Contrary to popular belief, architecture is an important aspect of agile software development efforts, just like traditional efforts, and is a critical part of scaling agile approaches to meet the real-world needs of modern organizations. But, agilists approach architecture a bit differently than traditionalists do. This article addresses the following issues:

1. Towards Agile Architecture

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.

Here are a few ideas to get you thinking:

1. **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.
2. **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.
3. **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 plain old whiteboard (POW) sketches potentially 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

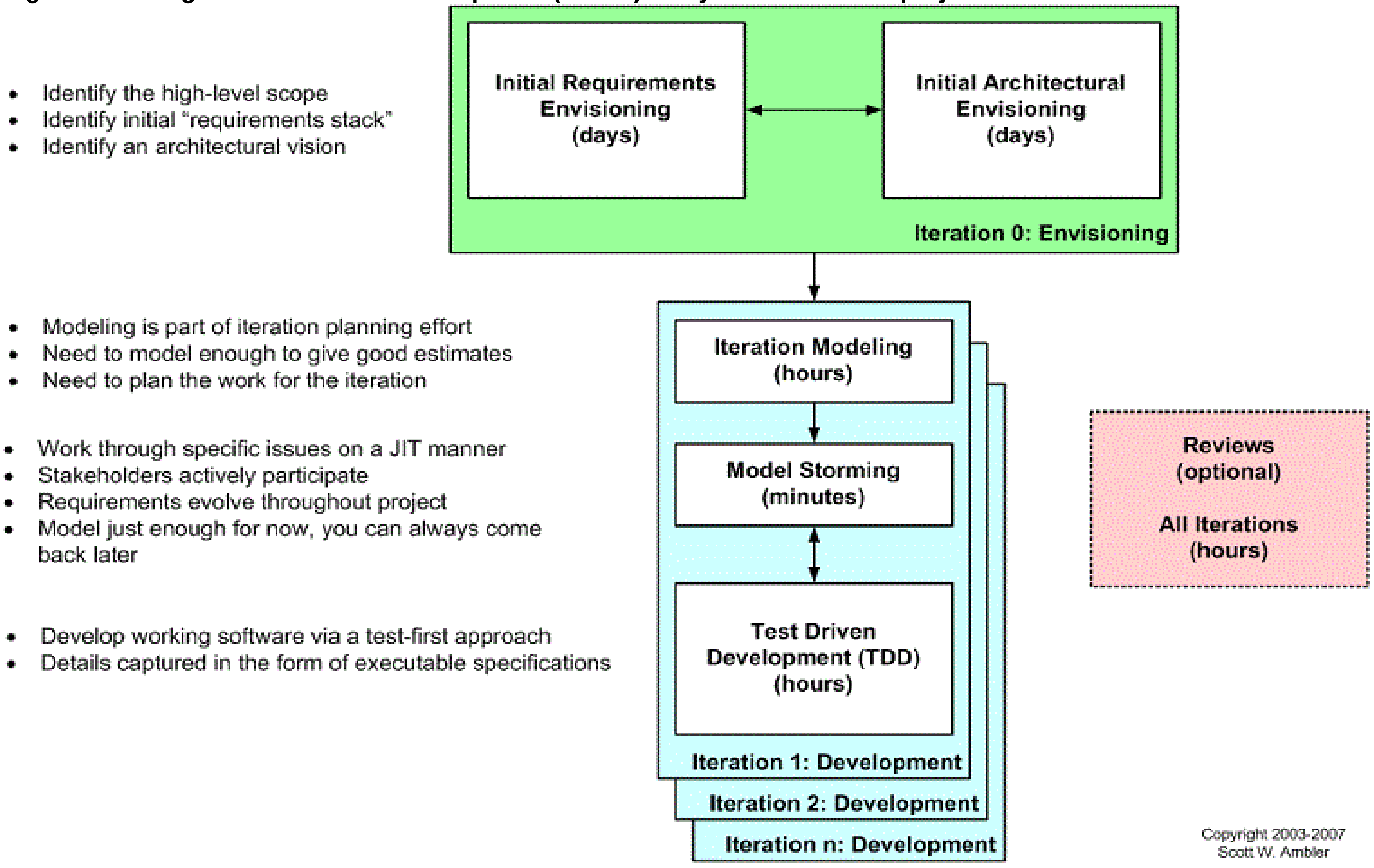
developed and maintained throughout the project. Other teams - teams not following XP, larger teams, teams where people are not co-located - will find that the greater communication challenges inherent in their environment requires them to go beyond word-of-mouth architecture. These teams will choose 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.

4. **Architecture scales agile**. This is true of traditional techniques as well. Have a viable and accepted architecture strategy for a project is absolutely critical to your success, particularly in the complex situations which agile teams find themselves in at scale. Scaling issues include team size, regulatory compliance, distributed teams, technical complexity, and so on (see The Software Development Context Framework (SDCF) for details). An effective approach to architecture enables you to address these scaling issues.

2. Architecture Throughout the Lifecycle

Figure 1 depicts the lifecycle of Agile Model Driven Development (AMDD). During "iteration 0", the Inception phase in Disciplined Agile Delivery (DAD), you need to get your project organized and going in the right direction. Part of that effort is the initial requirements envisioning and architecture envisioning so that you are able to answer critical questions about the scope, cost, schedule, and technical strategy of your project. From an architectural point of view, during iteration 0 the goal is identify a potential technical direction for your team as well as any technical risks which you will potentially face (risks which should be addressed by proving it with code). At this point you don't need a detailed architectural spec, in fact creating such a spec at the beginning of a software development project is a very big risk. Instead, the details are identified on a just-in-time (JIT) basis during iterations via initial iteration modeling at the beginning of each iteration or by modeling storming throughout the iteration. 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. In other words, you want to think about the future but wait to act.

