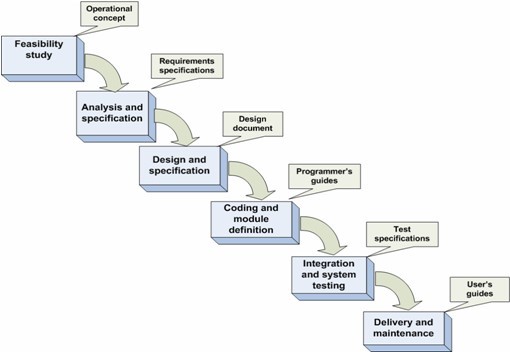
**1. Waterfall Model**

The classic linear-sequential life-cycle model for software engineering is sometimes referred to as the *waterfall model*. The basis of the *waterfall model* originated in the late 1960s to address needs for a complex military software development. The waterfall model consists of a systematic sequential approach to software development that begins at the system level and progresses through several phases or steps. At each phase or step it is recommended that one or more technical documents be produced as products that are completed and "signed off" before continuing on to the next phase.  In 1970, Royce introduced the model following an iterative approach to software development in which he stressed the importance of documentation at each step. In Royce's paper, he also argued that the waterfall model was flawed and needed strengthening by providing additional supporting development steps to make it less risky.

In figure 1, we can trace through the phases of the life cycle and view the major supporting technical documentation proposed by Royce. You should note there can be several documents produced at each phase. We have only mentioned the major documents to help guide you through the phases in the life cycle and to guide you through your course assignment. We will discuss the software requirements specification, design document, and test specifications throughout the subsequent modules in this course.



**Figure 1**

**Waterfall Model and Major Technical Documents**

The lack of feedback between each phase and in getting the approval for each document of the life cycle has proved to be costly and has required a significant amount of rework when changes were introduced during later phases. For these reasons, the waterfall model is best suited for projects in which requirements are stable and well-defined. Introducing changes late in the life cycle is difficult due to the sequential nature and dependency of phases followed in the waterfall model.

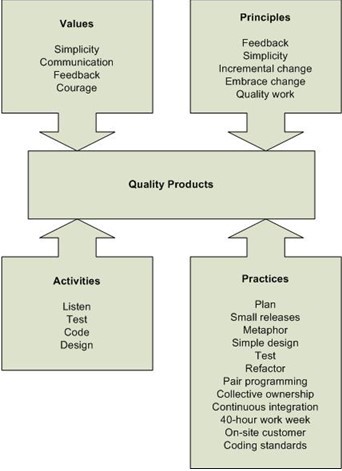
The waterfall model is the oldest and most widely used paradigm in the industry and is known to have inherent deficiencies.

**2. Agile Approach to Software Development**

The agile approach was conceived in response to weaknesses discovered in earlier software development practices and was spurred by the Internet. The growing trend for web applications heavily influenced the need for agility in a development approach. The nature of web programming requires flexibility to adapt and respond to continuous and quick changes in requirements and design. In our discussion on the traditional waterfall model, we mentioned that stability in requirements was necessary, and changes were hard to accommodate. Therefore, this inflexibility to allow change has led researchers to develop a more agile model to accommodate development of applications for the web.

We  will discuss a popular agile process model, Extreme Programming, which can be used for projects that require flexibility to adapt to a changing requirement environment.

Extreme Programming (XP) has been emerging since the late 1990s as a proposed solution for small to medium-sized teams developing projects with continuously changing requirements. The basic philosophy of XP is built from a set of values, principles, practices, and activities (Baird, 2002) that are outlined in figure 2 to produce high-quality software products.



**Figure 2**

**Extreme Programming Philosophy**

In figure 2 we can see a set of values that is used to establish the overall tone throughout the XP development process. The focus of XP is to keep the design and structure as simple as possible to avoid adding unnecessary features to the software and to limit documentation. Because support documentation is kept at a minimum, it's necessary to establish two-way communication between all involved parties. It is preferred that the customer be located on-site so that the means of communication can be oral. Informal means of communications, such as daily stand-up meetings, are used to transfer information between parties. Written documents, reports, and plans are kept at a minimum to save time and effort. Quick and constructive feedback from the customer is encouraged, which builds confidence among the team members. The confidence level of the team fosters quick redevelopment when needed to keep activity moving forward.

The strength of the XP methodology is found in several principles that are used to direct developmental activities. We mentioned feedback and simplicity in XP values but now we will extend these to include short feedback loops and to limit the design to a current iteration. The way in which changes in requirements are handled is the most notable difference in XP as compared to traditional approaches to development. XP encourages small and frequent changes to address and resolve problems as they occur. It’s easy to understand how this philosophy plays an important role in developing applications for the web, because the Internet environment is constantly changing. The principles of XP contrast with the traditional philosophies but the focus is the same – to produce a quality product. XP focuses on quality of work by concentrating on the coding and testing activities.

We will now turn the focus of our discussion to how these values and principles are implemented in the actual activities of the life cycle. The basis for all practices in XP is verbal communications; practitioners therefore must master good listening skills. Another interesting concept in XP is on testing activities. Test cases are continually written as the requirements emerge before any code is actually developed. This school of thought is based on building quality into the code instead of catching errors at a later stage in development. Another prominent feature of XP is the pair programming concept. Two programmers are selected for compatibility and assigned to work on a code module to produce better quality code quickly with fewer errors.

We have finished our discussion on the core of XP methodology. You should now be able to apply the values, principles, and activities as outlined in figure 1.2 to the XP practices. We encourage you to conduct research on the Internet to read more about XP practices and its success stories in the computer industry.

**3. Unified Modeling Language (UML)**

In this section we will take a brief look at the history of the Unified Modeling Language (UML) and how it can be applied to the life cycle phases of software development.

UML has emerged in the software industry and has steadily matured over the decades to become a standard. As a modeling language, UML represents best practices in analysis and design of large-scale software systems.

We can trace the roots of UML to the beginnings of object-oriented languages. With the onset of early object-oriented languages, Stimula-67 and Smalltalk, Objective-C, techniques for object-oriented analysis and design began to appear in software engineering. In the early 1990s, a variety of diagramming techniques began to appear in publications written by common authors Coad, Yourdon, Booch, Rumbaugh, Premerlini, Eddy, Lorensen, Jacobsen, Christerson, Jonsson, and Overgaard.

The Unified Modeling Language provides a definition of a set of elements and a set of rules on how to define these elements to make programs. The elements in UML are defined by using a set of symbols, lines, rectangles, ovals, and other shapes, which are grouped together to create a graphical representation of a conceptual idea. These visual representations increase understanding among all interested parties involved in the design model.

In the current version of UML, there are different types of diagrams that are defined and divided into three major categories: structure, behavior, and interaction.

As we proceed through the phases of the life cycle you will learn and practice with a few of these diagrams. The diagrams are summarized in Table 1 and mapped to the documents where you can use them throughout the life cycle.