ABSTRACT

Software for real-time (time critical) control applications has been shown in military and industry studies to be a very expensive type of software effort. This type of software is not typically addressed in discussions of software architecture design methods and techniques, therefore the software engineer is usually left with a sparse design “tool kit” when confronted with overall system design involving time critical and/or control problems. This paper outlines the successful application of data flow and transaction analysis design methods to achieve a structured yet flexible software architecture for a fairly complex antenna controller used in automatic tracking antenna systems. Interesting adaptations of, and variations on, techniques described in the literature are discussed; as are issues of modularity, coupling, morphology, global data handling, and evolution (maintenance). Both positive and negative aspects of this choice of design method are outlined, and the importance of a capable real-time executive and conditional compilation and assembly is stressed.

Keywords: Transaction Analysis, Data Flow Design, Structured Software Design Methods, Real-time/control Applications

INTRODUCTION

The ACU-6 is an extremely flexible microprocessor based antenna control unit capable of intelligent control of single and dual axis pedestals, antennas and receivers in telemetry automatic tracking systems [1]. The system configuration and behavior is almost entirely software driven. The sometimes complex (and operator-friendly) or automatic behaviors of
This system are primarily a result of careful and thoughtful design that exploits the software architecture [2]. This software architecture was developed primarily using ideas and techniques of data flow and transaction analysis design as developed by Yourdan & Constantine [3]. A very useful and practical guide to these design techniques has been achieved by Page-Jones [4]. The reader is referred to these authors for detailed discussion of the principal ideas contained herein.

This paper deals primarily with the design of a software SYSTEM, and only briefly touches on some detailed design issues. The software detailed design may involve the methods discussed herein and many others such as state machines, Warnier-Orr diagrams, Nassi-Shneiderman Charts, Petri Nets, etc. No matter what techniques and coding languages are selected, the use of a Program Design Language (PDL) with a competent analyzer is strongly recommended.

SOME SOFTWARE SYSTEM DESIGN METHODS

Transaction analysis seems a natural structured design method for real-time control applications, especially those that are response oriented. Before discussing this approach, some other approaches that seem inadequate or inappropriate are briefly described. It should be understood that all these methods are useful tools and most can and are used in conjunction with transaction analysis and data flow design.

Structured Decomposition vs. Structured Design

Structured decomposition is often mentioned as a structured design method. I have found it more a goal than a method, with the Parnas Decomposition Criteria [5] as a good example. I have also noted that many writers in the current journals do not make this distinction, and I have worked directly with programmers who do not make this distinction and are somewhat confounded when asked to explain why they favor a particular decomposition, or worse yet, how they arrived at it. The analysis of any moderately sized problem or requirement, and the proposal of a candidate solution almost ALWAYS involve decomposition of some sort; our limited mental capacities usually dictate that we work with a very small number of concepts, procedures, representations or what-have-you at a time. Most individuals will readily agree that structure is good, but cannot explain the what, why, or how of achieving that structure. To decompose the problem into a structured solution is the goal - how to go about this is still somewhat the sorcery aspect of software engineering.
Data-structure Design: The Jackson Method

The Jackson Structured Programming/Design (JSP and JSD) [4], also called data-structure design, has been received favorably in the industry. Many have found it to represent an effective strategy for implementing small, distinct modules; but not for overall design. This is because data-structure design is founded on understanding and defining the data structures involved in order to eliminate structure and boundary clashes, and also because of the method’s emphasis on defining the task to be performed in terms of the elementary operations available, allocating each of those operations to suitable components of the program structure. One of the most obvious problems here is the difficulty of resolving the clashes that almost invariably arise when confronted with several, probably unrelated sets of data - a frequent occurrence in the design of even moderately complex applications. For real-time applications where data pooling is a common practice, this problem is compounded by the fact that the individual data items are usually interrelated but not necessarily structured. This will be dealt with below in the discussion of information clusters. The other problem with the method arises from the elementary operations referred to above - typically implied to be a high level language (COBOL, FORTRAN, Pascal) statements. This tends to support multiple levels of abstraction - (expressing the implementation of a large system in terms of smaller, interrelated systems) - only when the entire problem is small. If faced with an assembly language implementation, this problem is compounded considerably.

