

Conquering complexity through modeling

□

A structured approach to designing software involves breaking the problem at hand into algorithms, formulas, and procedures that sufficiently describe the designer's view of a system. It depends upon the designer's ability to cover the problem with a solution based on computer storage, mathematical operations, and the correct sequence and timing of calls to subroutines that each perform smaller steps. In structured programming, the designer has at his disposal only four operations, or algorithms: *concatenation* (the serial execution of lines of code in order), *iteration* (repeated execution of small steps in order to accomplish a larger task, as in a for-loop), *alternation* (a yes/no decision, as in an "if" statement), and *transfer of control* (a supervisory decision to jump to a subroutine based on conditions). Data structures (records and arrays) help in organizing the storage and presentation of data. Memory and I/O ports query the environment and deliver results to users. As problems become larger and more complex, the likelihood of offering complete coverage using these small steps becomes difficult. Designers and teams must be adept at extending the architecture of a computer to problems that are not so easily described using concatenation, iteration, alternation, and transfer of control. The term "solution space" describes this computer-centric view of target systems using the language of a computer itself: algorithms, formulas, procedures, arrays, records, memory, and I/O.

□

The Object Oriented view of computer-solvable problems came about as computers themselves moved from the arena of research and math, to applications in the real world. □ The move can be seen as one from the "solution space" to the "problem space", or to a view dictated by an attempt to model the domain of the problem. In solving a problem using object oriented techniques, the designer must know very little of the computer, but must know a lot of the environment to which the computer will be applied. It is an exercise in modeling.

□

In a structured environment, for example, a robot control program will be constructed of routines to calculate position errors into position vs. time arrays, to call derivative formulas to construct velocity vs. time arrays, and to move motors forward and backward accordingly. This is a perfectly acceptable way to control a robot arm. An object-oriented solution would have different concurrently running control tasks for each joint. The architecture of the software would itself resemble an arm: a shoulder, an elbow, a wrist, and a grasp. Each joint would have at its disposal all of the control information it needed to participate in life-like movement - desired and actual positions, and the current positions and speed of all other joint motors. The structured approach breaks down when joints must be added (a waist, for example), or when a single joint becomes more complicated than the rest (a wrist that moves laterally and that rotates). In the object oriented approach, the cohesive, decoupled nature of each task allows for the designer to address its singular contribution to the whole.

□

Those tasks are *objects*, and they exist concurrently and are simultaneously persistent. They each operate independently of the system, yet have access to the system's collective resources. They have a known, dependable interface - or contract - with the system to provide responses to stimuli in a dependable form. Their creation and destruction, activation and sleep are, at the control of the designer. They have a capability that no structured module can approach: their exact form, their behavior, the domain in which they operate, and their use of system resources can be customized at run-time. They are perfectly suited to model complex, large, real world problems and systems.

□

Build a little, fix a little

□

The world to which we apply computers is complex enough. Deriving a software solution to problems based in the complex real world is more complicated still. As software engineers, we must manage the complexity of the design process as well. Historically, we have relied mostly on the *waterfall* method: building large systems in a single large effort, throwing the switch, and hoping that all of the interactions, data boundaries, time slices, and pointers to memory work as planned. This "big bang" approach to design and implementation depends on knowing and building everything up front. Even with careful analysis and documentation, this is rarely how complex systems evolve... a design team's understanding of the problem naturally gets better through fielding and evaluation of better and better attempts. Otherwise, the concept of software versions might never be needed.

□

An alternative might be to derive a simple but representative subset of the system, and to build that in standalone form, to serve as a foundation to which the remainder of the system can be added. It reduces the complex problem to a small start. It allows a fielding and evaluation of the software as a tool for specification of system requirements without wasted or architectural-weak efforts, as is often done in rapid prototyping.

□

That subset must be stable and flexible. It must be representative of the end system, therefore it must include input, output, and calculation components (that is why it is often referred to as an "integration thread", because it is a single strand of reduced capability that extends through the entire system). It must be able to be augmented without weakening. Once stabilized, it must be added-to in a massively parallel fashion. The project team should be able to pick the add-ons and simultaneously begin working on as many of them as the schedule calls for. Over the course of a project schedule, the integration thread is always stable and working, with add-ons

providing increased functionality. Errors are easy to find because they are likely due to the piece just added. The system evolves, but always works.

□

Structured design does not lend itself to incremental, evolutionary, staged, or spiral design - simply because pieces of the architecture are not independent enough of each other or the system as a whole to be designed and built. The extent to which structured components can be implemented as stand alone, cohesive, decoupled capabilities (each contributing a dependable piece of the end system) is the extent to which they behave like objects. And yet, the environment in which they operate does not allow them the freedom to act like objects. A structured subroutine can not be constructed, destroyed, exist concurrently with other components that act simultaneously, be created in altered form depending on run-time conditions... all of which are traits of objects in an object oriented environment.

