Affordable Software Architecture

Stephen J. Mellor

StephenMellor@StephenMellor.com

www.StephenMellor.com

Abstract

A successful architecture must appear to have been constructed by a single mind, even though we know that many minds are actually involved. An effective approach is to build the architecture as a set of rules that are independent of the application. Because such an architecture is independent of the size of the application it is more affordable, especially for large systems.

This paper describes how an application-independent architecture provides coherent rationale for architectural decisions and when to apply them, consistency across the application depending on circumstances, opportunities for automation, and—because the same decisions are not being taken (slightly differently) repeatedly across the application—a significantly more affordable and easy-to-test implementation.

1. The Fractal Fencing Frenzy

Last century in a discussion on the Advisory Board to *IEEE Software*, Terry Bollinger of Mitre Corp. sent this (slightly edited) email[1]. "If only small groups can really design software well, what are those really big groups doing to keep so busy? That's an interesting question in its own right, and it has to do in part with the self-perpetuating nature of complexity.

Imagine for a moment that you have been assigned the task of building a corral to keep in a herd of Chaos Cattle. Chaos Cattle are troublesome beasts that like to roam all around when given the slightest opportunity to do so. They also bear an unpleasant resemblance to the kinds of problems that software developers face every day: requirements that either don't exist or keep changing so rapidly that you never really know what they look like.

You have a choice of letting the Chaos Cattle run free while you sit down and design a really good corral, or of starting work immediately with the one hundred fence builders assigned to you. Naturally, since this is an example of what happens when you choose to use lots of people, you choose to tell all one hundred of your fence builders to get to it!

[1] Used with permission.

You give them a general outline of where to build the corral and where each one of them should position themselves, but that's about it, as you don't really have anything more that you can tell them. You know that they are a smart bunch, so you just hope for the best.

And indeed, they *are* a smart and creative bunch. In fact, they are so creative that each one begins to design their own version of a fence. One bright young lady develops a low-cost design using earth with a few wooden supports. Another one decides to go with an entirely wooden design that relies mostly on vertical posts. Another one chooses an all-wood design also, but relies more on horizontal boards with only occasional vertical posts. Yet another chooses heavy-duty wire mesh. One designer (male, incidentally) who plays too much Quake III Arena[2] on the side decides to go with state-of-the art razor wire with flamethrower backup and a flashing red light to give the cattle a chance of not being converted abruptly into roast beef sandwiches. The list of novel solutions goes on ...

And because they really are all good designers, *every one* of the resulting fence segments proves to be 100% effective at keeping Chaos Cattle from getting though. There is just one problem… which is that none of the fence segments can be easily linked together to create a single, unified corral. Instead of being stopped by the fences, the cattle imply mosey through the many gaps between fences that don't link up. You're, ah, “90% done with the coding” but still pretty much 100% away from actually keeping the cattle in. But no problem! You've still got 100 fence builders available, and about 100 holes that still need to be plugged. You request a schedule extension, announce Phase II of the project (“Pre-Release Enhancements”), and put the designers to work on the gaps.

Alas, you quickly discover that far from being simpler, the new set of problems is actually more complex than the original set. Back in Phase I, all you had to do was build fences, without any constraints other than the need to keep cattle in. Your fence builders had free reign, and were able to work creatively and highly effectively. In Phase II, the builders must now understand each other's fences and find ways to link them together.

Some of these “integration tasks” prove to be straightforward, such as the case where the vertical post wooden fence builder happened to be adjacent to the horizontal board builder, in which case they added vertical posts until they reached the end of the boards. Others, such as tying the earthen fence to the mesh one, proved a lot more difficult. And in the case of the Quake III Arena player's fence, no one even wanted to go near it!

So after a Phase II that was every bit as long as Phase I (actually, 50% longer), you still don't have a cattle-resistant corral. To be frank, what you have is a non-trivial number of gaping holes. But it's time to publish or perish, so to speak, so the corral becomes “operational”—and the cows immediately start leaving through the remaining gaps.

