condition is ‘if a user presses the enter key’. The subject of the specification is ‘the shell’ and when the condition is satisfied, the shell must ‘process the command’.

2.5 Conditions other than ‘If’

When the transaction processor receives a request it must process the request and return a response.

So long as no keys are pressed, the monitor must show the idle display.

There is a whole vocabulary of words you can use other than ‘if’. For example, if the condition is only a matter of time, ‘when’ might be better than ‘if’.

In the second example, the condition is ‘so long as’; it’s an English language equivalent for something like a structured programming ‘while’ statement.

Nevertheless, the best structure for expressing conditional requirement is generally: condition, subject, ‘must’ and then action.

2.6 Beware of Absurdities

Now let’s diverge for a few minutes to look at the problem of absurdities.

The Oxford English Dictionary defines an absurdity as something ‘wildly unreasonable, illogical or inappropriate.

In specifications, absurdities are generally caused by lazy language usage that creates a wildly unreasonable, illogical or inappropriate requirement.

A member of my department wrote this quite recently:

‘The game control console must enlarge the previous dynamic message status display window and font size.’

Just right now, I’m having difficulty imagining how the game control console might be able to get up and type the keystrokes necessary to change the program and enlarge the window and font size.

2.7 What did She Mean?

What she wrote:

> The game control console must enlarge the previous dynamic message status display window and font size.

What I think she meant:

> The game control console of the new release must have a larger status display window and the status message must be shown in a larger font than in the previous release.

By carefully thinking about the subject of our specification, and setting out what the subject must do or be, we can avoid writing absurd requirements.

In this case, the game control console had to have a larger status display window. That is not the same thing as the game control console enlarging the status display window.

2.8 Exercises

Translate the following statements of requirements into the 'if X it must Y' form.

1. "As a general rule, reset (RST) must be sent whenever a segment arrives which apparently is not intended for the current connection. A reset must not be sent if it is not clear that this is the case."[1]
2. "The minimum time between the transmission [by a slot machine] of the last byte of an SDB [standard data block] or MDB [manufacturer data block] and the transmission of the first byte of the next MDB is to be at least 200 milliseconds. (The time between the transmission of the first byte of an SDB and an MDB or two successive MDB's is to be at least 400 milliseconds.) This is to allow the system to process the SDB or MDB before receiving the next MDB."[2]
3. "Reception of [a change cipher spec] message causes the receiver to instruct the Record Layer to immediately copy the read pending state into the read current state."[3]
4. Devices that are very busy (for example slot machines showing a graphic animation) may respond to any command with a "Low level Busy" response. This response is "0x01,0x81". The effect is that the CLIF restarts its timeouts and resends the message using a new frame counter. The slot machine may therefore not process the message when return a "Low level Busy" response. This response should only be used if absolutely necessary.[4]

---

[1] J. Postel (ed.), *Transmission Control Protocol*, RFC 793, September 1981.

[2] Australian Gaming Machine Manufacturers Association, *AGMMA MDB Data Interface Specifications, Version 2.01*, September 1997.

[3] T. Dierks, C. Allen, *The TLS Protocol*, RFC 2246, January 1999.

[4] GRIPS Electronic Gmbh, *Crystal Web™ Slotmachine Protocol*, 2003.

3 WEEK 3 – OTHER LOW-LEVEL TECHNIQUES

3.1 Preamble

Over the past two weeks we have considered appropriate phraseology for expressing basic and conditional requirements.

This week we will look at a number of miscellaneous techniques for specifying system requirements.

3.2 RFC 2119 – Key Words

RFC 2119 sets out the meanings of several words that can be used to signify the importance of requirements in specifications.

