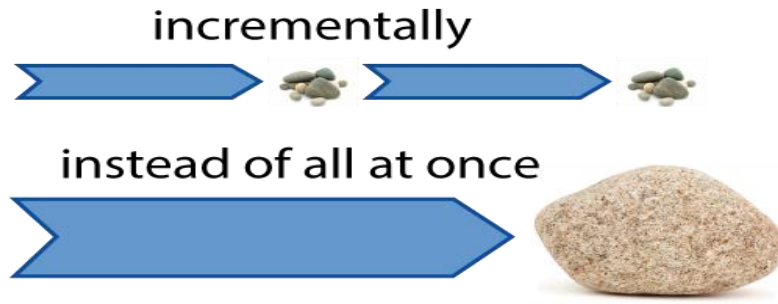
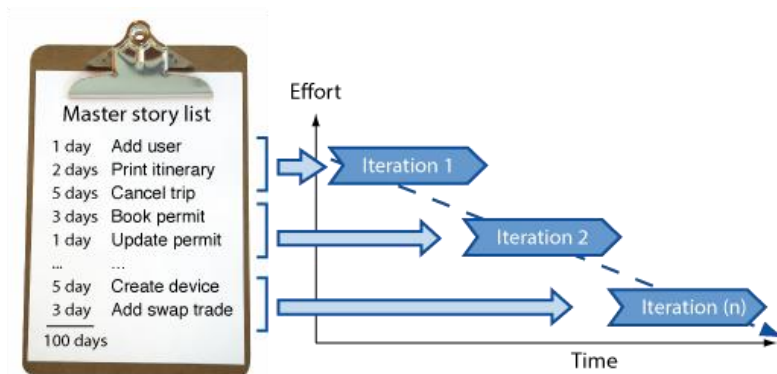


What is Agile?

Agile is a time boxed, iterative approach to software delivery that builds software incrementally from the start of the project, instead of trying to deliver it all at once near the end.



It works by breaking projects down into little bits of user functionality called [user stories](#), prioritizing them, and then continuously delivering them in short two week cycles called [iterations](#) or sprints.



What does Agile mean?

The dictionary meaning of Agile is quick moving. Now how does that apply to software? The Agile development methodology considers software as the most important entity and accepts user requirement changes. Agile advocates that we should accept changes and deliver them in small releases. Agile accepts change as a norm and encourages constant feedback from the end user.

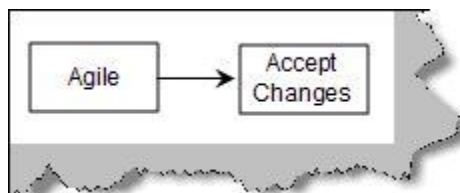
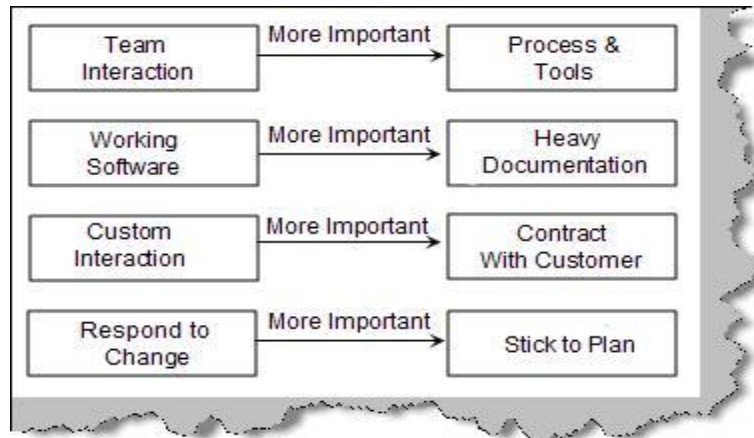


Figure: Agile

The figure below shows how Agile differs in principle from traditional methodologies.

**Figure: Change of Agile thinking**

- It's not necessary to have hi-fi tools and processes but a good team interaction can solve a lot of problems.
- A working software is more important than documentation.
- The management should not just pay attention to customer contracts rather interact with customers and analyze the requirements.
- In traditional methodologies, we pledge to stick to our plans but Agile says "If the customer wants to change, analyze and change your plans accordingly".

Below are the principles of the Agile methodology:

- Welcome changes and adapt to changing requirements.
- A working software is the main measure of progress.
- Customer satisfaction is the most important thing and can be attained by rapid, continuous delivery of useful software.
- Day to day meetings between business people and the development team is a must.
- Business and developers must work together. Face to face communication is the most important thing.
- Deliver and update software regularly. In Agile, we do not deliver software in one go, but rather deliver frequently and deliver the important features first.
- Build projects around teams of motivated and trustful people.
- Design and execution should be kept simple.
- Strive for technical excellence in design and execution.
- Allow teams to organize themselves.

Can you explain Agile modeling?

Agile modeling is an approach to the modeling aspects of software development. It's a practice for modeling and documentation for software systems. In one line:

It's a collection of best practices for software modeling in a light-weight manner.

In abstraction, we can say it augments other software processes. For instance, let's say your company is using UML and then Agile applies approach practices on UML. For example, "Keep things simple" is an Agile approach. So it means that we do not need to use all the diagrams in our project, use only those which are needed. If we summarize, we can say Agile modeling says "Do only what's needed and nothing more".

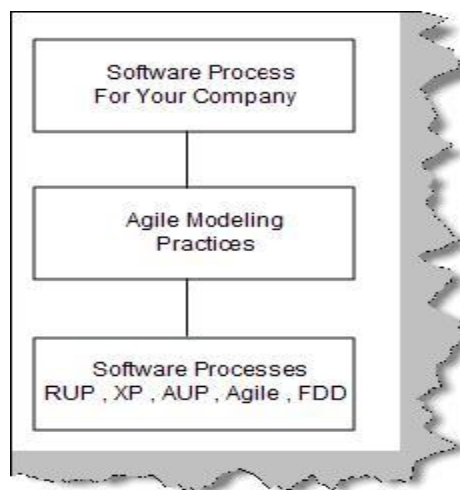


Figure: Agile modeling

What are the core and supplementary principles in Agile modeling?

