Agile Modelling in Software Engineering

Audrey Nemeth andVladimir Borisov

May 22, 2006

Contents

1 Introduction

1.1 Abstract

Agile Modelling is a concept invented in 1999 by Scott Ambler as a supplement to Extreme Programming (XP) [Source: `Agile Modelling Values`]. Strictly defined, Agile Modelling (AM) is a chaordic, practices-based methodology for effective modelling and documentation [Source: `Interview with SA by Clay Shannon`]. AM is based upon three components that each consist of sets of ideals and strategies. These components are: values, principles, and practices. The collection of AM practices is based upon AM values and principles. By learning how to apply the elements of Agile Modelling to software engineering, teams can profit from effective, streamlined systems of communication which produce models and documentation that fulfill developers' and customers' needs while costing team members minimal time and effort.

The aim of this paper is to give a detailed introduction to the topic of Agile Modelling. Starting in 1.2 with a definition and explanation of the elements of AM, we will then discuss in 2.1 what types of projects AM is best suited for, and also where AM should not be used. Chapter 3 deals with compatible methodologies that are often used as a complement to AM in software engineering. These include AMDD, Extreme Programming (XP), Unified Process, etc. Finally, Chapter 4 will focus on optimization, and specifically, how to obtain the best results using Agile Modelling.

1.2 Definition and Elements of Agile Modelling

First, let's examine the definition of Agile Modelling. AM is characterized as "chaordic" and "practice-based." Chaordic is describes *systems that are simultaneously chaotic and ordered.* It refers to a harmonious coexistence of chaos and order displaying characteristics of both, with neither chaotic nor ordered behaviour dominating. For example, chaordic systems have been proposed as a possible approach to consensus decision-making that is neither hierarchical nor anarchic. [Source: `Wikipedia`] AM is called practices-based, as opposed to prescriptive, which means it does not define detailed procedures for how to create a given type of model; instead, it provides advice for how to be effective as a modeller [Source: `AM FAQ`].

As stated before, Agile Modelling consists of three components which are related in the following way: AM *practices* are based on several *values* and proven software engineering *principles* [Source: `Agile Modeling`]. To understand Agile Modeling, first we must understand the purpose of these components and how they can be applied in Software Engineering.

1.2.1 Values

A value is defined by The American Heritage® Dictionary as *a principle, standard, or quality considered worthwhile or desirable* [Source: `dictionary.com`]. In Agile Modeling, there are five principal values. These are communication, simplicity, feedback, courage, and humility [Source: `An Introduction to Agile Modeling`]. The first three values pertain to the role of models within a project, and the remaining values describe desirable personality traits of software designers.

Models facilitate *communication* between the designers and the customer, and between the designers and developers. Models promote *simplicity* both by clarifying design problems and representing programming concepts in a more understandable or generalized manner. Models help designers to obtain *feedback* quickly, allowing them to act on the advice they receive.

Designers require *courage* to make decisions, and also to admit mistakes and change direction if necessary. Designers require *humility* to recognize the abilities of other team members and value their contributions.

1.2.2 Principles

Next, let's see how these values lead to principles. WordNet 2.0[1] defines a principle as *a basic generalization that is accepted as true and that can be used as a basis for reasoning or conduct* [Source: `dictionary.com`]. Agile Modeling contains two kinds of principles: core principles and supplementary principles [Source: `Agile Modeling Principles`]. Core principles of Agile Modeling include:

- Assume Simplicity – "*The simplest solution is the best solution.*"
- Embrace Change – *Requirements and available technologies change over time. An approach to development can only be successful if it embraces the reality of change.*
- Enabling the Next Effort is Your Secondary Goal – *Ensure robustness of the current system before you even think about the next version.*
- Incremental Change – *Don't insist that a model be perfect the first time. Start with an incomplete or generalized model, and refine it over time.*
- Maximize Stakeholder Investment – *Customers deserve final say in determining the distribution of resources.*
- Model With a Purpose – *This requires knowing your audience. Before creating a model, determine its purpose and its audience, and then develop the model just until it is sufficiently detailed and accurate to serve its purpose, and not more.*
- Multiple Models – *Be aware of the types of models available: UML Class & Activity Diagrams, Change Cases & Use Cases, CRC Models, Communication Diagrams, Constraint definitions, UML Deployment Diagrams, Flowcharts, etc. Just as a particular repair job at home is best completed using more than one tool from your toolbox, a particular project generally requires multiple modelling techniques.*
- Quality Work – *Do quality work.*
- Rapid Feedback – *Obtained by working closely with the customer.*
- Software Is Your Primary Goal – *Remember that documentation, models, etc. are a means to an end, and not the end itself. The final product is the software.*
- Travel Light – *Keep the number & complexity of models within your project to a minimum, to reduce the cost of updating and maintaining models.*

