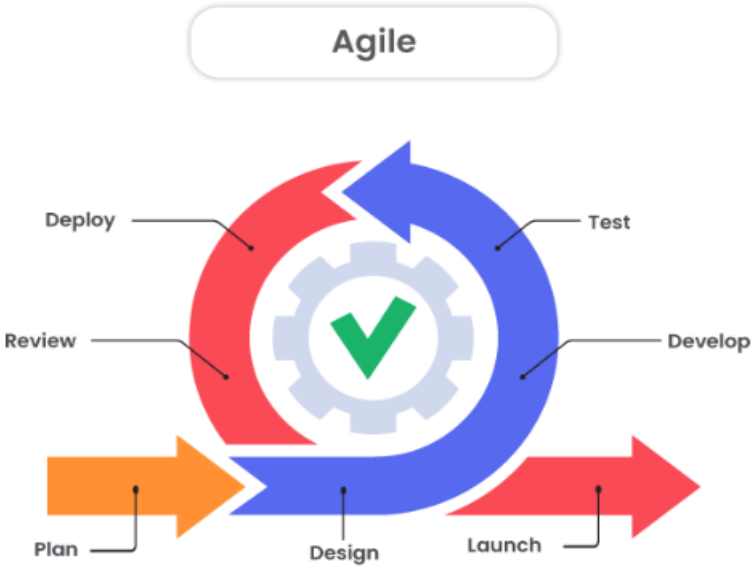
What is agility?

Agility means characteristics of being dynamic, content specific, aggressively change embracing and growth oriented.



Agile Process

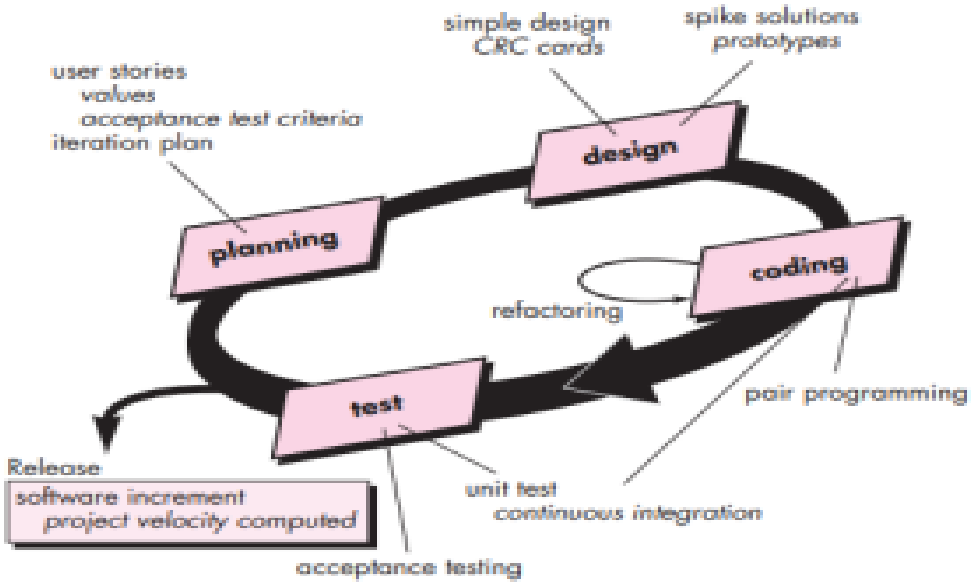
The Processes which are adaptable to changes in requirements, which have incrementality and work on unpredictability. These processes are based on three assumptions which all do refer to the unpredictability in different stages of software process development such unpredictability at time requirements, at analysis and design or at time construction.So these processes are adaptable at all stages on SDLC.

.

Agile Process models

1. Extreme Programming(XP)
2. Adaptive Software development(ASD)
3. Dynamic software Development Method(DSDM)
4. Scrum

5. Crystal
6. Feature Driven development (FDD)
7. Agile Modeling(AM)
1. Extreme Programming(XP)



This diagram represents the **Extreme Programming (XP) development cycle**. Here's a breakdown of each term and its role:

1. Planning Phase

- **User Stories**: Short, simple descriptions of a feature told from the perspective of the end user.
- **Values**: Core principles guiding the development (e.g., communication, simplicity, feedback).
- **Acceptance Test Criteria** *(italicized in diagram)*: Conditions that must be met for a story to be accepted as complete.
- **Iteration Plan**: A short-term schedule (often 1–2 weeks) defining what will be delivered.

2. Design Phase

- **Simple Design**: Creating the simplest system that works, avoiding overengineering.
- **CRC Cards**: "Class-Responsibility-Collaboration" cards — a lightweight tool for object-oriented design.

- **Spike Solutions** *(bold in diagram)*: Quick experiments to explore potential solutions before committing to a design.

- **Prototypes** *(italicized in diagram)*: Simplified versions of the final product to test ideas early.

3. Coding Phase

- **Refactoring**: Improving existing code without changing its behavior, to make it cleaner and easier to maintain.

- **Pair Programming**: Two developers work together at one workstation — one writes code, the other reviews in real time.

4. Testing Phase

- **Unit Test**: Automated tests checking that individual code components work as expected.

- **Continuous Integration** *(italicized in diagram)*: Frequently integrating and testing new code to detect problems early.

- **Acceptance Testing**: Verifying the system meets the business requirements set in acceptance criteria.

5. Release Phase

- **Software Increment**: A working piece of software with new features delivered at the end of an iteration.

- **Project Velocity Computed** *(italicized in diagram)*: Measuring how much work was completed in an iteration to predict future progress.

Extreme Programming (XP) is an **agile software development methodology** focused on delivering high-quality software quickly and responding flexibly to changing requirements.

It was introduced in the late 1990s by **Kent Beck** and is especially suited for projects with rapidly changing or unclear customer needs.

Core Idea

XP emphasizes:

- **Frequent releases** in short development cycles

- **Close collaboration** between developers and customers

- **Continuous feedback** from automated tests and stakeholders
- **Simplicity** in design
- **Courage** to refactor and improve code at any time

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Key XP activities are summarized in the paragraphs that follow

Planning. The planning activity begins with listening—a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality. Listening leads to the creation of a set of "stories" or user stories that describe required output, features, and functionality for software to be built. Each story is written by the customer and is placed on an index card. The customer assigns a value (i.e., a priority) to the story based on the overall business value of the feature or function. Members of the XP team then assess each story and assign a cost—measured in development weeks—to it. If the story is estimated to require more than three development weeks, the customer is asked to split the story into smaller stories and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time. Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team. Once a basic commitment is made for a release, the XP team orders the stories that will be developed in one of three ways: (1) all stories will be implemented immediately (within a few weeks), (2) the stories with highest value will be moved up in the schedule and implemented first, or (3) the riskiest stories will be moved up in the schedule and implemented first. After the first project release (also called a software increment) has been delivered, the XP team computes project velocity. Stated simply, project velocity is the number of customer stories
implemented during the first release. Project velocity can then be used to (1) help estimate delivery dates and schedule for subsequent releases and (2) determine whether an overcommitment has been made for all stories across the entire development project. As development work proceeds, the customer can add stories, change the value of an existing story, split stories, or eliminate them. The XP team then reconsiders all remaining releases and modifies its plans accordingly.

Design. XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. XP encourages the use of CRC cards as an effective mechanism for thinking about the software in an object-oriented context. CRC (class responsibility collaborator) cards identify and organize the object-oriented classes that are relevant to the current software increment. The CRC cards are the only design work product produced as part of the XP process.

If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a spike solution, the design prototype is implemented and evaluated. The intent is to lower risk when true implementation starts and to validate the original estimates for the story containing the design problem

A central notion in XP is that design occurs both before and after coding commences. Refactoring means that design occurs continuously as the system is constructed. In fact, the construction activity itself will provide the XP team with guidance on how to improve the design.

Coding. After stories are developed and preliminary design work is done, the team does not move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment).8 Once the unit test9 has been created, the developer is better able to focus on what must be implemented to pass the test. Nothing extraneous is added (KIS). Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers. A key concept during the coding activity is pair programming. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for realtime problem solving and real-time quality assurance It also keeps the developers focused on the problem at hand. In practice, each person takes on a slightly different role. For example, one person might think about the coding details of a particular portion of the design while the other ensures that coding standards are being followed or that the code for the story will satisfy the unit test that has been developed to validate the code against the story.

