CHAPTER 3

PROCESS MODEL DEVELOPMENT

3.1 APPROACHES FOR SOFTWARE PROCESS MODELS

Over the years, a variety of software process models have been designed to structure the software systems construction process. Software development is a conjunction of organizational environment, social environment and technological environment. Usually we go about a predictable process, making it a basis for our project designing and working. But many a times, it is not possible to follow those predictable processes. It is seen that people who work on methodology are not very good at identifying the boundary conditions, which describes the methodology as appropriate or inappropriate. Thus people may use a methodology in the wrong circumstances, such as using a predictable methodology in a unpredictable situation.

Although the most popular waterfall model is often considered as old fashioned, still it is a reasonable choice when requirements are very well understood. But disadvantage is that real projects rarely follow the sequential flow that the model proposes. This model requires all requirements explicitly, which is often difficult for the customer to state explicitly. Also a working version of the program is not available until late in the project’s time-span. This model is time consuming and has little room for iteration apart from being difficult to respond to changes as it does not allow much revision. Once an application is in the testing stage it is difficult to go back and change. [99, 100]
Another popular development methodology is spiral. The approach it uses is quite easy to explain to the user. On the other hand, it is highly customized thus limiting the preferred concept of re-usability. This approach is applied differently for each application and has risk of not meeting the budget or the schedule. This model is intended for large, expensive and complicated projects. It demands considerable expertise in risk evaluation and reduction. This approach is quite complex and relatively difficult to follow strictly.

Usually process models are only concerned with the control-flow aspects of a process, and neglect other equally important aspects. A process model is intended to provide an aggregated view of an organization by bringing together different organizational aspects in a consistent manner. The order of the tasks are defined and other equally important aspects are also considered which mainly include the organizational resources (humans or machines or applications) participating in the process, and the objects (software) that are produced and consumed by the process. The lack of techniques for representing variations in the other process aspects lead to limited expressive power. Firstly, it is not possible for software process modelers to capture the unavailability of given entities in the process, such as human resources or physical artifacts. Secondly, it is not possible to understand the impact that it can have on the tasks of the process. Despite the fact that these situations occur quite often in reality, there is currently no technique to capture them in process models.

Process models can differ from each other depending on a number of factors. They can differ in the order and type of their activities, in the involved resources (human) and business objectives. Process configuration deals with the capturing and managing variations among the process models. [101, 102, 103]

In general some factors that influence the process models and enhance the capabilities of the processes are as follows:
- to be able to reuse and manage the process assets
- to improve the efficiency of using the processes
- to reduce costs in software process programs
- to work collaboratively in the phases of software projects
- to use as few elements in the model as possible
- to minimize the routing paths per element
- to use one start and one end event
- to model as structured as possible
- to avoid OR routing elements

Once the purpose and specifications of the software are determined, the software developers start design in order to develop a plan for a solution. Software design is a process of problem solving and planning for a software solution. It includes low-level component and algorithm implementation as well as the architectural view.

The design concepts help the software designer to implement more sophisticated methods. Fundamental design concepts, which have evolved, are as follows:
1. **Abstraction** - It is the process or result of generalization by reducing the information content of a concept, in order to retain only the relevant information.

2. **Refinement** - It is the process of elaboration. A hierarchy is developed by decomposing a statement of function stepwise until programming language statements are reached. In each step, instructions of a given program are decomposed into more detailed instructions. Abstraction and refinement are complementary concepts.

3. **Modularity** - Software architecture is divided into the components called modules.

4. **Software Architecture** - It refers to the overall structure of the software. A good software architecture will yield better performance, good quality, on-time schedule and effective cost.

5. **Control Hierarchy** – It is a program structure that represent the organization of a program components and implies a hierarchy of control.

6. **Structural Partitioning** - The program structure can be divided both horizontally and vertically. Horizontal partitions define separate branches of modular hierarchy for each major program function. Vertical partitioning suggests that control and work should be distributed top down in the program structure.

