WaterFall Model

When It Is Used

- **Requirements are clear and fixed** → School's payroll rules don't change often.
- **Product definition is stable** → The system's purpose is just payroll calculation, not something innovative.
- **Technology is stable** → Can be built with well-known tools (like Java + MySQL).
- **No ambiguity** → All rules (salary slabs, tax rates) are documented.
- **Short project** → System may take only a few months.

Advantages

- **Easy to manage**: Clear, step-by-step phases (Requirements → Design → Coding → Testing → Deployment).
- **Defined deliverables**: At the end of the "Design" phase, you have design documents; after "Coding," you have the program.
- **Works well for small projects**: Payroll system is small and straightforward.
- **Well-documented**: Every phase produces documentation → future maintenance becomes easier.

Disadvantages

- **No working software until late**: The school cannot see anything until coding and testing are done.
- **No iteration support**: If halfway through the project the school wants a new feature (like online salary slip download), it's hard to add.
- **Hard to capture all requirements upfront**: The school may forget to mention allowances or leave encashment rules at the beginning.

- **Requires patience**: Teachers/admins only see the final product at the end, not in parts.
- **High risk in large projects**: If the payroll system were nationwide (millions of employees), risks would increase, and waterfall would not handle changes well.

RAD Model

Example Scenario

Imagine a company needs a **Hospital Appointment Booking System** .

- Features include: booking appointments, doctor availability, patient records, and billing.
- Requirements may change as doctors and hospital staff test the system.

Advantages in Action

- **Changing requirements**: If doctors want to add an *"emergency slot booking"* feature later, it can be added in a new prototype.
- **Measurable progress**: The hospital sees a working appointment module after just 2 weeks.
- **Faster development**: Different teams build patient records, appointment booking, and billing **in parallel**.
- **Customer involvement**: Doctors and staff give feedback on prototypes immediately.
- **Early integration**: Appointment module is integrated early with patient records → compatibility issues are solved quickly.
- **Reusability**: A billing component can be reused in other hospital systems.

- **Reduced time**: Hospital gets a working system in a few months instead of a year.

Disadvantages in Action

- **Skilled team needed**: Developers must quickly build prototypes with advanced RAD tools.
- **Only modular systems**: Hospital app can be divided into modules; but RAD won't work for something like an AI research project that isn't modular.
- **High cost**: RAD tools (like advanced prototyping platforms) are expensive → not suitable for a low-budget clinic.
- **Continuous user involvement**: Hospital staff must test and give feedback regularly.
- **Management complexity**: Coordinating multiple parallel teams (appointments, billing, records) is challenging.
- **Best for short timelines**: Works well since the hospital needs the system urgently.

Spiral Model

Example Scenario: Online Shopping Platform (like Amazon) 🛒

A company wants to build a new **e-commerce platform**, but:

- Customer requirements are **not fully clear**.
- There are **risks** with scalability, security, and payment integration.

How Spiral Works Here:

1. **1st Spiral (Planning & Risk Analysis)**
    - Gather initial requirements (product catalog, cart, payment).
    - Identify risks: payment security, server performance under high traffic.
    - Prototype: Build a simple catalog + cart system.
2. **2nd Spiral (Engineering & Evaluation)**
    - Add payment gateway prototype.
    - Analyze risks like transaction failures, fraud.
    - Get customer feedback on usability.
3. **3rd Spiral (Expansion)**
    - Add order tracking + shipping module.
    - Risk analysis: Can system handle logistics integration?
4. **4th Spiral (Deployment in Phases)**
    - Release beta version for a limited region.
    - Collect user feedback (issues with checkout, delivery).
    - Improve and continue to next spiral.

Advantages in this Example

- **Iterative & realistic**: Customers see prototypes early (cart, payment).
- **Accurate requirements**: Feedback helps refine features (e.g., "Add cash-on-delivery").
- **Risk-first approach**: Payment and scalability risks addressed before full rollout.
- **Flexibility**: Combines Waterfall's structure with adaptability to evolving needs.

Disadvantages in this Example

- **Expertise needed**: Strong risk analysis for payment security and server scaling.
- **Complex to manage**: Many modules (catalog, payment, logistics, delivery) must align.
- **Unclear project end date**: Could take many spirals to reach a stable, large-scale system.
- **High cost**: Not suitable for a small startup with limited budget.
- **Too many iterations**: Each new customer request could lead to another spiral.

Agile Model

Example Scenario: Food Delivery App (like Swiggy/Zomato) 🍔📱

A startup wants to build a **food delivery app**, but customer needs may change quickly.

How Agile Works Here:

1. **Iteration 1**: Deliver a basic app with restaurant listing + order placement.
    - Customers can browse restaurants and place orders.
    - Feedback: “Add payment options.”
2. **Iteration 2**: Add payment gateway + order tracking.
    - Customers test online payment.
    - Feedback: “Add live delivery tracking.”
3. **Iteration 3**: Add GPS-based delivery tracking.
    - Customers test it in real time.
    - Feedback: “Add reviews & ratings.”
4. **Iteration 4**: Add reviews, discounts, and loyalty points.
    - App becomes more engaging.

