

The Official Agile Modeling (AM) Site



Essay

Agile Architectural Modeling

by [Scott W. Ambler](#), Copyright 2001-2004

Architecture provides the foundation from which systems are built and an architectural model defines the vision on which your architecture is based. The scope of architecture can be that of a single application, of a family of applications, for an organization, or for an infrastructure such as the Internet that is shared by many organizations. Regardless of the scope, my experience is that you can take an agile approach to the modeling, development, and evolution of an architecture.

First and foremost, there is nothing special about architecture. Heresy you say! Absolutely not. Agile Modeling's value of *humility* states that everyone has equal value on a project, therefore anyone in the role of architect and their efforts are just as important but no more so than the efforts of everyone else. Yes, good architects have a specialized skillset appropriate to the task at hand and should have the experience to apply those skills effectively. The exact same thing can be said, however, of good developers, of good coaches, of good senior managers, and so on. Humility is an important success factor for your architecture efforts because it is what you need to avoid the development of an ivory tower architecture and to avoid the animosity of your teammates. The role of architect is valid for most projects, it just shouldn't be a role that is fulfilled by someone atop a pedestal.

You should beware ivory tower architectures. An ivory tower architecture is one that is often developed by an architect or architectural team in relative isolation to the day-to-day development activities of your project team(s). The mighty architectural guru(s) go off and develop one or more models describing the architecture that the minions on your team is to build to for the architect(s) know best. Ivory tower architectures are often beautiful things, usually well-documented with lots of fancy diagrams and wonderful vision statements proclaiming them to be your salvation. In theory, which is typically what your architect(s) bases their work on, ivory tower architectures work perfectly. However, experience shows that ivory tower architectures suffer from significant problems. First, the "minion developers" are unlikely to accept the architecture because they had no say in its development. Second, ivory tower architectures are often unproven, ivory tower architects rarely dirty their hands writing code, and as a result are a significant risk to your project until you know they actually work through the concrete feedback provided by a technical prototype. Third, ivory tower architectures will be incomplete if the architects did nothing else other than model because you can never think through everything your system needs. Fourth, ivory tower architectures promote overbuilding of software because they typically reflect every feature ever required by any system that your architect(s) were ever involved with and not just the features that your system actually needs.

Every system has an architecture but it may not necessarily have architectural models describing that architecture. For example, a small team taking the XP approach that is working together in the same room may not find any need to model their system architecture because everyone on the team knows it well enough that having a model doesn't provide sufficient value to them. Or, if an architectural model exists it will often be a few simple whiteboard sketches backed by a defined project metaphor. This works because the communication aspects of XP, including *Pair Programming* and *Collective Ownership*, negate the need for architecture model(s) that need to be developed and maintained throughout the project. Other teams – teams not following XP, larger teams, teams where people are not co-located – will find that the greater communication challenges inherent in their environment requires them to go beyond word-of-mouth architecture. These teams will choose to create architectural models to provide guidance to developers as to how they should build their software. Fundamentally, the reason why you perform architectural modeling is to address the risk of members of your development team not working to a common vision.

So how do you take an agile approach to architecture? You need to:

- Choose your architecture team.
- Base your architecture on requirements.
- Model your architecture.
- Travel light.
- Prove your architecture with concrete experiments.
- Communicate your architecture.
- Take an iterative and incremental approach to evolve your architecture over time.
- Think about the future, just wait to act.

Choose Your Architecture Team

The practice *Model With Others* tells you that you really don't want to be working alone, and frankly architecture is far too important to leave in the hands of a single person no matter how bright they are, therefore architecture should be a team effort. My strategy for choosing an architecture team is typically based on the size of the project team. On a small project team, say of ten people or less, I prefer to include all of the developers because it allows everyone involved to have their say in the architecture. This increases everyone's understanding and acceptance of the architecture because they worked on it together as a team. It also increases the chance that developers are willing to change aspects of the architecture when the architecture proves insufficient, perhaps it doesn't scale as well as you initially thought, because it is the group's architecture and not just theirs. When

something is developed by a single person it becomes “their baby” and nobody likes to hear that their baby is ugly – when you find a problem with their architecture they are likely to resist any criticisms of it. When an architecture is developed by the entire team then people are often far more willing to rethink their approach because it’s a team issue and not a personal issue.

