//INTEGRA/ELS/PAGINATION/ELSEVIER UK/OMP/3B2/FINALS/0750665076-CH003.3D – 81 – [81–108/28] 29.6.2005
12:38PM

Chapter
3

# FUNCTIONS

## What systems 'do'

GLOSSARY CONCEPTS

Function
Function Requirement
Mission

//INTEGRA/ELS/PAGINATION/ELSEVIER UK/OMP/3B2/FINALS/0750665076-CH003.3D – 81 – [81–108/28] 29.6.2005
12:38PM

## 3.1 Introduction: Function and Function Requirement Specification

Function specifications define '*the business we are in*'. Functions are '*what*' a system does. Functions must be distinguished from *how well* a system performs (the stakeholder performance attributes, such as quality) and from *how much* a system costs (the resources expended).

EXAMPLE

Manual Dialing:
Type: Function Requirement.
Description: The user capability, by any available means {finger on key, voice, name list}, to select or provide and, transmit a {telephone or internet} {number or address} and any other required symbols, to reach and access any available services. It specifically includes any keying or other activity needed in connection with communication, such as accessing lists. It specifically excludes any non-communication activity such as game playing.

### Separation of Functions from Design Ideas

Functions must also be distinguished from design ideas (how a system is going to achieve its requirements).[1] It is all too easy to mix them up but, if you do, you cheat yourself of the results you might get from a better design idea. The test is simple. Ask "Why this {function or design idea}?" If the answer is "because that is what our system 'must have' to be 'our' system at all," you are probably talking about a function: something so fundamental that it is not for the systems engineer to modify or choose.

If the answer to "Why?" is "in order to get a performance improvement" or "for cost reduction," then that specification is a *design idea*, not a function. For example, for a bank, 'lending and dispensing money' are clearly basic functions. The automated teller machines (ATMs) in the wall are clearly a 'design idea' from the bank's point of view. This is because the ATM is one way to make the functions (of lending and dispensing money) have certain performance attributes (such as "to make it easier for our customers to withdraw money at the time they want to").

Making this 'function or design?' distinction perhaps even more difficult, is the issue that the 'objects' we analyze are not purely 'function' or 'design.' Returning to the ATM example, at the 'bank'

[1] Chapter 2 discussed how requirements must be separated from design in general. Now, here we are specifically discussing how function requirements must be clearly separated from design and explaining some of the associated issues.

level, the ATM functionality is a 'design idea'. However, at the level of the 'ATM project' the ATM functions become undisputed 'functionality'. The classification is dependent on your viewpoint.

In Planguage, we classify as 'function' or 'design' in order to convey the information about what is fundamental in a given situation (function) and what is a 'currently selected option' (a design idea). In this sense, we could say that the classification 'function' acts as a constraint[2] on the system designer. This distinction is one made in our minds, because we want the designer to have, or not to have, freedom to improve things. The 'system itself' is unaware of the distinction. An outside observer might not be able to see the distinction by merely looking at the system. For example, is air-conditioning in a car a function or a design? Is it an option or a fundamental concept? It all depends on the attitude of the people involved.

To give another example: at one stage the concept of putting a 'motor' into a horse-drawn carriage (creating the auto-mobile, the horse-less carriage) was clearly a 'design' intended to give certain performance and cost attributes, which 'horses' did not have. At this present stage of culture, the 'mechanical engine' in a car is taken almost completely for granted and has become a function, 'providing mechanical engine power.' This function clearly requires design ideas to implement it, which contribute to the overall characteristics of the car (some engine fuel design options are gasoline, diesel, steam, electricity and nuclear power).

With 'functions' you are not empowered to change them. You can't decide that a car will have no wheels; 'wheel functionality' is too much of a 'given' function. However, you can decide about many features of the design of the wheels, to ensure they have interesting attributes. You can also decide about the design of the 'motor' function, to give both it, and consequently the car, better attributes. But you cannot suddenly change the 'motor' function and opt for the horse again!

One advantage of making the design/function distinction clear is that if new design ideas come along (which could replace current design), you are psychologically ready to evaluate them, and accept the ones that on balance are better than the current design.[3] Another advantage is that you are more likely actively to look for alternative designs. In overall effect, the design/function distinction can free us up to design systems more competitively.

[2] Of course, it is only a 'true' requirement constraint if declared as a 'function constraint.' See later discussion in Chapter 7 on Priority Determination.

[3] *How* we estimate the relative contribution to requirement satisfaction of design ideas (their 'impact') relies on the methods in the following chapters: the quantification of attributes, and the estimation of the impact on these attributes, of the design ideas.

Keep uppermost in your mind that this classification process is simply about *giving information* to systems engineers about what they should take as 'givens', and about what they should 'engineer.' Any engineer, who has a true (engineering or systems architecture) design process, should care about this distinction: the information about what they *should* design is of crucial importance to them.

Classification as either 'function' or 'design' depends on:

- the circumstances:

  A 'selected design' or 'design constraint' becomes viewed as providing 'required functionality' as seen from later and lower levels of the decision-making hierarchy.

- the stage of development:

  One stakeholder's design idea becomes another development process person's 'required function'.

- the current culture:

  Yesterday's design may become today's 'given' function.

- the intent of the specification:

  If it is specified in order to deliver performance or savings requirements, it is a design.
  If it is there because it is 'fundamental', 'because that is how we do things,' then it is a function.

- the degree of freedom of a given type of planner/designer/architect to actually *change* the specification:

  If they are free to change it, then it is more likely design.

The above are some, hopefully useful, ideas to help you classify a specification as a function or a design. But, do not get over fixated by this process. It is finally one of degree and subjective judgment. A specification 'is what it is specified to be' – no matter how we classify it. The classification is intended to give us better ideas of our responsibilities for the specification and our options (Must implement as it is? Or, OK to improve it?).

## Function Requirements

Any required function, which is essential and fundamental to the future system, is called a 'function requirement'. It must be specified as 'pure' function and it must be specified with information about the conditions [time, place, event] under which the functionality exists (otherwise there is no actual requirement!).

