Software Engineering Notes

Krishnaveni S
Assistant Professor (SrG)
Department Of Software Engineering
SRM University
INTRODUCTION TO SOFTWARE ENGINEERING

The term *software engineering* is composed of two words, software and engineering. **Software** is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product**.

**Engineering** on the other hand, is all about developing products, using well-defined, scientific principles and methods.

So, we can define *software engineering* as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

IEEE defines software engineering as:

*The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.*

We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are absolutely necessary to achieve a good quality software cost effectively.

Without using software engineering principles, it would be difficult to develop large programs. In industry, it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes. Software engineering helps to reduce this programming complexity. Software engineering principles use two important techniques to reduce problem complexity: *abstraction* and *decomposition*. 

---

**UNIT I**

**INTRODUCTION**

NEED OF SOFTWARE ENGINEERING

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

- **Large software** - It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.

- **Scalability** - If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.

- **Cost** - As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.

- **Dynamic Nature** - The always growing and adapting nature of software hugely depends upon the environment in which the user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.

- **Quality Management** - Better process of software development provides better and quality software product.

**Software Applications**

Software may be applied in any situation for which a prespecified set of procedural steps (i.e., an algorithm) has been defined (notable exceptions to this rule are expert system software and neural network software). Information content and determinacy are important factors in determining the nature of a software application.

**System software:** System software is a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) process complex, but determinate, information structures. Other systems applications (e.g., operating system components, drivers, telecommunications processors) process largely indeterminate data.

**Real-time software:** Software that monitors/analyzes/controls real-world events as they occur is called real time. Elements of real-time software include a data gathering component that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment, and a monitoring component that coordinates all other components so that real-time
response (typically ranging from 1 millisecond to 1 second) can be maintained.

**Business software:** Business information processing is the largest single software application area. Discrete "systems" (e.g., payroll, accounts receivable/payable, inventory) have evolved into management information system (MIS) software that accesses one or more large databases containing business information. Applications in this area restructure existing data in a way that facilitates business operations or management decision making. In addition to conventional data processing application, business software applications also encompass interactive computing (e.g., point of sale transaction processing).

**Engineering and scientific software.** Engineering and scientific software have been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

**Embedded software:** Intelligent products have become commonplace in nearly every consumer and industrial market. Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

**Personal computer software:** The personal computer software market has burgeoned over the past two decades. Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.

**Web Apps (Web applications) network centric software.** As web 2.0 emerges, more sophisticated computing environments is supported integrated with remote database and business applications.

**AI software** uses non-numerical algorithm to solve complex problem. Robotics, expert system, pattern recognition game playing
A GENERIC VIEW OF PROCESS–A LAYERED TECHNOLOGY

• Software engineering encompasses a process, the management of activities, technical methods, and use of tools to develop software products.

• Fritz Bauer defined Software engineering as the “establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines. “

• IEEE definition of software engineering (1) the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

• We need discipline but we also need adaptability and agility.

• Software Engineering is a layered technology as shown below. Any engineering approach must rest on an organizational commitment to quality.

• The bedrock that supports software engineering is a quality focus.

![Layers of Software engineering](image)

- The foundation for software engineering is the process layer. It 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, 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 software. Methods encompass a broad array of tasks that include communication, req. analysis, design, coding, testing and support.
- Software engineering tools provide automated or semi-automated support for the process and the methods.
A PROCESS FRAMEWORK

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.

Software process models can be prescriptive or agile, complex or simple, all-encompassing or targeted, but in every case, five key activities must occur. The framework activities are applicable to all projects and all application domains, and they are a template for every process model.

- Software process
- Process framework
- Umbrella activities
- Framework activity
- Software Engineering action
A GENERIC PROCESS FRAMEWORK FOR SOFTWARE ENGINEERING
ENCOMPASSES FIVE ACTIVITIES:

Communication. Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders). The intent is to understand stakeholders’ objectives for the project and to gather requirements that help define software features and functions.

Planning. Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a “map” that helps guide the team as it makes the journey. The map—called a software project plan—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.

Modelling. Whether you’re a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a “sketch” of the thing so that you’ll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you’re going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction. This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

Deployment. The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.
SOFTWARE PROCESS MODEL (OR) SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle. A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to retirement. Different life cycle models may map the basic development activities to phases in different ways. Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out.

THE NEED FOR A SOFTWARE PROCESS MODEL

The development team must identify a suitable life cycle model for the particular project and then adhere to it. Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner. When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure. This problem can be illustrated by using an example. Suppose a software development problem is divided into several parts and the parts are assigned to the team members. From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part, another might decide to prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him. This would be one of the perfect recipes for project failure. A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied. So without software life cycle model the entry and exit criteria for a phase cannot be recognized. Without software life cycle models, it becomes difficult for software project managers to monitor the progress of the project.
PRESCRIPTIVE MODEL

- Prescriptive process models advocate an **orderly approach to software engineering**
  - Organize framework activities in a certain order
- Process framework activity with set of software engineering actions.
- Each action in terms of a task set that identifies the work to be accomplished to meet the goals.
- The resultant process model should be adapted to accommodate the nature of the specific project, people doing the work, and the work environment.
- Software engineer choose process framework that includes activities like;
  - Communication
  - Planning
  - Modeling
  - Construction
  - Deployment

DIFFERENT SOFTWARE LIFE CYCLE MODELS OR PROCESS MODELS

Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

1. Classical Waterfall Model
2. V Model
3. Iterative Waterfall Model
4. Incremental Model
5. Prototyping Model
6. Evolutionary Model
7. Spiral Model
8. RAD Model

**1. CLASSICAL WATERFALL MODEL**

Sometimes called the classic life cycle or the waterfall model, the linear sequential model suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing, and support. Figure illustrates the linear sequential model for software engineering. Modeled after a conventional engineering cycle, the linear sequential model encompasses the following activities.
**Requirement Gathering:** Because software is always part of a larger system (or business), work begins by establishing requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when software must interact with other elements such as hardware, people, and databases. System engineering and analysis encompass requirements gathering at the system level with a small amount of top level.

**Analysis:** Information engineering encompasses requirements gathering at the strategic business level and at the business area level. Software requirements analysis. The requirements gathering process is intensified and focused specifically on software. To understand the nature of the program(s) to be built, the software engineer ("analyst") must understand the information domain for the software, as well as required function, behaviour, performance, and interface. Requirements for both the system and the software are documented and reviewed with the customer.

**Design:** Software design is actually a multistep process that focuses on four distinct attributes of a program: data structure, software architecture, interface representations, and procedural (algorithmic) detail. The design process translates requirements into a representation of the software that can be assessed for quality before coding begins. Like requirements, the design is documented and becomes part of the software configuration.
**Code generation:** The design must be translated into a machine-readable form. The code generation step performs this task. If design is performed in a detailed manner, code generation can be accomplished mechanistically.