On a large project, or for organization-wide architectural efforts, you will require a core architecture team^[1]. Your core architecture team should be comprised of developers experienced in the technologies that your organization is working with and have the ability to work on architecture spikes to explore new technologies. They should also have a good understanding of the business domain and have the necessary skills to communicate the architecture to developers and to other project stakeholders.

At the beginning of a large project I will identify my most experienced developers and abstract thinkers, as well as a few people that I want to see get some architectural experience, and invite them to be members of the core architecture team. I do this for two reasons. First, I want good people on this team. Second, when I organize the large project team into smaller subteams that will each focus on developing one or more subsystems I will see to it that each subteam includes one or two members of the core architecture team. This helps to increase the chance that each subteam learns and follows the architecture as well as increases the chance that the core architecture team will not ignore portions of the system. Furthermore, it ensures that each subteam has some senior people on it.

The core architecture team is responsible for identifying the initial architecture then bringing it to the rest of the project team for feedback and subsequent evolution. To avoid an ivory tower architecture the members of the core architecture team will take active roles on the various subteams on the project, communicating the architecture to the subteams and working with them to prove portions of the architecture via *concrete experiments*. From the point of view of the development subteams the architects act as consultants whose expertise is the corporate architecture as well as active members of the subteam. The core architecture team should view the development subteams as their customers, as project stakeholders for their architecture, customers that will need help from them and who will have new requirements that necessitate change to the architecture. In short, the core architecture team should actively work with their project stakeholders, in this case the development subteams.

The core architecture team will find that they need to get together occasionally to evolve the architecture as the project progresses, negotiating changes to the architecture and updating their architectural model(s), if any, as appropriate. These meetings will be frequent at the beginning of a project and will be needed less and less as the architecture solidifies. It will be common for members of the development subteams, who may not be members of the core architecture team, to attend some meetings to present information, perhaps they were involved with some technical prototyping and have findings to share with the architects. The best meetings are short, often no more than half an hour in length, and are often held standing up around a whiteboard – everyone should come prepared to the meetings, willing to present and discuss their issues as well as to work together as a team to quickly come to resolutions.

Requirements-Driven Architecture

Your architecture must be based on requirements otherwise you are hacking, it's as simple as that. The practice *Active Stakeholder Participation* is critical to your success when it comes to identifying architectural requirements – remember, requirements come from project stakeholders, not developers. Good sources for technical architecture requirements will include your users and their direct management as they will often have some insight into technical requirements and constraints. Operations staff will definitely have requirements for you pertaining to your deployment architecture. The best sources for business-oriented requirements are exactly who you would expect – your users, their managers. Senior management within your organization will have insights that may lead to potential change cases for your system.

As you would expect the practices *Apply The Right Artifact(s)* and *Create Several Models in Parallel* apply to your architectural requirements effort. When you are working on the technical aspects of your architecture you will want to base it on technical requirements, constraints, and possibly change cases. Similarly, when you are working on business aspects of your architecture, potentially identifying software subsystems or business components, you will likely need to focus on essential use cases or user stories that describe critical usage requirements and potentially the key business rules applicable to your system.

A common mistake that architecture teams will make is to ignore existing and pertinent artifacts, such as network or deployment diagrams that describe your organizations existing technical infrastructure, enterprise-level business models (use case models, process diagrams, workflow diagrams, corporate business rules, and so on), or corporate deployment standards (for workstations, branch offices, etc.) that your system is expected to conform to. Yes, the existing artifacts may be out of date or simply not apply to your effort, but you should at least make an effort to examine them and take advantage of the existing work wherever possible. A little bit of reading or discussion with the right people is likely to save you significant effort later on. In other words, don't forget to follow the practice *Reuse Existing Artifacts*.

An important concept to understand about architectural modeling is that although it typically occurs early in your project it never occurs first. Fundamentally, you will always invest time identifying some requirements first. Anything else is hacking, and hacking certainly isn't agile.

