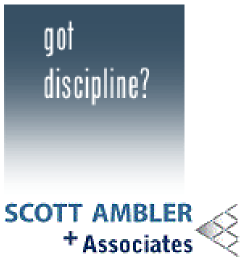
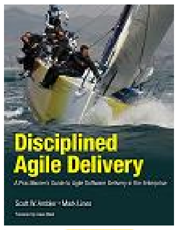
Agile Modeling

Agile Design

Home Start Here Best Practices Disciplines Artifacts Resources Contact Us #AgileModeling

Search





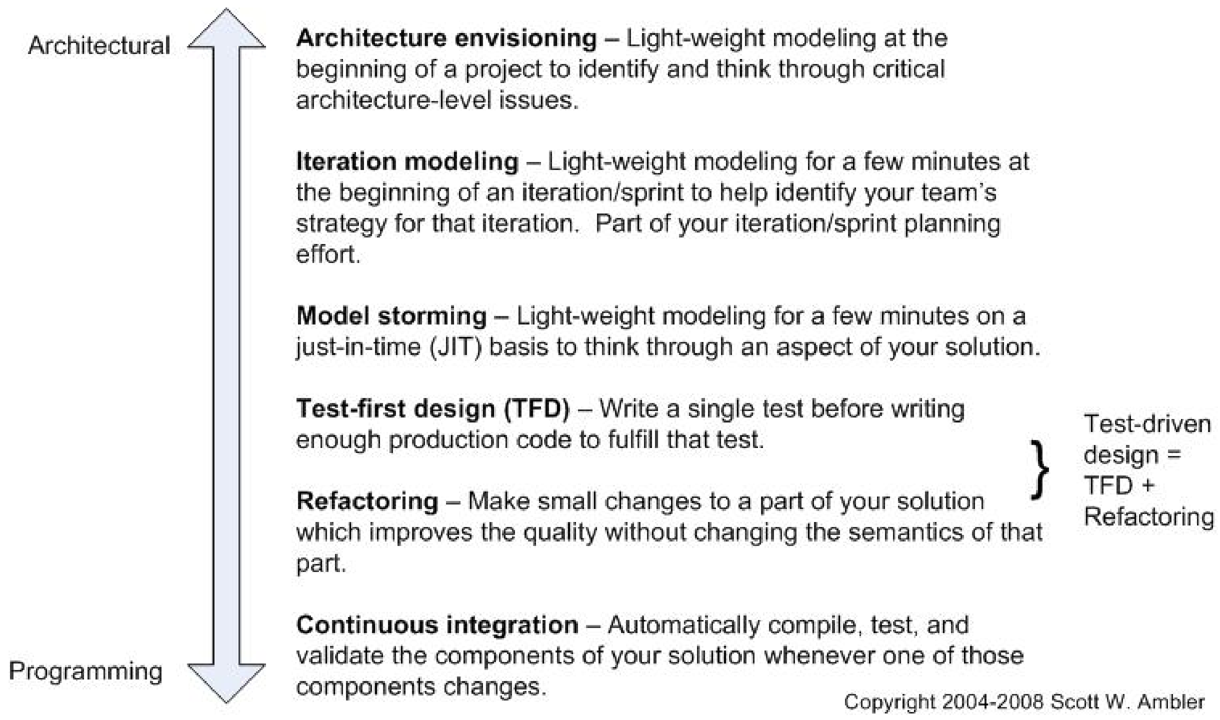
This article overviews design strategies for agile software development teams. These strategies are critical for scaling agile software development to meet the real-world needs of modern IT organizations. The Agile approach to design is very different than the traditional approach, and apparently more effective too. It is important to understand:

- Agile Design Practices
- Agile Design Philosophies
- Design Throughout The Agile Lifecycle

1. Agile Design Practices

There is a range of agile design practices, see Figure 1, from high-level architectural practices to low-level programming practices. Each of these practices are important, and each are needed if your team is to be effective at agile design.

Figure 1. Agile design practices.