**Testing:** Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, ensuring that all statements have been tested, and on the functional externals; that is, conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results.

**Maintenance:** Software will undoubtedly undergo change after it is delivered to the customer (a possible exception is embedded software). Change will occur because errors have been encountered, because the software must be adapted to accommodate changes in its external environment (e.g., a change required because of a new operating system or peripheral device), or because the customer requires functional or performance enhancements. Maintenance involves performing any one or more of the following three kinds of activities: Correcting errors that were not discovered during the product development phase. This is called corrective maintenance. Improving the implementation of the system, and enhancing the functionalities of the system according to the customer’s requirements. This is called perfective maintenance. Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called adaptive maintenance.

**Advantages**
1. Easy to understand and implement.
2. Widely used and known (in theory!).
3. Reinforces good habits: define-before-design, design-before-code.
4. Works well for projects where requirements are well understood.
5. Clearly defined stages.
6. Easy to manage. Each phase has specific deliverable and a review.

**Disadvantages**
1. The requirements must be stated before beginning the design.
2. There is no overlap between stages
3. The waterfall method does not prohibit returning to an earlier phase, this involves costly rework
4. Because the actual development comes late in the process, one does not see results for a long time
5. Not suitable for projects of long duration because in long running projects requirements are likely to change.
2. V Model

The V-Model is a unique, linear development methodology used during a software development life cycle (SDLC). The V-Model focuses on a fairly typical waterfall method that follows strict, step-by-step stages. While initial stages are broad design stages, progress proceeds down through more and more granular stages, leading into implementation and coding, and finally back through all testing stages prior to completion of the project.

The Process of the V-Model

Much like the traditional waterfall model, the V-Model specifies a series of linear stages that should occur across the life cycle, one at a time, until the project is complete. For this reason, V-Model is not considered an agile development method, and due to the sheer volume of stages and their integration, understanding the model in detail can be challenging for everyone on the team, let alone clients or users.

The V-shape of the V-Model method represents the various stages that will be passed through during the software development life cycle. Beginning at the top-left stage and working, over time, toward the top-right tip, the stages represent a linear progression of development similar to the waterfall model.

Below we’ll briefly discuss each of roughly nine stages involved in the typical V-Model and how they all come together to generate a finished product.

Requirements

During this initial phase, system requirements and analysis are performed to determine the feature set and needs of users. Just as with the same phase from the waterfall model or other similar methods, spending enough time and creating thorough user requirement documentation is critical during this phase, as it only occurs once.

Another component unique to the V-Model is that during each design stage, the corresponding tests are also designed to be implemented later during the testing stages. Thus, during the requirements phase, acceptance tests are designed.
System Design
Utilizing feedback and user requirement documents created during the requirements phase, this next stage is used to generate a specification document that will outline all technical components such as the data layers, business logic, and so on.
System Tests are also designed during this stage for later use.

Architecture Design
During this stage, specifications are drawn up that detail how the application will link up all its various components, either internally or via outside integrations. Often this is referred to as high-level design.

Module Design
This phase consists of all the low-level design for the system, including detailed specifications for how all functional, coded business logic will be implemented, such as models, components, interfaces, and so forth.
Unit tests should also be created during the module design phase.

Implementation/Coding
At this point, halfway through the stages along the process, the actual coding and implementation occur. This period should allot for as much time as is necessary to convert all previously generated design and specification docs into a coded, functional system. This stage should be fully complete once the testing phases begin.

Unit Testing
Now the process moves back up the far side of the V-Model with inverse testing, starting with the unit tests developed during the module design phase. Ideally, this phase should eliminate the vast majority of potential bugs and issues, and thus will be the lengthiest testing phase of the project. That said, just as when performing unit testing with other development models, unit tests cannot (or should not) cover every possible issue that can occur in the system, so the less granular testing phases to follow should fill in these gaps.
ADVANTAGES OF THE V-MODEL

Suited for Restricted Projects: Due to the stringent nature of the V-Model and its linear design, implementation, and testing phases, it’s perhaps no wonder that the V-Model has been heavily adopted by the medical device industry in recent years. In situations where the project length and scope are well-defined, the technology is stable, and the documentation & design specifications are clear, the V-Model can be a great method.

Ideal for Time Management: Along the same vein, V-Model is also well-suited for projects that must maintain a strict deadline and meet key milestone dates throughout the process. With fairly clear and well understood stages that the whole team can easily comprehend and prepare for, it is relatively simple to create a time line for the entire development life cycle, while generating milestones for each stage along the way. Of course, the use of BM in no way ensures milestones will always be met, but the strict nature of the model itself enforces the need to keep to a fairly tight schedule.

DISADVANTAGES OF THE V-MODEL

Lacks Adaptability: Similar to the issues facing the traditional waterfall model on which the V-Model is based, the most problematic aspect to the V-Model is its inability to adapt to any necessary changes during the development life cycle. For example, an overlooked issue within some fundamental system design, that is then only discovered during the implementation phase, can present a severe setback in terms of lost man-hours as well as increased costs.

Timeline Restrictions: While not an inherent problem with the V-Model itself, the focus on testing at the end of the life cycle means that it’s all too easy to be pigeonholed at the end of the project into performing tests in a rushed manner to meet a particular deadline or milestone.

Ill-Suited for Lengthy Life Cycles: Like the waterfall model, the V-Model is completely linear and thus projects cannot be easily altered once the development train has left the station. V-Model is therefore poorly suited to handle long-term projects that may require many versions or constant updates/patches.

Encourages ‘Design-by-Committee’ Development: While V-Model is certainly not the only development model to fall under this criticism, it cannot be denied that the strict and methodical nature of the V-Model and its various linear stages tend to emphasize a development cycle befitting managers and users, rather than developers and designers. With a method like V-Model, it can be all too easy for project managers or others to overlook the vast complexities of software development in favour of trying to meet deadlines, or to simply feel overly confident in the process or current progress, based solely on what stage in the life cycle is actively being developed.
Advantages
1. Simple and easy to use.
2. Each phase has specific deliverables.
3. Higher chance of success over the waterfall model due to the early development of test plans during the life cycle.
4. Works well for small projects where requirements are easily understood.

Disadvantages
1. Very rigid like the waterfall model.
2. Little flexibility and adjusting scope is difficult and expensive.
3. Software is developed during the implementation phase, so no early prototypes of the software are produced.
4. This Model does not provide a clear path for problems found during testing phases.
3. INCREMENTAL PROCESS MODELS

- The iterative model is a particular implementation of a software development life cycle (SDLC) that focuses on an initial, simplified implementation, which then progressively gains more complexity and a broader feature set until the final system is complete. When discussing the iterative method, the concept of incremental development will also often be used liberally and interchangeably, which describes the incremental alterations made during the design and implementation of each new iteration.

The process models in this category tend to be among the most widely used (and effective) in the industry.