Agile modeling defines a set of practices which can show us the way towards becoming successful Agile modelers. These practices are divided into two sections: "Core Principles" and "Supplementary Principles". The figure below shows this in a pictorial format:

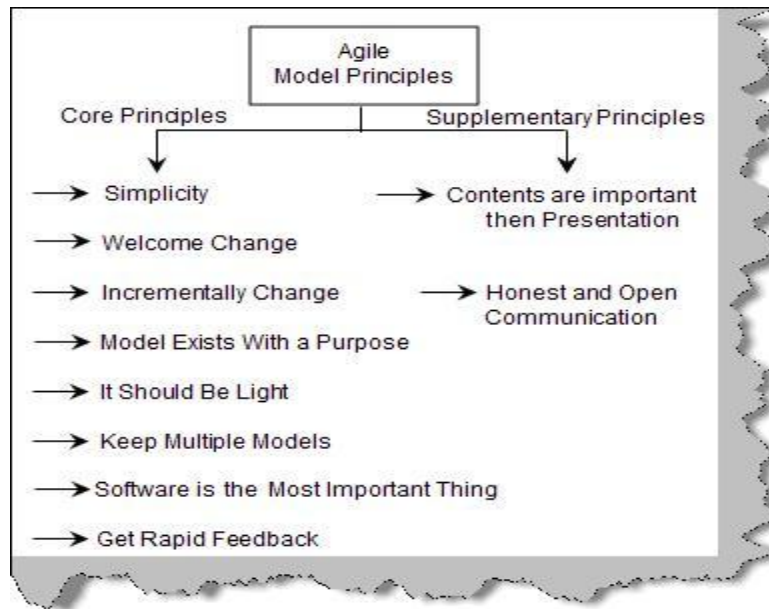


Figure: Agile model principles

Let's understand one by one what these principles mean.

Core principles

- **Simplicity:** Do not make complex models, keep it simple. When you can explain to your team with a pen and paper, do not complicate it by using modeling tools like Rational Rose. Do not add complexity to show off something. If the developer understands only a flow chart, then explain to him with a flow chart; if he understands pseudo-code then use pseudo-code, and so on. So look at your team to see what they understand and prepare documents accordingly.
- **Welcome change:** Requirements grow with time. Users can change requirements as the project moves ahead. In a traditional development cycle, you will always hear the words "freeze the requirement", this has changed with Agile coming in. In Agile, we welcome changes and this is reflected in the projects.
- **Incrementally change:** Nothing can be right at the first place. You can categorize your development with "the most required", "needed features", and "luxury features". In the first phase, try to deliver the "the most required" and then incrementally deliver the other features.
- **Model exists with a purpose:** The model should exist for a purpose and not for the sake of just existing. We should know our target audience for whom the model is made. For instance, if you are making a technical document, it's for developers, a PowerPoint presentation is for the top management, and so on. If the model does not have target audience, then it should not exist. In short, "just deliver enough and not more".
- **It should be light:** Any document or artifact you create should be updated over a period of time. So if you make 10 documents then you should note that as the source code changes, you also need to update those documents. So make it as light as possible. For instance, if your technical document is made of diagrams existing in UML, it becomes important to update all the diagrams over a period of time, which is again a pain. So keep it lightweight, make a simple

technical document, and update it when you have logical ends in the project rather than updating it periodically.

- **Keep multiple models:** Project issues vary from project to project, and the same project behavior can vary from organization to organization. So do not think that one model can solve all issues, keep yourself flexible and think about multiple models. Depending on the situation, apply the models. For instance, if you are using UML for technical documentation, then every diagram in the UML can reflect the same aspects in different ways. For instance, a class diagram shows the static view of the project while a flowchart shows a dynamic view. So keep yourself flexible by using different diagrams and see which best fits your project or a scenario.
- **Software is the most important thing:** The main goal of a software project is to produce high quality software which can be utilized by your end customer in an effective manner. Many projects end up with bulky documents and management artifacts. Documentation is for the software and not software for the documentation. So any document or activity which does not add value to the project should be questioned and validated.
- **Get rapid and regular feedback:** Software is finally made for the user. So try to get feedback on a regular basis from the end user. Do not work in isolation, involve the end user. Work closely with the end customer, get feedback, analyze requirements, and try to meet their needs.

Supplementary principles

- **Content is important than presentation:** The look and feel is not important, rather the content or the message to be delivered by the content is important. For instance, you can represent the project architecture using complex UML diagrams, a simple flow chart, or by using simple text. It will look fancy that you can draw complex UML diagrams but if the end developer does not understand the UML, then it ends nowhere. A simple textual explanation could have met the requirement for communicating your architecture to the developer / programmer.
- **Honest and open communication:** Take suggestions, be honest, and keep your mind open to new models. Be frank with the top management if your project is behind schedule. An open and free environment in a project keeps resources motivated and the project healthy.

What is the main principle behind Agile documentation?

The main deliverable in Agile is a working software and not documentation. Documentation is a support to get the working software. In traditional delivery cycle, a lot of documentation is generated in the design and requirement phases. But we are sure most of the documentation was created just for the sake of it. Below are some of the key points to make documentation Agile:

- Before creating any document, ask the question do we need it and if we do, who is the stakeholder. The document should exist only if needed and not for the sake of existence.
- The most important thing is we need to create documentation to provide enough data and no more than that. It should be simple and should communicate to stakeholders what it needs to communicate. For instance, the below figure 'Agile documentation' shows two views for a simple class diagram. In the first view, we have shown all the properties for the **Customer** and the **Address** classes. Now have a look at the second view where we have only shown the

broader level view of the classes and relationships between them. The second view is enough and not more. If the developer wants to get into the details, we can do that during development.