Supplementary principles of Agile Modelling include:

- Content is More Important Than Representation – *Consider whiteboards or post-it notes as an alternative to CASE tools. These allow you to "take advantage of the benefits of modelling without incurring the costs of creating and maintaining documentation."* [Source: `Agile Modelling Principles`]
- Open and Honest Communication – *Enables people to share ideas or even admit shortcomings, which in turn enables the team to make decisions based upon more accurate information.*

Previously, there were more principles in Agile Modeling, which were deprecated in favour of simplification in January, 2005. These included: Everyone Can Learn From Everyone Else, Know Your Models, Know Your Tools, Local Adaptation, and Work With People's Instincts.

1.2.3 Practices

Finally, let's examine how the values and principles discussed above lead to the practices of Agile Modelling. We'll start with the definition of practices given by The American Heritage Dictionary

[1] WordNet ® 2.0, © 2003 Princeton University

[Source: `dictionary.com`]: practice is the process of doing something such as a performance or action. Again, Agile Modelling subdivides practices into two groups: core and supplementary practices. Core practices of Agile Modelling include:

- Active Stakeholder Participation – *This is analogous to XP's On-Site Customer. The idea is to have the customer and end user participate in requirements elicitation and prioritization of requirements.*
- Apply the Right Artifact(s) – *Know the types of models available, and choose one that concisely conveys information while allowing flexibility to change the design.*
- Collective Ownership – *Applies to all models and all 'artifacts'.*
- Create Several Models in Parallel – *This technique helps to create a fuller understanding of the system or subsystem, while helping to determine which model represents the system most simply, fully and accurately.*
- Create Simple Content – *Analogous to XP's practice of Simple Design.*
- Depict Models Simply – *Models should basically highlight key features.*
- Display Models Publicly – *Preferably create a "modelling wall."*
- Iterate to Another Artifact – *If one modelling tool is insufficient, try another.*
- Model in Small Increments – *Incremental development allows more rapid prototyping.*
- Model With Others – *Use modelling to develop a common vision on a project.*
- Prove it With Code – *Prove the efficacy of finished models by implementing them.*
- Single Source Information – *Minimize the redundancy between models.*
- Use the Simplest Tools – *Consider Whiteboards or paper as an alternative to modelling tools.*

