Agile Best Practices:

To attain agility(rapid and adaptive response to changes) we use agile best practices.

Following are the best practices of agile.

1. Iterations.
2. Customer-oriented approach.
3. Product backlog.
4. User stories.
5. Agile roles.
6. Value stream analysis.
7. Timeboxing.
8. Scrum meetings.
9. Sprint demo meetings.
10. Retrospective meetings.
11. Integration.
12. Test-driven development.
13. Burndown chart.
14. Automated tests.
15. Requirement prioritization.
16. In-pairs programming.
17. Release planning.
18. Refactoring.
19. Small release cycles.
20. Coding standard
21. Collective ownership
22. Behaviour driven development.
23. Document late

Iteration:

An iteration, in the context of an Agile project, is a timebox during which development takes place, the duration of which:

- may vary from project to project, usually between 1 and 4 weeks
- is in most cases fixed for the duration of a given project

A key feature of Agile approaches is the underlying assumption that a project consists exclusively of a sequence of iterations, possibly with the exception of a very brief "vision and planning" phase prior to development, and a similarly brief "closure" phase after it.
In general iterations are aligned with calendar weeks, often starting on Mondays, and ending on Fridays; this is more a matter of convenience than an explicit recommendation and many teams adopt different conventions.

The fixed length of iterations gives teams a simple way to obtain, based on velocity and the amount of work remaining, a usually accurate (though not very precise) estimation of the project's remaining duration.
Iteration execution is the process of how the work takes place. During the iteration, the team completes the 'do' portion of the cycle by building and testing the new functionality. Teams deliver Stories incrementally, demoing their work to the Product Owner as soon as they are done, enabling teams to arrive at the iteration review ready to show their completed work.
The iteration review is the 'check' step in the cycle. This review is where the teams demonstrate a tested increment of value to the Product Owner, and other relevant customer,
and receive feedback on what they've produced. The iteration review provides the opportunity to assess progress as well as make any adjustments ahead of the next iteration. Some stories will be accepted; others will be refined by the insights gained during the iteration. The team will then do some final backlog refinement for the upcoming iteration planning.

Empowering Agile teams to focus on rapid value delivery fuels them with energy, motivation, and purpose. It instils a better sense of mission than traditional management and development models. The centrepiece of this approach is developing high-quality system increments every iteration. Teams employ a variety of practices to achieve that result, but the focus is always the same: to deliver the stories they committed to during iteration planning to meet their Iteration Goals.
The iteration retrospective is the 'adjust' step for the overall iteration. Here, the team evaluates its process and reviews any improvement stories it had from the previous iteration. They identify new problems and their causes—as well as emphasizing bright spots—and create improvement stories that enter the team backlog for the next iteration. This regular reflection is one of the ways to ensure relentless improvement is happening within each team. Iteration retrospectives may also identify systemic problems that will need to be addressed at the next Inspect and Adapt (I&A) event.

Customer Oriented Approach:

In Customer Oriented Approach, agile team should **provide all the information needed to the clients** and inform them of the **progress**.

Customer communication: It is simply not enough for an agile product development organization to create great code and ship the resulting product like a clockwork. You also need to talk about it, particularly at the beginning of your agile transition. Marketing the agile journey of product and engineering to the rest of the organization—and thus getting their buy-in—is a critical success factor to step up the game: You want to become agile, not "do agile."
Do good and talk about it—a simple necessity, particularly if your agile transition is supposed to be embraced by the whole organization over time. Deciding early in the process how to communicate with internal stakeholders usually makes the difference between "doing agile" and "becoming agile" in the end.

Keep in mind that a lot of stakeholder egos, as well as personal agendas, are tied one way or another to executing "the plan". And you're trying to sell them that quitting this plan will turn out to be mutually beneficial.
Developing empathy for stakeholders is particularly relevant, when engineering and product don't have the best standing within the company at the beginning of the transition. The good news is, that no matter whether both are perceived as a black hole by others, a well communication strategy has a good chance to win over the rest of the organization.

The Agile Manifesto also addresses communication, valuing working software over comprehensive documentation. Although documentation has value, working functionality has more importance on an agile project.

Agile Customer Experience means more than quick fixes. Doing the quick and easy things to silence those customers who choose to give their feedback or complain, may not have a great impact on or value for the customer, in the long run. If, however, an agile approach to CX is employed to develop and introduce small changes that combined have cumulative big impact for customers, then it could be what is needed.

An agile business mindset or strategy would help businesses move towards greater customer centricity if done well, i.e. If it focuses on customer intelligence rather than quick wins, at its heart. Perhaps there is an argument for such an approach instead of a huge, all-encompassing customer transformation programme.

Agile would mean that Customer Experience change, or improvements are easier to implement, benefits can be seen more quickly, and the process is more inclusive. It would also bring a broader range of people from around the business on the journey through day to day agile team-based working.