Model Your Architecture

The primary goal of architectural modeling should be to come to a common vision or understanding with respect to how you intend to build your system(s). In other words, you will *Model to Understand*. My experience is that 99.999% of all software project teams need to invest some time modeling the architecture of their system, and that this is true even of XP teams that rely on a

metaphor to guide their development efforts. While your XP team is identifying your system's metaphor, something that you and your teammates may think about for weeks as you are developing your initial releases, that you will often draw sketches of how you think your system will work. You may not keep these sketches, following AM's practice *Discard Temporary Models*, often because they were ideas that didn't work out or simply because you were modeling to understand an issue and once you did so the diagram no longer had value to you. Having said that, there is nothing wrong for an XP team to develop architecture models. The models may be something as simple as a sketch that you keep around on a publicly visible white board, because although metaphors can be very effective things an architectural model often provides the greater detail that your team requires. As you would expect RUP teams will also do some architectural modeling, not only because it is an effective thing to do but because it is a fundamental activity of the Analysis & Design workflow during the Elaboration phase.

How do you model your architecture in an agile manner? I typically strive to create one or more navigation diagrams, diagrams that present an overview of the "landscape" of your system. Just like a road map overviews the organization of a town, your navigation diagram(s) overviews the organization of your system. Navigation diagrams are the instantiation of your system's architectural views. When you read books and papers about architectural modeling a common theme that the authors put forward is the need for various architectural views, with each author presenting his or her own collection of critical views that you need to consider. My experience is that no one set of architectural views is right for every project, that instead the nature of the project will help to define the types of views that you should consider creating. The implication is that the type of navigation diagram that you create depends on the nature of the system that you are building. This is conceptually consistent with AM's practice *Apply the Right Artifact(s)* which tells you that you should use the right modeling technique for the task at hand. For example, a team building a complex business application using J2EE-based technology will likely find that a UML component diagram and a workflow diagram are appropriate for use as architectural navigation diagrams. However, a team building a corporate data warehouse will likely gravitate towards a data model and UML deployment diagram on which to base their architecture. Different projects, different architectural views, hence different types of navigation diagram(s). It is interesting to note that both projects needed two navigation diagrams, consistent with AM's principle *Multiple Models*. You need to be flexible in your approach because one size does not fit all.

A common mistake that organizations will make is to base their architectural efforts on their organization structure. For example, an organization with a strong data group will likely want to have a data model as the primary artifact for their architecture regardless of the actual nature of the system. When you have hammer specialists every problem looks like a nail to them. This problem is quite common when you are working with new technologies or attempting to develop a new class of system that your organization has little experience with – organization structures that worked well for you in the past may no longer work for you in your new environment. For more about the implication of architecture and organization structure please refer to the organizational pattern *Conway's Law* (Coplien & Harrison, 2001).

To create a navigation diagram the primary driver of your modeling efforts should be AM's principle *Assume Simplicity*. The practice *Create Simple Content* indicates that you should strive to identify the simplest architectural approach(es) possible – The more complicated your architecture the greater the chance that it won't be understood by individual developers and the greater the opportunity for error and breakdown. Let the principle *Work With People's Instincts* guide your efforts when you are trying to identify the simplest approach at this time –your implementation efforts will quickly reveal what is simple and what isn't, right now the best you can do is make an educated guess. Furthermore, your architectural models should contain the right level of information, showing how various aspects of your system work together but not the details (this is what design is all about) following the practice *Depict Models Simply*. You should also *Use the Simplest Tools* to do the job, many times a whiteboard sketch is all that you need to model the critical aspects of your architecture. Don't use a CASE tool when a drawing tool will do. Don't use a drawing tool when a whiteboard will do. Don't use a whiteboard when paper and Post-It notes will do.

An important point to be made is that navigation diagrams are typically sufficient to describe your architecture when all of your communication is face-to-face. When this isn't the case, when your architects are not able to work closely with the developers (perhaps some developers are at a distant location) then you will need to supplement your diagrams with documentation.

