Post 5.0 -- Repetitive Stimulus Sets

Functionality Tree Recap

A functionality tree is a horizontal tree-like notation that serves as an implementation-independent schematic of the external interface of a system. The relationship depicted in a functionality tree is the New Stimulus Set Response (NSR) by which a stimulus of one stimulus set activates another stimulus set. The NSR stimulus activation relationship among stimulus sets gives the external interface a hierarchical structure. Because Freedom recognizes a specification of the external interface as a specification of system requirements, the hierarchical architecture of the external interface is also the architecture of the functionality requirements of the system.

History List Expansion in the AAR Functionality Tree

In the last post, expansion of the 'history list' stimuli of the Adobe Acrobat Reader (AAR) functionality tree was left as an exercise. The following is a subset of the AAR Functionality Tree showing the history list stimuli in unexpanded form:

AAR Functionality Tree

Level 0	Level 1	Level 2
Main Opts	File SS	
---	---	
File	Open	
	Close	
	Save A Copy	
	Document Properties	
	Document Security	
	Page Set Up	
	Print	
	history list	***EXERCISE***
	Exit	
Edit		
Document		
View		
Window		
Help		

Experimenting with the history list stimuli as part of performing the history list expansion reveals that a history list stimulus has an extensive NSR response -- an entire new instance of AAR is launched in a separate window.

How do we deal with this in the functionality tree? One approach is to replicate the entire functionality tree to the right of the 'history list" stimuli, like so:

AAR Functionality Tree

Level 0	Level 1	Level 2	Level 3
Main Opts	File SS		
---	---		
File	Open		
	Close		
	Save A Copy		
	Document Properties		
	Document Security		
	Page Set Up		
	Print		
		Main Opts	File SS
		---	---
	history list	File	Open
			Close
			Save A Copy
			Document Properties
			Document Security
			Page Set Up
			Print
			history list ***EXERCISE***
			Exit
	Exit		
		Edit	
		Document	
		View	
		Window	
		Help	
Edit			
Document			
View			
Window			
Help			

However, this approach quickly leads to trouble since the history list expansion needs to be replicated yet again, and again, and again, ... The problem is infinitely recursive.

Fortunately, there is an easy way to handle such cases. The solution is to expand the functionality tree using a referential placeholder:

AAR Functionality Tree

Level 0	Level 1	Level 2
Main Opts	File SS	
---	---	
File	Open	
	Close	
	Save A Copy	
	Document Properties	
	Document Security	
	Page Set Up	
	Print	
		Main Opts
		---
	history list	(reused SS)
	Exit	
Edit		
Document		
View		
Window		
Help		

In the above (correct) example, the name of the Main Ops stimulus set is placed to the right of the 'history list' stimuli (since a history list stimulus triggers the Main Ops stimulus set), but its stimuli are not explicitly listed. In their place, a note is inserted indicating that this is a reused instance of a stimulus set that has been defined previously.

There are several advantages to this approach:

1. Mechanical recording problems associated with recursion are avoided.
2. Any future changes to the stimuli in the explicit Main Ops stimulus set definition at Level 0 are automatically "inherited" by the stimulus set to the right of 'history list' (as well as by any other Main Ops references, regardless of where they may occur), making maintenance of the functionality tree easier.
3. The referential placeholder serves as an indication that the referenced stimulus set has greater than single-use utility, i.e., the stimulus set may be a candidate for requirements reuse.

Repetitive and Reusable Stimulus Sets

In the above example, the referential placeholder to the Main Ops stimulus set at Level 2 may be considered as either a repetitive or a reusable stimulus set. The distinction is as follows:

A repetitive stimulus set is a stimulus set that can be activated by stimuli in two or more different stimulus sets of an application.

A reusable stimulus set is a stimulus set (SS) that is activated by stimuli in two or more different application programs. (Note: different physical instances of the SS are typically activated.)

A reusable stimulus set may also be repetitive, that is, it may, if desired, be activated from multiple stimulus sets of the same application. However, a repetitive stimulus set is not necessarily reusable, i.e., just because a stimulus set is activated from multiple places in a single application does not necessarily imply it will have utility in an entirely different application.

As an example, consider the Main Ops stimulus set above. It is clearly repetitive due to being activated by stimuli in two different stimulus sets. However, is it reusable? That is, is the entire AAR application useful as a subtree in an entirely different application program? It certainly could be. For instance, it is easy to envision a mail reader which invokes AAR to read pdf files sent as attachments. How easy this may be in practice depends on the implementation details of the AAR application in general, and the programmatic invocation interface to the Main Ops stimulus set in particular. One would have to have programming level documentation to AAR to determine if Adobe implemented AAR in a way that promoted easy reuse from other applications.

Had Adobe used a functionality tree to design the AAR external interface, one of the first clues that AAR might be a candidate to implement as a reusable component is the fact that Main Ops is repetitive in the functionality tree. While not all repetitive stimulus sets are reusable, all reusable stimulus sets are repetitive across different applications. Hence, the fact that a stimulus set is repetitive within a single application serves as a flag that the stimulus set should be evaluated more closely for possible utility as a reusable stimulus set.