Programming Calculus

Another design method referred to occasionally in the literature is Dijkstra’s Programming Calculus. My very limited encounters with this method left me with the distinct impression that it was exceptionally rigorous, emphasizing precision in the formulation of algorithms, and offering inherent proof of correctness of a chosen algorithm. It also seemed to me that it offered no help in the decomposition of the problem or selection of the algorithm/design being implemented. Being based on Dijkstra’s well known predicate transformers, it was also difficult (at least for me) to grasp, and seemed that application to anything but very small problems would be very difficult.

Object Oriented Systems

Object oriented programming is one of the newest concepts in the field, and is one of the basic foundations of ADA. I am not personally experienced with any object oriented system; but my casual reading leaves me with the impression that this is a programming system - a virtual machine construct - and, as such, is not truly a design method but an implementation vehicle.
**State Machine Design**

This approach is a very useful design technique, but is typically difficult to use when dealing with a large system design because of the unmanageably large single system states that result. This technique is very effective when dealing with small subsystems that are hardware interactive, or designed to simulate hardware logic.

Petri Nets [5] are a form of state machine very useful in the design of concurrent processes and, since they are not based on any concept of a central system state, do not suffer the same limitations as finite state machines.

**MEASURES OF QUALITY IN A SOFTWARE SYSTEM DESIGN**

Probably the most useful concepts developed by Yourdan/Constantine [3] are the ideas of coupling and cohesion. These are measures of the independence of modules and therefore a form of assessment as to the individual degrees of modularity. This paper cannot go into deep discussions of these ideas, but a very brief description is required.

Coupling is simply the degree to which modules are interconnected; it represents the strength of these interconnections in an attempt to answer the question: How much must we know about module B in order to understand module A?

Cohesion is a term used in sociology and engineering to represent very similar concepts. Cohesion in software modules has the same connotation, and is an assessment of the characteristics of the relational associations of processing elements within a module. It represents the strength of these associations in an attempt to answer the question: Why are elements X, Y, and Z in module A?

Various levels and types of coupling and cohesion are defined and used to measure the quality of a design. These levels are referred to in Table 1 and Table 2. Also shown are assessments of the ACU-6 characteristics. The reader should know that these assessments were made by the principle system designer (the author), and may therefore feel that some bias is present. It probably is, but is also probably a critical bias; it seems that the creator is always the first to become disenchanted with his creation.
Data Flow Design

Data flow design views problems from a perspective that is not usually taken in control applications - that of data flow through a series of transformations. This can be an exceptionally useful perspective, especially for achieving higher levels of cohesion than typically result from thinking in terms of procedures, flow-charts, state machines, or other forms of control flow (flow chart thinking typically yields cohesion no higher than procedural).

Data flow also seems to describe very naturally some of the central control problems in this application. As an example, consider the data flow diagram for the position loop of the azimuth pedestal axis. Figure 1 shows the normal position mode flow graph from raw axis encoder data to position error, which is the input stream for a transform bubble called “Do Position Loop” which is detailed in Figure 3. In Figure 2 we see the autotrack data flow from raw demodulator data (from an A to D converter) to the development of commanded angle and position error, which again feeds the data flow graph of Figure 3. Note that the development of present position is identical in both flow graphs and represents a common transform, as does the transform bubble detailed in Figure 3. Selection of the data flow used is a control decision based on servo mode (see below). Figure 3 shows the data flow once the position error is known, no matter what the source.

As can be seen (Fig. 2), in track mode the commanded position is developed from tracking error and present position. In one of the normal position modes (Fig. 1) position error is developed from commanded position and present position, with commanded position sourced from another process.

The design features that result from effective factoring of the data flow are typically very useful. In this example, the development of position error from an original source of a demodulator voltage allows the divorce of the demodulator scaling, offset, and crosstalk factors from the servo loop transfer function and its coefficients, and also allows display and reporting of the tracking error in degrees.

