module differs from the other modules in that both the secrets and the interfaces are determined by software designers. Changes in these modules are more likely to be motivated by a desire to improve performance or accuracy than by externally imposed changes.

The module guide goes on to explain how conflicts among these categories (e.g., is a required algorithm part of the behavior or a software decision?) are arbitrated by a complete and unambiguous requirements specification and then provides the second-level decomposition. The following sections describe how the Software Decision Module is decomposed.

Application Data Type Module—The Application Data Type Module supplements the data types provided by the Extended Computer Module with data types that are useful for avionics applications and do not require a computer-dependent implementation. Examples of types include distance (useful for altitude), time intervals, and angles (useful for latitude and longitude). These data types are implemented using the basic numeric data types provided by the Extended Computer; variables of those types are used just as if the types were built into the Extended Computer.

The secrets of the Application Data Type Module are the data representation used in the variables and the procedures used to implement operations on those variables. Units of measurement (such as feet, seconds, or radians) are part of the representation and are hidden. Where necessary, the modules provide conversion operators that deliver or accept real values in specified units.

Data Banker Module—Most data are produced by one module and consumed by another. In most cases, the consumers should receive a value that is as up to date as practical. The time at which a datum should be recalculated is determined both by properties of its consumer (e.g., accuracy requirements) and by properties of its producer (e.g., cost of calculation, rate of change of value). The Data Banker Module acts as a "middleman" and determines when new values for these data are computed.

The Data Banker Module obtains values from producer procedures; consumer procedures obtain data from Data Banker access procedures. The producer and consumers of a particular datum can be written without knowing when a stored value is updated. In most cases, neither the producer nor the consumer need be modified if the updating policy changes.

The Data Banker provides values for all data that report on the internal state of a module or on the state of the aircraft. The Data Banker also signals events involving changes in the values that it supplies. The Data Banker is used as long as consumer and producer are separate modules, even when they are both submodules of a larger module. The Data Banker is not used if consumers require specific members of the sequence of values computed by the producer or if a produced value is solely a function of the values of input parameters given to the producing procedure, such as $\sin(x)$.[1]

[1] The Data Banker Module is an example of the use of the blackboard architectural pattern (see Chapter 5, Achieving Qualities).

The choice among updating policies should be based on the consumers' accuracy requirements, how often consumers require the value, the maximum wait that consumers can accept, how rapidly the value changes, and the cost of producing a new value. This information is part of the specification given to the implementor of the Data Banker Module.

Filter Behavior Module—The Filter Behavior Module contains digital models of physical filters. They can be used by other procedures to filter potentially noisy data. The primary secrets of this module are the models used for the estimation of values based on sample values and error estimates. The secondary secrets are the computer algorithms and data structures used to implement those models.

Physical Models Module—The software requires estimates of quantities that cannot be measured directly but can be computed from observables using mathematical models. An example is the time that a ballistic weapon will take to strike the ground. The primary secrets of the Physical Models Module are the models; the secondary secrets are the computer implementations of those models.

Software Utility Module—The Software Utility Module contains those utility routines that would otherwise have to be written by more than one other programmer. The routines include mathematical functions, resource monitors, and procedures that signal when all modules have completed their power-up initialization. The secrets of the module are the data structures and algorithms used to implement the procedures.

System Generation Module—The primary secrets of the System Generation Module are decisions that are postponed until system generation time. These include the values of system-generation parameters and the choice among alternative implementations of a module. The secondary secrets of the System Generation Module are the method used to generate a machine-executable form of the code and the representation of the postponed decisions. The procedures in this module do not run on the onboard computer; they run on the computer used to generate the code for the onboard system.

The module guide describes a third- (and in some cases a fourth-) level decomposition, but that has been omitted here. Figure 3.4 shows the decomposition structure of the A-7E architecture down to the third level. Notice that many of the Device Interface modules have the same names as Function Driver modules. The difference is that the Device Interface modules are programmed with knowledge of how the software interfaces with the devices; the Function Driver modules are programmed with the knowledge of values required to be computed and sent to those devices. This suggests another architectural relationship that we will explore shortly: how the software in these modules cooperates to accomplish work.

But the module decomposition view is not yet complete. Recall from Chapter 2 our definition of architecture as including the behavioral specification for each of the elements. Carefully designed language-independent interfaces are crucial for maintaining portability and achieving interoperability. Here, each module must have an interface specified for it. Chapter 9 discusses documentation for software interfaces.