Architectural
Programming
Test-driven design = TFD + Refactoring
Copyright 2004-2008 Scott W. Ambler

Architecture envisioning – Light-weight modeling at the beginning of a project to identify and think through critical architecture-level issues.

Iteration modeling – Light-weight modeling for a few minutes at the beginning of an iteration/sprint to help identify your team's strategy for that iteration. Part of your iteration/sprint planning effort.

Model storming – Light-weight modeling for a few minutes on a just-in-time (JIT) basis to think through an aspect of your solution.

Test-first design (TFD) – Write a single test before writing enough production code to fulfill that test.

Refactoring – Make small changes to a part of your solution which improves the quality without changing the semantics of that part.

Continuous integration – Automatically compile, test, and validate the components of your solution whenever one of those components changes.

2. Agile Design Philosophies

1. **Agile designs are emergent, they're not defined up front**. Your overall system design will emerge over time, evolving to fulfill new requirements and take advantage of new technologies as appropriate. Although you will often do some initial architectural modeling at the very beginning of a project during "iteration 0" this will be just enough to get your team going. Agilists don't need to get a fully documented set of models in place before you may begin coding (although sometimes, just sometimes, you may need to perform look-ahead modeling).
2. **Your unit tests form much of your detailed design documentation**. With a test-driven development (TDD) approach to development you write a test and then you write just enough domain code to fulfill that test. An important side effect of this approach is that your unit tests not only validate your code, they also form the majority of your design documentation in the form of executable specifications. TDD is complementary to AMDD and is actually scaled by AMDD.
3. **Design models need to be** just barely good enough. You don't need to model every single detail in your models, the models don't need to be perfect, and they certainly don't need to be complete. Remember the last time you coded from a design spec (if you ever did)? Did you really look at all the fine-grained details? No, because you were competent enough to handle the details yourself.
4. **Multiple models**. Effective developers realize that each type of model has its strengths and weaknesses, therefore they need to apply the right model(s) for the job at hand. Because software development is complex you quickly realize that you need to know about a wide range of models in order to be effective. All of the models mentioned in this newsletter, and more, are described at the Agile Models Distilled page.
5. **You typically only need a subset of the models**. Although there are many modeling techniques available to your, the fact is that any given project team will only require a subset. Think of it like this: in your toolbox at home you have a wide

array of screwdrivers, wrenches, pliers, and so on. For any given repair job you will use only a few of the tools. Different jobs, different tools. You never need all of your tools at once, but over time you will use them in a variety of manners.