When you are architectural modeling you should consider taking advantage of the wealth of architectural patterns available to you but you should do so in an effective manner. The book *A System of Patterns: Pattern-Oriented Software Architecture* (Buschmann et. al., 1996) is an excellent place to start learning about common architectural patterns such as *Layers, Pipes and Filters, Broker, Model-View-Controller, and Blackboard*. As with analysis and design patterns, you should follow the practice *Apply Patterns Gently* – introduce them into your architecture only when they are clearly required. Until then if you suspect that an architectural pattern may be appropriate, perhaps you believe that you will have several sources of critical services that will need to be brokered, then model your architecture so that you can apply this pattern when this actually becomes the case. Remember that you are developing your system incrementally, following the practice *Model in Small Increments*, and that you don't need to get your architecture right on the very first day (nor could you achieve this goal even if you wanted to).

You should recognize that your architectural models will reveal your system's dependencies on other systems or their dependencies on yours. For example, your system may interact with a credit-card processing service via the Internet, access data from a legacy relational database, or produce an XML data structure for another internal application. Network diagrams and UML deployment diagrams are very useful for identifying these dependencies, as are process-oriented models such as workflow diagrams, UML activity diagrams, and data-flow diagrams. The implication is that these dependencies indicate the potential need to follow the practice *Formalize Contract Models* between your team and the owner(s) of the systems that yours share dependencies with. Ideally many of these models will already be in place, the credit card processor likely has a strictly defined protocol that you must follow and the legacy database likely has a physical data model defined for it, although new functionality such as the XML data structure will require adequate definition. Sometimes you will need to perform an analysis of the existing interface to a legacy system if accurate documentation is not in place and other times you will need to design a new interface. In both cases a corresponding contract model will need to be developed, either by your team, the other team(s), or co-jointly as appropriate.

How should you organize your architectural modeling efforts? At the beginning of a project I will typically gather the architecture team together in a single room for an initial modeling session. Ideally this session will last for no more than several hours but on some larger projects it may last for a few days. As always, the longer the modeling session the greater the chance of going off course due to lack of feedback. The goal of this modeling session will be to come to an initial agreement as to the landscape of the system that we are building, perhaps not consensus but sufficient agreement so we can start moving forward as a team.

Travel Light

One goal of your architectural efforts should be to *Travel Light*, to be as agile as possible. Don't create a fifty-page document when a five-page one will do. Don't create a five-page document when a diagram will do. Don't create a diagram when a metaphor will do. Not sure of how much to create? Err on the side of not having enough because you can always go back to the whiteboard if you need to, but the time you've wasted creating artifacts that you didn't need or adding unnecessary detail to artifacts is gone for ever. Your goal should be to think through the critical issues faced by your project team (or organization or even industry depending on your scope), it shouldn't be to create reams of documentation. The principle *Maximize Shareholder Investment* tells you to focus on high-value activities such as working through the hard problems as a team and coming to a common vision. Similarly, it tells you to avoid low-value activities such as writing detailed documentation or developing scores of pretty diagrams. These activities are often comforting because they provide the illusion of progress, and provide you a source of digression if you are trying to avoid dealing with difficult issues, but in reality are rarely as effective as you imagine because they are rarely looked at by anyone other than the author(s). The principle *Software is Your Goal* implies that you should model your architecture until the point where you believe you have a viable strategy, and at that point you should move on and start developing software instead of documentation.

When will you want to write architectural documentation? There are two instances where it makes "agile sense" in my opinion. First, when you have a distributed development team and you cannot find a more effective manner of communication, such as face-to-face conversation, then documentation is an option. Second, at the end of a project when you want to leave behind sufficient documentation so that someone else can understand your approach later on. The reality is that for reasonably complex systems it's incredibly difficult, if not impossible and certainly not desirable, to document everything in your code. Sometimes the best place to describe your architecture is in a brief overview document. This document should focus on explaining the critical aspects of your architecture, likely captured by your navigation diagrams, it might include a summary of key architectural requirements, and an explanation of the critical decisions behind "questionable" aspects of what you did. As always, if you're going to create an architecture document then it should add positive value and should ideally do so in the most effective way possible.