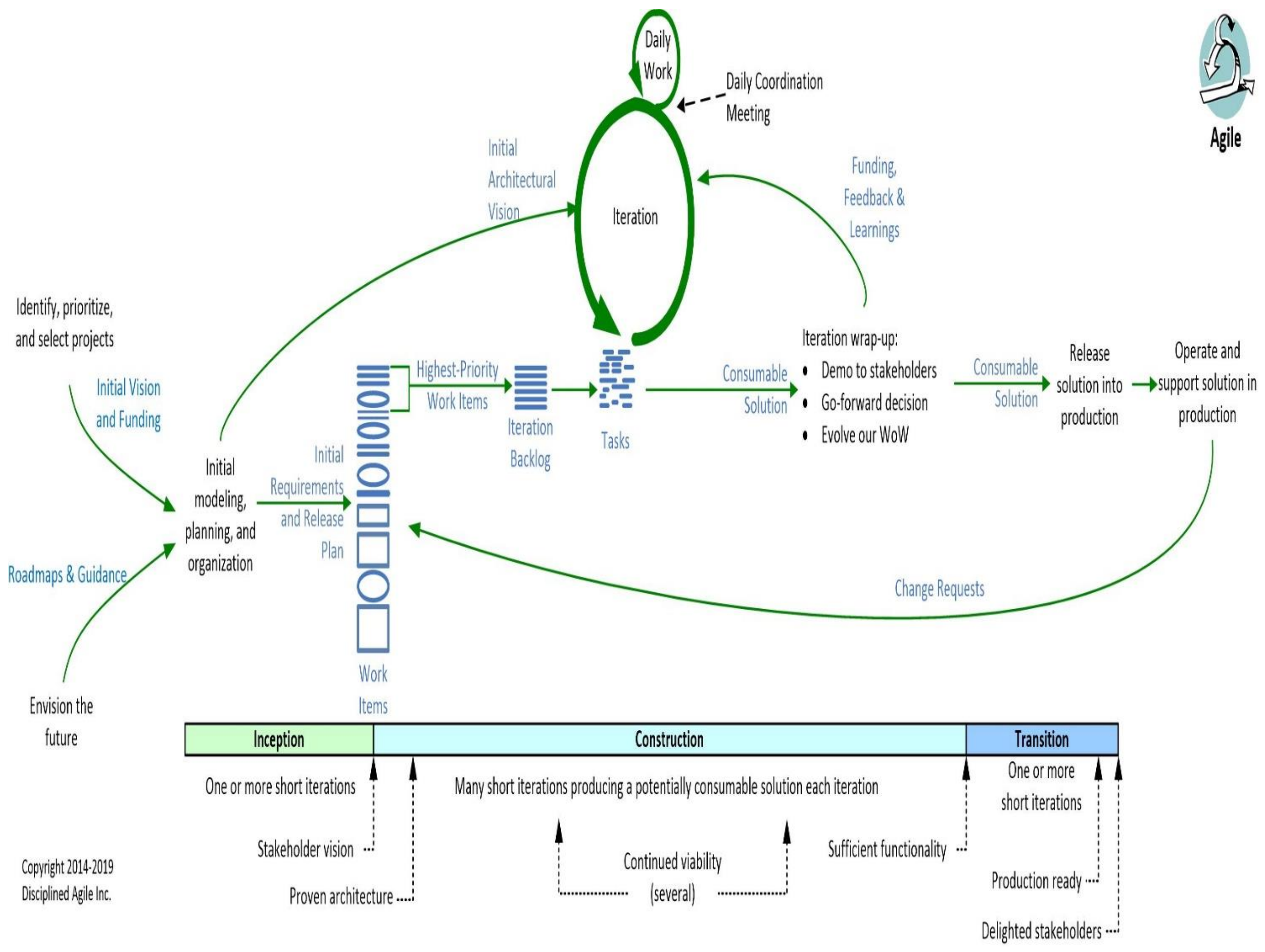
Figure 1. The Agile Model Driven Development (AMDD) lifecycle for software projects.



Identify the high-level scope
Identify initial "requirements stack"
Identify an architectural vision
Initial Requirements Envisioning (days)
Initial Architectural Envisioning (days)
Iteration 0: Envisioning
Modeling is part of iteration planning effort
Need to model enough to give good estimates
Need to plan the work for the iteration
Work through specific issues on a JIT manner
Stakeholders actively participate
Requirements evolve throughout project
Model just enough for now, you can always come back later
Develop working software via a test-first approach
Details captured in the form of executable specifications
Iteration Modeling (hours)
Model Storming (minutes)
Test Driven Development (TDD) (hours)
Iteration 1: Development
Iteration 2: Development
Iteration n: Development
Reviews (optional)
All Iterations (hours)
Copyright 2003-2007 Scott W. Ambler

Figure 2 depicts the agile/basic lifecycle described by the Disciplined Agile (DA) tool kit. The DA tool kit has all of the architecture strategies described in this article built right in. DA is a hybrid that takes strategies from a wide variety of sources, inlcuding Agile Modeling, Scrum, XP, Agile Data, and many others. In effect DA does the "heavy lifting" with regard to process in that it captures how ideas from these various methods fit together. Because DA isn't prescriptive, the software development portion of it, Disciplined Agile Delivery (DAD), supports several lifecycles. The lifecycle of Figure 2 is DAD's Scrum-based, or "basic", agile delivery lifecycle but it also supports a lean/Kanban type of lifecycle and continuous delivery lifecyclse as well. The idea is that your team should adopt the lifecycle that makes the most sense for the situation that you face.