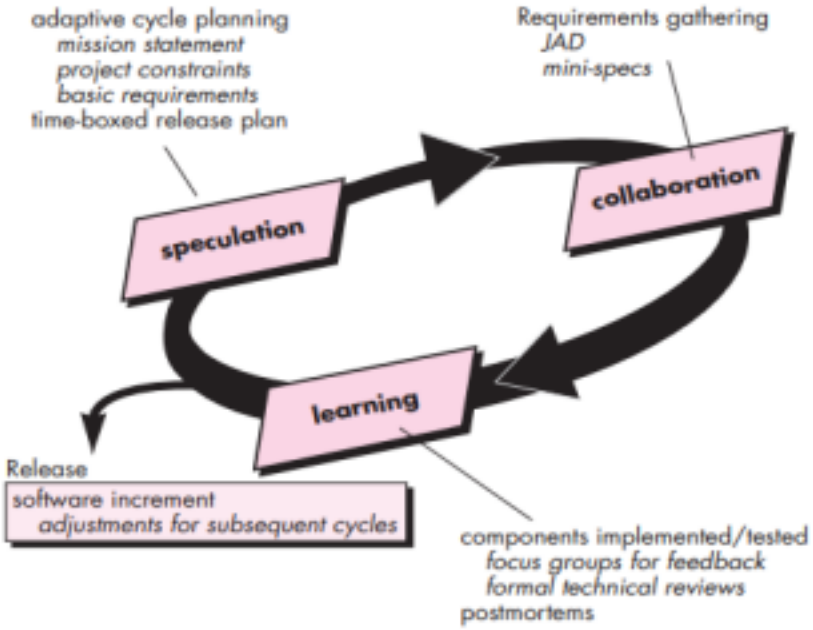
As pair programmers complete their work, the code they develop is integrated with the work of others. In some cases this is performed on a daily basis by an integration team. In other cases, the pair programmers have integration responsibility. This “continuous integration” strategy helps to avoid compatibility and interfacing problems and provides a “smoke testing” environment that helps to uncover errors early.

Testing The unit tests that are created should be implemented using a framework that enables them to be automated This encourages a regression testing strategy whenever code is modified As the individual unit tests are organized into a “universal testing suite” [Wel99], integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells [Wel99] states: “Fixing small problems every few hours takes less time than fixing huge

problems just before the deadline." XP acceptance tests, also called customer tests, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

2. **Adaptive Software development(ASD):**

Adaptive Software Development (ASD) has been proposed by Jim Highsmith, as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization. He defines an ASD "life cycle" that incorporates three phases, speculation, collaboration, and learning.



This diagram represents the **Adaptive Software Development (ASD) life cycle**, which is an **iterative and change-tolerant agile methodology**. Let's break down each part and term:

1. Speculation Phase

This is about **planning with the expectation of change** — unlike rigid plans in traditional methods.

- **Adaptive Cycle Planning**: Planning that allows for adjustments as the project evolves.
- **Mission Statement** *(italicized in diagram)*: A clear, concise description of the project's purpose.

- **Project Constraints** *(italicized)*: Limitations like budget, time, resources, or technology choices.
- **Basic Requirements** *(italicized)*: Core features that the product must have, known at the start.
- **Time-Boxed Release Plan**: A fixed timeframe for delivering a working release, regardless of scope changes.

2. Collaboration Phase

Here, **teamwork and stakeholder involvement** are key.

- **Requirements Gathering**: Collecting details about what the customer needs.
- **JAD (Joint Application Design)**: Structured workshops where stakeholders and developers collaborate on requirements and design.
- **Mini-Specs** *(italicized)*: Short, focused requirement descriptions for specific features.

3. Learning Phase

This is where the team **evaluates results, gains feedback, and adapts**.

- **Components Implemented/Tested**: Delivering and validating parts of the system.
- **Focus Groups for Feedback** *(italicized)*: Gathering user opinions to refine the product.
- **Formal Technical Reviews** *(italicized)*: Structured peer reviews to ensure quality and compliance.
- **Postmortems**: Retrospective meetings to analyze successes, failures, and lessons learned.

4. Release

After the learning phase:

- **Software Increment**: A functional portion of the software delivered to the customer.
- **Adjustments for Subsequent Cycles** *(italicized)*: Using lessons learned to improve the next iteration.
-

Adaptive Software Development (ASD) is an **agile software development methodology** designed for projects where requirements are **uncertain, changing, or emerging over time**.

It was introduced by **Jim Highsmith** in the 1990s as an evolution of Rapid Application Development (RAD) and emphasizes adaptability over strict planning.

Core Idea

ASD treats software development as a **continuous learning process** rather than a predictable, linear process. Instead of assuming we know everything at the start, ASD embraces uncertainty and change, using **short iterative cycles** to learn and adapt.

Its cycle has **three main phases**:

1. **Speculate** – Plan with the expectation that the plan will evolve.
2. **Collaborate** – Work closely with team members and stakeholders to respond to change.
3. **Learn** – Reflect on results, gather feedback, and adapt for the next cycle.

During speculation, the project is initiated and adaptive cycle planning is conducted. Adaptive cycle planning uses project initiation information—the customer's mission statement, project constraints , and basic requirements—to define the set of release cycles that will be required for the project. No matter how complete and farsighted the cycle plan, it will invariably change. Based on information obtained at the completion of the first cycle, the plan is reviewed and adjusted so that planned work better fits the reality in which an ASD team is working.

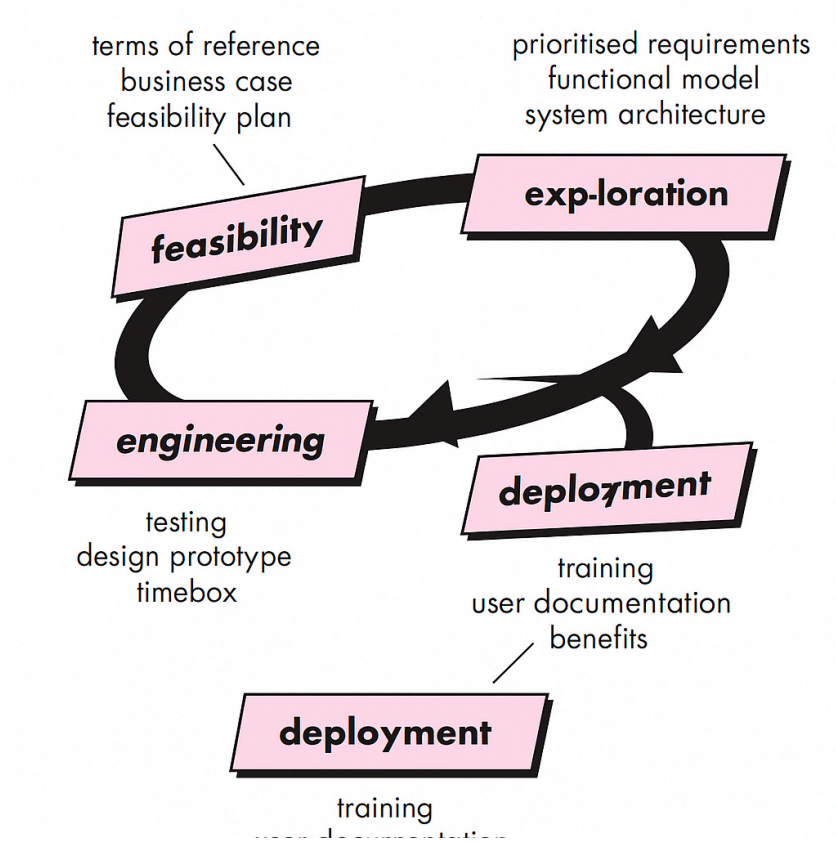
Motivated people use collaboration in a way that multiplies their talent and creative output beyond their absolute numbers. This approach is a recurring theme in all agile methods. But collaboration is not easy. It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking. It is, above all, a matter of trust. People working together must trust one another to (1) criticize without animosity, (2) assist without resentment, (3) work as hard as or harder than they do, (4) have the skill set to contribute to the work at hand, and (5) communicate problems or concerns in a way that leads to effective action.

As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on "learning" as much as it is on progress toward a completed cycle. software developers often overestimate their own understanding (of the technology, the process, and the project) and that learning will help them to improve their level of real

understanding. ASD teams learn in three ways: focus groups , technical reviews, , and project postmortems.

The ASD philosophy has merit regardless of the process model that is used. ASD's overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning yield software project teams that have a much higher likelihood of success.

3. Dynamic software Development Method(DSDM) :



This diagram illustrates the Dynamic Systems Development Method (DSDM) life cycle, showing its major stages and key activities at each stage. Here's a breakdown:

1. Feasibility

Purpose: Confirm the project is worth doing and can be done.

- Terms of Reference: A short statement describing the scope, objectives, and constraints of the project.
- Business Case: Justification for the project in terms of expected benefits and costs.

- Feasibility Plan: A plan that outlines how the feasibility study will be conducted, including resources, timelines, and evaluation criteria.

2. Exploration

Purpose: Understand requirements and explore possible solutions.

- Prioritised Requirements: Requirements ranked based on business value and urgency.
- Functional Model: Early representation (often prototypes) showing how the system will work.
- System Architecture: High-level design showing how system components fit together.

3. Engineering

Purpose: Build and refine the system components.

- Testing: Verifying that each component meets its specifications.
- Design Prototype: Detailed design of system components before full build.
- Timebox: A fixed period in which a set of tasks must be completed (encourages focus and discipline).

4. Deployment

Purpose: Deliver the working solution to the users.

- Training: Educating users on how to use the new system.
- User Documentation: Written guides and manuals for system use.
- Benefits: Realization of business value from the system.

Iteration & Flow

- The arrows between stages show iterative cycles — you can move between exploration, engineering, and deployment multiple times to refine the product.

