Effective Practices for Object-Oriented System Software Architecting

Rich McCabe and Mike Polen
Systems and Software Consortium

Development programs for software-intensive systems are increasingly attempting to employ object-oriented (OO) techniques and technologies – including OO design, the Unified Modeling Language (UML), and UML-based modeling and software development tools – in expectation of achieving greater flexibility, evolution, and productivity. However, these programs frequently experience a number of challenges when they insert OO design into their traditional practices. Unless both development organizations and acquisition offices make a thoughtful transition to OO design, they are likely to experience difficulties that may well endanger the anticipated benefits. This article describes some typical pitfalls of OO development and recommends a number of architectural practices that will help programs avoid or mitigate these dangers.

Object-oriented (OO) design as a software programming technique has been around almost 40 years, although it has become prominent in military and government systems development in only the last few years. Its use has grown from just a programming trick to a complete approach to analyzing and solving complex problems.

As with any technology, OO design has not always been applied successfully. Literature is full of misapplied OO techniques as well as impractical recommendations. The authors have seen both good and bad OO programming in large military systems. This article will try to enumerate the most common pitfalls and offer practices that have been shown to work.

The practices recommended in this article are motivated by the desire to use OO technology to its best effect. OO techniques provide the opportunity to capture the logic of a complex domain or problem environment in a structure that reflects or is congruent with interrelationships among actual domain elements. This structure encapsulates the impact of volatile requirements and design decisions, and provides a natural mapping from requirements to design. Consequently, the design is flexible, facilitating system integration and repeated adjustments or enhancements to the implementation during initial development and subsequent maintenance. Therefore, these attributes of congruence and flexibility reduce system life-cycle costs and are worth striving for rather than settling for any design that can be made to *work*.

Practices the authors have used to help struggling OO projects include the following:

- System-wide software architecture.
- Layered software architecture with domain layer.
- Goal-directed black box use cases.
- Spiral design.
- Architecture and code iterations.
- Spare modeling.
- Experienced OO guides.

These practices have been widely recommended by many OO experts. The authors' experience is that they are just as valid for complex system development within government acquisition programs. The authors have incorporated many of these recommendations into a prescriptive methodology [1].

"OO techniques provide the opportunity to capture the logic of a complex domain or problem environment in a structure that reflects or is congruent with interrelationships among actual domain elements."

The following sections describe these practices in more detail. However, to keep this article within an acceptable length, it is assumed the reader is familiar with typical development practices that are referenced during the discussion.

System-Wide Software Architecture

One of the most effective means of preserving congruence between the domain and the design is to use OO techniques to decompose the software across the entire system. Working from a software perspective with such a broad scope yields a unified, system-wide, software architecture, expressing domain concepts with semantic consistency throughout the software.

Too often, programs follow past patterns of system design that preclude good OO design. System designers, too, readily define logical design components to match organizational structures, physical elements of the design, or sub-functions of deep functional analysis performed without sufficient feedback (e.g., any) from software design. While these decompositions may match conveniently to the specialties of different groups, correspond to the administrative hierarchy of the development team, or enable easy traceability, they constrain the scope of any software perspective to individual *boxes*, conflict with OO structuring, and increase the difficulty of maintaining overall semantic consistency (e.g., consistent interpretation of data) among the components.

Semantic consistency among different parts of a system is extremely important. Trying to maintain that consistency by merely matching interfaces of system components is not sufficient, especially if the major logical interfaces are forced to align with physical interfaces. OO approaches need to work with the software as a whole to define major class interfaces for flexibility and congruency, without the imposition of other decompositional schemes.

The only way to preserve flexibility and congruence of software design in balance with other tradeoffs is for OO developers to work closely with other team members during multiple iterations of the requirements and design. Although software is malleable enough to fit into almost any decomposition, taking for granted the benefits of a system-wide OO structure is almost certain to compromise them.

Layered Software Architecture With Domain Layer

Grouping related classes into *layers* is another technique for managing and coordinating many classes in large systems. The term *layer* is something of a misnomer in that the layers are not strictly arranged like a cake. However, the classes in a layer should all deal with some common aspect of the design and have limited access to other layers (upper layers call upon the services of lower layers but not vice versa). Figure 1 is a representative example. Usually, only one or a few key classes present all the services of the layer to other layers, the other classes of the layer are hidden behind this interface.

One of these layers should be focused on the domain specifics of the system and be the key to understanding the domain. This layer is typically called the *domain layer* (or system or business logic layer), or is named for the particular domain (e.g., accounting, or battle management). Classes in this layer are defined to represent the essential concepts and relationships in the domain in terms that the subject matter experts understand [2]. It should have no knowledge of the underlying hardware, communication protocols, operating system features, or other aspects of the design known to other layers. The domain layer is key to the overall understandability and flexibility of the design.