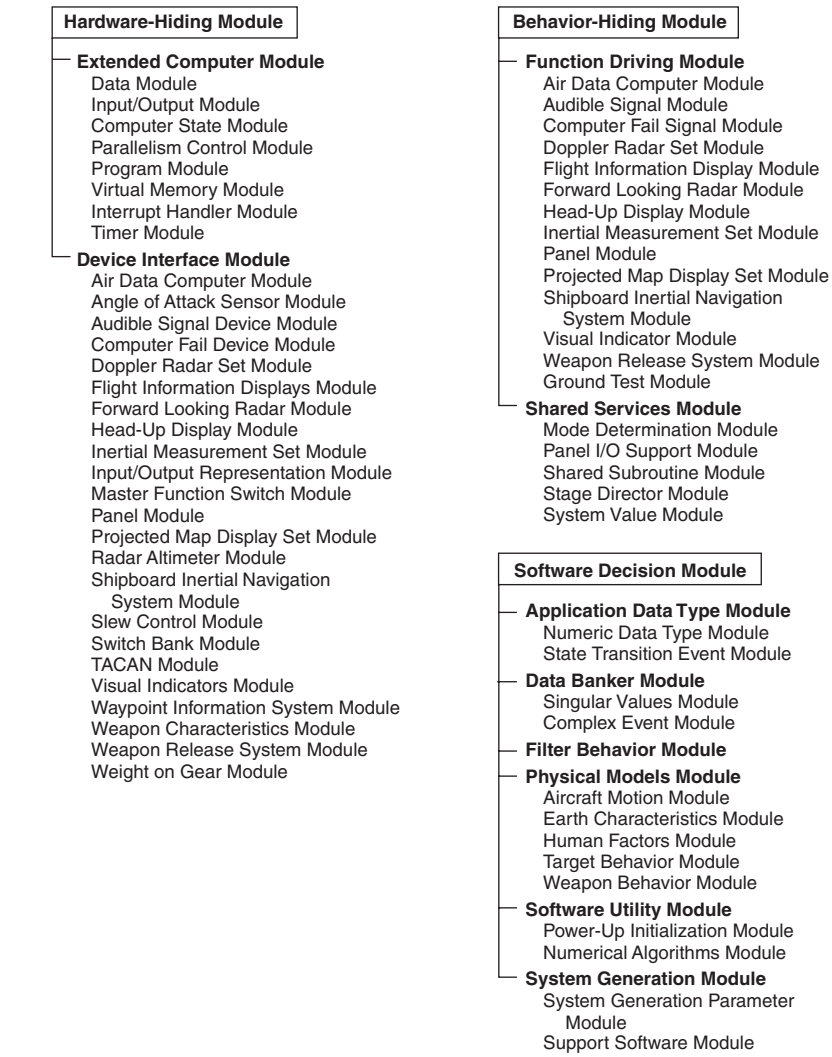

FIGURE 3.4 The module decomposition view of the A-7E software architecture

In the previous chapter, we remarked that architectures serve as the blueprint for the developing project as well as for the software. In the case of the A-7E architecture, this second-level module decomposition structure became enshrined in many ways: Design documentation, online configuration-controlled files, test plans, programming teams, review procedures, and project schedule and milestones all used it as their unit of reference.

USES STRUCTURE

The second major structure of interest in the A-7E architecture is the uses structure. The decomposition structure carries no information about runtime execution of the software; you might make an educated guess as to how two procedures in different modules interact at runtime, but this information is not in fact in the module decomposition. Rather, the uses structure supplies the authoritative picture of how the software interacts.

The Uses Relation. The concept behind the uses structure is the uses relation. Procedure A is said to *use* procedure B if a correctly functioning procedure B must be present in order for procedure A to meet its requirements. In practice this relation is similar to but not quite the same as the calls relation. Procedure A usually calls procedure B because it uses it. However, here are two cases where uses and calls are different:

1. Procedure A is simply required to call procedure B in its specification, but the future computation performed by A will not depend on what B does. Procedure B must be present in order for procedure A to work, but it need not be correct. A calls, but does not use, B. B might be an error handler, for example.
2. Procedure B performs its function without being called by procedure A, but A uses the results. The results might be an updated data store that B leaves behind. Or B might be an interrupt handler that A assumes exists and functions correctly. A uses, but does not call, B.