- A final deployment stage ensures the solution is in the users' hands with all necessary support.

Dynamic Systems Development Method (DSDM) is an agile project delivery framework that originated in the UK in the mid-1990s as an improvement over Rapid Application Development (RAD).
It focuses on delivering business value quickly through time-boxed, iterative development while maintaining strict control over cost, time, and quality.

Core Idea

DSDM works on the principle that:

> "The greatest business value comes from delivering the right solution on time, even if that means delivering less functionality."

It achieves this by fixing time and resources, but flexibly adjusting scope.

The Dynamic Systems Development Method is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment". The DSDM philosophy is borrowed from a modified version of the Pareto principle—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete application. DSDM is an iterative software process in which each iteration follows the 80 percent rule. That is, only enough work is required for each increment to facilitate movement to the next increment. The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated. The DSDM Consortium is a worldwide group of member companies that collectively take on the role of "keeper" of the method. The consortium has defined an agile process model, called the DSDM life cycle that defines three different iterative cycles, preceded by two additional life cycle activities:

Feasibility study—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.

Business study—establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.
Functional model iteration—produces a set of incremental prototypes that demonstrate functionality for the customer. The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.

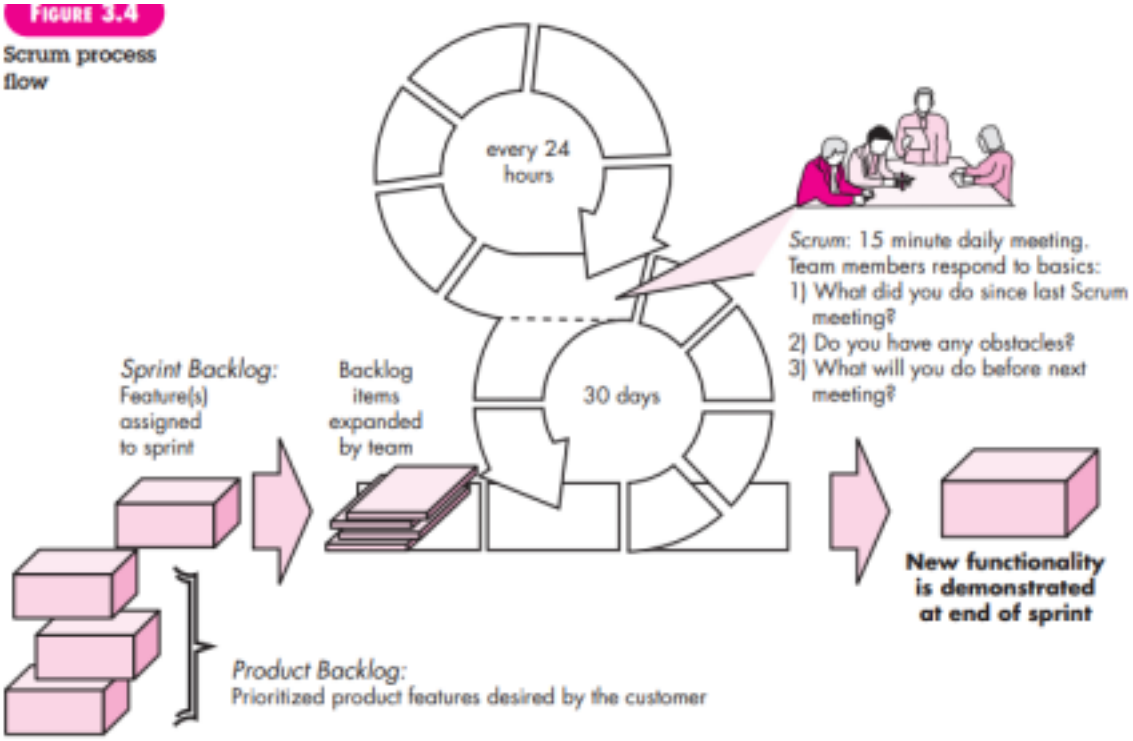
Design and build iteration—revisits prototypes built during functional model iteration to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, functional model iteration and design and build iteration occur concurrently.

Implementation—places the latest software increment into the operational environment. It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place. In either case, DSDM development work continues by returning to the functional model iteration activity.

DSDM can be combined with XP to provide a combination approach that defines a solid process model with the nuts and bolts practices (XP) that are required to build software increments. In addition, the ASD concepts of collaboration and self-organizing teams can be adapted to a combined process model.

4. Scrum :

FIGURE 3.4
Scrum process flow



Scrum (the name is derived from an activity that occurs during a rugby match) is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. In recent years, further development on the Scrum methods has been performed by Schwaber and Beedle.

Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks occur within a process pattern called a sprint. The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team.

Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development actions:

Backlog—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time (this is how changes are introduced). The product manager assesses the backlog and updates priorities as required.

Sprints—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box14 (typically 30 days).

Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment. Scrum meetings—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members.

• What did you do since the last team meeting?

• What obstacles are you encountering?

• What do you plan to accomplish by the next team meeting?

A team leader, called a Scrum master, leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to "knowledge socialization" [Bee99] and thereby promote a self-organizing team structure.

Demos—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.
The Scrum process patterns enable a software team to work successfully in a world where the elimination of uncertainty is impossible.

5. Crystal :

Alistair Cockburn and Jim Highsmith created the Crystal family of agile methods15 in order to achieve a software development approach that puts a premium on "maneuverability" during what Cockburn characterizes as "a resource limited, cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game".

To achieve maneuverability, Cockburn and Highsmith have defined a set of methodologies, each with core elements that are common to all, and roles, process patterns, work products, and practice that are unique to each. The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects. The intent is to allow agile teams to select the member of the crystal family that is most appropriate for their project and environment.

The **Crystal Method** is a family of **lightweight, people-centric agile methodologies** created by **Alistair**

Cockburn in the 1990s.

It's built on the idea that **every project is unique**, so the development process should be tailored to the project's **size, criticality, and priorities** instead of applying a rigid, one-size-fits-all approach.

Core Idea

Crystal focuses on:

- **People over processes** – the skills, communication, and interaction of team members matter most.
- **Adaptability** – adjust the methodology based on project needs.
- **Frequent delivery** – produce usable software early and often.
- **Continuous improvement** – regularly reflect and refine practices.

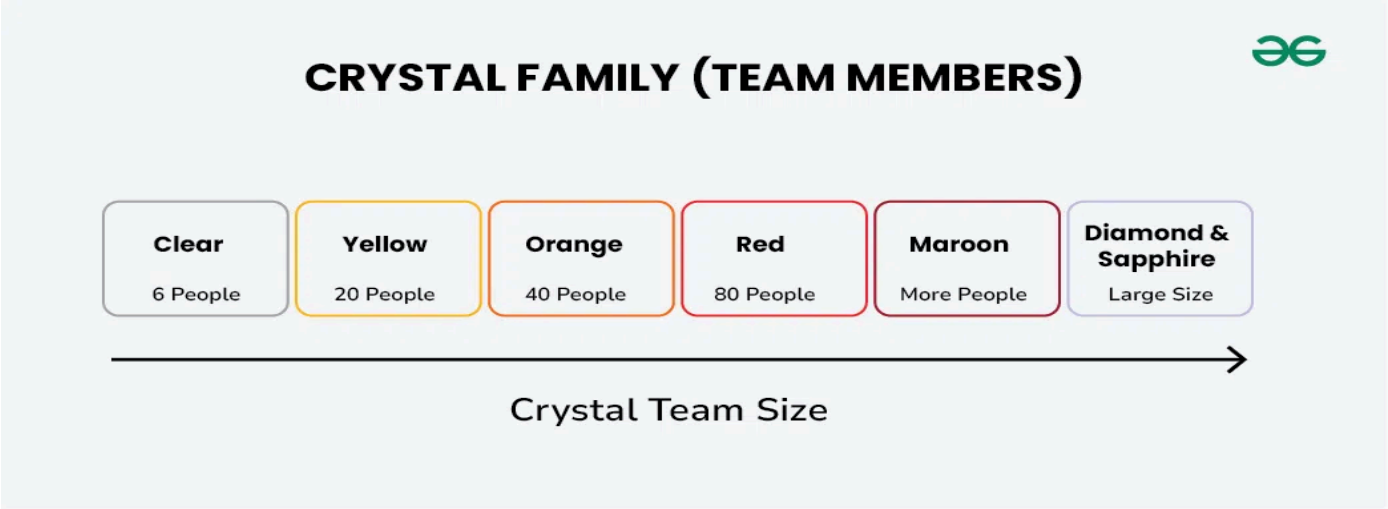
The "Colors" of Crystal

Crystal has variations (like **Crystal Clear, Crystal Yellow, Crystal Orange, etc.**) depending on:

- **Team size** (small to large)
- **System criticality** (loss of comfort → loss of money → loss of life)
- **Project complexity**

For example:

- **Crystal Clear** → For small teams (≤6 people), low-criticality projects.
- **Crystal Orange** → For medium-sized teams (up to ~40 people), medium-criticality.
- **Crystal Red** → For large projects with high criticality.