Typical subsystem partitions are not equivalent to OO layers and usually chop up what would correspond to the domain layer. As discussed in the previous section, the design is often prematurely split into traditional subsystems before any consideration is given to overall domain congruence or flexibility. Frequently, the authors have seen designs structured in the form of all-knowing device-centered subsystems, each containing bits of domain knowledge intertwined with hardware interfaces and other types of design knowledge, and each interconnected to all the others. This kind of design arises from trying to scale up a simple data pipeline or *thread* to a system with multiple, interconnected data pipelines. The result is a fragile, inflexible design with bits of domain and design knowledge expressed in multiple locations.

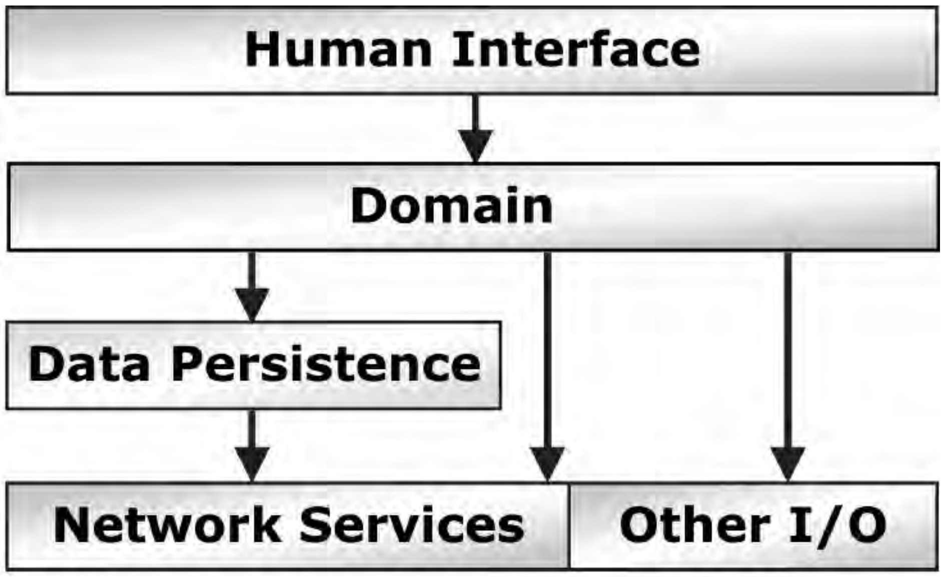

Figure 1: *Generic Layers*

Goal-Directed, Black Box Use Cases

The use case technique [3], when applied appropriately, has the advantage of clarifying the intent of system stakeholders without unduly presuming the design. Use cases are a narrative of system interaction with external entities. Best practice structures each use case around a single goal of a system user or stakeholder, and treats the system as a black box with observable behaviors. Although use cases are a technique for requirements analysis, their black box orientation prevents design assumptions from creeping into requirements. Furthermore, the emphasis on system interaction with external entities in meeting goals helps OO developers identify essential entities and concepts in the domain (as discussed in the previous section).

"Too often, programs follow past patterns of system design that preclude good OO design."

Like any technique, use cases can be misapplied as in the following:

- Although use cases can be applied at different levels (notably to analyze systems containing people: at the outer level to express system interactions with external actors, and at an inner level to express interactions of the hardware/software with people within the system), it is a mistake to mix levels.
- Ignoring stakeholder goals as the organizing principle for use cases leads to haphazard narratives that do not clearly reveal how the system produces value for the stakeholders.
- Another common mistake is failing to abstract away from details of the interfaces, data, or interaction protocols in the use case narrative, or otherwise attempting to develop complete use cases that cover every possible detail or scenario. This treatment leads to a proliferation of endless use cases of rapidly diminishing value.

Use cases should be employed judiciously to illuminate key system behaviors, not to exhaustively document requirements. The opposite extreme, treating use cases as no more than the Unified Modeling Language (UML) use case diagram, is pointless.

Often, use cases are ignored entirely. True, use cases need to be augmented with other requirements techniques, but use cases provide a unique and important perspective.

Spiral Design

A spiral approach to OO design generally yields the best results. Here, a spiral approach means that the designers begin by addressing just a few key issues in an initial design, and then incrementally address other concerns and complications in multiple revisions. Initially, developers make only a rough allocation of responsibilities to components, and gradually resolve the details as design issues are introduced and decided. Here is a partial ordering of issues that appears to work well:

- Domain congruence and flexibility.
- Mapping to physical architecture.
- System attributes.
- Concurrency.