> Adapted from S. Bradner, RFC 2119, March 1997:
>
> 'MUST' means that the characteristic is an absolute requirement of the specification;
>
> 'MUST NOT' means that the characteristic is an absolute prohibition of the specification;
>
> 'SHOULD' means that the characteristic is recommended, although there may exist valid reasons not to implement it. Nevertheless, the full implications must be understood and carefully weighed before doing so;
>
> 'SHOULD NOT' means that the characteristic is not recommended, although there may be valid reasons to implement it. Nevertheless, the full implications must be understood and carefully weighed before doing so.
>
> 'MAY' means that a characteristic is truly optional.

Among others, it sets out the meanings of 'MUST', 'MUST NOT', 'SHOULD', 'SHOULD NOT' and 'MAY'.

RFC 2119 is targeted at protocol specifications, but it is an excellent recommendation for all software specifications.

The word 'must' means that an implementation that does not have the specified characteristic does not satisfy the requirements of the specification.

The word 'must not' means that an implementation that has the prohibited characteristic does not satisfy the requirements of the specification.

The word 'should' means that an implementation that does not comply with the recommendation may satisfy the requirements of the specification, but the failure of the implementation to comply with the recommendation may have undesirable side-effects or may lead to a failure to satisfy other, mandatory requirements.

The term 'should not' has the complimentary meaning with respect to characteristics that are not recommended.

The word 'may' means that an implementation may or may not have the specified characteristic, but in either case it will still satisfy the requirements of the specification.

I recommend that specifications make full use of these standard keywords.

3.3 Data Dictionaries

The report must contain:

1. for each selected customer:
    a. the customer's customer identifier,
    b. the customer's name,
    c. the date the customer was added to the database,
    d. the customer's age,
    e. the customer's outstanding balance;
2. the number of selected customers;
3. the total outstanding balance of all selected customers together.

After you've learned how to use the word 'must' and the art of expressing conditional requirements, I suggest you master the art of using data dictionaries.

A data dictionary sets out the information to be included in a dataflow. For example, this data dictionary specifies the fields to be included in a report, which is an outgoing dataflow.

It follows the 'it must work' paradigm discussed in the first session. It then lists each of the fields that the report must contain. This is important. A list of fields by itself is not a requirement. Who knows what a list by itself might mean. Maybe it could be a shopping list included in the specification by accident. The important thing is to write 'the report must contain' and then list the fields.

You can use the same technique to specify the contents of an incoming dataflow, such as the fields entered into a dialogue box.

3.4 Function Declarations

You can use function declarations to specify the interfaces between software components. For example, the read system call is an interface through which an application can receive data from an operating system that conforms to the POSIX standard.

The read function must have the following declaration:

```
ssize_t read (int fd, void *buf, size_t count);
```

`fd`
: is an open file descriptor.

`buf`
: is a pointer to a memory location to be loaded with the received data.

`count`
: is the maximum number of octets to be loaded.

`return value`
: is the number of octets actually loaded if the function succeeds, or if the function

```
        fails it is static_cast<ssize_t>(-1).
```

The first line is a declaration of the function conforming to the syntax rules of its programming language.

This is followed by a series of stylised sentences that describe each of the function's parameters and its return value.

Note that the description of each parameter is a legitimate sentence in its own right. The only departure from normal English usage is that the first letter of the parameter name is not capitalised.