EXAMPLE Type: Function Requirement: {F1, F2, F3}.
F1 [USA]: 911 Emergency Dialling Capability.
F2 [Finland]: Character Capability {Finnish, Swedish, English}.
F3 [End Next Year, California]: Exhaust Emission Testing.

In addition, any instance of a real-world function always comes attached with a set of resource and performance attributes. So when we specify a function requirement, we have to consider what has to be done about its associated attributes. All function requirements must respect the full set of performance and resource requirements, which apply to 'their level' of the system.

EXAMPLE Availability.Q1 [F1, F2, F3]:
Type: Quality Requirement.
Scale: % Uptime. Fail: 99.0%. Goal: 99.5%.

EXAMPLE Availability.Q1:
Type: Quality Requirement.
Scale: % Uptime.
Fail [F1]: 90%, [F2]: 92.5%, [F3]: 95%.
Goal [F1, F2, F3]: 99.5%.
*An example of how to specify the specific attachment of performance levels to functions. Availability Q1 is 'attached' to the three named functions, F1, F2 and F3 using qualifiers. In the first instance all goals are attached to the three functions. In the second instance only the one Goal is attached to the three functions and the Fail levels are attached individually and differently.*

Any global scope requirements automatically apply to a function or sub-function, unless they are specifically contradicted by more specific local requirements.

Of course, it may be the case that certain key functions may require even higher performance levels (say for reliability and efficiency) than other functions. In these specific cases, the definition of the function requirement must *explicitly be linked* to appropriate specific requirements.

EXAMPLE Reliability: Scale: MTBF. Goal [Function X]: 99.98%.

Service Performance: Scale: Time in seconds to <reply to inquiries>.
Goal [Function: FX]: 1 second/reply.
*Explicitly linking an attribute to a function.*

By the time a function requirement (or part of a function requirement) is planned for delivery in an Evo step, its performance and resource requirements, and the conditions surrounding its delivery

//INTEGRA/ELS/PAGINATION/ELSEVIER UK/OMP/3B2/FINALS/0750665076-CH003.3D – 81 – [81–108/28] 29.6.2005 12:38PM

should be precisely pinned down using specification parameters like Risks, Is Impacted By, Dependency and Authority.

> All function requirements will, ultimately, have a set of performance and resource attributes associated with them.
> Systems engineering is about getting control over these attributes.

## 3.2 Practical Example: Function Analysis

Consider the (real) proposal to the Board of Directors asking for $60 million, which we first considered in Section 2.8:

> **Proposal to the Board of Directors**
>
> A special effort is underway to improve the timeliness of Engineering Drawings. An additional special effort is needed to significantly improve drawing quality. This Board establishes an Engineering Quality Work Group (EQWG) to lead Engineering to a breakthrough level of quality for the future. To be competitive, our company must greatly improve productivity. Engineering should make major contributions to the improvement. The simplest is to reduce drawing errors, which result in the AIR (After Initial Release) change traffic that slows down the efficiency of the manufacturing and procurement process. Bigger challenges are to help make CAD/CAM a universal way of doing business within the company, effective use of group classification technology, and teamwork with Manufacturing and suppliers to develop and implement truly innovative design concepts that lead to quality products at lower cost.
>
> The EQWG is expected to develop ‘end state’ concepts and implementation plans for changes of organization, operation, procedures, standards and design concepts to guide our future growth. The target of the EQWG is breakthrough in performance not just ‘work harder.’ The group will phase their conceptualizing and recommendations to be effective in the long term and to influence the large number of drawings now being produced by Group 1 and Group 2 design teams.

Now let’s further *analyze* it. Who are the stakeholders? What are the functions?

***Stakeholders:***

Management (Engineering Manager, Board of Directors and others)

//INTEGRA/ELS/PAGINATION/ELSEVIER UK/OMP/3B2/FINALS/0750665076-CH003.3D – 81 – [81–108/28] 29.6.2005
12:38PM

Engineering Design Teams

  EQWG
  Group 1 Design Team
  Group 2 Design Team

Procurement
Manufacturing
Suppliers
Customers

***Functions:***

Carry out Research and Development
Create Designs

  Produce Engineering Drawings

Procure Materials
Manufacture Goods
Establish Work Environment
Maintain Work Standards and Practices
Maintain Organizational Structures

*Note: These lists of stakeholders and functions show an alternative simpler formatting for Planguage sets (Parenthesis brackets '{ }' and commas are dispensed with).*

As the functions become 'lower level,' they begin to constrain the design options! Great care must be taken that function specification is not taken down too far to the wrong level of decomposition. For example, 'Produce Engineering Drawings' is possibly beginning to dictate certain aspects of the solution.

Notice that by separating the different concepts of functions, design ideas, performance, resources and stakeholders, you get much greater clarity about what is really being said. The basis for further system improvement is also laid. For example, the performance attributes should next be taken a stage further, and be given better definitions that include numeric values stating the requirement levels. You are then able to start evaluating the 'impact of the proposed design ideas' on 'all the requirements.'

Now you try! Take some recent and important system requirements from your own work, and analyze it into these components: {Function Requirements, Performance Requirements {Qualities, Workload Capacities, Resource Savings}, Resource Requirements, Design Constraints, Condition Constraints, Design Ideas, Assumptions and Comments}.

## 3.3 Language Core: Function and Function Requirement Specification

Here are some formats for referencing and specifying functions, including structuring them. If this seems more than you need for the moment, then all you *really* have to know is the basic format, *'Function Tag: <function description>.'*

Note: Function specification is not always for function *requirements*. You need to specify functions for *other purposes* as well, such as describing *existing* systems and clarifying functional concepts.