Product Backlog:

A product backlog is a list of the new features, changes to existing features, bug fixes, infrastructure changes or other activities that a team may deliver in order to achieve a specific outcome.

The product backlog is the single authoritative source for things that a team works on. That means that nothing gets done that isn't on the product backlog. Conversely, the presence of a product backlog item on a product backlog does not guarantee that it will be delivered. It represents an option the team has for delivering a specific outcome rather than a commitment.

It should be cheap and fast to add a product backlog item to the product backlog, and it should be equally as easy to remove a product backlog item that does not result in direct progress to achieving the desired outcome or enable progress toward the outcome.

Product backlog items take a variety of formats, with user stories being the most common. The team using the product backlog determines the format they chose to use and look to the backlog items as reminders of the aspects of a solution they may work on.
Product backlog items vary in size and extent of detail based in large part in how soon a team will work on them. Those that a team will work on soon should be small in size and contain sufficient detail for the team to start work. The team may establish a definition of ready to indicate their agreement on the information they'd like to have available in order to start working on a product backlog item. Product backlog items that are not slated for work may be fairly broad and have little detail.
The sequence of product backlog items on a product backlog changes as a team gains a better understanding of the outcome and the identified solution. This reordering of existing product backlog items, the ongoing addition and removal of product backlog items, and the continuous refinement of product backlog items gives a product backlog its dynamic characteristic.

A team owns its product backlog and may have a specific role – product owner – with the primary responsibility for maintaining the product backlog. The key activities in maintaining the product backlog include prioritizing product backlog items, deciding which product backlog items should be removed from the product backlog, and facilitating product backlog refinement.
A product backlog can be an effective way for a team to communicate what they are working on and what they plan to work on next. Story Maps and information radiators can provide a clear picture of your backlog for the team and stakeholders.

User Stories:

A user story in Agile project management means a unit of work that should be completed in one Sprint.

A 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.

The purpose of a user story is articulate how a piece of work will deliver a particular value back to the customer. Note that "customers" don't have to be external end users in the traditional sense, they can also be internal customers or colleagues within your organization who depend on your team.

User stories are a few sentences in simple language that outline the desired outcome. They don't go into detail. Requirements are added later, once agreed upon by the team.

User stories are also the building blocks of larger agile frameworks like epics and initiatives. Epics are large work items broken down into a set of stories, and multiple epics comprise an initiative. These larger structures ensure that the day to day work of the development team (on stores) contributes to the organizational goals built into epics and initiatives.

User stories serve a number of key benefits:

- **Stories keep the focus on the user.** A To Do list keeps the team focused on tasks that need checked off, but a collection of stories keeps the team focused on solving problems for real users.
- **Stories enable collaboration.** With the end goal defined, the team can work together to decide how best to serve the user and meet that goal.
- **Stories drive creative solutions.** Stories encourage the team to think critically and creatively about how to best solve for an end goal.
- **Stories create momentum.** With each passing story the development team enjoys a small challenge and a small win, driving momentum.

Once a story has been written, it's time to integrate it into your workflow. Generally, a story is written by the product owner, product manager, or program manager and submitted for review.

During a sprint or iteration planning meeting, the team decides what stories they'll tackle that sprint. Teams now discuss the requirements and functionality that each user story requires. This is an opportunity to get technical and creative in the team's implementation of the story. Once agreed upon, these requirements are added to the story.

Agile Roles:

There are several roles, which have different names depending on the methodology being followed, common to agile teams. Roles are not positions, any given person takes on one or more roles and can switch roles over time, and any given role may have zero or more people in it at any given point in a project. The common agile roles are:

- **Team lead**. This role, called "Scrum Master" in Scrum or team coach or project lead in other methods, is responsible for facilitating the team, obtaining resources for it, and protecting it from problems. This role encompasses the soft skills of project management but not the technical ones such as planning and scheduling, activities which are better left to the team as a whole (more on this later).
- **Team member**. This role, sometimes referred to as developer or programmer, is responsible for the creation and delivery of a system. This includes modeling, programming, testing, and release activities, as well as others.
- **Product owner**. The product owner, called on-site customer in XP and active stakeholder in AM, represents the stakeholders. This is the one person responsible on a team (or sub-team for large projects) who is responsible for the prioritized work item list (called a product backlog in Scrum), for making decisions in a timely manner, and for providing information in a timely manner.
- **Stakeholder**. A stakeholder is anyone who is a direct user, indirect user, manager of users, senior manager, operations staff member, the "gold owner" who funds the project, support (help desk) staff member, auditors, your program/portfolio manager, developers working on other systems that integrate or interact with the one under development, or maintenance professionals potentially affected by the development and/or deployment of a software project.
- **Technical experts**. Sometimes the team needs the help of technical experts, such as build masters to set up their build scripts or an agile DBA to help design and test their database. Technical experts are brought in on an as-needed, temporary basis, to help the team overcome a difficult problem and to transfer their skills to one or more developers on the team.
- **Domain experts**. As you can see in Figure 2 the product owner represents a wide range of stakeholders, not just end users, and in practice it isn't reasonable to expect them to be experts at every single nuance in your domain. As a result the product owner will sometimes bring in domain experts to work with the team, perhaps a tax expert to explain the details of a requirement or the sponsoring executive to explain the vision for the project.
- **Independent tester**. Effective agile teams often have an independent test team working in parallel that validates their work throughout the lifecycle. This is an optional role, typically adopted only on very complex projects (or at scale).
- **Scrum master** A Scrum Master holds a position that's relatively narrow in scope, yet extremely broad in influence throughout any organization. In practice, however, a Scrum Master is working behind the scenes and is not involved in product ideation or strategy. They work more as a conduit between product/line-of-business owners and development teams as a project manager. Because agile processes are entirely dependent on people and collaboration, Scrum Masters must also marry soft skills with the latest tools and methods.

Value Stream Analysis:

It is a simple assessment of where value is added and where non-value (waste) accumulates in a process. Value stream analysis involves drawing up a process flow chart for the business and then asking, at each stage (including the stages between activities) whether cost/waste or value is being added. This often highlights unnecessary space, distance travelled, processing inefficiency, etc.

Timeboxes can be used at varying time scales.

It can be applied equally well to service activities - for example, the process of carrying out paper processing in sales or in developing insurance quotations or processing claims.

It is used to highlight improvement areas for process innovation. It is similar to value analysis which is used in product development and improvement.

As part of agile, it can highlight areas for improvement - for example, by elimination or re-design of process stages.

This technique is designed to identify internal strengths and weaknesses - what is it about the firm which helps achieve the strategic goal of competitive advantage - and what gets in the way?

It can be applied inside the firm or along its wider supply and distribution chain, but the principle is the same.

The overall goal is giving the customer what they want - providing customer value - and the challenge is to find any and every place where value is not being added.

It could be because of a poor machine or a duplicated process or lengthy queues or… The point is that such an analysis quickly focuses on where change is needed and throws up opportunities for change

It is based on the idea that the firm consists of a sequence of activities, each of which is designed to add some value to the product or service as it moves through. Eventually it finds its way to the customer.

At each stage we hope that value is being added but of course there are also costs of running the relevant activities, etc.

There is also an overhead which adds to the cost and supports the overall running of the business.

<u>Time Boxing:</u>

Timeboxing refers to the act of putting strict time boundaries around an action or activity. For example, you may want to timebox a meeting to be 30 minutes long to help ensure that the meeting will begin and end on time with no exceptions. When you timebox an event the result is a natural tendency to focus on the most important “stuff” first. If you time box a meeting, you would expect then that the absolutely required content of the meeting be completed before the end of the meeting. Other meeting agenda items that are not as important may get moved to another meeting.

In essence, timeboxing is a constraint used by teams to help focus on value.

One important timebox that Agile promotes is the project itself. Contrary to Agile mythology, Agile teams prefer to have a timeboxed project since it offers a fixed schedule and a fixed team size. With these project constraints the team can better work with customers to maintain a laser focus on value, which ensures the team is building and delivering the most valuable work as soon as possible leaving the less critical tasks to the end.

Timeboxed projects may mean that some requirements won’t get implemented. However, it will help to ensure that what is developed is truly the most essential of the required features.

Timeboxing also helps to prevent a common problem in software development called “feature creep,” where teams incrementally add features to software without scrutinizing relevance or need.

Feature creep leads to wasted effort in both the development and maintenance of code and can significantly impact quality and timelines on a project.

Timeboxed project teams work to minimize the effort and resources needed to achieve the expected value. Some refer to this as the minimum viable product or the minimum viable feature set. Timeboxing iterations places emphasis on ensuring that teams do not experience feature creep.

Agile teams not only like to have timeboxed projects, they also prefer to break projects down into smaller timeboxed durations, commonly referred to as iterations.

Scrum Meetings:

Sprint Planning meeting In Scrum, every iteration begins with a sprint planning meeting. At this meeting, the Product Owner and the team negotiate which stories a team will tackle that sprint. This meeting is a time-boxed *conversation* between the Product Owner and the team. It's up to the Product Owner to decide which stories are of the highest priority to the release and which will generate the highest business value, but the team has the power to push back and voice concerns or impediments.
When the team agrees to tackle the work, the Product Owner adds the corresponding stories into the sprint backlog. We usually recommend this be physically represented by moving a Post-It note or index card with a story written on it from the backlog into the sprint backlog.
The Daily Standup Every day, the Scrum team gathers in front of their talkboard to discuss the progress made yesterday, goals for today, and any impediments blocking their path.