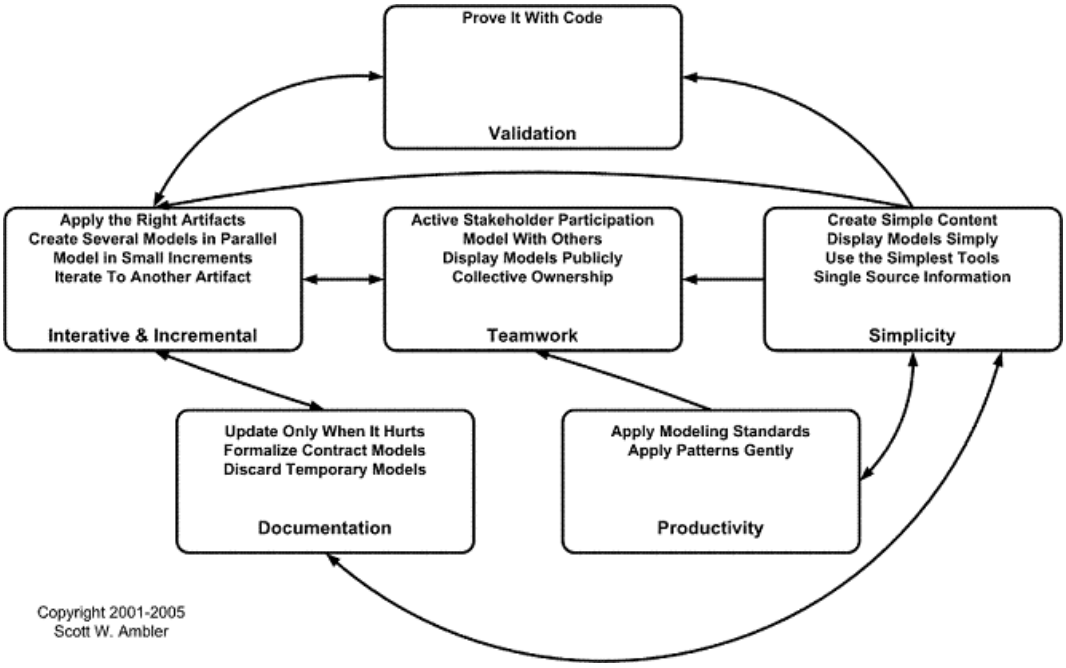

Figure 1: How AM Practices Fit Together [Source: `AM Practices`]

Supplementary practices include:

- Apply Modelling Standards – *No explanation necessary.*
- Apply Patterns Gently – *When applying a pattern, create the minimal infrastructure necessary for the pattern while structuring the code in such a way that it would be easy to refactor and apply the full-fledged pattern later, if that turns out to be the easiest solution.*

- Discard Temporary Models – *Do not update or keep models that have outlived their usefulness.*
- Formalize Contract Models – *Create a contract model whenever an external group controls an information resource that the system requires, such as a database, legacy application or information service.*
- Update Only When It Hurts – *or only when you absolutely have to.*

Deprecated practices include: Consider testability, model to communicate, model to understand, and reuse existing resources.

Finally, not included as practices are the following complementary strategies, generally grouped as 'Really Good Ideas': Refactor, and practice Test-First Design.

1.3 What is AM used for? What is it not used for?

Scott Ambler summarizes it best with the following list. Agile modelling is a good solution for you if:

- You are taking an agile approach to development in general
- You plan to work iteratively and incrementally
- The requirements are uncertain or volatile
- The primary goal is to develop software
- The active stakeholders are supportive and involved
- The development team is in control of its destiny
- The developers are responsible and motivated
- Adequate resources are available for the project

Agile Modelling does not work well in organizations that have a prescriptive culture, or for teams that are very large and/or distributed [Source: `Modeling and Documentation Rethunk`].

2 AM in Software Development Projects

As described above, AM is a collection of values, principles, and practices for modeling software that can be applied to a software development project in an effective and light-weight manner. As shown in Figure 2, AM is meant to be tailored into other, full-fledged methodologies, enabling you to develop a software process which truly meets your needs.

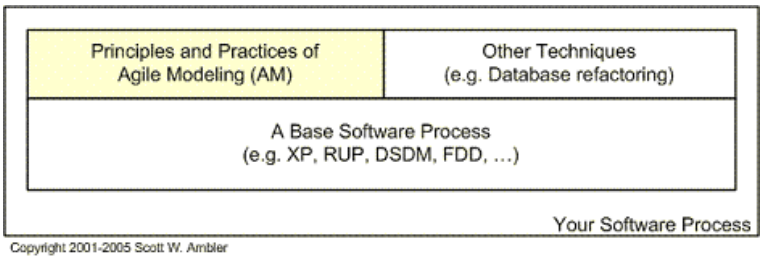

Figure 2:

2.1 Agile Model Driven Development (AMDD)

AMDD is a more agile version of Model Driven Development (MDD). MDD is an approach to software development where extensive models are created before source code is written. With MDD a serial approach to development is often taken. MDD is quite popular with traditionalists, although as the RUP/EUP shows it is possible to take an iterative approach with MDD. The difference with AMDD is that instead of creating extensive models before writing source code you instead create agile models which are just barely good enough.