- The incremental model combines elements of the waterfall model applied in an iterative fashion. The model applies linear sequences in a staggered fashion as calendar time progresses.
- Each linear sequence produces deliverable “increments” of the software. (Ex: a Word Processor delivers basic file mgmt., editing, in the first increment; more sophisticated editing, document production capabilities in the 2nd increment; spelling and grammar checking in the 3rd increment.
- When an increment model is used, the 1st increment is often a core product. The core product is used by the customer.
- As a result of use and / or evaluation, a plan is developed for the next increment.
- The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.
- The process is repeated following the delivery of each increment, until the complete product is produced.
- If the customer demands delivery by a date that is impossible to meet, suggest delivering one or more increments by that date and the rest of the Software later.
The Process

Unlike the more traditional waterfall model, which focuses on a stringent step-by-step process of development stages, the iterative model is best thought of as a cyclical process. After an initial planning phase, a small handful of stages are repeated over and over, with each completion of the cycle incrementally improving and iterating on the software. Enhancements can quickly be recognized and implemented throughout each iteration, allowing the next iteration to be at least marginally better than the last.

- **Planning & Requirements**: As with most any development project, the first step is go through an initial planning stage to map out the specification documents, establish software or hardware requirements, and generally prepare for the upcoming stages of the cycle.

- **Analysis & Design**: Once planning is complete, an analysis is performed to nail down the appropriate business logic, database models, and the like that will be required at this stage in the project. The design stage also occurs here, establishing any technical requirements (languages, data layers, services, etc) that will be utilized in order to meet the needs of the analysis stage.
- **Implementation**: With the planning and analysis out of the way, the actual implementation and coding process can now begin. All planning, specification, and design docs up to this point are coded and implemented into this initial iteration of the project.

- **Testing**: Once this current build iteration has been coded and implemented, the next step is to go through a series of testing procedures to identify and locate any potential bugs or issues that have cropped up.

- **Evaluation**: Once all prior stages have been completed, it is time for a thorough evaluation of development up to this stage. This allows the entire team, as well as clients or other outside parties, to examine where the project is at, where it needs to be, what can or should change, and so on.

**ADVANTAGES OF THE ITERATIVE MODEL**

- **Inherent Versioning**: It is rather obvious that most software development life cycles will include some form of versioning, indicating the release stage of the software at any particular stage. However, the iterative model makes this even easier by ensuring that newer iterations are incrementally improved versions of previous iterations. Moreover, in the event that a new iteration fundamentally breaks a system in a catastrophic manner, a previous iteration can quickly and easily be implemented or “rolled back,” with minimal losses; a particular boon for post-release maintenance or web applications.

- **Rapid Turnaround**: While it may seem like each stage of the iterative process isn’t all that different from the stages of a more traditional model like the waterfall method — and thus the process will take a great deal of time — the beauty of the iterative process is that each stage can effectively be slimmed down into smaller and smaller time frames; whatever is necessary to suit the needs of the project or organization. While the initial run through of all stages may take some time, each subsequent iteration will be faster and faster, lending itself to that agile moniker so very well, and allowing the life cycle of each new iteration to be trimmed down to a matter of days or even hours in some cases.

- **Suited for Agile Organizations**: While a step-by-step process like the waterfall model may work well for large organizations with hundreds of team members, the iterative model really starts to shine when its in the hands of a smaller, more agile team. Particularly when combined with the power of modern version control systems, a full “iteration process” can effectively be performed by...
a number of individual team members, from planning and design through to implementation and testing, with little to no need for outside feedback or assistance.

- **Easy Adaptability**: Hinging on the core strength of constant, frequent iterations coming out on a regular basis, another primary advantage of the iterative model is the ability to rapidly adapt to the ever-changing needs of both the project or the whims of the client. Even fundamental changes to the underlying code structure or implementations (such as a new database system or service implementation) can typically be made within a minimal time frame and at a reasonable cost, because any detrimental changes can be recognized and reverted within a short time frame back to a previous iteration.

**DISADVANTAGES OF THE ITERATIVE MODEL**

- **Costly Late-Stage Issues**: While not necessarily a problem for all projects, due to the minimal initial planning before coding and implementation begin, when utilizing an iterative model, it is possible that an unforeseen issue in design or underlying system architecture will arise late into the project. Resolving this could have potentially devastating effects on the time frame and costs of the project as a whole, requiring a great deal of future iterations just to resolve one issue.

- **Increased Pressure on User Engagement**: Unlike the waterfall model, which emphasizes nearly all user/client engagement within the initial stages of the project during a brief crunch time period, the iterative model often requires user engagement throughout the entirety of the process. This is sometimes an unfortunate obligation, since each new iteration will likely require testing and feedback from users in order to properly evaluate any necessary changes.

- **Feature Creep**: Not only does the iterative model require user feedback throughout the process, but this also inherently means the project may be subject to undesired feature creep, whereby users experience the changes in each iteration, and are inclined to constantly put forth new requests for additional features to be added to future versions.
4. ITERATIVE DEVELOPMENT

- The project is divided into small parts allows the development team to demonstrate results earlier on in the process and obtain valuable feedback from system users, each iteration is actually a mini-Waterfall process with the feedback from one phase providing vital information for the design of the next phase.

- Iterative process starts with a simple implementation of a subset of the software requirements and iteratively enhances the evolving versions until the full system is implemented. At each iteration, design modifications are made and new functional capabilities are added. The basic idea behind this method is to develop a system through repeated cycles (iterative) and in smaller portions at a time (incremental).

- The following illustration is a representation of the Iterative and Incremental model – Iterative and Incremental development is a combination of both iterative design or iterative method and incremental build model for development. "During software development, more than one iteration of the software development cycle may be in progress at the same time." This process may be described as an "evolutionary acquisition" or "incremental build" approach.

- In this incremental model, the whole requirement is divided into various builds. During each iteration, the development module goes through the requirements, design, implementation and testing phases. Each subsequent release of the module adds function to the previous release. The process continues till the complete system is ready as per the requirement.

- The key to a successful use of an iterative software development lifecycle is rigorous validation of requirements, and verification & testing of each version of the software against those requirements within each cycle of the model. As the software evolves through successive cycles, tests must be repeated and extended to verify each version of the software.
Iterative Model - Application

- Like other SDLC models, Iterative and incremental development has some specific applications in the software industry. This model is most often used in the following scenarios –
- Requirements of the complete system are clearly defined and understood.
- Major requirements must be defined; however, some functionalities or requested enhancements may evolve with time.
- There is a time to the market constraint.
- A new technology is being used and is being learnt by the development team while working on the project.
- Resources with needed skill sets are not available and are planned to be used on contract basis for specific iterations.
- There are some high-risk features and goals which may change in the future.