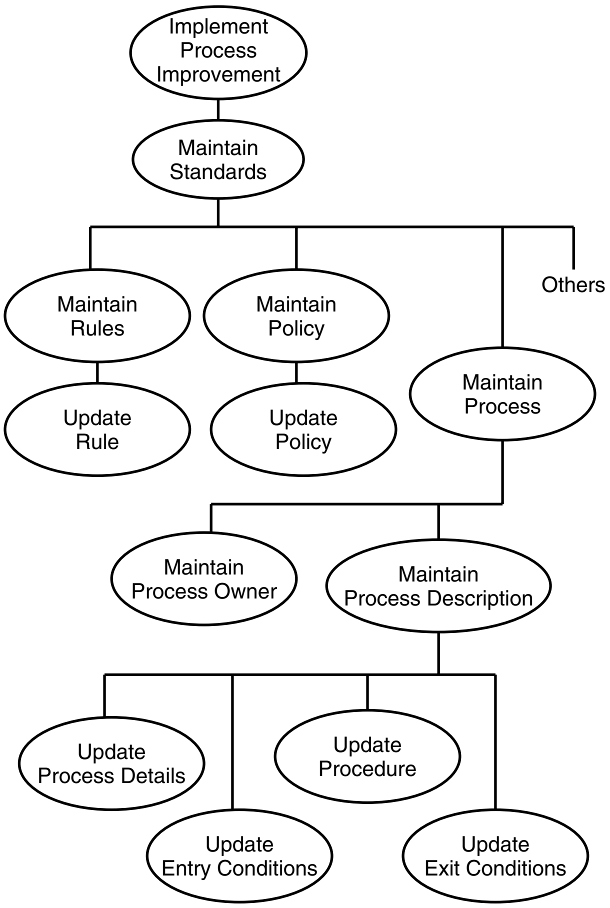

**Figure 3.1**
Diagram showing the relationships amongst the functions used in the examples in this section.

//INTEGRA/ELS/PAGINATION/ELSEVIER UK/OMP/3B2/FINALS/0750665076-CH003.3D – 81 – [81–108/28] 29.6.2005
12:38PM

## Referencing Functions

Functions are identified with a tag. Some variations on the tag structure are given here below. You can use the format: 'Parent Tag.Kid Tag' or, if it is 'unique in context,' just 'Kid Tag.' The first/left tag indicates a parent function, and the following tag indicates a kid/child function.

EXAMPLE Tag: Maintain Standards.Maintain Rules.
Tag: Maintain Rules.Update Rule. "or just Update Rule if it is not ambiguous."

EXAMPLE *You can also use the format:*
Maintain Standards: Includes: Maintain Rules.
Maintain Standards: Includes: {Maintain Policy, Maintain Rules, Maintain Process}.
*Or*
Maintain Standards: Includes: Maintain Rules: Includes: Update Rule.
*The latter example is explicit about the hierarchy.*
*Note: '{. . .}' is the Planguage symbol for a 'set' of things.*

## Specifying an Arbitrary Set of Functions

There is no implication when specifying functions and functional relationships, that all siblings are specified or that the functions listed are even direct descendants of the same parent. Any set of functions can be given a common collective tag for reference:

Arbitrary Function Set: Type: Function {Function Tag 1, Function Tag 2, . . . , Function Tag N}.

EXAMPLE Maintain Standards:
Type: Function.
Defined As: {Maintain Rules, Maintain Policy, Maintain Process, Others}.
*Or more briefly:*
Maintain Standards:
Function: {Maintain Rules, Maintain Policy, Maintain Process}.
*Note: use of the parameters 'Type' and 'Defined As' are optional. It is a matter of style and readability.*

## Inheritance of Higher Level Requirements

A function will automatically inherit any relevant specifications' parameters from relevant higher system levels. This includes higher-level (system-wide) performance requirements, budgets and any condition constraints. These inherited parameters apply by implication, unless there are other parameters that specifically override them in more local function specifications.

## Function Specification

A function specification defines all the currently interesting functional aspects. It optionally includes the function description, functional relationships (that is, the names of relevant supra-functions, sub-functions and sibling functions), associated specific performance and resource attributes, condition information [time, place, event], source information, risks and assumptions, as well as many other parameters. It even includes the implied or 'default' attributes' properties inherited from higher levels of the system of which the function is a member!

Here is an example of a client's function specification (edited for confidentiality):

EXAMPLE

Emergency Stop:
Type: Function.
Description: <Requirement detail>.
Module Name: GEX.F124.
Users: {Machine Operator, Run Planner}.
Assumptions: The User Handbook describes this in detail for all <User Types>.
User Handbook: Section 1.3.5 [Version 1.0].
Planned Implemented: Early Next Year, Before Release 1.0.
Latest Implementation: Version 2.1. "Bug Correction: Bug XYZ."
Test: FT.Emergency Stop.
Test [System]: {FS.Normal Start, FS.Emergency Stop}.
Hardware Components: {Emergency Stop Button, Others}.
Owner: Carla.

The main parameters for function specification are described in the following paragraphs.

### *Function Description*

A function description describes *only* the action(s) of the function.

Function Tag 1: Function: <function description> <-Source.

EXAMPLE

Refugee Transport: Moving refugees back to home villages. <- Charity Aid Manual.
*"The mode of transport will be determined by safety, and cost factors."*

A more explicit 'parameter-driven' format may also be used for clarity:

EXAMPLE

Tag: Refugee Transport.
Type: Function Requirement.
Description: Moving refugees back to home villages.
Source: Charity Aid Manual [Version = Last Year].
Dependency: The mode of transport will be determined by safety and cost factors.

//INTEGRA/ELS/PAGINATION/ELSEVIER UK/OMP/3B2/FINALS/0750665076-CH003.3D – 81 – [81–108/28] 29.6.2005 12:38PM

The choice of more or less formality is governed by factors such as size of plan, size and type of readership, familiarity with Planguage and stage of planning (for example, 'drafting ideas' or 'making a formal plan').

#### *Functional Relationships*

Functional relationships are used to define the relationships amongst functions. For a specific function, the different kinds of relationship include:

- **Sub-functions**: These are any lower-level functions that comprise a function. Any sub-functions at the *immediate lower level* to the specific function are known as **Kid** (Child) functions.
- **Supra-functions**: These are any higher level functions, which the specific function forms a part of (is 'sub-function' of). The immediate supra-functions of a function are called the **Parent** functions. The ultimate, hierarchical top level function, within an organization or project, is usually called a '**mission.**'
- **Sibling functions**: These are any functions sharing at least one parent function with another 'sibling' function.

Here are some examples of specifying functional relationships (see Figure 3.1):

## Defining Supra-functions (as a set of functions)

Supra-functions: Function {Function Tag 1, Function Tag 2, . . . , Function Tag N}.

