Unit 1

***LECTURE 1***

**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*.** The principle of abstraction implies that a problem can be simplified by omitting irrelevant details. In other words, the main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose. Once the simpler problem is solved, then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on. Abstraction is a powerful way of reducing the complexity of the problem. The other approach to tackle problem complexity is decomposition. In this technique, a complex problem is divided into several smaller problems and then the smaller problems are solved one by one. However, in this technique any random decomposition of a problem into smaller parts will not help. The problem has to be decomposed such that each component of the decomposed problem can be solved independently and then the solution of the different components can be combined to get the full solution. A good decomposition of a problem should minimize interactions among various components. If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

**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.

**CHARACTERESTICS OF GOOD SOFTWARE**

A software product can be judged by what it offers and how well it can be used. This software must satisfy on the following grounds:

* Operational
* Transitional
* Maintenance

Well-engineered and crafted software is expected to have the following characteristics:

**Operational**

This tells us how well software works in operations. It can be measured on:

* Budget
* Usability
* Efficiency
* Correctness
* Functionality
* Dependability
* Security
* Safety

**Transitional**

This aspect is important when the software is moved from one platform to another:

* Portability
* Interoperability
* Reusability
* Adaptability

**Maintenance**

This aspect briefs about how well a software has the capabilities to maintain itself in the everchanging environment:

* Modularity
* Maintainability
* Flexibility
* Scalability

In short, Software engineering is a branch of computer science, which uses well-defined engineering concepts required to produce efficient, durable, scalable, in-budget and on-time software products

**Software Crisis** is a term used in computer science for the difficulty of writing useful and efficient computer programs in the required time. The software crisis was due to using the same workforce, same methods, same tools even though rapidly increasing in software demand, the complexity of software, and software challenges. With the increase in the complexity of software, many software problems arise because existing methods were insufficient. If we will use the same workforce, same methods, and same tools after the fast increase in software demand, software complexity, and software challenges, then there arise some problems like software budget problems, software efficiency problems, software quality problems, software managing and delivering problem, etc. This condition is called a software crisis.



Causes of Software Crisis:

1. The cost of owning and maintaining software was as expensive as developing the software
2. At that time Projects were running over-time
3. At that time Software was very inefficient
4. The quality of the software was low quality
5. Software often did not meet user requirements
6. The average software project overshoots its schedule by half
7. At that time Software was never delivered
8. Non-optimal resource utilization.
9. Difficult to alter, debug, and enhance.
10. The software complexity is harder to change.

Let’s now understand which factors are contributing to the software crisis.

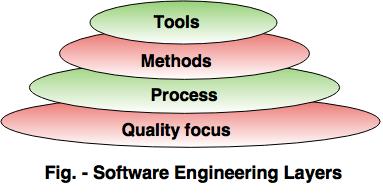
1. Poor project management.
2. Lack of adequate training in software engineering.
3. Less skilled project members.
4. Low productivity improvements.

**Solution of Software Crisis:** There is no single solution to the crisis. One possible solution to a software crisis is Software Engineering because software engineering is a systematic, disciplined, and quantifiable approach. For preventing software crises, there are some guidelines:

1. Reduction in software over budget.
2. The quality of software must be high.
3. Less time is needed for a software project.
4. Experienced and skilled people working over the software project.
5. Software must be delivered.
6. Software must meet user requirements.

## **Software engineering - Layered technology**

* Software engineering is a fully layered technology.To develop a software, we need to go from one layer to another.All these layers are related to each other and each layer demands the fulfillment of the previous layer.



**The layered technology consists of:**

**1. Quality focus: The characteristics of good quality software are:**

1. Correctness of the functions required to be performed by the software.
2. Maintainability of the software
3. Integrity i.e. providing security so that the unauthorized user cannot access information or data.
4. Usability i.e. the efforts required to use or operate the software.

**2. Process**

* It is the base layer or foundation layer for the software engineering.The software process is the key to keep all levels together. It defines a framework that includes different activities and tasks. In short, it covers all  activities, actions and tasks required to be carried out for software development.

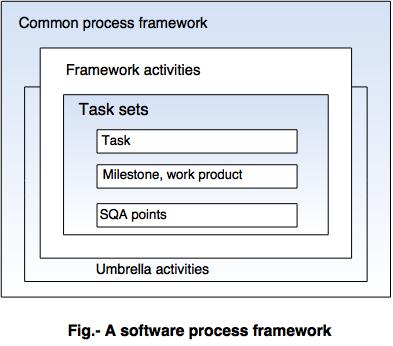
**3. Methods**

* The method provides the answers of all 'how-to' that are asked during the process. It provides the technical way to implement the software. It includes collection of tasks starting from communication, requirement analysis, analysis and design modeling, program construction, testing and support.

**4. Tools**

* The software engineering tool is an automated support for the software development. The tools are integrated i.e the information created by one tool can be used by the other tool. **For example:** The Microsoft publisher can be used as a web designing tool.

**Software Process Framework** The process of framework defines a small set of activities that are applicable to all types of projects. The software process framework is a collection of task sets. Task sets consist of a collection of small work tasks, project milestones, work productivity and software quality assurance points.



## **Umbrella activities**

Typical Umbrella Activities:

**1. Software project tracking and control**

* In this activity, the developing team accesses project plan and compares it with the predefined schedule.
* If these project plans do not match with the predefined schedule, then the required actions are taken to maintain the schedule.

**2. Risk management**

* Risk is an event that may or may not occur.
* If the event occurs, then it causes some unwanted outcome. Hence, proper risk management is required.

**3. Software Quality Assurance (SQA)**

* SQA is the planned and systematic pattern of activities which are required to give a guarantee of software quality.  
  **For example,** during the software development meetings are conducted at every stage of development to find out the defects and suggest improvements to produce good quality software.

**4. Formal Technical Reviews (FTR)**

* FTR is a meeting conducted by the technical staff.
* The motive of the meeting is to detect quality problems and suggest improvements.
* The technical person focuses on the quality of the software from the customer point of view.

**5. Measurement**

* Measurement consists of the effort required to measure the software.
* The software cannot be measured directly. It is measured by direct and indirect measures.
* Direct measures like cost, lines of code, size of software etc.
* Indirect measures such as quality of software which is measured by some other factor. Hence, it is an indirect measure of software.

**6. Software Configuration Management (SCM)**

* It manages the effect of change throughout the software process.

**7. Reusability management**

* It defines the criteria for reuse the product.
* The quality of software is good when the components of the software are developed for certain application and are useful for developing other applications.

**8. Work product preparation and production**

* It consists of the activities that are needed to create the documents, forms, lists, logs and user manuals for developing software.

A software process is a collection of various activities.  
**There are five generic process framework activities:**  
  
**1. Communication:**  
The software development starts with the communication between customer and developer.  
  
**2. Planning:**  
It consists of complete estimation, scheduling for project development and tracking.  
  
**3. Modeling:**

* Modeling consists of complete requirement analysis and the design of the project like algorithm, flowchart etc. The algorithm is the step-by-step solution of the problem and the flow chart shows a complete flow diagram of a program.

**4. Construction:**

* Construction consists of code generation and the testing part. Coding part implements the design details using an appropriate programming language. Testing is to check whether the flow of coding is correct or not. Testing also check that the program provides desired output.