Advantages

- Results are obtained early and periodically
- Progress can be measured
- Less costly to change the scope/requirements.
- With every increment, operational product is delivered.
- Testing and debugging during smaller iteration is easy

Disadvantages

- More resources may be required
- More management attention is required.
- Defining increments may require definition of the complete system.
- Each phase of an iteration is rigid with no overlaps.
5. PROTOTYPE MODEL

- The Software Prototyping refers to building software application prototypes which displays the functionality of the product under development, but may not actually hold the exact logic of the original software.

- Software prototyping is becoming very popular as a software development model, as it enables to understand customer requirements at an early stage of development. It helps get valuable feedback from the customer and helps software designers and developers understand about what exactly is expected from the product under development.

**What is Software Prototyping?**

- Prototype is a working model of software with some limited functionality. The prototype does not always hold the exact logic used in the actual software application and is an extra effort to be considered under effort estimation.

- Prototyping is used to allow the users evaluate developer proposals and try them out before implementation. It also helps understand the requirements which are user specific and may not have been considered by the developer during product design.

Following is a stepwise approach explained to design a software prototype.

**Basic Requirement Identification**

- This step involves understanding the very basics product requirements especially in terms of user interface. The more intricate details of the internal design and external aspects like performance and security can be ignored at this stage.

**Developing the initial Prototype**

- The initial Prototype is developed in this stage, where the very basic requirements are showcased and user interfaces are provided. These features may not exactly work in the same manner internally in the actual software developed. While, the workarounds are used to give the same look and feel to the customer in the prototype developed.

**Review of the Prototype**

- The prototype developed is then presented to the customer and the other important stakeholders in the project. The feedback is collected in an organized manner and used for further enhancements in the product under development.
Revise and Enhance the Prototype

- The feedback and the review comments are discussed during this stage and some negotiations happen with the customer based on factors like – time and budget constraints and technical feasibility of the actual implementation. The changes accepted are again incorporated in the new Prototype developed and the cycle repeats until the customer expectations are met.

Software Prototyping - Types

- There are different types of software prototypes used in the industry. Following are the major software prototyping types used widely –

  **Throwaway/Rapid Prototyping**

- Throwaway prototyping is also called as rapid or close ended prototyping. This type of prototyping uses very little efforts with minimum requirement analysis to build a prototype. Once the actual requirements are understood, the prototype is discarded and the actual system is developed with a much clear understanding of user requirements.

  **Evolutionary Prototyping**

- Evolutionary prototyping also called as breadboard prototyping is based on building actual functional prototypes with minimal functionality in the beginning. The prototype developed forms the heart of the future prototypes on top of which the entire system is built. By using evolutionary prototyping, the well-understood requirements are included in the prototype and the requirements are added as and when they are understood.

  **Incremental Prototyping**

- Incremental prototyping refers to building multiple functional prototypes of the various sub-systems and then integrating all the available prototypes to form a complete system.

  **Extreme Prototyping**

- Extreme prototyping is used in the web development domain. It consists of three sequential phases. First, a basic prototype with all the existing pages is presented in the HTML format. Then the data processing is simulated using a prototype services layer. Finally, the services are implemented and integrated to the final prototype. This process is called Extreme Prototyping used to draw attention to the second phase of the process, where a fully functional UI is developed with very little regard to the actual services.
Software Prototyping - Application

- Software Prototyping is most useful in development of systems having high level of user interactions such as online systems. Systems which need users to fill out forms or go through various screens before data is processed can use prototyping very effectively to give the exact look and feel even before the actual software is developed.

- Software that involves too much of data processing and most of the functionality is internal with very little user interface does not usually benefit from prototyping. Prototype development could be an extra overhead in such projects and may need lot of extra efforts.

Software Prototyping - Pros and Cons

- Software prototyping is used in typical cases and the decision should be taken very carefully so that the efforts spent in building the prototype add considerable value to the final software developed. The model has its own pros and cons discussed as follows.

The advantages of the Prototyping Model are as follows –

- Increased user involvement in the product even before its implementation.

- Since a working model of the system is displayed, the users get a better understanding of the system being developed.

- Reduces time and cost as the defects can be detected much earlier.

- Quicker user feedback is available leading to better solutions.

- Missing functionality can be identified easily.

- Confusing or difficult functions can be identified.

The Disadvantages of the Prototyping Model are as follows –

- Risk of insufficient requirement analysis owing to too much dependency on the prototype.

- Users may get confused in the prototypes and actual systems.

- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.

- Developers may try to reuse the existing prototypes to build the actual system, even when it is not technically feasible.

- The effort invested in building prototypes may be too much if it is not monitored properly.
6. RAD Model

- Rapid Application Development (RAD) is an incremental software process model that emphasizes a short development cycle. RAD is a “high-speed” adaptation of the waterfall model, in which rapid development is achieved by using a component based construction approach.
- If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a fully functional system within a short period of time. (e.g., 60 to 90 days)
- Used primarily for information systems applications, the RAD approach encompasses the following phases

**Business modeling.**
The information flow among business functions is modeled in a way that answers the following questions: What information drives the business process? What information is generated? Who generates it? Where does the information go? Who processes it?
**Data modeling.** The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business. The characteristics (called attributes) of each object are identified and the relationships between these objects defined.

**Process modeling.** The data objects defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.

**Application generation.** RAD assumes the use of fourth generation techniques Rather than creating software using conventional third generation programming languages the RAD process works to reuse existing program components (when possible) or create reusable components (when necessary). In all cases, automated tools are used to facilitate construction of the software.

**Testing and turnover.** Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time. However, new components must be tested and all interfaces must be fully exercised.

**What are the drawbacks of the RAD model?**

1. For large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
2. If developers and customers are not committed to the rapid-fire activities necessary to complete the system in a much abbreviated time frame, RAD project will fail.
3. If a system cannot properly be modularized, building the components necessary for RAD will be problematic.

**7. THE SPIRAL MODEL**

The spiral model, originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. 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 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 both technical and management risks.
- **Engineering**—tasks required to build one or more representations of the application.
- **Construction and release**—tasks required to construct, test, install, and provide user support (e.g., documentation and training).
- **Customer evaluation**—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering.
**Advantages**

1. High amount of risk analysis.
2. Good for large and mission-critical projects.
3. Software is produced early in the software life cycle.
4. Changing requirements can be accommodated.
5. Allows extensive use of prototypes.
6. Requirements can be captured more accurately.
7. Users see the system early.
8. Development can be divided into smaller parts and the risky parts can be developed earlier which helps in better risk management.

**Disadvantages**

1. Can be a costly model to use.
2. Risk analysis requires highly specific expertise.
3. Project’s success is highly dependent on the risk analysis phase.
4. Doesn’t work well for smaller projects.
5. Management is more complex.
6. End of the project may not be known early.
7. Not suitable for small or low risk projects and could be expensive for small projects.
8. Process is complex
9. Spiral may go on indefinitely.
10. Large number of intermediate stages requires excessive documentation.