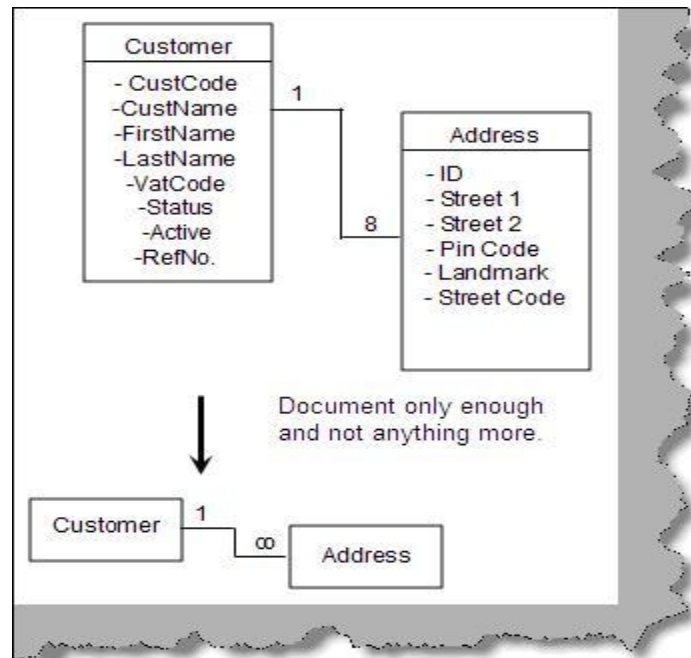


Figure: Agile documentation

- Document only for the present and not for the future. In short, whatever documentation we require now, we should produce and not what we need in the future. Documentation changes its form as it travels through every cycle. For instance, in the requirement phase it's the requirement document, in design it's the technical documentation, and so on. So only think which document you want to create now and not for the future.

What are the different methodologies to implement Agile?

Agile is a thinking approach to software development which promises to remove the issues we have with the traditional waterfall methodology. In order to implement Agile practically in projects, we have various methodologies. The figure 'Agile methodologies' shows this in a detailed manner.

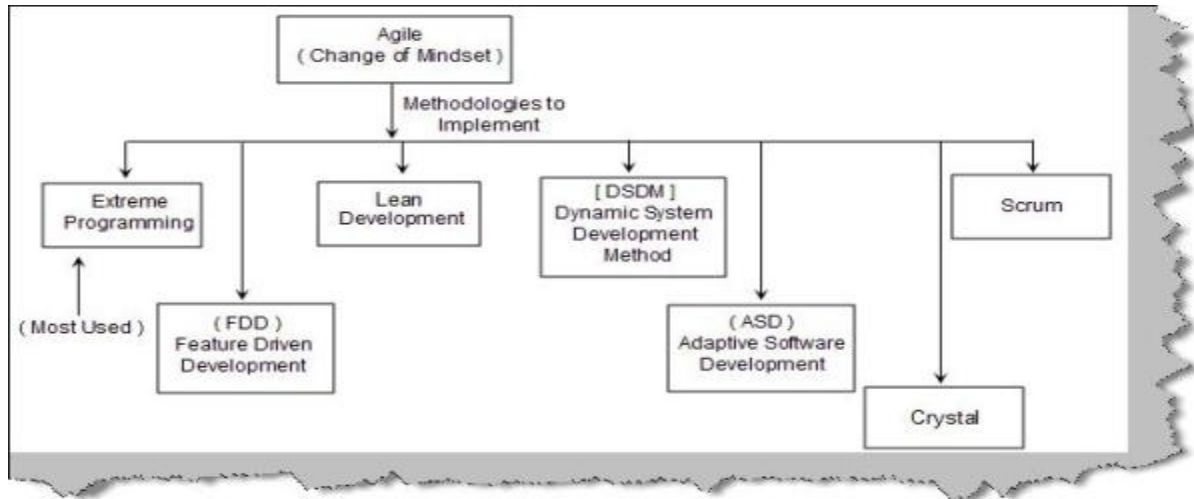


Figure: Agile methodologies

Note: We will cover each methodology in detail in the coming sections.

What is XP?

Extreme Programming (also termed as XP) is an agile software development methodology. XP focuses on coding of software. XP has four core values and fourteen principles.

XP has these four core values:

- **Communication:** The team should communicate on a regular basis, share information, discuss solutions, and so on. Teams that communicate very often are able to solve problems more efficiently. For instance, when an issue gets resolved in a cryptic fashion, send an email to the whole team. This ensures that the knowledge is shared with everyone and in your absence some other developer can solve a similar problem.
- **Simplicity:** Keep things simple. From a process angle, technical angle, or from a documentation point of view. An over complicated process or technical architecture is only calling for problems.
- **Feedback:** Regular feedback from the end user helps us to keep the project on track. So regular feedback should be enabled from the end user and testing team.
- **Courage:** To bring change or to try something new needs courage. When you try to bring changes in an organization, you are faced with huge resistance. Especially when your company is following traditional methodologies, applying XP will always be resisted.

From the above four core values, 14 principles can be derived. The values give a broader level view while the 14 principles go deep into how to implement XP.

- **Rapid feedback:** Developers should receive rapid feedback from the end user. This avoids confusion during the last minute of delivery. In the waterfall model, feedback is received very late. This is minimized in XP.
- **Keep it simple:** Encourage simplicity in the project design and process. For instance, rather than using complex tools, simple handwritten flowcharts on a board can solve a problem.
- **Give incremental changes:** Whenever you update patches and updates, release it in small pieces. If you are updating numerous patches in one go and if there is a defect, it will be difficult to track them.
- **Embrace change:** Do not be rigid with the customer saying that we have already signed the requirement so we can not change the architecture. The customers or users are finally human beings so they can change as the project moves ahead....accept changes if they are logical.
- **Lightweight:** Keep the documentation and process as simple as possible. Do not overdose the developer with unnecessary documentation. A developer's main work is coding and ensuring that the code is defect free, so he should be concentrating on the code rather than the documentation.
- **Deliver quality:** Any code you deliver should be defect free. Be committed to your work and deliver defect free code.
- **Start small and grow big:** Many times the end customer wants to start with a big bang theory. He starts with a big team, wants all the functionalities at the first roll out, and so on. Start with a small team and the "must have" features to be delivered. As we add features and the work load increases, gradually increase your team strength.
- **Play to win:** Take all steps needed to make a project success. Any type of deadline and commitment should be met with true spirit.
- **Encourage honest communication:** Promote honest communication. If communication happens face to face, then there is less leakage of requirements. Encourage the end user to sit with developers and give feedback; this makes your project stronger.
- **Conduct testing honestly:** Test plans should not be created for the sake of creation. Test plans should prove that you are on track record.
- **Adapt according to a situation:** No two projects are the same, no two organizations are same, and different people behave differently. So it's very essential that our approach also adapts according to situations.
- **Metric honesty:** Do not gather metrics for the sake of gathering or showing off to external people how many metrics your project derives. Pick metrics which makes sense to your project and helps you measure your project health.
- **Accept responsibility:** Do not impose or assign people on task which they do not like. Rather question the resource once which tasks he like and assign accordingly. This will increase productivity to a huge level and maintain your project enthusiasm high.
- **Work with people's instincts:** Normally in a project team there are highly motivated people, moderately motivated ones, and people with less motivation. So give power to your motivated team members and encourage them.