EXAMPLE Tag: Update Rule.
Type: Function.
Supra-functions:
Function: {Maintain Rules, Maintain Standards, Implement Process Improvement}.

## Referencing Supra-functions for a Function

A hierarchy of tags can be used to show the function hierarchy. You can use **bold** or underline to emphasize which tag you are focusing on. The non-emphasized part is the information about the supra-function ancestry or genealogy.

A hierarchy of tags is connected by dots: 'Tag 1.Tag 2.Tag 3'.

EXAMPLE Maintain Standards.Maintain Rules.**Update Rule**: Type: Function.

## Defining Sibling Functions

The format examples below define siblings.

EXAMPLE Some Kids: Includes: {Kid1, Kid2}.

EXAMPLE All Kids: Consists Of: {Kid1, Kid2, Kid3, Kid Last}.

### *Attributes of a Function*

Attributes of a function are any specific performance or resource attributes specified in the function definition. They include past benchmarks {Past, Record, Trend} describing a function's current or historic attributes and, if a function *requirement* is being specified, they also include future target requirements {Goal, Budget, Stretch, Wish} and constraints {Fail, Survival}. Qualifiers must be used in those attribute specifications to define the specific instances of the past or future use of the function.

EXAMPLE Goal [End Dec Next Year]: 22,000.

Attributes of a function can be described and directly connected to the function in the following way:

TEMPLATE <Function Tag 1>:
Type: Function.
Description: <describe the function here, well enough to allow testing of it>.
Attribute 1: Scale: <?> Goal: <?>.
(Attribute 2: Scale: <?> Budget: <?>.)

*Template for specifying the attributes of a Function.*

Note: Fuzzy brackets, '< . . . >' are used in a template to indicate what to 'fill in'. The fuzzy brackets may contain some instruction, which will always be wiped out when the brackets are filled in. The parenthesis, '( . . . )' are used to indicate (optional) specification types.

EXAMPLE Flagship Product:
Type: Function.
Description: Provide a mobile telephone service [Product Code 9998].
Reliability: Scale: Mean Hours between Faults. Goal [End Dec Next Year]: 22,000.
Battery Life: Scale: Hours Life. Goal [Standby]: 500, [Calling]: 50.
*Function with specific local performance requirements. The specification of function, Flagship Product contains explicit and local definition of two performance attributes. Other attributes and specifications may be implied by other specifications elsewhere.*

### *Qualifiers*

Qualifiers can be used to specify the set of conditions [time, place, event] applying to a specific function.

Qualifiers can also be applied to functions in the following way:

TEMPLATE <Function Tag 1>:
Type: Function Requirement.
Qualifier: [time condition, place condition, event condition].
Description: <Define the function here>.
*Template for a function with conditions. Note: the function is only 'required' or 'valid' when all elements of the qualifier are 'true'.*

EXAMPLE Installation:
Type: Function Requirement.
Qualifier: [Next Year, Activity = Emergency Repair, Major Cities].
Description: Any job <our installers> must perform.

Any useful set of qualifiers is valid.

## 3.4 Rules: Function and Function Requirement Specification

Gist: Specific Rules for specification of Functions and Function Requirements.

Tag: Rules.FR.

Version: October 7, 2004.

Owner: TG.

Status: Draft.

Base: The rules for generic specification, Rules.GS and the rules for requirement specification, Rules.RS apply.

R1: **Functionality**: Function requirements will specify what the system must do and all specified functionality must be required by specified stakeholders (Type: Function Requirement).

*Function requirements are not themselves 'unconditionally required.' Their actual implementation will depend on their relative priority – as specified by qualifiers and other parameters (such as 'Authority').*

R2: **Detail**: The function requirement specification should be specified in *enough detail* so that we know precisely what is expected, and do not, and cannot, inadvertently assume or include

//INTEGRA/ELS/PAGINATION/ELSEVIER UK/OMP/3B2/FINALS/0750665076-CH003.3D – 81 – [81–108/28] 29.6.2005 12:38PM

function requirements, which are not actually intended. It should be 'foolproof'.

*Detailed definition within sub-functions can satisfy this need for clarity, the higher level function does not need to hold all the information.*

EXAMPLE Fuzzy Function Environment:
Gist: Ensuring Environmental Considerations.
*This is a defective specification, given Rule R2. A more detailed function definition is given in the following example.*

EXAMPLE Ensuring Environmental Considerations:
Type: Function Requirement.
Description: All legally and competitively necessary functionality, immediate and potential, regarding environmental protection, in the widest interpretation possible, to protect us against lawsuits, and give us a clear positive reputation amongst consumers.

R3: **Not Degrees**: Elementary function specifications must *not* be described in terms of degrees or variability.

*Elementary functions are binary (present or absent in totality) in nature. If something is 'variable in degrees,' then it probably needs to be reclassified, and redefined as a performance or resource specification linked to a function.*

R4: **Not Design**: The specified 'function' requirement *must not be* a design idea (for example, a strategy, a device, a method or a process) whose only or main justification is to satisfy a performance or resource requirement of the system.

*If the 'function specification' is really a design idea, then it shall be re-classified as 'Type: Design Idea'. If it was intended to support yet undefined performance or resource requirements (like Design X Impacts Performance Y), then action will be taken to properly define these attribute requirements. Such action might justify rewriting the so-called 'function' as a design specification, as there is now at least one requirement that the design idea can impact.*

*We must avoid 'false' function requirements, which are really just designs, which someone assumes would be good for meeting unspecified and unofficial performance requirements. (Local version of Rules.RS.R9: Design Separation.)*[4]

[4] This rule intentionally duplicates RS.R9 as it is considered so important for functions. Whenever such duplication occurs, specific reference should be made to the rule being duplicated.

