AGILE SOFTWARE DEVELOPMENT

Digital Notes

UNIT-1

Introduction

Topic: Introduction

What Is Agile Methodology?

Agile methodology is a project management approach that prioritizes cross-functional collaboration and continuous improvement. It divides projects into smaller phases and guides teams through cycles of planning, execution, and evaluation.

Agile methodologies overview:

The Agile Manifesto for Software Development put forth a ground-breaking mindset on delivering value and collaborating with customers when it was created in 2001.

Agile's four main values are:

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

What are the benefits of using Agile methodology?

Agile is one of the most popular approaches to project management because it is flexible, it is adaptable to changes and it encourages customer feedback.

Many teams embrace the Agile approach for the following reasons:

- **Rapid progress:** By effectively reducing the time it takes to complete various stages of a project, teams can elicit feedback in real time and produce working prototypes or demos throughout the process
- **Customer and stakeholder alignment:** Through focusing on customer concerns and stakeholder feedback, the Agile team is well positioned to produce results that satisfy the right people
- **Continuous improvement:** As an iterative approach, Agile project management allows teams to chip away at tasks until they reach the best end result

Types of Agile methodologies

Agile project management is not a singular framework but an umbrella term that includes a wide range of methodologies, including Scrum, Kanban, Extreme Programming (XP), and the Adaptive Project Framework (APF).

Scrum: It is ideal for projects with rapidly changing requirements, using short sprints. **Kanban:** It visualizes project progress and is great for tasks requiring steady output. **Lean:** It streamlines processes, eliminating waste for customer value. **Extreme Programming (XP):** It enhances software quality and responsiveness to customer satisfaction.

Adaptive Project Framework (APF): Works well for projects with unclear details, as it adapts to constantly evolving client needs.

Agile Manifesto Values

Agile enables software development teams to stay adaptable.

With an iterative and adaptive approach, the aim is to produce the highest-quality software product that puts the customer at the heart of the process. By prioritizing flexibility, Agile teams can quickly react to changes, deliver products faster, and thrive in a collaborative environment.

By building Agile teams with the right qualities — such as self-organization and effective collaboration — you can accelerate the software development process while leaving space for vital customer feedback.

One of the most compelling reasons to adopt the Agile approach in software development is the dynamic workflows and work systems that contribute to a better end product. By listening to customer feedback and carrying out several iterations and rounds of software testing, you can iron out any kinks along the way and build the best possible software.

The Agile software development life cycle helps you break down each project you take on into six simple stages:

1. **Concept:** Define the project scope and priorities
2. **Inception:** Build the Agile team according to project requirements
3. **Iteration:** Create code factoring in customer feedback
4. **Release:** Test the code and troubleshoot any issues
5. **Maintenance:** Provide ongoing tech support to ensure the product remains serviceable
6. **Retirement:** The end of the product lifespan, which often coincides with the beginning of a new one.

FOUR Agile Manifesto Values

According to the Agile Manifesto, there are four fundamental principles of Agile:

1. Individuals and interactions over processes and tools

Success in a project depends on having the appropriate people on your team. Even the best tools are rendered toothless in the wrong hands. The manner in which they engage with each other is even more critical. The engagement among team members encourages them to work together and resolve any possible issues.

2. Working software over comprehensive documentation

It used to take software engineers a long time to provide thorough documentation. They did that prior to developing a single line of code. While it's not always a bad thing, you should eventually concentrate on giving your clients software that works. According to the Agile Manifesto, one of the top priorities is providing the software to your consumers. After that, you can collect suggestions to enhance upcoming releases.

3. Customer collaboration over contract negotiation

Contracts used to be considered supreme. Contracts would be drafted with your clients, who would then specify the final product. Due to this, there often was a discrepancy between what was declared in the contract, what the product actually achieved, and what the customer truly required. The Agile Manifesto states that continuous development should actually be the main priority. To ensure your product works for your clients, you must create a feedback loop with them.

4. Responding to change over following a plan

We do not exist in a static world, which makes static blueprints redundant. Priorities are rapidly changing, as are needs and requirements. That static approach will soon become defunct. The Agile Manifesto states that a software team must be capable of pivoting and shifting course whenever required, with a flexible roadmap that considers that. Agile teams can adjust to the transformations that a dynamic strategy may undergo from quarter to quarter or even from month to month.

TWELVE Agile Manifesto Principles

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

You may concentrate on the project's primary goal—delivering what the customer wants, not what you planned—by cutting down on time it takes between the project documentation, reporting to your client, and then obtaining feedback.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

It is time consuming to handle large and complicated work while managing project activities. Therefore, a better strategy is to break the task into manageable, sizeable chunks. In addition, it would be simpler for the team members to see possible bottlenecks and deal with delays if the clients were always kept informed.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for a shorter timescale.

According to the Agile methodology, working software is frequently delivered in a shorter amount of time. Team members must consistently raise their performance standards as a result of this iterative process.