Repetitive stimulus sets are significant in another respect as well. As we shall see in the next post, repetitive stimulus sets influence the number of behavior tables that need to be specified.

Rick Lutowski

Post 6.0 -- Behavior Tables
Repetitive Stimulus Sets Recap

A repetitive stimulus set is a stimulus set that is activated by stimuli in two or more different stimulus sets of an application. A reusable stimulus set is a stimulus set that is activated by stimuli in two or more different applications. Repetitive stimulus sets in a functionality tree are a flag of potential requirements reusability. Repetitive stimulus sets are recorded in a functionality tree by reference to a defining instance. The defining instance may be in the same or a different functionality tree.

Responses to Stimuli

Specification of a partial functionality tree defines an initial set of stimuli for the application. The second step in the three step Freedom requirements specification process is to specify the responses to those stimuli.

Responses may be characterized by their visibility, desirability, and prescriptiveness.

There are two kinds of responses based on their visibility: external internal. External responses are detectable outside the software system black box, and therefore are part of the system's requirements information set. Internal responses are not detectable outside the software system black box, and thus are part of the system's design and implementation information sets.

There are two kinds of responses based on desirability: normal and error responses. Normal responses are the expected response to a stimulus in normal operation. Error responses are undesirable or exceptional responses, and generally indicate a deviation from normal operation.

Freedom explicitly recognizes a subtype of normal response, called a new stimulus response (NSR). NSR is treated as a separate category of normal response due to its special relevance to the functionality tree and stimulus set organization, as noted in the previous post. A NSR is always an external response, i.e., it is always part of the requirements information. (Note: An Error NSR response is also a logical possibility, but is not significant enough to warrant a separate type.) This results in a total of three types of responses based on desirability: normal, NSR, and error.

There are two kinds of responses based on prescriptiveness: binding and guidance. Binding responses are specified by the customer, and must be implemented by the developers exactly as specified unless a change to the response is explicitly agreed to by the customer.

Guidance responses are specified at the discretion of the development team. They may be changed at will by the development team using whatever specification control mechanisms the team desires or, perhaps, is imposed on the team by a project management methodology external to Freedom. As far as Freedom is concerned, guidance responses are completely non-

binding on the development team and need not be implemented as specified, or need not be implemented at all.
The only important responses to Freedom are the binding responses. Providing the binding responses are met, any non-binding response may be used in support at the complete discretion of the development team.

Response Classification

The above response attributes may be combined into a composite list of response types to produce what has been variously called a "classification list" (Lutowski, 1978 [not on web]) or an "option field" (Warfield, mid-80s):

Unexpanded attributes:

Visibility	Desirability	Prescriptive
external	normal	binding
internal	NSR	guidance
	error	

Expanded attributes (Classification List):

Visibility	Desirability	Prescriptive	Response Category
external	normal	binding	Normal
external	normal	guidance	invalid, requirements always binding
external	NSR	binding	NSR
external	NSR	guidance	invalid, requirements always binding
external	error	binding	Error
external	error	guidance	invalid, requirements always binding
internal	normal	binding	D&I Constraint, normal
internal	normal	guidance	D&I Guidance, normal
internal	NSR	binding	invalid, NSR always external
internal	NSR	guidance	invalid, NSR always external
internal	error	binding	D&I Constraint, error
internal	error	guidance	D&I Guidance, error

Response Classification List:

Visibility	Desirability	Prescriptive	Response Category

external	normal	binding	Normal
external	NSR	binding	NSR
external	error	binding	Error
internal	normal	binding	D&I Constraint, normal
internal	normal	guidance	D&I Guidance, normal
internal	error	binding	D&I Constraint, error
internal	error	guidance	D&I Guidance, error

Behavior Tables

The response behavior to a stimulus is recorded in a notation called a Behavior Table (BT). One behavior table is created for each unique stimulus set in the functionality tree. The table contains a. One column for each Response Category defined above, and b. One row for each stimulus of the stimulus set.

In addition, the BT contains a column for Performance-Accuracy-Precision (PAP) attributes of the response. Performance includes any time restrictions on the response, such as maximum time to respond, or required periodicity. Accuracy includes requirements relating to maximum allowed deviation from a correct answer. For example, NP-incomplete problems such as the 'traveling salesman' shortest path problem are not solvable in reasonable amounts of time, but may be solvable rapidly if, say, a 20% deviation from the true shortest path is allowed. Precision relates to requirements such as number of significant digits in a result, or if an integer result is acceptable rather than a floating point result.

The general format of a behavior table is:

Behavior Table for SS _____

Stimulus	Normal	NSR	Error	D&I Constraint	D&I Guidance	PAP
stimulus						
:						
stimulus						

As the above format shows, each behavior table defines responses to all the stimuli in a single stimulus set. Each row of the behavior table specifies the response to one stimulus.

The above table merges normal and error responses for internal (Constraint and Guidance) responses since these are of less significance to requirements specification than external

responses. These columns may be expanded to segregate normal and error internal responses if desired.

