From Domain Models to Architectures

Paul C. Clements
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

1 Requirements for an architecture

A software system can be evaluated against criteria in two broad categories:

- *functional and performance attributes*: how well does the system, during execution, satisfy its behavioral, functional, and performance requirements? Does it provide the required results? Does it provide them in a timely enough manner? Are the results correct, or within specified accuracy and stability tolerances?
- *non-functional attributes*: how easy is the system to integrate, test, and modify? How expensive was it to develop?

These two categories are orthogonal; systems that unfailingly meet all of their requirements may or may not have been prohibitively expensive to develop, and may or may not be impossible to modify. Highly modifiable systems may or may not produce correct results.

Given a set of requirements for a system, the developer must choose an architecture that will allow the implementation of the system to proceed in a straightforward manner, producing a product that meets its functional and non-functional requirements. How is that done?

1.1 Producing architectures to meet functional requirements

There is, unfortunately, no reliable automatic or semi-automatic technology that will produce a satisfactory architecture given a set of functional and performance requirements. Systems with inappropriate architectures (e.g., requiring too much communication among components) may not meet their performance requirements. In practice, the development team adopts an architecture from a successfully developed system having similar functionality and performance constraints. This adoption may be performed after a formal survey; more likely, the survey is informal and drawn solely upon the experience of the senior designers.

1.2 Producing architectures to meet non-functional requirements

Choosing a software architecture that accommodates change requires adopting a structure to manage a set of anticipated modifications; including new computing environments, new platform deployments, new operator interfaces, incorporation of new or modified environmental models, adoption of new devices, etc. The goal is to produce a system structure such that each anticipated change will only affect a small number of system components, or at worst affect several components but in a manner in which the large change can be staged as a series of small, independently-testable changes.

The list of changes that a wise development effort will account for comes from at least three sources:

1. Changes anticipated throughout the life cycle of the system. The list of anticipated changes is often produced by tapping the experience of more mature, already-deployed systems with similar purpose.
2. In systems that are procured by one party and developed by another (even if both parties are part of the same organization), there are often areas of requirements that are controversial or subject to differing interpretations. No matter how each individual issue is settled, these may indicate functionality that might not be met with initial deployments, but which may be required subsequently.
3. Decreases in functionality brought about by programmatic restructuring. If a hypothetical flight simulator is initially scaled back, for example, so that it can only handle a single type of aircraft, it would be foolish to design it without anticipating subsequent addition of other aircraft types.

1.3 Domain analysis

As just discussed, we anticipate changes by exploiting knowledge about the current application, and related or similar applications. We choose an architecture to accommodate those predicted changes. In addition, we choose an architecture to meet functional and performance requirements by exploiting familiarity with architectures of similar systems that produced correct and timely results.

Domain analysis is the name we give to the process by which this experience or familiarity is captured, structured, and then exploited. In particular, domain analysis is the process by which applicable architectures are mined that give us confidence in meeting our functional and performance requirements, and the process by which the list of anticipated changes is captured.

2 What is software architecture?

Before we can produce a software architecture for a system, it is incumbent upon us to define exactly what that means. We find that many authors tend to blur several concepts together, concepts which the principle of separation of concerns persuades us should be carefully managed. For instance, in a paper that is gaining importance, Garlan and Shaw [4] write:

> *As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall structure of the system emerges as a new kind of problem. These "structural issues include gross organization and global control structure; protocols for communication [among components], synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives. This is the* ***software architecture*** *level of design.*

However, what does 'overall structure" mean? We can think of many kinds of software struc-

tures: module, uses [8], data flow, control flow, class, calls, communicates-with, synchronizes-with, and many others. Each represents a distinct view of the system that abstracts away a class of details, leaving a structure that is useful for a particular kind of analysis or understanding. Each is a view of the system's architecture. Parnas [9] counsels us that when confronted with software that claims to be "hierarchically structured," it is incumbent upon us to ask (a) what do the nodes in the structure represent? and (b) what is the relationship that is implied when two nodes are connected? We extend that valuable lesson to software architecture diagrams of any sort. Without knowing precisely what the components are, what the links mean, and what significance there is to position of components and/or direction of links, diagrams are not much help, and may actually be quite misleading.