4. Business people and developers must work together daily throughout the project.

In order to ensure that the business and development sides of the project can communicate effectively and, more importantly, collaborate, a bridge between them must be built. To facilitate an intellectual exchange that both parties can agree on, make use of the same tools you would have used in managing remote teams.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The project manager must establish a supportive and stimulating environment where team members are free to express their ideas and make recommendations for enhancing the output of the group. This results in a massive improvement in their general performance, eventually aiding the project.

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Efficient communication among the parties concerned is stressed strongly in the Agile manifesto. Thanks to improvements in communication technologies, it's now simpler. Instead of having a quick conference in the office, all participants can now meet via video conferencing.

7. Working software is the primary measure of progress.

Delivering a functional product that pleases the consumer is the single determinant that can guarantee success. Before Agile, numerous success metrics decreased the quality of the finished product.

8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Burnout will occur if you work on a project for a long time. It's inevitable. Avoid placing too much of a workload on your employees. The value of your project will be affected. So, assemble the best team for the job that will work hard but refrain from overworking themselves and endangering the project's quality.

9. Continuous attention to technical excellence and good design enhances agility.

Any Agile team's main goal should be to provide value to the client. Therefore, a multi-skilled team that can manage all the project's technical components and offers the chance for continual improvement is crucial.

10. Simplicity — the art of maximizing the amount of work not done — is essential.

You should avoid adding extraneous complexity to a project if you want to complete it swiftly. You can accomplish this in various ways, including by using agile tools, which eliminate busywork and offer you more significant influence over all project-related decisions.

11. The best architectures, requirements, and designs emerge from self-organizing teams.

Simply said, a self-organized workforce with decision-making autonomy would function better since each team member would be responsible for meeting client expectations rather than a lone project manager.

12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile techniques are constructed on the notion of iteration, where teams consistently enhance their game by learning from their previous wrongdoings. Project managers should inspire team meetings where everyone evaluates their work and discusses how to develop their management and technical skills.

Agile SDLC Models/Methods

The meaning of Agile is swift or versatile."**Agile process model**" refers to a software development approach based on iterative development. Agile methods break tasks into smaller iterations, or parts do not directly involve long term planning. The project scope and requirements are laid down at the beginning of the development process. Plans regarding the number of iterations, the duration and the scope of each iteration are clearly defined in advance.

Each iteration is considered as a short time "frame" in the Agile process model, which typically lasts from one to four weeks. The division of the entire project into smaller parts helps to minimize the project risk and to reduce the overall project delivery time requirements. Each iteration involves a team working through a full software development life cycle including planning, requirements analysis, design, coding, and testing before a working product is demonstrated to the client.

Phases of Agile Model:

Following are the phases in the Agile model are as follows:

1. Requirements gathering
2. Design the requirements
3. Construction/ iteration
4. Testing/ Quality assurance
5. Deployment
6. Feedback

1. Requirements gathering: In this phase, you must define the requirements. You should explain business opportunities and plan the time and effort needed to build the project. Based on this information, you can evaluate technical and economic feasibility.

2. Design the requirements: When you have identified the project, work with stakeholders to define requirements. You can use the user flow diagram or the high-level UML diagram to show the work of new features and show how it will apply to your existing system.

3. Construction/ iteration: When the team defines the requirements, the work begins. Designers and developers start working on their project, which aims to deploy a working product. The product will undergo various stages of improvement, so it includes simple, minimal functionality.