In addition to this, since in track mode commanded angle is actually developed from the track (position) error and the present position, this angle may be reported as the target position - dynamic pedestal lag is automatically accounted for (within practical limits of demodulator linearity, dynamic range, instrumentation precisions, etc.).
The consideration of the position loop as a separate entity is not a contrivance to create an example. The position loop task is a fairly independent process, albeit environmentally coupled via the machine state information cluster (more on this shortly). The important consideration here is that this system (like most interactive, real-time automatic control systems) is homologous, i.e., it is by design somewhat non-hierarchical. Functionally distinct modules are assumed to be running simultaneously, i.e., concurrently. This is achieved by an executive discussed later.

**Transaction Analysis**

Transaction analysis, as a design method, is a subset or supplementary strategy of data flow design. It usually results from data flow graphs resembling Figure 4 where an input stream is split into many output streams. The method simply recognizes that data flows of this type may be mapped into a particular modular structure. This structure is known as a transaction center.

A transaction may mean many things in software engineering, and especially tends to imply something that is processed in a business-oriented, data processing application. In the most general sense:

> A transaction is any element of data, control, signal, event or change of state that causes, triggers, or initiates some action or sequence of actions. [3]

According to this definition, a large number of real-time control situations may be analyzed in terms of transactions. This is in fact how the ACU-6 control problems were stated and analyzed.

In this system, a primary transaction source is the front panel. Transactions are also sourced from any remote control link, and from concurrently running processes. One of the techniques used in this system that I have not seen developed anywhere else is the use of transaction feedback. This occurs when a subordinate routine, perhaps many levels deep in the transaction processing chain, sends a transaction code back through the transaction source channel to effect some new or additional action or sequence. This mechanism obviates response flags having to be sent back up many levels of hierarchy (a form of coupling known as tramp data [4]). It may at first seem to be a software shaman’s trick to hide a control coupling, but since any processing that takes place as a result of the request is completely determined by the transaction dispatching logic in charge of the requester, it is more akin to recursion or incremental execution.

This technique is one of the variations on the basic transaction analysis that copes with what is probably the fundamental defect in the design method:
... state dependent decision procedures run counter to the basic requirement that each transaction-level subsystem independently complete the processing of a given transaction so that the transaction processor (dispatch) remains simple. [3]

This defect in the method has been dealt with in the ACU-6 by using a hierarchy of transaction centers to which transactions are routed in a mode dependent manner as shown in Figure 7. The method has drawbacks, most of which are manifested in the form of complexity. Some of the complexity may only be a reflection of the application. The desired behavior of the ACU-6 is at times surprisingly complex.

**Information Clusters**

Many real-time systems utilize what can easily become a fatal design flaw: Some form or another of global data pool. In fact, the global data pool is technically a pathological data coupling, and most pathological couplings of any kind are just as bad as the name implies. The reason for using a pool is simple. When concurrent processes that must interact with each other are assumed, the global data pool represents a static, easily understood (?) collection of control and value items. While parameter passing is the first choice in terms of minimal coupling, it is sometimes hard to imagine how one relatively independent process running at an undetermined time would most effectively communicate with another. We again turn to the example of the software position loop task. In the ACU-6 this task runs every 20 milliseconds. The process that is sourcing the commanded angle is unknown to the position loop, yet it must have a valid commanded angle every time it runs. While this could be easily implemented with a mailbox, it is only one of the dozen or so data items required by the position loop. Other system and hardware design considerations may also come into play.

The ACU-6 was intended from conception to provide unattended, remote operation. This required the machine state have two additional properties:

1. It should be a contiguous block of memory allowing it to exist in, or be easily copied to, battery-backed RAM, and containing all necessary data items to allow the system to tolerate momentary power failures or fades.

2. It should somehow effect automatic change reports to a front panel report process so that the front panel display is always a true representation of the internal state. Mutual exclusion was required because of concurrent processing. On top of all of this, I had decided that in the remote control network master version of the ACU-6, it would be very convenient to distribute processing, and had in mind machine state accesses indexed
according to the currently selected slave. The structure that allows all this is an information cluster.

An information cluster is a pool of data that is globally accessible, but only via access mechanisms. The data itself can be accessed only by the access mechanisms. Processes using the mechanisms to alter or retrieve data are aware only of the interface requirements - the actual data representation is completely hidden. A binary entity may be a software flag, or it may be a bit packed in a pedestal control byte. Although I typically avoid the practice, actual data processing may also be imbedded. This has proven useful in one or two instances. Indexed quantities, both explicit (i.e. the user supplies the index) and implicit, are used. For example, a process may request selected signal strength and get the signal strength value for the currently selected receiver without having any knowledge of the selected receiver, the number of receivers, or the selection method.