**AGILE MODEL OR AGILE METHODOLOGY:**

**What is Agile?**

The word ‘agile’ means:

Able to move your body quickly and easily. Able to think quickly and clearly.
In business, ‘agile’ is used for describing ways of planning and doing work wherein it is understood that making changes as needed is an important part of the job. Business ‘agility’ means that a company is always in a position to take account of the market changes.

Agile SDLC model is a combination of iterative and incremental process models with focus on process adaptability and customer satisfaction by rapid delivery of working software product. Agile Methods break the product into small incremental builds. These builds are provided in iterations. Each iteration typically lasts from about one to three weeks. Every iteration involves cross functional teams working simultaneously on various areas like:

- Planning
- Requirements Analysis
- Design
- Coding
- Unit Testing and
- Acceptance Testing.

At the end of the iteration, a working product is displayed to the customer and important stakeholders.

What is Agile?

Agile model believes that every project needs to be handled differently and the existing methods need to be tailored to best suit the project requirements. In Agile, the tasks are divided to time boxes (small time frames) to deliver specific features for a release. Iterative approach is taken and working software build is delivered after each iteration. Each build is incremental in terms of features; the final build holds all the features required by the customer.
Here is a graphical illustration of the Agile Model:

![Agile Model Diagram]

**Agile Manifesto**
A team of software developers published the Agile Manifesto in 2001, highlighting the importance of the development team, accommodating changing requirements and customer involvement. The Agile Manifesto states that:
We are uncovering better ways of developing software by doing it and helping others do it. Through this work, we have come to value:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.
- That is, while there is value in the items on the right, we value the items on the left more.

**Characteristics of Agility**
Following are the characteristics of Agility:

- Agility in Agile Software Development focuses on the culture of the whole team with multi-discipline, cross-functional teams that are empowered and self-organizing.
- It fosters shared responsibility and accountability.
- Facilitates effective communication and continuous collaboration.
- The whole-team approach avoids delays and wait times.
- Frequent and continuous deliveries ensure quick feedback that in turn enable the team align to the requirements.
- Collaboration facilitates combining different perspectives timely in implementation, defect fixes and accommodating changes.
- Progress is constant, sustainable, and predictable emphasizing transparency.
Following are the Agile Manifesto principles:

- **Individuals and interactions** – In Agile development, self-organization and motivation are important, as are interactions like co-location and pair programming.

- **Working software** – Demo working software is considered the best means of communication with the customers to understand their requirements, instead of just depending on documentation.

- **Customer collaboration** – As the requirements cannot be gathered completely in the beginning of the project due to various factors, continuous customer interaction is very important to get proper product requirements.

- **Responding to change** – Agile Development is focused on quick responses to change and continuous development.

**Agile Vs Traditional SDLC Models**

- Agile is based on the adaptive software development methods, whereas the traditional SDLC models like the waterfall model is based on a predictive approach. Predictive teams in the traditional SDLC models usually work with detailed planning and have a complete forecast of the exact tasks and features to be delivered in the next few months or during the product life cycle.

- Predictive methods entirely depend on the requirement analysis and planning done in the beginning of cycle. Any changes to be incorporated go through a strict change control management and prioritization.

- Agile uses an adaptive approach where there is no detailed planning and there is clarity on future tasks only in respect of what features need to be developed. There is feature driven development and the team adapts to the changing product requirements dynamically. The product is tested very frequently, through the release iterations, minimizing the risk of any major failures in future.

- Customer Interaction is the backbone of this Agile methodology, and open communication with minimum documentation are the typical features of Agile development environment. The agile teams work in close collaboration with each other and are most often located in the same geographical location.
Agile Model - Pros and Cons

Agile methods are being widely accepted in the software world recently. However, this method may not always be suitable for all products. Here are some pros and cons of the Agile model.

The advantages of the Agile Model are as follows:

- Is a very realistic approach to software development.
- Promotes teamwork and cross training.
- Functionality can be developed rapidly and demonstrated.
- Resource requirements are minimum.
- Suitable for fixed or changing requirements.
- Delivers early partial working solutions.
- Good model for environments that change steadily.
- Minimal rules, documentation easily employed.
- Enables concurrent development and delivery within an overall planned context.
- Little or no planning required.
- Easy to manage.
- Gives flexibility to developers.

The disadvantages of the Agile Model are as follows:

- Not suitable for handling complex dependencies.
- More risk of sustainability, maintainability and extensibility.
- An overall plan, an agile leader and agile PM practice is a must without which it will not work.
- Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines.
- Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction.
- There is a very high individual dependency, since there is minimum documentation generated.
• Transfer of technology to new team members may be quite challenging due to lack of documentation.

Scrum

What is Scrum?

Scrum is easily one of the most well-known and commonly implemented software development frameworks in the world. At its core, scrum is actually an Agile framework. However, scrum is also technically a subset of Agile, since scrum adheres to a rather strict set of processes and practices. The word "scrum" is most commonly used in the sport of rugby football. In rugby, a scrum is a method of restarting play after a minor rule infringement has occurred, specifically in a situation where no advantage would be gained for the team that obeyed the rules. A scrum is formed with eight players from each team linking together into three tight rows on each side, before facing one another with heads down and attempting to push the opposing team out of position. The ball is thrown into the tunnel (the gap under the players), and the as the scrum begins, both teams attempt to move the ball using their legs and by repositioning themselves, such that the ball emerges behind their team’s formation and can be taken into possession. I’m no expert and this is a very rudimentary explanation compared to the full rules of a scrum, which are far more in-depth and interesting, but this origin of scrum from rugby transitions well into the use of the word within software development.

In the January 1986 issue of Harvard Business Review, two professors at Hitotsubashi University in Tokyo Japan, Hirotaka Takeuchi and Ikujiro Nonaka, published an article titled The New New Product Development Game, which emphasizes the importance of speed and flexibility when developing new products. In the article, Takeuchi and Nonaka detail the lessons learned from manufacturing practices, which they refer to as the "rugby approach," in which the team “tries to go the distance as a unit, passing the ball back and forth.” The article even goes on to explicitly make use of the term scrum to define these practices of pushing the team forward, which is considered the introduction of the term to modern software and project development.
A few years later, a software developer named Ken Schwaber began implementing the scrum framework into software development practices within his company. Over the next few decades, the scrum framework began to take shape, with the publication of books and guides outlining its use, including the definitive book by Schwaber and colleague Jeff Sutherland, *The Scrum Guide*. As it happens, the Guide provides us with a definition of what scrum is: “A framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value.”

**The Scrum Theory, Pillars, and Values**

The theory of scrum is focused on being an *empirical* process: a framework that attempts to gain experienced-based knowledge, then make decisions based on that learned knowledge. To meet that goal, scrum is founded on three core pillars:

- **Transparency**: All relevant aspects of the project must be well-defined and visible to everyone on the team that shares in the responsibility for said aspects.
- **Inspection**: Scrum Artifacts must be frequently inspected, to measure progress toward Sprint Goals (both of which we’ll discuss shortly).
- **Adaptation**: If an inspector decides that an aspect of the project is failing to meet its intended goals, that aspect should be adjusted as quickly as possible.