Consider a "layered architecture." Again, from [14]:

> *A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. In some layered systems inner layers are hidden from all except the adjacent outer layer, except for certain functions carefully selected for export. Thus in these systems the components implement a virtual machine at some layer in the hierarchy... The connectors are defined by the protocols that determine how the layers will interact.*

Close examination of this description reveals that separate concerns are being mixed. For one thing, "hidden" is a concept that has no meaning at run-time; it is purely a concept that applies at program-write time, and specifies what facilities a particular programmer is or is not allowed to use when writing his or her portion of the system. "Providing service" is the run-time interaction mechanism, but it could reasonably mean any of the following: calls, uses, signals, sends data to. It also fails to capture any notion of concurrency, real or potential. Can software in different layers run simultaneously, or are there mutual exclusion constraints between layers? If we are concerned about the feasibility of fielding our system on a multi-processor environment, shouldn't we be able to discern this information as part of the answer to the question "What is the architecture of a layered system?"

We conjecture that when a developer asks "What architecture shall I choose?" , he or she is seeking the answer to one or more of the following questions:

- What structure shall I employ to assign workers, to write my work breakdown, to exploit already-packaged components, and to plan for modification?
- What structure shall I employ so that the system, at run-time, fulfills its behavioral and performance requirements?

It is not obvious that the first question has a single answer, but in practice it does: a system's modular structure is used to coordinate the allocation of project resources. The modular structure of a system is a purely static concept; when the code from each module is compiled and linked, the modular structure vanishes. For this reason, we will call this form of architecture a system's *static architecture*.

The second question refers to a structure that survives through execution; we call such a

structure part of the system's *dynamic architecture*. Garlan and Shaw [4] enumerate a list of common patterns of dynamic architecture, such as "pipes and filters", "blackboard architecture", "client-server," and so forth. All of these refer to ways in which components that exist at run-time interact with each other in various ways. It is less clear that the second question has a single answer, as we have seen.

Which notion of architecture is the right one? Obviously they both are. It is an axiom of this paper that assuming the two structures are the same is a fundamental design mistake, since the two structures are optimized to meet completely different criteria.

We conclude that "architecture" is not "the structure" of a system, but is *a collection of one or more useful structural views of the same system* and we will use that meaning in this paper. It is not, therefore, a particularly meaningful term *in isolation*. We urge readers to understand that the concept is ambiguous; in the spirit of Parnas, we further urge that when confronted with it, one should inquire exactly to which structure(s) of software the speaker is referring.

3 Deriving static architectures from domain models

After domain analysis has been performed to define the program family of interest, we assume that the following products are available:

- a definition of the scope of the program family; put another way, a definition of the boundary between system and environment of each member of the family;
- the information that flows between each member of the family and its environment and the semantics of that information; and,
- the set of features offered by each member of the family,
- the behavioral and qualitative requirements to be met by each member of the family.

What we seek to produce is an architecture for the family, not just the single member of that family that may be currently under development. An architecture for the family is sufficient to implement any member of the family. It may include pieces (work assignments, modules, etc.) that will only be included in certain members of the family. Put another way, it is possible that the instantiation of certain pieces, in order to attain a target member of the family, will map to the empty program.

3.1 Using the scope and flow of information

The domain analysis capture of scope and trans-environment information flow suggests a set of objects whose secret is the concrete details of the interface between a member of the program family and its environment. The scope and information flow information are used to craft objects that encapsulate the details of the interaction between the system and its environment.[1]

1. In SPC's CoRE method, these objects are known as "boundary classes" [11]. In the Software Cost Reduction project architecture, they were known as "Hardware-Hiding modules" [2][1], but later refinement suggested the more general "Environment-Hiding modules".