Figure 2. The DAD Agile lifecycle (click to expand).



Daily Work
Daily Coordination Meeting
Agile
Initial Architectural Vision
Iteration
Funding, Feedback & Learnings
Identify, prioritize, and select projects
Initial Vision and Funding
Highest-Priority Work Items
Iteration Backlog
Tasks
Consumable Solution
Iteration wrap-up:
• Demo to stakeholders
• Go-forward decision
• Evolve our WoW
Consumable Solution
Release solution into production
Operate and support solution in production
Initial modeling, planning, and organization
Initial Requirements and Release Plan
Roadmaps & Guidance
Change Requests
Work Items
Envision the future
Inception
Construction
Transition
One or more short iterations
Many short iterations producing a potentially consumable solution each iteration
One or more short iterations
Stakeholder vision
Proven architecture
Continued viability (several)
Sufficient functionality
Production ready
Delighted stakeholders
Copyright 2014-2019 Disciplined Agile Inc.

An alternative to this light-weight approach to initial architecture modeling, is to attempt to define your architecture completely before implementation begins. This extreme is 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 or to the "one data truth". 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 evolutionary (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.

3. Who is Responsible for Architecture?

This question is more complex than you think. The easy answer, one that works well for small agile teams (which is the vast majority), is that everyone on the team is responsible for architecture. 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. On a small project team, say of fifteen 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.

However, there are there are two basic problems with the "everyone owns the architecture" strategy:

1. **Sometimes people don't agree**. This strategy can fall apart dramatically when the team doesn't come to agreement, hence you need someone in an architecture owner role to facilitate agreement.
2. **It doesn't scale**. When your team is large or geographically distributed, two of the eight scaling factors called out in the Software Development Context Framework (SDCF), you will organize your team into a team of subteams. Architecture at scale requires a coordinating body in such situations.

4. Have an "Architecture Owner"

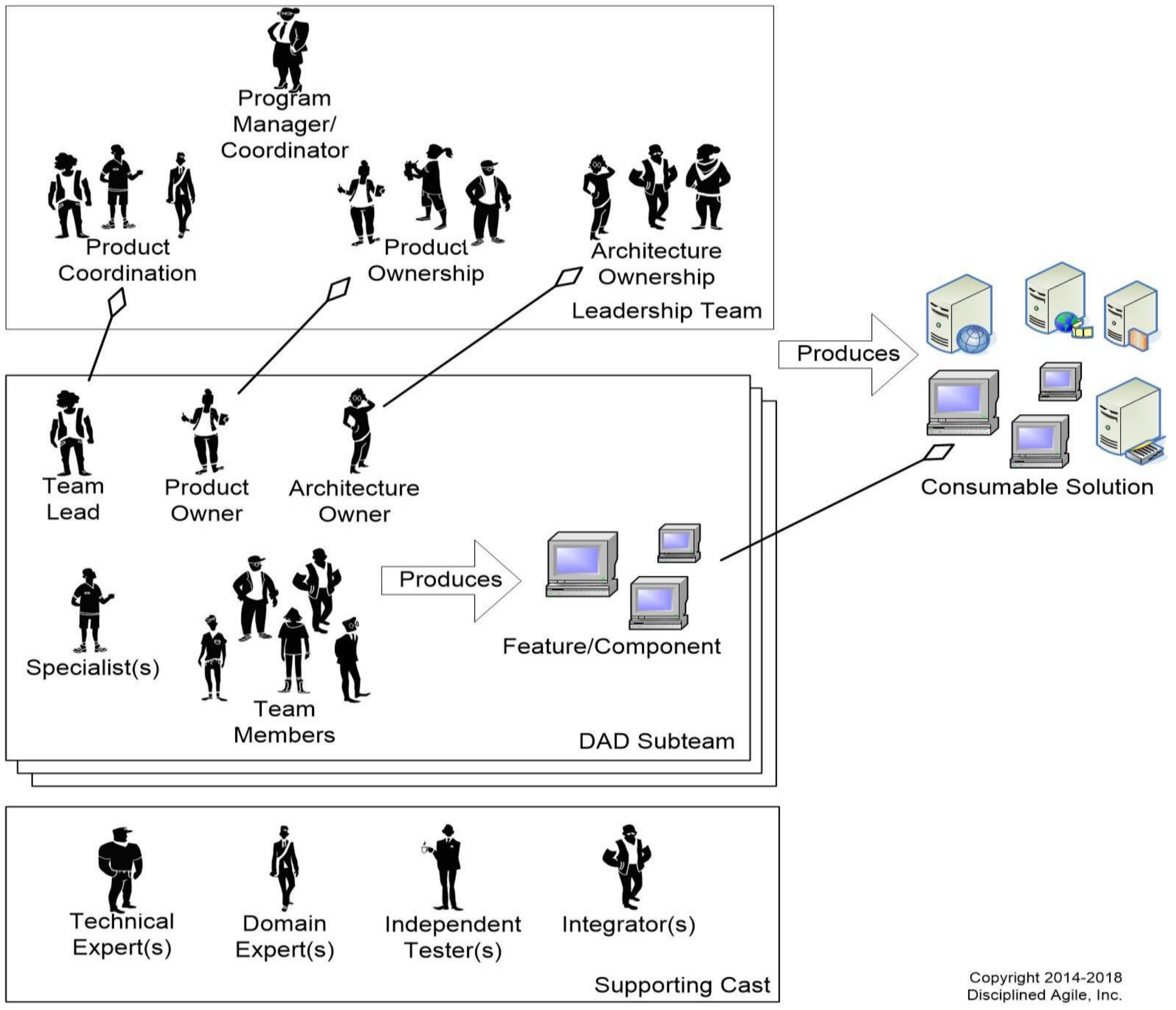
For any reasonably complex system you're going to need to invest some time architecting it. You'll do some up front architecture envisioning to get you started in the right direction and then the architecture will need to evolve from there. Many agile teams find that they need someone in the role of "architecture owner", sometimes called an agile solution architect. This person is often the most technically experienced person on the team, who is responsible for facilitating the architectural modeling and evolution efforts. Just like Scrum's product owner role is responsible for the team's requirements, the architecture owner is responsible for the team's architecture. Architecture owner is one of the primary roles in the Disciplined Agile (DA) tool kit.

