Presented at OOPSLA 2002

Challenging Encapsulation in the Design of High-Risk Control Systems

Daniel Dvorak
JPL / California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
1-818-393-1986
Daniel.Dvorak@jpl.nasa.gov

ABSTRACT

In the hardware/software design of control systems it is almost an article of faith to decompose a system into loosely coupled subsystems, with state variables encapsulated in device and subsystem objects. The engineering advantages of such an approach are so attractive that it is sometimes applied inappropriately, yielding a design that hides a tangle of special-case subsystem-to-subsystem couplings behind a façade of modular decomposition. The limitations of a subsystem/device architecture become apparent in the design of high-risk control systems—such as nuclear power plants and planetary rovers—where the world is full of physical side-effects that have little "respect" for conventional subsystem boundaries. Here, the very notion of decomposition by subsystem, and its attendant state encapsulation, actually complicates the design. Fundamentally, there is a clash between a subsystem-device-object metaphor and the laws of physics. A more appropriate architectural approach is to acknowledge the underlying physics and to elevate the concepts of *state* and *models* to first-class design elements that are *not* encapsulated within subsystem objects.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures – *domain-specific architectures, information hiding.*

General Terms

Design.

Keywords

Encapsulation, isomorphism, state, model, control system, design, architecture.

1. INTRODUCTION

Software systems vary enormously in the extent to which they interact with the physical world and deal with its subtleties. At one extreme are enterprise applications such as management information systems that deal primarily in a tidy world of data management, queries, and reporting. Such systems are typically deployed in an environment of plentiful resources — plenty of data storage, network throughput, electrical power, and air conditioning. In such an environment most physical side effects can be safely ignored. For example, powering up a disk drive in a server room consumes power, generates heat, and imparts a rotational torque on the disk drive assembly. These are real physical effects, but we can safely ignore them as irrelevant side effects when the resources that they affect are virtually unlimited. In this situation, power and air conditioning and rotational inertia are all virtually unlimited.

At the other extreme are resource-limited robots such as unmanned spacecraft and Mars surface rovers. Since the amount of mass launched into space is a major cost driver for space missions, these systems are engineered to carry only enough resources to accomplish mission objectives, plus a small margin. Mission activities must be designed to operate within tightly engineered constraints on electrical power, battery energy, non-volatile memory, communication link throughput, and many other resources. For example, turning on a camera to take pictures draws from a limited power budget, consumes non-volatile memory to store images, and requires the rover basebody to be pointed appropriately. This activity uses precious resources that are then *not* available to other activities. The net result is that in a resource-limited system many physical side effects become *non*-negligible and therefore must be consciously managed; designs become more complex because the couplings are more numerous and often cross conventional subsystem boundaries.

This paper compares two architectures with respect to their suitability for resource-limited control systems. One architecture is subsystem/device-oriented, having objects associated with hardware units, such as drive motors and camera, plus objects associated with traditional engineering subsystems such as electrical, thermal, and navigation subsystems. This architecture encapsulates state variables inside such objects—objects that logically seem to "own" those state variables. The other architecture is state/model-oriented, having first-class objects associated with physical states, such as camera temperature and battery energy, plus objects associated with models of physical couplings, such as the effect of a heater on power consumption and the effect of temperature on a sensor measurement. Despite the appeal of decomposition by subsystem, the structure of a

subsystem/device architecture depends on an assumption of loose coupling that simply doesn't hold in the realm of resource-limited systems. Such systems must manage numerous tight couplings of the real world. Dealing with this in a disciplined way demands an architecture that acknowledges and adequately represents the underlying physics of the world.

2. DEVICE/SUBSYSTEM ARCHITECTURE

Most physical systems that people deal with on a daily basis are designed as a composition of modular subsystems that interact in a few obvious and easily controlled ways. Such systems are easier to understand, easier to monitor and control, and easier to diagnose when faulty. For example, the different subsystems of a modern home—electrical, heating/cooling, plumbing, telephone, and cable—are relatively independent of each other, with only a few important forms of coupling. In truth, there are *many* physical couplings among these subsystems, but most are negligible. For example, the electrical subsystem is a source of electromagnetic interference to the telephone and cable subsystems, but the signal-to-noise ratio is high enough that the effects can be ignored. Similarly, the circulation of hot water in the plumbing system affects the heating/cooling system, but the effect is negligible compared to the heating/cooling subsystem's ability to notice and compensate. These and other physical effects are negligible because the system has abundant resources: plenty of electrical power, abundant heating/cooling capacity, large thermal mass, substantial electrical and thermal insulation, plenty of electromagnetic shielding, etc.