7. **Data Structure** - It is a representation of the logical relationship among individual elements of data.
8. **Software Procedure** - It focuses on the processing of each modules individually.

9. **Information Hiding** - Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for that information.

### 3.2 MODEL DEVELOPMENT AND THE EXECUTION PROCESS

Structured planning and execution of the activities is required for model development and simulation. It is called the Model Development and Execution Process. There are highly specialized model development processes for different application areas and different types of basic processes have also evolved. By and large the purpose of a process description is to support structuring, organizing, and managing the products and activities within the process. Stages of model development are identified and associated with intermediate or interim products that result from each stage. The diverse information about the model available at these stages provides additional insight into the final product, the model, and the simulation results. To increase the credibility of models and simulation results in most effective and efficient manner, the model development process should embed the quality assurance measures and verification and validation (V&V) activities.

Model is useful in understanding and changing a system, application, or a component. A model helps to visualize how the system works, to clarify users need, define the architecture of the system, analyze the code, and ensure that the code meets the requirements.

Models can help in numerous ways:
• Drawing modeling diagrams helps to clarify the concepts involved in requirements, architecture, and high-level design.

• Working with models can help to expose inconsistencies in requirements.

• Communicating with models help to communicate important concepts less ambiguously than with natural language.

• Models can be used to generate code or other artifacts such as database schemas or documents.

Models can be used in a wide variety of processes.

• **Use Models to Reduce Ambiguity**

Modeling language is less ambiguous than natural language, and it is designed to express the ideas required during software development. If a project has small team and uses agile practices, we can use models to help clarify user stories. In discussions with the customer about their needs, creating a model can generate useful questions much faster, covering a broader area of the product, than writing spike or prototype code. If the project is large and includes teams in different parts of the world, we can use models to help communicate the requirements and architecture much more effectively than we can in plain text. Creating a model almost always results in a significant reduction in inconsistencies and ambiguities.

• **Use Models with Other Artifacts**
A model by itself is not a requirements specification or architecture. It is a tool for expressing some aspects more clearly but all the concepts required during software design cannot be expressed. The models should therefore be used together with other means of communication, such as OneNote pages or paragraphs, Microsoft Office documents, work items in Team Foundation, or sticky notes on the project room wall.

- **Use Models in Iteration Planning**

Although all projects are different in their scale and organization, a typical project is planned as a series of iterations of between two and six weeks. It is important to plan enough iterations to allow feedback from early iterations to be used to adjust the scope and plans for later iterations. [104]

3.2.1 Modeling activities and intermediate products

The activities carried out during model development can be organized in phases, according to their contents and chronological occurrence. Numerous model development processes exist, which can be distinguished by the selection of phases and the way in which the feedback between the phases is organized. Not only the iteration of development phases, but also the resolution or level of detail of the activity descriptions making up each phase can vary. This mainly depends on the architect of the model development process and the aspects of model development that seem to be most important to the particular group.

To yield meaningful results it is essential that:

- purpose of modeling and the application of simulation results is communicated correctly and completely, and is documented precisely
the real system subjected to modeling is analyzed, abstracted, and idealized. The basic purpose should be considered with well-defined requirements, and all its constraints should be documented and limitations should be formulated explicitly.

- the Conceptual Model of the real system is specified completely and unambiguously using formal and/or mathematical methods as a Formal Model.

- the Executable Model that will actually substitute the real system for experimentation should be free of implementation errors and should correctly implement the Formal Model.

- the experimental application of the Executable Model is conducted with suitable input data, taking care of the limitations.

The analysis of the real system and the final creation of the model are time and cost intensive, specially for large-scale systems. To avoid mistakes during the development of the model, the development process must be manageable, which requires clear structuring of all the developmental activities. Development of the model requires clear and unambiguous specification and documentation of the processes, which an organization should follow for the development of a software product.