Various methodologies in the Crystal family also known as weights of the Crystal approach are represented by different colors of the spectrum.
The Crystal family consists of many variants like Crystal Clear, Crystal Yellow, Crystal Red, Crystal Sapphire, Crystal Red, Crystal Orange Web, and Crystal Diamond.

1. **Crystal Clear-** The team consists of only 1-6 members that is suitable for short-term projects where members work out in a single workspace.
2. **Crystal Yellow-** It has a small team size of 7-20 members, where feedback is taken from Real Users. This variant involves automated testing which resolves bugs faster and reduces the use of too much documentation.
3. **Crystal Orange-** It has a team size of 21-40 members, where the team is split according to their functional skills. Here the project generally lasts for 1-2 years and the release is required every 3 to 4 months.
4. **Crystal Orange Web-** It has also a team size of 21-40 members were the projects that have a continually evolving code base that is being used by the public. It is also similar to Crystal Orange but here they do not deal with a single project but a series of initiatives that required programming.
5. **Crystal Red-** The software development is led by 40-80 members where the teams can be formed and divided according to requirements.
6. **Crystal Maroon-** It involves large-sized projects where the team size is 80-200 members and where methods are different and as per the requirement of the software.

7. **Crystal Diamond & Sapphire-** This variant is used in large projects where there is a potential risk to human life.

6. Feature Driven Development (FDD):

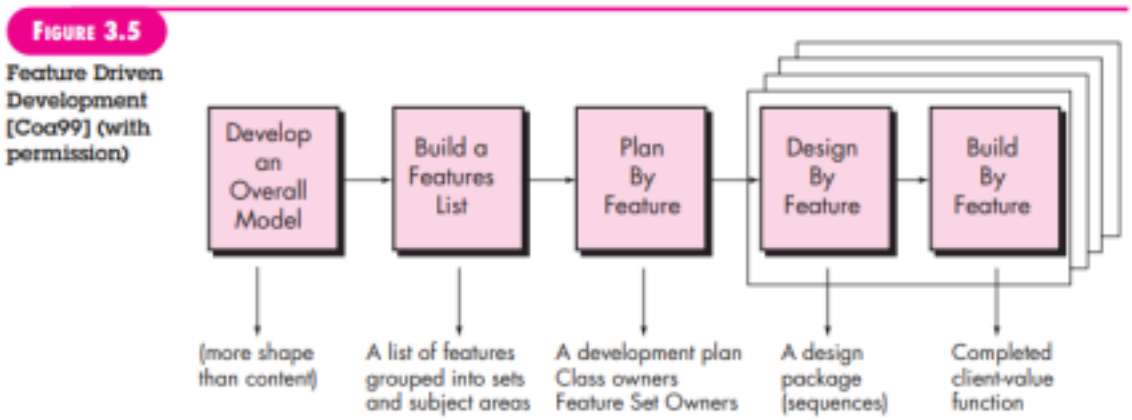


FIGURE 3.5
Feature Driven Development [Coa99] (with permission)

Typically used in large-scale development projects, five basic activities exist during FDD:

- Develop overall model
- Build feature list
- Plan by feature
- Design by feature
- Build by feature

Stage 0: Gather Data

As with all Agile methodologies, the first step in FDD is to gain an accurate understanding of content and context of the project, and to develop a clear, shared understanding of the target audience and their needs. During this time, teams should aim to learn everything they can about the why, the what, and the for whom about the project they're about to begin (the next few steps will help clarify the how).

Develop an overall model

Continuing the research paper metaphor, this stage is when the outline is drafted. Using the "thesis" (aka primary goal) as a guide, the team will develop detailed domain models, which will then be merged into one overall model that acts as a rough outline of the system. As it develops and as the team learns, details will be added.

Build a features list

Use the information assembled in the first step to create a list of the required features. Remember, a feature is a client-valued output. Make a list of features (that can be completed in two weeks' time), and keep in mind that these features should be purposes or smaller goals, rather than tasks.

Plan by Feature

Enter: Tasks. Analyze the complexity of each feature and plan tasks that are related for team members to accomplish. During the planning stage, all members of the team should take part in the evaluation of features with the perspective of each development stage in mind. Then, use the assessment of complexity to determine the order in which each feature will be implemented, as well as the team members that will be assigned to each feature set.

Design by Feature

A chief programmer will determine the feature that will be designed and build. He or she will also determine the class owners and feature teams involved, while defining the feature priorities. Part of the group might be working on technical design, while others work on framework. By the end of the design stage, a design review is completed by the whole team before moving forward.

Build by Feature

This step implements all the necessary items that will support the design. Here, user interfaces are built, as are components detailed in the technical design, and a feature prototype is created. The unit is tested, inspected and approved, then the completed feature can be promoted to the main build. Any feature that requires longer

time than two weeks to design and build is further broken into features until it meets the two-week rule.

7. Agile Modeling: (AM) :

There are many situations in which software engineers must build large, businesscritical systems. The scope and complexity of such systems must be modeled so that (1) all constituencies can better understand what needs to be accomplished, (2) the problem can be partitioned effectively among the people who must solve it, and (3) quality can be assessed as the system is being engineered and built.

Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. Agile models are more effective than traditional models because they are just barely good, they don't have to be perfect.

Agile modeling adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refactor. The team must also have the humility to recognize that technologists do not have all the answers and that business experts and other stakeholders should be respected and embraced.

Following are the characteristics of a good SRS document:

1. **Correctness:**

   User review is used to ensure the correctness of requirements stated in the SRS.

   SRS is said to be correct if it covers all the requirements that are actually expected from the system.

2. **Completeness:**

Completeness of SRS indicates every sense of completion including the numbering of all the pages, resolving the to be determined parts to as much extent as possible as well as covering all the functional and non-functional requirements properly.

3. **Consistency:**

   Requirements in SRS are said to be consistent if there are no conflicts between any set of requirements. Examples of conflict include differences in terminologies used at separate places, logical conflicts like time period of report generation, etc.

4. **Unambiguousness:**

   A SRS is said to be unambiguous if all the requirements stated have only 1 interpretation. Some of the ways to prevent unambiguousness include the use of modelling techniques like ER diagrams, proper reviews and buddy checks, etc.

5. **Ranking for importance and stability:**

   There should be a criterion to classify the requirements as less or more important or more specifically as desirable or essential. An identifier mark can be used with every requirement to indicate its rank or stability.

6. **Modifiability:**

   SRS should be made as modifiable as possible and should be capable of easily accepting changes to the system to some extent. Modifications should be properly indexed and cross-referenced.

7. **Verifiability:**

   A SRS is verifiable if there exists a specific technique to quantifiably measure the extent to which every requirement is met by the system. For example, a requirement stating that the system must be user-friendly is not verifiable and listing such requirements should be avoided.

8. **Traceability:**

   One should be able to trace a requirement to design components and then to code segments in the program. Similarly, one should be able to trace a requirement to the corresponding test cases.

9. **Design Independence:**

   There should be an option to choose from multiple design alternatives for the final system. More specifically, the SRS should not include any implementation details.

10. **Testability:**

    A SRS should be written in such a way that it is easy to generate test cases and test plans from the document.

11. **Understandable by the customer:**

    An end user maybe an expert in his/her specific domain but might not be an expert in computer science. Hence, the use of formal notations and symbols should be avoided to as much extent as possible. The language should be kept easy and clear.

12. **Right level of abstraction:**

If the SRS is written for the requirements phase, the details should be explained explicitly. Whereas, for a feasibility study, fewer details can be used. Hence, the level of abstraction varies according to the purpose of the SRS.

Understanding Requirements and Requirement modelling

Topics to be covered in Understanding Requirements

- Requirements Engineering
- Building the requirements model
- Establishing the groundwork
- Eliciting, Validating, Negotiating Requirements
- Requirements modelling

Requirement Engineering

Requirements engineering (**RE**) is the process of defining, documenting, and maintaining requirements in the engineering design process.

It is a common role in systems engineering and software engineering

Requirements engineering helps software engineers to better understand the problem they will work to solve

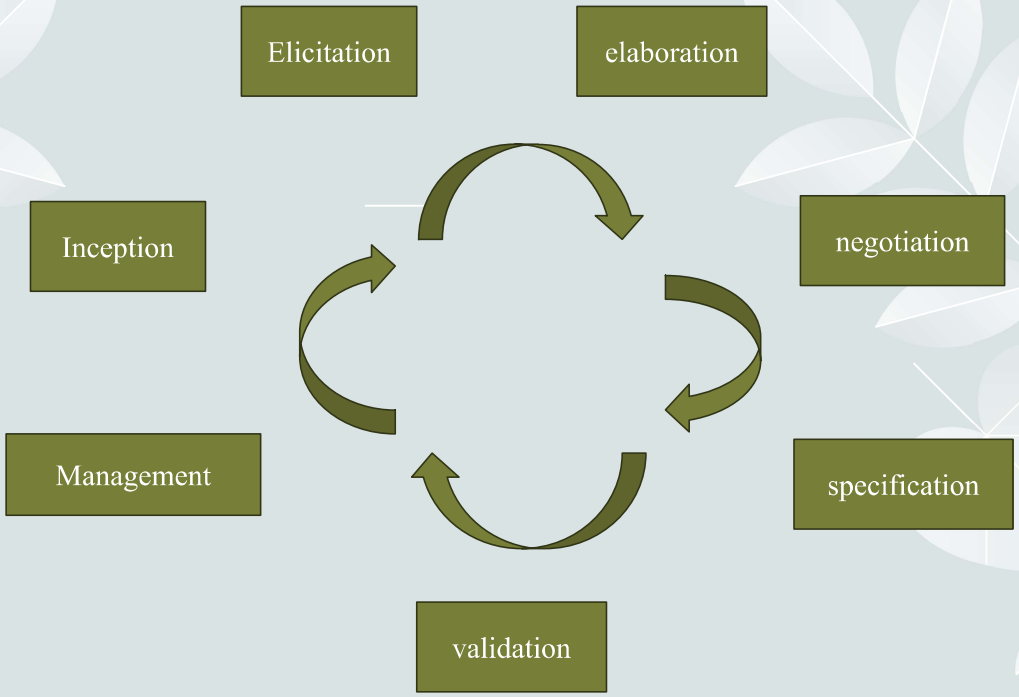
Conti..