Architecture owner is different than the traditional role of architect. In the past the architect would often be the primary creator of the architecture and would be one of the few people who worked on it. They would often develop the architecture and then "present it" to, or more accurately force it upon, the development team. An architecture owner collaboratively works with the team to develop and evolve the architecture. Although they are the person with the final decision-making authority when it comes to the architecture, those decisions should be made in a collaborative manner with the team. Effective architecture owners are developers experienced in the technologies that you organization is working with and have the ability to work on architecture spikes to explore new strategies. 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.

5. Agile Architecture at Scale

On large agile teams, geographically distributed agile teams, or for enterprise-wide architectural efforts, you will require an Architecture Owner team or Enterprise Architecture team (in Agile Modeling I originally called this a core architecture team, a term I never really liked). Large agile teams are often organized into smaller subteams, as you can see in Figure 3.
The architecture owner on each sub-team is a member of architecture owner team, which helps to increase the chance that each subteam understands and follows the overall architecture as well as increases the chance that the overall architecture strategy will address the full needs of the overall solution. There will be an an overall Chief Architecture Owner, this could be a rotating role, who is responsible for facilitating the group.

Figure 3. Agile teams at scale are organized into collections of subteams.



There are four basic strategies for organizing agile teams at scale:

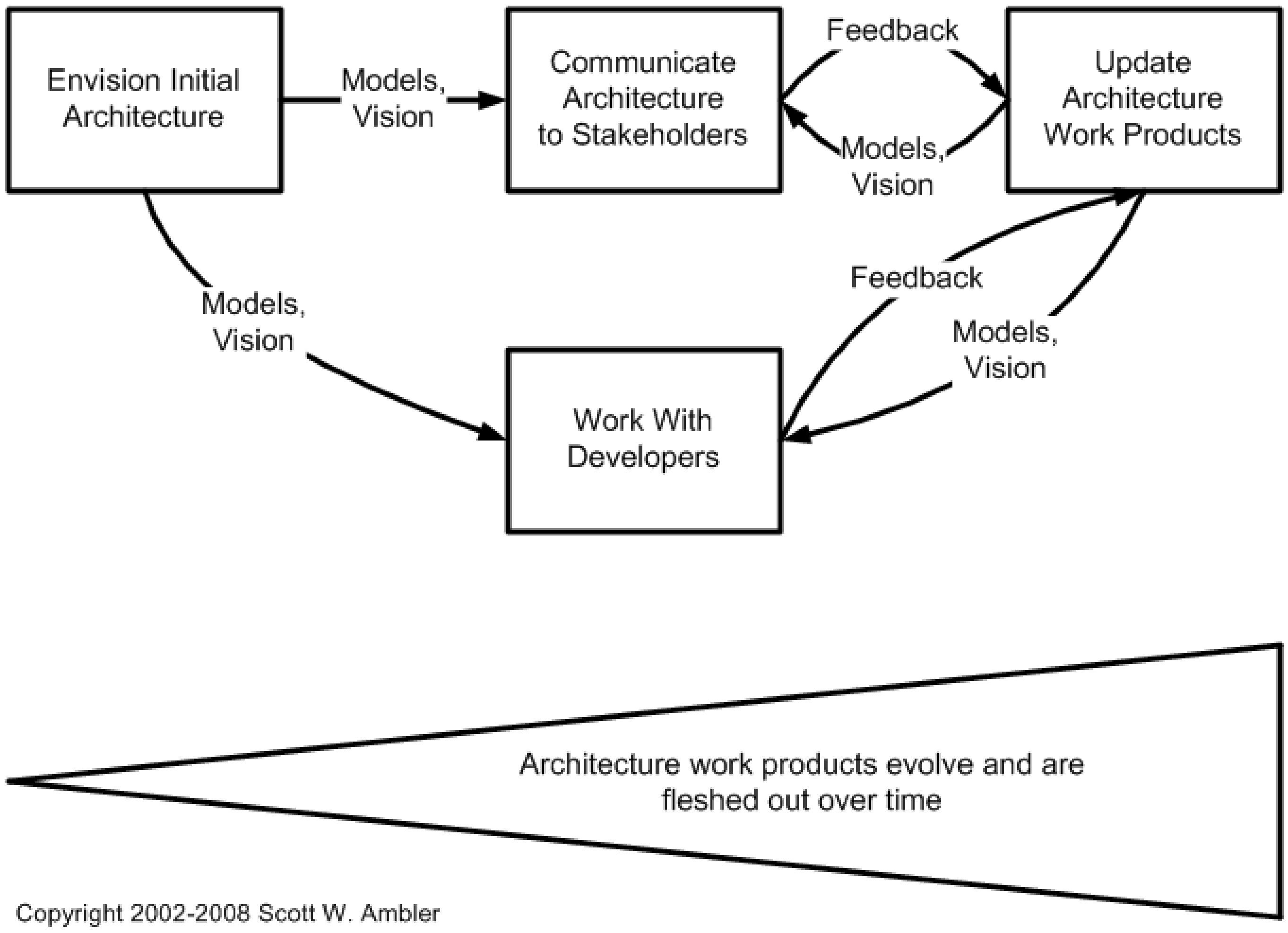
1. **Architecture-driven approach**. With this strategy you organize your subteams around the subsystems/components called out in your architecture. This strategy works well when your architecture is of high quality (it's loosely coupled and highly cohesive) and the interfaces to the subsystems have been identified before the subteams really get going (the interfaces will evolve over time, but you want to get a good start at them initially). The challenge with this strategy is that it requires your requirements to be captured in a way which reflects the architecture. For example, if your architecture is based on large-scale business domain components then a requirement should strive to focus on a single business domain if possible. If your architecture is based on technical tiers -- such as a 3-tier architecture with user interface (UI), business, and data tiers -- then requirements should focus on a single tier if possible.
2. **Feature-driven approach**. With this strategy each subteam implements a feature at a time, a feature being a meaningful chunk of functionality to your stakeholders. I would apply this strategy in situation where the architecture exhibits a lot of coupling AND where you have sophisticated development practices in place. The challenge with this approach is that the subteams often need to access a wide range of the source code to implement the feature and thereby run the risk of collisions with other subteams. As a result these teams sophisticated change management, continuous integration, and potentially even parallel independent testing strategies in place (to name a few).