What are User Stories in XP and how different are they from requirements?

User Story is nothing but an end user requirement. What differentiates a user story from a requirement is that they are short and sweet. In one sentence, they are just enough and nothing more than that. A user story ideally should be written on index cards. The figure 'User story index card' shows a card. It is a 3 x 5 inches (8 x 13 cm) card. This will keep your stories as small as possible. The requirement document goes in pages. As we are keeping the stories short, it is simple to read and understand. Traditional requirement documents are verbose and they tend to lose the main requirement of the project.

Note: When I was working in a multinational company, I remember the first 50 pages of a requirement document had things like history, backtracking, author of the document, etc. I would be completely drained by the time I started reading the core requirement.

Every story has a title, short description, and estimation. We will come to the estimation part later.

Note: Theoretically, it's good to have cards, but in a real scenario, you will not. We have seen in actual scenario the project manager keeping stories in documents and every story is not more than 15 lines.

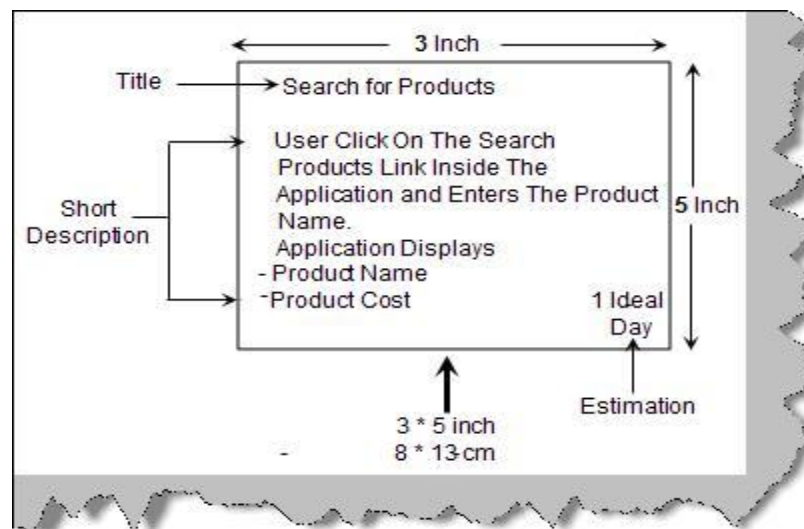


Figure: User story index card

Who writes User Stories?

It's written and owned by the end customer and no one else.

When do we say a story is valid?

A story is valid if it can be estimated.

When are test plans written in XP?

Test plans are written before writing the code.

Can you explain the XP development life cycle?

The XP development cycle consists of two phases: 'Release Planning' and 'Iteration Planning'. In release planning, we decide what should be delivered and in what priority. In iteration planning, we break the requirements into tasks and plan how to deliver those activities decided in release planning. The below figure 'Actual essence' shows what actually these two phases deliver.

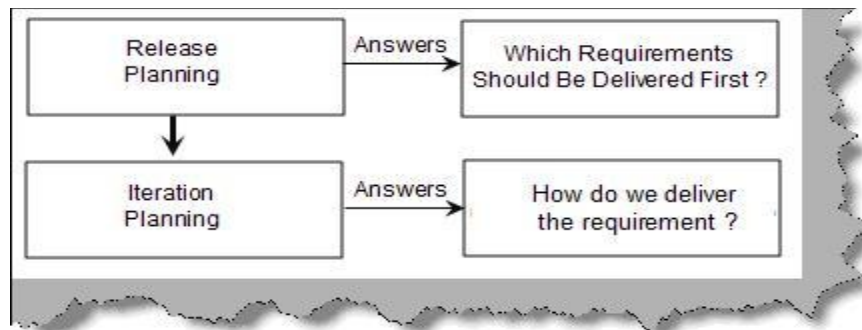


Figure: Actual essence

If you are still having the old SDLC in mind, the below figure 'Mapping to traditional cycle' shows how the two phases map to SDLC.

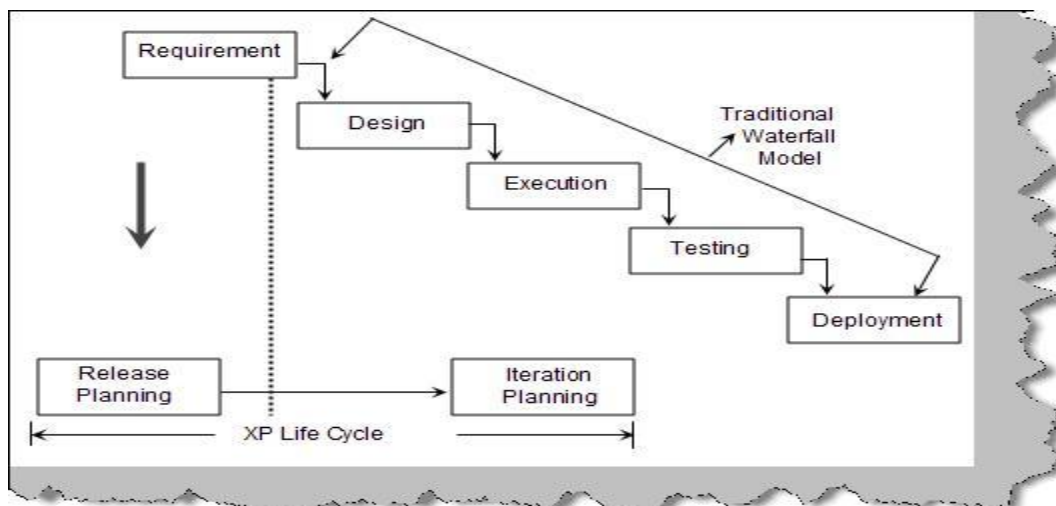
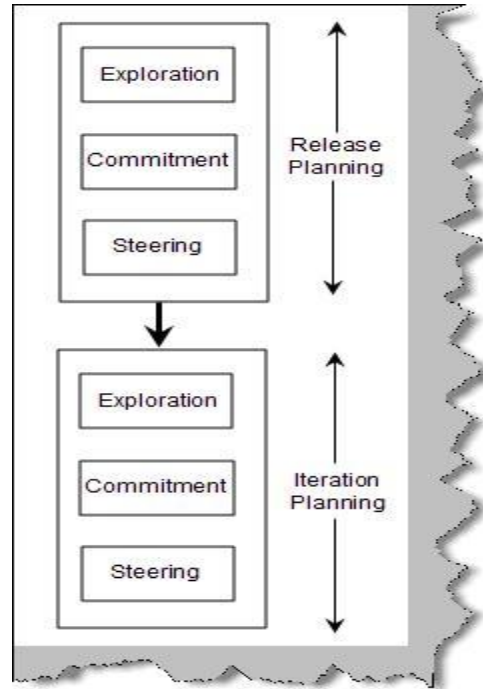


Figure: Mapping to traditional cycle

So let's explore both these phases in a more detailed manner. Both phases "Release Planning" and "Iteration Planning" have three common phases: Exploration, Commitment, and Steering.