- The broad spectrum of tasks and techniques that lead to understanding of the requirements is called requirement engineering.
- From s/w process perspective, requirements engineering is a major s/w engineering action that begins during the communication activity and continues into modeling activity
- Requirements engineering establishes a solid base for design and construction
- Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they transformed into an operational system
- It encompasses 7 distinct tasks, it is important to note that some of these tasks occur in parallel and all are adapted to the needs of the project
- All strive to define what the customer wants
- All serve to establish a solid foundation for the design and construction of the software

Conti..

The tasks are

1. Inception
2. Elicitation
3. Elaboration
4. Negotiation
5. Specification
6. Validation
7. Management



Inception

This is the first phase of the requirement analysis process or starting of the project

In this phase, it establishes a basic understanding of problem and nature of the solution

The requirement engineering itself is a communication “intensive activity” as it initial step to design, the customer and developer meet and decide the overall scope and nature of the problem statement

The aim is to,

- Have the basic understanding of the problem
- To know the people who will use the s/w
- To know exact nature of the problem that is expected from customer perspective
- To have collaboration between customer and developer

Through this the requirement engineer needs to

- Identify the stake holders
- Recognize multiple viewpoints
- Work toward collaboration between customer and developer
- Break the ice and initiate the communication

Elicitation

In this phase, the customer, users and others are involved and decides the objectives for the system or product

Meanwhile they also decides what to be accomplished and how the product fits into the needs of the business on the day to day basis

This phase focuses on gathering of the requirements from the stakeholders

The following problems can occur in the elicitation phase,

Problems of scope:

- the boundary of the system is not defined properly,
- The customer/ users specify unnecessary technical details that may confuse, rather clarify, overall system objectives

Problems of understanding

- The customer/ user are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment
- Problem domain is not clear

Problems of volatility

- The requirements change over time which leads to loss and wastage of resources and time

To overcome these problems, requirements gathering should be in an organized manner

Elaboration

- The information obtained from the customer during inception and elicitation phase is expanded and refined during elaboration phase
- This phase/ tasks mainly focuses on developing a refined requirements model that identifies various aspects of s/w function, behavior and information
- Elaboration is driven by the creation and refinement of user scenarios that describe the end user will interact with the system
- Each user scenario is parsed to extract analysis classes(attributes, business entities, services that are required by each class) that are visible to end user
- The relationship between classes are identified and variety of supplementary diagrams are produced
- This phase is also known as analysis modelling phase
- In elaboration phase mainly it focuses on gathering of information and defines the class domains that are pointing towards to end users
- It doesn't focus on design phase
- The problem with this phase is the developers must know when to stop

Negotiation

- Negotiation is between the developer and the customer about the limited availability of the resources, delivery time, project cost, overall estimation of the project
- Customers are requested to prioritize the requirements for the development & also discuss conflicts in priority
- Using an iterative approach that prioritizes requirements, assess their costs, risks and address internal conflicts.
- If the conflicts arises, it has to reconcile through the iterative approach in a process of negotiation
- With the same approach requirements are eliminated, combined or modified so that each party achieves some measure of satisfaction

Specification

- In the context of computer based system, the term specification means different things to different people
- A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype or any combination of these
- In this phase, the requirement engineer constructs final work product, the work product is in the form of SRS (S/w requirement specification) document
- For larger systems, a written document, graphical models, combining natural language descriptions are the best approach
- For smaller products or systems usage scenarios that reside within well understood technical environments
- ER, DFD, Data dictionary is models are used in this phase

SRS document- A s/w requirement specification document is a document that is created when a detailed description of all aspects of the s/w to be built must be specified before the project is to commence

Validation

- In this phase, the developers validates each phase such that the requirements have been refined and stated correctly or not as per stakeholders/ endusers perspective
- Requirement validation examines the specification to ensure all the requirements have been stated unambiguously
- It checks for errors, inconsistencies, omissions, have been detected and corrected
- A key concern in requirement validation is consistency
- Use the requirement analysis model to ensure that requirements have been consistently stated

Requirement Management

Requirement management is set of activities that help the project team identify, control, and track requirement & requirement changes to requirements at any time as the project proceeds

Many of these activities are identical to the s/w configuration management (SCM)

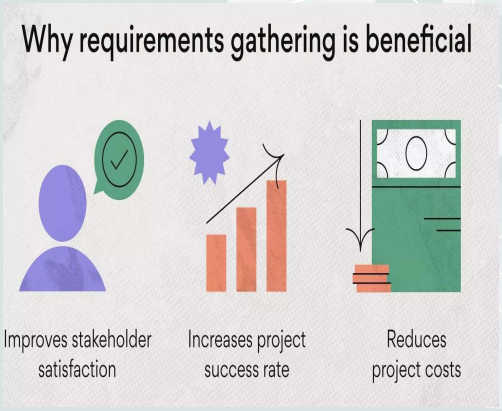
Based on this phase, working model will be analyzed carefully and ready to be delivered to the customer

Formal requirements management is initiated only for the large projects that have hundreds of identifiable requirements

For small projects, the requirement engineering action is considerably less formal

Establishing the groundwork

- Identifying stakeholders
- Recognizing multiple view- points
- Working towards collaboration
- Asking the first questions




Identifying Stakeholder
The "Anyone who directly or indirectly benefits from the system being built" is a little too vague in my opinion. Instead, at inception, determine who you want to get input from.
Recognizing Multiple Viewpoints
Don't pick only people from the same department. Getting a variety of views is important to determine who/what is the best options and input to contribute to the project's success.
Working toward Collaboration
When working with different groups, that each have their own ideas and agendas, it is important to work toward a common goal and get people to work together on what the actual end goals are going to be.
Asking the First Question
The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits.

Identifying stakeholders

Stakeholder-

- A stakeholder is anyone who has a direct interest in or benefits from the system that is to be developed
- Each stakeholder has a different view of the system, achieves different benefits when the system is successfully developed, and is open to different risks if the development effort should fail
- At inception phase, you should create a list of people who will contribute input as requirements are elicited
- The initial list will grow as stakeholders are contacted

Recognizing multiple viewpoints

- For every developing of a project there exists different stake holders, because of this the requirements of the system will be explored from many different points of view
- For example,

✔ Marketing group is interested in functions & features that will excite the potential market,

✔ Business managers are interested in a feature set that can be built within budget & that will be ready to meet defined market windows

✔ End users may want features that are familiar to them and that easy to use and learn

Conti..

- ✔ s/w engineers may be concerned with functions that are invisible to nontechnical stakeholders
- ✔ Support engineers may focus on the maintainability of the s/w
- Each of these constituencies will contribute information to the requirements engineering process
- As information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another
- You should categorize all stakeholder information in a way that will allow decision makers to choose an internally consistent set of requirements for the system

Working towards collaboration

- In this, all the stake holders, customers and other roles in the development project must collaborate among themselves to meet all requirements
- Meetings are conducted and attended by both s/w engineers and customers along with other interested stakeholders
- Rules for preparation and participation are established
- A “facilitator” (can be customer, developer or any outsider) controls the meeting
- A “definition mechanism” (can be worksheets, flip charts, electronic bullet board, chat room, virtual room) is used.
- Collaboration does not necessarily mean that requirements are defined by committee
- In many cases, stakeholders collaborate by providing their view of requirements but a business manager or senior technologist, may make the final decision about which requirements make the cut

Asking first questions

Questions asked at inception of the project should be " context free"

The first context free questions focuses on the customer and other stakeholders, the overall projects goals and benefits, some of the questions

- Who is the behind the request for this work?
- Who will use this solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all the stakeholders who will have interest in the s/w to be build

The next set of questions enables you to gain a better understanding of the problem & allows the customer to voice his/her perceptions about the solution

How would you characterize "good" output that would be generated by a successful solution?

What problems will this solution address?

Will special performance issues or constraints affect the way solution is approached?