Of course, a correct domain model for this application will have anticipated and captured the differences between these two versions of the system; nevertheless, the variation could complicate the way in which we use the scope information to assign responsibilities to the components in our derived architecture.

Information-hiding dictates that if a likely difference between devices is that one device may produce unrefined data while another produces refined data, then that difference should be encapsulated by an object. The design of the object is such that other objects should be unaffected by the change. In this case, it means that the object should return position directly; the secret (whether the position is computed by software or returned directly by the device) is protected.[2]

The flow of information outward from the system can also be used to craft a set of objects, one for each output value, whose secrets are the rules for how and when to calculate the values of the assigned output. Since the only visible behavior of the system is via its output values, these objects may be thought of as driving the system. They will make use of other system resources in order to produce the correct output values at the correct times. In practice, these objects are implemented as processes.

3.2 Using the feature catalog and behavioral requirements

The behavioral requirements will provide the rules that each value-producing object (outlined in the previous paragraph) must follow in order to perform its function. Rules that are common to more than one output, or values that must be computed in the course of more than one output, may be relegated to value-producing objects that encapsulate the shared rule or calculation. They constitute the Domain Utilities Layer in [6].[3]

3.3 Designing other objects

Certain types of differences among family members may not be captured by Feature-Oriented Domain ANalysis (FODA) or similar domain analysis methods. The scope and information flow deals with the system and its environment. The features in the feature catalog, by definition, deal with aspects of the system that “directly affect end users” and are thus externally visible. The information model captures some of the overall functionality of the system. None of these system views is relevant to certain internal aspects that may differentiate members of the family.

Examples of these aspects include:

---

2. Of course, another changeable aspect might be the method by which the position is calculated from velocities; this could be a difference across those family members that use the more primitive device. In this case, information-hiding also calls for an object to encapsulate the calculation method, as long as the calculation method is not tied exclusively to the type of device present.

3. These value-producing objects are simply called *objects* in the Object Connection Architecture (OCA) paradigm [10]. They constitute “Shared Services modules” or “Software Utility modules” in SCR [2].

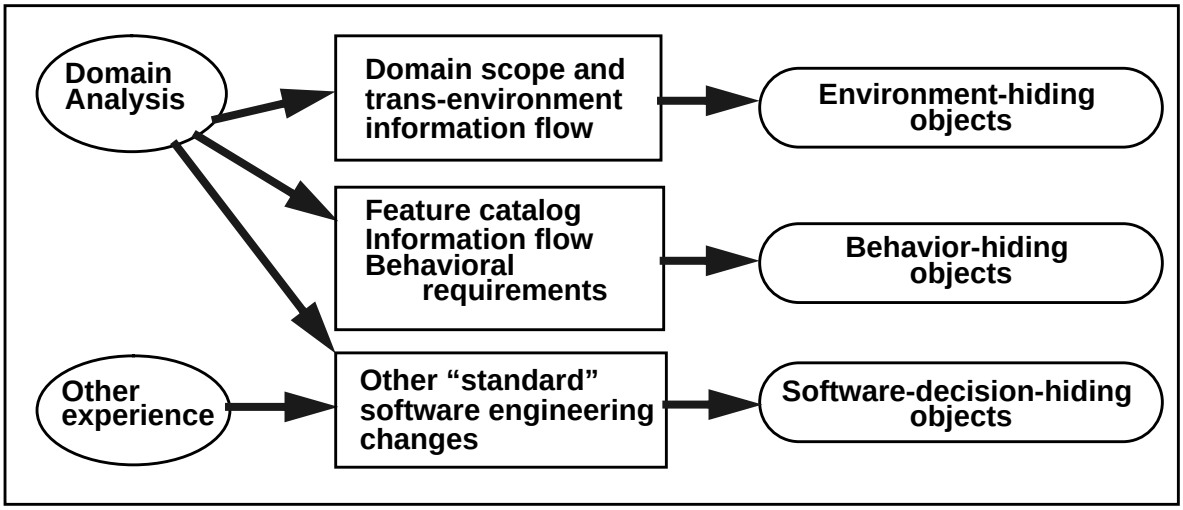
- specific algorithms to perform calculations[4];
- choice of representation of data via abstract data types;
- organization and layout of system internal data stores.