**THE ACU-6 SOFTWARE ARCHITECTURE**

Most of the following discussion focuses on the standalone system. Remote control slave versions are very similar to standalone systems. On the input side, remote control interactions are kept to a minimum by sourcing transactions into the main system, and, in a few cases, by directly altering the machine state. Since most remote control reports are generated automatically by changes in the machine state, output interactions are also minimal.

**Real-Time Executive Capabilities**

The system is built around a real-time, multitasking executive, and many of the design/structure characteristics are dependent on the capabilities of the executive. While this may seem a confluence of two temporally distinct design stages, structure design and packaging, the benefits of assuming concurrent processes and considering them as functionally distinct are considerable. The real-time exec has the property of creating a virtual machine, with instructions for task control and prioritization, time delays, intertask communication, mutual exclusion, declaration of internal and external events, etc. In the ACU-6, this virtual machine capability is used to achieve a homologous morphology instead of more interdependent incremental modules such as coroutines. This is more a matter of preference than anything else. Even if incremental modules were used to decouple some modules from the information clusters, it would be difficult to implement the features and intelligent behavior of the system without a good executive.
Logical vs. Physical Entities

A very useful distinction, again arising from factored data flow graphs, is that of logical to physical mappings. Keypresses (both on and off) are mapped to transaction codes and appropriately tagged before being shipped off to the transaction processor.

On the output end, several logical to physical mappings are used for various hardware interfaces. One of the most useful is that of the logical indicator, an entity that remains constant regardless of the actual display hardware used. A single logical indicator may be mapped to any number (including zero), of incandescent lamps on any panel, or to any number of LED indicators. This, along with the ability to command any combination of three flash rates for all indicators, makes for a very flexible front panel.

Major System Processing Flow

For the most part, the reader will want to refer to Figure 6 through Figure 8 during this discussion. As mentioned above, several transaction sources can be active at any time. The primary distinction in the remote control transaction sourcing is the possibility of data items also being transferred. Two remote control versions exist, a universal slave that communicates using an ASCII command language, and a network slave that uses a protocol that rather tightly couples it with the master. This is purely an efficiency issue, but of some importance in minimizing communications overhead in networks with many (up to 6) slaves linked with only voice grade lines.

The input channel to the transaction centers is shown as a deque (double-ended queue) [7]. Specifically, this is an output-restricted deque that allows data to enter from either end, but allows emptying from only one end. This capability is used only by internal transaction sources (e.g., a control task) that requires the transaction center to take immediate action, prior to processing requests from other sources.

The channels may be seen to temporally decouple the transaction sourcing input processes from the main transaction processing, and the machine state access mechanisms from the output reporting processes. This temporal decoupling is not a design figure-of-merit, but simply a way to isolate the various subsystems, allowing them to run at different priorities, and, in the case of the output channel, to minimize the machine state access time by deferring the reporting of changes.

The report channels are referred to in the figures as execution channels. This is simply to highlight a design technique that smacks of incremental execution. Data passed to the report tasks are simply pointers to (addresses of) report functions to be executed. The receiving task really performs no imbedded processing of its own.
Mode Dependent Transaction Processing

As mentioned above, transactions are processed in a mode (state) dependent manner by a hierarchy of transaction centers (Figure 7). These transaction centers are also called mode processors. The mode hierarchy of the ACU-6 from highest to lowest is: System Mode, Auxiliary Mode, Axis Control Mode and Servo Mode. Azimuth and Elevation axis and servo modes are independent. The modeling of the system mode hierarchy has proven to be a critical element in the system design.

The system mode may take values of operate, test, or program. These three cases are quite different, and three top level transaction processors are used, one for each mode.

The auxiliary mode is typically used when control of both axes must be seized, or when front panel control normally affecting axis control must be used for some other purpose. The values of the auxiliary mode that represent the first case are immediate designate, pre-programmed designate, joystick, plunge, and keyhole. The only value (so far) that represents the second case is threshold adjust. Additionally, the mode value of null results in transfer to axis mode control.