4. Testing: In this phase, the Quality Assurance team examines the product's performance and looks for the bug.

5. Deployment: In this phase, the team issues a product for the user's work environment.

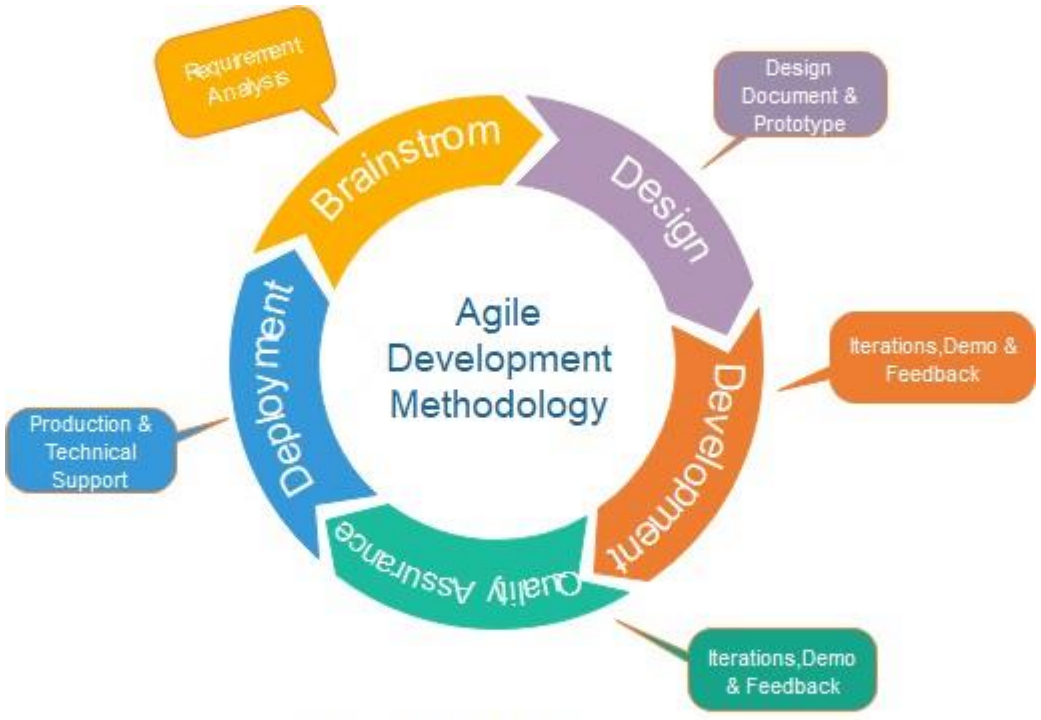

Requirement Analysis
Brainstrom
Design Document & Prototype
Design
Agile Development Methodology
Iterations,Demo & Feedback
Development
Production & Technical Support
Deployment
Quality Assurance
Iterations,Demo & Feedback

Fig. Agile Model

6. **Feedback:** After releasing the product, the last step is feedback. In this, the team receives feedback about the product and works through the feedback.

Agile Methods

There are five core methods being worked on and with at the present time.

- eXtreme Programming (or XP),
- DSDM – Dynamic system development method
- SCRUM
- FDD - Feature-Driven Development
- Agile Modelling

- Extreme Programming (XP): Extreme Programming uses specific practices like pair programming, continuous integration, and test-driven development to achieve these goals. Extreme programming is ideal for projects that have high levels of uncertainty and require frequent changes, as it allows for quick adaptation to new requirements and feedback.

- Scrum: Scrum methodology serves as a framework for tackling complex projects and ensuring their successful completion. It is led by a Scrum Master, who oversees the process, and a Product Owner, who establishes the priorities. The Development Team, accountable for delivering the software, is another key player.

- Feature-driven development (FDD): FDD approach is implemented by utilizing a series of techniques, like creating feature lists, conducting model evaluations, and implementing a design-by-feature method, to meet its goal. This methodology is particularly effective in ensuring that the end product is delivered on time and that it aligns with the requirements of the customer.

- Dynamic Systems Development Method (DSDM): DSDSM methodology is tailored for projects with moderate to high uncertainty where requirements are prone to change frequently. Its clear-cut roles and responsibilities focus on delivering working software in short time frames. Governance practices set it apart and make it an effective approach for teams and projects.

DSDM- Dynamic systems development method

- DSDM is An iterative code method within which every iteration follows the 80% rule that simply enough work is needed for every increment to facilitate movement to the following increment. The remaining detail is often completed later once a lot of business necessities are noted or changes are requested and accommodated.

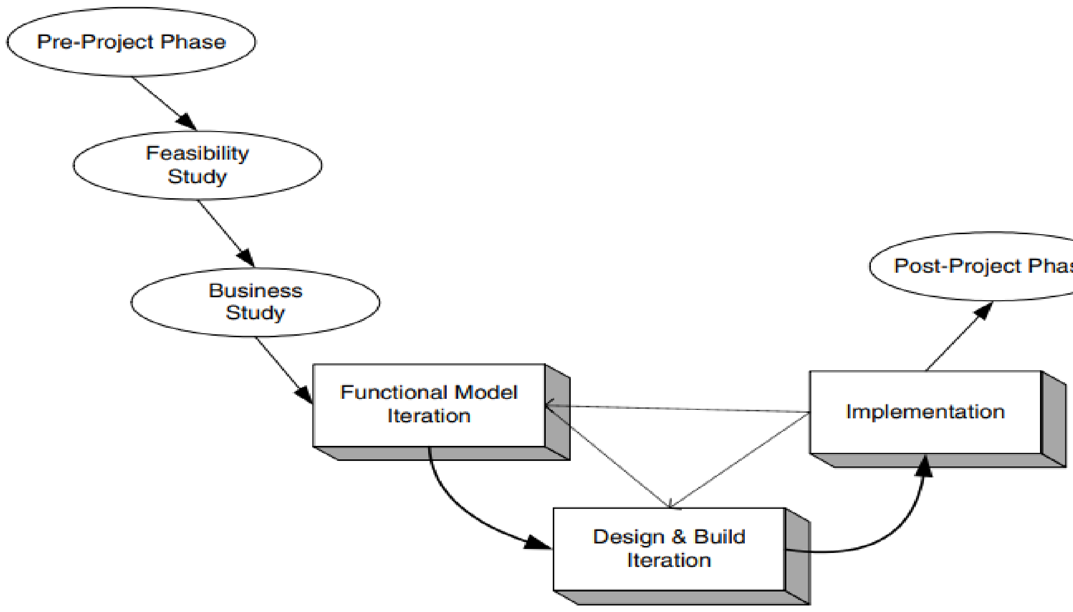
DSDM is based on nine overriding principles, these are:

1. Active user involvement is imperative.
2. The team must be empowered to make decisions.
3. The focus is on frequent delivery of products.
4. Fitness for business purpose is the essential criterion for acceptance of deliverables.
5. Iterative and incremental development is necessary to converge on an accurate business solution.
6. All changes during development are reversible.
7. Requirements are base lined at a high level.
8. Testing is integrated throughout the life cycle.
9. Collaboration and cooperation between all stakeholders is essential.