The different activities executed during the stepwise development of the model are summarized according to their aim and chronological order, thus allowing the decomposition of model development into phases. In the literature [91], numerous representations of model development processes exist, which can be distinguished by the way in which activities are mapped to phases, by the knowledge required to
perform those activities, the types of feedback loops reflecting the possible repetition of particular phases (iteration), and by the contents of the intermediate products constituting the result of the developmental work of each phase.

The assessment of the phase-wise decomposition of model development varies depending on the perceived relative importance of particular activities. There are several classes of process models for the development of models, software products, or simulation results, which include the following:

- Waterfall processes
- V-type processes
- Iterative and Incremental processes
- Spiral processes
- Agile processes
- Code and Fix processes

The model development process itself is a model of model development; there is no universally accepted presentation of the model development process. When choosing a model development process, one is responsible for choosing a process most suitable for the intended purpose. The above different classes of process models are presented, which are usually used or may be used in modified form for the description of the model development. [91, 105]
3.2.2 Waterfall processes

Waterfall processes are linear representations of model development with a very limited number of feedback loops. A sample waterfall process for software development is provided by Boehm, Shannon and in given in IEEE 1516 2001.

Model development processes of the waterfall type are clear, comprehensive, and easy to handle. The unambiguous separation between the phases theoretically allows a clear assignment of activities to phases. Though, in reality such an idealized flow in model development is rare, that leads to the conclusion that the waterfall model is not suitable to describe the real process of model development as it is. This type of idealized description is considered as a desirable goal. [91]

In waterfall processes, we proceed from one phase to the next in a sequential manner. First complete the requirements specifications and when the requirements are fully completed, then proceed to design. The software is designed and a blueprint is drawn for implementers who are usually the designers or the programmers, who follow this design as a plan for implementing the requirements given. When the design is fully completed, the designers or the programmers make an implementation of that design. At the later stages of the implementation phase, distinct software components produced are combined to introduce new functionality and remove bugs. Thus waterfall model maintains that one should move to a phase only when the preceding phase is completed.

A bug found in the early stages of the software lifecycle (e.g. in requirements specification or design stage) is cheaper, in terms of money, effort and time, to fix than the same bug found later on in the process. A defect that is left until construction or maintenance will cost 50 to 200 times as much to fix, as it would have cost to fix at
requirements time. A waterfall model places emphasis on documentation such as SRS. [106,107]

Entrance/advancement criteria for each phase are:

- Document driven
- Non-overlapping phases
- Product visibility only at end of process. Lack of user feedback
- Brittle in the face of requirements change

3.2.3 V-Type processes

The “V-Style” model development processes focus on direct mirroring of its early and late phases. In addition to the direct steps back to the predecessor phase, and also include special feedback loops to associated earlier phases. A sample of V-type IT system development process is provided in the text written by Bundesministerium für Verteidigung 1997.

This model has clear separation of the development phases that leads to a high degree of clarity of model development processes. Theoretically, a clear assignment of activities to each phase is possible. This form shows the dependencies of the contents of the phases. Even though the feedback loops are numerous, additional feedback loops can be ignored.

The V-model represents a software development process that may be considered an extension of the waterfall model. Instead of moving down in a linear way, the process steps are bent upwards after the coding phase, to form the typical V shape. The V-Model demonstrates the relationships between each phase of the development life
cycle and its associated phase of testing. The horizontal and vertical axes represent time or project completeness and level of abstraction, respectively. [110]

Just like the waterfall model, the V-Shaped life cycle is a sequential path of execution of processes. Each phase must be completed before the next phase begins. Testing is emphasized in this model more so than the waterfall model though. The testing procedures are developed early in the life cycle before any coding is done, during each of the phases preceding implementation.

Like the waterfall model this model begins with the requirements. Before the development starts, a system test plan is created. The test plan focuses on meeting the functionality specified in the requirements gathering.