In such an environment of plentiful resources, control system software designers can appropriately treat subsystems as largely independent, with state variables encapsulated in the subsystem or device object that has the dominant effect or "ownership" of that state. For example, hot water temperature could be encapsulated with a water heater object since the water heater has the dominant effect on that state. Of course, there are some subsystem couplings that cannot be ignored because they have intentional effects or major side effects. For example, an electric hot water heater depends on power from the electrical subsystem in order to operate, so this one-way dependency must appear somewhere in the system logic. This kind of subsystem coupling is so simple to describe and so few in number that it's typical for a software designer to express it on a case-by-case basis, often in the logic of one or two operations of a class. Figure 1 shows an ideal decomposition by subsystem, where the only couplings are those between a child subsystem and its parent subsystem.

2.1 Example: Home Heating System

An illustrative example of the subsystem/device approach to control systems is the "Smalltalk Home Heating System" described by Booch [1, chapter 8]. As Booch notes, the home heating system naturally decomposes into relatively independent subproblems. The heating system contains top-level objects of a furnace, heat flow regulator, operator interface, and a home consisting of multiple rooms. Each room has a current temperature sensor, a desired temperature setting, a room occupancy sensor, and a water valve.

Note that top-level software objects model the obvious physical elements of a home heating system, and that makes sense as long as most or all of the couplings exist *within* containment and/or attachment relationships. For example, in this system there is nothing outside of a room that affects the measurements from the current temperature sensor, so it makes sense to encapsulate the current temperature state within the room or its temperature sensor.

In summary, a device/subsystem architecture is appropriate for many everyday control systems because they exhibit simple subsystem couplings. Such an architecture is extremely attractive to most designers because software structure reflects hardware structure and follows familiar subsystem-oriented decomposition. The only danger, which this paper explores, is that designer familiarity with this appealing architecture can lead to its use in applications where physical couplings are complex and have little "respect" for the hardware structure and subsystem boundaries.

2.2 Couplings Due to Physics

The limitations of a subsystem/device architecture aren't apparent in "everyday systems" in much the same way that the limitations of Newtonian physics aren't apparent in everyday experiences, where velocity is a tiny fraction of the speed of light. In the case of the subsystem/device architecture the limitations start to appear in complex systems where the everyday assumptions of loose coupling simply don't hold. To illustrate this point, consider a common, everyday device: a battery-powered electronic thermostat.

The job of a thermostat is to regulate temperature by sensing the temperature and by issuing on/off control actions to maintain temperature within a specified range. The thermostat itself is a self-contained hardware unit with two simple couplings: it senses ambient temperature and it opens and closes electrical contact between two terminals. Since we're talking about software design, let's assume that this electronic thermostat has a microprocessor running software that performs the temperature regulation. This looks like a perfect example of a hardware device that is loosely coupled with respect to an overall heating system, and indeed it is if that's an everyday home heating system.

Now consider the same task of temperature regulation, but in a very different environment: a Mars rover. Two things make this system very different: electrical power is extremely limited due to battery and solar panel limitations, and fault protection is a major concern since there are no repair technicians on Mars (as far as we know ☺). A consequence of limited power is that temperature regulation cannot be treated as an isolated activity; it has to be coordinated with other power-consuming activities such as driving, communication, and science instrument usage. Thus, as much as we would *like* to think of temperature regulation as the duty of a self-contained thermostatic unit, it can't be designed that way when it relies on the availability of an extremely limited resource.

In a similar way, designing for fault protection reveals other flaws with the idea of a loosely coupled thermostat. For example, if the temperature sensor fails, how do you estimate ambient temperature? Well, thanks to physics, there are other sources of evidence about temperature that can and should be used. For example, the recent history of the heater's on/off state provides important evidence about heating. The position of the Sun and its heating effect can be predicted with a thermal model. Likewise, power usage of nearby instruments and other devices has a heating effect that can be predicted, provided that those power states are accessible. Now, our once-simple thermostat has a lot more couplings and a lot more to think about. Also, suppose that the heater fails. How then can you control temperature? Well, again, physics offers the clues. One way may be to turn on nearby instruments solely for their heating effect, subject to power

availability. Another way is to reschedule activities that depend on temperature regulation to occur during mid-day on Mars when solar heating is at a maximum. Again, our once-simple thermostat is being asked to exercise control that goes far beyond its original role in the "thermal subsystem". The very concept of a self-contained thermostat is falling apart because its design rests on an assumption of loose coupling that simply doesn't hold in this more complex domain.