Of course, program-specific circumstances impact this general scheme such as whether the development team has previously developed OO designs for similar systems.

Many programs attempt to directly create a design that simultaneously addresses all system issues and provides detailed interface descriptions as well. This invariably leads to an overly complicated design because the designer has too many concerns to juggle at once. The mental overload causes many errors and results in a *big ball of mud* [4]. Instead, the designers need to start with a simple yet admittedly inadequate design and work in complications one at a time, rebalancing earlier design elements as necessary.

Domain Congruence and Flexibility

The initial design captures the essentials of the domain to ensure flexibility and understandability (see previous section "Layered Software Architecture With Domain Layer"). The intent is to preserve the flexibility of the initial design as much as possible, but compromise it where necessary to address other demands.

Mapping to Physical Architecture

Multi-processor architectures potentially introduce a number of complications best delayed until the domain essentials have been identified. Assignment of functionality to various processors has subtle implications for both timing and reliability. Trying to force these decisions too early before more information is available about other aspects of the design (and even the implementation) is both difficult and dangerous, and tends to emphasize physical interfaces over logical (domain) interfaces. Delaying such decisions is usually advantageous, especially when facilitated by infrastructure technologies such as Common Object Request Broker Architecture (CORBA).

Similarly, OO designs typically encapsulate hardware interfaces inside classes, protecting the rest of the system from volatilities in the device interfaces. This technique usually relegates many of the decisions in *allocating requirements to software or hardware* as a secondary concern.

System Attributes

Timeliness, reliability, safety, security, and other such system attributes are difficult to address individually and often have tricky interdependencies. Designers tend to begin with their greatest concern (e.g., critical timing paths) and orient the design toward that aspect. In the authors' experience, designing for flexibility first and adjusting for timeliness as proves necessary is much more effective than designing first for timeliness and subsequently for flexibility. Similarly, designers need a preliminary design combining both hardware and software before they can meaningfully analyze the impact of other attributes.

"In the authors' experience, designing for flexibility first and adjusting for timeliness as proves necessary is much more effective than designing first for timeliness and subsequently for flexibility."

Concurrency

Concurrency decisions are another area best left until later. Concurrency can add tremendous design complexity. Start with as few concurrent elements that will possibly work (one is often the best starting point). Add new concurrent elements only after a performance test or real-time analysis has shown that the implementation will not work. The authors have seen a narrow design focus on timeliness lead to a (unsuccessful) system design with more than 100 concurrent elements on a single processor.

Architecture and Code Iterations

Although architectural analysis is important, systems today are too complex to rely solely on analysis to ensure that a design will exhibit the expected attributes. The flexibility of a design hinges on too many details that only become apparent with coding. Yet these coding details can have implications for the larger design.

Fortunately, software is well-suited to evolutionary development. Where testing and configuration management discipline is continuously applied, multiple iterations of design, code, and test (in parallel with deepening requirements analysis) are more productive and effective than a waterfall process. Coding the most critical or highest-risk portions of the design validates the solution approach. A robust, flexible design is discovered and becomes increasingly stable through multiple iterations. This practice also fits well with spiral design.

Even though the waterfall process is rarely, if ever, suitable to manage the risks of complex system development, it is still prevalent in government contracting. With the recent release of Department of Defense 5000 [5], the government is trying to rectify the traditional bias toward waterfall-planned programs, but the waterfall process remains predominant. Typically, teams doing OO development today within a waterfall process spend their time creating, editing, and reviewing UML diagrams. Unfortunately, translating from UML diagrams into code is not automatic and often exposes major flaws in the design.

When designers are inexperienced in both OO programming and the underlying infrastructure (such as CORBA) the resulting designs are often misguided or simply infeasible to implement. A common design defect is attempting to manage classes with a large number of objects under tight timing considerations. The kind of schedule pressure created by the emphasis of a waterfall process on getting everything right the first time leads developers to opt for the *most expedient fix* in code, rather than rethinking the design.

Generally, programs, as reflected in their plans and practices, do not appreciate just how much good design depends on feedback from prototypes, or preferably, from early implementations of partial or simplified designs. As program schedules become more compressed, development teams are, in fact, coding and designing simultaneously. However, rather than plan for rapid design and code iterations in an open and well-managed fashion, they often attempt to mask this reality beneath a simplified, waterfall model being projected for the program and, consequently, create chaos rather than success.