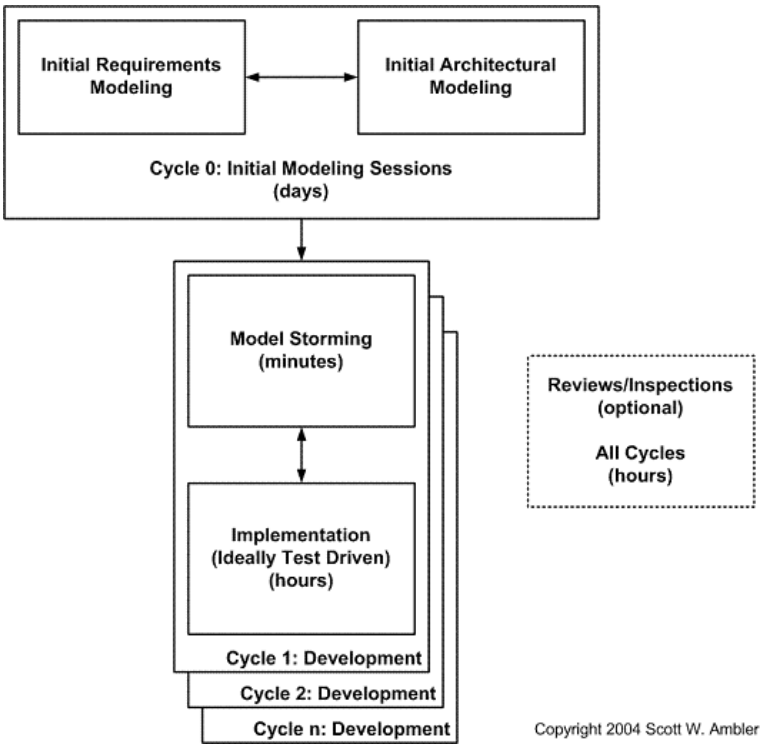

Figure 3:

Figure 3 depicts a high-level lifecycle for AMDD for the release of a system. Each box represents a development activity. The initial modeling activity includes two main sub-activities, initial requirements modeling and initial architecture modeling. The other activities include model storming, reviews, and implementation. The time indicated in each box represents the length of an average session.

2.2 eXtreme Programming

There are several common misconceptions that people seem to have regarding modeling on an XP project. The three most common misconceptions are that software designers:
- don't model
- don't document
- if they do model, only use modeling artifacts of UML

2.2.1 Modeling

User stories are a fundamental aspect of XP and artifacts such as Class Responsibility Collaborator (CRC) cards are common to XP efforts. User stories provide a high-level overview of the require-

ments for a system – they are reminders to have a conversation with your project stakeholders regarding their requirements – and are used as a primary input in estimating and scheduling, and drive the development of acceptance test cases. CRC cards are used to explore structure, perhaps for conceptual modeling to understand the problem domain or for design to work through the structure of your software. User stories and CRC cards are both models so therefore modeling is clearly a part of XP. XP developers will also create sketches, often on a whiteboard or a piece of paper, whenever user stories and CRC cards aren't the best option. In Extreme Programming Explained, the first book written about XP, Kent Beck includes hand-drawn sketches of class diagrams and other free-form diagrams. In fact, in the second edition he includes a mind map in the inside cover overviewing XP. The bottom line is that modeling is a fundamental aspect of XP, something that we will explore in detail in this article.

2.2.2 Documentation

Documentation is also an important part of XP. Ron Jeffries offers the following advice:

"Outside your extreme programming project, you will probably need documentation: by all means, write it. Inside your project, there is so much verbal communication that you may need very little else. Trust yourselves to know the difference."

There are several interesting implications of that statement. First and foremost, the XP community recognizes that documentation should be produced for people external to your team, people that AM would term project stakeholders. Second, it points out that verbal communication between team members reduces the need for documentation within the team. This is the result of project team members being co-located, making communication easier, as well as aspects of XP such as Pair Programming and Collective Ownership that promote communication between developers. Third, it recognizes that sometimes you do, in fact, need internal documentation for your team. This is consistent with the advice presented in Extreme Programming Installed where the authors point out that information resulting from conversations with your project stakeholders regarding user stories are captured as additional documentation attached to the card. Fourth, it suggests that XP team members should know when documentation is required and be allowed to act accordingly. Fifth, it implies that you should trust the team and give them control over their own destiny. This can be hard in many organizations. If the team is untrustworthy then you have a serious problem that needs to be dealt with. This is true regardless of whether they are following XP, or if they are trustworthy but your organizational culture doesn't allow you to act based on that trust then once again you have a serious problem to deal with. Another problem is that when you are an outsider to an XP team, when you haven't been actively involved in the conversations and interactions that have replaced the need for documentation, that it appears that there isn't enough documentation. When this is the case, instead of forcing the team to write documentation instead invest the time to determine if they need the documentation that you believe is missing – suggest the documentation to the team, and if there is an actual need for it then they'll create it. As Ron Jeffries likes to say, "It's called Extreme Programming not stupid programming". Finally, the most important implication for XP teams is that if you need documentation then write it.