- What have I done since the last Scrum meeting (yesterday)?
- What will I do before the next Scrum meeting (tomorrow)?
- What prevents me from performing my work as well as possible?

This meeting should not exceed 15 minutes. If members of the team need to discuss an issue that cannot be covered in that amount of time, we recommend they attend a *sidebar* meeting following the stand-up. This allows team members to attend meetings that directly involve their work, instead of sitting through irrelevant meetings.
Unfortunately, daily Scrums often last longer than 15 minutes. To compensate, many teams use stop watches or timers to uphold the time limitations. Also, to limit distracting small talk, many teams employ a talking stick or mascot, which a team member must hold to speak in the meeting
Sprint Review Meeting When the sprint ends, it's time for the team(s) to demonstrate a *potentially shippable product increment* to the Product Owner and other stakeholders. The Product Owner declares which items are truly done or not. Teams commonly discover that a story's final touches often excise the most effort and time. Partially done work should not be called "done."
This public demonstration replaces status meetings and reports, as those things do not aid transparency. Scrum emphasizes *empirical* observations such as working products.

Sprint Retrospective Meeting After the sprint review meeting, the team and the Scrum Master get together in private for the retrospective meeting. During this meeting, the team inspects and adapts their process. When the Scrum Master and outer organization create an environment of psychological safety, team members can speak frankly about what occurred during the Sprint and how they felt about it.

Sprint Demo Meetings:

The sprint demo is invaluable for keeping stakeholders up to speed with the progress of product development. It allows them to feedback and discuss with the Product Owner and Scrum team any possible amendments to the Product Backlog which would help to maximise value.

Such discussions can inform the planning of the next sprint and the contents of the next sprint backlog. They will often result in stories being added further down the product backlog too.
The sprint demo takes place at the end of the sprint and is attended by the whole Scrum team, including Product Owner and ScrumMaster, as well as relevant stakeholders, management and developers from other teams.

When an organisation has more than one Scrum team working on the same project the teams should consider running their demo together. It's not just easier to arrange this way – it keeps teams abreast of what the others are working on, which facilitates the sharing of insight and helps to avoid the duplication of work. However, at a certain size this may not be practical and representatives from Scrum teams attending each other's demos may be more practical.

Each Scrum team takes it in turn to update on their progress against their sprint goal and demo the working iteration of the product that resulted from their sprint.

Sometimes the team will nominate a member to present the demo and sometimes the task will be shared, with individuals demonstrating the particular parts of the increment they worked on. Again, a personal preference, but I think demos work well where one person leads on the talking and one leads on the demo element.

The sprint demo shouldn't take up too much of a Scrum team's time. Ensuring that the sprint review meeting is an informal affair – where questions, feedback and discussion are welcome – allows for prep time to be kept to a minimum.
Time should not be spent putting long slide decks together – the focus should be on the work and the demo should only include stories that meet the team's definition of Done.

Generally a day or two before the end of the sprint I hold a short demo run-through with the team in which we agree the order of the stories in the demo – and make notes on anything we need to set up in order to make the demo flow well. This meeting is kept short (say 15 mins) – but ensures the team have thought the demo through.

Retrospective Meetings:

An Agile retrospective is a meeting that's held at the end of an iteration in Agile software development (ASD). During the retrospective, the team reflects on what happened in the iteration and identifies actions for improvement going forward.

Each member of the team members answers the following questions:

- What worked well for us?
- What did not work well for us?
- What actions can we take to improve our process going forward?

The Agile retrospective can be thought of as a "lessons learned" meeting. The team reflects on how everything went and then decides what changes they want to make in the next iteration.

The retrospective is team-driven, and team members should decide together how the meetings will be run and how decisions will be made about improvements.

An atmosphere of honesty and trust is needed in order for every member to feel comfortable sharing their thoughts.

Because Agile stresses the importance of continuous improvement, having a regular Agile retrospective is one of the most important of Agile development practices.

The Ninth Agile principle outlined in the Agile manifesto states, "At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly".

A final point is that attendance at sprint review and retrospective meetings is not optional to development team members. Regular face to face attendance should not be a problem for collocated teams. However, for geographically distributed development teams, getting together in the same location, while possible, would be very expensive and impracticable. Conference telephone calls can be set-up as a substitute to face to face meetings.

Coming up with the best solution for distributed teams to physically meet and work together is very important in all scrum meetings and should be a key managerial concern. Distributed development teams never perform as well as a collocated team, but since it is not always possible, organizations must find ways to help distributed teams work as best they can.

Integration:

Agile integration is a software development practice that favors short, continuous code releases over longer cycles in an attempt to minimize instability and complexity when releasing software.