3. **Open source approach**. With this strategy one or more subsystems/components are developed in an open source manner, even if it is for a single organization (this is called internal open source). This strategy is typically used for subsystems/components which are extensively reused by many teams, for example a security framework, and which must evolve quickly to meet the changing needs of the other systems accessing/using them. This strategy requires you to adopt tools and processes which support open source approaches.
4. **Combinations thereof**. Most agile teams at scale will combine the previous three strategies as appropriate.

Figure 4 depicts the process for architecture activities on an agile project at scale. You typically see take this sort of approach on large projects (often referred to as programmes), geographically distributed projects, complex (either domain or technical) projects, or at the enterprise level (typically to support agile enterprise architecture). There are four important aspects to this approach:

1. **Envision the initial architecture**. Minimally the architecture owner team is responsible for initial architecture envisioning and then bringing it to the sub teams for feedback and subsequent evolution. In the case of a large project/programme there are often other agile team members involved with this initial modeling effort, including the product owner and even key project stakeholders. The architecture envisioning efforts can go on for several days and in the case of very large or complex project several weeks. For enterprise architecture efforts the enterprise architecture team will often include project-level application/solution architects in their initial modeling efforts and often executive stakeholders.
2. **Working with the development teams**. On large projects/programmes, as you saw in Figure 3, the members of the architecture owner 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. For enterprise architecture efforts, the enterprise architects will minimally act as consultants whose expertise is the corporate architecture, but better yet they will be active members of the critical project teams taking on the role of architecture owner on those teams. Due to the collaborative nature of agile development it isn't sufficient for architecture owners to simply do initial architecture envisioning, and perhaps "support" project teams by reviewing their work occasionally, but instead they must "roll up their sleeves" and become active members of the project teams. This will help them to avoid creating "ivory tower architectures" which sound good on paper yet prove impractical in the real world. It also helps to increase the chance that the project team(s) will actually leverage the architecture.
3. **Communicating the architecture to architectural stakeholders**. For project teams the architectural stakeholders include the product owner(s) working with the agile delivery team(s), key project stakeholders, and of course the rest of the development team. These people need to understand the architecture vision, the trade-offs that have been made, and the current status of where you are implementing the architecture.
4. **Updating architectural work products**. The architecture owner 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 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.

Figure 4. Agile architecture process at scale.


Envision Initial Architecture
Models, Vision
Communicate Architecture to Stakeholders
Feedback
Models, Vision
Update Architecture Work Products
Models, Vision
Feedback
Models, Vision
Work With Developers
Architecture work products evolve and are fleshed out over time
Copyright 2002-2008 Scott W. Ambler

6. 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 (or for smaller projects the architecture owner) 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 reuse existing artifacts whenever possible.

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.

7. 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 Scrum/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 Disciplined Agile Delivery (DAD) teams will also do some architectural modeling.

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 the Multiple Models principle. 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.