Some explanations will clarify what these auxiliary modes do. The designate modes allow the operator to command the pedestal to a discrete position (instead of incrementally moving it). The immediate designate mode allows him to enter the angles at the front panel. The pre-programmed designate mode allows him to select one of sixteen positions entered in the program mode. Both of these modes provide cable unwrap as required. The joystick mode is a form of dummy mode, since the joystick is hardwired to the pedestal velocity command output when its button is pushed. The reason for this mode is to allow for proper axis mode shutdown and startup, and commanded position following. The plunge and keyhole modes are used to handle overhead passes.

Threshold adjust allows the acquire threshold to be adjusted, but requires the use of switches that are also used in some axis modes.

The axis control mode is a familiar concept to most experienced with positioners and tracking systems. It may take on the values of Manual, Handwheel Position, Handwheel Rate, Search, Slave, or Autotrack.

The Search mode is implemented with one mode processor (transaction center) that is interactive with both Azimuth and Elevation in dual axis systems. This is because of the highly axis-interactive behavior of the search mode in two areas: Front panel switches for increasing/decreasing the search sectors must be shared, and the major sweep axis of the
raster performed when in a dual axis search is automatically selected to be the largest of the two sectors.

Rate and Position Memory Active conditions are declared as flags (binaries) in the machine state, but since the autotrack processor does not alter the axis mode until given the command to do so by the rate/position memory task, the effect is that of submodes.

The servo mode has already been mentioned above in the position loop design examples. It may take values of velocity, type 1, type 2M, track, or rate emory. Type 2 will probably be added by the time this is published. The astute reader will have noticed that both the axis mode and the servo mode can take values of track. This may seem redundant, but it has proven very useful for implementing the rate and position memory functions. In these autotrack submodes, the axis remains in the track mode while the servo mode is commanded to rate memory and/or position (more on mode hierarchy later).

Machine State Access/Alteration

The various items found in the machine state information cluster are far too numerous to list here. Each item is defined with a macro that declares the desired data storage, the several access mechanisms associated with that item, and any reporting activity desired.

The access mechanisms currently defined are Gets (for all), Puts (for value cells), Sets, Resets, and Toggles (for binaries), and Front Panel Reports (for all) which does nothing except force a report for that item. This is made available because the automatic reports occur only when an item is changed. The report-on-change mechanism allows periodic tasks to repeatedly make the same Set, Reset, or Put with no external effect or unnecessary reporting overhead.

Control and Monitoring Tasks

These are the tasks that represent functional parallelism. Included in this group is the position loop task, the signal strength and acquire task, the pedestal status task, the receiver tuning task, and so on. They are internal, typically periodic concurrent processes that are not normally in the data and/or control flow dealt with by the transactions centers. This is mostly dependent on the system mode, as the program and test mode processors exercise start/stop control over these as required. They are always active when in the operate mode.
Display Tasks

The dynamic display tasks run at a relatively low priority, and have the functions of displaying dynamic data such as signal strength. The separation of the display functions into different tasks is primarily driven by efficiency considerations, but also by the functional independence of output display as an operation. The efficiency considerations are simply to relieve high frequency, high priority tasks of the burden of display formatting and I/O. Since these tasks run at low frequencies (5 to 10 times per second) and request display access via monitor calls, they can be usurped easily by other tasks or processes wishing to use the displays.

NEGATIVE ASPECTS OF THE DESIGN

Pancaked Central Transaction Center

This module is large, and is pancaked; i.e., has a very high fanout. Both the size of the module and the morphology may be improved by increasing the number of types (tags) of transactions. This would allow further factoring of the center into more subordinate transaction centers based on transaction type, but this module will probably always remain large.

Major Areas of Control Interaction and Coupling

One of the major areas of control interaction in the system is found in the test mode. This mode must control the antenna, pedestal axes, and front panel in ways that are very unlike the operate mode control. Since the tasks performed by the test mode may require overriding normal system processes and/or exerting more direct control over lower levels in the hierarchy (even direct control of hardware), this will probably remain a weak area in the design. This may be improved in certain areas by additional factoring of subroutines.