Finally, because a behavior table is created for each _unique_ stimulus set in the functionality tree, one behavior table represents all instances of a repetitive or reusable stimulus set. It makes no difference how many instances of the stimulus set there may be, or which functionality tree (application program) it may occur in, only one behavior table need be defined for a repetitive or reusable stimulus set.

Example Behavior Tables

For an example of a behavior table for an actual application program. we again turn to our example of Adobe Acrobat Reader (AAR) 5.08. Since AAR was not developed using Freedom, BTs were never created for it, so we will attempt to reverse engineer one. Our example will show only requirements (externally visible) responses. Reverse engineering the total response (external+internal) requires internal visibility, i.e., source code. Therefore, we will ignore the Constraint and Guidance internal response columns.

For our example, we will focus on the Level 0 Main Ops stimulus set of the AAR functionality tree:

AAR Functionality Tree

Level 0	Level 1	Level 2
Main Opts	File SS	
---	---	
File	Open	
	Close	
	Save A Copy	
	Document Properties	
	Document Security	
	Page Set Up	
	Print	
		Main Opts
		---
	history list	(reused SS)
	Exit	
	Edit SS	
	- - - - - -	
Edit	...	
	Document SS	
	- - - - -	
Document	...	

	View SS
	- - - - -
View	...
	Window SS

Window	...
	Help SS
	- - - - - - -
Help	...

A minimal BT showing the basic NSR responses can be created directly from the functionality tree:

AAR Behavior Table for Main Opts SS

Stimulus	Normal	NSR	Error	D&I Constraint	D&I Guidance	PAP
File						
		activate File SS				
Edit						
		activate Edit SS				
Document						
		activate Document SS				
View						
		activate View SS				
Window						
		activate Window SS				
Help						
		activate Help SS				

Experimentation with these six stimuli shows no other externally detectable response. However, one addition is needed to complete this BT.

All behavior tables may optionally include an initialization stimulus. The response to the initialization stimulus is any action taken by the stimulus set as a direct result of its activation. The initialization stimulus is associated with the activation relationship of the FT,

and is not usually shown as an explicit stimulus in the functionality tree. It appears explicitly in the BT if a specific response is associated with activation. If there is no activation response, the initialization stimulus need not appear in the BT.

In the case of AAR, experimentation shows that including the name of a pdf file on the command line from which AAR is run results in that pdf file being displayed on AAR start up. There are a couple different ways this can be handled in Freedom, but the simplest for our example is to treat this response as an initialization response of Main Ops:

AAR Behavior Table for Main Opts SS

Stimulus	Normal	NSR	Error	D&I Constraint	D&I Guidance	PAP
init	INPUT pdf_file pdf_file_name pdf_file_content IF pdf_file_name specified IF pdf_file is found and valid display Main Ops activate Right Display Region SS activate Left Display Region SS show pdf_file_content in Right Display Region 1.0 sec max ENDIF IF pdf_file not found or invalid activate File Error SS display Main Ops ENDIF ENDIF IF pdf_file_name not specified display Main Ops ENDIF					
File		activate File SS				
Edit		activate Edit SS				
Document		activate Document SS				
View		activate View SS				
Window						

		activate Window SS					
--------	--------	--------	--------	----------	--------	--------	
Help							
		activate Help SS					
--------	--------	--------	--------	----------	--------	--------	

Now the BT is starting to look more interesting. Let's look more closely at the initialization response behavior:

	Stimulus	Normal	NSR	Error	D&I Constraint	D&I Guidance	PAP
	--------	--------	--------	--------	----------	--------	--------
01	init	INPUT					
02		pdf_file					
03		pdf_file_name					
04		pdf_file_content					
05							
06		IF pdf_file_name specified					
07		IF pdf_file is found and valid					
08		display Main Ops					
09		activate Right Display Region SS					
10		activate Left Display Region SS					
11		show pdf_file_content in Right Display Region 1.0 sec max					
12		ENDIF					
13		IF pdf_file not found or invalid					
14		activate File Error SS					
15		display Main Ops					
16		ENDIF					
17		ENDIF					
18		IF pdf_file_name not specified					
19		display Main Ops					
20		ENDIF					
	--------	--------	--------	--------	----------	--------	--------

Line 6:
The conditional nature of the response is captured using a typical IF clause. (Let's call it a 'clause' not a 'statement' because we are not programming here, but capturing behavior using what amounts to structured English.) The phrase after the IF denotes some _externally-visible_ condition, in this case the existence of a file name associated with launching AAR. Checking for this file name is part of normal operation, so the statement is aligned with the Normal external response column.

Lines 7-12:
If the file is found and is a valid pdf file, Main Ops is displayed and the Left and Right Display Region SSs are activated. (Note: Main Ops has already been activated because we are responding to its initialization stimulus. It merely remains to make it visible.) The content of

the pdf file is then displayed in the Right Display Region. See post 3 for a discussion of the Right and Left Display regions.