DSDM Life cycle

The actual DSDM lifecycle is broken down into seven different phases, these are:

- Pre-Project Phase
- Feasibility Study
- Business Case Study
- the Functional Model Iteration (FMI)
- the Design and Build Iteration (DBI)
- the Implementation Phase
- the Post-Project Phase.



Feasibility Study:

It establishes the essential **business necessities** and constraints related to the application to be designed then assesses whether or not the application could be a viable candidate for the DSDM method.

Business Study:

It establishes the use and knowledge necessities that may permit the application to supply business value; additionally, it is the essential application design and identifies the maintainability necessities for the application .

Functional Model Iteration:

It produces a **collection of progressive prototypes** that demonstrate practicality for the client. (Note: All DSDM prototypes are supposed to evolve into the deliverable application.) The intent throughout this unvarying cycle is to collect **further necessities** by eliciting feedback from users as they exercise the paradigm.

Design and Build Iteration:

It revisits prototypes designed throughout useful model iteration to make sure that everyone has been designed during a manner that may alter it to supply operational business price for finish users. In some

cases, useful model iteration and style and build iteration occur at the same time.

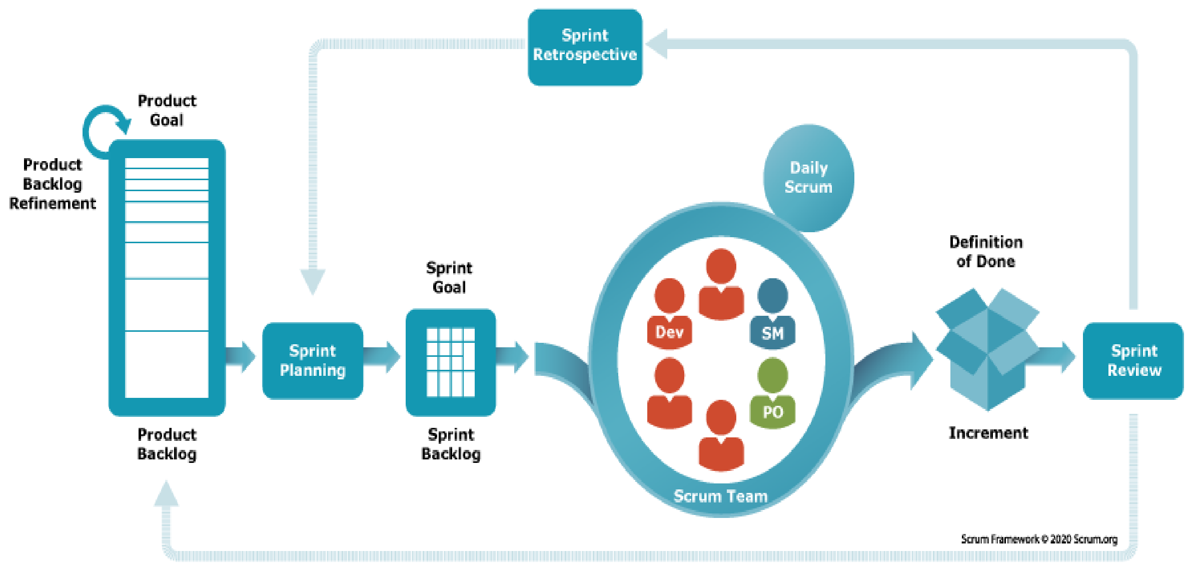
Implementation:

It places the newest code increment (an "operationalized" prototype) into the operational surroundings. It ought to be noted that:

- the increment might not 100% complete or,
- changes are also requested because the increment is placed into place. In either case, DSDM development work continues by returning to the useful model iteration activity.

SCRUM

- Scrum aims to manage and control the production of software using iterative, incremental and lightweight processes
- Scrum is an approach to managing complicated projects that may have to adapt to changes in scope or requirements. By emphasizing productivity, focus and collaboration



- **The Process**
- When a customer (internal or external) comes to the team with a certain need, the final product is broken up into individual chunks. (Traditionally this has been a software need, but the process also works for any project that is comprised of multiple stages and pieces, such as a marketing launch.) The pieces are prioritized and tackled in a series of short bursts called sprints. Teams can determine their own sprint length,

provided it's less than 4 weeks (one to two weeks is common). At the end of each sprint, the team delivers a product increment — essentially, a version of the product that could be shipped if necessary. Transparency is a key principle in Scrum, so teams and stakeholders review the results of each sprint together. This ensures everyone's on the same page about priorities and deliverables, and any adjustments can be made right away.

- Teams promote internal transparency through daily standups. During these brief, 15-minute meetings, everyone reports what they accomplished yesterday, what they plan to work on that day, and any current "impediments" (factors that are keeping them from working more efficiently). This visibility helps uncover problems and bring them to the forefront quickly, so the team can tackle and overcome them together.