Spare Modeling

Models can be a very powerful tool in working on a complex problem. Formal models (see [6]) can support automated checks for logical consistency and automated generation of effective test suites. Visual representations can contain rich semantics that are hard to convey in words alone. When used judiciously, they are of great value as an aid to communication among developers, especially when generated quickly, informally, and cheaply.

However, the idea that a system should be *completely* modeled using UML diagrams is of dubious value, if not outright harmful. The cost of developing and maintaining an extensive set of UML diagrams for a system far outweighs its benefit. The UML has been justifiably called a *cartoon* [7] in that it is not semantically sufficient to address all the nuances that must be communicated to a tester or coder. Development teams find it far too easy to expend indefinite effort on UML diagrams of arbitrary detail without any assurance that these diagrams connect to implementation reality.

Code, on the other hand, accompanied by tests, is a good model (an *executable model*) for clearly and unambiguously expressing system behavior in detail. The UML is much better used sparingly to capture only key classes and relationships, and critical execution paths [8].

Experienced OO Guides

If you have not climbed a mountain before, you should really bring a guide who knows the slopes, unless you lust for the thrill of danger. Similarly, a serious OO development effort should really have at least one, if not a few developers with extensive OO experience. If no expert is available, plan to iterate and redesign quite a lot (as discussed above) as your development team learns the ropes, or expect to churn indefinitely during integration and test, trying to patch a fragile, naïve design.

Not surprisingly, teams using OO design for the first time usually fall back on familiar, non-OO patterns. Each developer defines a large controlling class for the developer's area of responsibility that is all function (a *functoid*) and encapsulates no data. The data is thinly wrapped in other classes that contain only the data and access operations (*datoids*). This design is essentially functionally oriented, only superficially structured into classes and objects.

Conclusion

The recommendations in this article are not really new, but their descriptions here contrasted with typical pitfalls seen in government contracting may help you to better understand how to apply them to good effect.

OO design and evolutionary development fit well together. Many of the pitfalls discussed here are characteristic of programs practicing waterfall-style processes. Attaining the potential benefits of OO development is more difficult in a waterfall process. The introduction of OO to an organization should change both its development practices and the designs it produces for systems. If an organization makes only superficial changes (draws more diagrams or uses a different programming language) then what was the point of *changing to* OO development?◆

References

1. Software Productivity Consortium. "Object-Oriented Approach for Software-Intensive Systems (OOASIS)." SPC-2000001-MC. Herndon, VA: SPC, 2000 <www.software.org/membersonly/ooasis>.
2. Evans, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.
3. Cockburn, Alistair. Writing Effective Use Cases. Addison-Wesley Professional, 2000.
4. Foote, Brian and Joseph Yoder. "Big Ball of Mud." Fourth Conference on Patterns Languages of Programs, Monticello, IL, Sept. 1997 <www.laputan.org/mud/mud.html>.
5. U.S. Department of Defense. "DoD 5000 Series." Washington: DoD, 2003 <http://akss.dau.mil/darc/darc.html >.
6. Blackburn, Mark, Aaron Nauman, Bob Busser, and Bryan Stensvad. "Defect Identification with Model-Based Test Automation." Herndon, VA: Software Productivity Consortium, 2002 <www.software.org/pub/taf/downloads/SAE_2003.pdf>.
7. Binder, Robert. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional, 1999.
8. Ambler, Scott. Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process. John Wiley & Sons Canada, Ltd., 2002.

About the Authors

Rich McCabe is a principal member of the technical staff at the Systems and Software Consortium (formerly the Software Productivity Consortium). McCabe co-authored the consortium's Object-Oriented Approach to Software-Intensive Systems (OOASIS) methodology. He also has headed the consortium's pioneering work in the product-line approach for systematic reuse since its inception in the early 1990s. Outside the consortium, he has nearly 15 years of software and system development experience with Bell Laboratories and other firms.

Systems and Software Consortium
2214 Rock Hill RD
Herndon, VA 20170-4227
Phone: (703) 742-7289
Fax: (703) 742-7200
E-mail: mccabe@systemsandsoftware.org

Michael Polen is a senior member of the technical staff at the Systems and Software Consortium (formerly the Software Productivity Consortium). He co-authored the Consortium's Object-Oriented Approach to Software-Intensive Systems (OOASIS) methodology and consults with consortium members on their practice. Lately, Polen has been merging OOASIS with agile techniques. He has more than 14 years of software and system development experience with Motorola, Booz, Allen and Hamilton, and other firms.

Systems and Software Consortium
2214 Rock Hill RD
Herndon, VA 20170-4227
Phone: (703) 742-7281
Fax: (703) 742-7200
E-mail: polen@systemsandsoftware.org