For the sake of illustration, we pretend the customer specified that the time to launch AAR with a pdf file specified not exceed the time to launch without a pdf file specified by more than 1 second. This requirement is captured by associating a Performance requirement of 1.0 second maximum with the 'show pdf file content' response.

Notice that the response behavior clauses are phrases of the form "verb noun" or "action quantity" or "operation object", and are NEVER prose. This form results in response behavior clauses that are clear and succinct. Following these form guidelines:

1 helps reduce ambiguity, misinterpretation and misunderstanding inherent in prose,
2 helps eliminate grammatical wordsmithing which diverts focus from the goal of defining response behavior, and
3 reduces the 'conceptual distance' between requirements specification and implementation, i.e., the behavior specs read like English fragments to customers while reading like code fragments to developers.

Lines 1-4:
Freedom's response recording convention is to list, at the top of the behavior spec for each stimulus, INPUT, OUTPUT, and LOCAL quantities referenced by the response behavior clauses for the stimulus. The quantity names specified under the INPUT, OUTPUT, and LOCAL headings should be used consistently throughout the behavior specification clauses. This helps avoid referring to the same quantity be multiple names, which can lead to confusion and misunderstandings.

When listing the quantities, it is very important to subject each quantity to a test:

"Is this quantity externally visible?"

If the answer is "no", i.e., the quantity is internal to the black box, then that quantity must appear in the LOCAL list, and any clauses it appears in must _not_ appear in any external response column; they may only appear in a Constraints or Guidance column. If the quantity was defined by the customer as implementation direction, the quantity and all clauses it appears in are listed under the D&I Constraints column and are binding, If it was defined by a developer, it and all clauses it appears in are listed under the Guidance column and are non-binding. Thus, explicitly listing all quantities as INPUT and OUTPUT (external) or LOCAL (internal) helps avoid the serious error of inadvertently specifying internal responses as requirements.

Lines 13-16
If the file is not found or is not a valid pdf file, the File Error stimulus set is activated. Main Ops is then displayed. Since there is no valid pdf file specified, the Left and Right Display Regions are not activated.

Note that activation of the File Error SS is both an Error and an NSR response. Since its status as a NSR is clear, we list it in the Error external response column to highlight its exceptional nature.

Line 18-20:
If no file name is specified on AAR start up, the response is simply to display Main Ops.

Iteration with Functionality Tree

Recall that line 14 of the init response specified activation of the File Error SS. Because the existence of this stimulus set was identified during response behavior experimentation, this stimulus set did not previously appear in the functionality tree. We must now go back and correct this oversight.

The File Error SS consists of a single stimulus labeled "OK". Also, the stimulus set is activated from the initialization stimulus of Main Ops. This results in the following revised FT:

AAR Functionality Tree

Level 0	Level 1	Level 2
Main Opts	File Error SS	
----------	----------	
Init	OK	
	File SS	

File	Open	
	Close	
	Save A Copy	
	Document Properties	
	Document Security	
	Page Set Up	
	Print	
		Main Opts
		----------
	history list	(reused SS)
	Exit	
	Edit SS	
	- - - - - -	
Edit	...	
	Document SS	

	- - - - -
Document	...
	View SS
	- - - - -
View	...
	Window SS

Window	...
	Help SS
	- - - - - - -
Help	...

The above makes the FT consistent with the BT, so nominally all is well. However, explicit appearance of a stimulus set initialization stimulus in the FT is a warning flag that the requirements specification may not be quite right. This is, in fact, the case. The above FT is an over-simplification of the actual FT for AAR, intended to keep the posts shorter, less complex, and less confusing. The need to list init in the FT is the result of my having omitted something that would be important were we seriously attempting to model the AAR requirements. The omission is this:

Main Ops does not really occur at Level 0, i.e, it is not the first stimulus set to become active when AAR is launched. The first stimuli that AAR is primed to detect are command line arguments -- another set of human user stimuli totally separate from the GUI. The pdf_file_name quantity in the Main Ops BT is received in the form of stimuli (input) from the command line. The AAR command line arguments form a command-data protocol that results in another entire sub-branch of the FT, starting at the real Level 0. The activation of Main Ops is part of the response to one or more of these command line stimuli. So is activation of File Error SS. In fact, most if not all of the Main Ops init responses are properly responses to command line stimuli, and not the Main Ops init stimulus. Where the command line part of the FT completed, the need for including the Main Ops init response in the Main Ops BT would likely disappear. There would also be no need to include init in the FT, thus lowering the warning flag that something was amiss.

However, this omission and the resulting anomalies did not prevent the above behavior table from serving its instructional purpose, and probably helped. It afforded an opportunity to introduce the init stimulus and associated issues. The response clauses to the init stimulus provided a representative picture of Freedom's response behavior recording syntax and conventions.