**5. Deployment:**

Deployment step consists of delivering the product to the customer and take feedback from them. If the customer wants some corrections or demands for the additional capabilities, then the change is required for improvement in the quality of the software.

**Software Process Models** A software process model is a simplified representation of a software process. Each model represents a process from a specific perspective.These generic models are abstractions of the process that can be used to explain different approaches to the software development. They can be adapted and extended to create more specific processes.Some methodologies are sometimes known as software development life cycle (SDLC) methodologies, though this term could also be used more generally to refer to any methodology.

***LECTURE NOTE 2***

**SOFTWARE DEVELOPMENT LIFE CYCLE**

**LIFE CYCLE MODEL**

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 LIFE CYCLE 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.

***Different software life cycle models***

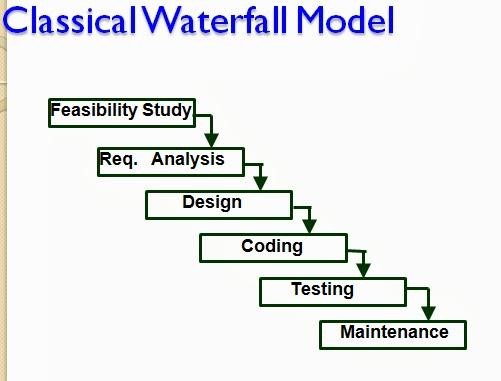
Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

* Classical Waterfall Model
* Iterative Waterfall Model
* Prototyping Model
* Evolutionary Model
* Spiral Model

1. **CLASSICAL WATERFALL MODEL**

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it cannot be used in actual software development projects. Thus, this model can be considered to be a *theoretical way of developing software*. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model. Classical waterfall model divides the life cycle into the following phases as shown in fig.2.1:

Fig 2.1: Classical Waterfall Model



**Feasibility study** - The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product.

* + At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system.
  + After they have an overall understanding of the problem they investigate the different solutions that are possible. Then they examine each of the solutions in terms of what kind of resources required, what would be the cost of development and what would be the development time for each solution.
  + Based on this analysis they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

**Requirements analysis and specification: -** The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

* + Requirements gathering and analysis
  + Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.

The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation.

**Design: -** The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

* + **Traditional design approach -**Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.
  + **Object-oriented design approach** -In this technique, various objects that occur in the problem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

**Coding and unit testing:-**The purpose of the coding phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested. During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.

**Integration and system testing: -**Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:

* + α – testing: It is the system testing performed by the development team.
  + β –testing: It is the system testing performed by a friendly set of customers.
  + Acceptance testing: It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing-related activities that must be performed, specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case.

**Maintenance: -**Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratios. 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.

***Shortcomings of the classical waterfall model***

The classical waterfall model is an idealistic one since it assumes that no development error is ever committed by the engineers during any of the life cycle phases. However, in practical development environments, the engineers do commit a large number of errors in almost every phase of the life cycle. The source of the defects can be many: oversight, wrong assumptions, use of inappropriate technology, communication gap among the project engineers, etc. These defects usually get detected much later in the life cycle. For example, a design defect might go unnoticed till we reach the coding or testing phase. Once a defect is detected, the engineers need to go back to the phase where the defect had occurred and redo some of the work done during that phase and the subsequent phases to correct the defect and its effect on the later phases. Therefore, in any practical software development work, it is not possible to strictly follow the classical waterfall model.

***LECTURE NOTE 3***

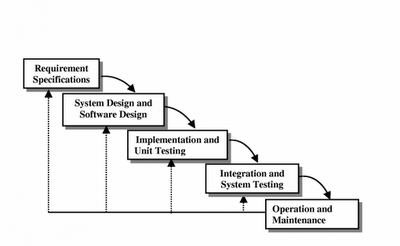
1. **ITERATIVE WATERFALL MODEL**

To overcome the major shortcomings of the classical waterfall model, we come up with the iterative waterfall model.

Fig 3.1

:

Iterative Waterfall Model



Here, we provide feedback paths for error correction as & when detected later in a phase. Though errors are inevitable, but it is desirable to detect them in the same phase in which they occur. If so, this can reduce the effort to correct the bug.

The advantage of this model is that there is a working model of the system at a very early stage of development which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget.

The disadvantage with this SDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.

1. **PRTOTYPING MODEL**

**Prototype**

A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations. A prototype usually turns out to be a very crude version of the actual system.

***Need for a prototype in software development***

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer’s needs:

* + - * how the screens might look like
      * how the user interface would behave
      * how the system would produce outputs

Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.

A prototyping model can be used when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues associated with the product development. Often, major design decisions depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.

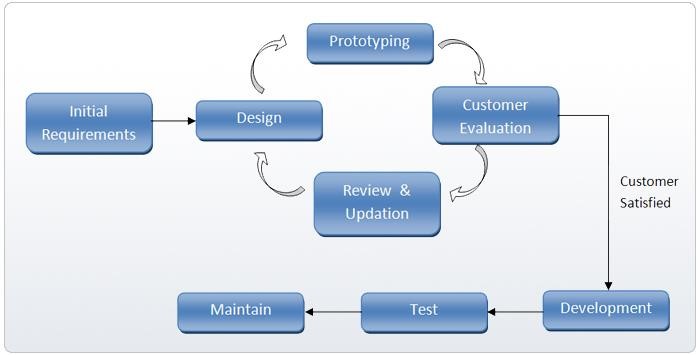
A prototype of the actual product is preferred in situations such as:

* + User requirements are not complete
  + Technical issues are not clear

Fig 3.2

:

Prototype Model



1. **EVOLUTIONARY MODEL**

It is also called *successive versions model* or *incremental model*. At first, a simple working model is built. Subsequently it undergoes functional improvements & we keep on adding new functions till the desired system is built.

Applications:

* + Large projects where you can easily find modules for incremental implementation. Often used when the customer wants to start using the core features rather than waiting for the full software.
  + Also used in object oriented software development because the system can be easily portioned into units in terms of objects.

Advantages:

* + User gets a chance to experiment partially developed system
  + Reduce the error because the core modules get tested thoroughly.

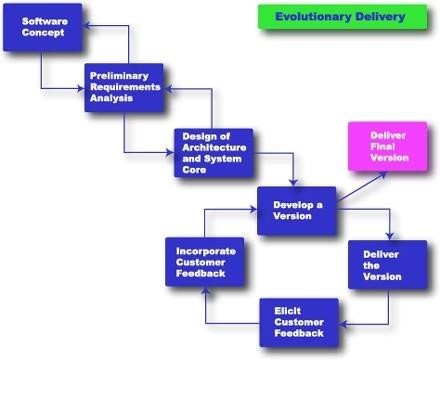
Disadvantages:

* + It is difficult to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented & delivered.

Fig 3.3

:

Evolutionary Model



***LECTURE NOTE 4***

1. **SPIRAL MODEL**

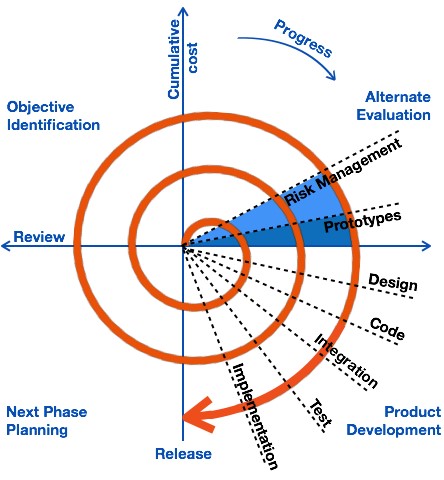
The Spiral model of software development is shown in fig. 4.1. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study, the next loop with requirements specification, the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in fig. 4.1. The following activities are carried out during each phase of a spiral model.