3.5 Report and Screen Layouts

```
         1         2         3         4         5         6         7         8
1...5....0....5....0....5....0....5....0....5....0....5....0....5....0....5....0
DD-MM-YY HH:MM                CUSTOMER PROFILE                           PAGE X

Customer id:       X------X
Customer name:     X--------------------------------------X
Telephone number:  X----------------------X

    Number  Service    Number        Amount       Cost of         Gross
   of days  category of trans          sold         sales       revenue
         1  X------X  XXX,XXX  X,XXX,XXX.XX  X,XXX,XXX.XX  X,XXX,XXX.XX
            X------X  XXX,XXX  X,XXX,XXX.XX  X,XXX,XXX.XX  X,XXX,XXX.XX
                      -------  ------------  ------------  ------------
            Total     XXX,XXX  X,XXX,XXX.XX  X,XXX,XXX.XX  X,XXX,XXX.XX

         7  X------X  XXX,XXX  X,XXX,XXX.XX  X,XXX,XXX.XX  X,XXX,XXX.XX
            X------X  XXX,XXX  X,XXX,XXX.XX  X,XXX,XXX.XX  X,XXX,XXX.XX
                      -------  ------------  ------------  ------------
            Total     XXX,XXX  X,XXX,XXX.XX  X,XXX,XXX.XX  X,XXX,XXX.XX

                      :          :             :             :

    XX,XXX  X------X  XXX,XXX  X,XXX,XXX.XX  X,XXX,XXX.XX  X,XXX,XXX.XX
            X------X  XXX,XXX  X,XXX,XXX.XX  X,XXX,XXX.XX  X,XXX,XXX.XX
                      -------  ------------  ------------  ------------
            Total     XXX,XXX  X,XXX,XXX.XX  X,XXX,XXX.XX  X,XXX,XXX.XX
```

I personally think that report and screen layouts are overrated.

I would only include a report or screen layout in a specification when the layout actually mattered. If I did so I would generally say that the report **should** be laid out in accordance with the layout. I would generally not make conformance with the layout a mandatory requirement. It's just too easy to get layouts wrong.

When negotiating a specification, it is much easier to work with data dictionaries, and only when the data dictionaries are finalised, figure out a layout. As a rule, defining report layouts can be left to a competent programmer in the programming phase of the project.

Nevertheless, if you do need to describe an unusual reporting or data entry feature, sketching a report or screen layout is probably the best way to do it.

3.6 Pseudo-Code

The traffic light program must execute the following sequence:

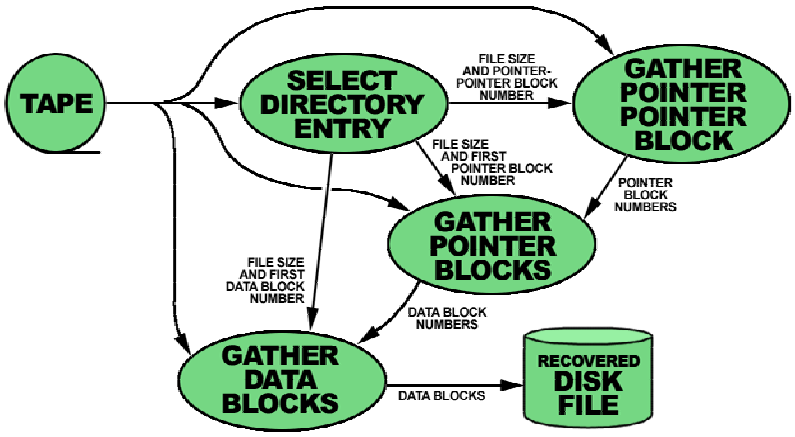
1. Set the branch with right-of-way to NORTH;
2. Loop:
    a. Turn on the green light for the branch with right-of-way and turn on the red light for all the other branches;
    b. Wait for the green time of the branch;
    c. Turn off the green light for the branch with right-of-way and turn on its amber light;
    d. Wait for the amber time of the branch;
    e. Read the vehicle waiting information;
    f. Use the vehicle waiting information to select the next branch to be given right-of-way;
3. End loop.

If you need to describe an algorithm in a specification, pseudo-code is generally the best way to do it.

I agree completely with Brooks when he wrote in 1975 that 'the flowchart is a most thoroughly oversold piece of program documentation'. In comparison to pseudo-code flowcharts are hard to read and considerably more troublesome to maintain.

3.7 Dataflow Diagrams

Having said that, I must distinguish flowcharts from dataflow diagrams. A dataflow diagram helps describe interfaces between sub-systems. If you need to separately specify system components, a dataflow diagram can help you show how the components fit together.