EXAMPLE Usability: "An example that violates R4 as the Type classification is incorrect! The Description also has errors."
Type: Function Requirement.
Description: A state-of-the-art, user-friendly interface.
*'Usability' is a performance attribute, and needs definition (using a Scale and other parameters, such as Goal). If your intuition doesn't tell you this, then 'state of the art' is a clue as to 'variability' or 'degree of goodness.'*
*'Interface' is a 'thing to be designed' in order to achieve various attributes, including, but not limited to, 'Usability.' Specify this interface amongst the 'design ideas.' It is not a 'what,' but it is a 'how' (a design idea).*

R5: **Function Priority**: If there is a required simple priority for a function requirement, then it should be explicitly stated with information about its authority and/or the source reference and the reason for the priority.

Use the Priority parameter 'Priority: After Y' or use suitable qualifiers '[Before X].' Use the Authority, Source and Rationale parameters to specify the supporting information.

EXAMPLE Rationale: We must address Service Level Agreements as soon as possible to enable the correct level of support to be given when a customer phones with a problem. That is where we are incurring too much cost and tying up engineering support resources. <- Customer Services Director.
*See also Section 7.7, which discusses Priority Determination.*

R6: **Testable**: A function *must* be specified, so that it is *possible* to define an unambiguous test, to prove that it is later implemented *(Local version of Rules.RS.R8: Testable).*

R7: **Test**: Any notions of how or what needs to be tested, in order to validate a function *may* be described using the Planguage parameter 'Test,' with the function name as the qualifier.

The Test information is either specified with the function definition or as a separate item.

EXAMPLE Function Y:
Type: Function Requirement.[5]
Description: Charging to Accounts.
Test [Function Y]: Tests shall be developed to demonstrate that this function is available for all counties in this state, and prove that no other states or countries can access it.

[5] Note: By default, a 'Function Requirement' is assumed to be a 'Function Target'.

Audit: Test [Function Requirement: Function Y]: We must demonstrate to internal auditors that no counties, which are <financially insolvent> are allowed access to this function <- Audit Report [August This Year].

## 3.5 Process Description: Function Requirement Specification

### Process: Function Requirements

Gist: A process for specification of function requirements.

Tag: Process.FR.

Version: October 7, 2004.

Owner: TG.

Status: Draft.

#### *Entry Conditions*

E1: The Generic Entry Conditions apply.

E2: You have the ability to observe comparable 'real' systems (see P3, below).

#### *Procedure*

P1: Describe the hierarchical structures of the high-level function(s), as sets of related function and sub-function tags (for example, F1.F2.F3).

P2: For each function tag (this also includes tags for sub-functions and supra-functions as relevant), define the function, in the detail required by the rules for function requirement specification (see Rules.FR in Section 3.4).
*Note: Focus on real functionality ('what it does') and exclude any design ideas intended to satisfy performance and resource requirements.*

P3: Where relevant: sample comparable 'real systems' to check the accuracy of the function specifications. Correct the specifications as necessary.

P4: Check accuracy and completeness of function requirements, with the people who are currently using similar existing systems. Correct the specifications as necessary.

P5: Perform Specification Quality Control (SQC) on the draft function specifications. Check the quality level against the required quality level, as specified by the exit conditions (see X1). If SQC fails, rewrite/

//INTEGRA/ELS/PAGINATION/ELSEVIER UK/OMP/3B2/FINALS/0750665076-CH003.3D – 81 – [81–108/28] 29.6.2005
12:38PM

correct the function specifications (that is, revisit P1 to P4). Continue P5 until the appropriate quality level is reached.

P6: Once the function specifications have exited SQC, get real current system users (if any) to sign off agreement to them.

P7: Repeat any procedure above until the exit conditions can be satisfied.

### *Exit Conditions*

X1: The Generic Exit Conditions apply.
*By default, no more than 0.2 remaining major defects/page are allowed in any of the function specifications. (A page is 300 words of non-commentary text.)*

X2: The relevant system users, if any, must have signed off the function specification.

**Simplified Function Requirement Specification Process**

Process: Function Requirement Simplified.
Gist: An alternative simplified variation for Function Requirement Specification.
Tag: Process.FRS.
Version: October 7, 2004. Owner: TG. Status: Draft.

**Entry Conditions**
None.

**Procedure**
P1: Declare a specific, 'already specified' and 'currently operational' system to be the 'living map' of the function requirements.

*There is usually an old existing system of some kind. It is likely that a future system must replace this old system, in order for the business or organization to remain viable in the future.*

*Where relevant, use [qualifiers] to aid the mapping of the old to the new.*

The function specification detail is then continuously observable 'in the real system.' It should only be analyzed 'as needed.' Exit immediately.

**Exit Conditions**
None.

*Note: This method is useful when doing evolutionary delivery of changes to impact performance/resource attributes and minor changes to real functionality for an existing large system. You focus on 'improvement results,' not 'supporting functionality.' I normally apply this method to most real projects I get involved in (TG).*

## 3.6 Principles: Function and Function Requirement Specifications

These are principles for recognizing what is, and is not, a function and also for working with functions.

1. **The Principle of 'What Function?'**
   Function is 'what' a system does, *never* 'how well' it does it or 'how it does it so well.'
2. **The Principle of 'Thing with Attributes'**
   A function is the thing, which has the performance and resource attributes attached to it.
3. **The Principle of 'Living Map'**
   Function specification is sometimes best done by declaring the existing system to be a living map.
4. **The Principle of 'Part of Totality'**
   Functions are always part of some larger function and can probably be described by their own sub-functions.
5. **The Principle of 'Each to their Own'**
   Different functions require different performance and resource attributes; so, one reason we specify the functions is to identify and distinguish their required attributes.
6. **The Principle of 'Timing'**
   Different functions can be delivered to customers at different times, so another reason to specify functions is to know 'what to do when.'
7. **The Principle of 'Conditional Function'**
   Some functions may not be necessary, *except* under specified conditions or events, and these conditions should be specified and exploited in project planning. You don't have to do what is not *yet* required!
8. **The Principle of 'Room with a View'**
   A function definition is not absolute; it is a viewpoint, and many overlapping function views can be made and used fruitfully to satisfy different needs.
9. **The Principle of 'Terrain does not change with the Maps'**
   The real system does not change just because you document function viewpoints and function hierarchies: correctly or incorrectly.
10. **The Principle of 'False Function Foils Fruits'**
    If you mistakenly request a design, as basic functionality, you will limit your ability to improve the design to give better competitive attributes.

