1 Software Development Process

1.1 Introduction

The **software development process** is a set of activities and associated results that produce a software product.

Fundamental Process Activities

Software specification:	**Definition** of the software to be produced and the constraints of the operation
Software development:	**Design**, **implementation**, **verification** of the software
Software validation:	**Ensure** that the software behaves according to the **requirements**
Software evolution:	**Adaptation** and **modification** of the software to cope with changing requirements

1.1.1 Motivation

Size of the Task

Organize a potentially large team
- Assign responsibilities
- Define modes of collaboration

Complexity of the Task

Many different kinds of activities
- Dependencies among tasks, when to do what

Quality Control

Need to be able to know whether things go wrong or right

1.1.2 Software Engineering Process Models

Software Engineering Process Models are simplified and abstract descriptions of software processes that present **one view** of that process.

They may include activities that are part of the software process, software products (e.g. architectural descriptions, source code, documentation…) and the **roles** of people involved in software engineering.

Large projects may use different, multiple software process models to develop different parts of the software.

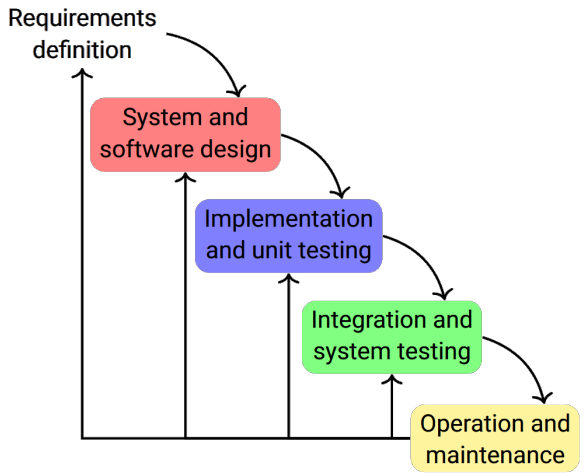
1.2 Waterfall Model

The **waterfall model** is a **linear** development process model that has the following phases:

Waterfall Phases

1. **Requirements Analysis and Definition:**
   - Requirements are established by consultation with system users
   - **Requirements are defined in detail and serve as system specification**
2. **System and Software Design:**
   - Definition of the overall system **architecture**
   - Identification of the **fundamental abstractions** and their relations
3. **Implementation and Unit Testing:**
   - Software design is realizes as a set of **program units**
   - **Testing** verifies that each unit meets its specification.
4. **Integration and System Testing:**
   - Program units are **integrated** and tested as a complete system
5. **Operation and Maintenance**

- The result of each phase is a set of approved artifacts
- Following phase start **after** the previous one is finished
- In case of errors the previous phases are **repeated**
- Aligns with traditional (physical) engineering process models



1.2.1 Criticism

The waterfall model in general does not work well for software engineering.

- **Not iterative:** early prototyping is not possible
- **Change** of requirements, design...difficult
  - Major Changes are undesirable, even minor changes are expensive
- **Testing** starts only at the later stages
- **Different phases** executed by **different teams**
  - Might no longer be available once a phase is finished

1.2.2 V-Model

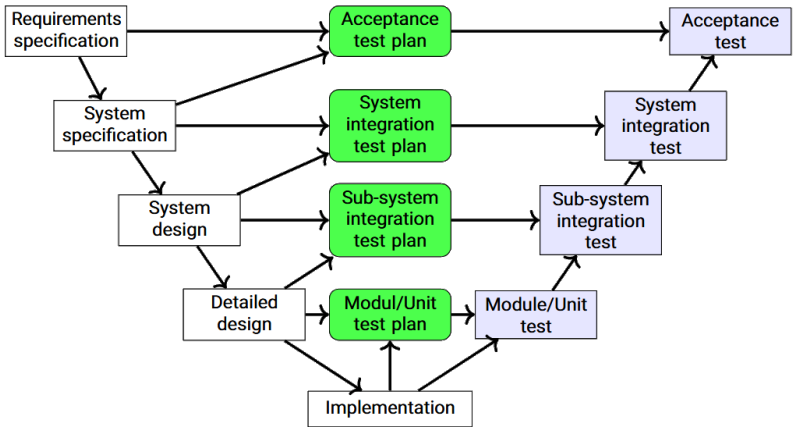
The V-Model is closely related to the waterfall method.