Fig

4.1

:

Spiral Model



**First quadrant (Objective Setting)**

* + - During the first quadrant, it is needed to identify the objectives of the phase.
    - Examine the risks associated with these objectives.

**Second Quadrant (Risk Assessment and Reduction)**

* + - A detailed analysis is carried out for each identified project risk.
    - Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

**Third Quadrant (Development and Validation)**

* + - Develop and validate the next level of the product after resolving the identified risks.

**Fourth Quadrant (Review and Planning)**

* + - Review the results achieved so far with the customer and plan the next iteration around the spiral.
    - Progressively more complete version of the software gets built with each iteration around the spiral.

***Circumstances to use spiral model***

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

***Comparison of different life-cycle models***

The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model cannot be used in practical development projects, since this model supports no mechanism to handle the errors committed during any of the phases.

This problem is overcome in the iterative waterfall model. The iterative waterfall model is probably the most widely used software development model evolved so far. This model is simple to understand and use. However this model is suitable only for well-understood problems; it is not suitable for very large projects and for projects that are subject to many risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood. This model is especially popular for development of the user-interface part of the projects.

The evolutionary approach is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery. This model is also widely used for objectoriented development projects. Of course, this model can only be used if the incremental delivery of the system is acceptable to the customer.

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

The different software life cycle models can be compared from the viewpoint of the customer. Initially, customer confidence in the development team is usually high irrespective of the development model followed. During the lengthy development process, customer confidence normally drops off, as no working product is immediately visible. Developers answer customer queries using technical slang, and delays are announced. This gives rise to customer resentment. On the other hand, an evolutionary approach lets the customer experiment with a working product much earlier than the monolithic approaches. Another important advantage of the incremental model is that it reduces the customer’s trauma of getting used to an entirely new system. The gradual introduction of the product via incremental phases provides time to the customer to adjust to the new product. Also, from the customer’s financial viewpoint, incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

**Agile Methods**

Agile software development refers to software development methodologies centered around the idea of iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. The ultimate value in Agile development is that it enables teams to deliver value faster, with greater quality and predictability, and greater aptitude to respond to change. Scrum and Kanban are two of the most widely used Agile methodologies. Below are the most frequently asked questions around Agile and Scrum, answered by our experts.

WHAT IS AGILE?

Agile software development refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. Agile methods or Agile processes generally promote a disciplined project management process that encourages frequent inspection and adaptation, a leadership philosophy that encourages teamwork, self-organization and accountability, a set of engineering best practices intended to allow for rapid delivery of high-quality software, and a business approach that aligns development with customer needs and company goals. Agile development refers to any development process that is aligned with the concepts of the Agile Manifesto. The Manifesto was developed by a group of fourteen leading figures in the software industry, and reflects their experience of what approaches do and do not work for software development.

# Agile Software Development

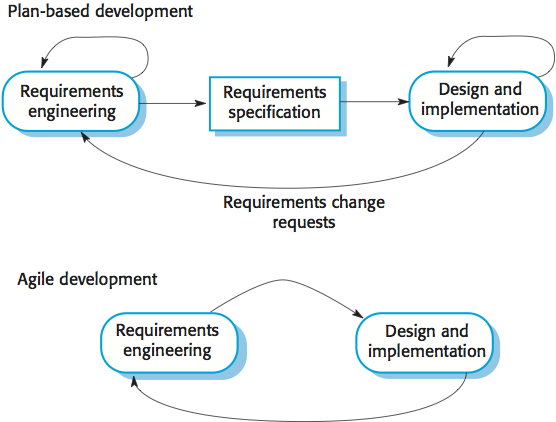
**The big picture**

Rapid development and delivery is now often the most important requirement for software systems. Businesses operate in a fast-changing requirement and it is practically impossible to produce a set of stable software requirements. Software has to evolve quickly to reflect changing business needs.

**Agile development** methods emerged in the late 1990s whose aim was to radically reduce the delivery time for working software systems:

* Program specification, design, and implementation are interleaved
* The system is developed as a series of frequent versions or increments
* Stakeholders involved in version specification and evaluation
* Extensive tool support (e.g. automated testing tools) used to support development
* Minimal documentation - focus on working code

**Plan-based v/s agile development**



**Plan-driven development**

A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance. Not necessarily waterfall model: plan-driven, incremental development is possible. Iteration occurs within activities.

**Agile development**

Specification, design, implementation and testing are inter-leaved and the outputs from the development process are decided through a process of negotiation during the software development process.

Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on many technical, human, and organizational issues.

**Agile methods**

Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:

* **Focus on the code** rather than the design.
* Are based on an **iterative approach** to software development.
* Are intended to deliver working software quickly and evolve this quickly to **meet changing requirements**.

The aim of agile methods is to reduce overheads in the software process (e.g. by limiting documentation) and to be able to respond quickly to changing requirements without excessive rework.

[**Manifesto for Agile Software Development**](http://agilemanifesto.org/):

*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.*

The **principles** of agile methods:

**Customer involvement**

Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.

**Incremental delivery**

The software is developed in increments with the customer specifying the requirements to be included in each increment.

**People not process**

The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.

**Embrace change**

Expect the system requirements to change and so design the system to accommodate these changes.

**Maintain simplicity**

Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

Agile method **applicability**:

* Product development where a software company is developing a **small or medium-sized product**.
* Custom system development within an organization, where there is a clear **commitment from the customer** to become involved in the development process and where there are not a lot of external rules and regulations that affect the software.
* Because of their focus on small, tightly-integrated teams, there are **problems in scaling** agile methods to large systems.

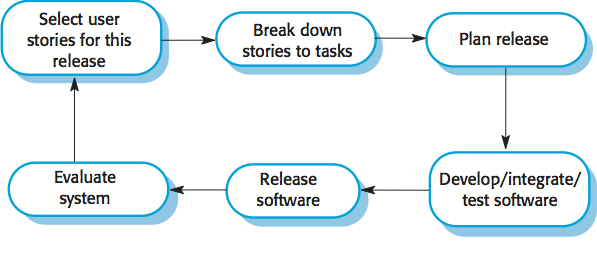
**Problems** with agile methods:

* It can be difficult to keep the interest of **customers** who are involved in the process.
* **Team members** may be unsuited to the intense involvement that characterizes agile methods.
* Prioritizing changes can be difficult where there are **multiple stakeholders**.
* **Maintaining simplicity** requires extra work.
* **Contracts** may be a problem as with other approaches to iterative development.

**Extreme programming**

Perhaps the best-known and a very influential agile method, Extreme Programming (XP) takes an 'extreme' approach to iterative development:

* New versions may be built several times per day;
* Increments are delivered to customers every 2 weeks;
* All tests must be run for every build and the build is only accepted if tests run successfully.