Agile integration is a methodology that employs automation, strict version control and immediate issue resolution in order to ensure that a stable update is available for release at any moment. The goal is higher quality software with fewer defects and a more controlled, manageable development process.

The principles of agile integration, also commonly referred to as continuous integration, arose in the 1990s in part to address perceived weaknesses of the traditional waterfall development process. Today it is a cornerstone of the agile methodology, which has steadily gained credibility and adherence over the last two decades.

Because agile integration focuses on small, frequent releases, it is used widely in customer or consumer web-based software products or mobile applications in which daily or even hourly releases are common. However, the overarching principle of smaller releases and greater automation in build-and-test processes have gained broad acceptance, even in organizations that have not formally adopted agile methodology.

Agile software development values frequent integration. “Frequent” can vary by team, but it’s rarely truly continuous. A change requires work on the developer’s part, so delays between the time when code is changed and when it is integrated are almost impossible to avoid.

In the case of a team that values frequent integration, the question is *how* frequent. In many cases, the practical answer is that code be integrated to the master once or twice a day.

Agile integration is the approach that allows those microservices to be dropped into the architecture seamlessly or to be removed or updated without disrupting other services. Routing, orchestration, messaging, or data services are the points where a new service interacts with the environment. This is an application or service level perspective of integration.

Test Driven Development:

"Test-driven development" refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).
It can be succinctly described by the following set of rules:

- write a "single" unit test describing an aspect of the program
- run the test, which should fail because the program lacks that feature
- write "just enough" code, the simplest possible, to make the test pass
- "refactor" the code until it conforms to the simplicity criteria
- repeat, "accumulating" unit tests over time

Expected Benefits

- many teams report significant reductions in defect rates, at the cost of a moderate increase in initial development effort
- the same teams tend to report that these overheads are more than offset by a reduction in effort in projects' final phases
- although empirical research has so far failed to confirm this, veteran practitioners report that TDD leads to improved design qualities in the code, and more generally a higher degree of "internal" or technical quality, for instance improving the metrics of cohesion and coupling.

There are two levels of TDD

1. **Acceptance TDD (ATDD):** With ATDD you write a single acceptance test. This test fulfills the requirement of the specification or satisfies the behaviour of the system. After that write just enough production/functionality code to fulfill that acceptance test. Acceptance test focuses on the overall behaviour of the system. ATDD also was known as **Behavioural Driven Development (BDD).**
2. **Developer TDD:** With Developer TDD you write single developer test i.e. unit test and then just enough production code to fulfill that test. The unit test focuses on every small functionality of the system. Developer TDD is simply called as **TDD.**

   The main goal of ATDD and TDD is to specify detailed, executable requirements for your solution on a just in time (JIT) basis. JIT means taking only those requirements in consideration that are needed in the system. So, increase efficiency.

Burndown Chart:

A burndown chart is a graphic representation of how quickly the team is working through a customer's user stories, an agile tool that is used to capture a description of a feature from an end-user perspective. The burndown chart shows the total effort against the amount of work for each iteration.

The quantity of work remaining is shown on a vertical axis, while the time that has passed since beginning the project is placed horizontally on the chart, which shows the past and the future. The burndown chart is displayed so everyone on the team can see it and is updated regularly to keep it accurate.

Variations of the Burndown Chart

There are two variants that exist for a burndown chart. A sprint burndown is for work remaining in the iteration. When illustrating the work remaining for the entire project, the chart is called a product burndown.

The obvious benefit of a burndown chart is that it provides an updated status report on the progress of the project. Having a visual representation of this most important data keeps everyone on the same page.

Also, by displaying it prominently for all to see, it keeps everyone involved and encourages the team to deal with issues before they become problems. Therefore, the bigger the chart, the better. It should be the focal point of the workspace, so that it cannot help but direct conversation towards the project and its progress.

The burndown chart is extremely helpful, too, because of its simplicity. It's a great way to see the velocity history of the project. Velocity is an agile term that means the total effort estimates associated with user stories that were completed during an iteration.

Automated Test:

Automation testing can solve your problem for fast testing methods if proper tools are used in an effective manner. There are various tools that are available for automation testing process such as: Selenium, TestNG, Appium, Cucumber, Test Studio, etc.

These tools require a test case to be developed according to the needs of application/software to be tested.

These test cases can then be run multiple times while performing continuous builds. This ensures that every step being taken is bug free or if a bug is introduced then it becomes easy to identify at what stage it has entered into our program.