*Alternatively,*
Don't request 'functions' which are really 'designs for performance',
You might not get the performance you really want.

//INTEGRA/ELS/PAGINATION/ELSEVIER UK/OMP/3B2/FINALS/0750665076-CH003.3D – 81 – [81–108/28] 29.6.2005
12:38PM

## 3.7 Additional Ideas: Function and Function Requirement Specification

### Mission

As mentioned earlier in this chapter, the ultimate top-level function of any system is termed its '*mission*.' A mission describes 'what' a specific system does. Many organizations have explicit mission statements.

We could just as well call 'mission' the 'top-level function.' But the concept and term 'mission' is well known, and for many purposes works better than 'function.' For example, 'The mission of this project is Mars Exploration' sounds better with 'mission' rather than 'function.' Keep in mind that any 'mission' is still really a sub-function of some larger functional context.

Of course, a mission only provides a high-level description of a system's function. Further detail is provided by its sub-functions and by its associated performance and resource attributes. Also, to fully understand a system, we must have information about its environment. A system *interacts* with the environment in which it *operates*.

Note it is important that we not confuse 'mission' with 'vision.' A vision statement is a higher-level concept. It can include ideas about *how well* the mission will be conducted. For example, "Our vision for the 'Mars Mission' is to get back alive, with substantial new scientific knowledge."

### Elementary and complex concepts

Functions and many other Planguage types can be described as being elementary or complex concepts. The meaning of these terms, regarding functions, is as follows:

- An *elementary* function is not decomposed into sub-functions. It may be the case that it is unable to be broken down any further or a deliberate decision may have been taken not to further decompose it.
- A *complex* function is composed of a set of at least two sub-functions. The set of sub-functions can be any mix of relevant complex and elementary functions. At the lowest level of functional decomposition a complex function is defined completely in terms of elementary functions.

EXAMPLE Planning a Project:
Type: Complex Function.
Includes: Elementary Function: {Reviewing Evo Step Feedback, Checking Requirement Specification is Valid, Selecting Next Evo Step, Allocating Staff to Evo Step}.
*An example of elementary and complex functions.*

//INTEGRA/ELS/PAGINATION/ELSEVIER UK/OMP/3B2/FINALS/0750665076-CH003.3D – 81 – [81–108/28] 29.6.2005 12:38PM

## Measuring Functionality

Functions are either 'present or absent;' they have a binary nature. They are either documented or observable (testable) in a real system or they are not.

*Sets* of complex functions can be thought of numerically as 'percentage amounts' of their 'defined lists' of elementary functions. *Elementary* functions are (by definition) not divided into sub-functions.

Complex functions are either 100% present (all elementary functions in the defined complex function set are present) or they do not 'wholly' exist. For a complex function called ABC, you can talk about, say, 90% of the set of elementary functions comprising ABC being defined or implemented. In such a situation ABC itself, the entire complex function, doesn't exist yet; only known degrees of it are defined or in place.

## Additional Examples of Function Specification

Here are some Planguage ideas, additional to the ones shown in Section 3.3, which can be applied to function specification. They give more detail on the use of qualifiers.

EXAMPLE

**F3** [F499]: Receiving e-mail from Customers.
*F3 is a valid function if, and only if, F499 is active or in existence. F499 is a 'condition' (specified in the format of a [qualifier]). F499 is detailed 'elsewhere'. F3 is a complex function specification because it has a qualifier, which must be determined by the qualifier's own definition*
SYSX. **F5**: Sending e-mail to defined [Group].
*'F5' is a 'kid' element of the complex function SYSX. The actual function implementation will differ depending on the current definition of 'Group'.*
**F6** [Date = After First Release]: Get approval by electronic signature.
*F6 is not 'valid' (for implementation, for testing) until after 'First Release' event has taken place. First Release is a specification variable, depending on the actual release date.*
**F7**:
Qualifier: [Country = {European Union Countries, Norway, Not USA, Not Canada}].
Definition: Maintain System XYZ Standards.
*The function, F7 is valid for a defined set of countries. The qualifier parameter, 'Country = ' illustrates how a more explicit format can provide better readability for Planguage novices.*
**F8**:
Description: Answering direct-line telephone.
**Speed**: Scale: Number of <whole rings> heard at Receiving End, before Answer Signal is <sensed>.

Past [Condition = Employee Not At Desk]: 4.
Goal [Condition = Employee Not At Desk]: 1.
**Answer**: Scale: Probability that Caller is satisfied with the Given Answer.
Past [OK]: 50%, Goal [OK, Version 6]: 90%.
OK: Defined As: {Condition = Correct Employee, Hours = 0800 to 1700}.
*In this example, the function, F8 is defined* together *with multiple attributes, both benchmark and target. Notice the qualifier 'OK' is defined in a separate statement to make reuse of it easier. This also prevents repetition, saves space and saves time when making changes.*

## 3.8 Further Example/Case Study: Function Specification for an Airborne Command and Control System

This is an extract from the top-level function specification for a real system (The system is now operational and delivered to customers).

Note: Mapping functions in detail is not the prime intention when using Planguage. The aim is to establish an evolutionary plan, which focuses on result delivery to some defined system stakeholders. This aim does not necessarily require *any* 'delivery' of additional functionality! Delivering 'designs,' to just improve performance and resource attributes for existing functionality is quite common. The level of understanding of the functions needed at the planning stage is merely that required to support the system designers and others involved in the requirement specification process. Specifically, this means that a complete, in-depth description of all the system functions and processes is not required. I strongly recommend investigating functions in detail *only as required*, at the design stage of each evolutionary delivery step. (There may well be exceptions to this, but don't waste resources.)

Airborne Command and Control: **ACC**.
Type: System.
Includes: Type: Sub-system:
**M**: Mission.
**P**: Planning.
**D**: Data Handler.
**C**: Communications Intelligence.