In the operate mode, most control interactions involve the signal strength and acquire task. This task may directly or indirectly cause antenna selection, side lobe lock comparison and/or prevention, and transitions to autotrack in either axis. While this interaction has caused no maintenance or other problems, ways to reduce the coupling between this and other modules are being investigated.

Major Areas of Low Cohesion

Most of the low cohesion found in the design results from initialization modules (temporal cohesion). Some of this lack of low cohesion is dictated by hardware considerations; e.g., start-up is done in two phases - device programming and diagnostics are performed before
the real time clock is programmed and enabled to interrupt the system. The second phase of startup is primarily dictated by the requirement to start concurrent processes in a controlled order, and to insure that all front panel reports are generated regardless of the contents of the machine state. There seems little that can be done about this.

**Binding Time**

In addition to the primary measures of goodness of design - coupling and cohesion - a second-order measure of design goodness is the binding time when most or all of the system is configured to the application and/or environment. The ACU-6 is intentionally designed for low binding time characteristics to eliminate dead code, improve code efficiency and minimize memory consumption while supporting a very wide variety of versions and customizations.

The current design requires reassembly of 70% to 80% of the system for each application. This assembly is directed by a series of definition files. The software configurer is required to create only five modules for a standalone system that requires no new/custom features: The master definition file which declares all system level configuration controls; the keyboard mapping table; the indicator mapping table; and the linker command file, which lists the files that are to be linked. For a universal slave, one additional definition file must be created, and three additional modules compiled and assembled. Note that creation of the configuration file(s) is most often the result of the alteration of an existing file from a similar application and, given a competent editor, usually goes very quickly. This process of configuration, compilation, assembly, linking and printing of complete listings typically takes less than 10 hours, of which 4 to 5 hours are print time (assuming a 200 cps printer).

**System Testing**

This system is not easy to test for two reasons:

1. The conditional compilation and assembly mean that new capabilities added may not be tested in all configurations. A processing element may be successfully tested on several systems then fail when used with certain other options conditioned in.

2. The complexity of the application and the sheer amount of code are intimidating. This is compounded by the requirement to test and debug concurrent processes and their interactions - never an easy job.
System Maintenance

This system is not easy for most programmers to understand because of three factors:

1. Most programmers are not familiar with transaction analysis based designs.

2. Homologous, multitasking systems require the understanding of several distinct concurrent processes in order to comprehend the system at large. It is not easily described in a top-down manner, even though it may have been designed using top down techniques.

3. The application itself is complex. If the software expert is not also a subject matter expert, i.e., understands automatic tracking system problems, he must learn both the problem and the solution.

Needless to say, attempts at maintenance without adequate comprehension of the application result in side effects, kludges, and lots of frustration. Those programmers that have overcome the above factors find the system relatively easy to modify.

POSITIVE ASPECTS OF THE DESIGN

The positive aspects of the design are easily stated: It is structured, easily understood at the conceptual level (once the ideas of data flow and transaction analysis are assimilated), flexible, intelligent in it’s behavior and operator interactions, can implement a powerful remote control capability (with minimum overall system impact) and in general accomplishes a remarkable amount of work with the rather limited resources of Z80A processor. The ACU-6 has been characterized as a chameleon because of the very large number of options available. In addition, almost every one of the dozen or so systems delivered to date has incorporated new features or customer-requested modifications. The software architecture has generally accommodated these design extensions and customizations in a straightforward and relatively painless manner.

The cost effectiveness of the design may be demonstrated by considering the following. The TRW Wolverton software cost study surveyed relatively large scale, near real-time command and control software projects implemented in the late 1950’s and early 1970’s. The resulting cost model was stated in terms of cost per object instruction (oi) as a function of relative degree of difficulty, novelty of application and type of project [8]. The costs range from $15/oi for easy, old algorithms to $85/oi for time critical code of any kind. Most of the ACU-6 software was new, control oriented of probably 60% difficulty, to which the study would assign around $42/oi. Since much of the ACU-6 software is also time critical and 80%-100% implemented in assembly language, one might more reasonably expect an average cost of $60/oi or even $70/oi. If a 100% increase in software
productivity has occurred in the last ten years, and another 200% reduction in cost is allowed for the difference in project sizes (this is not likely), the expected cost of the ACU-6 object instructions could be as low as $10. The current, worst-case estimate of ACU-6 software costs yields approximately $5 per object instruction.