Following are the scenarios in which we can apply Automation Testing in Agile Development,

- If a single test case is to be tested repeatedly.
- If your test cases are very tedious and time-consuming.
- If you have to run the test cases with different data and conditions several times.
- If you a similar test suite that needs to be executed for different user sets.
- If you have got a narrow release window, and saving time is your top priority.
- When test cases need to be executed with various browsers and environments

Following are the reasons why agile developers love automation testing,

- If a single test case is to be tested repeatedly.
- If your test cases are very tedious and time-consuming.
- If you have to run the test cases with different data and conditions several times
- If you a similar test suite that needs to be executed for different user sets.
- If you have got a narrow release window, and saving time is your top priority.
- When test cases need to be executed with various browsers and environments

Requirement Prioritization:

Software development project just like any other project has multiple requirements, budgetary constraints, and tight deadlines. Therefore, there is a need to do software requirements prioritization as it is just impossible to do everything at once. So, you need to make decisions on which set of requirements need to be implemented first and which ones can be delayed until a later release.

Normally, agile project managers want to develop software which is both high-quality and high-value, and the easiest way to develop high-value software is to implement the highest priority requirements first. This enables them to maximize stakeholder ROI.

As normally requirements change frequently, you need a streamlined, flexible approach to requirements change management, for example product backlog (Scrum). So, in agile development, software requirements prioritization is considered a vital part of the project. And for example, to prioritize user stories we could use top 5 prioritization criteria, such as the value users place on product vision, urgency, time constraints, technical complexity, and stakeholder preferences.

Also, very often projects need to be properly prioritized for both the main project objectives and the specific tasks that will achieve the objectives. So we deal with the prioritization on two levels: product level and task level. Especially taking into account that customers normally tend not to understand that they can't get all the features they want in release 1.0 of a new software product. Every single project should have priorities of the requested features, use cases, and functional requirements. Software requirements prioritization helps the project manager resolve conflicts, plan for staged deliveries, and make the necessary trade-off decisions.

One of the biggest differentiators between companies that thrive and those that fail is the ability to prioritize effectively, but software requirements prioritization is never easy. It often involves painful decisions and difficult trade-offs, and for early and expansion-stage companies, there is the added pressure of trying to accomplish a great deal with limited time and limited resources. In order to achieve growth, a team should be directed solely on the things that truly matter and are going to have maximum impact.

Software requirement prioritization is one of the biggest challenges a software team faces. Actually, 25% of tech leaders said prioritizing software development is their biggest challenge.

Software requirements prioritization doesn't have to be a constant source of pain. With the right method, it can empower your decision making and lead you to vastly improved results. Numerous methods and technologies on how to prioritize requirements have been developed as it is a very big issue for many teams and companies.

In-Pair Programming:

Pair programming consists of two programmers sharing a single workstation (one screen, keyboard and mouse among the pair). The programmer at the keyboard is usually called the "driver", the other, also actively involved in the programming task but focusing more on overall direction is the "navigator"; it is expected that the programmers swap roles every few minutes or so.

Also Known as More simply "pairing"; the phrases "paired programming" and "programming in pairs" are also used, less frequently.

Common Pitfalls

- Both programmers must be actively engaging with the task throughout a paired session, otherwise no benefit can be expected
- A simplistic but often raised objection is that pairing "doubles costs"; that is a misconception based on equating programming with typing – however, one should be aware that this is the worst-case outcome of poorly applied pairing
- At least the driver, and possibly both programmers, are expected to keep up a running commentary; pair programming is also "programming out loud" – if the driver is silent, the navigator should intervene.
- Pair programming cannot be fruitfully forced upon people, especially if relationship issues, including the most mundane (such as personal hygiene), are getting in the way; solve these first.

Expected Benefits

- Increased code quality: "programming out loud" leads to clearer articulation of the complexities and hidden details in coding tasks, reducing the risk of error or going down blind alleys.
- Better diffusion of knowledge among the team, in particular when a developer unfamiliar with a component is pairing with one who knows it much better.
- Better transfer of skills, as junior developers pick up micro-techniques or broader skills from more experienced team members.
- Large reduction in coordination efforts, since there are N/2 pairs to coordinate instead of N individual developers.
- Improved resiliency of a pair to interruptions, compared to an individual developer: when one member of the pair must attend to an external prompt, the other can remains focused on the task and can assist in regaining focus afterwards.

Release Planning:

Agile release planning is an approach to product management that takes into account the intangible and flexible nature of software development—as part of this approach, teams plan iterative sprints across incremental releases.

In other words, instead of trying to develop every proposed feature in one large, regimented project, agile planning breaks down the development process into stages called releases.

In this context, releases are periods of time set apart to work on a limited scope of the overall project.

An agile release plan maps out how and when features (or functionality) will be released and delivered to users.

By scheduling a project into agile releases, product managers can better manage project constraints and adapt to evolving needs or challenges that arise through the development stage while regularly producing product deliverables for the end user.

Elements of a product release map

Despite its name, agile release planning is highly structured. Each step is carefully outlined and measured to create high-level project calendars for teams to follow.