The high-level design phase focuses on system architecture and design. An integration test plan is created in this phase in order to test the software systems ability to work together. The low-level design phase is where the actual software components are designed, and unit tests are created in this phase. The implementation phase is, where all coding takes place. Once coding is complete, the path of execution continues up the right side of the V where the test plans developed earlier are now put to use. [108]

3.2.4 Iterative and Incremental processes

Iterative and Incremental development starts with an initial planning and ends with deployment with the cyclic interactions in between. Iterative and incremental development are essential parts of the Rational Unified Process, Extreme Programming and various agile software development frameworks.
The system is developed through repeated cycles (iterative) and in smaller portions at a time (incremental), allowing software developers to take advantage of what was learned during development of the versions of the system. Knowledge comes from the development and use of the system, where the process starts with a simple implementation of a subset of the software requirements and iteratively enhances the emergent versions until the full system is implemented. At each iteration, design modifications are made and new functional capabilities are added.

The procedure or process consists of the initialization step, the iteration step, and the Project Control List. The initialization step creates a basic version of the system. The goal of initial implementation is to create a product to which the user can respond. It should offer a model of the key aspects of the problem and provide a solution that is simple to understand and implement easily. To conduct the iteration process, a project control list is created that contains a record of all tasks that need to be performed. It includes the new features to be implemented and areas of redesign of the existing solution. The control list is constantly revised according to the analysis phase.

The iteration involves the redesigning and implementation of task from the project control list, and the analysis of the current version of the system. The aim of design and implementation of any iteration is to be simple, modular, supporting redesign or add a task to the project control list. The analysis of iteration is based upon the user feedback, and the program analysis facilities available. It involves analysis of the structure, modularity, usability, reliability, efficiency, and achievement of the goals. The project control list is modified in accordance with the analysis results.

Contrast with the waterfall development is that the waterfall development completes the project-wide work-products of each discipline in one step before moving on to the next discipline in the next step. Business value is delivered all at once, and only at the end of the project. Whereas in an iterative approach the backtracking is possible.
Internal software development prefers the iterative and incremental approach as:

1. Design and development phase activities are managed using an iterative, incremental approach.

2. Software development that is, creation of the software, is a design and development phase activity.

3. The ability to turn a software design (source code) into working software by compiling and linking it extensively reduces the costs (in terms of both, time and money) of an iteration. [110, 111, 112]

3.2.5 Spiral processes

Spiral model discards the separation between phases of model development. Intermediate products associated with the end of a phase are not demanded, nor are intermediate products are defined. They support the visualization of iteration cycles and the chronological order of modeling activities. Without phases and with iteration cycles there are no feedback loops required. The phase-less incremental spiral model is a very realistic representation of the model development process, as it currently seems to be conducted in most enterprises in the USA and Europe.

The lack of clarity of mandatory intermediate products makes this process model unsuitable for conceptual and theoretical considerations with respect to quality assurance (QA) and integration of verification and validation (V&V) activities. [114]

The spiral model is a software development process that combines the elements of the design and prototyping in stages, in order to combine advantages of top-down and
bottom-up concepts. Barry Boehm in his article A Spiral Model of Software Development and Enhancement, 1985, defined the spiral model. This model was not the first model to discuss iterative development, but it was the first model to explain why the iteration matters. Originally the iterations were typically 6 months to 2 years long.

Each phase starts with a design goal and ends with the client reviewing the progress up to now. Analysis and engineering efforts are applied at each phase of the project, keeping in mind the end goal of the project.

The Spiral model is used most often in large projects and needs constant review to maintain the target. For smaller projects, the concept of agile software development is becoming a practical alternative.

Advantage of using the Spiral model is that the estimates like budget, schedule etc. get more rational as the work progresses. [113]

3.2.6 Agile processes

Agile software development is a group of software development methodologies based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. The Agile Manifesto introduced the term in 2001.