DATA FLOW DIAGRAM

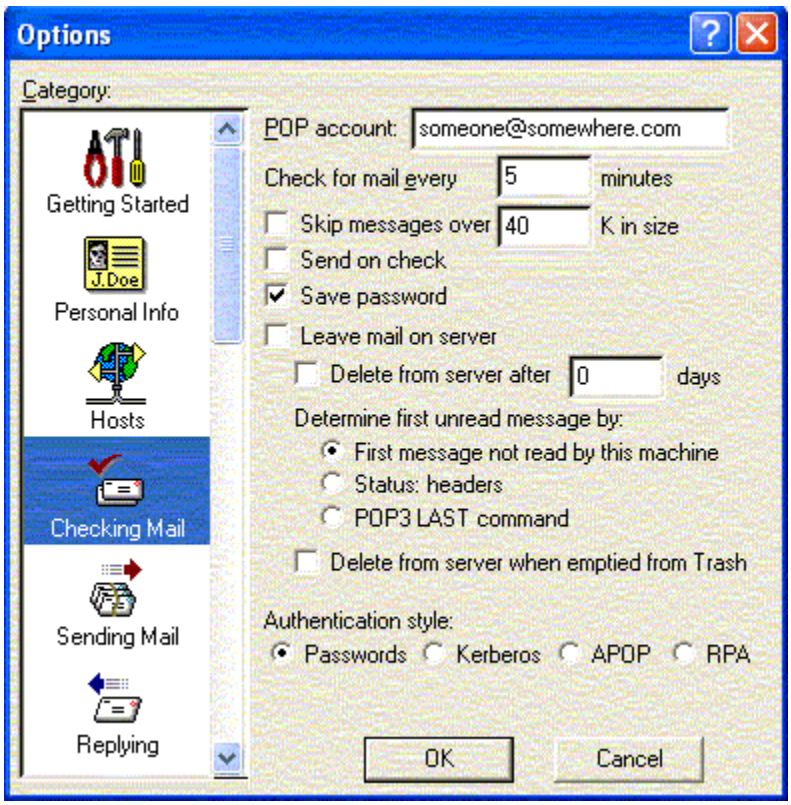
Dataflow diagrams are characterised by processes, which could each, at least potentially, have its own program counter. The processes can be interconnected by data flows of various types.

In this example, we see the data from the tape being distributed to four separate processes, with various types of data being transferred between the processes to create a recovered disk file.

In comparison, the only thing moving between the boxes of a flowchart is an imaginary control token. It's not the same thing at all.

3.8 Exercises

1. Use a data dictionary to specify the contents of the following email client dialog box:



2. Use a data dictionary to specify the contents of an imaginary electricity bill.
3. Use a function declaration to specify the calling syntax of a function that displays the above dialog box and passes the options selected by the user back to the caller.
4. Use pseudo-code to specify a binary search.
5. Use a dataflow diagram to describe a hypothetical interface between an order entry system and an inventory control system. The order entry system must only accept customer orders if uncommitted stock is available. It must separately process the dispatch of ordered items to the customer. The inventory control

system must keep track of the amount of stock in the warehouse and the amount of stock committed to customer orders.

4 WEEK 4 – THE COMPLETE SPECIFICATION

4.1 Preamble

Over the past few weeks we have looked at how to express basic requirements, how to describe conditional requirements and various special-purpose techniques such as data dictionaries, function declarations and pseudo-code.

This session is the last session in this series and we will be working on how to put all those components together to create a real specification.

4.2 Specifications Become Law

Specifications become Law

In the first week, I introduced the idea that specifications become law. One of the main reasons that we write specifications is so that a system consumer and a system supplier can agree on the features that need to be included in a computer system.

If the consumer and supplier are able to agree on the features and a price, the next thing that happens is that a contract is created that requires the supplier to deliver the system and the consumer to pay for it. If the contract is any good, it will refer to the specification, thereby turning the specification into an integral part of the contract. In effect, it will make the specification part of the legal obligations of the supplier and consumer. The specification will have become law.

Now if you wouldn't write your own sale-and-purchase agreement for buying a new house, what makes you think you're able write your own specification for a computer system for your employer which may well be worth many times as much?