👉 With each iteration, **customers see working features early** and give feedback, shaping the final product.

Advantages in this Example

- **Realistic**: Customers don't wait for 1 year → they use the app after just 2–3 weeks.
- **Teamwork & cross-training**: Developers, testers, and designers collaborate closely.
- **Rapid development**: A simple working app is ready quickly.
- **Handles changing needs**: New features like live tracking can be added mid-project.
- **Early partial solutions**: Even early versions are usable (basic food ordering).
- **Customer satisfaction**: Continuous feedback ensures the app meets expectations.

Disadvantages in this Example

- **Effort estimation issues**: Hard to know at the start how long *all features* will take.
- **Less focus on design/docs**: The app may lack detailed documentation for future teams.
- **Risk of confusion**: If customers don't know what they want, the app could keep changing.
- **Skilled team needed**: Developers must handle rapid changes and quick deliveries.

Extreme Programming

Step 1: XP Planning

- Team collects **User Stories** from the customer:
    - *"As a user, I want to check my account balance."*
    - *"As a user, I want to transfer money securely."*
- Team estimates cost and effort for each story.
- Stories are grouped into **increments**.
- First increment includes "Balance Check" + "Login."
- Delivery date fixed: 3 weeks.
- After this, **project velocity** (rate of completed stories) helps plan future increments.

Step 2: XP Design

- Team applies **KIS (Keep It Simple)**: avoid unnecessary complexity.
- Uses **CRC Cards**:
    - Class: *Account*
    - Responsibility: *Maintain balance info*
    - Collaborator: *Transaction*

- For tricky features (like secure fund transfer), the team builds a **spike solution** prototype to test encryption methods.

- Continuous **refactoring**: improves login security logic without changing external behavior.

Step 3: XP Coding

- Developers write **unit tests before coding** (Test-Driven Development):
    - Test if login rejects invalid passwords.
    - Test if transfer deducts the right amount.
- **Pair Programming**: Two developers sit together → one codes, other reviews instantly.
- This ensures fewer bugs and better design decisions.

Step 4: XP Testing

- **Daily execution of unit tests**: ensures yesterday's features don't break today's code.
- **Acceptance Tests**: Defined by customer:
    - *"When I transfer ₹1000, my balance should reduce, and beneficiary should see credit instantly."*
- Once tests pass, the feature is marked as complete.

Adaptive Software Development

Step 1: Speculation

- **Mission Statement**: “Enable doctors to remotely monitor patients’ health in real-time.”
- Initial requirements:
    - Connect wearable devices.
    - View vitals dashboard.
    - Alert doctors on abnormal readings.
- Delivery dates are set, but planning is **adaptive** because requirements may change as doctors give feedback.

Step 2: Collaboration

- Team forms a **self-organizing group** of developers, healthcare experts, and UX designers.
- **Joint Application Development (JAD)** workshops are held: doctors explain what’s essential (*e.g., alerts for heart rate spikes*).
- Everyone collaborates daily to refine features and handle new challenges.

Step 3: Learning

- First cycle: Build a basic prototype that tracks **heart rate + BP**.
- Doctors test it and say: *"We also need glucose monitoring integrated."*
- Team gathers feedback via **reviews, focus groups, and postmortems**.
- Next cycle: Refine vitals dashboard + add glucose tracking.
- The process continues, with each cycle adding **tested, working features**.

DSDM

Scenario:

A **government tax department** 🏛️ wants to launch an **Online Tax Filing System** before the upcoming financial year. Time is fixed (it must go live before April 1st), requirements may evolve, and business value (citizens being able to file taxes easily) is the priority.

Step 1: Feasibility Study

- Business need: Citizens must be able to **file taxes online** securely.
- Constraints: Must be ready in 6 months, with strict government deadlines.
- Decision: DSDM chosen since it supports **tight deadlines + incremental delivery**.

Step 2: Business Study

- High-level requirements defined:

    - User login with Aadhaar/PAN.
    - Tax form filling.
    - Document uploads (salary slips, investment proofs).
    - Payment gateway for tax payment.
- Requirements are **baselined** → prevents endless scope creep.

Step 3: Functional Model Iteration

- **First prototype**: Simple login + form-filling page.
- **Feedback**: Officials suggest adding "auto-calculate tax" feature.
- **Second prototype**: Adds tax calculator + PDF download of form.
- **Third prototype**: Adds partial payment + save-for-later options.
  👉 Each iteration delivers a **working system**, not just documents.

Step 4: Design and Build Iteration

- Early prototypes made robust for real-world use:
    - Secure encryption for Aadhaar login.
    - Scalable server setup to handle peak loads (March rush).

- Features refined until system provides **business value** → easy, reliable tax filing.

Step 5: Implementation

- The latest increment (operational prototype) is deployed in **pilot mode** for government employees first.
- After fixing issues, it is rolled out nationally before the April deadline.

Key Principles in Action

- **Active user involvement**: Tax officers + pilot users give continuous feedback.
- **Empowered teams**: Developers make quick changes (e.g., updating slab rates).
- **Frequent delivery**: Citizens see functional prototypes early.
- **Business fitness**: Focus on core features (secure login, tax calculation) instead of fancy extras.
- **Reversible changes**: If a new payment option causes errors, it can be rolled back.
- **Time-boxing**: Each phase delivers within fixed deadlines → ensures April 1st launch.