TIP: Architectural Documentation Isn't As Important As You Think

A lot of people get worried when they discover that an architecture isn't "well documented," whatever that means. I'm not so worried about this issue, but what I do worry about is whether the architecture is realistic and whether the developers understand and accept it if it is. If you were to prioritize having your architecture documentation, having a workable architecture, having the architecture understood by your developers, and having it worked to by all the developers I suspect that documentation would come in dead last on that list. Think about it.

Prove Your Architecture

The principle *Concrete Experiments* points out that a model is merely an abstraction, one that may appear to be very good may not actually be so in practice, that the only way you can know for sure is to validate your model through implementation. The implication is that you should prove that your architecture works, something that XP calls spikes (Jeffries, Anderson, & Hendrickson, 2001) and RUP calls architectural prototypes (Kruchten, 2000; Ambler Constantine, 2000b). When your architecture calls out for something that is new to you, perhaps you are using two or more products together for the first time, you should invest the time to explore whether or not this approach will work as well as how it works in accordance to the principle *Rapid Feedback*. Remember to obtain permission from your project stakeholders to perform this effort because it is their money you are spending. Sometimes you will discover through your efforts that your original approach doesn't work, something that I would prefer to find out sooner rather than later, and sometimes you discover how your approach actually works (instead of how you thought it would work). The development of an architectural spike/prototype helps to reduce risk to your project because you quickly discover whether your approach is feasible, that you haven't simply produced an ivory tower architecture.

Communicate Your Architecture

There are two primary audiences for which communication of your architecture is important: your development team and your project stakeholders. To promote communication within your development team I am a firm believer that you should follow the practice *Display Models Publicly* for all of your architectural diagrams because an architecture that is a closely guarded secret isn't an architecture, it's merely an egotistical exercise in futility. I've worked on several projects where we have successfully maintained a whiteboard that was reserved specifically for architectural diagrams, making them visible publicly to every developer on the project as well as to anyone else

who happened to walk by. We would also allow anyone who wanted to add comments or suggestions to the diagrams, along the lines of the principle *Open and Honest Communication* and the practice *Collective Ownership*, because we wanted their feedback on our work. We had nothing to hide and trusted that others would be willing to help us out (and they did).

At the start of a project, and less so throughout your project, you will often find that you need to make your diagrams “look pretty” so you can present them to your project stakeholders. Your stakeholders want to get a good idea as to what approach you intend to take to determine whether you are investing their resources wisely, which means you’ll need to *Model to Communicate* and orient some of your models so that others can understand them. This may mean you need to invest the time to clean up your models to make them presentable as well as write overview documentation for them. To remain as agile as possible the practice *Model With A Purpose* tells you that you should know exactly who you are developing the model(s) for and what they will use them for so you can focus on the minimum effort required. Presentations to important project stakeholders, efforts that are often annoying and distracting for developers, are critical to your project’s success as they provide opportunities for you to garner support for your project and to obtain needed resources. Furthermore, you can promote the importance of having stakeholders available to you that are able to actively participate on the project (you can’t follow the practice *Active Stakeholder Participation* if you don’t have stakeholders available to you).

Tip: Presentation Slides Don’t Need To Be As Pretty As You Think

Your project stakeholders have very likely attended hundreds if not thousands of presentations in their careers. They are no longer impressed by good looking power point slides and are very likely bored by the majority of presentations that they attend. Recognizing this I will often present hand-drawn diagrams in management presentations that I give, often digital pictures of diagrams drawn on whiteboards. Yes, this causes some consternation at first but when I explain to my audience that I had to make the choice between spending several days making my slides look good or working on new functionality they often agree with my “novel” approach. Never forget the principles *Content is More Important Than Representation* and *Maximize Stakeholder Investment*. Also, if you discover that management has a problem with this approach it’s a very good indication that they haven’t truly bought into the concept of agile software development at all.