6. **Each model can be used for a variety of purposes.** A UML class diagram can be used to depict a high-level domain model or a low-level design, not to mention things in between. Use cases can be used to model the essential nature of a process or the detailed system usage description which takes into account architectural decisions. Never underestimate how flexible you can be with models.
7. **Designers should also code**. Whenever a model is handed over to someone else to code there is significant danger that the programmer will not understand the model, will miss some of its nuances, or may even ignore the model completely in favor of their own approach. Furthermore, even when hand-offs are successful you will discover that you need far more details in your models than if you had simply coded it yourself. In short, separating design from programming is a risky and expensive proposition. It is far more effective to have generalizing specialists on your team that can both design and code.
8. **Prove it with code**. Never assume your design works; instead, obtain concrete feedback by writing code to determine if it does in fact work.
9. **Feedback is your friend**. Never forget that you are a mere mortal just like everyone else on your team. Expect to receive feedback -- I suggest you actively seek it -- about your work and be prepared to consider it and act accordingly. Not only will your system be the better for it, you will likely learn something in the process.
10. **Sometimes the simplest tool is a complex CASE tool**. When it comes to requirements I prefer inclusive tools such as paper and whiteboards, but when it comes to design I tend to lean towards sophisticated tools which (re)generate code for me. Like my grandfather always said, you should use the right tool for the job.
11. **Iterate, iterate, iterate**. With an iterative approach to development you work a bit on requirements, do a bit of analysis, do a bit of design, some coding, some testing, and iterate between these activities as needed. You will also iterate back and forth between working on various artifacts, working on the right artifact at the right time.
12. **Design is so important you should do it every day**. It is critical to think through how you're going to build something, to actually design it, before you build it. Your design efforts may take on the form of a sketch on a whiteboard, a detailed model created with a sophisticated modeling tool, or a simple test that you write before you write business code. Agile developers realize that design is so important that they do it every day, that design isn't just a phase that you do early in the project before getting to the "real work" of writing the source code.
13. **Design for your implementation environment judiciously**. Take advantage of features of your implementation environment, but be smart about it. Trade-offs are normal, but understand the implications and manage the risks involved. Every time you take advantage of a unique performance enhancement in a product (such as a database, operating system, or middleware tool) you are likely coupling your system to that product and, thus, reducing its portability. To minimize the impact of your implementation environment on your systems, you can layer your software and wrap specific features to make them appear general to their users.
14. **Document complicated things**. If it is complicated, then document it thoroughly. Better yet, invest the time to design it so it is simple. Remember the AM practice Create Simple Content.
15. **Do not over documen**t. You need to document your design, but you shouldn't over document either. Remember, users pay you to build systems, not to document them. There is a fine line between under documenting and over documenting, and only through experience are you able to find it. Be as agile as possible when it comes to documentation.
16. **Don't get sidetracked by the data community**. Unfortunately many within the data community believe that you require a serial approach to design, particularly when it comes to databases. This belief is the result of either not understanding evolutionary development or some misguided need to identify the "one truth above all else". Evolutionary database design techniques such as agile data modeling, database refactoring, and database regression testing work incredibly well in practice.

3. Design Throughout The Lifecycle

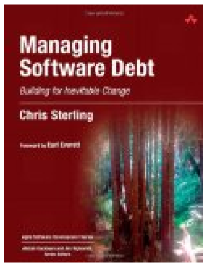


Figure 2 depicts the generic agile software development lifecycle. For the sake of discussion, the important thing to note is that there is no design phase, nor a requirements phase for that matter, which traditionalists are familiar with. Agile developers will do some high-level architectural modeling during Iteration 0, also known as the warm-up phase, and detailed design during development iterations and even during the end game (if needed).

Figure 2. The Agile SDLC (click to expand).

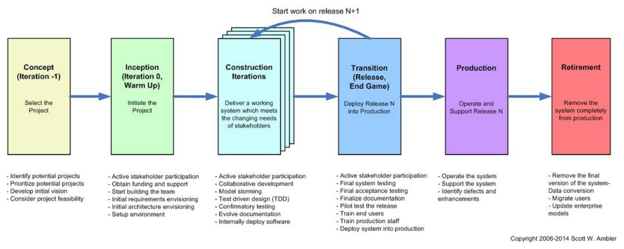

Start work on release N+1
Concept (Iteration -1)
Select the Project
Inception (Iteration 0, Warm Up)
Initiate the Project
Construction Iterations
Deliver a working system which meets the changing needs of stakeholders
Transition (Release, End Game)
Deploy Release N into Production
Production
Operate and Support Release N
Retirement
Remove the system completely from production
- Identify potential projects
- Prioritize potential projects
- Develop initial vision
- Consider project feasibility
- Active stakeholder participation
- Obtain funding and support
- Start building the team
- Initial requirements envisioning
- Initial architecture envisioning
- Setup environment
- Active stakeholder participation
- Collaborative development
- Model storming
- Test driven design (TDD)
- Confirmatory testing
- Evolve documentation
- Internally deploy software
- Active stakeholder participation
- Final system testing
- Final acceptance testing
- Finalize documentation
- Pilot test the release
- Train end users
- Train production staff
- Deploy system into production
- Operate the system
- Support the system
- Identify defects and enhancements
- Remove the final version of the system-
Data conversion
- Migrate users
- Update enterprise models
Copyright 2006-2014 Scott W. Ambler

Figure 32 depicts the Agile Model Driven Development (AMDD) lifecycle, the focus of which is how modeling fits into the overall agile software development lifecycle. Early in the project you need to have at least a general idea of how you're going to build the system. Is it a mainframe COBOL application? A .Net application? J2EE? Something else? During Iteration 0 the developers on the project will get together in a room, often around a whiteboard, discuss and then sketch out a potential architecture for the system. This architecture will likely evolve over time, it will not be very detailed yet (it just needs to be good enough for now), and very little documentation (if any) needs to be written. The goal is to identify an architectural strategy, not write mounds of documentation.