**ACC.D.MOP**: ''MOP stands for 'Manual Operation(s)'.''
Type: Function.

| Includes: | |
|---|---|
| BIT: | Provide 'Built In Test.' |
| DATAB: | Provide database diagnostics. <Various levels of checking>. Not including <on mission>. |
| DATAELEMENT: | Check <Data element reasonableness> when <on mission>. |
| DATALINK: | Interchange data links manually by operators. ''*Component from our mother company*.'' |
| DIAG: | Display all faults to operator and log on file. |
| DISPLAY: | Display error and fault detection data to operator. |
| ESM: | Display error messages from communications/ non-communications system. |
| FMS: | Display any loss of data from Flight Management System. |
| HEARTBEAT: | Supervise computer node <status> by heart-beat <signaling>. |
| INIT: | Test data destructive HW when initializing the <system>. |
| LINK: | Display <status information> of the data links. |
| LOG: | <Save on file> fault detection data and detailed test information. |
| PRINTER: | Report <printer status> from the AX-BUS when any fault occurs. |
| RADAR: | Display loss of data from radar. |

This was from the first draft of the function specification. Many concepts are marked with <fuzzy brackets> and require further work to precisely define them.

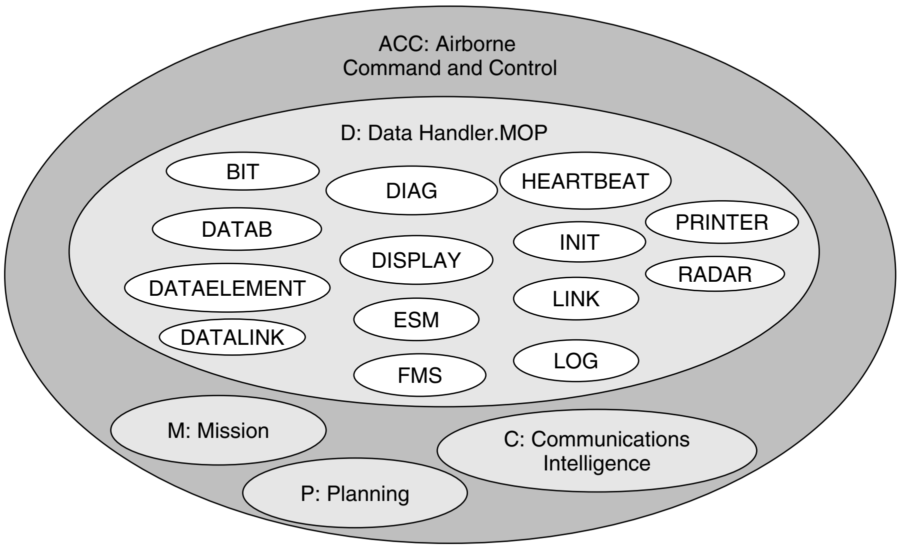

**Figure 3.2**
Functions within the Airborne Command and Control (ACC) System.

Here is an example of a function requirement:
*(Notice the use of a Planguage template. The template parameters are given in **bold**. See the next section for a more detailed specification of this template.)*

**Example of a Function Requirement**

**Tag**: DATADIAG:
**Type**: Function Requirement. ''Note, DATADIAG is not 'real', it is an example made up using the basic ideas named in the real case above. It is for teaching purposes only.''
**Version**: October 7, 2004 21:38.
**Owner**: Quality Assurance Division.
Implementer: Database Team.
**Stakeholders**: Quality Assurance Division, Maintenance Support.
**Gist**: Obtain Database Diagnostics.
Description:
S1: To monitor database quality. <Various levels of checking>.
S2: To report database diagnostics.
S3: To integrate with the automatic recovery system.
S4: To run in parallel with the operational use of the database as a background function.
S5: Monitoring operation to be optional. For example, to be off when <on mission>.
S6: Monitoring operation to be user-driven by parameters to enable selected sampling of specific classes of database records, data elements and relationships.
**Supra-function**: ACC.D.MOP. ''This is the specification of the supra-function of DATADIAG from some viewpoint.''
**Sub-functions**: None specified.
**Supports**: {System Recovery, Bug Maintenance, Database Integrity}.
**Assumptions**:
A1: This sub-system will not degrade operational database performance by more than 5%.
A2: It will be cheaper to automate this function than to do analysis manually.
A3: It will be faster and more reliable than manual checking.
**Dependencies**: D1: The database system itself must be defined and operational.
**Risks**: R1: Failure to update this function in parallel with the database structure.
**Priority**: This function must be available to some degree in first customer use releases. It will also be used in pre-release systems testing to some undefined degree.
**Test**: T1: This function shall be used in system testing and an early version of it can and should be made available in parallel with the development of the database itself. The function shall be tested by insertion of artificial database defects, and shall discover 100% of these.
**Financial Budget**: The cost of developing and maintaining this function is assumed to be between 10% and 50% of the cost of building and maintaining the database software in total.
Function Intranet Location: ACC. Software.DB-Diagnosis.

