Software Process Models

Waterfall model

Requirements definition → System & software design → Implementation & user testing → Integration & system testing → Operation & maintenance

Cons: inflexible to change, errors in previous phase may require workarounds, issues discovered late

Use cases: embedded systems, critical systems, larger software systems

Reuse-based development

Pros: reduces amount to be developed, faster delivery

Cons: requirement compromises, control of evolution lost

Incremental model

Specification ⇔ Initial version
Development ⇔ Intermediate version
Validation ⇔ Final version

Pros: lower cost of accommodating change, rapid delivery

Cons: less measurable progress, degrading system structure

Agile

No requirement specification, as requirements are changed often.

Idea: deliver value in increment, avoid waste, feedback loop, adaptiveness to changing requirements

Waste: hardship in daily work, partially done work, extra processes, extra features, task switching, waiting, motion, defects

Scrum: transparency, inspection, adaptation

Problems: business side slow to embrace, general resistance, mixed systems, siloed systems, scalability

Xtreme Programming

release plan → iteration plan → acceptance test → stand up meeting → pair negotiation → unit test → pair programming → code

DevOps

Combines development and operations.

Measures: change lead time, deployment frequency, change fail rate, failed deployment recovery time

Value stream mapping: Identify and analyze value streams to reduce/eliminate inefficiencies

Requirement Engineering

Complexity

Accidental complexity (things that can be solved, like memory leaks) vs Essential complexity (things that must be done)

Wicked problems

Problems with no definitive formulation, no stopping rule, no true/false, no ultimate test, one-shot operational solutions only, esentially unique, no enumerable set of solutions, symptoms of other problems

Functional and non-functional

Functional: What system should do
Non-Functional: Requirements not directly concerned with services

NFRs

Should be written **quantitatively** to make them measurable.

Quality Attribute Scenarios

	Definition	Example
Source	Source of stimulus	user
Stimulus	Event arriving at system	downloads a new application
Artifact	Where stimulus arrives	existing platform
Environment	Set of circumstances scenario takes place	Runtime
Response	Activity occuring as result of arrival	using it productively
Response measure	How good response is	within 2 minutes of experimentation

Metrics:

Usability: system features learning time, efficiency of using system, amount of user error, adapting system for user needs

Performance: latency, no./% of satisfied requests, no./% of unsatisfied requests, variance

Availability: availability %, time interval when system available, time to detect fault, time to repair fault, time system can be ind egraded mode, proportion of faults prevented/handling without failing

Modifiability: cost in terms of affected artifacts, effort, elapsed time, money, new defects, adaptation time

Deployability: cost of deploying, percentage of failed deployments, repeatability of process, traceability of process, cycle time of process

Energy efficiency: max/avg kW load on system, avg/total amount of energy saved, time period where system must stay powered on

Security: size of compromise, accuracy of attack detection, amt. of vulnerable data

Safety: amt./% of entries into unsafe states that are avoided, of unsafe states that are recoberables, of shut down time

Stakeholders

Main activities: **Elicitation and analysis**, **Specification**, **Validation**

Elicitation & Analysis: Methods

Interviewing: Closed (predefined set of questions) or open (no predefined agenda).
Consider how questions are asked: no leading questions, no questions with bias, no true/false

Document Analysis: examine existing docs
Pros: might reduce time needed for stakeholder interaction, gets exisiting corporate/industry specs/standards
Cons: might be out of date

Questionnaires
Pros: works well for large group, useful for prioritisation
Cons: no follow up, closed-ended qns

Workshops
Pros: more effective for resolving disagreements, helpful when quick turnaround needed
Cons: resource intensive

Focus groups
Pros: useful for commercial products w/o access to end-users

Prototyping Develop simplified model of system

Observation
Pros: discover implicit system reqs.
Cons: cannot identify new features

Personas Archetype of user group to use in meetings

Elicitation & Analysis: Specification

User stories: Used for agile systems.
As a {user}, I want {goal} so that {benefit}.
Card: written description for planning
Conversation: verbal exchange to flesh details
Confirmation: acceptance tests to determine completion

Use cases: Used for plan-driven systems
{Verb} .. {Object}
Can be structured as actors, description, data, stimulus, response, comments.

Elicitation & Analysis: Validation

Informal reviews:
Peer/informal review/formal/inspection

Requirements document checks: validity checks, consistency check, completeness check, realism check, verifiability.

Feasability: system contribution to overall objective, if system can be implemented within schedule & budget using current tech., if system can be integrated

Prototyping: develop version of system to check requirements

Process models

Spiral model:
elicitation (user, system) → specification (user, system) → validation (prototyping, reviewing) done iteratively

Discovery: discover (domain) requirements
Classification & Organisation: group related requirement, consider each stakeholder as a viewpoint
Prioritisation & Negotiation: resolve req. conflict
Specification: document requirements and input into next round of spiral

User	System
natural language	natural language/other models
understandable by users	expanded reqts used by engineers
only external behaviours	complete & detailed spec of system

Requirements Management Planning

Requirement identification: each requirement must be uniquely identified for cross reference in traceability assessment
Change management process: to assess impact/cost of changes **Traceability policies**: define relationships between each requirement and between requirements and system design

Modeling

Developed for **Requirements Engineering** (derive detailed reqts.) and **System Design** (describe system for impl.)