The uses relation allows rapid identification of functional subsets. If you know that procedure A needs to be in the subset, you also know that every procedure that A uses must also be there. The transitive closure of this relation defines the subset. It therefore pays to engineer this structure, to impose a discipline on it, so that every subset needn't consist of the entire system. This means specifying an allowed-to-use structure for programmers. After implementation is complete, the actual uses can be cataloged.

The unit of the uses (or allowed-to-use) structure is the access procedure. By dictating what procedures are allowed to use which other procedures (and, by implication, what procedures are *not* allowed to be used by which other procedures), the uses structure is defined.

Although the unit of the uses structure is a procedure, in practice all of the procedures of a module may share usage restrictions. Hence, the name of a module might appear in the uses structure; if so, it is shorthand for all of the access procedures in that module.

The uses (allowed-to-use) structure is conceptually documented with a binary matrix; each row and column lists every procedure in the system. Thus, if element (*m,n*) is true, then procedure *m* uses (is allowed to use) procedure *n*. In practice, this is too cumbersome, and a shorthand was introduced in which rules for whole modules (as opposed to individual procedures within each module) were adopted.

Table 3.3 summarizes the role of the uses structure in the A-7E software architecture.

The A-7E Uses Structure. Recall that the uses structure is first documented in a specification showing the allowed-to-use relation; actual uses are extracted after implementation. The allowed-to-use specification for the A-7E architecture is a seven-page table of which Table 3.4 is a short excerpt. The two-character preface refers to the second-level modules. The names to the right of the period refer to submodule names that we have mostly omitted from this chapter.

TABLE 3.3 How the A-7E Uses Structure Achieves Quality Goals

Goal	How Achieved
Incrementally build and test system functions	Create "is-allowed-to-use" structure for programmers that limits procedures each can use
Design for platform change	Restrict number of procedures that use platform directly
Produce usage guidance of manageable size	Where appropriate, define uses to be a relationship among modules

TABLE 3.4 Excerpt from the A-7E Allowed-to-Use Specification

Using procedures: A procedure in . . .	. . . is allowed to use any procedure in . . .
EC: Extended Computer Module	None
DI: Device Interface Module	EC.DATA, EC.PGM, EC.IO, EC.PAR, AT.NUM, AT.STE, SU
ADC: Air Data Computer	PM.ECM
IMS: Inertial Measurement Set	PM.ACM
FD: Function Driver Module	EC.DATA, EC.PAR, EC.PGM, AT.NUM, AT.STE, SU, DB.SS.MODE, DB.SS.PNL.INPUT, DB.SS.SYSVAL, DB.DI
ADC: Air Data Computer Functions	DB.DI.ADC, DI.ADC, FB
IMS: IMS Functions	DB.DI.IMS, DI.IMS
PNL: Panel Functions	EC.IO, DB.SS.PNL.CONFIG, SS.PNL. FORMAT, DI.ADC, DI.IMS, DI.PMDS, DI.PNL
SS: Shared Services Module	EC.DATA, EC.PGM, EC.PAR, AT.NUM, AT.STE, SU
PNL: Panel I/O Support	DB.SS.MODE, DB.DI.PNL, DB.DI.SWB, SS.PNL.CONFIG, DI.PNL
AT: Application Data Type Module	EC.DATA, EC.PGM
NUM: Numeric Data Types	None additional
STE: State Transition Events	EC.PAR

Notice the pattern that emerges:

- No procedure in the Extended Computer Module is allowed to use a procedure in any other module, but all other modules are allowed to use (portions of) it.
- Procedures in the Application Data Type Module are allowed to use only procedures in the Extended Computer Module and nothing else.
- Procedures in the Device Interface Module (at least the part shown) are allowed to use only Extended Computer, Application Data Type, and Physical Models procedures.
- Function Driver and Shared Services procedures can use Data Banker, Extended Computer, Application Data Type, and Device Interface procedures.
- No procedure can use any procedure in the Function Driver Module.
- Only a Function Driver procedure can use a Shared Services procedure.

What we have is a picture of a system partitioned into *layers*. The Extended Computer Module is the bottommost layer, and the Application Data Type Module is built right on top of it. The two form a virtual machine in which a procedure at a particular level is allowed to use a procedure at the same or any lower level.