Who's Who: Scrum Roles

- There are three main roles in Scrum: the product owner, the scrum master, and the development team.
- **Product Owner:** Product owners represent the customer's interests. They decide what the team will work on next, so the team's efforts stay focused on high-priority tasks that create the most value. The Kanban product owner must always be available to provide input or guidance to the development team, although it's important to note that product owners are not managers — scrum teams self-organize.
- **Scrum Master:** The Scrum master's #1 goal is to help the development team be self-sufficient. Scrum masters intercept and remove barriers to team progress, and act as a buffer between the team and any outside forces that might interfere with productivity. S/he leads daily standup meetings, so while the product owner is responsible for what the team will produce, the scrum master oversees the how.
- **Development Team:** Development teams are made up of cross-functional team members, so the group has all the necessary skills to deliver the final product. The team focuses on only one project at a time; members don't multitask or split their efforts between multiple projects. Once the product owner makes an ordered list of what needs to be done, the development team decides how much they can complete in a single sprint and plan accordingly.

Core Values

As an Agile framework, Scrum shares the values of the Agile Manifesto. But it also creates its own guidelines. These are the five golden rules in Scrum:

- **Openness:** Scrum sees collaboration as the most effective way to create the best possible product. So teamwork and transparency are essential. Rather than anxiously downplaying problems, Scrum team members are open about their progress and any roadblocks they encounter.
- **Focus:** With Scrum, multitasking is out. Since productivity is key, splitting the team's attention across multiple projects, or redirecting their efforts mid-sprint by shifting priorities, is avoided at all costs. Instead, teams concentrate on the task at hand for the highest velocity and best quality product.
- **Courage:** Teams must have the tenacity to commit to an ambitious (but attainable) amount of work for each sprint. Scrum masters must also be able to stand up to stakeholders if necessary, and the product owner must guide the development team with authority.
- **Commitment:** Each sprint is itself commitment: teams must agree on what they're going to accomplish and stick to it. This value is reflected in each team's unique "Definition of Done," a list of criteria to determine

whether a feature or deliverable is truly finished — that it's not only fully functional, but meets the team's standards for quality.

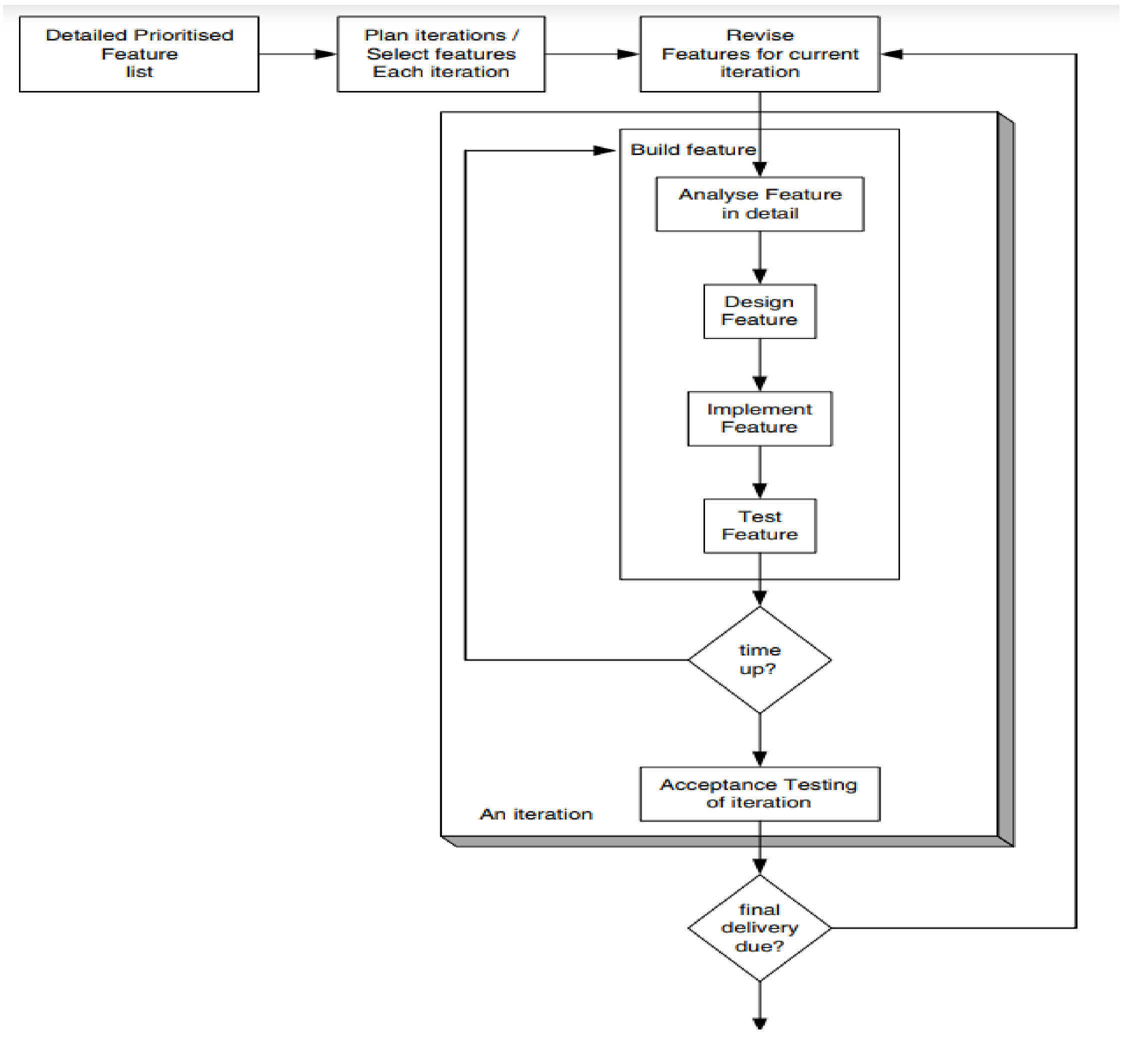
- **Respect:** In the service of true collaboration, roles and responsibilities are transparent. Each member of the team is respected equally, regardless of job description, seniority, or status. The development team must honor the product owner's authority in deciding what the team works on, and the product owner needs to respect the team's need follow whatever work process is best for them.

