Jim Coplien:

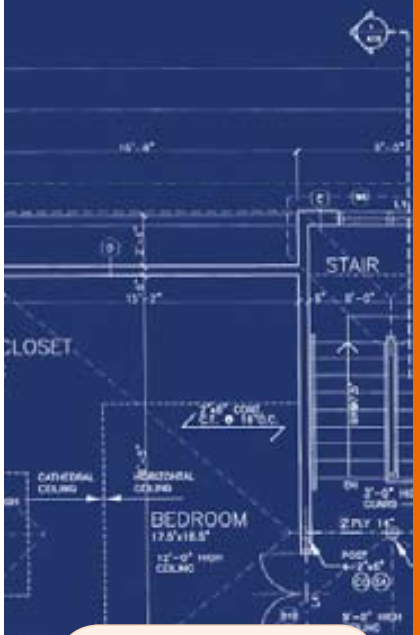
LEAN & AGILE AND THE MATTER OF ARCHITECTURE

By Jim Coplien, Gertrude & Cope

Software Architecture was often neglected in the early years of the agile movement. However in recent years most developers have learnt to appreciate its importance. In this article, Jim Coplien – the author of Wiley's upcoming book "Lean Software Architecture" (see page 22) – gives an overview of architecture's role in the Lean and Agile movements, and tells us about new interesting concepts that are emerging.

The pendulum of change

Mary Poppendieck described in a 2008 talk at Øresund Agile how the pendulums of practice swing back and forth over the years. I've seen this in my 40 years in the industry, and software architecture has always been one of these pendulums. I can't quite find the metaphor that fine-tunes Mary's vision to describe how the pendulum slams from one opinion to the other, and back again. The metaphor should invoke a vision of moving deliberately through levels of learning. Perhaps our entire industry is Agile at its very foundations, reacting eagerly to changes it induces itself. Instead, perhaps we should »



Jim Coplien

Jim ("Cope") Coplien (Gertrud&Cope, Denmark), Ph.D., CST, CSM, CSP, is the father of Organizational Patterns, is a co-founder of the Software Pattern discipline, a pioneer in practical object-oriented design, and a widely consulted authority, author, and trainer in software design and organizational development.

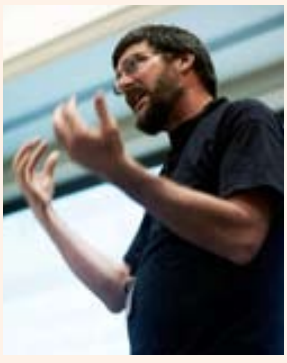


Photo: Christopher Roos

» be responding responsibly to the changes in our environment.

Changing fashions of software architecture

Software architecture made it into the software vernacular after a talk between Jerry Weinberg and Fred Brooks at IBM where Jerry encouraged Fred to follow through with his metaphor. The pendulum had a firm beginning at center-right. Architecture stayed in vogue for large projects until the early 1990s when it became unfashionable. Why? It had started to become overdone: sometimes a detached exercise for its own sake, driven out of fear of uncertainty and change, creeping ever more strongly to the right. Then, the pendulum slammed to the left. But ignoring architecture also proved problematic, and now the pendulum is moving back the other way. Bob Martin says, 'One of the more persistent myths of agile development is that upfront architecture and design are bad … Pardon me, but that's Horse Sh--.'

What is lean software architecture?

Lean architecture comes from applying the principles of the Toyota Production System to software architecture. "Lean" means to get rid of waste (like unnecessary documentation), inconsistency (like mismatched interfaces), and irregularity spaced development work in production. Lean understands that you do deliberate analysis and planning before going into production, using techniques like set-based design that explore every viable alternative. The word "Lean" applies to both the assembly line and to the car being built, but also describes the processes behind them. Lean is both about the thing and the process, reminiscent of what good generative patterns are. Lean architecture is both about an architecture with no fat, and about the consistency and reduction of waste in the process surrounding its creation and use.

A place for everything

Lean means discipline in maintenance, too. Yes, the Toyota Way goes beyond just the Toyota Production System (TPS) into Total Production Maintenance (TPM). One common aspect of TPS and TPM is that everything has its place. In TPS, there is a technique called poka-yoke or "fool-proofing" that ensures that pieces are put together correctly. It is like the concept of a design jig in craftsmanship. In software, architectural partitioning and interfaces guide feature programmers to the code for a specific domain, clarifying the code's place in the context of the entire system. In TPM, the tool board has tool outlines for each service tool, so each has its place. These organizations, relationships, and loci are carefully planned up front.

Forming the shape of the system

In Lean software architecture, we use Domain-Driven Design (DDD) to come up with the system form. The result of this process is the shape of the system. In the same sense that the essence of a Toyota steering wheel is captured in the plastic injection mould used to build it, so the essence of the system is captured in its architecture. We can tailor the steering wheel in many ways, just as we can tailor an abstract base class with many derived classes suitable to their respective markets.

Lean architecture delivers APIs: usually abstract base classes, with argument declarations and other code annotations that describe the relationships between them. It doesn't include details of data structure or method definition. It is architecture in the true historic sense of the word as a kind of pure form that delays structure.

The structure, we deliver just-in-time, prompted by the need to support a use case. This just-in-time notion is another key Lean tenet.

Is a lean architecture agile?

Now back to the other buzzword: Agile. Should software be both Lean and Agile? If we look at the words carefully, Lean applies to the system form and how it relates to the domain structure of the business and of technology. It is a complicated structure created by a complicated process. However, it needn't be complex. If something is complicated, I can take it apart and put it back together again, as an auto mechanic can do with a car. Complex things, on the other hand, are more than the sum of their parts. Software is both complicated and complex. Most of software's complexity comes not from form, but from the domain of time and software behavior. Use cases are what make software complex, partly because of the high rate of change within the system during a use case, and partly because a human being is usually involved. We tend to be complex creatures.