The choice of these aspects, each of which may be chosen to increase efficiency or to make use of previous analysis, documentation, or coding, are independent of the externally-visible behavior of the system; each, however represents a viable difference between members of the program family. As such, their binding needs to be encapsulated in objects.[5] Those which can be imported from other domains would qualify as Common Utilities [6].

3.4 Producing a system

The goal of this object-oriented approach to architecture from domain analysis is to engineer a set of objects such that

- The objects, taken as a set, satisfy the required functionality and qualitative requirements for every member of the domain (although some domain members may only need possibly empty subsets of some objects to be present).
- The interface to the object remains unchanged across all member of the domain (except that subsetting is permitted).
- A particular member of the domain is realized by specifying which facilities it requires to be present on the interface of each object, and by assigning one of several possible implementations toeach required facility.



4. We distinguish between algorithms to perform calculations and the equations that define those calculations. The latter may very well be required. For a simple example, a calculation to double an integer value may be implemented by multiplying by 2 or adding the value to itself. Doubling is the requirement; the choice of how to carry it out is a changeable secret that should be encapsulated in a module. This is a trivial example that encapsulates a trivially changeable secret and thus yields a trivial module, but more complicated equations -- and algorithms that implement them in one of a number of ways -- are common.

5. These are the "Software Decision-Hiding modules" in the SCR architecture [2].

4 Conclusions

Domain analysis is the process by which candidate architectures (particularly dynamic architectures) are selected for a program family. It is also the process by which life-cycle modifications to a system may be anticipated, and by which the members of the product family may be enumerated. As such, it is an invaluable first step to producing an architecture for a system.

"Architecture" refers to the collection of useful structural views of a system; some views only exist at design time, while others persist through execution. The products of domain analysis map straightforwardly onto static architectural views such as the modular view.

A companion paper, in preparation, will discuss how domain analysis can lead to the adoption of the dynamic architecture to support a program family.

5 References

1. Britton; Parker, and Parnas; "A Procedure for Designing Abstract Interfaces for Device Interface Modules",. *Proc. ICSE5*, pp. 195-204, 1981.
2. Britton, Parnas, *A-7E Software Module Guide*, NRL report, 1980.
3. Clements, "Software Cost Reduction through Disciplined Design", *1984 Naval Research Laboratory Review*, available as National Technical Information Service order number AD-A159000, pp. 79-87, July 1985.
4. Garlan and Shaw, "An Introduction to Software Architecture," in *Advances in Software Engineering and Knowledge Engineering*, vol. I, World Scientific Publishing Company,1993.
5. Hayes-Roth, "Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program," 14 January 1994.
6. Kang, Cohen, Hess, Novak, and Peterson; *Feature-Oriented Domain Analysis Feasibility Study: Interim Report*; technical report CMU/SEI-90-TR-21 ESD-90-TR-222, August 1990.
7. Kazman, Bass, Abowd, and Webb; "SAAM: A Method for Analyzing the Properties of Software Architectures," *Proc. ICSE16*, 1994.
8. Parnas, D.; "Designing Software for Ease of Extension and Contraction," *IEEE Trans. Software Engineering*, vol. SE-5, no. 2, pp. 128-137, 1979.
9. Parnas, "On a 'Buzzword': Hierarchical Structure," *Proc. IFIP Congress 74*, pp. 336-3390, 1974.
10. Peterson and Stanley; *Mapping a Domain Model and Architecture to a Generic Design*, technical report CMU/SEI-93-TR-{tbd}, December 1993.
11. Software Productivity Consortium, *Consortium Requirements Engineering Guidebook*, SPC-92060-CMC, December 1993.