So you set up emergency response fence building teams to create new “mini-corral” patches to capture any cattle that get through. But guess what: All of your fence builders are still convinced that *their* designs were really the best, and they immediately start to slip back into old habits. The mini-corral patches rather quickly become as complex in

[2] I told you it was last century!

design as the original leaky corral, as various builders add their own “innovations” on different days and in different ways. The patches themselves consequently often leak too, and yet more "mini-mini-corrals" have to be added to take care of those.

At some point you realize that what you've actually created is a Fractal Fencing Frenzy. Due to a lack of consistent communication of intent, each segment of the fence seems to exhibit an almost unlimited ability to spawn smaller and smaller problems that must each be addressed by the (costly) attention of an individual fence builder.

You know you are in trouble when your boss asks for an all-day status report on Tuesday, and you find yourself genuinely tempted on Monday to see how many of your own bones you can break by “accidentally” skiing into a tree.

Eventually, though, you convince your boss that what is going on is just the normal sequence of events in software, and that you have simply entered the “maintenance” phase, which as all fence builders know is the most expensive phase... even though that's not generally true in any other engineering field.

And finally, your Fractal Fence just keep fracturing, since of course there is always an unending stream of requests from the disgruntled users to add more Chaos Cattle. Every time you extend the fence, you end up spending more time fixing the parts that you break than actually building more fence.

Avoiding The Fractal Fencing Frenzy

In another universe was the woman who did this: She designed a single, universal design for the fence first. She did this in a very interactive, highly creative fashion, with the help both of a couple of friends used to dealing with this particular local breed of Chaos Cattle, and with another fence designer who checked out her design and kept asking her annoying but important design questions.

Once her design was complete, she implemented it using only builders who fully and exactly understood the constraints involved, and *why* those constraints were needed (e.g., to make sure the fence could be modified easily). Since her original intensive design work focused every bit or more on how to keep the fence usable and adaptable over time as it did to keeping the cattle in, the implemented fence never entered into the Fractal Frenzy as needs changed over time. Instead, simple changes in capacities and expected needs resulted in comparably simple changes, often little more than simply moving (reconfiguring) the fence.

And cost-wise, the only ones who got milked were... the cows.”

In other words, she constructed an *architecture*.

2. Software Architecture

The abstract organization of the software in a system is called the *software architecture*. It proclaims and enforces system-wide rules for the organization of:

- data,
- control,
- structures, and
- time

Data. The software architect decides how to organize data in the system. Consider, for example, the data that describes a chemical plant. There will be data describing the recipes by which we manufacture the chemicals, data to describe valves, pumps, tanks and so on, data to describe heating and cooling procedures, and the like. This data could be organized by rows—one 'row' per recipe, tank, valve etc. Or it could be organized by column, whereby the desired temperatures and pressures of all the tanks are organized into columns for fast iterative access during a control cycle. Or the data could be organized into a tabular structure or even no structure at all—simply free floating data elements. Other data in the system, say for histogramming, may require special purpose data structures, such as trees, bins, or linked lists.

It is possible, of course, to use several schemes, perhaps a column-wise organization for variables that require fast iterative control, but a row-wise organization for devices that are controlled individually. The issue here is to find the minimum set of data organization techniques so as to reduce the cost of building, learning about, and maintaining data access logic, and to make the code for the system smaller.

Similarly, the architect must decide how to access data. Should it be directly by name, or using a function that encapsulates the data structure, thus protecting its clients from any change in its organization? For data that requires fast access another possibility is to employ a facsimile of encapsulation at system construction time. In this approach, the software developer writes logic as if the data were encapsulated. A preprocessing tool or perhaps the compiler in languages such as Eiffel, then transforms the 'function' into a direct memory access. This is the data-access analog of an inline function in C and C++.

Control. The architect must decide what causes a task to execute. Some tasks may execute periodically, while others execute on demand. A task may execute for a single instance—say, to control a single valve—or it may execute for many instances—say, to control the temperatures of all tanks currently in a 'Heating' state.