## 3.9 Diagrams/Icons: Function and Function Requirement Specification

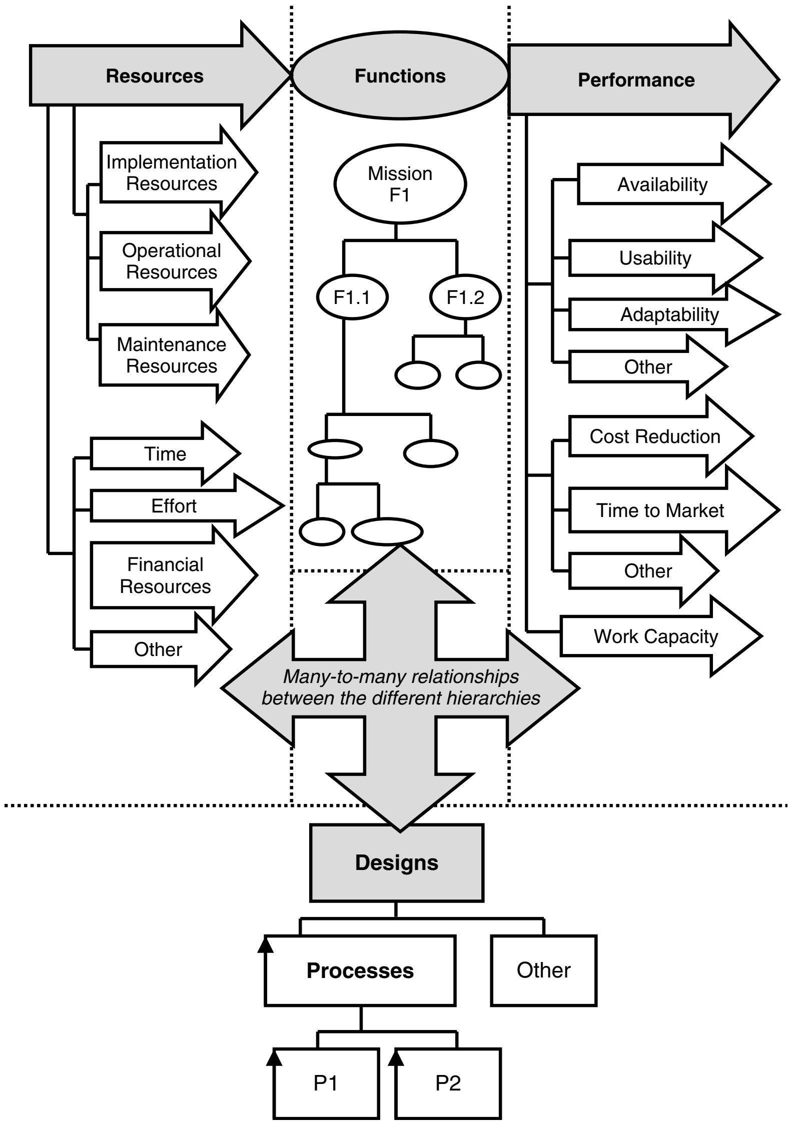

**Figure 3.3**
This shows the four main system attribute types: resource, function, performance and design. It also shows the processes, which implement the functions. Using Planguage, the complex relationships amongst these four different types can be specified. For example, a specific performance level might apply only to a handful of functions rather than the entire system, or a function might be implemented by several processes, or different resources can be specifically allocated to different functions.

**Template for Function Specification <with hints>**

**Tag**: <Tag name for the function>.
**Type**: <{Function Specification,
Function (Target) Requirement,[6]
Function Constraint}>.
========================= **Basic Information** ========================
**Version**: <Date or other version number>.
**Status**: <{Draft, SQC Exited, Approved, Rejected}>.
**Quality Level**: <Maximum remaining major defects/page, sample size, date>.
**Owner**: <Name the role/email/person responsible for changes and updates to this specification>.
**Stakeholders**: <Name any stakeholders with an interest in this specification>.
**Gist**: <Give a 5 to 20 word summary of the nature of this function>.
**Description**: <Give a detailed, unambiguous description of the function, or a tag reference to some place where it is detailed. Remember to include definitions of any local terms>.
=========================== **Relationships** ==========================
**Supra-functions**: <List tag of function/mission, which this function is a part of. A hierarchy of tags, such as A.B.C, is even more illuminating. Note: an alternative way of expressing supra-function is to use Is Part Of>.
**Sub-functions**: <List the tags of any immediate sub-functions (that is, the next level down), of this function. Note: alternative ways of expressing sub-functions are Includes and Consists Of>.
**Is Impacted By**: <List the tags of any design ideas or Evo steps delivering, or capable of delivering, this function. The actual function is NOT modified by the design idea, but its presence in the system is, or can be, altered in some way. This is an Impact Estimation table relationship>.
**Linked To**: <List names or tags of any other system specifications, which this one is related to intimately, in addition to the above specified hierarchical function relations and IE-related links. Note: an alternative way is to express such a relationship is to use Supports or Is Supported By, as appropriate>.
=========================== **Measurement** ==========================
**Test**: <Refer to tags of any test plan or/and test cases, which deal with this function>.

==================== **Priority and Risk Management** ====================
**Rationale**: < Justify the existence of this function. Why is this function necessary? >.
**Value**: <Name [Stakeholder, time, place, event>]: <Quantify, or express in words, the value claimed as a result of delivering the requirement>.
**Assumptions**: <Specify, or refer to tags of any assumptions in connection with this function, which could cause problems if they were not true, or later became invalid>.
**Dependencies**: <Using text or tags, name anything, which is dependent on this function in any significant way, or which this function itself, is dependent on in any significant way>.
**Risks**: <List or refer to tags of anything, which could cause malfunction, delay, or negative impacts on plans, requirements and expected results>.
**Priority**: <Name, using tags, any system elements, which this function can clearly be done *after* or must clearly be done *before*. Give any relevant reasons>.
**Issues**: <State any known issues>.
======================== **Specific Budgets** ========================
**Financial Budget**: <Refer to the allocated money for planning and implementation (which includes test) of this function>.

**Figure 3.4**
A template for Function Specification.

[6] Note: By default, a 'Function Requirement' is assumed to be a 'Function Target'.

## 3.10 Summary: Function and Function Requirement Specification

Functions are '*what*' a system does. The concept of a pure 'function' does not include information about the function's performance attributes (how *well* a function is done); nor about the function's conditions [when, where, if]; nor about design ideas (*how*, a function achieves its attributes *at the required levels*).

My view of the discipline of functions is that they are 'boring, but essential, necessities.' They are the basics of the business or field you are dealing with, and probably exactly the same as those of your competitors in the same market.

The 'real competitive action' lies in identifying the interesting (competitive) performance and resource *attributes* for the functions, then establishing their required *competitive levels* and, then finding interesting ways (designs) to achieve them.

So, you can view functions as providing the framework 'supporting' the performance and resource attributes necessary for winning.

Any attempt to implement a function without trying to gain control over its performance and cost attributes, will result in unplanned, uncontrolled and thus probably undesired attributes. You must control attributes of functions to control the 'Risks.'

Many of the common problems, which systems engineers experience (such as deadline control, cost overruns and bad quality) are, in my view, significantly caused by:

- Specifying poorly-justified and insufficiently-detailed 'design' and calling it 'Function Requirements.'
- Articulating the performance and costs of functions in ways that can't be measured or tested.
- Focusing on testing functions alone, rather than the key stakeholder-value performance and cost attributes.

**Functions are merely real-world reference points. They are not the interesting 'problem' for competitive systems engineering**.