As a result most of the criticism of the waterfall method carries over.

- Not iterative
- Change of requirements expensive

Should only be used when:

- Project requirements **stable** and **well-known**
- **Small, short** project
- **Very precise and detailed** requirements and documentation (e.g. safety critical, like medical devices, avionics…)



1.3 Agile Development

Agile development is **centered on maintenance**.

Goal

Develop software **quickly** in presence of **rapidly changing** requirements

Development cycles should be small and fast: **agile**.

Originally for small teams (3-9 teams members).

1.3.1 Requirements

- Analyst, **customer**, developer, tester etc. work together as a team
  - Necessitates that all role players are always available
- Employ practices that provide necessary **discipline** and **feedback**
- Employ desing principles that keep software **flexible** and **maintainable**

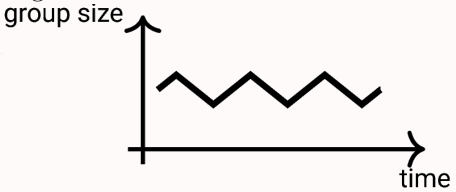
Agility is not a substute for validation: Customer **must** commit to become a team member

1.3.2 Manifesto

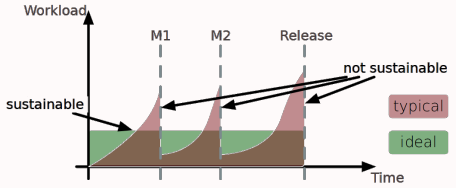
1: Individuals and Interactions over Process and Tools

The best tools will not help, if your team does not work together.

Team size should start small - Grow or shrink as needed



Workload should be sustainable: No bursts



Team should regularly reflect on process, work environment…

2: Working Software over Comprehensive Documentation

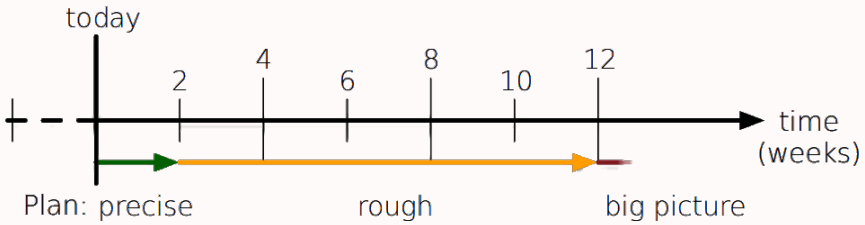
System structure and rationales for the design should be documented, but **incremental** rather than upfront. Not an excuse for **lack of documentation**, simply a recommendation of workflow. In agile development, code plays a central role so it must be **even better** documented than elsewise.