To create a navigation diagram the primary driver of your modeling efforts should be to 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. 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 plain old whiteboard (POW) will do. Don't use a POW 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 architecture owners 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 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 typical gather the architecture team together in a single room for an initial envisioning session. Ideally this session will last for no more than several hours but on some larger projects it may last for a few days or even a few weeks (I would seriously question any effort more than two weeks). 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.

8. Consider Several Alternatives

As lean software development tells us, we shouldn't commit early to an architectural strategy but instead should consider several alternatives and to keep those alternatives "open" to us as long as they remain viable. The implication is that when you are envisioning the architecture early in the project you should really be envisioning several possible architectures. To be fair this strategy isn't new and in fact is one that has been promoted, although not always followed, within the IT architecture community for decades.

9. Remember Enterprise Constraints

All but the newest organizations have an existing technical infrastructure in place. More mature organizations may have:

- A "to be" vision for their technical infrastructure which they're working towards
- Enterprise-level standards and guidelines -- for development, data, user interfaces, security, and so on -- which project teams should follow

- A strategic reuse strategy for reducing overall development and operational costs
- Enterprise architecture, strategic reuse, and/or similar groups tasked with evolving and promoting these things
- An operations and support organization(s), sometimes referred to as a systems management organization, responsible for running your organization's systems in production

The point is that these enterprise-level considerations provide both challenges and opportunities to development teams. Although it would be wonderful to start with a clean architectural slate every time you build a new system, the reality is that strategy would be very inappropriate in the vast majority of situations. I've seen several agile project teams over the years that have been abysmal failures because they chose to start fresh, claiming that their architecture emerged over time, that they had the courage to worry about tomorrow's problem tomorrow, that they produced potentially shippable software on a regular basis, and basically parroting any other agile rhetoric which they believed justified their fooling around. Disciplined teams build systems whose architecture emerges within the organizational environment in which they're working. They have the humility to recognize that they're not in a position to make all of the technical decisions which they would like to, but instead are constrained by the existing infrastructure and vision for it. Furthermore, they produce potentially consumable solutions which work within their organizations ecosystem.

Luckily there are also many opportunities presented by having an enterprise focus. By leveraging the existing infrastructure teams can deliver faster because they have less to build. By using existing technologies, or at least by using new technologies (well, new to your organization) called out in the enterprise vision, they reduce the total cost of ownership (TCO) of their system by helping to minimize operational costs. By following corporate development guidelines they help to increase the consistency and quality of their work, increasing the maintainability of it for people tasked to evolve and maintain it in the future. As an aside, the enterprise discipline scaling factor of the Software Development Context Framework (SDCF) is the only scaling factor of the eight which has the benefit of things getting potentially easier for development teams as the factor moves away from a project-level focus (the "easy" situation) to an enterprise-level focus (the "hard" situation).

So how do you do this? Minimally, the enterprise groups such as your enterprise architects, operations groups, and so on are important stakeholders who should be represented by your product owner(s). In the case of your enterprise architecture group, one or more of them may become active members of your development team in the role of architecture owner(s). For other groups the product owner may choose to get them involved with your team as domain or technical experts on an as-needed and as-appropriate impromptu basis.

10. 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. Remember the "They Ain't Gonna Read It (TAGRI)" principle.

Not sure of how documentation 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 ROI 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 the are rarely looked at by anyone other than the author(s). The principle Software is Your Primary 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.

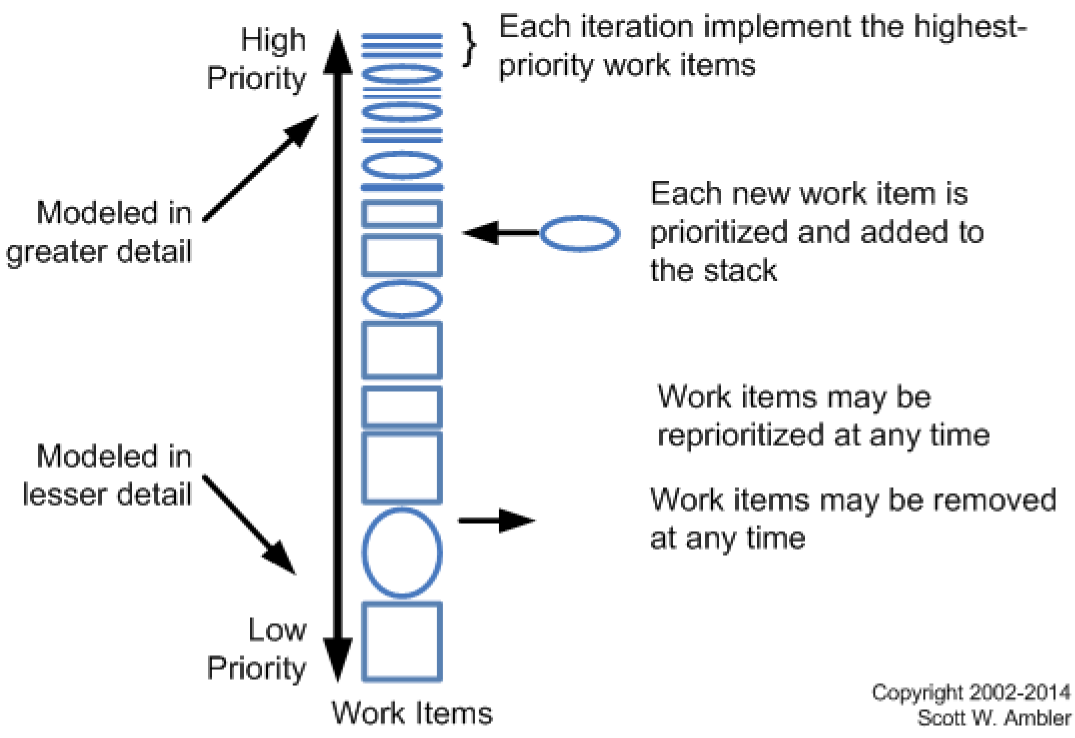
A lot of people get worried when they discover that an architecture isn't "well documented," whatever that means. I'm not so worried about this issue, but what I do worry about is whether the architecture is realistic and whether the developers understand and accept it if it is. If you were to prioritize 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.