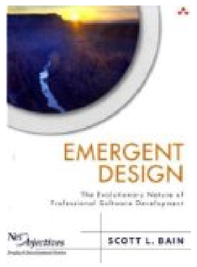
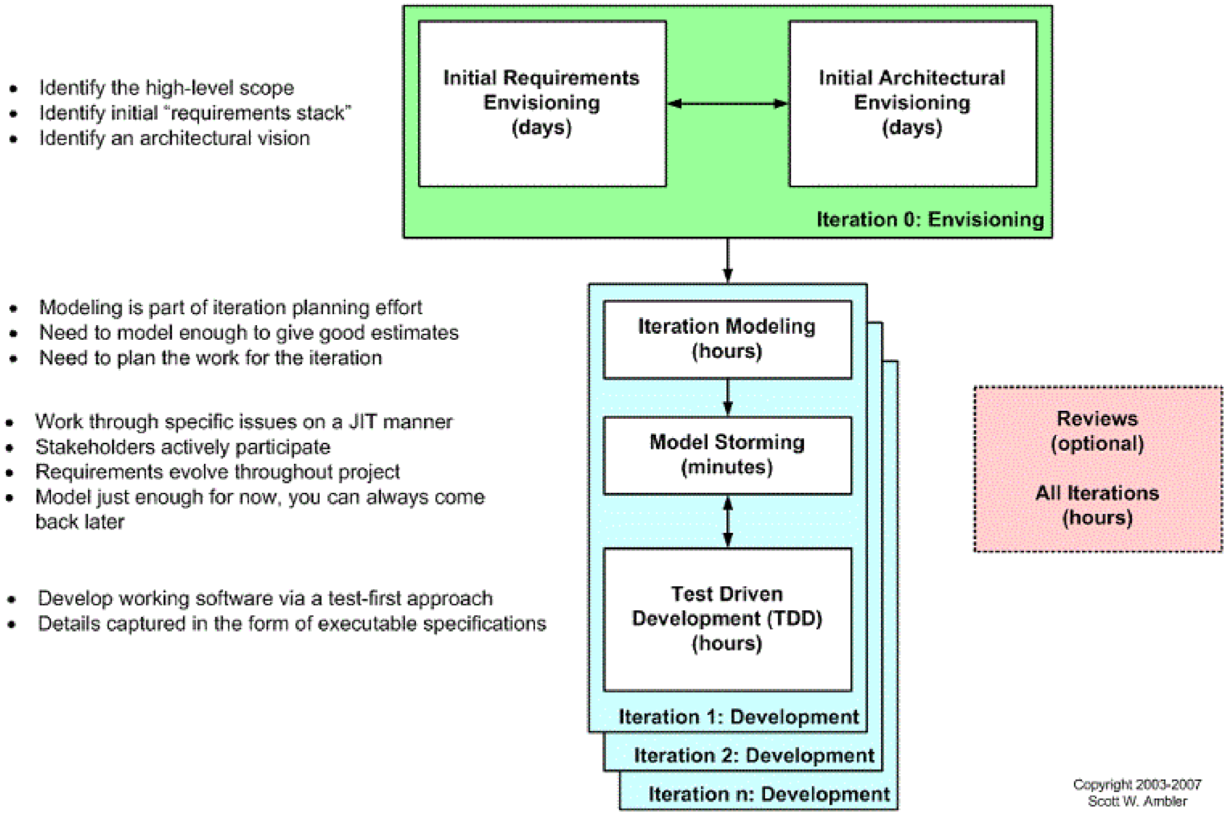


Figure 3. The AMDD lifecycle.