Scrum

A startup is building a **Food Delivery App** 🍕📱 (like Zomato/Swiggy). Requirements will evolve based on customer feedback and market competition.

Step 1: Backlog Creation

- The **Product Backlog** (list of requirements) is created:
    - Restaurant listing
    - Menu display
    - Cart & order placement
    - Payment gateway
    - Live delivery tracking
    - Ratings & reviews
- Backlog is **prioritized**: Restaurant listing + order placement are most important.

Step 2: Sprint Planning

- **Sprint 1 (2 weeks)**: Focus only on *restaurant listing* + *menu display*.
- The team commits to delivering these features in the sprint.

Step 3: Sprint Execution

- During Sprint 1:
    - Developers build restaurant listing + menu screen.

    - Testers continuously test new code.
    - Documentation is updated in parallel.

- **Daily Scrum Meetings (15 min)**:
    - “What did I do yesterday?”
    - “What will I do today?”
    - “Any blockers?”

This keeps everyone aligned.

Step 4: Demo & Review

- At the end of Sprint 1 → A **working increment** is demoed:
    - Users can browse restaurants and view menus.
- Customer feedback: *“Add filters like veg/non-veg.”*
- This feedback goes into the backlog for future sprints.

Step 5: Next Sprint

- **Sprint 2 (2 weeks)**: Add cart + order placement.
- **Sprint 3**: Add payment gateway.

- **Sprint 4**: Add live delivery tracking.
- Each sprint produces a **usable, tested increment**.

Principles & Benefits in Action

- **Packets**: Work broken into small chunks (listings, cart, payments, tracking).
- **Backlog**: Priorities adjusted based on user feedback (filters moved up).
- **Sprints**: 2-week time-box ensures regular delivery.
- **Daily Scrum**: Keeps team on track, resolves blockers quickly.
- **Demos**: Customers see a working product early (after just 2 weeks).
- **Ongoing Testing & Documentation**: Bugs are caught early, and docs stay updated.

Crystal

Scenario:

A small company wants to build an **Internal Project Management Tool** 🗂️ for its 10-member marketing team to track campaigns, tasks, and deadlines. The project is not life-critical, but must be delivered quickly and adapted as the team's needs evolve.

Step 1: Choosing Crystal Variant

- Since the team is **small (10 people)** and project is **low criticality**, they select **Crystal Clear** (a lightweight Crystal method).

- If it were a bigger or safety-critical project, a heavier Crystal variant (like Crystal Orange) would be chosen.

Step 2: Teamwork & Communication

- The team works closely together in the same office.
- Daily informal check-ins → "What's done? What's next? Any issues?"
- Strong **face-to-face communication** instead of heavy documentation.

Step 3: Simplicity & Reflection

- Initial release focuses only on **basic task creation and assignment**.
- After first delivery, the team reflects:
    - *"Tasks need tagging and deadlines."*
    - They adapt the process and add these features in the next cycle.

Step 4: Delivery of Working Software

- First increment (2 weeks): Simple task management system.
- Second increment (2 weeks): Add tagging + deadlines.

- Third increment: Add campaign tracking dashboards.
  👉 At every step, **users get useful, working software**.

Step 5: Preparing for the Future ("Next Game")

- After tool is stable, the team prepares for **future needs**:
    - May integrate with email/calendar systems.
    - May expand tool for larger departments.

FDD

Example: Online Banking Application

Project Goal: Build a new online banking system.

1. **Feature List (client-valued functions):**
    - "Calculate monthly interest"
    - "Transfer funds between accounts"
    - "Generate account statement"
    - "Send transaction notification"
2. **Feature Decomposition:**
    - Feature: "Transfer funds between accounts"
        - Action: Transfer

        - Result: Funds moved successfully
        - Object: From one account to another

3. **Development Process:**
    - Iteration 1: Implement “Calculate monthly interest”
    - Iteration 2: Implement “Transfer funds between accounts”
    - Iteration 3: Implement “Generate account statement”
4. **Quality Checks:**
    - Before completing each feature, perform **code inspection**.
    - Conduct **SQA audits** to check compliance.
    - Collect **metrics** like how many features are implemented per week.
5. **Communication:**
    - Developers discuss **feature design verbally** in meetings.
    - Draw **UML diagrams** for “Transfer funds” flow.
    - Write **documentation** for the implemented feature.

Agile Modeling

Practical Scenario: E-commerce Website

- **Purpose:** Ensure smooth checkout workflow.
- **Models Used:**

    - **Sequence Diagram:** Shows interaction between user, cart, and payment gateway.

    - **Class Diagram:** Shows structure of Cart, Product, and Order classes.

- **Lightweight Approach:** Only create diagrams for checkout and payment, not every minor page.

- **Focus on Content:** Diagrams clearly communicate logic to the team, not how they look.

- **Local Adaptation:** Team decides to draw diagrams digitally for easier updates.