The architect must decide on what basis a task gives up control of the processor, and how this relates to priorities.

Of critical importance is how access to shared data can be controlled. The problem is to ensure that, whenever a task runs, it accesses the correct version of data. There are many approaches to the problem: data locks, sequentializing tasks; permitting parallel execution of tasks if they access different data sets and so on.

Structures. We define a *structure* to be any packaging of code or data in a system. Examples are tasks, classes, objects, functions, and shared data areas. It is the responsibility of the architect to select the structures to be used in the system and to decide on the basis for allocating portions of the application to these structures.

Time. The architecture must also provide for both absolute and relative time.

This collection of architectural decisions, i.e. the abstract organization of the software, is an *application-independent software architecture*. Note that several authors [1, 2, 3] do not require "application independence." Exercise caution in this area when reading their work.

3. Executable Models

If we can build an application-independent software architecture, then presumably we can also build architecture-independent applications. To be useful it should be executable.

One form of executable modeling is based on a limited subset of UML assembled in a particular way so that the diagrams have meaning. For example, in UML one may build a state model for a class, an object, a method, a use case, a component, a package—just about anything. While the execution semantics of the state model are defined in isolation, this leaves open the question of how the various elements relate to one another.

Executable UML [4] comprises only three diagrams, the class diagram, which outlines the conceptual entities in the domain, a state model for each class that models the lifecycle of each object, and a set of actions that establish the state, called an activity.

The classes must be abstracted based on both similar behavior and characteristics. This requirement allows a state model to be built for each class that applies uniformly to each object, avoiding logic to determine what type of object each action is executing on.

The state model captures the lifecycle of each object. There is one state model per class because the behavior of each object is exactly the same. In some cases, there is no need to build a state model because the only operations are synchronous data access under the control of other objects. For other classes that manage contention, we build a single state model for the class as a whole. Conceptually, then, a class may have no state models, or one model that applies uniformly to each object, and an (optional) other that apples to the class as a whole.

An activity is a collection of actions that is executed on entry to a state. An activity is almost the same as an action, except that an activity allows for parameters, while actions rely on data flows. Specifically, when an event is received by a state model, the event may carry data other than the target object, such as the phone numbers involved in a call. This "supplemental data" is made available to all actions in the activity on data flows.

Activities on state models do not have return parameters. All data must (logically) be stored as attributes of objects, before the activity completes.
An executable model operates on data about objects. Each class defines attributes for objects, and each attribute may have an initial value for each object. When the model executes, each object has its own state, and each object executes concurrently and asynchronously with respect to all others. Each state model instance (a state machine), then, is in its own (just one) state, either executing an activity to completion, or not. The actions inside activities may read data through (logically) encapsulated functions of other classes at the same time as the target object is executing an activity. It is the modeler's responsibility to avoid data access conflict by synchronizing the behavior of the various objects, if required, using a state model.

The state models recognize events. Each event may be a signal sent from another object, from the outside, or it can be the result of a "deferred event" that signals expiry of a timer or of some absolute time.

Thus, an executable UML "program" comprises a collection of concurrently executing state machines communicating by sending signals. This highly concurrent model can re reorganized to be fully synchronous or distributed without changing its behavior. Indeed, it is possible to reorganize the model so that some states are executed periodically in a separate task, and the attributes of a single object are stored in multiple processors. Just so long as the behavior is the same.

4. Translating Executable Models

A *model compiler* comprises a collection of reusable components together with a set of formalized rules on how to embed the application within it. The reusable components should include ways to access program data such as lists and trees, to store persistent data, to sequence actions, to activate and deactivate tasks and so on. The formalized rules show the usage of the reusable components in context. For example, if a task synchronizes with others by sending messages of a particular form, then the formalized rule shows the construction of the message, and the call to the message handling service. The content of the message is left as a placeholder to be filled in from the application.