This is how XP supports **agile principles**:

* Incremental development is supported through **small, frequent system releases**.
* Customer involvement means **full-time customer engagement** with the team.
* People not process through **pair programming**, **collective ownership**, and a process that avoids long working hours.
* Change supported through **regular system releases**.
* Maintaining simplicity through **constant refactoring** of code.

**Influential XP practices**

Extreme programming has a technical focus and is not easy to integrate with management practice in most organizations. Consequently, while agile development uses practices from XP, the method as originally defined is not widely used.

Key practices of XP include:

**User stories for specification**

In XP, a customer or user is part of the XP team and is responsible for making decisions on requirements. User requirements are expressed as user stories or scenarios. These are written on cards and the development team breaks them down into implementation tasks. These tasks are the basis of schedule and cost estimates. The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.

**Refactoring**

Conventional wisdom in software engineering is to design for change. It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle. XP, however, maintains that this is not worthwhile as changes cannot be reliably anticipated. Rather, it proposes constant code improvement (refactoring) to make changes easier when they have to be implemented. Programming teams look for possible software improvements and make these improvements even where there is no immediate need for them. This improves the understandability of the software and so reduces the need for documentation. Changes are easier to make because the code is well-structured and clear. However, some changes require architecture refactoring and this is much more expensive.

**Test-first development**

Testing is central to XP and XP has developed an approach where the program is tested after every change has been made.  
**Test-driven development:** writing tests before code clarifies the requirements to be implemented. Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly (usually relies on a testing framework such as Junit). All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors.  
**Customer involvement:** The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system. The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs. However, people adopting the customer role have limited time available and so cannot work full-time with the development team. They may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

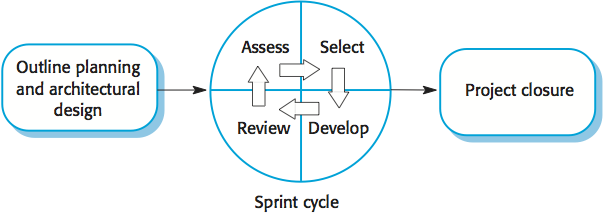
**Pair programming**

Pair programming involves programmers working in pairs, developing code together. This helps develop common ownership of code and spreads knowledge across the team. It serves as an informal review process as each line of code is looked at by more than one person. It encourages refactoring as the whole team can benefit from improving the system code. In pair programming, programmers sit together at the same computer to develop the software. Pairs are created dynamically so that all team members work with each other during the development process. The sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave. Pair programming is not necessarily inefficient and there is some evidence that suggests that a pair working together is more efficient than two programmers working separately.

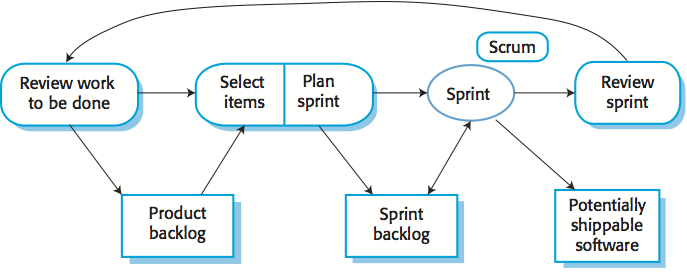
**Scrum**

The Scrum approach is a general agile method but its focus is on managing iterative development rather than specific agile practices. There are three phases in Scrum:

1. The initial phase is an outline planning phase where you establish the general objectives for the project and design the software architecture.
2. This is followed by a series of **sprint** cycles, where each cycle develops an increment of the system.
3. The project closure phase wraps up the project, completes required documentation such as system help frames and user manuals and assesses the lessons learned from the project.



* Sprints are fixed length, normally 2-4 weeks. They correspond to the development of a release of the system in XP.
* The starting point for planning is the **product backlog**, which is the list of work to be done on the project.
* The selection phase involves all of the project team who work with the customer (**product owner**) to select the features and functionality to be developed during the sprint. Once these are agreed, the team organize themselves to develop the software. During this stage the team is relatively isolated from the product owner and the organization, with all communications channelled through the **ScrumMaster**.
* The role of the ScrumMaster is to protect the development team from external distractions. At the end of the sprint the work done is reviewed and presented to stakeholders (including the product owner).
* **Velocity** is calculated during the **sprint review**; it provides an estimate of how much product backlog the team can cover in a single sprint. Understanding the team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring and improving performance. The next sprint cycle then begins.



The ScrumMaster is a facilitator who arranges short daily meetings (**daily scrums**), tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with the product owner and management outside of the team. The whole team attends daily scrums where all team members share information, describe their progress since the last meeting, problems that have arisen and what is planned for the following day.

**Advantages** of scrum include:

* The product is broken down into a set of **manageable and understandable chunks**.
* Unstable requirements do not hold up **progress**.
* The whole team has visibility of everything and consequently **team communication** is improved.
* Customers see **on-time delivery** of increments and gain feedback on how the product works.
* **Trust** between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.

**Scaling agile methods**

Agile methods have proved to be **successful for small and medium sized projects** that can be developed by a small co-located team. It is sometimes argued that the success of these methods comes because of improved communications which is possible when everyone is working together. Scaling up agile methods involves changing these to cope with larger, longer projects where there are multiple development teams, perhaps working in different locations.

Two perspectives on scaling of agile methods:

**'Scaling up'**

Use agile methods to develop large software systems which cannot be developed by a small team. For large systems development, it is not possible to focus only on the code of the system; you need to do more up-front design and system documentation. Cross-team communication mechanisms have to be designed and used, which should involve regular phone and video conferences between team members and frequent, short electronic meetings where teams update each other on progress. Continuous integration, where the whole system is built every time any developer checks in a change, is practically impossible; however, it is essential to maintain frequent system builds and regular releases of the system.

**'Scaling out'**

How agile methods can be introduced across a large organization with many years of software development experience. Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach. Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods. Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities. There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

**Use Story**

User story is the smallest unit of work in an agile framework. It's an end goal, not a feature, expressed from the software user's perspective. A user story is an informal, general explanation of a software feature written from the perspective of the end user or customer.

User Story is a note that captures what a user does or needs to do as part of her work. Each User Story consists of a short description written from user's point of view, with natural language. Unlike the traditional requirement capturing, User Story focuses on what the user needs instead of what the system should deliver. This leaves room for further discussion of solutions and the result of a system that can really fit into the customers' business workflow, solving their operational problems and most importantly adding value to the organization.

**Concept of 3C's**

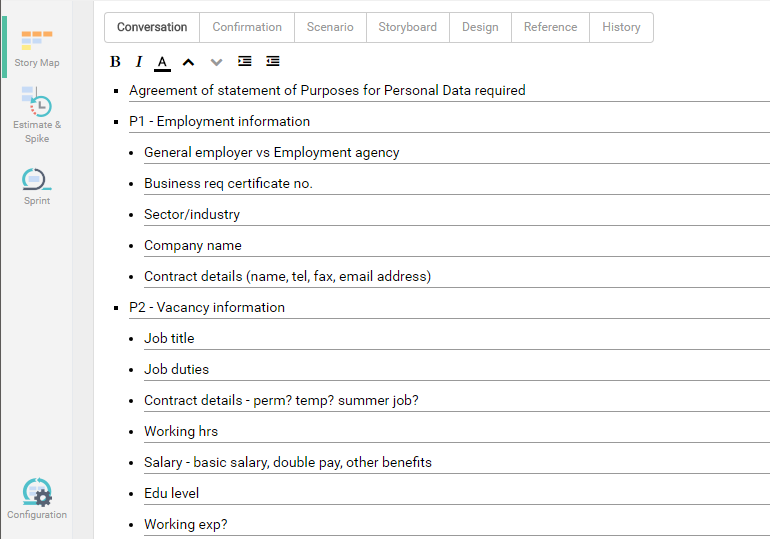
The 3C's refer to the three critical aspects of good user stories. The concept was suggested by Ron Jeffries, the co-inventor of the user stories practice. Nowadays, when we talk about User Stories, we usually are referring to the kind of User Stories that are composed of these three aspects.