11. Prove Your Architecture

The practice Prove it With Code 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 and RUP calls architectural prototypes. 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.

Figure 5 overviews an agile approach to the prioritized requirements "best practice". With a risk-value approach to the delivery lifecycle, an extension to Scrum's value-driven lifecycle, you will identify the handful of requirements which address the primary technical risks of your project. For example if you have a requirement which states that your system must be able to process 4,000 transactions a second over a period of 10 hours then that would be a requirement which clearly encapsulates some technical risk. This is the sort of requirement that you want to implement early in the project to ensure that your architecture actually works. My general rule is that it's better to find out that your architecture strategy needs to be rethought when you're only 18 days into a 6-month project instead of concluding that at the 8-month point of your "6 month project". The implication is that if the technically risky requirements aren't at the top of the backlog then you need to work closely with your product owner to convince them to reprioritize the few requirements which aren't at the top. However, if you can't convince your product owner to do so (I've never run into this problem in practice, but recognize that it could happen) then you need to respect their decision and accept the risk of proving your architecture later in the lifecycle.

Figure 5. A work backlog.


High Priority
Each iteration implement the highest-priority work items
Modeled in greater detail
Each new work item is prioritized and added to the stack
Work items may be reprioritized at any time
Modeled in lesser detail
Work items may be removed at any time
Low Priority
Work Items
Copyright 2002-2014 Scott W. Ambler

12. 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 your 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 principle 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).

Be prepared to communicate your architecture in a presentation, and there is no reason why you cannot keep the presentation agile.

13. Think About The Future But Don't Overbuild (Defer Commitment)

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[1] 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 fulfill 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:

1. **It's hard to estimate the actual value that you're producing**. 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.
2. **You're guessing**. 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.
3. **You're increasing your maintenance burden**. Anything that you overbuild today will need to be tested and maintained throughout the life of your project, violating the principle Travel Light.
4. **It isn't clear to what extent it needs to be tested**. 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. Furthermore, most development teams test to the risk, but if you're guessing at the requirements then you're also guessing at the level of risk.

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. It is a two phase strategy:

1. **Initial modeling**. Do some initial architecture envisioning to think things through, to explore critical concepts and thereby get you going in the right direction. This doesn't mean that you need to create mounds of architectural documentation, although it is likely that you will do some modeling and yes, egads!, even create sufficient architectural documentation to meet your needs.
2. **Defer design decisions**. One of the principles of lean software development is to defer commitment to the last possible moment that you need to make the decision, thereby increasing your flexibility and raising your chances of success.