That said, the only person who might be available, who can do the job at all, may well turn out to be you, so instead of throwing your hands up in despair, it's probably better to learn how to do it now, before the pressure of a real deadline is upon you.

4.3 Prototype Specifications

It must work.

If X, it must Y.

In the first two weeks we learned how to apply two basic prototype sentences. The first was 'it must work' for basic requirements; and the second was 'if X, it must Y' for conditional requirements.

When you're writing a specification, think about these prototypes, and remember that 'it' should be the article that you're specifying and not the user. The user can do anything. What you should specify is what the system must do when the user does whatever it is that he likes.

4.4 Other Low-Level Techniques

- RFC 2119 keywords ('must', 'must not', 'should', 'should not' and 'may');
- Data dictionaries;
- Function declarations;
- Report and screen layouts;
- Pseudo-code;
- Dataflow diagrams.

Last week, we went over the RFC 2119 keywords: 'must', 'must not', 'should', 'should not' and 'may', which may be used to distinguish mandatory requirements from recommendations and 'we just don't care' concessions.

We also looked at data dictionaries, function declarations, report and screen layouts, pseudo-code and dataflow diagrams. These techniques can be used to describe specific types of requirements.

4.5 How Much should We Write?

Now this is an important question:

How much should we write?

Should we specify the bare minimum, such as 'the system must be a general-purpose accounting system complying with all applicable accounting standards'

or

should we specify the input, output and processing of each function in all its wordy detail?

Both these approaches have applications, but they should be used in different situations.

If you are writing a specification to be sent to potential suppliers to obtain quotes, as a rule you're better off writing the first kind of specification. Just specify what you really need, in this case 'a general purpose accounting system complying with all the applicable accounting standards'. This style of specification does not unnecessarily restrain suppliers, so you will get more bids and can choose from a wider variety of better-priced alternatives.

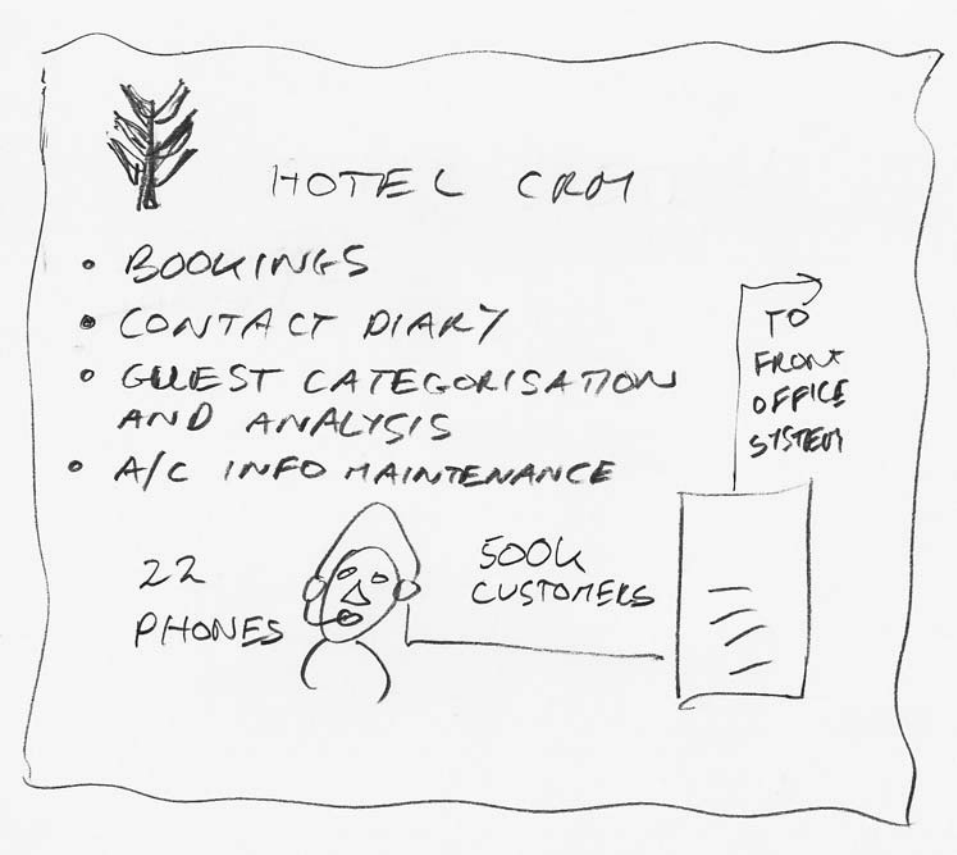
If you use the second type of specification for procurement, invariably the existing systems do not comply with one or another of the arbitrarily specified details, and you will compel potential suppliers to bid a custom-made solution, which will generally be more expensive and less reliable. When you see a consultant coming in and writing the second kind of specification for a purchasing exercise, well, what can I say, it might be a good time to sell your stock. There's probably going to be a 7-digit write-off.