We base the concept of a model compiler on the observation that to understand the fundamental architecture of a system, we need only to understand the structure of each of its prototypical elements and how they interrelate. For example, to understand the design of a Unix-based application, we need only understand pipes and filters. We don't need to understand the details of the semantics of an application–in fact they are a hindrance.

The details of model compilation are beyond the scope of this paper, but Figure 1 summarizes them. The meaning of the application is modeled and stored in a repository, as shown at the top of the diagram. This meaning includes all the concepts described in the sections above: class, state, action, and so on. (The graphical layout is also stored, but for model translation this is not relevant.)

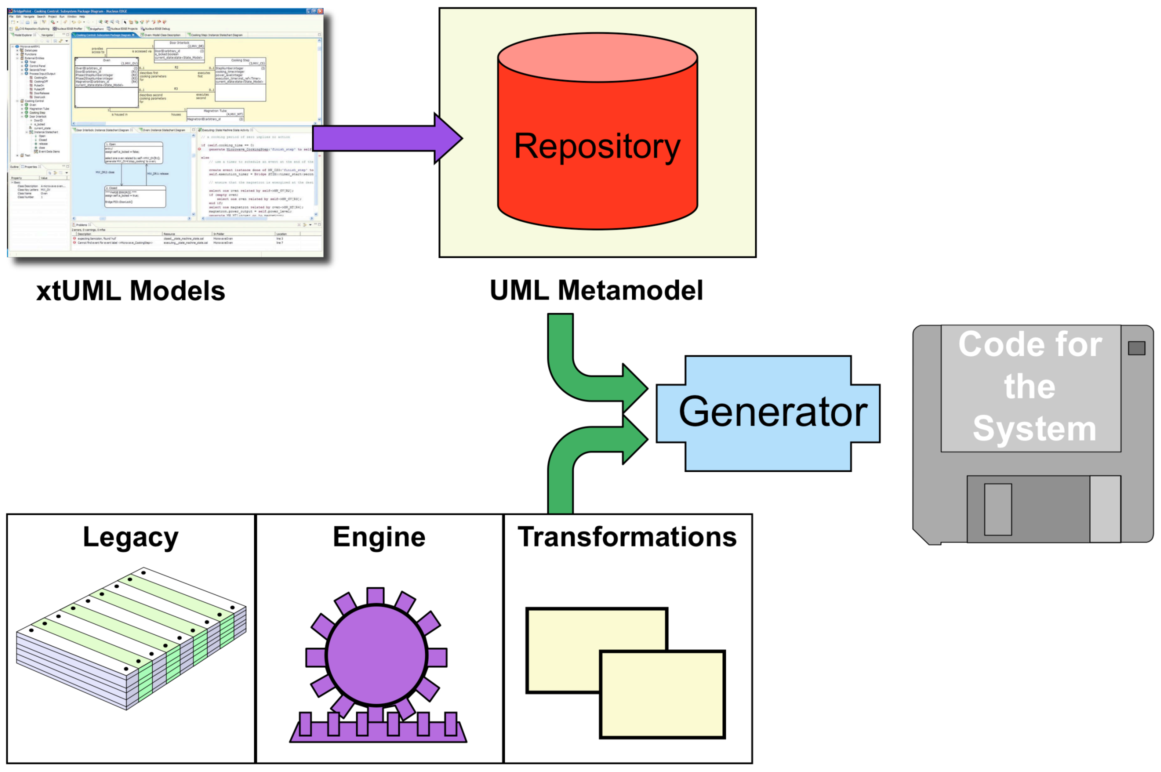

Figure 1: The structure of Model Compilation

The model compiler is at the bottom of the diagram. It consists of a library of reusable elements, such as list or memory management functions, and a set of rules. Each rule generates text. For example, a class may have a rule that creates a `class`, a `struct` or even a VHDL `entity`. The rule language has been standardized by the Object Management Group, and can be found in [5].

