2-1 software Engineering: A Layered Technology

Software engineering is a layered technology. Referring to Figure 2.1, any engineering approach (including software engineering) must rest on an organizational commitment to quality.

The foundation for software engineering is the process layer. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products(Models, documents, data, reports, form, etc.) Are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering methods provide the technical how-to’s for building soft-ware. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and sup-port.

Software engineering tools provide automated or semi -automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.



Figure 2-1 Software engineering layers

2-2 the Software Process

A process is a collection of activities, actions, and tasks that are performed when some work product is to be created. In the context of software engineering, a process is not a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

A process framework establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of umbrella activities that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:

1-**Communication:** Before any technical work can commence, it is critically important to communicate and collaborate with the customer to understand stakeholders’ objectives for the project and to gather requirements that help define software features and functions.

2-**Planning**: The planning activity creates a “map” that helps guide the team as it makes the journey. The map defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

3-**Modeling**: A software engineer creates models to better understand software requirements and the design that will achieve those requirements.

4-**Construction**: This activity combines code generation and the testing that is required to uncover errors in the code.

5-**Deployment:** The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

For many software projects, framework activities are applied iteratively as a Project progresses. That is, communication, planning, modeling, construction, and deployment are applied repeatedly through a number of project iterations. Each project iteration produces a software increment that provides stakeholders with subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

Software engineering process framework activities are complemented by a number of umbrella activities. In general, umbrella activities are applied throughout a soft-ware project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

1-***Software project tracking and control***—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

2-***Risk management***—assesses risks that may affect the outcome of the project or the quality of the product.

3-***Software quality assurance***—defines and conducts the activities required to ensure software quality.

4-***Technical reviews***—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

5-***Measurement***—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders’ needs; can be used in conjunction with all other framework and umbrella activities.

6-***Software configuration management***—manages the effects of change throughout the software process. Reusability management—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

7-***Work product preparation and production***—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.



Figure 2-2 the Software Process

A software process can be characterized as shown in Figure 2.2. A common process framework is established by deﬁning a small number of framework activities that are applicable to all software projects.

2-3 the Waterfall Model

It is also referred to as a **linear-sequential life cycle model**. It is very simple to understand and use. In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases. The waterfall Model illustrates the software development process in a linear sequential flow. This means that any phase in the development process begins only if the previous phase is complete. In this waterfall model, the phases do not overlap.

There are separate identified phases in the waterfall model:

1. ***Requirements analysis and definition***: The system’s services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.

2. ***System and software design***: The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.

3. ***Implementation and unit testing***: During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

4. ***Integration and system testing***: The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

5. ***Operation and maintenance:*** Normally (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system’s services as new requirements are discovered.

″ The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. In principle, a phase has to be complete before moving onto the next phase.

**2-4 the Prototypeing Model**

Often, a customer deﬁnes a set of general objectives for software but does not identify detailed input, processing, or output requirements. In other cases, the developer may be unsure of the efﬁciency of an algorithm, the adaptability of an operating system, or the form that human/machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach.

The prototyping paradigm (Figure 2.5) begins with requirements gathering. Developer and customer meet and deﬁne the overall objectives for the software.

A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user.

The prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

Figure 2-3 the prototyping Paradigm

Prototyping can also be problematic for the following reasons:

note: A **mockup** is a visual way of representing a product. Used for teaching, demonstration, design evaluation, promotion, and other purposes. A mockup is a [*prototype*](https://en.wikipedia.org/wiki/Prototype)

**2-5 The incremental Model**

 Incremental development is based on the idea of developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been developed.

The incremental model combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping. Referring to Figure 2.6, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces a deliverable “increment” of the software.

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the ﬁrst increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process ﬂow for any increment can incorporate the prototyping paradigm. When an incremental model is used, the first increment is often a core product.

The incremental process model, like prototyping (Section 2.5) and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment.

Incremental development is particularly useful when stafﬁng is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment.



Figure 2-6 the incremental model

2-6 the Spiral Model

The spiral model was proposed by Boehm (1988). This is shown in Figure 2.11. Here, the software process is represented as a spiral, rather than a sequence of activities with some backtracking from one activity to another. The **spiral model is** similar to the incremental **development** for a system, with more emphasis placed on risk analysis.

* A spiral model is a realistic approach to the development of large-scale software products because the software evolves as the process progresses. In addition, the developer and the client better understand and react to risks at each evolutionary level.
* The model uses prototyping as a risk reduction mechanism and allows for the development of prototypes at any stage of the evolutionary development.
* It maintains a systematic step wise approach, like the classic Life Cycle model but incorporates it into an iterative framework that more reflect the real world.
* If employed correctly, this model should reduce risks before they become problematic as consideration of technical risks are considered at all stages.

It provides the potential for rapid development of incremental versions of the software. Using the spiral model, software is developed in a series of incremental releases. During early iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a number of framework activities, also called task regions. Typically, there are between three and six task regions. Figure 2.7 depicts a spiral model that contains six task regions:

• ***Customer communication***—tasks required to establish effective communication between developer and customer.

• ***Planning***—tasks required to deﬁne resources, timelines, and other project-related information.

• ***Risk analysis***—tasks required to assess technical and management risks.

• ***Engineering***—tasks required to build one or more representations of the application. This phase performed development and **testing** where actual work product made. The deliverables for the engineering phase will be source code, design documents, test cases, test summary.

• ***Construction and release***—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.



Figure 2-7 the spiral model

**Advantages of Spiral Development Model**

* Spiral Model mostly concentrates on risk analysis.
* Most useful for large and risk projects.
* Spiral Model used if requirement changing frequently.
* Focused model for all phases.
* Customer evaluation phase made this model useful.

**Disadvantages of Spiral Development Model**

* For risk, analysis phase required an expert person to make an analysis.
* Not useful for small projects.
* Project duration and cost could be infinite because of the spiral feature.
* Documentation could be lengthy.

**When to use Spiral Model:**

When creation of a Prototype is appropriate. When costs and risk evaluation is important. For medium to high-risk projects. Long-term project commitment unwise because of potential changes to economic priorities. Users are unsure of their needs. Requirements are complex. Significant changes are expected (research and exploration).