That's not to say that the second kind of specification does not have its uses.

If you're going to create a system, you need the second kind of specification. You need to describe each system function in excruciating detail and ratify the way the functions will work with the users who plan to use them. If you don't do this, you can easily embark on a complete development cycle costing millions of dollars, only to find out that you've created a white elephant, that because it fails to satisfy some fundamental requirements you were unaware of when you started, is no use to anybody for anything.

4.6 The Café Napkin Specification

When I started programming, the first drafts of specifications were almost always written on bar coasters. Now, it seems, they are written on café napkins. Alcohol has given way to caffeine, and the whole business has got a lot more serious.



Anyway, whatever the decade, the situation is that your boss walks up to your desk and gives you a sketch like this on a scrap of tissue paper and asks you to write it up. Apparently it's his idea of a spec for a hotel customer relationship management system.

What are you going to do?

4.7 Start with a Table of Contents

The way to start writing any specification is to draw up a table of contents.

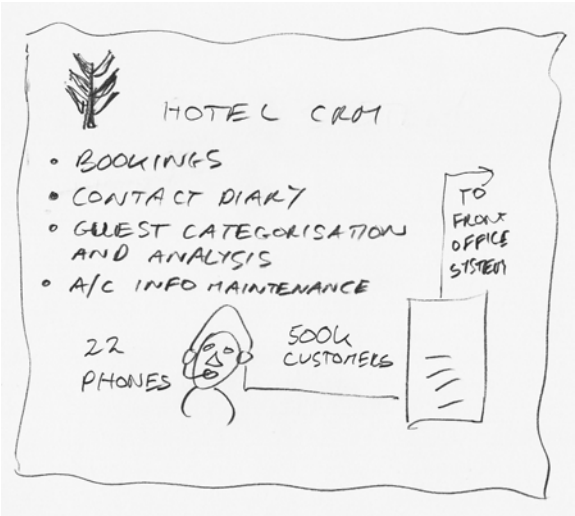


Table of Contents

The first section should always be an introduction. The introduction should explain the purpose of the specification; it should identify for whom the specification was written and who wrote the specification. It may also include references to documentation standards such as RFC 2119.

The rest of the sections are just a list of the requirements in a sensible order.

4.8 The Bookings Section

Now we've got to get creative. From the list of clues on the café napkin, we've got to take an educated guess about what is really needed. For example, the bookings section might expand into this:

2. Bookings

The CRM System must process room bookings entered by telephonists.

The CRM System must not accept any more room bookings for a particular type of room than the room-booking limit for that type of room. The CRM system must have a maintenance function for changing the room-booking limit of each type of room.

The CRM System must also process room cancellations. If a room booking is cancelled, the CRM System must reverse the room reservation so that another customer can take the room.

And so on …

4.9 Exercises

The exercise for today is to finish a preliminary specification for the Hotel CRM System.