3. COUPLING AND URGENCY

In a significant book that analyzed accidents in complex systems such as Apollo 13 command module and the Three Mile Island nuclear power plant, Perrow [4] summarized the inherent risks in different types of systems using a Coupling/Urgency chart[1], as shown in Figure 2. The horizontal axis of *coupling* ranges from linear to complex. Linear couplings are those in expected and familiar production or maintenance sequence, and those that are quite visible even if unplanned. Complex couplings are those of unfamiliar sequences, or unplanned and unexpected sequences, and either not visible or not immediately comprehensible. As Stevens et al [5] explain, "strong coupling complicates a system since a module is harder to understand, change, or correct by itself if it is highly interrelated with other modules."

The vertical axis of *urgency* ranges from low to high. Low urgency systems can incorporate shocks and failures and pressures for change without destabilization. Low urgency systems tend to have ambiguous or perhaps flexible performance standards. High urgency systems have more time-critical processes: they cannot wait or stand by until attended to. Chemical reactions, as in pharmaceutical plants, are almost instantaneous and cannot be delayed or extended.

The placement of systems on this chart is based entirely on Perrow's subjective judgments because there is no standard way to measure the two variables of coupling and urgency. Nonetheless, the chart offers a useful qualitative comparison of different kinds of systems, and the history of system-level accidents supports his finding that complex couplings and high urgency make systems more prone to mishaps.

This paper focuses on coupling as an architectural driver. While the time-sensitive aspect of urgency is certainly important in system design, it is not the main point of this paper.

3.1 Coupling in Space Missions

As Figure 2 shows, space missions exist in the upper right quadrant of complex coupling and high urgency. Systems in this quadrant are at the highest risk for system-level accidents because they are harder to design and operate correctly.

Space missions exhibit complex coupling because many resources are severely limited. Some limitations, such as battery energy and solar panel power production, are due to the high cost of launching mass into space. Smaller batteries and smaller solar panels help reduce that cost. Other limitations such as processing speed and instrument usage ensue from the power limitation; running the processor at a lower clock rate and using one instrument at a time reduces power consumption. Still other limitations arise from the vast distances of outer space, where data communication rates fall as the square of the distance between transmitter and receiver. That means that it takes a long time to transmit data, and that activity typically precludes other activities while the antenna is carefully pointed at a moving target (such as Earth). Antenna pointing usually depends on basebody pointing, which is another managed resource.

Coupling occurs in many ways, including coupling through shared busses, structure, thermal proximity, grounding, environment, and so on. Most couplings are a direct consequence of system-level design, such as an instrument that will be damaged if it is in the wrong mode when thrusters fire. In addition, some couplings result from hardware design flaws that are discovered too late to fix, prior to launch. Examples include motor commands that cause processors to do a power-on-reset and communication busses that lock up when the wrong combination of units is active. To exaggerate just a bit, in a resource-limited system "everything affects everything".

3.2 Problems of Device/Subsystem Approach

The main problem in applying a device/subsystem architecture to resource-limited systems is that the architecture provides no leverage in dealing with the many non-negligible inter-subsystem couplings. Each such coupling has to be handled as a special case, leading to a tangle of subsystem-to-subsystem interactions hidden behind a façade of modular decomposition. In effect, the original architecture becomes an appealing fiction.

If a system is to be controlled efficiently then these couplings must be taken into account, for otherwise some less efficient scheme would have to be used in a loosely-coordinated manner. An example of the latter in spacecraft operations has been to reserve generous resource margins to ensure that a desired activity succeeds in spite of its side effects on limited resources. For example, operators may hold a 25% power margin above and beyond the predicted needs of the planned activities. This conservative strategy is understandable given the unforgiving nature of outer space, but it causes a spacecraft or rover to be significantly underutilized relative to its potential.

Interestingly, the practice of iterative development coupled with a subsystem decomposition can lead a project into a kind of architectural trap. Iterative development enables a team to demonstrate early progress and gain confidence by building a solution to a simplified problem, and then iterating to extend and refine the design. Unfortunately, the initial simplifying assumptions may be quite compatible with subsystem decomposition, leading the project into an architecture that fails to help when it is needed most—late in the development lifecycle when high-fidelity behavior must be achieved. As new iterations require higher fidelity behavior, new couplings that cross device and subsystem boundaries must be handled. Each one by itself is a small blemish on an otherwise tidy architecture, but achieving true high-fidelity behavior for the final delivery can overshadow the original architecture with a mess of strapped-on couplings.