At the high end of the layering come the Function Driver and Shared Services modules, which have the freedom to use a wide variety of system facilities to do their jobs. In the middle layers lie the Physical Models, Filter Behavior, and Data Banker modules. The Software Utilities reside in parallel with this structure and are allowed to use anything (except the Function Drivers) necessary to accomplish their individual tasks.

Layered architectures are a well-known architectural pattern and occur in many of the case studies in this book. Layering emerges from the uses structure, but is not a substitute for it as layering does not show what subsets are possible. This is the point of the uses structure—a *particular* Function Driver Module will use a *particular* set of Shared Services, Data Banker, Physical Models, Device Interface, Application Data Type, and Extended Computer operations. The used Shared Services in turn use their own set of lower-level procedures, and so forth. The complete set of procedures derived in this manner constitutes a subset.

The allowed-to-use structure also provides an image of how the procedures of modules interact at runtime to accomplish tasks. Each Function Driver procedure controls the output value associated with one output device, such as the position of a displayed symbol. In general, a Function Driver procedure retrieves data (via Data Banker procedures) from data producers, applies rules for computing the correct value of its assigned output, and sends that value to the device by calling the appropriate Device Interface procedure. Data may come from one of the following:

- Device Interface procedures about the state of the world with which the software interfaces
- Physical Models procedures that compute predictive measures about the outside world (such as where a bomb will strike the earth if released now, given the aircraft's current position and velocity)

- Shared Services procedures about the current mode, the trustworthiness of current sensor readings, or what panel operations the pilot has requested

Once the allowed-to-use structure is designed, implementors know what interfaces they need to be familiar with in order to do their work. After implementation is complete, the actual uses structure can be documented so that subsets can be fielded. The ability to deploy a subset of a system is an important part of the Evolutionary Delivery Life Cycle (see Chapter 7, Designing the Architecture). When budgets are cut (or overrun) and schedules slip, delivering a subset is often the best way to put a positive face on a bad situation. It is probably the case that more subsets would be delivered (instead of nothing at all) if the architectural structure necessary to achieve them—the uses structure—had been carefully designed.

PROCESS STRUCTURE

The third structure of architectural importance to the A-7E is the process structure. Even though the underlying aircraft computer is a uniprocessor, the Extended Computer Module presents a virtual programming interface that features multiprocessing capabilities. This was to plan for if and when the A-7E computer was replaced with an actual multi-processor. Hence, the software was implemented as a set of cooperating sequential processes that synchronize with each other to cooperatively use shared resources. The set was arranged using offline (pre-runtime) scheduling to produce a single executable thread that is then loaded onto the host computer.

A process is a set of programming steps that are repeated in response to a triggering event or to a timing constraint. It has its own thread of control, and it can suspend itself by waiting for an event (usually by invoking one of the event-signaling programs on a module's interface).

Processes are written for two purposes in the A-7E. The first is for the function drivers to compute the output values of the avionics software. They are required to run periodically (e.g., to continuously update a symbol position on the heads-up display) or in response to some triggering event (e.g., when the pilot presses the weapon release button). It is natural to implement these as processes. Conceptually, function driver processes are structured as follows:

- Periodic process: do every 40 milliseconds
  - Call other modules' access procedures to gather the values of all relevant inputs
  - Calculate the resulting output value
  - Call the appropriate Device Interface procedure to send the output value to the outside world
- End periodic process
- Demand process
  - Await triggering event

  - Calculate the resulting output outcome
  - Call the appropriate Device Interface procedure to trigger the action in the outside world
- End demand process

Processes also occur, although less frequently, as a way to implement certain access procedures. If the value returned by an access procedure is expensive to compute, a programmer might meet the timing requirements by continuously computing the value in the background and simply returning the most recent value immediately when the access procedure is called. For example,

- Process: do every 100 milliseconds
  - Gather inputs to compute value
  - Compute value
  - Store in variable most_recent
- End process
- Procedure get_value(p1)
  - p1 := most_recent.
  - return
- End procedure

The process structure, then, consists of the set of processes in the software. The relation it contains is "synchronizes-with," which is based on events that one process signals and one or more processes await. This relation is used as the primary input to the scheduling activity, which includes deadlock avoidance.