Agile methodologies use iterations. Small teams work together with stakeholders to define quick prototypes, proof of concepts, or other visual means to describe the problem to be solved. The team defines the requirements for the iteration, develops the code, and runs the integrated test scripts, and the users verify the results. Verification occurs much earlier in the development process than with waterfall
model, this allows stakeholders to check and correct the requirements while it is still relatively easy to change.

There are only Agile teams and the processes we describe as Agile are environments for a team to learn how to be Agile. The way a team works together is far more important than any process. A new process can improve team productivity, whereas enabling the team to work effectively as a cohesive unit can improve productivity by several times.

Agile processes allow for changing requirements throughout the development cycle and stress collaboration between software developers and customers and early product delivery. [115]

One of the most important differences between the agile and waterfall approaches is that waterfall features distinct phase with checkpoints and deliverables at each phase, while agile methods have iterations rather than phases. The output of each iteration is working code that can be used to evaluate and respond to changing and evolving user requirements. Waterfall assumes that it is possible to have perfect understanding of the requirements from the start. But in software development, stakeholders do not know what they want and cannot articulate their requirements. [116,117]

Variations of agile processes are:

- **Extreme Programming (XP)**

  The first Extreme Programming project was started March 6, 1996. Extreme Programming is one of several popular Agile Processes. Extreme Programming emphasizes teamwork. Managers, customers, and developers are all equal partners in a collaborative team. Extreme programming implements a simple, effective
environment enabling teams to become highly productive. The team self-organizes around the problem to solve it as efficiently as possible.

Extreme Programming improves a software project in communication, simplicity, feedback, respect, and courage. Extreme Programmers constantly communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software from the first day. They deliver the system to the customers as early as possible and implement changes as suggested.[110,118]

- **Scrum**

Scrum is an iterative, incremental framework for project management. Although the Scrum approach was originally suggested for managing product development projects, its use has focused on the management of software development projects, and it can be used to run software maintenance teams or as a general project/program management approach.

Scrum is an agile process for software development. With Scrum, projects progress using a series of iterations called sprints. Each sprint is typically 2-4 weeks long. While an agile approach can be used for managing any project, Scrum is ideally suited for projects with rapidly changing or highly emergent requirements.

3.2.7 Code and Fix processes

Code-and-fix rarely produces useful results, as it does not use a specific methodology. It is very dangerous as there is no way to assess progress, quality or risk. Code-and-fix is only appropriate for small throwaway projects like proof-of-concept, short-lived demos or throwaway prototypes.
Strengths of this model are that no time is spent on overheads like planning, documentation, quality assurance, standards enforcement or other non-coding activities and it requires little experience.

Weaknesses of this model are that it has no means of assessing quality or identifying risks. It has no visibility control or resource planning. The fundamental flaws in approach do not show up quickly.

Code-and-fix is only appropriate for small throwaway projects like proof-of-concept, short-lived demos or throwaway prototypes. [120,121]

3.3 DESIGN CONSIDERATIONS

Software design is a process of problem solving and planning for a software solution. Once the purpose and specifications of software are determined, software developers will design to develop a plan for a solution. It includes low-level component and algorithm implementation issues as well as the architectural view. There are many aspects to consider in the design of a software. The model design should be strong enough to contain the basic considerations while developing a system. The importance of each aspects should be taken care of which would reflect the goals the software which are tried to achieve. Some of the aspects which can be considered are:

- **Compatibility** - The software should be able to operate with other products that are designed for interoperability with another product. For example, a software may be backward-compatible with an older version of itself.
- **Extensibility** - New capabilities can be added to the software without major changes to the underlying architecture.
• **Fault-tolerance** - The software is resistant to and able to recover from component failure.

• **Maintainability** - The software can be restored to a specified condition within a specified period of time. For example, antivirus software may include the ability to periodically receive virus definition updates in order to maintain the software's effectiveness.

• **Modularity** - The resulting software comprises of well defined, independent components. It leads to better maintainability. The components could be then implemented and tested in isolation before being integrated to form a desired software system. This allows division of work in a software development project.