**Card**

User Stories are written as cards. Each User Story card has a short sentence with just-enough text to remind everyone of what the story is about.

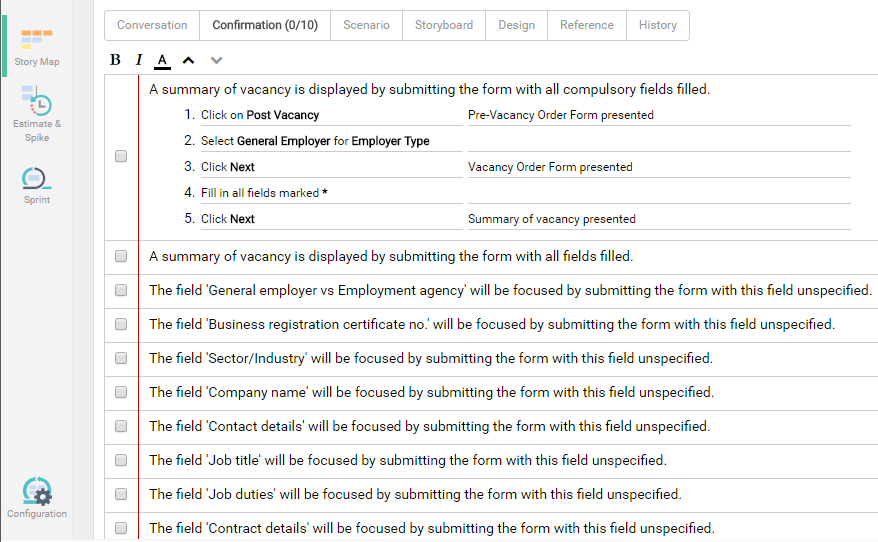
**Conversation**

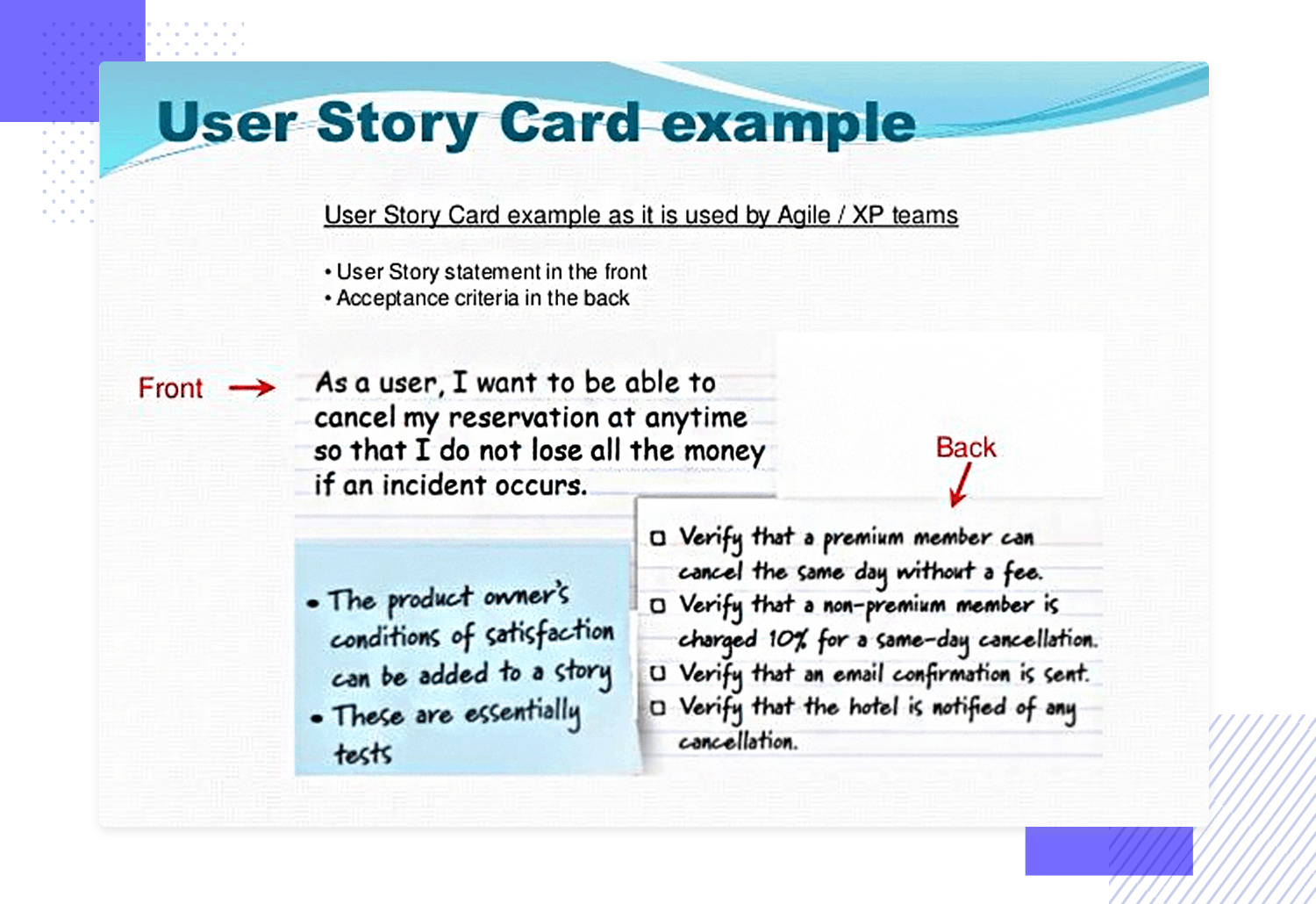
Requirements are found and re-fined through continuous conversations between customers and development team throughout the whole software project. Important ideas and decisions would be discovered and recorded during the stakeholder meetings.