Release maps will vary slightly between organizations, but the general elements will include:

- The proposed release(s) for the project
- Plans for each release
- Subsequent iterations for the release(s)
- Plans for each iteration
- Feature development within an iteration
- Individual tasks necessary to deliver a feature

This level of planning, combined with an iterative schedule to account for the dynamic nature of software, is what makes agile product development so valuable.

The iterative release schedule gives teams the space to make course corrections without derailing the entire project, while the detailed roadmap and focus on the planning stage ensure everyone is on the same page.

When all conditions are met, the team can confirm a release is completed. The “Definition of Done” usually means the team has completed every task outlined under a user story and documented the work for the product owner to review.

Refactoring:

Refactoring consists of improving the internal structure of an existing program's source code, while preserving its external behavior.

The noun "refactoring" refers to one particular behavior-preserving transformation, such as "Extract Method" or "Introduce Parameter."

Common Pitfalls

Refactoring does "not" mean:

rewriting code

fixing bugs

improve observable aspects of software such as its interface
Refactoring in the absence of safeguards against introducing defects (i.e. violating the "behavior preserving" condition) is risky. Safeguards include aids to regression testing including automated unit tests or automated acceptance tests, and aids to formal reasoning such as type systems.

Expected Benefits

The following are claimed benefits of refactoring:
refactoring improves objective attributes of code (length, duplication, coupling and cohesion, cyclomatic complexity) that correlate with ease of maintenance
refactoring helps code understanding
refactoring encourages each developer to think about and understand design decisions, in particular in the context of collective ownership / collective code ownership
refactoring favors the emergence of reusable design elements (such as design patterns) and code modules.

REFACTORING AUTOMATION IN IDES

Refactoring is much, much easier to do automatically than it is to do by hand. Fortunately, more and more
Integrated Development Environments (IDEs) are building in automated refactoring support. For example,
one popular IDE for Java is eclipse, which includes more auto-refactoring all the time.
To refactor code in eclipse or IDEA, you select the code you want to refactor, pull down the specific refactoring
you need from a menu, and the IDE does the rest of the hard work. You are prompted appropriately by dialog boxes
for new names for things that need naming, and for similar input. You can then immediately rerun your tests to make sure that the change didn't break anything. If anything was broken, you can easily undo the refactoring and investigate.

Small Release Cycle:

Small release cycles generally are releasing miniature versions of your product to the public over short time intervals. Afterward, analyse the information received from your customers' reactions.

Small release cycles are essential for your XP as they show you the accuracy of your team towards the project. Because in the time of your planning stage, you only had estimated views/ ideas of customer reaction based on the computer-throughput and how user-friendly your software will be. But now with small releases, you get to know the actual reaction/ feedback from your customers.

The small release cycle gives you essential, timely feedback from customers also. This feedback helps you know which areas your software needs to change before its main release. Your publications have to be iterative meaning for each problem the customers received in one version should be fixed in the next small release.

Another vital function of small releases is that they allow you to have your customers involved in the XP process. Because involving customers in the XP process gives you their feedback which will determine the future action of the project. Customers will also see your determination to provide them with excellence. Making them trust you even more.

Make sure you are well prepared for the small releases before you begin the small releasing stage, you need to create a releasing plan. In a releasing plan, you will discuss how each iteration will occur including the time intervals between them. Making you better prepared for the small releases.

Use feedback from your previous release to improve the next. The feedback you will receive from your customers should be used to develop further your subsequent small releases. Henceforth increase the demand for your final product, as it meets your customer’s needs.

Try something new in each release. You also need to be adding new experimental features to your versions. Helping you get valuable information on how customers would react to the original idea.

Coding Standard:

Coding standards are a formalised set of rules and practices that developers can adhere to when writing code which ensures that the code is easily readable, maintainable and extensible and reduces the risk of introducing bugs.

As you may imagine, there are many coding standards and some of them conflict with each other. In some ways, if a coding standard adds benefit to the code base it doesn't really matter what it is, as long as the development team are consistent with its application.

Here are some commonly used coding standards that I have seen across the financial industry and in other industry sectors. I list these here as examples of standards. To learn more you should read the further reading books on this page. This is a must for developers.

Consistent case of methods and properties

This is purely a stylistic thing and it doesn't matter technically which case you use. Java and .Net both recommend different cases. My approach is to the use the C# recommended case for C# and Java for Java.

Tools like ReSharper (see below) can auto correct case sensitivity as the developer is coding.

Descriptive method and class names

Part of making code readable is to use good descriptive names for a class and methods. For example the method which says DoSomething() is totally useless as no one can tell what it does without going into the code.

A method which says CreateConnectionToDatabase() and passes back a connection object is obvious in its intent.

Keeping methods to a reasonable length

If methods are too long, they become unreadable and difficult to maintain. If a method is too long, you can use the extract method pattern to encapsulate some of the functionality into another method.