**Figure: XP planning cycle**

Release Planning

Release planning happens at the start of each project. In this phase, the project is broken into small releases. Every release is broken down into a collection of user stories. Now let's try to understand the three phases in release planning.

Exploration: In this phase, requirement gathering is done by using the user story concept (please read the previous question on user story to understand the concept of user story). In this phase, we understand the requirement to get a higher level understanding. Please note, only higher level. The user story card is sized normally at 3 X 5 inch, so you can not really go in detail in that size of a card. We think it's absolutely fine rather than writing huge documents; it sounds sensible to have to the point requirement paragraphs. So here is a step by step explanation of how the exploration phase moves:

- The first step is the user writes the story on the user card.
- Once the story is written, the developer analyzes it and determines if we can estimate the user story.

If the developer can not estimate, then it's sent back to the user to revise and elaborate the user story.

- Once the user story is clear and can be estimated, the ideal day or story (read about story point, ideal day, and estimation in the coming questions) are calculated.
- Now it is time to say to the user, OK we can not deliver everything at one go, so can you please prioritize? In this phase, the end user gives ranking to the user stories (in the next section, we will deal with how a user story is ranked).
- Once the user is done with story prioritization, it is time for velocity determination (in the coming section, we have a complete question on velocity determination).
- Agile is all about accepting end customer changes. In this phase, we give a chance to the end user to decide if they want to change anything. If they want to change, we request the user to update the story.
- If everything is OK, we go ahead for iteration planning.

The figure "Release planning" shows the above discussed steps in a pictorial format.

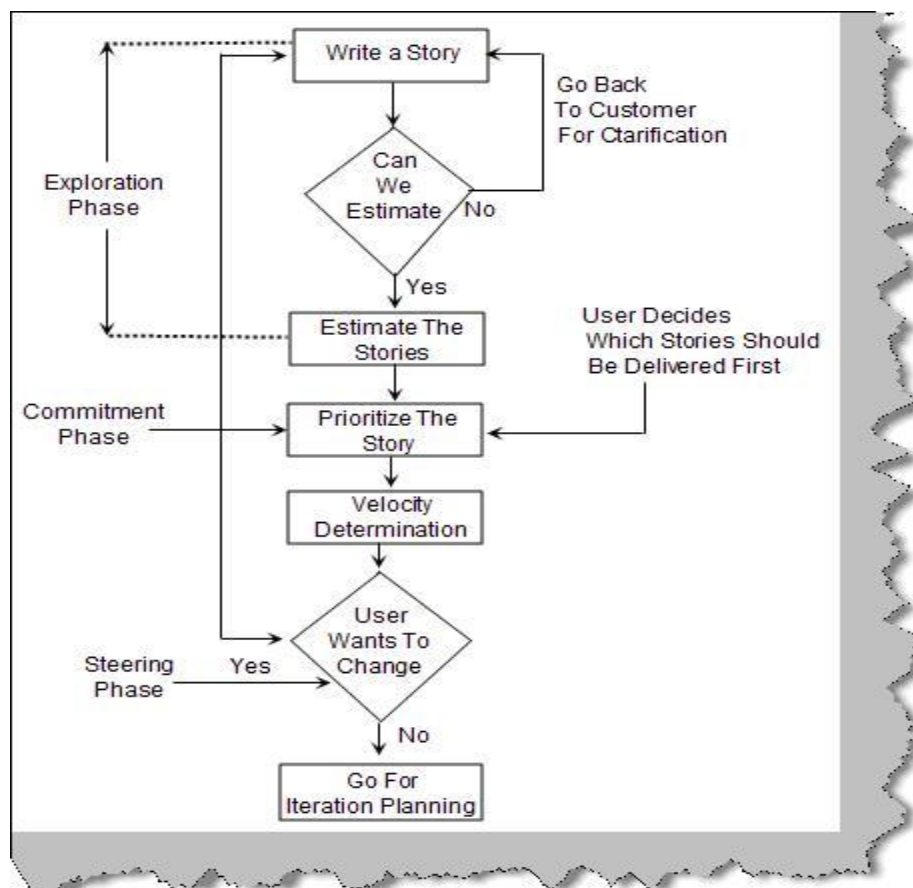


Figure: Release Planning

Iteration Planning

Iteration planning is all about going deep into every user story and breaking them into tasks. This phase can also be termed as detailing of every user story. Iteration planning is all about translating the user story into tasks. Below are the steps in detail for iteration planning:

- User stories which need to be delivered in this iteration are broken down into manageable tasks.
- Every task is then estimated. The result of the estimation is either ideal hours or task points (we will discuss about task points and ideal hours in the coming section).
- After the tasks are estimated, we need to assign the tasks to developers. Each programmer picks a task and owns responsibility to complete the task.
- Once a developer owns the responsibility, he should estimate the work and commit to completing it.
- In XP, on any development task, two developers should be working. In this phase, the developer makes the partner of his choice for developing a task.
- In this phase, we do the designing of the task. We should not make lengthy and comprehensive design plans; rather it should be small and concentrated on the task. In traditional SDLC, we have a fully devoted phase for designing and the output is a lengthy and complicated design document. One of the important characteristic of a software project is that as we come near execution, we are clearer. So it's best to prepare the design just before execution.
- Now that you and your partner are familiar with the design plan, it is time to write a test plan. This is one of the huge differences when compared to the traditional SDLC. We first write the test plan and then start execution. Writing test plans before coding gives you a clear view of what is expected from the code.
- Once the test plan is completed, it is time to execute the code.
- In this phase, we run the test plan and see if all test plans pass.
- Nothing is perfect, it has to be made perfect. Once you are done with coding, review the code to see if there is any scope for refactoring (refactoring is explained in more depth in the coming sections).
- We then run the functional test to ensure everything is up to the mark.