The offline scheduling techniques used in the A-7E software are beyond the scope of this treatment, but they avoid the overhead of a runtime scheduler, and they would not have been possible without the information contained in the process structure. The process structure also allows an optimization trick: merging two otherwise unrelated processes, which makes scheduling easier in many circumstances and avoids the overhead of context switching when one process suspends and another resumes. This technique is invisible to programmers, occurring automatically during system construction. Table 3.5 summarizes the role of the process structure in the A-7E architecture.

TABLE 3.5 How the A-7E Process Structure Achieves Quality Goals

Goal	How Achieved
Map input to output	Each process implemented as cycle that samples, inputs, computes, and presents output
Maintain real-time constraints	Identify process through process structure and then perform offline scheduling
Provide results of time-consuming calculations immediately	Perform calculations in background and return most recent value when queried

The process structure emerged after the other structures had been designed. Function Driver procedures were implemented as processes. Other processes computed time-consuming calculations in the background so that a value would always be available.

Two kinds of information were captured in the process structure. The first documented what procedures were included in the body of each process. This gave a picture of the threads that ran through the system and also told the implementors which procedures must be coded to be re-entrant (i.e., able to carry two or more threads of control simultaneously) by using protected data stores or mutual exclusion. It also gave designers early insight into which procedures were going to be invoked most often, suggesting areas where optimization would pay off.

The second kind of information in the process structure documented which processes (or sequential segments of process threads) could not execute simultaneously. The actual regions of mutual exclusion were not finalized until the processes were completely coded, but the early "excludes" relation among processes let the scheduling team understand some of the quantitative requirements of the offline scheduler and start planning on areas where automation would be most helpful.

Success or Failure?

Bob Glass, in his editorial in the November 1998 issue of *The Journal of Systems and Software* [Glass 98], argues that the A-7E was a failure because the software described in this chapter never flew. I have a great deal of respect for Bob, both personally and professionally, but in this case he is mistaken. He is evaluating a research system by commercial standards.

What do I mean by that? The research world and the commercial world have different cultures and different standards for success. One manifestation of this difference is how the two worlds "sell" to their customers. The commercial world prides itself on delivering, on time and on budget, what is specified. You would justifiably be upset if you went to your local automotive dealer to purchase a car and it wasn't delivered on time, at the cost you contracted for, and performing in the fashion you expected.

The research world "sells" on vision. That is, a research proposal specifies how the world will be different if the funder supports the proposed research. The funder should be upset if, at the end of the research, what is delivered is something that could be purchased at a local commercial establishment. Usually the funder is quite satisfied if the research produces new ideas that have the potential to change the world.

While these characterizations are admittedly idealized, they are by and large accurate. Commercial customers frequently want innovation. Research customers almost always want deliverables. Also, both camps must often promise deliverables that cannot be delivered as a means of securing sales. Still, the heart of this characterization is true.

The goal of the A-7E project described in this chapter was to demonstrate to a skeptical world that "object-oriented techniques" (although the terminology was different then) could be used to construct real-time high-performance software. This is a research objective. The goal was to change the world as it was seen then. From a research perspective, the success of the Software Cost Reduction program (of which the A-7E development was a portion) can be seen in the number of citations it has been given in the research literature (in the hundreds). It can also be seen in the general acceptance of much of what was revolutionary at the time in terms of encapsulation and information hiding.

So the A-7E was a commercial "failure," but it was a research success. To go back to Bob's argument, the question is Did the Navy get what they were paying for? This depends on whether the Navy thought it was paying for a production system or a research effort. Since the effort was housed in the Naval Research Laboratory, it seems clear that the A-7E was a research effort and should be judged by research standards.

— LJB

3.4 Summary

This chapter described the architecture of a highly capable avionics system in terms of three related but quite different structures. A module decomposition structure describes design-time relations among its components, which are implementation units that can be assigned to teams. A uses structure describes runtime usage relations among its components, which are procedures in modules. From it, a picture of a layered architecture emerges. The process structure describes the parallelism of the system and is the basis for assignment for the physical hardware.

It is critical to design each structure correctly because each is the key to a different quality attribute: ease of change, ease of extracting a subset, and increased parallelism and performance. It is also critical to document each structure completely because the information about each is duplicated in no other place.