*The Scrum Guide* also lays out the five key scrum values:

- **Commitment**: Team members individually commit to achieving their team goals, each and every Sprint.
- **Courage**: Team members know they have the courage to work through conflict and challenges together so that they can do the right thing.
- **Focus**: Team members focus exclusively on their team goals and the Sprint Backlog; there should be no work done other than through their Backlog.
- **Openness**: Team members and their stakeholders agree to be transparent about their work and any challenges they face.
- **Respect**: Team members respect each other to be technically capable and to work with good intent.
Scrum Artifacts

A Scrum artifact is simply a representation of value or work to be completed, which is well-defined and should be transparently visible to all team members. There are three main types of Scrum artifacts:

- **Product Backlog**: This is the Scrum equivalent of a work in progress list. The Product Backlog is simply an ordered list of all items that are (or may be) necessary throughout the entirety of the software development life cycle. The Backlog acts as the definitive requirements documentation for the project. The Product Backlog is maintained by the Product Owner.

- **Sprint Backlog**: A Sprint Backlog is a subset of items from the Product Backlog that have been explicitly selected to be part of a Sprint.

- **Increment**: The Increment is the summation of all Product Backlog items that were successfully completed during a particular Sprint, added to the value of all previous Increments.

Scrum Team Roles
There are three primary roles within the whole of the scrum team:

- **Product Owner**: This individual can best be thought of as the project lead or producer, and his or her aim is to maximize the value of the product by ensuring the Development Team produces the best possible work. The Product Owner largely focuses on maintaining the Product Backlog.

- **Development Team**: The group of developers that actually produce the work defined in the Product Backlog into a functional and releasable iteration. The Development Team should be self-organizing and fully independent from the Scrum Master.

- **Scrum Master**: Acts as both the referee and coach for the whole team when it comes to the proper use and implementation of scrum practices and processes.

**Scrum Events and Workflow**

Day-to-day activities within the scrum framework are all based around particular scrum events. All events are “time-boxed”, meaning they each have a maximum duration. This helps to ensure that the development life cycle remains constantly adaptive and properly Agile.

The scrum framework defines five types of events:

- **Sprint Planning**: During this event, the entire team collaborates to define what the Sprint Goal will be for the upcoming Sprint. This is accomplished by answering two simple questions: 1) What work should be accomplished within the next month? 2) How can that work be completed? The Sprint Planning session should be kept to a maximum of eight hours per month.

- **Sprint**: The bread and butter of scrum practices, a Sprint is a one-month period in which a potentially releasable iteration is created (commonly referred to as a “Done”). In spite of the name, a Sprint is not anything like “crunch time.” Instead, a Sprint is always active: when one Sprint ends, the next immediately begins.

- **Daily Scrum**: Every day, the Development Team meets for a maximum of 15 minutes to discuss the planned work for the next day.

- **Sprint Review**: Following the completion of a Sprint, the Sprint Review is an event with a maximum duration of four hours, in which the entire team discusses the Increment results and makes any necessary changes to the Product Backlog.
- **Sprint Retrospective**: At a maximum of three hours, the Sprint Retrospective occurs after the Sprint Review, but prior to the next Sprint, and is a meeting for the entire team to decide on potential improvements that can be made general to practices or procedures for the next Sprint.

**Advantages of Scrum**

- **Allows for Rapid Prototyping**: With a maximum of only one month to devote to any particular Sprint Goal, scrum allows for rapid coding and development of ideas or components that may be experimental or may even fail, without severe worry or potential downsides.
- **Keeps Customers in the Loop**: Since scrum is an Agile framework and is highly iterative, customers are able to quickly assess progress and provide feedback throughout the entirety of the development life cycle.
- **Encourages Consistent Productivity**: The daily meetings of Daily Scrums are a guaranteed way to get insight from all team members about their progress, so suggestions and guidance can be provided where necessary to keep development on track.

**Disadvantages of Scrum**

- **Abundance of Meetings**: Many people, particularly developers trying to maintain that sweet flow state, are unlikely to appreciate the need for numerous meetings, particularly the Daily Scrums.
- **Potential Difficulty With Estimations**: As with other Agile frameworks, it’s quite easy to simply jump into a project and begin development without much in the way of planning. While this is often a benefit, it also means that scrum can often obfuscate the actual time and monetary costs of a project (or even aspects of said project), often until a few months down the line.
- **Requires Lenient Leadership**: Since proper scrum practices emphasize the importance of separating the management of Development Teams from roles like Scrum Master and Product Owner, successful implementation of scrum requires that managers and leadership are able to trust the Development Team and give them the freedom they need to work independently.
EXTREME PROGRAMMING

What is Extreme Programming?

- Extreme Programming is a software development methodology designed to improve the quality of software and its ability to properly adapt to the changing needs of the customer or client. During the mid and late nineties, while working on the Chrysler Comprehensive Compensation System (C3) to help manage the company’s payroll, software engineer Ken Beck first developed the Extreme Programming methodology. In October 1999, he published *Extreme Programming Explained*, detailing the entire method for others, and shortly thereafter the official website was launched as well. Similar to other Agile Methods of development, Extreme Programming aims to provide iterative and frequent small releases throughout the project, allowing both team members and customers to examine and review the project’s progress throughout the entire SDLC.

- XP is a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop a software. eXtreme Programming (XP) was conceived and developed to address the specific needs of software development by small teams in the face of vague and changing requirements. Extreme Programming is one of the Agile software development methodologies. It provides values and principles to guide the team behavior. The team is expected to self-organize. Extreme Programming provides specific core practices where Each practice is simple and self-complete.

- Combination of practices produces more complex and emergent behavior. Embrace Change A key assumption of Extreme Programming is that the cost of changing a program can be held mostly constant over time. This can be achieved with Emphasis on continuous feedback from the customer
  - Short iterations
  - Design and redesign
  - Coding and testing frequently
  - Eliminating defects early, thus reducing costs
  - Keeping the customer involved throughout the development
  - Delivering working product to the customer
  - Extreme Programming in a Nutshell
**Extreme Programming involves:**

Writing unit tests before programming and keeping all of the tests running at all times. The unit tests are automated and eliminates defects early, thus reducing the costs.

Starting with a simple design just enough to code the features at hand and redesigning when required.

Programming in pairs (called pair programming), with two programmers at one screen, taking turns to use the keyboard. While one of them is at the keyboard, the other constantly reviews and provides inputs.

Integrating and testing the whole system several times a day.

Putting a minimal working system into the production quickly and upgrading it whenever required.

Keeping the customer involved all the time and obtaining constant feedback.