The need for documentation on an XP project is reduced by several of its practices. First, because of test-first development and a focus on acceptance testing there is always a working test suite that shows that your system works and fulfills the requirements implemented to that point. For the developers, these tests act as significant documentation because it shows how the code actually works. When you think about it, this makes a lot of sense. When you are learning something new, do you prefer to read a bunch of documentation or do you look for source code samples? Many developers prefer to start at source code samples, and the test suite provides these samples. Second, XP's focus on simplicity and practice of refactoring result in very clean and clear

code. If the code is already easy to understand, why invest a lot of time writing documentation to help you to understand it? This applies to both internal and external documentation – why add comments to code that is already clear and unambiguous? If the code isn't so, then refactor it to improve its quality or as a last resort write documentation. Even though some development environments make it easy to include documentation in your code, Java's Javadoc utility is such an example, you only want to invest in documentation when it makes sense to do so and not just because it is easy.

What confuses many people regarding XP and documentation is that XP doesn't specify potential documents to create during development. This is unlike the RUP which suggests a slew of potential project artifacts. Instead, the suggestion is to work together with your project stakeholders in an environment of rapid feedback and trust them to determine the things that they need, not just documents but any type of project enhancements. Once again, you need to have the courage to trust the people involved with the project.

One of the greatest misunderstandings people have about XP regards concept of traveling light – many people believe that it means you don't create any documentation, but nothing could be further from the truth. What traveling light actually means is that you create just enough models and documentation, too little or too much puts you at risk. A good rule of thumb to ensure that you're traveling light is that you shouldn't create a model or document until you actually need it – creating either thing too early puts you at risk of wasting your time working on something you don't actually need yet. An important thing to understand about documentation on an XP project is that it is a business decision, not a technical one. This is consistent with AM's philosophy regarding documentation. Jeffries says it best:
"If there is a need for a document, the customer should request the document in the same way that she would request a feature: with a story card. The team will estimate the cost of the document, and the customer may schedule it in any iteration she wishes."

2.3 Unified Process

All efforts, including modeling, are organized into disciplines (formerly called workflows) in the UP and are performed in an iterative and incremental manner. The lifecycles of the AUP are presented in Figure 4. The AUP is a subset of the RUP and the EUP a superset of the it. The UP is serial in the large and iterative in the small. The six phases of the EUP clearly occur in a serial manner over time. At the beginning of an UP project your focus is on project initiation activities during the Inception phase. Once your initial scope is understood your major focus becomes requirements analysis and architecture evolution during the Elaboration phase. Your focus shifts to building your system during the Construction phase. You deliver your software during the Transition phase. You operate and support your software in the Production phase, and finally you remove it from production during the Retirement phase. However, on a day-to-day basis you are working in an iterative manner, perhaps doing some modeling, some implementation, some testing, and some management activities.

In the RUP there are three disciplines that encompass modeling activities for a single project – Business Modeling, Requirements, and Analysis & Design – and the EUP adds Enterprise Business Modeling and Enterprise Architecture. The AUP on the other hand, being a subset of the RUP, combines the three modeling disciplines into a single Model discipline.

2.4 Feature-Driven Development

Feature-Driven Development (FDD) is a client-centric, architecture-centric, and pragmatic software process. The term "client" in FDD is used to represent what AM refers to as project stakeholders