Behavior Recording Notation Variants

One final word about the behavior recording syntax -- it is not locked in stone. Personally, every time I create behavior tables for a new project, I find myself trying a new variant of the recording syntax for use in the table "cells." The exact form of structured English (or PDL or whatever one wishes to call it) is less important than the information it is used to capture, and the characteristics of the capture process. Until such time as a BT-specific toolset is available that requires conformance to a specific behavior recording syntax, there is value in experimenting with different recording notation variants. After all, the software research community has been seeking improved ways of recording response behavior for decades without having found a notation that even a minority can agree upon. Thus, it would be highly presumptuous
to claim that the above IF-style syntax is the best, or even the recommended, notation.

What is strongly recommended is to keep the following guidelines in mind when adopting a response recording notation. A response behavior recording notation should:

A. be understandable to customers;
B. reduce the 'conceptual distance' to implementation for developers;
C. greatly reduce, if not eliminate, the need to 'wordsmith';
D. reduce the potential for ambiguity and misinterpretation;
E. avoid the use of multiple terms for the same quantity; and, perhaps most importantly,
F. provide mechanisms to help avoid specification of internal behavior as requirements.

Rick Lutowski

From Brad Appleton
Rick Lutowski wrote:
> Freedom explicitly recognizes a subtype of normal response, called a new stimulus response (NSR). NSR is treated as a separate category of normal response due to its special relevance to the functionality tree and stimulus set organization, as noted in the previous post. A NSR is always an external response, i.e., it is always part of the requirements information. (Note: An Error NSR response is also a logical possibility, but is not significant enough to warrant a separate type.) This results in a total of three types of responses based on desirability: normal, NSR, and error.

NSR looks interesting because it is either a "plug point" e.g. "hot spot") between other stimulus-sets, or else potential "coupling" point. So does an NSR increase, decrease, or have no effect upon the coupling between the stimulus-sets it appears to "connect"?

If a requirements change is in the form of a new/changed NSR, what kind of ripple effect does that tend to have on the functionality trees? On the behavior tables? How is that more or less than when a requirements change is in the form of some new/changes stimulus that is not part of (nor connected to) an NSR?

I'm also curious as the to impact (ripple effect or lack thereof) of "requirements" changes upon the behavior tables.

Post 7.0 -- Functionality Modules

Behavior Table Recap

A behavior table (BT) is created for each unique stimulus set of the functionality tree for the purpose of specifying the response behavior of its stimuli. A BT contains one row for each stimulus of the stimulus set, and one column for each type of external and internal response. Types of external responses (binding) are: Normal, NSR, and Error. Types of internal responses are: D&I Constraints (binding) and D&I Guidance (non-binding). A PAP column permits specification of Performance, Accuracy, and Precision attributes for each clause of the behavior specification. The behavior specification clauses may use any notation that meets the response recording notation guidelines. Both Mills and the author suggest PDL or "structured English." Other notations such as formal methods are not ruled out, but prose is discouraged.

Requirements Encapsulation Design Rule

Existence of a partial functionality tree and associated behavior tables permits the start of design and implementation for the specified part of the application. As we saw in post 1, the guiding design rule is:

> Create one functionality module for each unique stimulus set of the functionality tree.

To better illustrate application of this design rule, we turn to the functionality tree for Adobe Acrobat Reader (AAR) 5.08. A composite of all the AAR functionality tree fragments developed in previous posts is shown below. Since the design rule focuses on stimulus sets rather than stimuli, all stimuli that do not activate additional stimulus sets have been left out so as to simplify the functionality tree.

AAR Functionality Tree (partial, most stimuli not shown)

Level 0	Level 1	Level 2	Level 3
Main Opts	File SS	Open SS	
---	---	---	
File	Open	...	
		Save A Copy SS	
		- - - - - - - -	
	Save A Copy	...	
		Document Properties SS	Summary SS
		---	---
	Document Properties	Summary	...

			Fonts SS ----------
		Fonts ...	...
		Document Security SS ----------	
	Document Security	...	
		Page Set Up SS ----------	Paper SS --------
	Page Set Up	Paper ...	...
		Print SS --------	Browse SS ---------
	Print	Browse ...	...
		Main Opts - - - - - - -	
	history list ””	(reused SS)	
	Edit SS - - - - - -		
Edit	...		
	Document SS - - - - -		
Document	...		
	View SS - - - - -		
View	...		
	Window SS ----------		
Window	...		
	Help SS - - - - - - -		
Help	...		

The above FT can be simplified further by showing only the stimulus set names. This makes the stimulus set activation structure (defined by the New Stimulus Set Responses) of the FT more apparent.

AAR Functionality Tree (partial, stimulus sets only)

Level 0	Level 1	Level 2	Level 3
Main Opts	File SS	Open SS	
		Save A Copy SS	
		Document Properties SS	Summary SS
			Fonts SS
		Document Security SS	
		Page Set Up SS	Paper SS
		Print SS	Browse SS
		Main Opts (reused)	
	Edit SS		
	Document SS		
	View SS		
	Window SS		
	Help SS		

The total number of stimulus sets can now easily be determined. The above AAR partial functionality tree contains 18 stimulus sets, of which Main Ops has one repetitive instance. Hence, the number of unique stimulus sets is 17. The requirements encapsulation design rule thus implies that 17 functionality modules should be created to implement the above portion of the AAR functionality tree. Each functionality module encapsulates one stimulus set, where a stimulus set is a highly cohesive collection of requirements (stimulus-response pairs.)