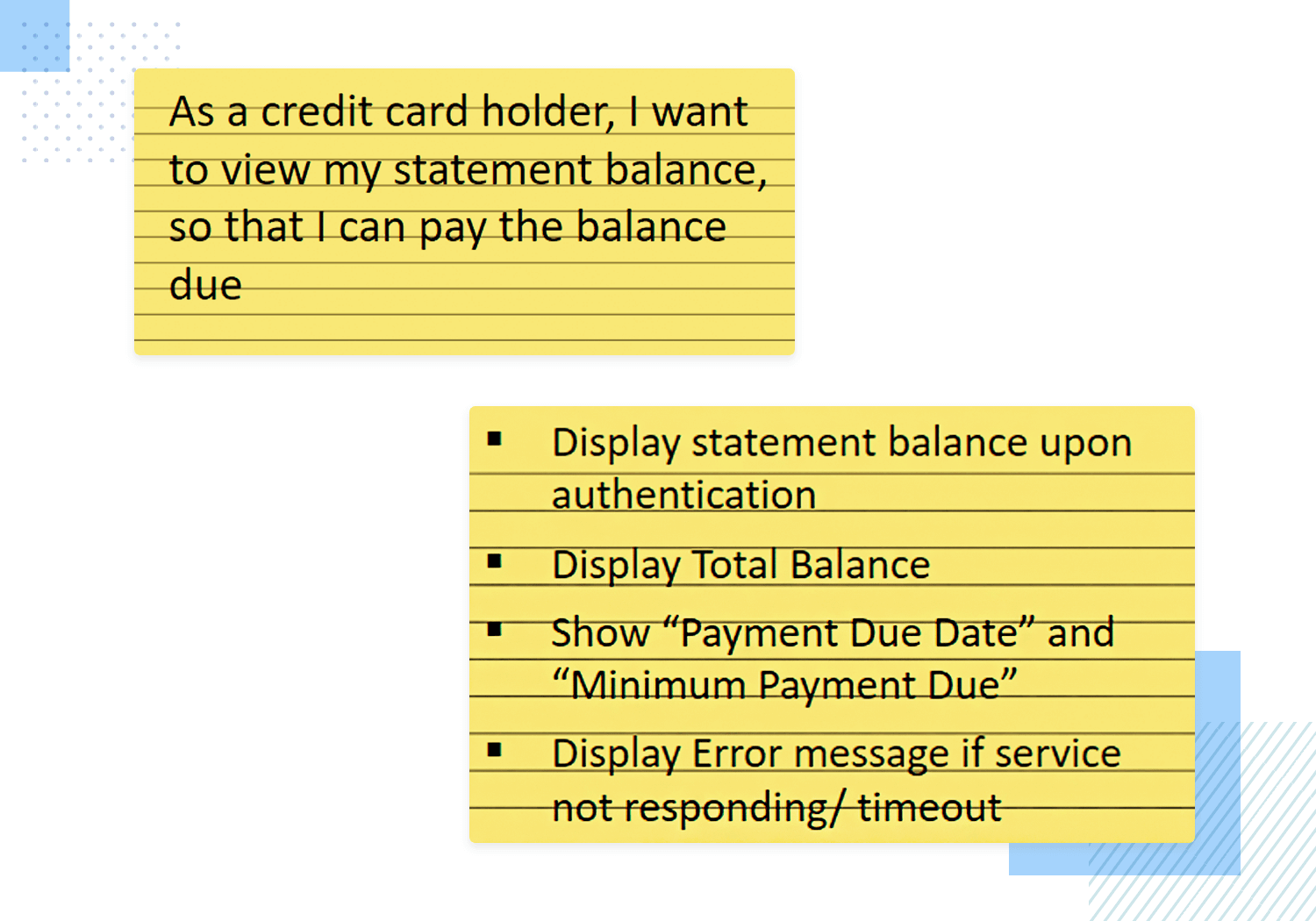


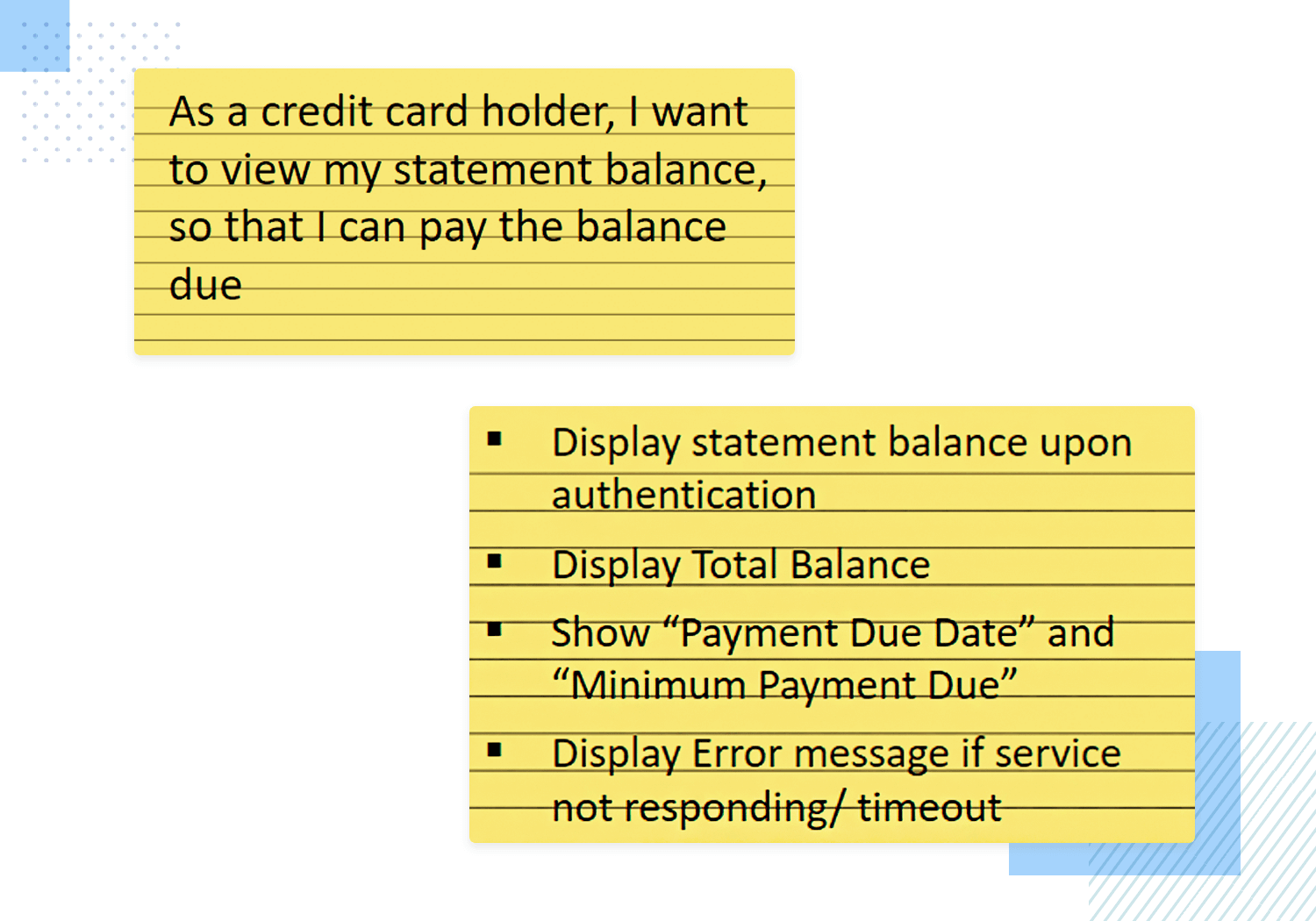
**Confirmation**

Confirmation is also known as the acceptance criteria of the User Story. During the discussion of requirements, the customer tells the analyst not only what he/she wants, but also confirming under what conditions and criteria the working software would be accepted or rejected. The cases defined are written as confirmation. Note that confirmation focuses on verifying the correctness of work of the corresponding User Story. It is not an integration testing.