See Martin Fowler's book on improving existing code.

Ensuring code standards are adhered to

It can be difficult to remember and annoying to go back and correct standards in the code you are writing. It is therefore essential to use some automated tool both at design time and checking in time that checks for certain elements of quality in the code.

Collective Ownership:

Teams typically adopt conventions governing who is allowed to modify some source code that was originally written by another, often referred to as “ownership”. These conventions can be written and explicit, merely oral, or entirely implicit. Many different modes exist; commonly only one developer “owns” each code file. Collective code ownership, as the name suggests, is the explicit convention that “every” team member is not only allowed, but in fact has a positive duty, to make changes to “any” code file as necessary: either to complete a development task, to repair a defect, or even to improve the code’s overall structure.

Expected Benefits

A collective code ownership policy:

- Reduces the risk that the absence (or unavailability) of any one developer will stall or slow work
- Increases the chance that the overall design results from sound technical decisions, rather than from social structure, as in “Conway’s Law”
- Is a favorable factor in the diffusion of technical knowledge
- Encourages each developer to feel responsible for the quality of the whole

Common Pitfalls

Tacit or implicit rules may exist that contradict a collective ownership policy adopted by the team. For example, a developer becomes ill-tempered whenever files from a particular component are modified, so that the team starts to treat them as “de facto” under his or her exclusive ownership. Be sure to check that the policy is consistent with actual behavior.

Potential Costs

While the idea of collective ownership is aligned with other principles of shared responsibility in Agile (such as shared responsibility, between customer and development team, for project outcomes), it has its detractors. Arguments against are also plausible and bear being careful about:

- In the limit, having everyone is responsible for quality can be a situation indistinguishable from having no one responsible for quality
- The lack of social enforcement of boundaries around code components can lead (by “Conway’s Law”) to a lack of well-defined interfaces, however well-defined interfaces are a key to effective design

Behaviour Driven Development:

Behavior Driven Development (BDD) is a synthesis and refinement of practices stemming from Test Driven Development (TDD) and Acceptance Test Driven Development (ATDD). BDD augments TDD and ATDD with the following tactics:

- Apply the "Five Why's" principle to each proposed user story, so that its purpose is clearly related to business outcomes
- thinking "from the outside in", in other words implement only those behaviors which contribute most directly to these business outcomes, so as to minimize waste
- describe behaviors in a single notation which is directly accessible to domain experts, testers, and developers, so as to improve communication
- apply these techniques all the way down to the lowest levels of abstraction of the software, paying particular attention to the distribution of behavior, so that evolution remains cheap

Also Known As BDD is also referred to as Specification by Example.

Expected Benefits

Teams already using TDD or ATDD may want to consider BDD for several reasons:

- BDD offers more precise guidance on organizing the conversation between developers, testers and domain experts
- originating in the BDD approach, in particular the given-when-then canvas, are closer to everyday language and have a shallower learning curve compared to those of tools such as Fit/FitNesse
- tools targeting a BDD approach generally afford the automatic generation of technical and end user documentation from BDD "specifications"

Common Pitfalls

Although Dan North, who first formulated the BDD approach, claims that it was designed to address recurring issues in the teaching of TDD, it is clear that BDD requires familiarity with a greater range of concepts than TDD does, and it seems difficult to recommend a novice programmer should first learn BDD without prior exposure to TDD concepts

The use of BDD requires no particular tools or programming languages, and is primarily a conceptual approach; to make it a purely technical practice or one that hinges on specific tooling would be to miss the poaint altogether.

Document Late:

As a part of the agile strategy, you document as late as possible, only before you require them. This practice is known as "document late" that means the best time to write system overviews will be towards the end of the development of a release.
This way you document what you have actually built.
The majority of the user and support documentation is pushed to the end of the lifecycle to ensure high quality. You still take notes throughout the development for capturing critical information that will help to formulate the documents at the end.

In agile documentation approach, one core principle followed is that the comprehensive documentation doesn't give surety of the project success. Rather it could lead to failure.
So the Agile Modeling (AM) practices are followed to use the simplest tools, to create simple content and to depict models simply.
This means rather than writing a document with 50 pages, a 5-page document with bullet points can be good enough to provide a useful context.
The higher the number of pages in the document, the higher would be the chances of error.
A short and concise document is easier to maintain and lesser prone to errors. In the agile documents, the high-level overview may not have detailed information, but it does provide a map to dive in the source code.
The documentation should be just good enough to serve the purpose of dealing with the situation at hand.
This requires building larger documents from smaller ones. You can create Wikis sort of documentation to create different single pages for single topics.
For example, you can have a single page describing the user interface architecture. There can be a single page to describe the user interface flow diagram. A table of contents pages for the system documentation and system guides. This will also ensure that there is no overlap of information.