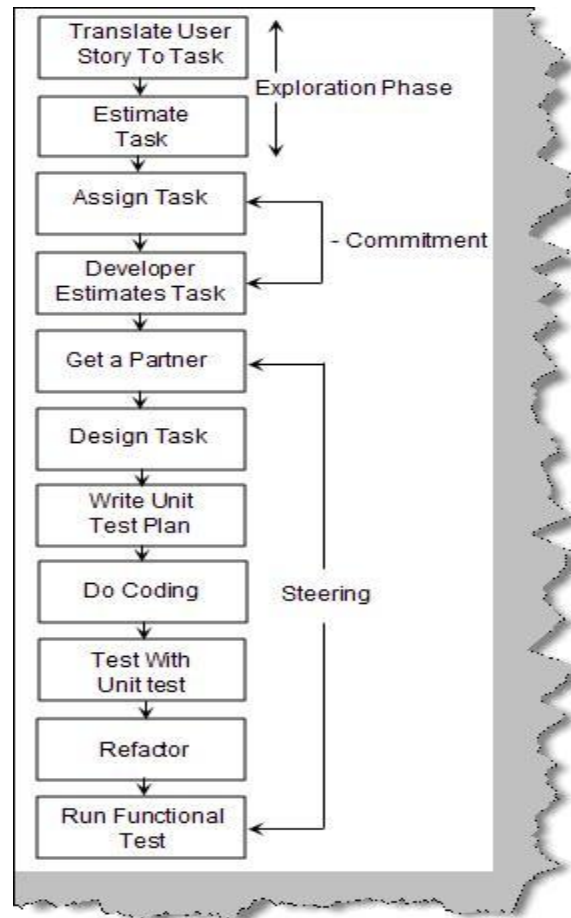


Figure: Iteration planning

One of the important points to realize is a project is broken down into a set of releases which are further analyzed using short user stories, the user stories are further broken into tasks, which are estimated and executed by the developer. Once a release is done, the next release is taken for delivery. For instance, the project shown in figure 'Release, stories, and task' has two releases.

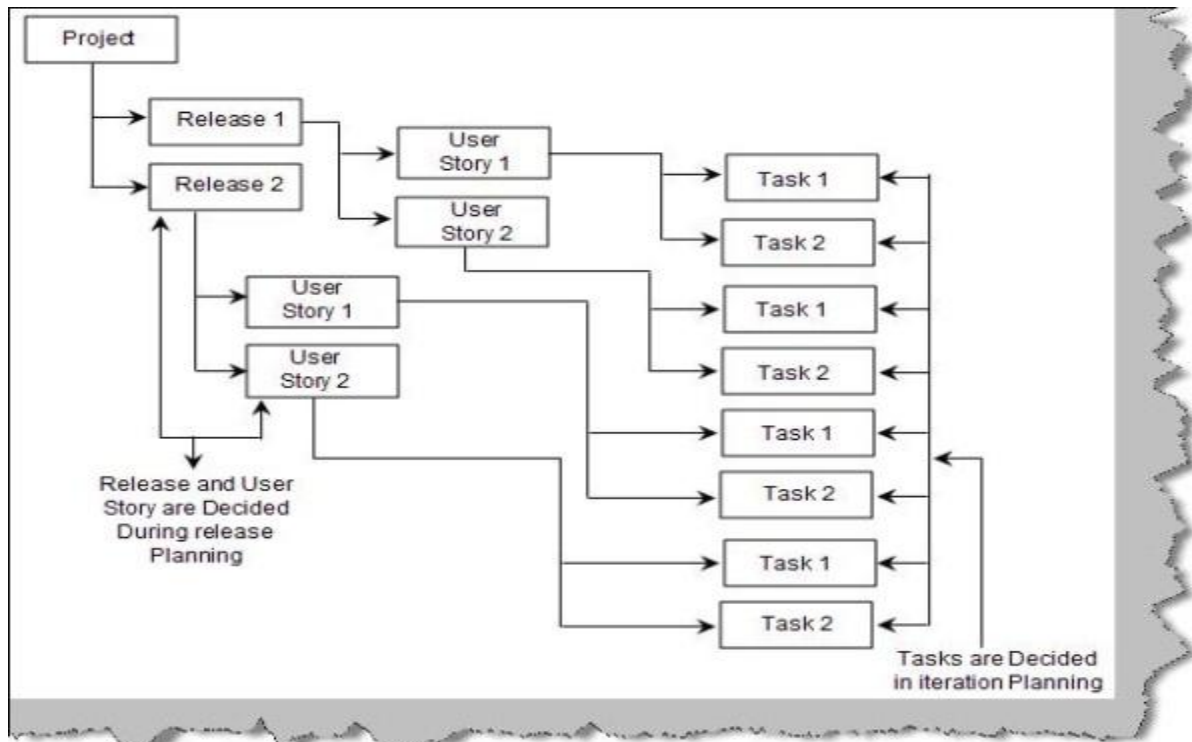


Figure: Release, stories, and tasks

Can you explain how the planning game works in Extreme Programming?

The answer to the above question answers this question as well.

How do we estimate in Agile?

If you read the Agile cycle carefully (explained in the previous section), you will see Agile estimation happens at two places:

- **User Story Level Estimation:** In this level, a user story is estimated using Iteration Team velocity and the output is Ideal man-days or Story points.
- **Task Level Estimation:** This is a second level estimation. This estimation is at the developer level according to the task assigned. This estimation ensures that the user story estimation is verified.

Estimation happens at two levels: when we take the requirement, and when we are very near to execution - that's at the task level. This looks very much logical because as we are very near to completing a task, estimation is more and more clear. So task level estimation just comes as a cross verification for user story level estimation.