4. STATE/MODEL ARCHITECTURE

If a subsystem/device architecture is problematic for resource-limited systems, then what's a better approach? At a minimum, it has to be an approach that facilitates a software description of physical interactions, since management of those interactions is a dominant force in the design of resource-limited systems. It has to describe how things affect each other in the physical world, and this is exactly the role of *models* in the state/model architecture.

[1] To more closely match computer science terminology, this paper uses the terms 'coupling' and 'urgency' in place of Perrow's 'interaction' and 'coupling', respectively.

As described below, there are three kinds of effects to model: measurement effects, command effects, and state effects. These models exist to support state estimation and state control, described later.

Just as the notion of *model* is elevated to a first-class entity, so also is the notion of *state variable*. Many state variables have no obvious encapsulating home within a subsystem-oriented architecture because many physical influences on their values have no "respect" for boundaries drawn by subsystem designers. Such state variables must stand on their own, apart from subsystems. The notion of state used here is broad, including many kinds of physical quantities such as temperature, pressure, switch position, device health, and position of one body relative to another. Together, state variables and models provide the means for describing physical interactions in software.

4.1 State Variables

In the realm of control systems, state variables are what system engineers identify and what operators monitor and control. Example states include the on/off position of a power switch and the orientation of a spacecraft. "State knowledge" always has associated uncertainty because sensors are imperfect, as are our models of how things work. Explicit representation of uncertainty enables estimators to be honest about the evidence and controllers to be cautious during periods of high uncertainty.

4.2 Models

4.2.1 Measurement Effects Models

Sensors are hardware devices that produce measurements. Most real-world sensors are designed to measure a particular physical quantity, but they inadvertently and/or unavoidably measure other quantities. For example, a voltage sensor will produce a voltage measurement, but its value may be sensitive to temperature and magnetic field strength. Its value is also sensitive to its own calibration parameters of bias and scale factor. Finally, its value is affected by the sensor's health state, which may be in any of several failure modes.

A measurement model is a mapping from state(s) to measurement. In the example above, the voltage sensor's measurement model is a function of six states: voltage, temperature, magnetic field strength, sensor bias, sensor scale factor, and sensor health. Notice that temperature and magnetic field strength are *external* influences on the voltage measurements. Hence, this measurement model expresses two interactions that are independent of a subsystem hierarchy.

4.2.2 Command Effects Models

Actuators generate physical effects in response to commands. In addition to their intended effect, many actuators have unintended and/or unavoidable side effects. For example, a command to turn on a science instrument on a Mars rover has the desired effect of activating the instrument, but it also draws power from a limited supply, it causes localized heating that may affect other things (such as the voltage sensor mentioned previously), it may generate a magnetic field that interferes with another instrument, and it may start transmitting on the data bus, using up part of its limited capacity. Finally, the effects always depend on the actuator's health state, which may be in any of several failure modes.

A command effects model predicts the multiple effects of a command issued to an actuator in a given state. In this example the command effects model must predict the effect of a particular command on the values of five states: instrument activation state, battery power, nearby temperature, nearby magnetic field, and bus data rate. Notice that all of these effects, except for instrument activation, are *external* to the instrument. Hence, this model expresses four couplings that would violate an idealized subsystem hierarchy.

4.2.3 State Effects Models

In the physical world some states affect other states according to laws of physics and/or consequences of hardware design. For example, Boyle's ideal gas law expresses the relation between pressure state, volume state, and temperature state ($PV = nRT$). Similarly, the voltage drop across a resistor in an electrical circuit is a consequence of Ohm's law ($V=IR$). Likewise, the open/closed state of a valve affects flow state as well as both downstream and upstream pressure states.

A state effects model expresses such functional relations among states, and just as with measurement effects models and command effects models, the effects often span subsystem boundaries. Further, these are not necessarily just one-way effects; the ideal gas law describes a constraint that holds among multiple variables, any of which may be controllable or uncontrollable in a given system.

4.3 Estimators and Controllers

The three kinds of models described above provide a disciplined way of representing interactions that *must* be reasoned about in resource-limited systems. Accordingly, the architecture should elevate the concepts of state and models as first-class elements so that the numerous inter-subsystem couplings can be exposed and represented, not concealed through back-door device-to-device and subsystem-to-subsystem connections.

Such an architecture must perform state determination and state control *somewhere*, but in general it can't be done inside device or subsystem objects because they don't have sole 'ownership' of the states. As the preceding sections on models illustrated, for any given state there may be different measurements from different sensors that provide evidence about its value. Likewise, for any given state, there may be different commands to different actuators that can affect its value.