The architect Stewart Brandt notes that architectures have shear layers: layers of different rates of change in a house. The stone foundations or load-bearing walls may be modified once a century. Other walls that serve to partition space may come and go on the scale of a few decades. Windows and doors may come and go every decade or so; the carpeting a bit more frequently, and the internal décor on the scale of the seasons. A good software system has a Lean architecture that captures the rather stable complexity of its application and solution domains, and the complex mapping between then. On top of that is the shear layer of features that respond day-by-day or month-by-month to customer requests.

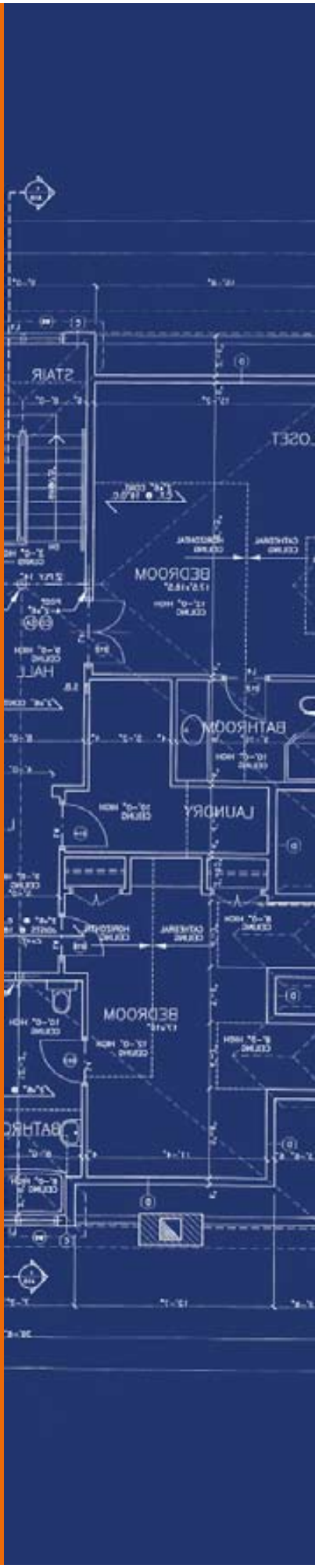
Therefore, Lean architecture has another side, which is its Agile application. In the same sense that a Toyota engineer develops a car so you can drive it through a complex race course in dynamic driving conditions, so a Lean architecture supports Agile adaptation of the system to the market.

Individuals and interaction, and usable code

Agile is about change. But Agile is also about individuals and interactions, and about software that works. A software system that works integrates seamlessly with the people who use it. That means that its structure should correspond to the mental model of the end users. End users interact with systems on the basis of their mental model of the objects on the other side of the screen. If the program objects don't map those in the end user's head, confusion results – and that violates the Agile provision for interactions with individuals. The programmer is also an individual, one who wants to separate the slow-changing foundations from DDD from the rapidly changing Use Cases. But the end user doesn't have this dichotomy! How do we resolve this?

Agile and Lean require new kinds of building blocks

The answer lies in the difference between classes, objects, and roles. Classes are units of source code and what the programmer writes. Objects are the units of program execution, and are part of the end-user cognitive model. Roles are the units of end-user model of action: a user understands an object in terms of the roles that it plays rather than in terms of the object itself. If we investigate the use case for a money transfer between accounts we will encounter roles like Source Account and Destination Account. Those aren't objects – my Salary



DCI

DCI (Data-Context-Interaction) is a so called Software Pattern for object oriented programming which has been developed by Trygve Reenskaug in recent years. It advocates a dynamic behavior of the software objects built on its role in each particular context, and de-emphasizes the ties between data and behavior which is central to the currently most popular pattern MVC (Model-View-Controller), developed by Reenskaug at Xerox PARC in the late seventies.

» Account is one of my accounts and my Savings Account is another, and either one may play either of roles Source Account or Destination Account. We want programmers to be able to deal with these separately because they change at different rates for different reasons, but we want the object that reflects the end-user model to exhibit all the behaviors of this role it is playing. In programming terms, that means being able to glue together the domain class and the role into a single class whose objects meet end user expectations.

The new building blocks: the DCI Architecture

Trygve Reenskaug's DCI architecture (Data, Context, and Interaction) is a way to organize this role-to-object mapping while properly balancing the concerns of the end users with those of the programmers. DCI starts with groupings of business functionality called Contexts (the "C" in DCI). A Context roughly corresponds to a use case, and a new Context object is instantiated at the start of each scenario. Contexts get their work done through Interactions (the "I" in DCI) between roles. An algorithm is a series of actions, where each action applies to some role like a Source Account or Destination Account. We can code up complete, generic algorithms in terms of methods on these roles. At the beginning of each scenario, the Context injects its roles into domain objects that do the work. These domain objects (the Data – the "D" in DCI) are the Models of MVC, or the basic building blocks from DDD.

The Context directly captures the scenarios of the end user mental model in terms of the roles by which end users conceptualize them, supporting the Agile agenda of customer collaboration. Programmers can reason about these algorithms directly, rather than hoping that the right behavior will emerge as a consequence of the interactions between objects. That supports the Agile agenda of working code– and goes further to usable code based on the end user model rather than on software engineering formalisms. The domain objects capture long-term stable system form that help the programmer contain change in the long term, supporting the Agile agenda of responding to change. De-coupling the algorithms from the data further supports responding to change. Furthermore, DCI reaches deep into such Lean principles as overall consistency, reduction of documentation, and just-in-time delivery. ■

Photo: clipart.com