The final set of questions focus on the effectiveness of the communication activity itself

These questions are also named as “meta questions” and proposed following list

- Are you the right person to answer these questions? are your answers are official
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

These questions will help to “break the ice” and initiate the communication that is essential to successful elicitation

Eliciting requirements

- Collaborative requirements gathering
- Quality function development
- Usage scenarios
- Elicitation work products

Collaborative requirements gathering

Requirements elicitation (also known as Requirements Gathering or Capture) is the process of generating a list of requirements (functional, system, technical, etc.) from the various stakeholders (customers, users, vendors, IT staff, etc.) that will be used as the basis for the formal Requirements Definition.

Requirements elicitation combines elements of problem solving, elaboration, negotiation, and specification

In order to encourage a collaborative, team-oriented approach to requirements gathering, stake holders work together to identify the problem, propose elements of solutions, negotiate different approaches & specifies set of preliminary solution requirements

- Meetings are conducted and attended by both s/w engineers and customers along with other interested stakeholders
- Rules for preparation and participation are established
- A “facilitator” (can be customer, developer or any outsider) controls the meeting
- A “definition mechanism” (can be worksheets, flip charts, electronic bullet board, chat room, virtual room) is used.
- Collaboration does not necessarily mean that requirements are defined by committee
- In many cases, stakeholders collaborate by providing their view of requirements but a business manager or senior technologist, may make the final decision about which requirements make the cut

JAD – joint application development is a popular technique for requirements gathering

Quality function deployment.

- QFD is a quality management technique that translates the needs of the customer into technical requirements of the s/w
- QFD defines requirements in a way that maximizes customer satisfaction
- To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys those values throughout the engineering process
- QFD identifies 3 types of requirements

1. normal requirements
2. expected requirements
3. exciting requirements

1. **Normal requirements**

The objectives and goals that are stated for a product or a system during meetings with the customer. Examples of normal requirements might be requested types of graphical displays, specific system functions, defined level of performance

2. **Expected requirements**

These requirements are implicit to the product or system and it is not fundamental that customer explicitly state them. Examples, ease of human/ machine interaction, overall operational correctness and reliability, ease of s/w installation

3. Exciting requirements:

These features go beyond the customer's expectations and prove to be very satisfying when present. Examples, s/w for new mobile phones comes with standard features but it is coupled with a set of unexpected capabilities.

Although QFD concepts can be applied across the entire s/w process, specific QFD techniques are applicabe to requirements elicitation activity

Usage scenarios

- After requirements gathered an overall vision of system functions and features begins to materialize
- However it is more difficult to get into more technical s/w engineering activities, to accomplish developers and users can create a set of scenarios, the scenarios are often called as use cases
- Use cases provide a description of how the system will be used.

Elicitation work products

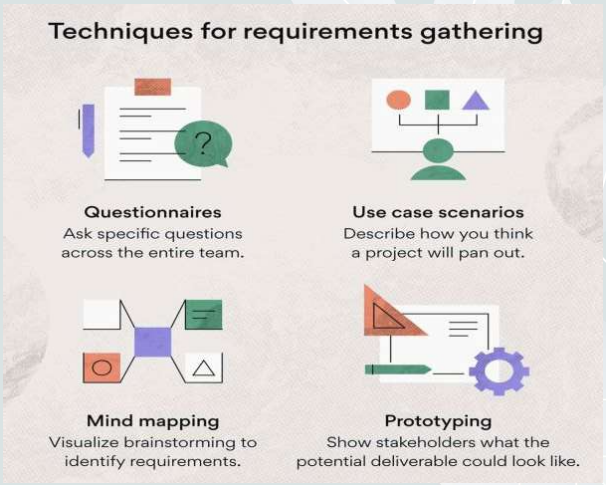
The work product produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built.

For most systems, the work product is

- A statement of need and feasibility
- A bounded statement of scope for the system or product.
- A list of customers, users, and other stake holders who participated in requirements elicitation

- A list of requirements and the domain constraints that apply each

  A description of the system’s technical environment



Developing use cases

- Use cases are defined from actor's point of view.
- An actor is a role that people (users) or devices play as they interact with the s/w
- The first step in writing the use cases is to define the set of "actors" that will be involved in the story
- Actors are different people or any devices that use the system or product within the context of the function and the behavior that is to be described
- It is important to note that an actor and an end user are not necessarily the same thing
- A typical user can play a number of different roles when using a system, where as an actor represent a class of external entities.
- As requirements elicitation is an evolutionary activity, not all actors are identified during first iteration.
- It is possible to identify primary actors at first iteration, primary actors interact to achieve required system function and derive the intended benefit from the system
- Secondary actors support the system so that primary actors can do their work
- Once actors are identified use cases can be developed

conti

- Some of the suggested questions to develop use cases are
- Who is primary actor, secondary actors?
- What are the actor goals?
- What preconditions should exist before the story begins
- What main tasks or functions are performed by the actor
- What information does the actor desire from the system
- What variations in the actor's interaction are possible
- Does the actor wish to be informed about unexpected changes

Building the Requirements Model

- Elements of the requirement model
- Analysis patterns

The analysis model's goal is to provide a description of the informational, functional, and behavioral domains required for a computer-based system.

The model evolves dynamically as you learn more about the system to be developed and other stakeholders gain a better understanding of what they require.

As a result, the analysis model represents a snapshot of requirements at any given time.

Elements of requirement model

There are many different ways to look at the requirements for a computer-based system.

The specific elements of the requirement-based model are come from the analysis modelling method that is to be used.

How ever a set of generic elements is common to most requirement methods

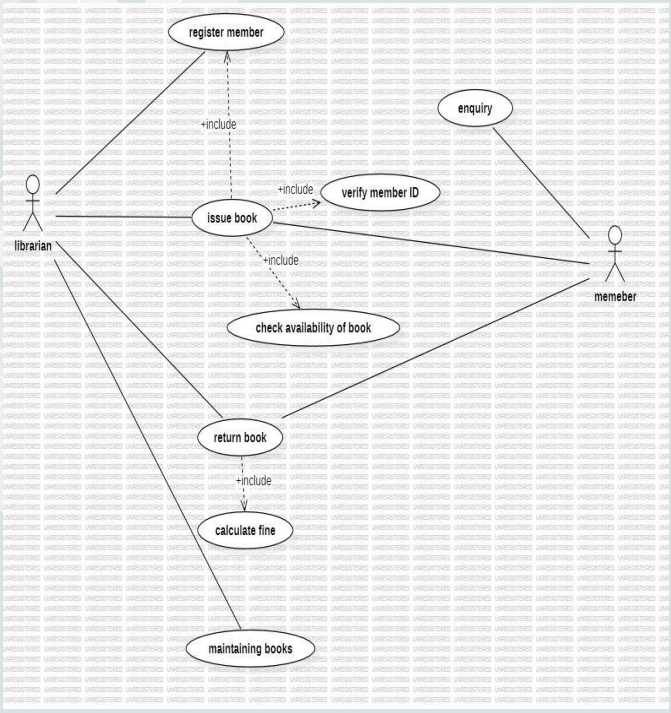
1. Scenario based elements
2. Class based elements
3. Flow oriented elements

4. **Scenario based elements**

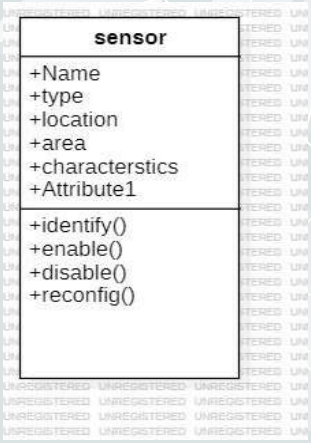
A scenario-based approach is used to describe the system from the perspective of the user. For example, basic use cases and their related use-case diagrams, evolve into more complicated template-based use cases. Scenario-based requirements model elements are frequently the first parts of the model to be produced.

Conti..

Fig : use case(scenario based) diagram for library management system



Class diagram



2. Class based elements

Each usage scenario implies set of objects that are manipulated as an actor interacts with the system. These objects are classified into classes

Classes – a collection of things that have similar attributes and common behaviours

A class consists of class name, attributes and operations as shown in fig

In addition to class diagrams, other analysis modelling elements depict the manner in which classes collaborate with one another and the relationships and the interactions between classes

3. Behavioral elements

- The behavior of computer -system can have a profound effect on the design that is chosen and the implementation approach that is applied, therefore the requirements model must provide modelling elements that depict behavior
- A state diagram is one method representing the behavior of a system by depicting its states and the events that cause the system to change state
- A state is any externally observable mode of behavior
- In adition the state diagram indicates actions taken as a consequence of particular event