3: Customer Collaboration over Contract Negotiation

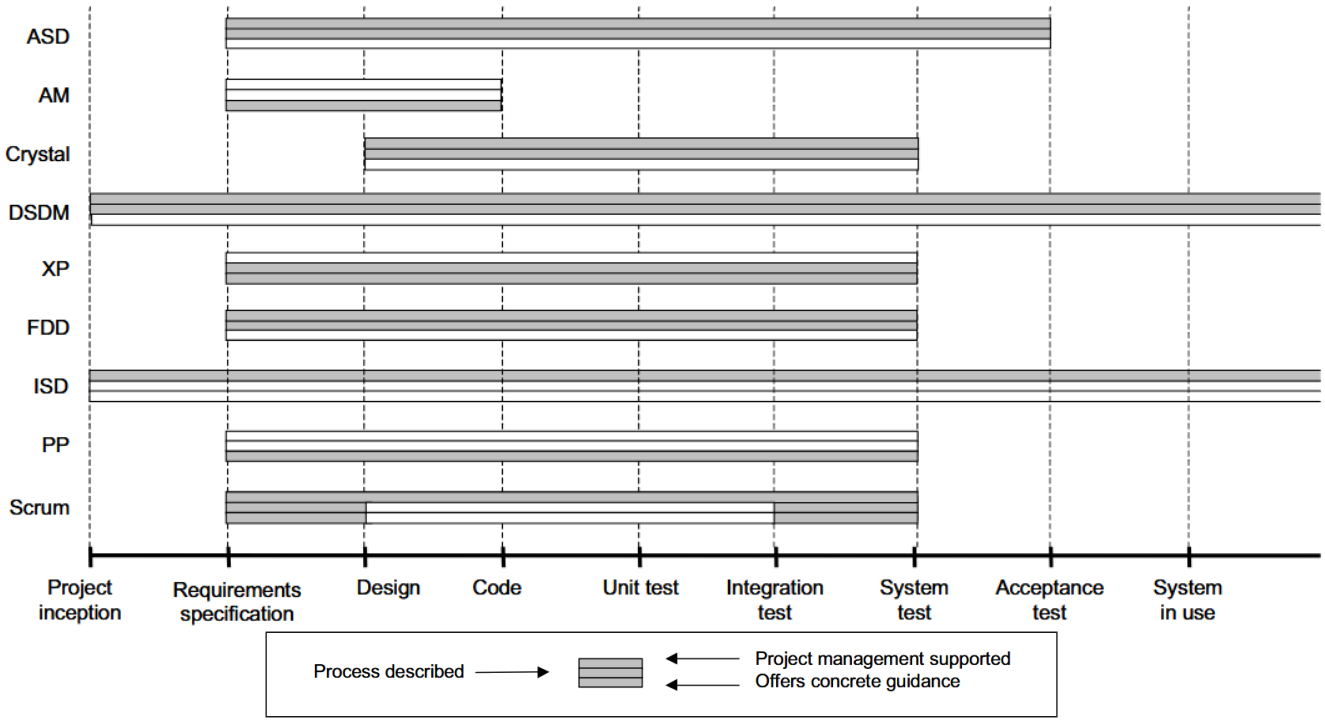
Contract should specify how the collaboration between development team and customer looks like.
A contract that merely specifies payment and delievery is **not enough**, a customer wants an idea of what their money is doing.

4: Responding to Change over Following a Plan

A plan outlined at the beginning of the project might be followed less and less over time, as requirements change and make the plan suboptimal. Sticking to a rigid plan might be counterproductive.



1.3.3 Agile Processes



1.4 Extreme Programming (XP)

XP is composed of a set of simple, interdependent elements / practices.

1.4.1 Practices & Elements

Element: Customer

- Defines and prioritizes features
- A member of the team and available to the team

Practice: User Story

- Requirements identified in discussion with the customer
- Very concise (succinct) text with an estimate of its relative difficulty
- Almost no details, technical details likely to change anyway

Good User Story:

Template

Long Template: As a <Role>, I want <Goal> so that <Reason>
Short Template: As a <Role>, I want <Goal>

Characteristics

Each story must
- be understandable to the customer
- provide something of value to the customer
- be sized so that several can be implemented per iteration
- be independent
- be testable

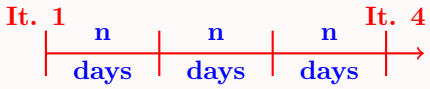
Work according to the **INVEST** principle:
- **I**ndependent:
  - Self-contained, **no inherent dependency** on other stories
- **N**egotiable:
  - User Stories can always be **changed** and **rewritten** (up until a **sprint**)
- **V**alueable:
  - Must deliver **value** to the end user
- **E**stimable:
  - Possible to **estimate** size (implementation effort) of story
- **S**ized appropriately (or **S**mall):
  - Not as big as impossible to plan/prioritize with certainty
- **T**estable:
  - Contain enough information to enable **test design**