Iterating facilitates the accommodating changes as the software evolves with the changing requirements.
Why is it called “Extreme?”

- Extreme Programming takes the effective principles and practices to extreme levels.
- Code reviews are effective as the code is reviewed all the time.
- Testing is effective as there is continuous regression and testing.
- Design is effective as everybody needs to do refactoring daily.
- Integration testing is important as integrate and test several times a day.
- Short iterations are effective as the planning game for release planning and iteration planning.

Extreme Values
These five fundamental values provide the foundation on which the entirety of the Extreme Programming paradigm is built, allowing the people involved in the project to feel confident in the direction the project is taking and to understand their personal feedback and insight is as necessary and welcome as anyone else.

Simplicity: We will do what is needed and asked for, but no more. This will maximize the value created for the investment made to date. We will take small simple steps to our goal and mitigate failures as they happen. We will create something we are proud of and maintain it long term for reasonable costs.
**Communication:** Everyone is part of the team and we communicate face to face daily. We will work together on everything from requirements to code. We will create the best solution to our problem that we can together.

**Feedback:** We will take every iteration commitment seriously by delivering working software. We demonstrate our software early and often then listen carefully and make any changes needed. We will talk about the project and adapt our process to it, not the other way around.

**Respect:** Everyone gives and feels the respect they deserve as a valued team member. Everyone contributes value even if it’s simply enthusiasm. Developers respect the expertise of the customers and vice versa. Management respects our right to accept responsibility and receive authority over our own work.

**Courage:** We will tell the truth about progress and estimates. We don’t document excuses for failure because we plan to succeed. We don’t fear anything because no one ever works alone. We will adapt to changes whenever they happen.

**Extreme Rules**
Initially published by Don Wells in 1999, the proprietor of the Extreme Programming website, this set of Extreme Programming Rules were originally intended to help to counter the claims that Extreme Programming fails to support some of the prominent disciplines necessary for modern development.

**Planning**
User stories are written.
Release planning creates the release schedule.
Make frequent small releases.
The project is divided into iterations.
Iteration planning starts each iteration.

**Managing**
Give the team a dedicated open work space.
Set a sustainable pace.
A stand up meeting starts each day.
The Project Velocity is measured.
Move people around.
Fix Extreme Programming when it breaks.
Designing
Simplicity.
Choose a system metaphor.
Use CRC cards for design sessions.
Create spike solutions to reduce risk.
No functionality is added early.
Refactor whenever and wherever possible.

Coding
The customer is always available.
Code must be written to agreed standards.
Code the unit test first.
All production code is pair programmed.
Only one pair integrates code at a time.
Integrate often.
Set up a dedicated integration computer.
Use collective ownership.

Testing
All code must have unit tests.
All code must pass all unit tests before it can be released.
When a bug is found tests are created.
Acceptance tests are run often and the score is published.

Extreme Practices
Created using what were considered the best practices of software development at the time, these twelve Extreme Programming Best Practices detail the specific procedures that should be followed when implementing a project using Extreme Programming.

Fine-scale feedback

Pair programming
In essence, pair programming means that two people work in tandem on the same system when developing any production code. By frequently rotating partners throughout the team, Extreme Programming promotes better communication and team-building.

Planning game
Often this takes the form of a meeting at a frequent and well-defined interval (every one or two weeks), where the majority of planning for the project takes place. Within this procedure exists the Release Planning stage, where determinations are made regarding what is required for impending releases. Sections of Release Planning include:

**Exploration Phase:** Story cards are used to detail the most valuable requirements from customers.

**Commitment Phase:** Planning and commitments from the team are made to meet the needs of the next schedule release and get it out on time.

**Steering Phase:** This allows for previously developed plans to be adjusted based on the evolving needs of the project, similar to many other Agile Model methodologies.

Following the Release Planning is also the Iteration Planning section, which consists of the same three sub-phases of its own, but with variants on their implementations:

**Exploration Phase:** All project requirements are written down.

**Commitment Phase:** Necessary tasks yet to be completed to meet the upcoming iteration release are assigned to developers and scheduled appropriately.

**Steering Phase:** Development takes place and, upon completion, the resulting iteration is compared to the outlined story cards created at the start of the Planning procedure.

**Test-driven development**

While an entire article could be written about test-driven development, the concept is fairly well known among developers and effectively means that tests are generated for each and every requirement of the project, and *only then* is code developed that will successfully pass those tests.

**Whole team**

As with many other SDLC methods and practices, Extreme Programming promotes the inclusion of customers and clients throughout the entire process, using their feedback to help shape the project at all times.

**Continuous process**

**Continuous integration**

Another common practice in modern development, the idea behind continuous integration is that all code developed across the entire team is merged into one common repository many times a day. This ensures that any issues with integration across the entire project are noticed and dealt with as soon as possible.

**Code refactoring**

Another very common practice, the idea behind code refactoring is simply to improve and redesign the structure of already existing code, without modifying its fundamental behavior. Simple examples of refactoring include fixing improperly names variables or methods, and reducing repeated code down to a single method or function.

**Small releases**
Very much in line with the practices of the Iterative Model, this concept ensures that the project will feature iterated, small releases on a frequent basis, allowing the customer as well, as all team members, to get a sense of how the project is developing.

**Shared understanding**

**Coding standards**
The coding standard is simply a set of best practices within the code itself, such as formatting and style, which the entire team abides by throughout the life cycle of the project. This promotes better understanding and readability of the code not only for current members, but for future developers as well.

**Collective code ownership**
This practice allows for any developer across the team to change any section of the code, as necessary. While this practice may sound dangerous to some, it speeds up development time, and any potential issues can be quelled with proper unit testing.

**Simple design**
There’s little reason to complicate things whenever a simpler option is available. This basic practice of keeping all components and code as simple as can be ensures that the entire team is always evaluating whether things could be done in an easier way.

**System metaphor**
Best thought of as part of the coding standards, the system metaphor is the idea that every person on the team should be able to look at the high-level code that is developed, and have a clear understanding of what functionality that code is performing.

**Programmer welfare**

**Sustainable pace**
A key concept for better work-life balance with developers on an Extreme Programming project is the notion that nobody should be required to work in excess of the normal scheduled work week. Overtime is frowned upon, as is the concept of “crunch time”, where developers are expected to work extreme hours near the end of a release to get everything completed on time.

**Extreme Programming Advantages:**
Extreme Programming solves the following problems often faced in the software development projects
- Slipped schedules – and achievable development cycles ensure timely deliveries.
- Cancelled projects – Focus on continuous customer involvement ensures transparency with the customer and immediate resolution of any issues.
Costs incurred in changes – Extensive and ongoing testing makes sure the changes do not break the existing functionality. A running working system always ensures sufficient time for accommodating changes such that the current operations are not affected.

Production and post-delivery defects: Emphasis is on – the unit tests to detect and fix the defects early.

Misunderstanding the business and/or domain – Making the customer a part of the team ensures constant communication and clarifications.