Feature Driven Development

- Feature-Driven Development (FDD) is a client-centric, architecture-centric, and pragmatic software process.
- The big advantage of using a feature-centric approach is managing an agile project, for handling the uncertainties that an agile approach introduces.
- A feature is a schedulable requirement associated with the activity used to realise it.
    - These requirements may be user related requirements (i.e., be able to open a bank account)
        - application behaviour requirements (make a backup every 10 min)
        - internal requirements (provide the ability to turn on debugging for system support).

- Thus, a feature mixes units of requirements with units of management. In addition, features should have the following attributes:
- Features should be small and "useful in the eyes of system stakeholders."
    - Features can be grouped into business-related groupings (called variously feature sets or work packages).
    - Features focus developers on producing elements within the system that are of tangible benefits to system stakeholders.
    - Features are prioritised.
    - Features are schedulable.
    - Features have an associated (estimated) cost. Can be grouped togetherinto shortiterations (possibly as short as twoweeks).

- Feature-centric management of agile projects offers present a five-step process that outlines how a feature-centric approach works, these five processes are:
    - Process 1: Develop an overall model of the domain and create an initial feature list.
    - Process 2: Build a detailed, prioritised feature list.
    - Process 3: Plan by feature.
    - Process 4: Design by feature.
    - Process 5: Build by feature


Detailed Prioritised Feature list
Plan iterations / Select features Each iteration
Revise Features for current iteration
Build feature
Analyse Feature in detail
Design Feature
Implement Feature
Test Feature
time up?
Acceptance Testing of iteration
An iteration
final delivery due?

- We can elaborate on that by considering an iterative feature based lifecycle. This is presented in Figure. In the diagram above, the flow illustrates the following steps:

  1. First identify a prioritized feature list. This can be done by considering the systems' requirements. These can be produced in whatever manner is appropriate. For example, through use cases, a formal requirements specification or user stories. What is required is that they are elaborated sufficiently to allow prioritization and an initial cost estimate to be associated with them.
  2. This initial feature list is then used to create a plan of the iterations to be undertaken. Each iteration should have one or more features associated with it and should not be too long. Each iteration should have a timebox associated with it that specifies when it starts and when it finishes.
  3. Before each iteration starts, the iteration should be planned in detail. This involves determining which features are still relevant for that iteration, any revised priorities and an ordering to the features.
  4. Once a iteration starts, each iteration is addressed in turn based on their priorities. At any one time, one or more features will be worked on depending on the size of the feature and the resources available (note that no assumption is made here about how many people will be needed to implement a feature, there could be one developer per feature, two per feature or variable depending upon the step within the feature that is currently being addressed).
  5. The iteration stops when the timebox finishes. At the end of the iteration, the current version of the software is tested to ensure it does what it should

6. If this is the final iteration, then the final version of the system is delivered (if it is not the final iteration, then the current system may still be delivered to end users for early and frequent feedback). This is possible as each feature should be useful in the eyes of the various project stakeholders in their own right.

Agile Modelling

Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation.

- Some important concepts:

  - **Model** - A model is an abstraction that expresses important aspects of a thing or concepts. Models may be visual (diagrams), non-visual (text descriptions), or executable (working code or equivalent). Models are sometimes called maps or roadmaps within the agile community.
  - **Agile model** - Agile models can be something as simple as stickies on a wall, sketches on a whiteboard, diagrams captured digitally via a drawing tool, or detailed models captured using a model-based software engineering (MBSE) tool.
  - **Modeling** - Modeling is the act of creating a model. Modeling is sometimes called mapping.
  - **Agile modeling** - Agile modeling is modeling performed in a collaborative and evolutionary manner.
  - **Document** - A document is a persistent representation of a thing or concept. A document is a model, but not all models are documents (most models are not persistent).
  - **Agile document** - Agile documents can be something as simple as point-form notes, detailed text, executable tests, or one or more agile models.

- Agile models are good enough when they exhibit the following traits:
  1. Agile models fulfill their purpose.
  2. Agile models are understandable
  3. Agile models are sufficiently accurate
  4. Agile models are sufficiently consistent
  5. Agile models are sufficiently detailed
  6. Agile models provide positive value
  7. Agile models are as simple as possible.