Functionality and Common Service Modules

Of course, an application consists of more than just functionality modules. Other modules encapsulate data structures, algorithms, and hardware interfaces just as in current OO design. In Freedom, however, these "traditional" information-hiding modules acquire a new meaning to their existence. They exist to provide common services in support of the required response behaviors specified in the behavior tables and implemented by the functionality modules. Because of this role, Mills called data and hardware-hiding OO modules "common service" modules (CSMs). Freedom adopts this terminology.

From a design architecture standpoint, the difference between Freedom and current OO is:
a. In Freedom, the CSMs exist to support the FMs. FMs directly satisfy the requirements.

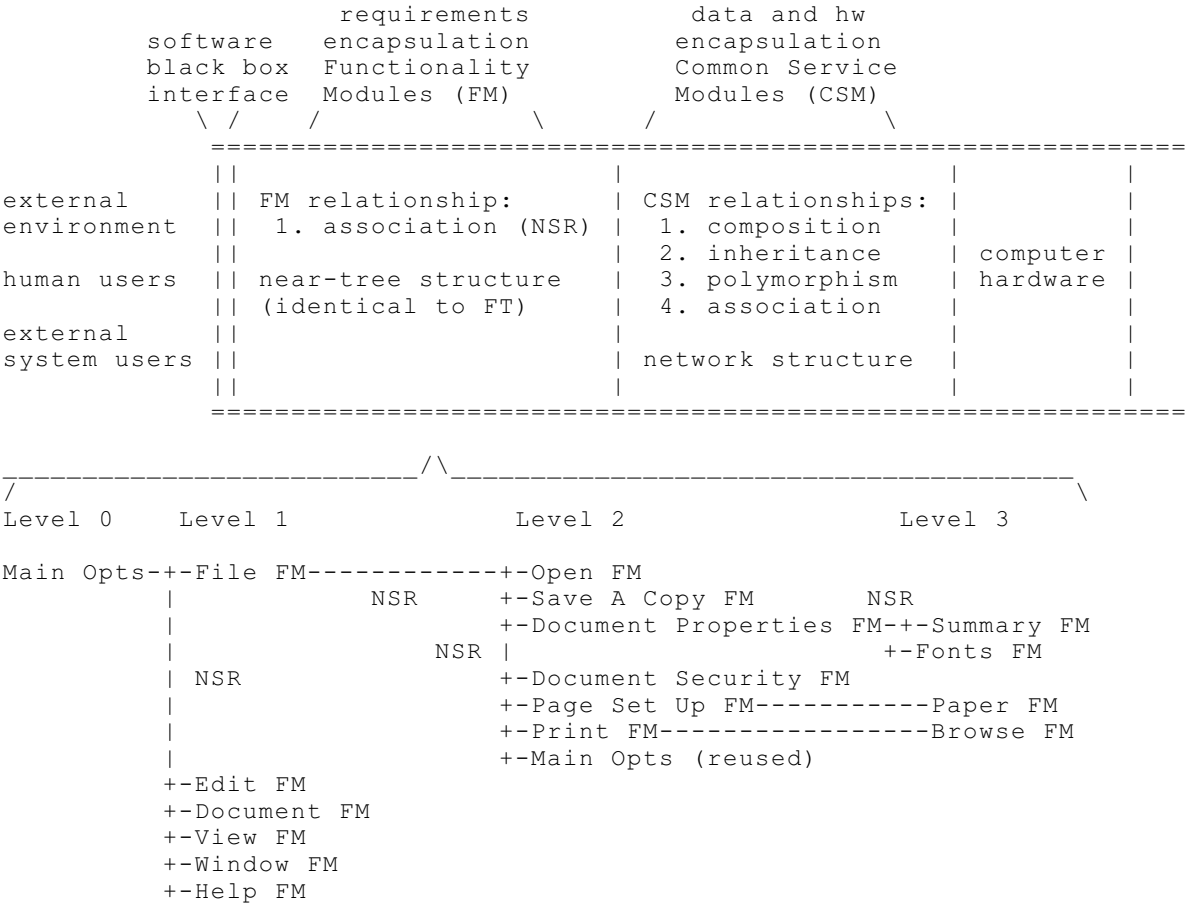
b. In traditional OO, all modules are CSMs because there are no FMs (i.e, requirements are not encapsulated). Thus, the CSMs exist to support one another while simultaneously satisfying the requirements.

The above would seem to imply that CSMs in Freedom are less complex than CSMs in current OO since the burden of directly satisfying the requirements has been removed from them. This is true for some CSMs, but not for all. For example, a printer device driver module or a
stack data structure module would be no different in Freedom than is currently the case. Modules that are conceptually closer to the external interface, however, may be significantly different in Freedom than in current practice due to the existence of the functionality modules.