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
Iteration Modeling (hours)
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
Model Storming (minutes)
Reviews (optional)
All Iterations (hours)
Develop working software via a test-first approach
Details captured in the form of executable specifications
Test Driven Development (TDD) (hours)
Iteration 1: Development
Iteration 2: Development
Iteration n: Development
Copyright 2003-2007 Scott W. Ambler

When a developer has a new requirement to implement they ask themselves if they understand what is being asked for. If not, then they do some just-in-time (JIT) "model storming" to identify a strategy for implementing the requirement. This model storming is typically done at the beginning of an iteration during the detailed planning effort for that iteration, or sometime during the iteration if they realize that they need to explore the requirement further. Part of this modeling effort will be analysis of the requirement as well as design of the solution, something that will typically occur on the order of minutes. In Extreme Programming (XP) they refer to this as a "quick design session".

If the team is taking a Test-Driven Development (TDD) approach the detailed design is effectively specified as developer tests, not as detailed models. Because you write a test before you write enough production code to fulfill that test you in effect think through the design of that production code as you write the test. Instead of creating static design documentation, which is bound to become out of date, you instead write an executable specification which developers are motivated to keep up to date because it actually provides value to them. This strategy is an example of the AM practice of single sourcing information, where information is captured once and used for multiple purposes. In this case for both detailed specification and for confirmatory testing.

When you stop and think about it, particularly in respect to Figure 2, TDD is a bit of a misnomer. Although your developer tests are "driving" the design of your code, your agile models are driving your overall thinking.

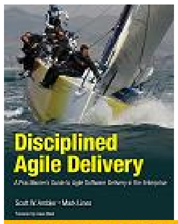
Share with friends: Tweet LinkedIn Facebook StumbleUpon Digg Baidu Google +

Let Us Help

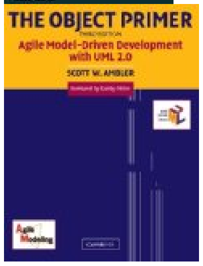
We actively work with clients around the world to improve their information technology (IT) practices, typically in the role of mentor/coach, team lead, or trainer. A full description of what we do, and how to contact us, can be found at Scott Ambler + Associates.

Recommended Reading

This book, Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise describes the Disciplined



Agile Delivery (DAD) process decision framework. The DAD framework is a people-first, learning-oriented hybrid agile approach to IT solution delivery. It has a risk-value delivery lifecycle, is goal-driven, is enterprise aware, and provides the foundation for scaling agile. This book is particularly important for anyone who wants to understand how agile works from end-to-end within an enterprise setting. Data professionals will find it interesting because it shows how agile modeling and agile database techniques fit into the overall solution delivery process. Enterprise professionals will find it interesting beause it explicitly promotes the idea that disciplined agile teams should be enterprise aware and therefore work closely with enterprise teams. Existing agile developers will find it interesting because it shows how to extend Scrum-based and Kanban-based strategies to provide a coherent, end-to-end streamlined delivery process.



The Object Primer 3rd Edition: Agile Model Driven Development with UML 2 is an important reference book for agile modelers, describing how to develop 35 types of agile models including all 13 UML 2 diagrams. Furthermore, this book describes the fundamental programming and testing techniques for successful agile solution delivery. The book also shows how to move from your agile models to source code, how to succeed at implementation techniques such as refactoring and test-driven development(TDD). The Object Primer also includes a chapter overviewing the critical database development techniques (database refactoring, object/relational mapping, legacy analysis, and database access coding) from my award-winning Agile Database Techniquesbook.

DISCIPLINED AGILE | Agile Modeling | Agile Data | Enterprise Unified Process | Agile Unified Process | Ambysoft | The Enterprise Transformation Advisor™

SCOTT AMBLER + Associates | @scottwambler