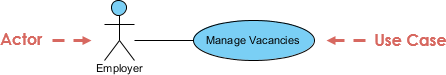


**Bank app user story example.**

The user in this user story example wants to pay their balance, so the first thing the design team might do is start working on a solution that gives them more or less instant access to their credit card balance. Either it should be the first thing they see when they open the app or there should be a clear option to see the balance of that card that’s just a tap away.

## **Use Cases**

Use Cases, introduced by **[Ivar Jacobson](https://en.wikipedia.org/wiki/Ivar_Jacobson" \t "_blank)** more than 20 years ago, are used to capture user (actor) point of view while describing functional requirements of the system. They describe the step by step process a user goes through to complete that goal using a software system.



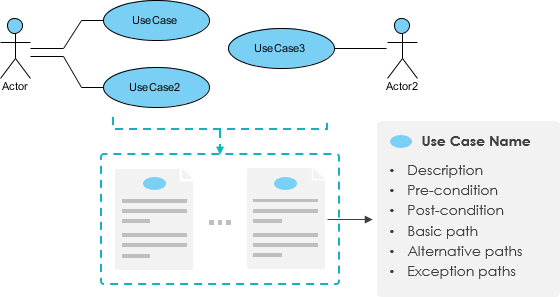
A Use Case is a description of all the ways an end-user wants to "use" a system. Use Cases capture all the possible ways the user and system can interact that result in the user achieving the goal. They also capture all the things that can go wrong along the way that prevent the user from achieving the goal.

A Use-Case model consists of a number of model elements. The most important model elements are:

* Use Cases,
* Actors
* and the relationships between them.

## **Detailed Use Case Specification**

A Use Case Specification is a textual description of the functionality provided by the system. It captures actor-system interaction. That is, it specifies how a user interacts with a system and how the system responds to the user actions. It is often phrased in the form of a dialog between the actor and the system. The Use Case Specification is represented in the Use Case Diagram by an oval, and is what most people think of when they hear the term Use Case.



## **Why We Still Need Use Cases?**

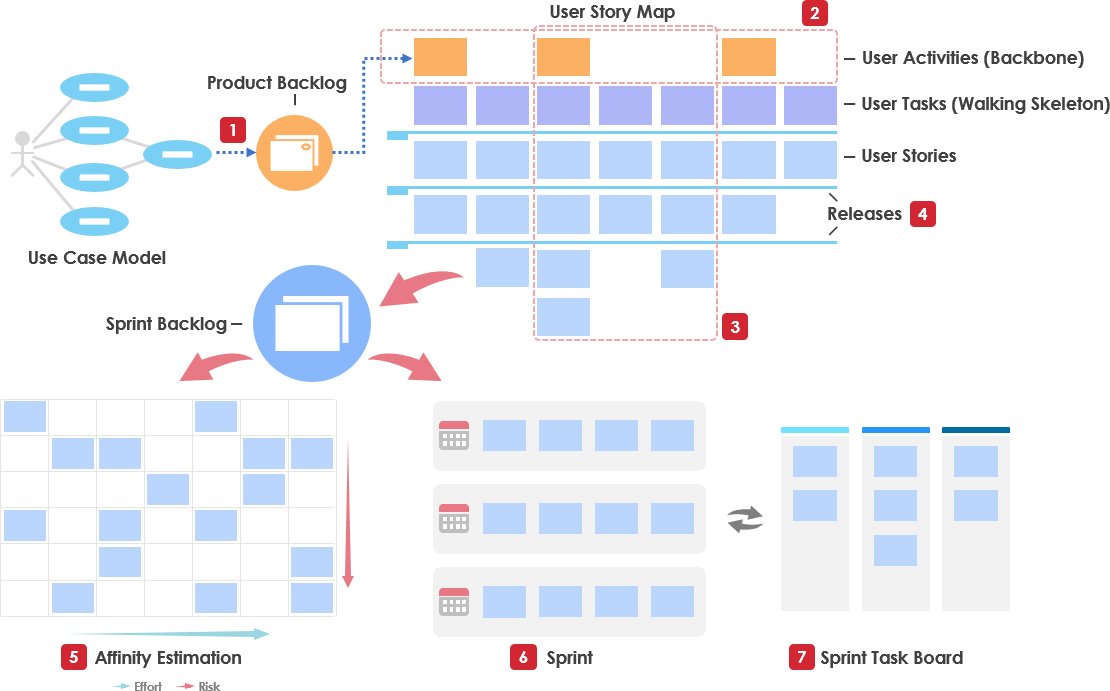
## [Alistair Cockburn](http://alistair.cockburn.us/) explains that he sees (with companies he consults to) three main problems with User Stories:

## Lack of context (what's the largest goal)

## Sense of completeness that all bases related to a goal are covered.

## No mechanism for looking ahead at upcoming work.

I**ntegrate Use Case, User Story and Story Mapping techniques**



**User Stories vs Use Cases**

User Stories often start out the same way as Use Cases, in that each describes one way to use the system, is centered around a goal, is written from the perspective of a user, uses the natural language of the business, and - on its own - does not tell the whole story.

**User Stories vs Use Cases - The Similarity**

If we consider the key component in both approaches:

* User Stories contain, with user role, goal and acceptance criteria.
* Use Cases contain equivalent elements: an actor, flow of events and post conditions respectively (a detailed Use Case template may contain many more other elements).

**User Stories vs Use Cases - The Difference**

The details of a User Story may not be documented to the same extreme as a Use Case.

* User Stories deliberately leave out a lot of important details. User Stories are meant to elicit conversations by asking questions during scrum meetings.
* Small increments for getting feedback more frequently, rather than having more detailed up-front requirement specification as in Use Cases.

WHAT IS SCRUM?

Scrum is a subset of Agile. It is a lightweight process framework for agile development, and the most widely-used one.

A “process framework” is a particular set of practices that must be followed in order for a process to be consistent with the framework. (For example, the Scrum process framework requires the use of development cycles called Sprints, the XP framework requires pair programming, and so forth.)

“Lightweight” means that the overhead of the process is kept as small as possible, to maximize the amount of productive time available for getting useful work done.

A Scrum process is distinguished from other agile processes by specific concepts and practices, divided into the three categories of Roles, Artifacts, and Time Boxes. These and other terms used in Scrum are defined below. Scrum is most often used to manage complex software and product development, using iterative and incremental practices. Scrum significantly increases productivity and reduces time to benefits relative to classic “waterfall” processes. Scrum processes enable organizations to adjust smoothly to rapidly-changing requirements, and produce a product that meets evolving business goals. An agile Scrum process benefits the organization by helping it to

1. Increase the quality of the deliverables
2. Cope better with change (and expect the changes)
3. Provide better estimates while spending less time creating them
4. Be more in control of the project schedule and state

WHAT ARE THE BENEFITS OF AGILE?

Benefits to Customer

Customers find that the vendor is more responsive to development requests. High-value features are developed and delivered more quickly with short cycles, than with the longer cycles favored by classic “waterfall” processes.

Benefits to Vendors

Vendors reduce wastage by focusing development effort on high-value features, and reduce time-to-market relative to waterfall processes due to decreased overhead and increased efficiency. Improved customer satisfaction translates to better customer retention and more positive customer references.