Iterative and Incremental Architecture

Agile developers iteratively model their architecture, prove new portions of it, apply it on their project team or subteams, and then rework the architecture as appropriate. The end result is that

architecture emerges over time in increments, faster at first because of the greater need to set the foundation of a project, but still evolving over time to reflect the greater understanding and knowledge of the development team. This follows the practice *Model in Small Increments* and reduces the technical risk of your project – you always have a firm and proven foundation from which to work.

An alternative to this approach, albeit an extreme one, is to attempt to define your architecture completely before implementation begins, something often referred to as big design up front (BDUF). Often the motivation behind this approach is that project management doesn't want anyone moving forward until a consensus has been reached as to the approach. Unfortunately, this approach typically results in nobody moving forward for quite a long time, an ivory tower architecture that more often than not proves brittle in practice, an architecture that is overkill for what you actually require, and/or development subteams moving forward on their own because they can't wait for the architects to finish their work. This approach is often the result of a serial mindset among the people involved, a legacy thought process leftover from the days of waterfall software development (the 1970s and 1980s, when many of today's managers were software developers).

The reality is that the development of architecture is very hard, an effort that is key to your success, and one that you're not going to get right from the start. An iterative and incremental approach addresses the risk of an inadequate or inappropriate architecture by developing it a bit at a time, and only when you need it.

Think About The Future But Don't Overbuild

I suspect that the most controversial concept about agile architectural modeling is that you should consider future changes but not act on them until you actually need to. In other words, don't overbuild your system but at the same time be smart about it. The XP community is fairly blunt about the concept of overbuilding software with their belief that "You Ain't Gonna Need It Anyway" (YAGNI). The basic idea is that you cannot accurately predict the future^[2] and therefore shouldn't attempt to build for future possibilities. Instead you should focus today on building what you need to today and building it cleanly so that your software is easy to change when you need to. Tomorrow, when you discover that you need to change your software to fulfil new requirements then change it then. When you overbuild your software to be more general, to fulfill future potential requirements, you are actually making very serious trade-offs. First, you are not focusing on meeting today's needs, resulting in you not producing something of immediate value to your users. I've been on several projects where the first several months, and in a few cases first several quarters, of effort focused on the development of common infrastructure (persistence frameworks, messaging frameworks, and so on). Technically interesting things, we definitely had a lot of fun building them, but of no value to our users. Second, you don't really know if you will even need whatever it is that you're building – you may be building a Porche when a Volkswagon would have been sufficient. Third, anything that you overbuild today will need to be tested and maintained throughout the life of your project, violating the principle *Travel Light*. Fourth, when you overbuild

something the only way you can accurately validate it is via imaginary feedback – nobody asked for whatever you've overbuilt so you have no one to go to that can validate your work. Yikes.

So how can you be smart about all of this? Although you don't want to overbuild your system based on future/mythical requirements there isn't anything wrong with thinking about the future. This is where change cases become a consideration for you. Change cases (Bennett, 1997; Ambler 2001) are used to describe new POTENTIAL requirements for a system or modifications to existing requirements. Change cases are requirements you may, or may not, need to support in the future but you definitely do not need to support today. Change cases are often the result of brainstorming with your project stakeholders, where questions such as "How can the business change?" "What legislation can change?" "What is your competition doing?" and "Who else might use the system and how?" are explored. On the technical side developers will often ask fundamental questions such as "What technology can change?" and "What systems will we need to interact with?" that will lead to the identification of change cases. Change cases should be realistic, for example "We enter the insurance business" for a bank or "We need to support the [INSERT FLASHY NEW TECHNOLOGY] in our system" are reasonable change cases but "Our sales staff is abducted by UFOs" isn't. Furthermore, change cases typically describe requirements that are reasonably divergent from what you are currently working on, requirements that would potentially cause major rework to fulfill. By identifying change cases you are now in a position to intelligently choose between what would otherwise appear to be equal architectural or design decisions. You should only bring relevant change cases into the decision making process when your current requirements are not sufficient to help you to choose between alternatives. Another advantage is you can now explain to your project stakeholders why you chose one approach over another, as I like to say you have a story to tell. However, I cannot stress enough that change cases should not be used as excuses to gold plate your system. Stay agile and don't overbuild your system.