These simple facts suggest that *estimators* and *controllers* also need to be first-class architectural elements, distinct from the software objects for sensors and actuators and their aggregations. After all, if there are multiple sources of evidence about a state's value, there should be one entity that combines that evidence into an estimate. Likewise, if there are multiple ways of influencing the value of a state, there should be one entity that has overall responsibility for controlling that state.

Estimation and control are seen as distinct elements in this architecture and should *not* be combined, as is often the case in a subsystem approach. The simplest reason is clarity and correctness; it is easier to design, develop, and test two software modules where each has a single purpose than one module that tries to do two distinct things.

The job of an estimator is to update state knowledge by interpreting many sources of evidence—from measurements, commands, and state variables—given models of how things work. Evidence may be noisy, inconsistent, corrupted, and incomplete. In contrast, the job of a controller is to issue commands, as appropriate, in an attempt to influence the value of

a state variable to satisfy a goal. Commands may have delayed effects and actuators may fail.

A second reason for separating estimation from control is more subtle; when the two tasks are combined, there is a temptation to shortcut the estimation process and never actually estimate the state to be controlled, but rather to modify flags and counters that the control logic "understands". This practice leads to systems that are hard for operators to monitor and understand because many key states are never explicitly estimated, and so the only way to understand them is to read the code.

4.4 Hardware Adapters

In this architecture the role of the hardware device object has been diminished as compared to the subsystem/device architecture. Its main role now is to provide access to the hardware sensors and actuators. Estimators obtain measurements from sensors as inputs to the state estimation process, and controllers submit commands to actuators to influence physical state. In many cases, state variables that seem to be owned by a device should *not* be encapsulated in such objects because fault diagnosis reasoning within estimators and fault response logic within controllers often need access to such "internal" states.

4.5 Mission Data System

The state/model architecture just described is the architecture of the Mission Data System (MDS). Although MDS is broadly applicable to control systems, it is particularly suited for resource-limited control systems such as unmanned spacecraft and planetary rovers [2]. The MDS architecture can be understood in terms of a few basic elements, as depicted in Figure 3.

- *State*. The MDS architecture is fundamentally state-based. States are what system engineers identify, what software engineers design and implement, and what operators monitor and control. Example states include the on/off position of a power switch and the orientation of a spacecraft. "State knowledge" always has associated uncertainty because sensors are imperfect, as are our models of how things work. Explicit representation of uncertainty enables estimators to be honest about the evidence and controllers to be cautious during periods of high uncertainty.

- *Models*. Much of what makes software different from mission to mission is domain knowledge about instruments, actuators, sensors, wiring, plumbing and many other things. By expressing such knowledge in inspectable models, apart from reusable software, the task of customizing MDS for a mission, then, becomes more a task of defining and validating models. Importantly, measurement models, command effects models, and state effects models provide an architectural basis for representing couplings.

- *Goals*. Goals are the basis for mission operations. A goal specifies operational intent as a constraint on the value of a state variable during a time interval. Importantly, a goal does not specify actions needed to accomplish it, thus leaving options open for autonomous control mechanisms. Goals enable operators to focus on *what* to accomplish rather than *how* to accomplish it. Active goals live in a goal network that specifies parent/child relationships and timing & ordering relationships.

- *State control*. State control encompasses the mechanisms devoted to goal achievement. This includes elaboration of a goal into subgoals, scheduling of goals on state timelines, time-based and event-based initiation of goal execution, delegation for real-time coordinated control, and hardware commanding.

- *State determination*. The task of estimating system state requires interpretation of many sources of evidence—such as measurements and commands—given a model of how things work. Evidence may be noisy, inconsistent, corrupted, and incomplete. State determination is a complicated enough job that it is deliberately separated from state control, thereby facilitating understandability, verification, and reuse.

5. RELATED WORK

In a 1995 joint study between NASA Ames and JPL known as the New Millennium Autonomy Architecture Prototype (NewMAAP) a number of existing concepts for improving flight software were brought together in a prototype form. These concepts included goal-based commanding, closed-loop control, model-based diagnosis, onboard resource management, and onboard planning. When the Deep Space One (DS-1) mission was subsequently announced as a technology validation mission, the NewMAAP project rapidly segued into the Remote Agent project [3]. In May 1999 the Remote Agent eXperiment (RAX) flew on DS-1 and provided the first in-flight demonstration of the concepts. The MDS project was established in April 1998 to define and develop an advanced multi-mission data system that unifies the flight, ground, and test elements in a common architecture. That architecture is shaped with the themes described in this paper, some of which were explored and refined by the RAX experience.