UML state diagram notation

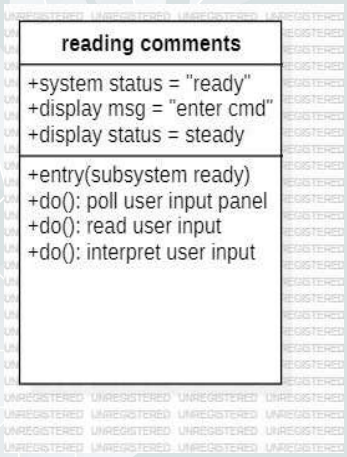
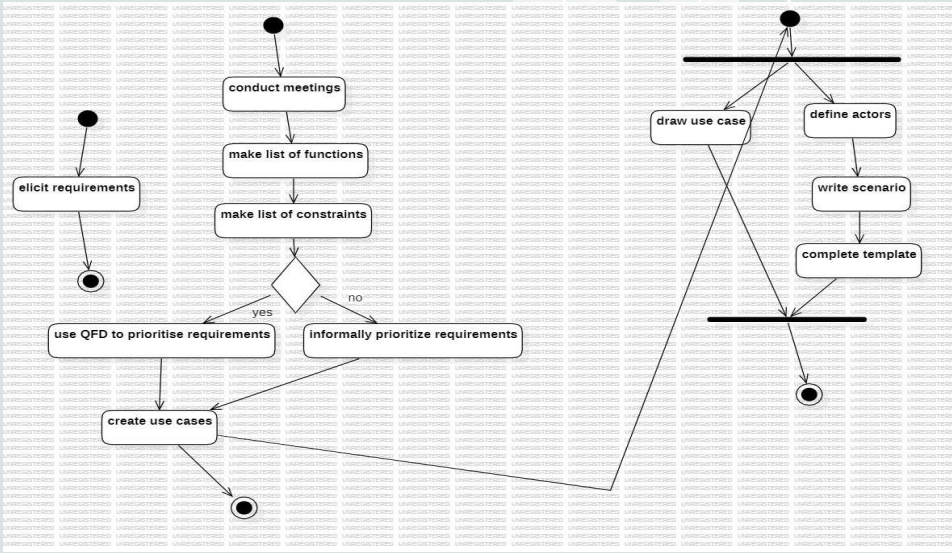


Fig: UML activity diagrams for eliciting requirements



Flow oriented elements

- Information is transferred as it flows through a computer-based system.
- The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms
- Input may be control signal transmitted by a transducer, a series of number typed by human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from the secondary storage

Analysis Patterns

Analysis patterns are integrated into the analysis model by reference to the pattern name. they are also stored in a repository so that requirements engineers can use search facilities to find and apply them

Information about an analysis pattern (and other type of patterns) also present in a standard template

Negotiating requirements

- The intent of this negotiation is to develop a project plan that meet stake holders needs while at the same time reflecting the real-world constraints
- The best negotiations strive for a “win-win” result which defines stake holders win by getting the system or product that satisfies majority of their needs and you as a member of s/w team win by working to realistic and achievable budgets and deadlines
- Boehm defines a set of negotiation activities at the beginning of each s/w process iteration
- Rather than a single customer communication activity, the following activities are defined

1. Identification of the system or subsystem’s key stakeholders
2. Determination of the stakeholders “win condition”
3. Negotiation of the stakeholders win conditions to reconcile them into a set of win-win conditions for all concerned

Validating requirements

Requirements Validation Techniques are used to ensure that the software requirements are complete, **Requirements validation** is the process of checking that requirements defined for development, define the system that the customer wants. To check issues related to requirements, we perform requirements validation. We typically use requirements validation to check errors at the initial phase of development as the error may increase excessive rework when detected later in the development process. In the requirements validation process, we perform a different type of test to check the requirements mentioned in the software requirement specification, these checks include:

1. **Completeness checks**
2. **Consistency checks**
3. **Validity checks**
4. **Realism checks**
5. **Ambiguity checks**
6. **Variability**

The output of requirements validation is the list of problems and agreed-on actions of detected problems.

Conti..

Requirement validation techniques

1. Test case generation
2. Prototyping
3. Requirement reviews
4. Automated consistency analysis
5. Check lists for validation
6. Simulation
7. Walk-through

Advantages	Disadvantages
Improved quality of the final product	Limited to functional requirements
Reduced development time and cost	Limited validation
Increased user involvement	Dependence on the tool
Traceability	Risk of changing requirements
Easy testing and validation	Risk conflicting requirements
Agile methodologies	Increased time and cost

Requirement modeling

Topic covered

- Requirement analysis
- Scenario based modelling
- Problem analysis
- SRS (s/w requirement specification)

Requirement analysis

- Requirement analysis results in the specification of s/w operational characteristics, indicates s/w ‘s interface with other system elements, and establishes constraints that s/w must meet
- Requirement analysis allows you to elaborate on basic requirements established during inception, elicitation, negotiation tasks that are part of requirement engineering
- The requirement modelling action results in one or more following steps of models

1. scenario-based modelling
2. data models
3. class-oriented models
4. flow-oriented models
5. behavioral models

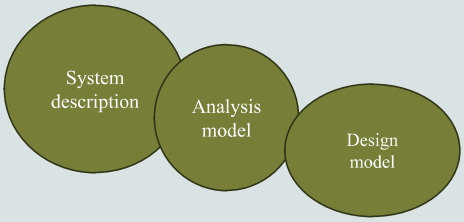


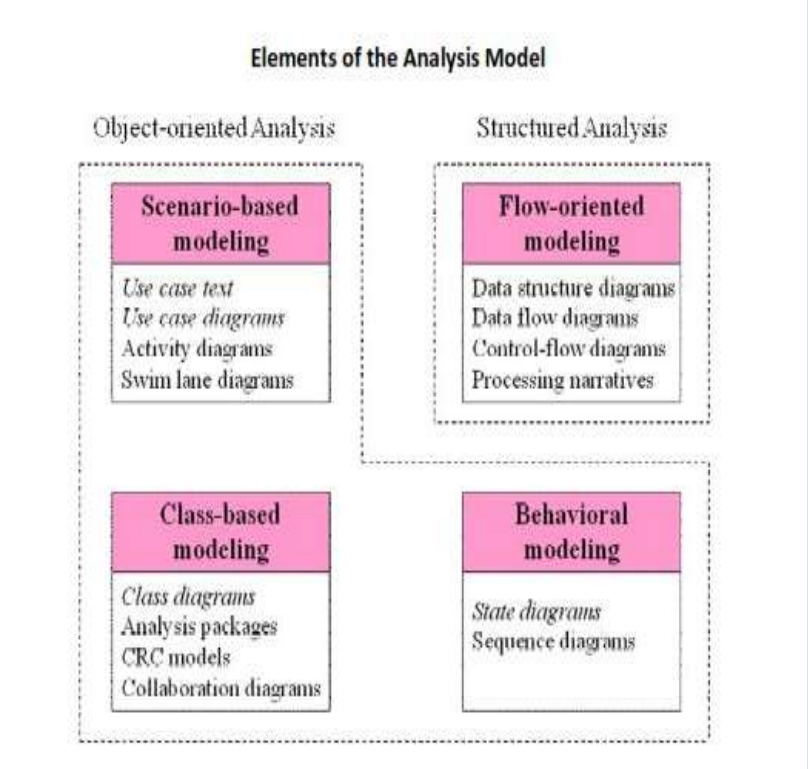
fig: the requirement model as bridge between the system description and the design model

These models provide a s/w designer with information that can be translated to architectural, interface, component level design

Problem analysis

Problem analysis is the process of understanding real-world problems and user's needs and proposing solutions to meet those needs. The goal of problem analysis is to gain a better understanding of the problem being solved, before development begins

Problem analysis is a series of steps for identifying problems, analyzing them, and developing solutions to address them.



Scenario based modelling

- Creating a preliminary use cases
- Refining a preliminary use cases
- Writing a formal use cases

(we have discussed these topics in previous slides)

Software Requirement specification (SRS)

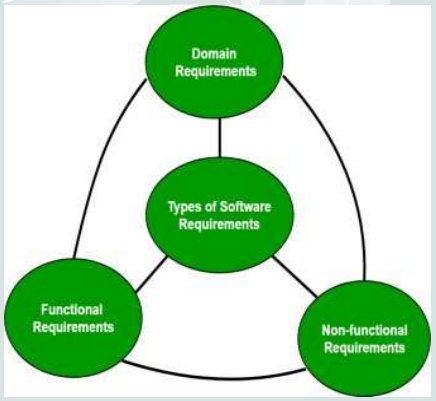
A software requirements specification (SRS) is a detailed description of a software system to be developed with its functional and non-functional requirements. The SRS is developed based the agreement between customer and contractors. It may include the use cases of how user is going to interact with software system.

SRS is **a formal report**, which acts as a representation of software that enables the customers to review whether it (SRS) is according to their requirements.

Software Requirements Specifications, also known as SRS, is the term used to describe an in-depth description of a software product to be developed. It's considered one of the initial stages of the software development lifecycle (SDLC).

The types of SRS

- Functional requirements
- Non-functional requirements
- Domain requirements



Functional requirements

- These are the requirements that the end user specifically demands as basic facilities that the system should offer.
- It can be a calculation, data manipulation, business process, user interaction, or any other specific functionality which defines what function a system is likely to perform.
- All these functionalities need to be necessarily incorporated into the system as a part of the contract.
- These are represented or stated in the form of input to be given to the system, the operation, performed and the output expected.
- Each high-level functional requirement may involve several interactions or dialogues between the system and the outside world.
- In-order to accurately describe the functional requirements, all scenarios must be enumerated.
- There are many ways of expressing functional requirements e.g., natural language, a structured or formatted language with no rigorous syntax and formal specification language with proper syntax.
- Functional Requirements in Software Engineering are also called Functional Specification.