Canonical Design Architecture

The existence of both functionality modules and common service modules in Freedom gives rise to a layered design architecture that is highly generic, i.e., it applies to all programs developed using Freedom. Because it applies to all programs, Freedom calls it the Canonical Design Architecture, where canonical means standard or orthodox (in the context of Freedom).

The diagram below depicts the Freedom Canonical Design Architecture.



The implications of the requirements encapsulation design rule go beyond merely creating a functionality module for each unique stimulus set. The New Stimulus Set relationship among the stimulus sets must also be honored (after all, it is part of the requirements spec also.) The NSR relationship among the stimulus sets carry over to the functionality modules (FM), and organize the FMs in the same manner as it organizes the stimulus sets. Thus, the requirements architecture of the stimulus sets is precisely mirrored in the design architecture of the requirements encapsulating functionality modules, as illustrated in the bottom half of the Canonical Design Architecture diagram, where the functionality module architecture implementing the sample AAR functionality tree is illustrated.

The architectural symmetry between requirements and design in Freedom makes maintainability easier because maintainers need to learn fewer architectures. Once maintainers learn the architecture of the external interface (information that is needed to properly maintain the interface), they also know the architecture of the implementing design architecture. The benefits work in the other direction as well -- when a developer or maintainer makes a change to the requirements architecture, no additional work is necessary to know the impact on the functionality layer of the design architecture. Thus, the symmetry benefits initial development as well as subsequent maintenance. However, because maintenance cost exceeds development cost by up to 4:1, the primary beneficiaries of Freedom's architectural symmetry are maintainers.

Functionality Modules

A functionality module is an object-oriented program unit that encapsulates the requirements (stimulus-response pairs) of one stimulus set. A FM is like any other OO module in that it consists of methods and module-wide field data. The main difference from current OO modules is that the encapsulated information is requirements (stimulus-response pairs) instead of data structures or hardware interfaces.

Encapsulating the stimulus-response pairs of a stimulus set involves creating code for:

a. the external interface protocols that exhibit the stimuli;
b. the data and algorithms that implement stimulus detection;
c. the data and algorithms that implement response generation;
d. the external interface protocols that exhibit the responses.

To illustrate the general structure of a functionality module, we will start with the behavior table for the Main Ops stimulus set from post 6. For this example, we will use the behavior table as it appears in post 6 and ignore the fact that the init response would likely not be needed in a fully-specified requirements spec for AAR.

AAR Behavior Table for Main Opts SS

Stimulus	Normal	NSR	Error	D&I Constraint	D&I Guidance	PAP
---	---	---	---	---	---	---

```
init          INPUT
                pdf_file
                pdf_file_name
                pdf_file_content

              IF pdf_file_name specified
                IF pdf_file is found and valid
                    display Main Ops
                          activate Right Display Region SS
                          activate Left Display Region SS
                    show pdf_file_content in Right Display Region  1.0 sec max
                ENDIF
                IF pdf_file not found or invalid
                    activate File Error SS
                    display Main Ops
                ENDIF
              ENDIF
              IF pdf_file_name not specified
                display Main Ops
              ENDIF
--------      --------      --------      --------      ----------    --------      --------
File
                          activate File SS
--------      --------      --------      --------      ----------    --------      --------
Edit
                          activate Edit SS
--------      --------      --------      --------      ----------    --------      --------
Document
                          activate Document SS
--------      --------      --------      --------      ----------    --------      --------
View
                          activate View SS
--------      --------      --------      --------      ----------    --------      --------
Window
                          activate Window SS
--------      --------      --------      --------      ----------    --------      --------
Help
                          activate Help SS
--------      --------      --------      --------      ----------    --------      --------
```