6. SUMMARY AND CONTRIBUTIONS

In the design of everyday control systems, the "divide and conquer" approach decomposes a system into loosely coupled subsystems that reflect traditional engineering disciplines such as power, thermal, navigation, telecommunication, science, etc. Each subsystem "owns" the estimation and control of particular states, so those state variables are encapsulated within the subsystem or its sub-subsystems, including proxy objects for devices that are considered to be part of the subsystem. This approach is workable—provided that the subsystems are loosely coupled—and is appealing because it supports a work breakdown according to engineering disciplines.

In contrast, high-risk control systems differ from everyday control systems in that the traditional subsystems are *not* loosely coupled, for two main reasons. First, in an environment where numerous activities compete for a share of limited resources, those activities must be coordinated in a way that is simply unnecessary when the resources are virtually unlimited. Second, in an environment where fault-tolerant control is a high priority, control decisions often must extend beyond the confines of a single subsystem. In short, the fundamental premise behind subsystem decomposition—loose coupling—does not hold, so a design based on such a decomposition would have to violate its own premise numerous times to deal with couplings that cross subsystem boundaries.

In designing for a high-risk control system, analysis of the physics of interactions suggests the shape of a more suitable architecture. States of the physical world clearly exist, but they do not owe

their existence to a subsystem; they simply "are", and the job of the control system is to estimate their values as best as possible and control them as best as possible, even in the presence of faults. Estimation and control both depend on knowledge of how things work and how they fail, and that knowledge must be expressed somewhere as models of states, commands, and measurements.

The engineering contributions of this paper lie in five design principles of the state/model architecture: (1) state variables and models as first-class elements rather than subsystems; (2) explicit use of models to express the physics effects of couplings; (3) clean separation of state determination logic from state control logic; (4) explicit management of physical resources (power, memory, etc); and (5) the use of state constraints for operational control.

Designing for high-risk systems requires a paradigm shift from subsystem-oriented to state-oriented thinking. "Divide and conquer" must give way to "state analysis and physics modeling". Managing interactions is the key to good design in this domain, and if architecture is to be a help rather than a hindrance, it must facilitate representation and reasoning about such interactions.

It is not the intention of this paper to criticize state encapsulation or information hiding but rather to rethink what kinds of states are encapsulated in what kinds of classes. In a subsystem-oriented architecture the classes represent subsystems and devices, and the encapsulated states are seen as states that are wholly owned and/or wholly controlled by its subsystem. In a state-oriented architecture some classes represent and encapsulate individual physical states and have query, update, and notification operations for appropriate clients. Other classes represent such clients for state estimation, real-time control, and deliberative control. The lesson here, in the context of high-risk control systems, is that some state variables should *not* be encapsulated within subsystem objects because there is no single subsystem having full responsibility for the variable's value.

7. EPILOGUE

Engineering disasters can be great learning experiences. The 1930s design of the first Tacoma Narrows Bridge followed a popular trend toward lightness, structural grace, and flexibility. In fact, the original design had a 25 foot deep stiffening truss, but was later changed to an eight foot shallow plate girder, resulting in a much lighter bridge. Although the bridge was the epitome of artistry, it collapsed spectacularly in 1940 due to wind-induced vibrations because aerodynamic phenomena had not been adequately addressed in the design.

The same dangers of esthetics versus physics exist in software design, especially since the appearance of a design in UML diagrams (its esthetics) tends to be more visible to software engineers than the physics at play. Another esthetic is the appeal of a popular pattern, such as decomposition by traditional engineering subsystem; it seems reassuring since it has worked so well before. The fact that it clashes with the physics of interactions is sometimes hard to see because software is so malleable; it's always easy to add "one more interface" to accommodate a newly discovered need. The architectural end result becomes an appealing fiction: a tidy set of subsystems that hide a tangle of private, back-door interactions.

As software architects we must be careful about applying comfortable metaphors since they have the power to lead us astray. Object-oriented analysis is appealing because people can engage in anthropomorphic storytelling as a design strategy. That encourages secondary metaphors like 'ownership', which then map into subsystems and encapsulation. The fact that this approach works well in everyday control systems encourages architects to apply it to all control system problems. With such a mindset, it is hard to recognize when a new design problem is qualitatively different from previous successfully solved problems. The best antidote for this is an objective analysis of the phenomena in play and the system-wide couplings that must be managed; those are the keys to good design. Only after that is done should one consider architectural styles.