The result of applying the rule (the Generator in the center of the diagram) is text. This text can be compiled with the libraries in the model compiler, other libraries, legacy code, etc. The result is a program, or set of programs, that can be compiled, linked, and run.

There is little value, therefore, in thinking of a design as the repetitive use of the design decisions as they relate to the application components. This is time-consuming, tedious, and an invitation to unnecessary and unwanted creativity. Instead, we should think of design as the definition of the reusable components and how to use them in the context of an application. Note that this approach scales well, so that a project can make best use of experienced designers, because the model compiler contains only information about the system design. The approach is also platform independent, meaning that a model expressed in this manner can be translated onto any platform. All you need is its model compiler.

5. Affordablity

There are a number of model compilers available and many of them are suitable for embedded systems. For example, one model compiler that generates very fast, small-footprint C can be had for under $60K. This costs less than half an effort-year.

Your choice is between taking $60K out of the capital budget or dribbling it out $1K a month per person over a period of years. If you choose the latter, you have to specify the architecture.

6. Specifying the Architecture

To specify the software architecture, begin with a system sketch, a brief description of the theory of operation for the architecture, and the rationales for decisions you have made. This is then supported by a description of the conceptual entities of the architecture and mapping rules from the application to those entities.

System Sketch. Sketch out the layout of the system to show:

- processors
- communication channels
- bandwidth
- protocols
- external actors
- etc.

Although such a sketch is informal, it nonetheless provides a reference model for the system that is readily understandable by client, architect and "newbie" alike.

Theory of Operation. This short document describes how the system works a whole. The document should describe the threads of control that run through the architecture, and cover all modes of system operation, including normal system operation; cold-start and initialization procedures; warm-start, restart, and failover operation, as required; and shutdown. The goal is to provide the "big picture" for how the architecture works.

Rationale. Decisions are made for (generally) good reasons. However, reasons change, so it is helpful to write down both the non-localized requirements and the constraints *together with their rationales*. Then, when the rationales change, it is a relatively easier job to find those decisions that require re-examination.

Conceptual entities. Next, we must capture the conceptual entities that make up the architecture. If there is a task that runs periodically, we should declare and define this "periodic task" describing what it is and how it fits into the rest of the architecture. Or there may be different types of processors, and certain functionality is assigned to each kind. Each of the roles taken on by processors should be defined. For another example, in object-oriented architecture, there are classes, inheritance structures and so on. It is important to outline the basis for abstraction of the classes. Are they based on real-world entities? Or more software-oriented concepts surrounding data or control organization?

Mapping Rules. Finally, we must define mapping rules for elements of the application onto the conceptual entities we have defined for the architecture. For example, some functionality will be allocated to the periodic task. Presumably, this will be functionality that is executed periodically. How should a programmer looking at an application model recognize this functionality? How, exactly, is it a programmer to code that functionality? These mapping rules, taken together, act as translation program from the models to code.

7. Summary

An affordable software architecture can be had if we recognize that an architecture is a set of inter-related decisions that must fit together into a coherent whole, rather than a set of individual decisions made independently from one another.

To define an architecture, we must specify the conceptual entities that make it up and define a set of mapping rules from a well-defined (that is, we have a complete, executable specification) application into that architecture. Writing the code is then a matter of carrying out these mappings.

Because the task is repetitive, it can be automated. Model compilers exist for embedded systems, and can be purchased for less than the cost of hiring programmers, except on the very smallest of systems.

References

[1] *Software Architecture: Perspectives on an Emerging Discipline*, D. Garlan and M. Shaw, Prentice Hall, Englewood Cliffs, N.J., 1996.

[2] "Comparing Architectural Design Styles", *IEEE Software*, Nov. 1995, pp. 27-41.

[3] *Pattern-Oriented Software Architecture: A System of Patterns*, F. Buschmann et al., John Wiley, Chichester, UK, 1996.

[4] Stephen, J. Mellor, Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*

[5] *MOF Model to Text Transformation Language*, www.omg.org/spec/MOFM2T/1.0