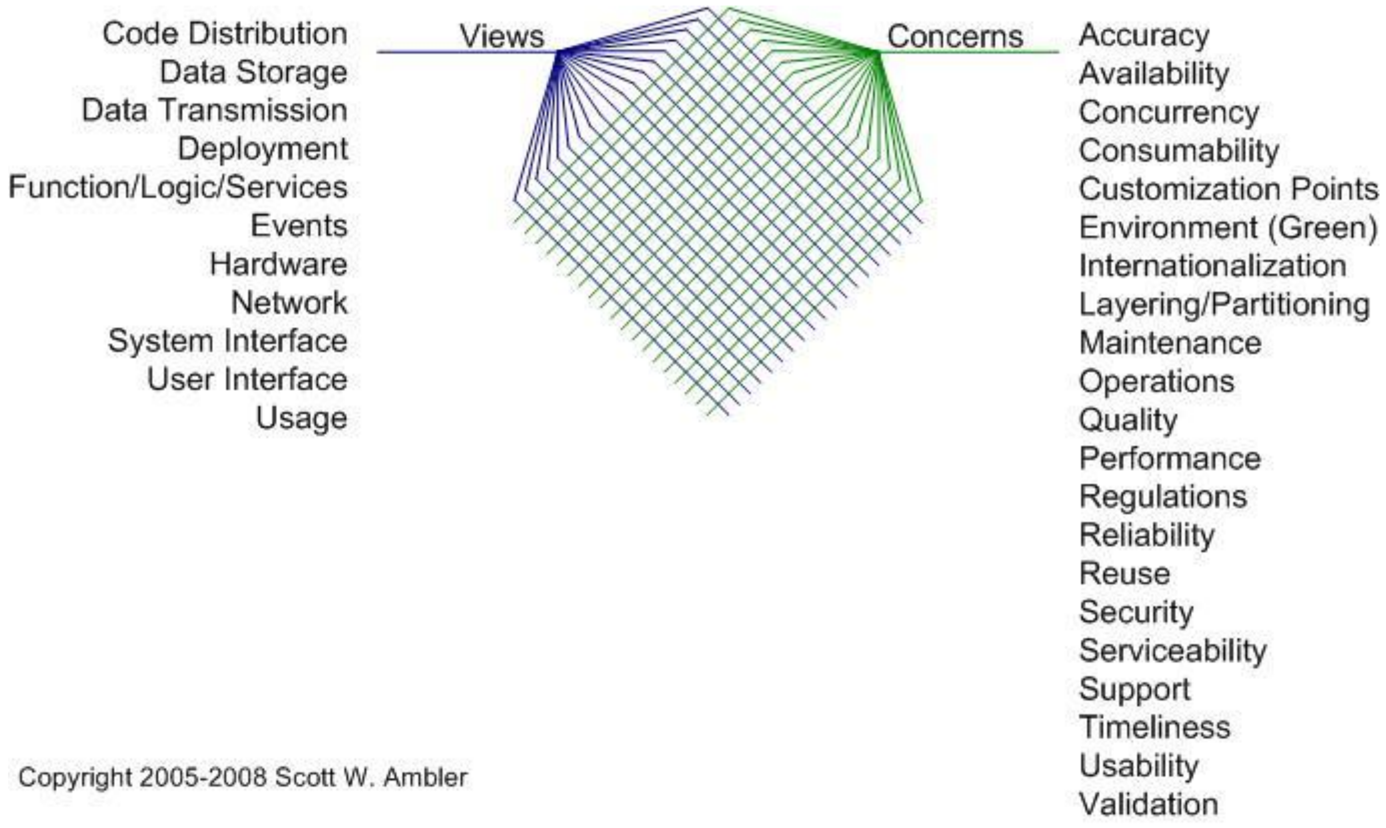
An interesting strategy is change case modeling. Change cases are used to describe new POTENTIAL requirements for a system or modifications to existing requirements. Change cases are requirements you may, or may not, need to support in the future but you definitely do not need to support today. Change 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.

14. Take a Multi-View Approach

Agile Modeling's Multiple Models principle advises you to recognize that because modern systems are complex that you will need to consider a range of views in your architecture. Although they take different approaches, a multi-view strategy is a fundamental concept in modern architectural frameworks such as the Zachman Framework, TOGAF, 4+1, and so on. Each of these frameworks have very good reasons for their choice of views, they all seem to work well in practice, they can all be approached in an agile manner, so my advice is to review your options and pick the architectural framework which best reflects the culture of your organization. My goal here isn't to propose yet another architecture framework, it is to make you aware of them and their underlying concepts. Figure 6 overviews the views and concerns (often referred to as quality of service requiremens) which software/system architects need to be concerned about.

Figure 6. Architectural views and concerns.



A view/viewpoint is captured as a combination of diagrams and text descriptions (such as use cases, technical specifications, or prose) . The potential issues, which may be views in and of themselves, which your views that your architecture should address include:

- Usage/business process
- User interface
- System interface

- Network
- Deployment
- Hardware
- Data storage
- Data transmission
- Events
- Code/component distribution
- Function/logic/services

To borrow the language of aspect oriented programming (AOP), there are also "cross-cutting concerns" which your architecture may also need to take into account. These concerns/perspectives should also be addressed by your architectural views and in some cases may be specific views in an of themselves. These concerns include:

- Layering/partitioning
- Reuse
- Quality and validation
- Accuracy and timeliness
- Reliability, Availability, Serviceability, and Performance
- Environment (green computing issues)
- Customization points
- Consumability (includes ease of (de)installation, level of support, usability, ...)
- Concurrency
- Security
- Internationalization
- Regulations
- Maintenance, operations, and support issues

The implication is that anyone in the role of architect needs to have a wide range of skills to be effective, that they need to move away from the traditional philosophy of over specialization and be more of a generalizing specialist. Minimally they should move away from being simply a data architect, or security architect, or network architect towards being an architect. Being just an architect is arguably too specialized as well, but that will vary depending on the situation. True professionals strive to have a wide range of skills as well as one or more specialties.

15. How Does This Work?

The architectural approach that I've described is markedly different that what a lot of organizations are currently doing today. Table 1 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 1. 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

16. Addressing the Myths Around Agile and Architecture

I wanted to end this article by addressing some of the common myths which I'm still running into when working with organizations around the world.

1. **Agilists don't do architecture**. My hope is that this article will put that myth firmly to rest.
2. **You need to do detailed up front architecture modeling**. You should do some up front architecture modeling to identify your general technical strategy, to identify potential technical challenges which you may run into, and to help build a consensus within your team around the technical direction. The point is that you don't need a lot of detail to achieve these goals. It's a choice to take a "big modeling up front (BMUF)" approach, a choice which can sound great in theory, particularly if you're a professional modeler, but which usually proves to be a rather poor one in practice. BMUF strategies lead to poor decisions and solutions which are less likely to meet the actual needs of stakeholders, reduce your ability to support evolving requirements, and lead to lower morale.
3. **Architects are responsible for architecture**. Although many organizations choose to support specialized architects who are primarily responsible for architectural activities, this proves to be a rather poor choice in practice because the developers are likely to perceive the architects as "ivory tower" and will often choose to ignore them. Effective architecture strategies, as you've seen in this article, are collaborative in nature, not dictatorial.