Agile Modeling (AM):

There are many situations in which software engineers must build large, business-critical systems. The scope and complexity of such systems must be modeled so that (1) all constituencies can better understand what needs to be accomplished, (2) the problem can be partitioned effectively among the people who must solve it, and (3) quality can be assessed as the system is being engineered and built.

Over the past 30 years, a wide variety of software engineering modeling methods and notation have been proposed for analysis and design (both architectural and component-level). These methods have merit, but they have proven to be difficult to apply and challenging to sustain (over many projects). Part of the problem is the "weight" of these modeling methods. By this I mean the volume of notation required, the degree of formalism suggested, the sheer size of the models for large projects, and the difficulty in maintaining the model(s) as changes occur. Yet analysis and design modeling have substantial benefit for large projects—if for no other reason than to make these projects intellectually manageable. Is there an agile approach to software engineering modeling that might provide an alternative?

Definition: Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. Agile models are more effective than traditional models because they are just barely good, they do not have to be perfect.

Model with a purpose. A developer who uses AM should have a specific goal in mind before creating the model. Once the goal for the model is identified, the type of notation to be used and level of detail required will be more obvious.

Use multiple models. There are many different models and notations that can be used to describe software. Only a small subset is essential for most projects.

"Traveling light" is an appropriate philosophy for all software engineering work. Build only those models that provide value … no more, no less.

Travel light. As software engineering work proceeds, keep only those mod els that will provide long-term value and jettison the rest. Every work product that is kept must be maintained as changes occur. This represents work that slows the team down.

Content is more important than representation. Modeling should im part information to its intended audience. A syntactically perfect model that imparts little useful content is not as valuable as a model with flawed nota tion that nevertheless provides valuable content for its audience.

Know the models and the tools you use to create them. Understand the strengths and weaknesses of each model and the tools that are used to create it.

Adapt locally. The modeling approach should be adapted to the needs of the agile team.

A major segment of the software engineering community has adopted the Unified Modeling Language (UML)16 as the preferred method for representing analysis and design models. The

Unified Process (Chapter 2) has been developed to provide a framework for the application of UML.

Agile Unified Process (AUP)

The Agile Unified Process (AUP) adopts a “serial in the large” and “iterative in the small” [Amb06] philosophy for building computer-based systems. By adopting the classic UP phased activities—inception, elaboration, construction, and transition—AUP provides a serial overlay (i.e., a linear sequence of software engineering activities) that enables a team to visualize the overall process flow for a software project. How   ever, within each of the activities, the team iterates to achieve agility and to deliver meaningful software increments to end users as rapidly as possible

• Modeling. UML representations of the business and problem domains are created. However, to stay agile, these models should be “just barely good enough” to allow the team to proceed.

• Implementation. Models are translated into source code.

• Testing. Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.

• Deployment. Like the generic process activity discussed in Chapters 1 and 2, deployment in this context focuses on the delivery of a software increment and the acquisition of feedback from end users.

• Configuration and project management. In the context of AUP, configuration management addresses change management, risk management, and the control of any persistent work products that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.

• Environment management. Environment management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

Although the AUP has historical and technical connections to the Unified Modeling Language, it is important to note that UML modeling can be using in conjunction with any of the agile process models