Even though the structures are orthogonal, they are related, in that modules contain procedures, which use each other and are strung together in processes. Other architectural structures could have been specified for this system. One, a data flow view (a component-and-connector view additional to those introduced in Chapter 2), would have looked something like the one in Figure 3.5. All data comes from the external world via the Device Interface modules and works its way through computation and storage modules to the Function Driver modules, which compute output values to send back to the devices. The A-7E designers never thought data flow views were useful—what quality attribute do they help achieve that the others do not?—but other designers might feel different. The point—and the lesson—about architectural views is that they should enhance

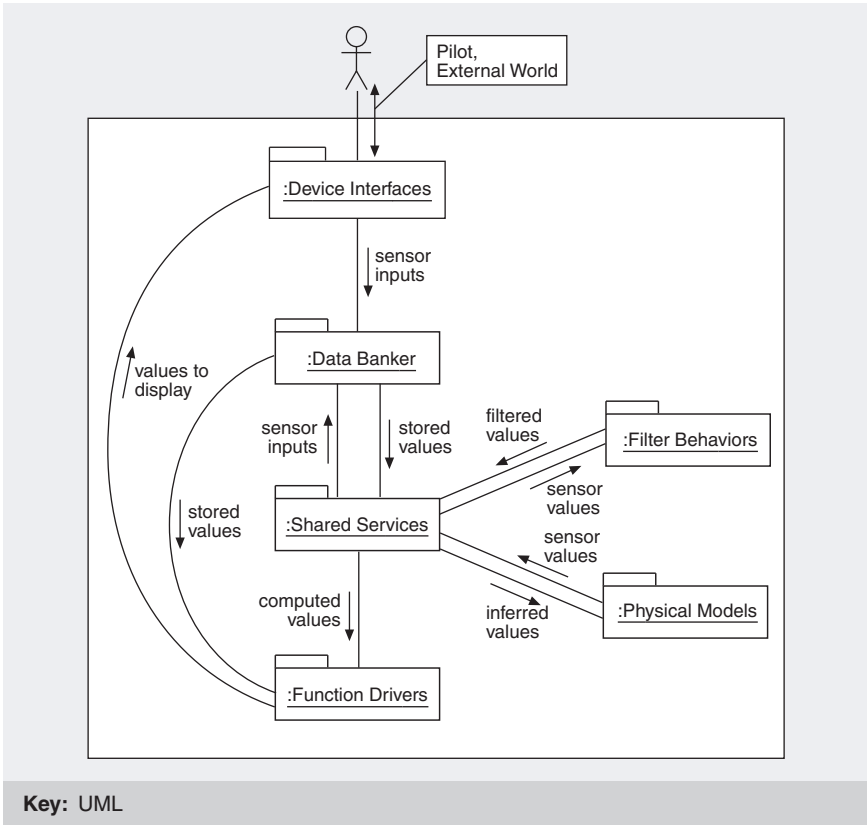

FIGURE 3.5 Coarse-grained data flow view for the A-7E software

understanding of and intellectual control over the system and its attributes. If a view meets these conditions, it is probably one you will want to pay attention to.

We also presented the architecture in terms of the qualities the designers wished to achieve: changeability and understandability. This leads us to the thesis that we explore in the next two chapters: Architectures reflect a set of desired qualities.

3.5 For Further Reading

The A7-E avionics project has been documented in [Parnas 85*a*]. The data collected about changes to the system was analyzed and described in [Hager 91] and [Hager 89]. Much of the material about the module structure was taken from the

A-7E module guide, which was written by Kathryn Britton and David Parnas [Britton 81].

3.6 Discussion Questions

1. Suppose that a version of the A-7E software were to be developed for installation on a flight trainer version of the aircraft. This aircraft would carry no weapons, but it would teach pilots how to navigate using the onboard avionics. What structures of the architecture would have to change, and how?

2. Chapter 7 will discuss using the architecture as a basis for incremental development: starting small and growing the system but having a working subset at all times. Propose the smallest subset of the A-7E software that still does something (correctly, in accordance with requirements) observable by the pilot. (A good candidate is displaying a value, such as current heading on some cockpit display.) Which modules do you need and which can you do without? Now propose three incremental additions to that subset and specify the development plan (i.e., which modules you need) for those.

3. Suppose that monitors were added to ensure that correct values were being stored in the Data Banker and computed by the Function Drivers. If the monitors detected a disparity between the stored or computed values and what they computed as the correct values, they would signal an error. Show how each of the A-7E's architectural structures would change to accommodate this design. If you add modules, state the information-hiding criteria for placing them in the module hierarchy.