Non-functional requirements

- These are basically the quality constraints that the system must satisfy according to the project contract.
- Nonfunctional requirements, not related to the system functionality, rather define how the system should perform. The priority or extent to which these factors are implemented varies from one project to other.
- They are also called non-behavioral requirements. They basically deal with issues like:

1. Portability
2. Security
3. Maintainability
4. Reliability
5. Scalability
6. Performance
7. Reusability
8. Flexibility

- Non-functional requirements handles with security and usability, which are observable at run time
- Evolution qualities like testability, maintainability, extensibility, and scalability that embodied in the static structure of the s/w system

Domain requirements

- Domain requirements are the requirements which are characteristic of a particular category or domain of projects.
- Domain requirements can be functional or nonfunctional.
- Domain requirements engineering is a continuous process of proactively defining the requirements for all foreseeable applications to be developed in the software product line.
- The basic functions that a system of a specific domain must necessarily exhibit come under this category.

Thank you

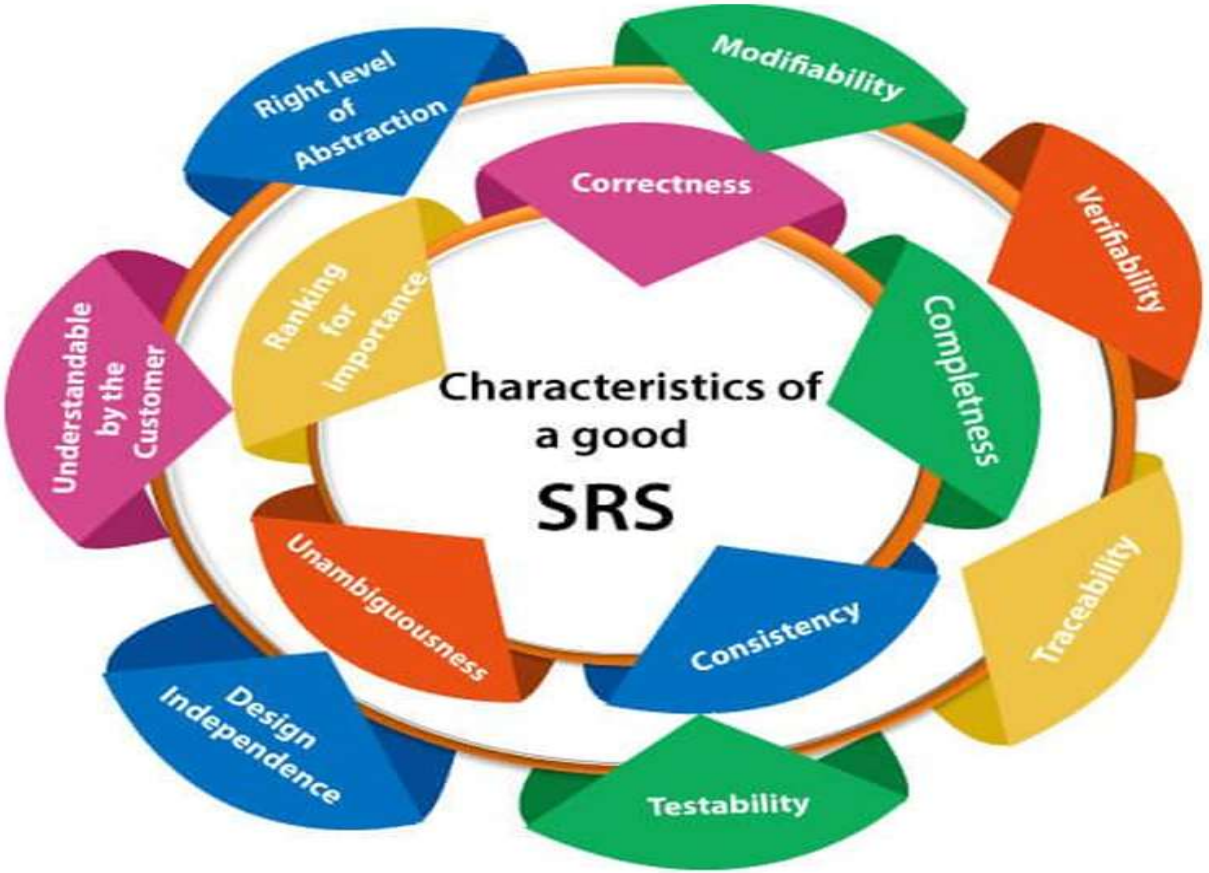
Software Requirement Specification

A software requirements specification (SRS) is a detailed description of a software system to be developed with its functional and non-functional requirements. The SRS is developed based the agreement between customer and contractors. It may include the use cases of how user is going to interact with software system. The software requirement specification document consistent of all necessary requirements required for project development. To develop the software system we should have clear understanding of Software system. To achieve this we need to continuous communication with customers to gather all requirements.

A good SRS defines the how Software System will interact with all internal modules, hardware, communication with other programs and human user interactions with wide range of real life scenarios. Using the *Software requirements specification* (SRS) document on QA lead, managers creates test plan. It is very important that testers must be cleared with every detail specified in this document in order to avoid faults in test cases and its expected results.

It is highly recommended to review or test SRS documents before start writing test cases and making any plan for testing. Let’s see how to test SRS and the important point to keep in mind while testing it.

Characteristics of good SRS



Quality characteristics of a good Software Requirements Specification (SRS) document include:

1. Complete: The SRS should include all the requirements for the software system, including both functional and non-functional requirements.
2. Consistent: The SRS should be consistent in its use of terminology and formatting, and should be free of contradictions.
3. Unambiguous: The SRS should be clear and specific, and should avoid using vague or imprecise language.
4. Traceable: The SRS should be traceable to other documents and artifacts, such as use cases and user stories, to ensure that all requirements are being met.
5. Verifiable: The SRS should be verifiable, which means that the requirements can be tested and validated to ensure that they are being met.
6. Modifiable: The SRS should be modifiable, so that it can be updated and changed as the software development process progresses.
7. Prioritized: The SRS should prioritize requirements, so that the most important requirements are addressed first.
8. Testable: The SRS should be written in a way that allows the requirements to be tested and validated.
9. High-level and low-level: The SRS should provide both high-level requirements (such as overall system objectives) and low-level requirements (such as detailed functional requirements).
10. Relevant: The SRS should be relevant to the software system that is being developed, and should not include unnecessary or irrelevant information.

Software Requirements Specification (SRS) Template

Example Case Study: Hospital Management System Using Scalable Architecture

1. Background

Hospitals face increasing pressure to manage patient data, appointments, billing, diagnostics, and staff coordination efficiently. Traditional systems often suffer from fragmentation, leading to delays, errors, and poor patient experience. This case study explores the development of a scalable, secure, and modular Hospital Management System (HMS) using distributed architecture principles.

2. Objective

To design and implement a cloud-based HMS that streamlines hospital operations, ensures data integrity, and supports real-time access for patients, doctors, and administrative staff.

3. System Modules

Module	Description
Patient Management	Registration, medical history, appointment scheduling
Doctor Management	Profiles, availability, patient assignments
Appointment System	Real-time booking, rescheduling, notifications
Billing & Payments	Invoice generation, insurance processing, payment gateway integration
Pharmacy Management	Inventory tracking, prescription fulfillment
Lab & Diagnostics	Test scheduling, report uploads, integration with diagnostic equipment
Admin Dashboard	Role-based access, analytics, audit logs

4. Technology Stack

- **Frontend**: React.js with Material UI
- **Backend**: Node.js with Express
- **Database**: MongoDB for patient records; PostgreSQL for transactional data
- **Authentication**: OAuth 2.0 with JWT
- **Cloud Platform**: AWS (EC2, S3, RDS, Lambda)
- **Streaming & Alerts**: Kafka for real-time notifications and updates
- **Security**: Role-based access control, encryption at rest and in transit, audit trails

5. Implementation Highlights

5.1 Data Flow

- Patient registers → appointment booked → doctor notified → consultation → prescription → billing → pharmacy/lab → discharge summary.

5.2 Real-Time Analytics

- Kafka Streams used to monitor appointment load, ER wait times, and pharmacy stock levels.

5.3 Scalability

- Microservices architecture allows independent scaling of modules (e.g., diagnostics vs. billing).

5.4 Security & Compliance

- Follows HIPAA guidelines for data privacy.
- Implements CoCoOn ontology for cloud security assurance.

6. Results

KPI	Before HMS	After HMS
Avg. Appointment Time	15 mins	3 mins
Billing Errors	12%	<1%
Patient Satisfaction Score	68/100	92/100
Staff Efficiency	Moderate	High

7. Challenges

- Integrating legacy systems with modern APIs
- Ensuring uptime during peak hours
- Training staff on new workflows

8. Future Enhancements

- AI-driven diagnostics and triage
- Federated learning for personalized treatment plans
- Wearable integration for remote patient monitoring
- Graph-based models for patient-doctor-network optimization