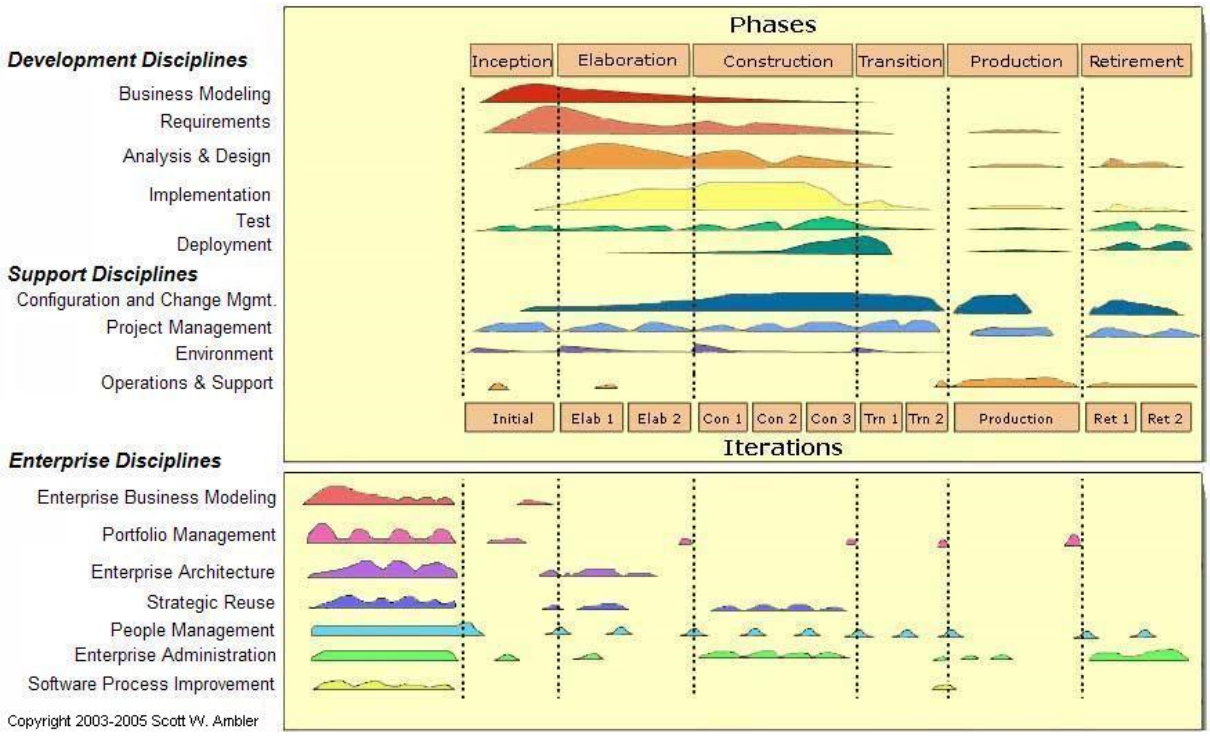

Figure 4:

or XP calls customers.