□

It would appear that an object oriented approach is tailor-made for incremental design and build: only the objects in an intelligently derived integration thread need to be built. Even the objects that are chosen need not be built completely, and yet the architecture of the whole is not compromised as the objects evolve. Teams can work on add-on objects in parallel. Objects can change, as more information becomes known about the system.□□

□

The application of OO to system longevity

□

As a system is changed, its structure is degraded.□Additional costs, over and above those of implementing the change, must be accepted if the structural degradation is to be reversed. Large systems have a dynamic of their own that is established at an early stage of the development process. This determines the gross trends of the maintenance process and limits the number of possible system changes. (i.e., as changes are made to a system, these changes introduce new system faults, self limiting the long term ability of a system to accept maintenance).□This is called the *structural* limit.

□

Organizations have a bureaucratic equivalent to this self limiting dynamic: major system changes require increasingly complex decisions based on customer expectations, marketing, scheduling, budget, and allocation of personnel, all of which have practical limits.□This is called the *justification* limit.

□

The maintainability of a program is related to its complexity. Complexity is not dependent on size but is based on the decision structure of the program. One of the reasons that maintenance costs are high (and that the structural and justification limits are reached early) is that the structure of the system, which has to be modified, is non-existent, either through lack of design, serpentine implementation, or the effects of continuing changes.

□

Object oriented design, in which system parts are isolated, decoupled, and cohesive and are therefore replaceable as units with a known interface to the rest of the system extends both the structural and justification limits indefinitely.

Incremental Development Approach

□

To increase the chance of developing a stable, working system, an incremental (staged, evolutionary) approach might be taken to build a system from a simple, working thread to a complicated, adaptive whole. In most cases, the simpler capabilities must be done first anyway, and greater capabilities added as part of a larger collection of features. For instance, for an IP telephony device, incoming voice call capability must be developed as a core before video conferencing.

□

The system can be built up from a simple working thread in a modular fashion. To that end, partitioning is everything, and module-based services should be provided so that other modules can solicit information and act on their own. Details of the inner workings of modules will be hidden, with well-defined interfaces upheld so that as progress is made, function and effect are isolated. Violating the constructs of loose coupling and strong cohesion must be negotiated for good reason. These are not constructs to be taken lightly, and probably will have more of an effect on eventual success than the entire collection of required functions. This will force simplicity and clarity over cleverness. It will also guarantee that the architecture will be stable over many iterations.

□

The software team will not be able to define every detail at once. It will define a "thread" of basic functionality on which the remainder of the system will be built. When those first subsets of functionality are operational, the remainder of the job will become clear-cut. Thread development is sometimes referred to as modified-spiral, incremental, evolutionary, or staged development. Intelligent thread development is the core of effective software engineering in this

model.

□

Sometimes, the issues of design methodology conflict with thread development, since thread development substitutes for much of the up-front specification associated with the generation of requirements and the documentation of design. Moreover, the best commercial practices of staged development sometimes appear as chaos when compared to MIL-STD. Unless the team wishes to produce and adhere to the 62 documents required of MIL-STD-2167 or 498, they need to manage the chaos by being flexible in attitude but disciplined in module design (staged development tends to work well through intelligent partitioning of the system architecture).

□

An Object-Oriented Architecture

□

Object Oriented design represents a philosophical difference from structured design. It is one of independence of function rather than supervisory control. Objects operate as small, independent computers; each with a specific job to do when the environment in which they operate presents the proper stimulus. The major design task in an object-oriented architecture is inter-object communication and the coordination of independently running "computers" (which objects can be viewed as) to collectively perform a coherent function. It more closely resembles the real world.

□

Objects must be developed in the architecture to deal with messaging, coordination, or prioritization of tasks. If the design degrades to specialty interfaces between highly specific functions, then that is a somewhat dangerous, non-extensible way to design a system. Systems like that tend to do only one thing, and not adapt well to changes. That may not bode well for products needed to evolve as the operating environment for products becomes more complicated and powerful.□

□

Starting the Project

□

The software design team needs to begin at a clean architecture and bring in functionality in a deterministic way. They should start by building a single thread of control, consisting of objects that will manage other objects: messaging, prioritization, sharing of resources, debugging aids (these beyond that made available by the OS), and most importantly a

consistent interface of objects to the system. Then decide on the first thread of real functionality. They then have the Integration Thread, the foundation to which all new software will be added. It is equivalent to a square piece of clay, to which new bits of shaped clay get added, until a sculpture emerges.

□

Some Immediate Benefits

□

Every day the software team builds on success and has something working. When errors occur, they are due to the new piece added, and are easy to isolate. Changes in function and parallelism of effort can be accomplished with no danger to the foundation. Demonstrable products exist right from the first thread. Project progress is easier to gauge because the software progresses through increasing observable functionality. At every iteration, the iterations before are more exhaustively tested.

□

□

□