A functionality module that implements this behavior table as a GUI menu, per AAR, would have the general structure shown below.

```
//
**************************************************************************
```

```
//              class that encapsulates requirements (stimulus-
//              response) pairs for AAR Main Ops behavior table
class MainOps extends Menu {

// ---------------- data declarations for stimulus GUI protocol -------------
 private MenuItem file;
 private MenuItem edit;
 private MenuItem document;
 private MenuItem view;
 private MenuItem window
 private MenuItem help;

// ---------------- data declarations for response behavior support ---------
 private <any needed variable declaration>;

// ---------------- constructor for MainOps object -------------------------
 public MainOps() {
//              create menu GUI layout
   createMainOpsMenu();
//              perform initialization response
   initResponse();
 }
//              end of constructor for MainOps object

//
=========================================================================
=======
//              Stimulus Methods
//
// ---------------- method for creating menu GUI layout ---------------------
 private createMainOpsMenu() {
//              create stimuli for menu GUI protocol
   file = new MenuItem ("File");
    :
   help = new MenuItem ("Help");
//              layout the stimuli in a horizontal row
   <GUI-specific layout code>
//              enable stimuli
   <GUI-specific stimulus activation code>
 }
//              end of method for creating menu GUI layout

// ---------------- method for detecting stimulus events --------------------
 public handleEvents(Event event) {
   if (<event is from File stimulus>) {
     fileResponse();
   }
```

```
   else if (<event is from Edit stimulus>) {
     editResponse();
   }
   else if (<event is from Document stimulus>) {
     documentResponse();
   }
   else if (<event is from View stimulus>) {
     viewResponse();
   }
   else if (<event is from Window stimulus>) {
     windowResponse();
   }
   else if (<event is from Help stimulus>) {
     helpResponse();
   }
 }
//              end of method for detecting stimulus events

// ========================================================
//              Response Methods
//
// ---------------- method that performs init response ----------------------
 private initResponse() {
//              INPUT
//                pdf_file
   <implementing code>
//                pdf_file_name
   <implementing code>
//                pdf_file_content
   <implementing code>
//
//              IF pdf_file_name specified
   <implementing code>
//                IF pdf_file is found and valid
   <implementing code>
//                  display Main Ops
   <implementing code>
//                      activate Right Display Region SS
   <implementing code>
//                      activate Left Display Region SS
   <implementing code>
//                  show pdf_file_content in Right Display Region
//                  (must respond in 1.0 sec max)
   <implementing code>
//                ENDIF
   <implementing code>
//              IF pdf_file not found or invalid
```

```
   <implementing code>
//                              activate File Error SS
   <implementing code>
//                  display Main Ops
   <implementing code>
//                ENDIF
   <implementing code>
//              ENDIF
   <implementing code>
//              IF pdf_file_name not specified
   <implementing code>
//                display Main Ops
   <implementing code>
//              ENDIF
   <implementing code>
 }
//              end of method that performs init response

// ---------------- method that performs File response ----------------------
 private fileResponse() {
//              activate File SS
   <implementing code>
 }
//              end of method that performs File response

// ---------------- method that performs Edit response ----------------------
 private editResponse() {
//              activate Edit SS
   <implementing code>
 }
//              end of method that performs Edit response

// ---------------- method that performs Document response ------------------
 private documentResponse() {
//              activate Document SS
   <implementing code>
 }
//              end of method that performs Document response

// ---------------- method that performs View response ----------------------
 private viewResponse() {
//              activate View SS
   <implementing code>
 }
//              end of method that performs View response

// ---------------- method that performs Window response --------------------
```

```
 private windowResponse() {
//              activate Window SS
   <implementing code>
 }
//              end of method that performs Window response

// ---------------- method that performs Help response ----------------------
 private helpResponse() {
//              activate Help SS
   <implementing code>
 }
//              end of method that performs Help response

}
//              end of class that encapsulates requirements (stimulus-
//              response) pairs for AAR Main Ops behavior table
//
**************************************************************************
```

Some observations regarding the above functionality module:

1 The functionality module is implemented as an OO class with the same name as the stimulus set. This makes it easy to identify the class that corresponds to a given stimulus set in the requirements.

2 The constructor creates the protocol manifestation of the stimuli, activates stimulus event detection, and calls a method that performs the initialization response, if any, specified by the behavior table.
In effect, the constructor is the primary vehicle for encapsulating implementation details of stimulus creation and activation for the stimulus set, although it will usually rely on common service methods to carry out most of the lower-level work. The constructor also calls a method that encapsulates any required initialization response.

3 A separate method is created to encapsulate the BT-specified response behavior for each stimulus.
The stimulus behavior-encapsulation methods are the primary vehicle for encapsulating response behavior. Therefore, these methods are declared private or otherwise visibility-restricted just as field data declarations are visibility-restricted when encapsulating data structures in objects.

4 As an aid to implementation of the required response behavior, the structured English or PDL from the behavior table is copied verbatim into the response behavior encapsulation methods as comments when the methods are declared.
In the above example, all BT clauses are binding requirements (i.e., external behavior) and so are not further annotated. Were any of the clauses D&I Constraints and/or D&I Guidance, the comments would be annotated to indicate this fact, such as by placing the term (Guidance) or (Constraint) at the end of the clause.

5 The code that implements each clause of the response behavior is placed immediately below its authorizing comment from the behavior table.

6 Developers should understand the difference between external (required) and internal (Constraint and Guidance) behavior clauses, and implement code for each according to the following guidelines.

Behavior-specific code that implements external behavior (requirements) will appear in its entirety in the functionality module, with the main logic below the authorizing BT clause comments. This is consistent with the concept that a functionality module encapsulates the implementation of its associated requirements. Parts of the response may be delegated to internal methods of the FM consistent with good modular design of a class. Of course, common service modules are used to provide general, i.e., non-behavior-specific, services.

Behavior-specific code that implements internal behavior (Constraints and Guidance) will appear below their respective BT clause comments in the form of method calls to common services modules. This is consistent with the concept that internal details, including internal response aspects of the total responses to external stimuli, are the domain of common service modules. An example of common service module code that implements internal response behavior of a stimulus might be a method that encapsulates "business logic" that has no direct externally visible effect, such as accessing an internal data cache.

The above guideline regarding segregation of internal behavior-specific code between functionality and common service modules should be adhered to unless the Quality Requirements indicate another mapping of the internal behavior to code modules results in improved quality. In all cases, however, external behavior-specific code should be located in the functionality module itself.

Rick Lutowski