CONCLUSION

The design methods discussed have proven to be valuable tools in the design and implementation of a real-time, multitasking control application. These methods provide the system designer with meaningful measures of quality of design and effective approaches to modeling and decomposing complex processing problems.

ACKNOWLEDGEMENTS

Many thanks to Brian Cooley and EMP, Inc. for producing the illustrations under a very tight schedule. Also thanks to C. K. Boehr, R. R. Curry and W. M. Ames for their editorial comments and assistance.

NOTES AND REFERENCES


Table 1  
Coupling Types

<table>
<thead>
<tr>
<th>Coupling type</th>
<th>Goodness</th>
<th>Degree found in ACU-6</th>
</tr>
</thead>
<tbody>
<tr>
<td>input-output</td>
<td>highest (10)</td>
<td>moderate to low</td>
</tr>
<tr>
<td>data structure</td>
<td>high (9)</td>
<td>high</td>
</tr>
<tr>
<td>environment</td>
<td>moderate (7)</td>
<td>high</td>
</tr>
<tr>
<td>content</td>
<td>moderate (5)</td>
<td>moderate</td>
</tr>
<tr>
<td>control (activation)</td>
<td>moderate (4)</td>
<td>moderate to low</td>
</tr>
<tr>
<td></td>
<td>low (2)</td>
<td>low to none</td>
</tr>
<tr>
<td>hybrid</td>
<td>lowest (0)</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 2  
Cohesion Types

<table>
<thead>
<tr>
<th>Cohesion</th>
<th>Goodness</th>
<th>Degree found in ACU-6</th>
</tr>
</thead>
<tbody>
<tr>
<td>functional</td>
<td>highest (10)</td>
<td>low</td>
</tr>
<tr>
<td>sequential</td>
<td>high (9)</td>
<td>moderate to high</td>
</tr>
<tr>
<td>communicational</td>
<td>moderate (7)</td>
<td>high</td>
</tr>
<tr>
<td>procedural</td>
<td>moderate (5)</td>
<td>low</td>
</tr>
<tr>
<td>temporal</td>
<td>low (3)</td>
<td>low</td>
</tr>
<tr>
<td>logical</td>
<td>low (1)</td>
<td>none</td>
</tr>
<tr>
<td>coincidental</td>
<td>lowest (0)</td>
<td>none</td>
</tr>
</tbody>
</table>
### Table 3
**Binding Times**

<table>
<thead>
<tr>
<th>Binding Time</th>
<th>Goodness</th>
<th>Degree found in ACU-6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>highest</td>
<td>none</td>
</tr>
<tr>
<td>Run time</td>
<td>high</td>
<td>low to none</td>
</tr>
<tr>
<td>Linking/loading</td>
<td>moderate</td>
<td>moderate to low</td>
</tr>
<tr>
<td>Compilation/Assembly</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>Coding</td>
<td>lowest</td>
<td>low to none</td>
</tr>
</tbody>
</table>

#### FIGURE 1.
POSITION LOOP DATA FLOW GRAPH
POSITION MODE
FIGURE 2.
POSITION LOOP DATA FLOW GRAPH
FIGURE 3.
POSITION LOOP DATA FLOW GRAPH "DO POSITION LOOP" NODE

FIGURE 4.
AN EXAMPLE OF A TRANSACTION CENTER IN A DATA FLOW GRAPH
FIGURE 5.
ACU-6 STANDALONE PRINCIPAL DATA FLOW
FIGURE 6.
ACU-6 SOFTWARE SYSTEM, STANDALONE
MAJOR COMPONENTS AND ORGANIZATION
FIGURE 7.
MAIN TRANSACTION PROCESSING
HIERARCHY (PARTIAL)
FIGURE 8.
ACU-6 UNIVERSAL (ASCII COMMAND LINE) SLAVE
PRINCIPAL DATA FLOW

FIGURE 9.
ACU-6 MASTER
PRINCIPAL DATA FLOW