Accuracy and Consistency

Another important aspect of Agile Modelling is that the models need to be only sufficiently accurate and consistent. That is, you do not need to worry about crossing every “t” and dotting every “i.” Allied to this idea is that the model (or models) should be comprehensible to their intended audience (but by implication not necessarily comprehensible to everyone or at least sufficient for everyone) and sufficiently detailed for that audience. Finally, the models should be as simple as possible without losing their message. That is, unnecessary details need not be included. For example, if I am using a street map to try to get from one location to another and I find that the map and the real world differ slightly (because of changes since the map was printed); I do not necessarily throw the map away. I may instead annotate the map at that point. Or use the map to find another route. Equally, my map probably does not show every house on my street, rather it gives an impression of a

number of houses. That is enough for me to know that this is a built up area and that if I go to this street I will find houses on it. In many cases that is sufficient for my needs. However, such a map would probably not be sufficient, accurate enough nor detailed enough for a utility company wishing to provide fibre optic cables to all the houses in my street. They would need a different type of map. Indeed such a map may well not only provide a great deal more detail of the actual houses and street, it may also show details that in my case I do not wish to know (such as what exactly is underneath the road outside my house).

Tool Misconceptions

At this point, it is worthwhile considering some misconceptions and myths relating to the use of tools, modelling and UML.

1. UML requires CASE tools.

This is certainly not true – I can draw a UML diagram freehand on paper, use a simple drawing package such as Paint or indeed with a tool such as Together. It may well be true that to strictly adhere to the UML notation it is easier to use something that knows about UML (and Visio might be such a tool). It may also be true that if you want to generate code from the UML diagrams, using a CASE tool such as Rose or Together makes life easier. But it is not a pre-requisite.

2. Modelling requires the use of CASE tools.

An extension of the last point is that if you are going to create models as part of your design you need a CASE tool that can manage, manipulate, cross reference, etc., your models. While these features may well be useful, they are not necessary. I can (and indeed have) used simpler tools such as Visio to perform all the modelling necessary for the design of a system. Obviously, the larger the system and the larger the amount of modelling performed, the better a CASE tool may be.

3. Agile modelers don’t use CASE tools.

This is a common misconception by those starting with Agile Modelling. This is partly due to the emphasis of Agile Modelling on using the simplest appropriate tool and if that tool is a white board or a piece of paper, use it. The key word here is appropriate. If I need to work something through with one of my colleagues, we might well use a white board or a piece of paper and not worry too much about the accuracy of the UML notation being used. If however, I am trying to describe a complex structure that will need to be referenced by a variety of developers, possibly in multiple locations, then a CASE tool might well be the most appropriate.

4. UML is all you need!

Some take the view that UML is all you need in terms of notation – if you can’t do it in UML (aka the design tool you are using), then it is either not relevant to an object-oriented system or not important enough to document. This is not true. There are many aspects of a software project that you may wish to document, but do not fit within the remit of a UML diagram, for example, GUI storyboarding, data modelling, etc.