Element: Acceptance Test

Details of user stories are captured in the form of **acceptance tests**.
Acceptance tests are written before or concurrently with the implementation of a user story.
Once a test passes, it is added to the set of passing acceptance tests and is never allowed to fail again. (Avoid **regression**)

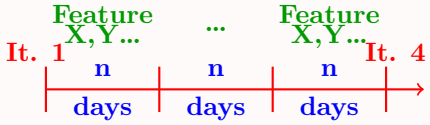
Practice: Short Development Cycles

Sprint / Iteration: A fixed sized time interval in which a set of software features is implemented.
At the end of each sprint is a piece of executable software that can be tested - may or may not be deployed into production.
Time Boxed: Not extended if planned features cannot be implemented in time - In this case, features must be **moved** to a later iteration



Practice: Planning Game

Division of responsibility between business and development
Business people decide the **importance** of a feature,
developers decide how much that feature will **cost** to implement.



Practice: Simplicity

Make the design as **simple** and **expressive** as possible.
Focus on **current** set of user stories - do not worry about **future** ones.

Element: Test-Driven Development (TDD)

Implementation starts with writing tests
Afterwards the actual feature / functionality is implemented
Any code is written to make failing tests (unit) **tests pass**.
Tests are written by:

- **Developers:** (unit tests)
- **Customers:** (functional / acceptance tests)

Practice: Continous Integration

Programmers commit their code and integrate their work **several times per day** - non blocking version of version control
After each commit: System is built and every test (incl. acceptance) is run

Requirements:

- Usage of version control system (git, svn, etc.)
- Automated build system
- Automated test execution

Element: Refactoring

Improve program structure **without changing** existing behaviour.
Refactor **frequently** to avoid code ”rots” due to adding feature after feature.
Before adding a feature refactoring should be considered.

Practice: Pair Programming

Programmers **pair up** to write code:
- One focuses on the **best way** to implement a feature
- The other looks at the ocde being written, but from **strategic** point of view

Pairing changes often to **spread knowledge.**

Requirements:
- Programmers must be at a **comparable skill level**
- Must **subdue proprietary impulses** to ”own” code

Practice: Collective Ownership

Team owns the code - any pair has the right to check out any module.
Everyone takes over responsibility for the whole system - no single person is blamed for problems

Element: Coding Standards

Establish appropriate coding standards:
- Promote **least** amount of work possible
- Respect ”**no duplicate code**” principle
- Emphasize **communication**
- **Accepted** by whole team

1.4.2 Planning

Initial Exploration (Start of the project)

- Developer & customers identify all **significant** user stories (not all user stories)
- Developers estime the stories relative to each other using **story points** (how long it’ll take to implement)
- Actual size determined by **velocity** (= story points of previous iteration, initially just a guess, gains accuracy as iterations progress)

Release Planning

- Developers & Customer agree on date for **first release** (2-4 months)
- Customer **picks** stories and rough **order** (limited by velocity)
- As velocity gains accuracy, **adjust release plan** (number of stories)

Iteration Planning

- Customer picks stories for iteration n (must not exceed velocity of iteration n - 1)
- Within one iteration the **order** is a technical decision
- Iteration ends on specified date (**time-boxed**, even when not all stories finished)
- Compute **velocity** of completed iteration: Sum of estimates of all **successfully finished** stories
- **Planned velocity** for iteration n + 1:= **Measured velocity** of iteration n

Task Planning

Stories broken down into tasks of 4 - 16 hours implementation time
Developers choose tasks freely

1.4.3 Additional Remarks on Processes

Different Types of Systems Need Appropriate Processes

- Software running an aircraft developed using different process than an e-commerce website
- Operating system developed differently than a word processor
- In large systems different parts may be developed using different processes

No Be-All-End-All Process Model

Processes must use the capabilities of the **people** in an organization
Processes must follow the specific **characteristics** of the developed software