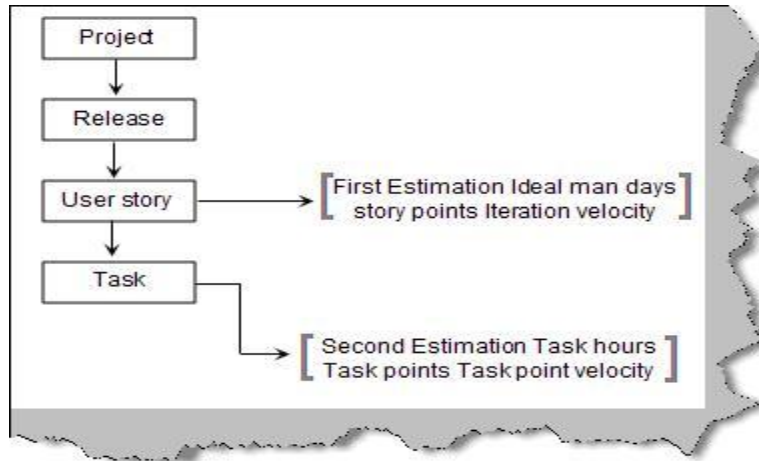


Figure: Agile estimation

User story level estimation

Estimation unit at user story in Agile is either “ideal days” or “story points”.

Ideal days are nothing but the actual time the developer spends or will spend on only coding. For instance, attending phone calls, meetings, time spent eating lunch and breakfast, etc., are not included in ideal days. In old estimation technologies, we estimate eight hours as the complete time a developer will do coding. But actually a developer does not code continuously for eight hours, so the estimates can be wrong if we consider the full eight hour days.

Estimation units can also be represented in story points. **Story points** are abstract units given to represent the size of a story. In a normal scenario, a story point equals an ideal day. A story point is a relative measure. If a story is one story point and another is two story points, that means the second story will take twice the effort compared to the first story.

Velocity determination defines how many user stories can be completed in one iteration. So first the user decides the length of the iteration. The length of the iteration is decided depending on the release dates. Velocity is normally determined from history. So whatever was the last team history velocity, the same will be used in further estimation. But if there is no history, then the formula below will be used:

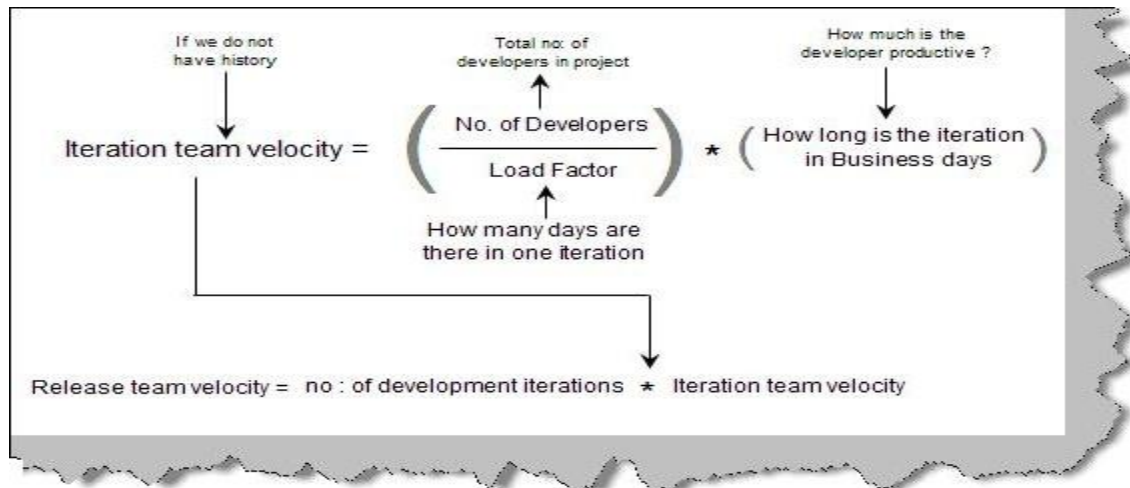


Figure: Velocity determination

There are two formulas in the above figure: the first formula is used when we do not have history about the project and the second formulae is used when we have a history of the iteration. Below are the details of all the parameters in the formulae:

- Number of developers: Total number of developers in the iteration.
- Load factor: This means how much productive time a developer will spend on the project. For instance, if the load factor is 2 then the developers are only 50% productive.
- How long is the iteration in business days: One iteration is how many man days.

The figure 'Iteration and release calculation' shows a simple sample with a team size of 5, load factor of 2; an iteration takes 11 business days and there are two releases in the project.

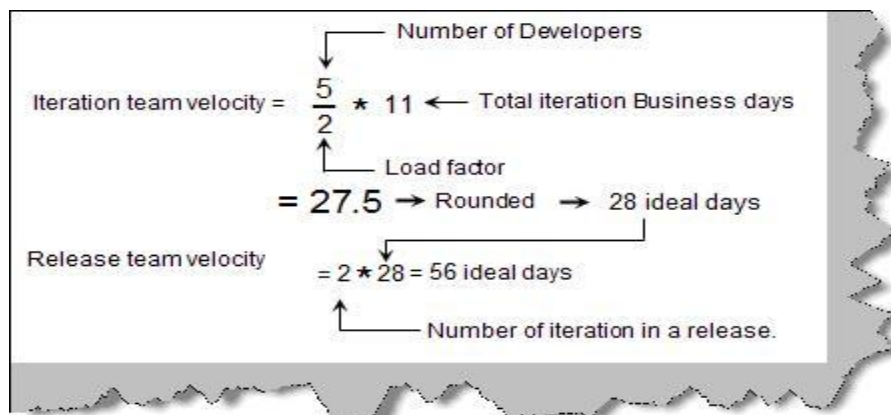
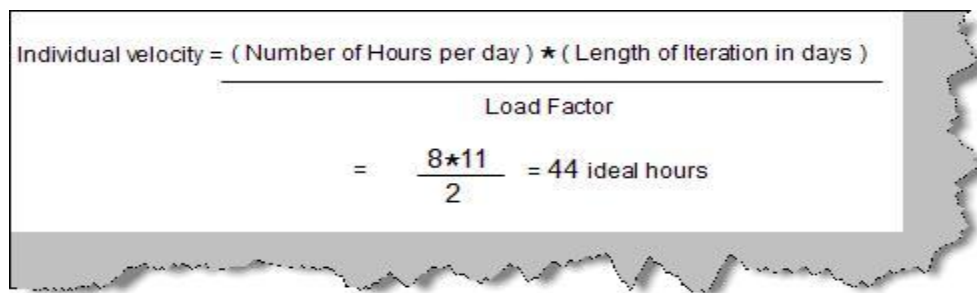


Figure: Iteration and release calculation

Task level estimation

As the Agile cycle moves ahead, a user story is broken down into tasks and assigned to developers. The level of effort at the task level is a form of task points or ideal hours. Ideally, a task point represents an ideal hour. An ideal hour is the time a developer spends only on coding and nothing else.