5. The CASE tool is master.

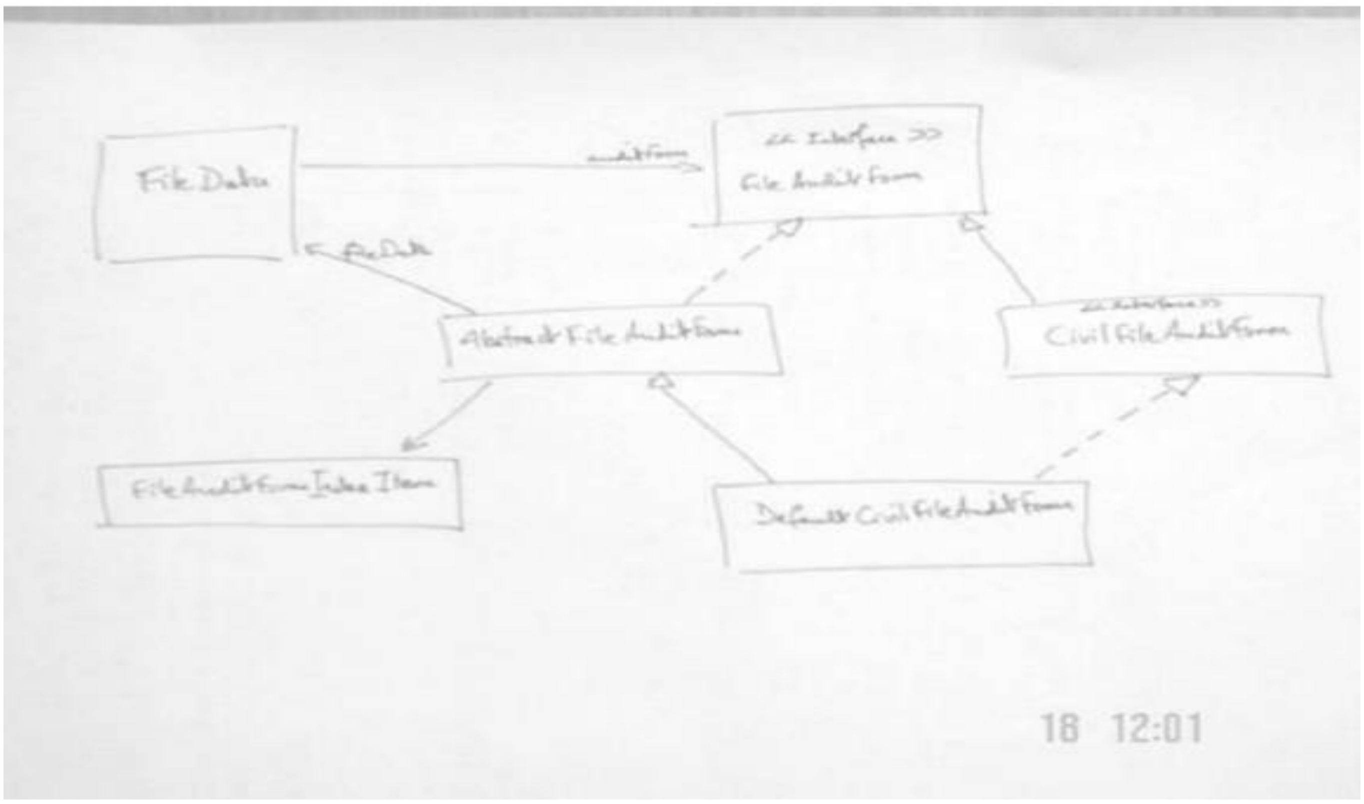
This is more a perception than a misconception. It is a perception because users often feel that they are battling with the CASE tool and that they have to work in the way prescribed by the CASE tool. Certainly, I know that some of the colleagues I have worked with over the years have an almost irrational hatred against one well known CASE tool because of the way it forces them to work. Some of this can be overcome with training and some by choosing a suitable tool. The important thing is that the CASE tool should not be the master but the servant. It should help you with your work and not hinder it. Thus, finding an appropriate tool (or tools) is important. For example, in one case Together proved to be particularly well suited to the organizations way of working and to the developers experience and background.

Updating Agile Models

Finally, we come to the question of when you should update the models you have created. The general gist of when this should be is “when it hurts not to,” that is when you actually need to. What does this mean? Take, for example, the case on a recent project I was involved with. On that project, we created models of what we would implement; these helped us to understand what was required and the structure that would be used. However, for various memory and performance reasons, it was found that the implementations had to change to try and reuse as many Java Swing components as possible and to cache any data not actually being displayed. This necessitated quite a few changes to the behavioral aspects of the system and some changes to the structure of the system. However, at that point we did not go back and rework the models as they had served their purpose – they had helped us to understand the requirements and how the system should be structured (in addition, they still gave a flavor of the system). If those models were never needed again, reworking them would have been a waste. Some 6 months later, this aspect of the system was to be updated. It was at this point that a software engineer updated the models by reverse engineering the classes. He also worked through the code to update the behavioral aspects of the model. This had two effects: first, the models were updated in a timely fashion, and second, the software engineer involved gained a detailed understanding of this part of the system before he commenced further design. In addition, some models may never be required again and can be thrown away. For example, the hand-drawn models used to allow myself and a colleague to understand how two areas of the system will interact does not necessarily need to be saved for posterity. The existing models may be more than suffice. Therefore, the hand-drawn model (created to help our understanding) can be thrown away. This means that it does not need to be maintained, fully documented, recorded, reviewed, etc. This can be taken further for more formal models, created using tools such as Rose. However, I tend towards caution and tend to feel that if the model was established enough to have been created in a CASE tool, then it should at least be stored in a version control tool (such as CVS or SCCS) so that it can be retrieved if necessary at a later date (otherwise the effort used to create the model may be lost altogether!).

Sort Of Models

Another key idea in Agile Modelling is that content is more important than presentation. For example, the hand-drawn diagram in Figure may not be the prettiest UML diagram you have ever seen, the lines are not straight, the classes not complete, etc.; however, it is the message it conveys that is important. If this diagram is sufficient and effective in conveying that message, then that is enough! One more point to note about Agile Modelling is that whatever a diagram or diagrams best suit the information you need to present, discussion or understanding should be done. This does not mean that you must produce many different diagrams. Merely that if what you need to describe is best presented as a class diagram, then use it. However, if it is better to use a Sequence diagram use that. In addition, if something is proving difficult to understand or work through in one type of diagram then move to another – it may be that this will help. Also do not feel afraid of mixing diagrams, placing some data modelling on a class diagram, which may well help describe your problem, etc. The key here is to use whatever tools and techniques are available to you to win the modelling battle. In general, it is likely that you will need to use multiple modelling techniques to understand a problem. Personally, I rarely find that a class diagram is suitable in isolation. In general, I will create a class diagram in parallel with at least one other behaviour-describing diagram (be it a sequence diagram, collaboration diagram, activity diagram or start chart or simple flow chart, etc.) and often more than one additional diagram



A hand-drawn UML diagram.