• **Packaging** - Printed material such as the box and manuals should match the style designated for the target market and should enhance usability. All compatibility information should be visible on the outside of the package. All components required for use should be included in the package or specified as a requirement on the outside of the package.

• **Reliability** - The software is able to perform a required function under the stated conditions for a specified period of time.

• **Reusability** - The software is able to add further features and modification with slight or no modification.

• **Robustness** - The software is able to operate under stress or tolerate unpredictable or invalid input. For example, it can be designed with a strength to accommodate low memory conditions.

• **Security** - The software is able to withstand hostile acts and influences.

• **Usability** - The software user interface must be usable for its intended audience. Default values for the parameters must be chosen so that they are a good choice for the majority of the users. [123]
3.4 SOFTWARE DESIGN DESCRIPTION

IEEE 1016-1998, also known as the Recommended Practice for Software Design Descriptions, is an IEEE standard that specifies an organizational structure for a software design description (SDD). An SDD is a document which is used to specify system architecture and application design in a software related project.

The software design description helps to understand the basic and essential requirements which should be kept in our mind before considering any factor for the development of a new design of a process. [112,124]

This standard specifies requirements on the information content and organization for Software Design Descriptions (SDDs). An SDD is a representation of a software design that is to be used for recording design information addressing various design concerns and communicating that information to the design’s stakeholders.

SDDs play crucial role in the development and maintenance of software systems. During its lifetime, an SDD is used by acquirers, project managers, quality assurance staff, configuration managers, software designers, programmers, testers, and maintainers. Each of these stakeholders has unique needs, both in terms of required design information and optimal organization of that information. Hence, a design description will contain the design information needed by those stakeholders.

The standard also specifies requirements on the design languages to be used when producing SDDs conforming to these requirements on content and organization. The standard specifies that an SDD be organized into a number of design views. Each view addresses a specific set of design concerns of the stakeholders. Each design view is prescribed by a design viewpoint. A viewpoint identifies the design concerns to be focused upon within its view and selects the design languages used to record that
design view. The standard establishes a common set of viewpoints for design views, as a starting point for the preparation of a SDD, and a generic capability for defining new design viewpoints thereby expanding the expressiveness of an SDD for its stakeholders.

This standard is intended for use in design situations in which an explicit software design description is to be prepared. These situations include traditional software design and construction activities leading to an implementation as well as “reverse engineering” situations where a design description is to be recovered from an existing implementation. This standard can be applied to commercial, scientific, military and other types of software. Applicability is not restricted by size, complexity, or criticality of the software. This standard considers both the software and its system context, including the developmental and operational environment. It can be used where software comprises the system or where software is part of a larger system characterized by hardware, software and human components and their interfaces. This standard is applicable whether the SDD is captured using paper documents, automated databases, software development tools or other media. This standard does not explicitly support, nor is it limited to, use with any particular software design methodology or particular design languages, although it establishes minimum requirements on the selection of those design languages.

This standard is consistent for use with IEEE/EIA Std 12207.0–1996, Software Life Cycle; it may also be applied in other life cycle contexts.
The standard format of a document should contain at least the following chapters:

1. INTRODUCTION
   1. Design Overview
   2. Requirements Traceability Matrix

2. SYSTEM ARCHITECTURAL DESIGN
   1. Chosen System Architecture
   2. Discussion of Alternative Designs
   3. System Interface Description

3. DETAIL DESCRIPTION OF COMPONENTS
   1. Component n
   2. Component n+1

4. USER INTERFACE DESIGN
   1. Description of the User Interface
      1. Screen Image
      2. Objects and Actions

5. ADDITIONAL MATERIAL

In March 2009, the IEEE-SA Standards Board approved a revision to IEEE 1016. The revision upgrades the recommended practice to a full IEEE standard. The standard is titled Standard for Information Technology — Systems Design — Software Design Descriptions. [123 - 133]