8. ACKNOWLEDGEMENTS

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. Robert Rasmussen, chief architect of the Mission Data System, originally identified the existence of complex interactions—and the need to manage them architecturally—as a key driver in the MDS state/model architecture. Erann Gat documented this driver in an early internal document and later helped to shape the MDS state architecture. In another internal document, Kirk Reinholtz, chief programmer of MDS, documented the key limitation of the subsystem/device architecture in resource-limited systems.

9. REFERENCES

[1] Booch, G. Object Oriented Design with Applications. Benjamin/Cummings Publishing, 1991.

[2] Dvorak, D., Rasmussen, R., Reeves, G., and Sacks, A. Software Architecture Themes in JPL's Mission Data System. Proceedings of the 2000 IEEE Aerospace Conference, Big Sky, Montana, March, 2000.

[3] B. Pell, D. Bernard, S. Chien, E. Gat, N Muscettola, P. Nayak, M. Wagner, B. Williams. An Autonomous Spacecraft Agent Prototype. Proceedings of the First Annual Workshop on Intelligent Agents, Marina Del Rey, CA, 1997.

[4] Perrow, C. Normal Accidents: Living with High-Risk Technologies. Basic Books, 1984.

[5] Stevens, W., Meyers, G., and Constantine, L. Structured Design, in Classics of Software Engineering, Yourdon Press, 1979.

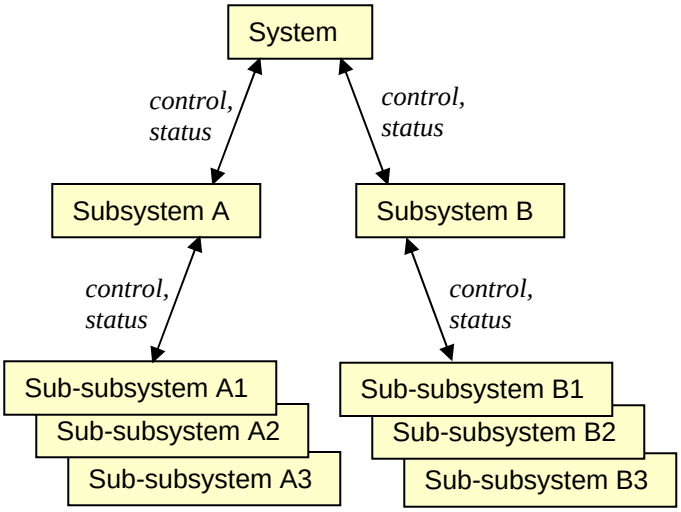

Figure 1. An architecture based on subsystem decomposition rests on an assumption of loose coupling, where interactions among subsystems are handled via hierarchical pathways of control and status. In such an architecture state variables are encapsulated within the object that has responsibility for its estimation and control.