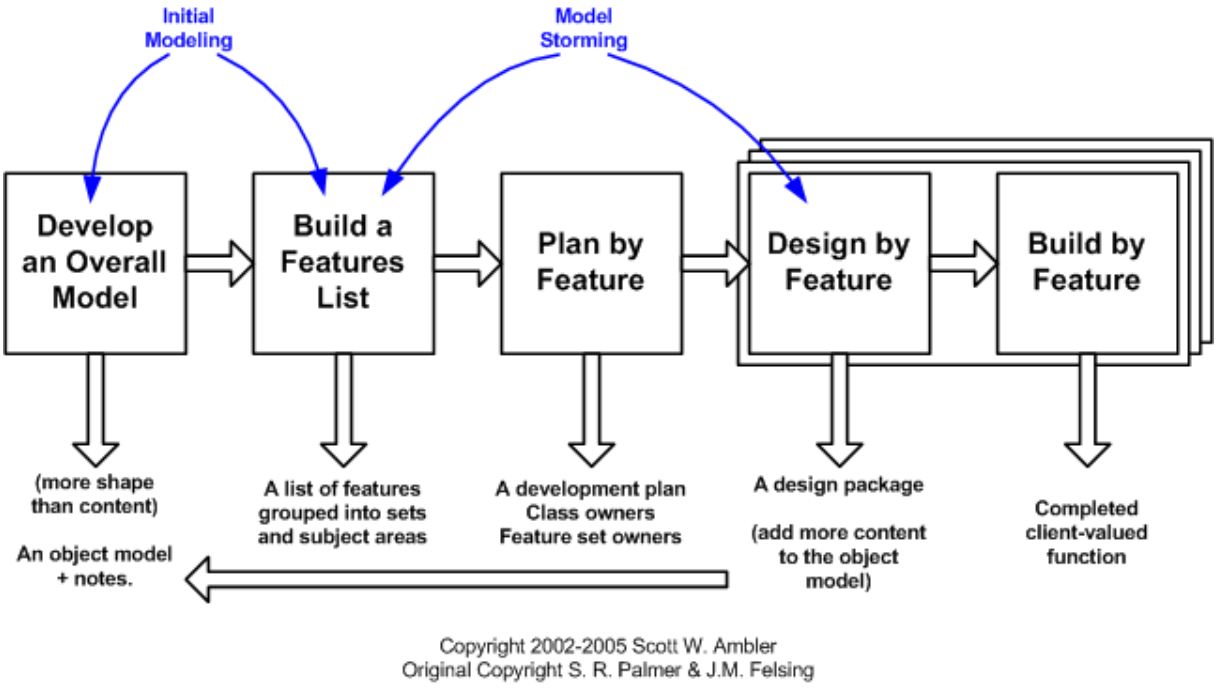

Figure 5: AgileModelingFig5.eps about here.

As the name implies, features are an important aspect of FDD. A feature is a small, client-valued function expressed in the form <action><result><object>. Features are to FDD what use

cases are to the Rational Unified Process and user stories are to XP – they're a primary source of requirements and the primary input into your planning efforts.

As you see in Figure 5 there are five main activities in FDD that are performed iteratively. The first is Develop An Overall Model, the initial result being a high-level object model and notes. At the start of a project your goal is to identify and understand the fundamentals of the domain that your system is addressing, and throughout the project you will flesh this model out to reflect what you're building. The second step is Build A Features List, grouping them into related sets and subject areas. These first two steps map to the Initial Modeling effort of AMDD. Next you Plan By Feature, the end result being a development, the identification of class owners (more on this in a minute), and the identification of feature set owners. The majority of the effort on an FDD project, roughly 75%, is comprised of the fourth and fifth steps: Design By Feature and Build By Feature. These two activities are exactly what you'd expect, they include tasks such as detailed modeling, programming, testing, and packaging of the system.

An FDD project starts by performing the first three steps in the equivalent of the RUP's Inception phase or XP's "iteration 0", the goal being to identify the scope of the effort, the initial architecture, and the initial high-level plan. Construction efforts occur in two-week (or less) iterations, similar to XP, although this can be a little extreme for most RUP teams, with the team iteratively working through all five steps as needed. As with other agile software development processes, systems are delivered incrementally by FDD teams.

3 Getting the Best Results Using Agile Modelling

[Source: `Agile Design Tips`]

3.1 How to Save Time in Modelling

Design models that are just barely good enough. You don't need to get a fully documented set of models in place before you can begin coding. Then iterate, iterate, iterate. 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.

For a given model, use the simplest tool that will fulfil the modelling requirement, but remember that sometimes the simplest tool is a complex CASE tool. For requirements, inclusive tools such as paper and whiteboards may be best; whereas for design, sophisticated tools which (re)generate code may be more efficient.

Effective developers make use of multiple models, and apply the right model(s) for the job at hand. Moreover, each model can be used for a variety of purposes. For example, UML class diagram can be used to depict a high-level domain model or a low-level design. Take advantage of all the flexibility offered by the models you know.

3.2 How to Save Time in Documentation

Unit tests can be used to form much of the detailed design documentation. With a test-driven development approach to development such as XP, unit tests not only validate project code, but also form the majority of the design documentation.

When dealing with a part of the software that is really complicated, either document it thoroughly or redesign it to make it simpler. Do not over-document, remembering the principle, Software Is Your Primary Goal.

3.3 Agile Design Tips

Design is so important designers should do it every day, and not just early in the project before getting to the "real work" of writing the source code. It is good mental exercise to think through how you're going to build something, to actually design it, before you build it.

Designers should also code. They should never simply assume that their design works; they must also be able to prove it with code.

Designers should actively seek feedback about the project and be prepared to consider it and act accordingly.

Designers must be judicious in choices that may restrict the implementation environment. Taking advantage of a product (such as a database, operating system, or middleware tool) couples that product to your system and reduces its portability.

4 Sources

1. http://www.agilemodeling.com
2. http://www.borland.com
3. http://bdn.borland.com
4. http://www.wikipedia.org
5. http://www.nebulon.com
6. http://www.ibm.com
7. http://www.enterpriseunifiedprocess.com
8. http://www.xprogramming.com
9. http://www.ambysoft.com