Individual velocity determination defines how many ideal hours a developer has within an iteration. The below figure 'Individual velocity calculation' shows in detail how to get the number of ideal hours in an iteration for a developer. Below is a sample calculation which shows with 8 hours a day, an iteration of 11 days, and a load factor of 2 (i.e., developer codes for only 50% time, i.e., 4 hours), we get 44 ideal hours for that developer in that iteration.



$$\text{Individual velocity} = \frac{(\text{Number of Hours per day}) \star (\text{Length of Iteration in days})}{\text{Load Factor}}$$

$$= \frac{8 \star 11}{2} = 44 \text{ ideal hours}$$

Figure: Individual velocity calculation

On what basis can stories be prioritized?

A user story should normally be prioritized from the business importance point of view. In real scenarios, this is not the only criteria. Below are some of the factors to be accounted when prioritizing user stories:

- **Prioritize by business value:** A business user assigns a value according to the business needs. There are three levels of rating for business value:
 - **Most important features:** Without these features, the software has no meaning.
 - **Important features:** It's important to have these features. But if these features do not exist, there are alternatives by which the user can manage.
 - **Nice to have features:** These features are not essential features but over the top cream for the end user.
- **Prioritize by risk:** This factor helps us prioritize by risk from the development angle. The risk index is assigned from 0 to 2 and is classified in three main categories:
 - **Completeness**
 - **Volatility**
 - **Complexity**

The below figure "Risk index" shows the values and the classification.

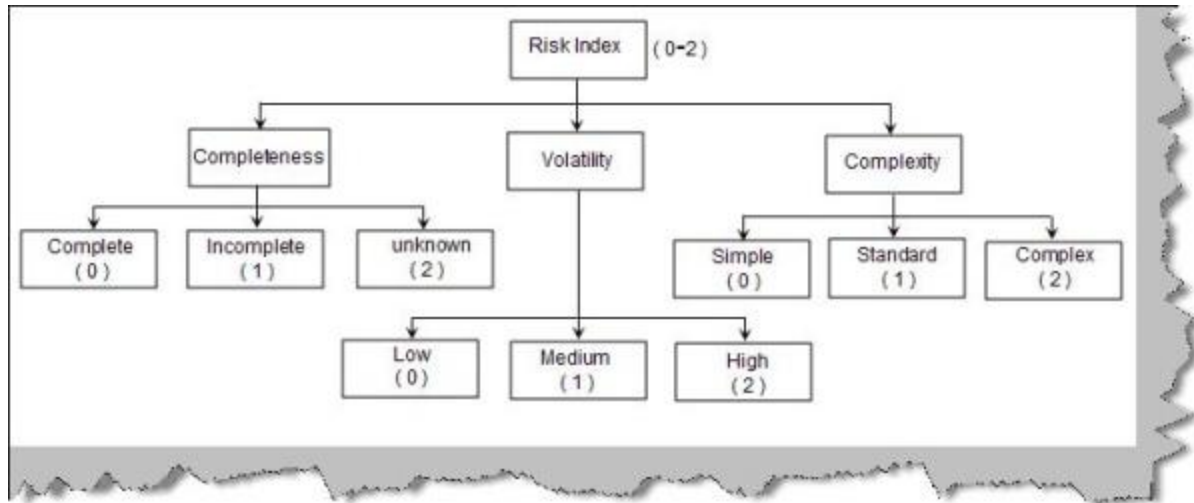


Figure: Risk index

Can you point out the simple differences between Agile and traditional SDLC?

The figure “Agile and traditional SDLC” points out some of the main differences. If you have worked on both these, you can point out more differences.

- Lengthy requirement documents are now simple and short user stories.
- Estimation unit man days and man hours are now ideal days and ideal hours, respectively.
- In the traditional approach, we freeze the requirement and complete the full design and then start coding. But in Agile, we design task wise. So the developer designs just before he starts a task.
- In traditional SDLC, we always hear ‘After signoff nothing can be changed’; in Agile, we work for the customer, so we accept changes.
- Unit test plans are written after coding or during coding in traditional SDLC. In Agile, we write unit test plans before writing the code.

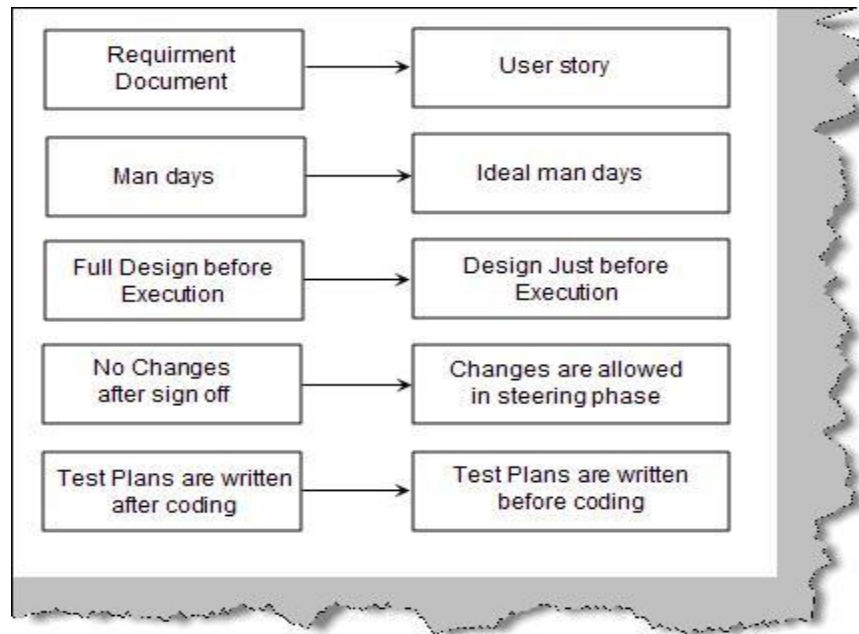


Figure: Agile and traditional SDLC