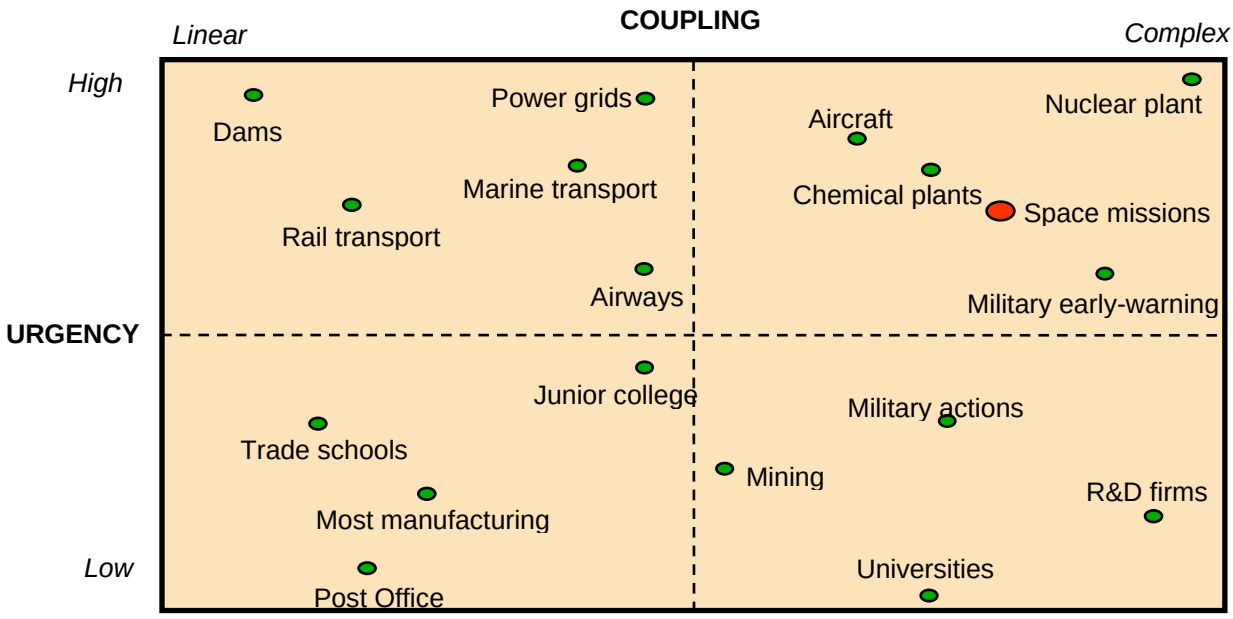

Figure 2. Systems that exhibit complex coupling and high urgency are considered high-risk because they are more prone to system accidents. This chart is due to Perrow [4].

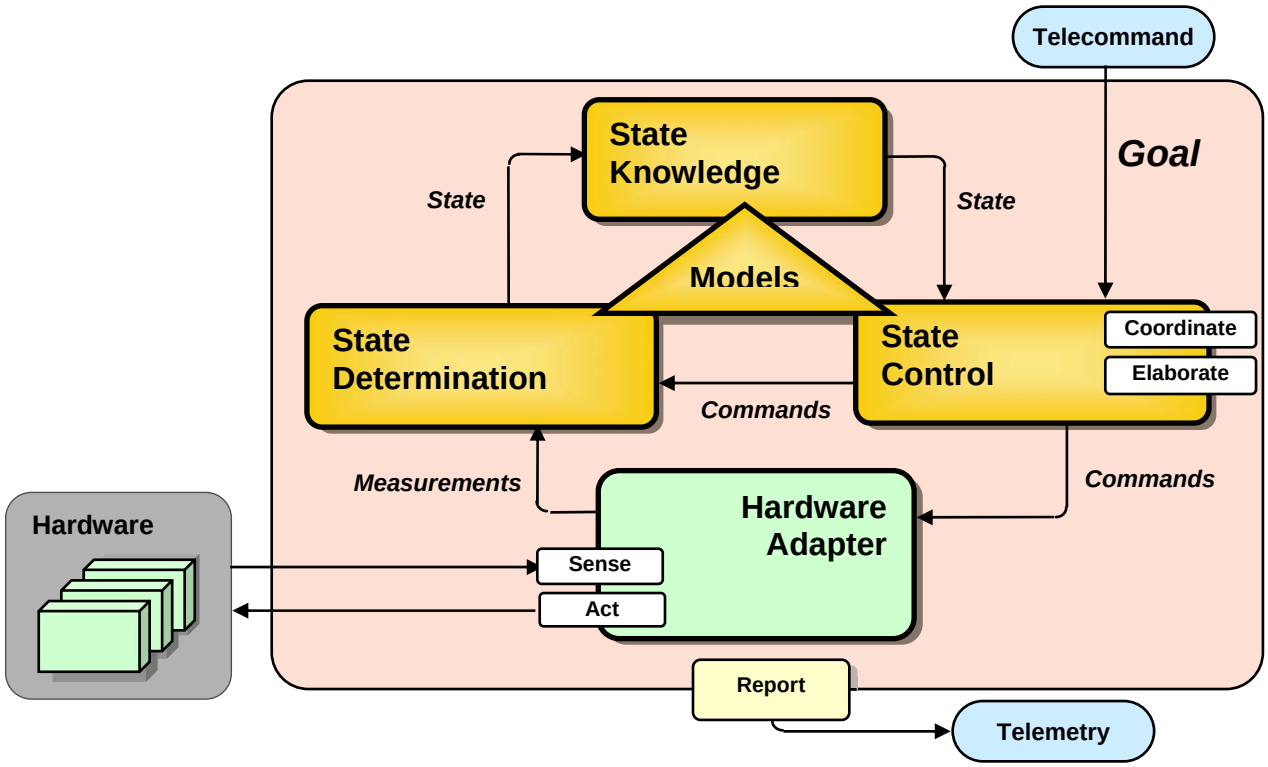

Figure 3. The state/model architecture of the Mission Data System emphasizes the central role of state knowledge and models, goal-driven operation, and separation of state determination from control.