So what do you do when you think you have a change case that you truly believe needs to be implemented now? Simple – discuss it with your project stakeholders. Ask them if the change case is an immediate requirement, and if so act accordingly. If it isn't an immediate requirement then accept the fact and move on. Never forget that it is the project stakeholder's responsibility to prioritize requirements, not yours.

Would there be harm in modeling for the future? This is a slippery slope because I suspect that if you model it then you are much more likely to build it. It would require great discipline not to overbuild, I believe, because once you've got it captured as a collection of bubbles and lines it will be far too easy to convince yourself that there's no harm in overbuilding just this once. Having said that there's nothing wrong with drawing a few throw-away sketches as you're discussing a change case, just don't over model any models you intend to keep.

How Does This Work?

The architectural approach that I've described is markedly different than what a lot of organizations are currently doing today. Table Arch1 compares and contrasts the architectural practices that are commonly found in many organizations with their agile counterparts. Clearly, there's a big

difference. The agile approach works because of its focus on people working together effectively as a team. Agile Modeling recognizes that people are fallible, that they aren't likely to get the architecture right to begin with and therefore need the opportunity for acting on feedback from implementation efforts. When agile architects are productive members of the development team, and when the development team has been involved with the architectural efforts to begin with, then comprehensive documentation isn't needed by them, the navigation diagrams are sufficient (granted, when this is not the case documentation, hopefully minimal, may be required). Architecture reviews aren't needed because the architecture is being proved through the concrete feedback of architectural prototyping/spikes and because people can see the architecture evolve because your models are displayed publicly for everyone to see. Agile architects have the courage to focus on solving today's problem today and trusting that they can solve tomorrow's problem tomorrow (Beck, 2000), and the humility to recognize that they cannot accurately predict the future and therefore choose not to overbuild their architectures.

Table Arch1. Comparing Common and Agile Architectural Practices.

Common Practice	Agile Practice
Architects are held in high esteem and are often placed, or even worse place themselves, on pedestals	Agile architects have the humility to admit that they don't walk on water
Architects are too busy to get their hands dirty with development	Agile architects are active members of development teams, developing software where appropriate and acting as architectural consultants to the team
Architecture models are robust to enable them to fulfill future requirements	Agile architects have the humility to admit that they can't predict the future and instead have the courage to trust they can solve tomorrow's problem tomorrow
The goal is to develop a comprehensive architecture early in a project	You evolve your architecture incrementally and iteratively, allowing it to emerge over time
Well-documented architecture model(s) are required	Travel light and focus on navigation diagrams that overview your architecture, documenting just enough to communicate to your intended audience

Architecture model(s) are communicated only when they are “suitable for public consumption”	Architecture model(s) are displayed publicly, even when they are a work in progress, to promote feedback from others
Architecture reviews are held to validate your model(s) before being put into use	Architectures are proved through concrete experiments

Let Us Help

[Ronin International, Inc.](#) continues to help numerous organizations to learn about and hopefully adopt agile techniques and philosophies. We offer both [consulting](#) and [training](#) offerings. In addition we host several sites - [Agile Modeling](#), [Agile Database Techniques](#), [Enterprise Unified Process \(EUP\)](#) - that you may find of value.

You might find several of my books to be of interest, including [The Object Primer 3/e](#), [Agile Modeling](#), [The Elements of UML Style](#), and [Agile Database Techniques](#).

For more information please contact Michael Vizdos at 866-AT-RONIN (U.S. number) or via e-mail (michael.vizdos@ronin-intl.com).

References and Suggested Reading

See the [references page](#).

[Agile Modeling Home Page](#)

[Agile Modeling Essays](#)

Footnotes:

[1] Sorry but I refuse to introduce the acronym CAT for core architecture team.

[2] If you can accurately predict the future then I highly suggest you give up developing software and instead use your amazing powers to get rich playing the stock market or lottery. The rest of us mere mortals should instead have the humility to admit that we don't have the mutant superpower of prediction and instead stick to what we're good at: developing software based on actual requirements.