The role of logical decomposition in system architecture

Murray Cantor

August 15, 2007

from The Rational Edge: In the case of software systems, complexity may be measured in terms of the number of paths activity may follow to arrive at a given state. The more possible paths, the more complex the system. Read how Murray Cantor develops this thought in the context of system architecture.

"Long ago, I tired of the debate between those who favor logical decomposition; i.e., decomposing a system into object classes, and those who favor functional decomposition, using a number of techniques to decompose the requirements and allocating them to system components, in system and software system architecting. I believe that both decompositions have their place and system engineers should understand both approaches and how they relate. However, this article is not about that. Rather, in this brief note, I will endeavor to explain why logical decomposition is growing in importance in system architecture.

Defining "complexity"

A good place to start is to discuss what is meant by "complexity." Here is a heuristic definition:

"... the degree to which a system or component has a design or implementation that is difficult to understand and verify".

More technical definitions come from such measures as cyclomatic complexity developed by McCabe and others. By inspecting the cyclomatic complexity measures, I have found it useful to reason about complexity via the following question: How hard is it to determine how a system got into a certain state?
By *system state*, I mean a configuration of the system elements. This can include such software variables as physically realized bits in memory and registers, physical positions such as the relative position of the components of a robotic arm or the parts of an automotive steering subsystem, voltages in electrical components, etc. The state space may be thought of as a big vector space with each basis element spanning the possible states of one system element (say a memory register or the cam angle). One must understand that systems evolve from one point to another in this state space. A system then can be said to be *simple* if there are not many paths to a given point and *complex* if there are lots of paths to a given point. In the latter cases, it is very hard to understand how to repair or interact with a system if every so often it is in an undesirable state.

**Software defects**

This perspective is practical because a system defect arises when the system is in some unexpected state that in turn causes some undesired outcome. If a system is simple, then it is not hard to understand how the system got into a given state and how to prevent that state from recurring. (This is "debugging.") If the system is a jumbled mess, then the system might get into an undesired state through any number of paths and so it is hard to debug. Further, such a system is hard to extend because it is so difficult how to predict how the added state will interact with the existing state.

Classically, purely physical systems that fulfill their purpose with a small number of parts are considered preferable to those with more parts -- partially because they have a smaller state space and so are less likely to have defects.

The number of state transitions (moving from one point to another in the state space) grows quadratically with the size of the state space and of course the number of possible paths in the state space grows exponentially. So unless something is done to divide and conquer the state space, systems become unmanageably complex. Cyclomatic complexity is a way to measure how tangled the paths are in the state space.

**The rise of object methods**

This unmanageable complexity emerged early in the days of computing. As software programs grew, they became complex is the sense I have been discussing. Soon it became clear that good programming practices could help organizations avoid complexity. For example, programmers soon learned to avoid the dreaded "goto" statement for just this reason. Coping with complexity led to structured languages such as Pascal and C, which created hierarchical structures of state variables that provided a useful partitioning of the state space. The next evolution was object languages, which bundled the state variable structures and the operations that affected them into entities called class objects. Creating a good partitioning of the software state space is a key goal of object design. The aim is to reduce complexity, and enable maintainability and extendibility, while still providing the functionality.

Object methods bring other advantages to designing stateful systems. If the running system has many instances of the same kind (say, an interface graphical element in an IT system or a sensor
message), each will have the same state model. Further, if there are similar items in the system, like different user controls on a tape recorder, object methods can ensure that they have equally similar state models. All of this aids in maintainability and extendibility.

Object methods are becoming increasingly useful for overall system architecture for two reasons:

1. Modern systems of all kinds are growing more stateful for a number of reasons, including the amount of embedded software, more capable hardware, and the integration enabled by modern network technologies.
2. The state is no longer easily partitioned by physical subsystems. Consider the automobile. Fifty years ago the state of the vehicle was easily partitioned into the state of the subsystems (steering, drive train, electrical, ...) and these systems had minimal, if any, interaction. Today each of these systems interacts through internal networks, sensors, and electronic control units (“black boxes”). These interactions enable collaborative functions like skid recovery or automated parallel parking. Each subsystem’s states directly affect the each other’s states, and so the state transitions of a modern vehicle are much more interdependent.

Positive implications for system engineering

Hence, the problems first faced by the software community -- dealing with the complexity of the state space -- now face the system engineering and design community. It is not surprising, then, that similar solutions are appropriate. In particular, the goal of object system design is to partition the system state to meet the same goals of maintainability and extendibility faced in software design.

Of course, system engineers must address a broader set of concerns than software engineers, such as the many "-abilities" (reliability, availability, usability, serviceability...) along with total cost of ownership, power consumption, and the like. So no one decomposition is sufficient. And while logical decomposition is often essential, it is not sufficient for system modeling. It is for that reason I recommend decomposition from a variety of viewpoints and joint realization methods.⁵

Notes


4 See http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html