Business changes – Changes are considered to be inevitable and are accommodated at any point of time.

Staff turnover – Intensive team collaboration ensures enthusiasm and good will. Cohesion of multi-disciplines fosters the team spirit.

**Dynamic Systems Development Method (DSDM)**

The Dynamic Systems Development Method (DSDM) is an agile project delivery framework, primarily used as a software development method. It is a framework which embodies much of the current knowledge about project management. DSDM is rooted in the software development community, but the convergence of software development, process engineering and hence business development projects has changed the DSDM framework to become a general framework for complex problem solving tasks. The DSDM framework can be implemented for agile and traditional development processes.

**DSDM is a,**

- Straight forward framework based on best principles to start implementing a project structure.
- Simple
- Extendible
- But not calming to be the solution to all kind of projects.

DSDM, dating back to 1994, grew out of the need to provide an industry standard project delivery framework for what was referred to as Rapid Application Development (RAD) at the time. While RAD was extremely popular in the early 1990 ‘s, the RAD approach to software delivery evolved in a fairly unstructured manner. As a result, the DSDM Consortium was created and convened in 1994 with the goal of devising and promoting a common industry framework for rapid software delivery. Since 1994, the DSDM methodology has evolved and matured to provide a comprehensive foundation for planning, managing, executing, and scaling agile process and iterative software development projects.
DSDM is based on nine key principles that primarily revolve around business needs/value, active user involvement, empowered teams, frequent delivery, integrated testing, and stakeholder collaboration. DSDM specifically calls out “fitness for business purpose” as the primary criteria for delivery and acceptance of a system, focusing on the useful 80% of the system that can be deployed in 20% of the time.

Requirements are baselined at a high level early in the project. Rework is built into the process, and all development changes must be reversible. Requirements are planned and delivered in short, fixed-length time-boxes, also referred to as iterations, and requirements for DSDM projects are prioritized using MoSCoW Rules:

M – Must have requirements
S – Should have if at all possible
C – Could have but not critical
W – Won’t have this time, but potentially later

All critical work must be completed in a DSDM project. It is also important that not every requirement in a project or time-box is considered critical. Within each time-box, less critical items are included so that if necessary, they can be removed to keep from impacting higher priority requirements on the schedule.
The DSDM project framework is independent of, and can be implemented in conjunction with, other iterative methodologies such as Extreme Programming and the Rational Unified Process.

**Why use DSDM?**

- Results of development are directly and promptly visible
- Since the users are actively involved in the development of the system, they are more likely to embrace it and take it on.
- Basic functionality is delivered quickly, with more functionality being delivered at regular intervals.
- Eliminates bureaucracy and breaks down the communication barrier between interested parties.
- Because of constant feedback from the users, the system being developed is more likely to meet the need it was commissioned for.
- Early indicators of whether project will work or not, rather than a nasty surprise halfway through the development
- System is delivered on time and on budget.
- Ability of the users to affect the project's direction.

There are 9 principles which are essential to any DSDM implementation, ignoring one of them will break with the frameworks philosophy and significantly increase project risks.

1) Active user involvement – Imperative.
2) Teams must be empowered to make decisions.
3) Focus on frequent delivery.
4) Criterion for accepted deliverable (Fitness for Business).
5) Iterative and incremental development – Mandatory.
6) All changes during development must be reversible.
7) Requirements are base lined at high level.
8) Testing is integrated throughout the life cycle.
9) Collaborative and co-operative approach.

The Agile Manifesto claims 4 values and 12 principles, and is considered the beginning of agile development methods. These principles need to associate with the agile manifesto to show how DSDM relates to the agile philosophy.
Project structure

The DSDM project consists of 7 phased steps which are organized and embedded in a rich set of roles and responsibilities and are supported by several core techniques.

- Roles and Responsibilities
- Team Organization and Size
- 7 Phases to Rule Them

1. Pre-Project
2. Feasibility Study
3. Business Study
4. Functional Model Iteration
5. Design & Build Iteration
6. Implementation
7. Post-Project

Feature-Driven Development (FDD)

The FDD variant of agile methodology was originally developed and articulated by Jeff De Luca, with contributions by M.A. Rajashima, Lim Bak Wee, Paul Szego, Jon Kern and Stephen Palmer. The first incarnations of FDD occurred as a result of collaboration between De Luca and OOD thought leader Peter Coad. FDD is a model-driven, short-iteration process. It begins with establishing an overall model shape. Then it continues with a series of two-week “design by feature, build by feature” iterations. The features are small, “useful in the eyes of the client” results. FDD designs the rest of the development process around feature delivery using the following eight practices:

- Domain Object Modeling
- Developing by Feature
- Component/Class Ownership
- Feature Teams
- Inspections
- Configuration Management
- Regular Builds
- Visibility of progress and results
FDD recommends specific programmer practices such as “Regular Builds” and “Component/Class Ownership”. FDD’s proponents claim that it scales more straightforwardly than other approaches, and is better suited to larger teams. Unlike other agile methods, FDD describes specific, very short phases of work, which are to be accomplished separately per feature. These include Domain Walkthrough, Design, Design Inspection, Code, Code Inspection, and Promote to Build.

**CRYSTAL METHODS**

Crystal Methods are a family of software development methodologies developed by Alistair Cockburn from his study and interviews of teams. The methods are color-coded to signify the risk to human life. For example, projects that may involve risk to human life will use Crystal Sapphire while projects that do not have such risks will use Crystal Clear. Crystal focuses on six primary aspects: people, interaction, community, communication, skills, and talents. Process is considered secondary. There are also seven common properties in Crystal that indicate higher possibility of success and they include frequent delivery, reflective improvement, osmotic communication, and easy access to expert users. The methods are very flexible and avoid rigid processes because of its human-powered or people-centric focus. Alistair Cockburn is also one of the original signatories of the Agile Manifesto.

The Crystal methodology is one of the most lightweight, adaptable approaches to software development. Crystal is actually comprised of a family of agile methodologies such as Crystal Clear, Crystal Yellow, Crystal Orange and others, whose unique characteristics are driven by several factors such as team size, system criticality, and project priorities.

This Crystal family addresses the realization that each project may require a slightly tailored set of policies, practices, and processes in order to meet the project ‘s unique characteristics. Several of the key tenets of Crystal include teamwork, communication, and simplicity, as well as reflection to frequently adjust and improve the process. Like other agile process methodologies, Crystal promotes early, frequent delivery of working software, high user involvement, adaptability, and the removal of bureaucracy or distractions.
Crystal Methods consider people as the most important, so processes should be modeled to meet the requirements of the team. It is adaptive, without a set of prescribed tools and techniques. It is also lightweight, without too much documentation, management or reporting. The weight of the methodology is determined by the project environment and team size. For example, Crystal Clear is for short-term projects by a team of 6 developers working out of a single workspace.