Benefits to Development Teams

Team members enjoy development work, and like to see their work used and valued. Scrum benefits Team members by reducing non-productive work (e.g., writing specifications or other artifacts that no one uses), and giving them more time to do the work they enjoy. Team members also know their work is valued, because requirements are chosen to maximize value to customers.

Benefits to Product Managers

Product Managers, who typically fill the Product Owner role, are responsible for making customers happy by ensuring that development work is aligned with customer needs. Scrum makes this alignment easier by providing frequent opportunities to re-prioritize work, to ensure maximum delivery of value.

Benefits to Project Managers

Project Managers (and others) who fill the ScrumMaster role find that planning and tracking are easier and more concrete, compared to waterfall processes. The focus on task-level tracking, the use of Burndown Charts to display daily progress, and the Daily Scrum meetings, all together give the Project Manager tremendous awareness about the state of the project at all times. This awareness is key to monitoring the project, and to catching and addressing issues quickly.

Benefits to PMOs and C-Level Executives

Scrum provides high visibility into the state of a development project, on a daily basis. External stakeholders, such as C-Level executives and personnel in the Project Management Office, can use this visibility to plan more effectively, and adjust their strategies based on more hard information and less speculation.

WHAT ARE THE SCRUM REQUIREMENTS?

Scrum does not define just what form requirements are to take, but simply says that they are gathered into the Product Backlog, and referred to generically as “Product Backlog Items,” or “PBIs” for short. Given the time-boxed nature of a Sprint, we can also infer that each set should require significantly less time to implement than the duration of the Sprint. Most Scrum projects borrow the “XP” (Extreme Programming) practice of describing a feature request as a “User Story,” although a minority uses the older concept of a “Use Case.” We will go with the majority view here, and describe three reasonably-standard requirements artifacts found in Product Backlogs.

**User Story**

A User Story describes a desired feature (functional requirement) in narrative form. User Stories are usually written by the Product Owner, and are the Product Owner’s responsibility. The format is not standardized, but typically has a name, some descriptive text, references to external documents (such as screen shots), and information about how the implementation will be tested. For example, a Story might resemble the following:

Name: Planner enters new contact into address book, so that one can contact the person later by postal or electronic mail

Description: Planner enters standard contact information (first and last name, two street address lines, city, state, zip / postal code, country, etc.) into contact-entry screen. One clicks “Save” to keep the data, and “Cancel” to discard data and return to previous screen.

How to test: Tester enters and saves the data, finds the name in the address book, and clicks on it. One sees a read-only view of the contact-entry screen, with all data previously entered.

The elements in this User Story are:

Name: The Name is a descriptive phrase or sentence. The example uses a basic “Role-Action-Reason” organization.

Another common style, popularized by Mike Cohn, follows the template “As a <type of user>, I want <some goal> so that <some reason>.” The choice of template is less important than having a workable standard of some kind.

**Description:** This is a high-level (low-detail) description of the need to be met. For functional (user-facing) requirements, the description is put in narrative form. For non-functional requirements, the description can be worded in any form that is easy to understand. In both cases, the key is that the level of detail is modest, because the fine details are worked out during the implementation phase, in discussions between team members, product owners, and anyone else who is involved. (This is one of the core concepts of Scrum: Requirements are specified at a level that allows rough estimation of the work required to implement them, not in detail.)

**Screens and External Documents:** If the Story requires user-interface changes (especially non-trivial ones), the Story should contain or link to a prototype of the changes. Any external documents required to implement the Story should also be listed.

**How to test:** The implementation of a Story is defined to be complete if, and only if, it passes all acceptance tests developed for it. This section provides a brief description of how the story will be tested. As for the feature itself, the description of testing methods is short, with the details to be worked out during implementation, but we need at least a summary to guide the estimation process.

There are two reasons for including the information about how to test the Story. The obvious reason is to guide development of test cases (acceptance tests) for the Story. The less-obvious, but important, reason, is that the Team will need this information in order to estimate how much work is required to implement the story (since test design and execution is part of the total work).

**Story**

Not all requirements for new development represent user-facing features, but do represent significant work that must be done. These requirements often, but not always, represent work that must be done to support user-facing features. We call these non-functional requirements “Technical Stories.” Technical Stories have the same elements as User Stories, but need not be cast into narrative form if there is no benefit in doing so. Technical Stories are usually written by Team members, and are added to the Product Backlog. The Product Owner must be familiar with these Stories, and understand the dependencies between these and User Stories in order to rank (sequence) all Stories for implementation.

**Defect**

A Defect, or bug report, is a description of a failure of the product to behave in the expected fashion. Defects are stored in a bug-tracking system, which may or may not be physically the same system used to store the Product Backlog. If not, then someone (usually the Product Owner) must enter each Defect into the Product Backlog, for sequencing and scheduling.

**WHAT ARE THE SCRUM ROLES?**

The three roles defined in Scrum are the ScrumMaster, the Product Owner, and the Team (which consists of Team members). The people who fulfill these roles work together closely, on a daily basis, to ensure the smooth flow of information and the quick resolution of issues.

**ScrumMaster**

The ScrumMaster (sometimes written “Scrum Master,” although the official term has no space after “Scrum”) is the keeper of the process. The ScrumMaster is responsible for making the process run smoothly, for removing obstacles that impact productivity, and for organizing and facilitating the critical meetings. The ScrumMasters responsibilities include

1. Teach the Product Owner how to maximize return on investment (ROI), and meet his/her objectives through Scrum.
2. Improve the lives of the development Team by facilitating creativity and empowerment.
3. Improve the productivity of the development Team in any way possible.
4. Improve the engineering practices and tools so that each increment of functionality is potentially shippable.
5. Keep information about the Team’s progress up to date and visible to all parties.

In practical terms, the ScrumMaster needs to understand Scrum well enough to train and mentor the other roles, and educate and assist other stakeholders who are involved in the process. The ScrumMaster should maintain a constant awareness of the status of the project (its progress to date) relative to the expected progress, investigate and facilitate resolution of any roadblocks that hold back progress, and generally be flexible enough to identify and deal with any issues that arise, in any way that is required. The ScrumMaster must protect the Team from disturbance from other people by acting as the interface between the two. The ScrumMaster does not assign tasks to Team members, as task assignment is a Team responsibility. The ScrumMaster’s general approach towards the Team is to encourage and facilitate their decision-making and problem-solving capabilities, so that they can work with increasing efficiency and decreasing need for supervision. The goal is to have a team that is not only empowered to make important decisions, but does so well and routinely.

**Product Owner**

The Product Owner is the keeper of the requirements. The Product Owner provides the “single source of truth” for the Team regarding requirements and their planned order of implementation. In practice, the Product Owner is the interface between the business, the customers, and their product related needs on one side, and the Team on the other

**Team**

The Team is a self-organizing and cross-functional group of people who do the hands-on work of developing and testing the product. Since the Team is responsible for producing the product, it must also have the authority to make decisions about how to perform the work. The Team is therefore self-organizing: Team members decide how to break work into tasks, and how to allocate tasks to individuals, throughout the Sprint. The Team size should be kept in the range from five to nine people, if possible. (A larger number make communication difficult, while a smaller number leads to low productivity and fragility.) Note: A very similar term, “Scrum Team,” refers to the Team plus the ScrumMaster and Product Owner.