Abstraction: Leaving out details to make systems easier to understand
Representation: Maintains all information of system being represented

Perspectives

External context or environment of system
Interaction interactions between system and environment, or between components
Structural model organisation of system or structure of data processed by system
Behavioural model dynamic behaviour of system and how it responds to events

Languages

C4, UML, BPMN, SysML, ERD

C4

System context: how software system in scope fits into world around it - focuses on people
Container: Software system in scope - refers to spparate runnable apps
Component: Zooms into individual components (Not recommended) - decomposed containers (as major structural blocks)
Code: Zooms into the code itself (not recommended)

UML

Class diagrams (structure, conceptual models, concept analysis of domain, architecture, interfaces)
Sequence diagrams (requirements elicitation, eliciting behaviours, instantiation history)
Activity diagrams (modelling concurrency, eliciting useful behaviours, ordering processes)
State machine diagrams
Use case diagrams (requirements)

System Architecture

Functionality does not impact architecture - important for non-functional requirements mostly.

Modifiability: local, non-local, behavioural change

Conway's Law: organisation which design systems are constrained to produce designs that are copies of the communication structures of these organisations

Architecturally significant requirements

Non functional requirements, core features, system constraints, environmental constraints

Attribute-Driven Design

Step 1 Get ASRs - reqts. which measurable impact on architecture
Step 2 Establish iteration goal
Step 3 Choose existing structure to improve within architecture
Step 4 Select multiple designs that might support ASR
Step 5 Instantiate patterns and tactics to the context
Step 6 Record design decisions
Step 7 Analyse partial design
Step 8 Iterate until satisfied

Architecture Patterns

Layered architecture: Structures the software system into individual grouping of modules that offer a cohesive set of services
Pros: Portability, reusability, modifiability
Cons: Might get in way by not providing all required lower-level abstractions, performance penalties

Pipe-and-filter: Consists of filters (processing components that take input and produce output) and pipes (connect filters together)
Pros: Reconfigurability, evolution, modifiability
Cons: Fixed format (for data transfer), performance (data is output in specific format)

Model-centered architecture: Components interact with central model
Pros: Independence of components, consistent data management
Cons: Single point of failure (central model), difficult to distribute repo

MVC: Model, view, controller
Pros: Highly modifiable, state can be managed and persisted, concurrency
Cons: Upfront significant complexity, burdensome for complex UIs

Plug-in architecture: Base system, with multiple plugin components
Pros: Modifiability, extensibility, testability
Cons: Security concerns

Client-server architecture: Server provides services to clients simultaneously
Pros: Low coupling among server, no coupling among clients, scalable, evolvable

Monolith: Single deployable unit

Service-oriented architecture: Focus on independent services separately deployed
Pros: Deployable, testable, reliable

Microservice architecture: Collection of independently deployable services, communicate only via messages through service interfaces
Pros: Quick time to amrket/deploy, independent, scalable
Cons: Network communication overhead, complex transactions, different tech. with maintenance cost, challenging to design & maintain multiple microservices

Architectural tactics

Decisions that influences a quality attribute.

Availability
Detect faults: Monitoring, ping/echo, heartbeat (periodic message exchange), timestamp, sanity checking (checks validity of specific ops on output), voting, exception detection, self-test
Recover from faults - preparation and repair: redundant space, rollback, exception handling, software upgrade, retry, ignore faulty behaviour, graceful degradation, reconfiguration
Recover from faults - reintroduce system: shadow, state resync, escalate restart (auto-restart at different granularities), nonstop forwarding
Prevent faults: removal from service, transactions (ACID properties (Atomic, Consistent, Isolation, Durability), predictive models, exception prevention, increase competence set

Performance

Control resource demand: manage work reqs, limit event response, prioritise events, reduce computational overhead, bound execution time, increase efficiency
Manage resources: increase resources, concurrency, maintain multiple copies of computations/data, bound queue sizes, schedule resources

Testing

Verification whether software conforms to spec
Validation whether software does what user requires

Test case

Test oracle mechanism to determine test pass/fail
Test input arguments to function, system/env state, seq. of actions, arg passed on CLI, button on GUI

Manual testing manual generation of test input
Automated testing automated generation of test input with test oracle.

Test levels

Unit tests: focus on single unit
Pros: fast, easy to control/write
Cons: not realistic, might not catch integration bugs, mocks needed

Integration tests: focus on units together

System tests: focus on single unit
Pros: realistic
Cons: slow, harder to write, prone to flakiness

Test flakiness Tests that might non-deterministically pass/fail
Reasons: concurrency, async wait, test order dependencies, timeouts, resource leaks

Testing & processes

User acceptance testing: Validation - involves customer, contrasts system testing

TDD Write test → check it fails → write simplest code to pass it → check all tests pass → refactor as needed
Pros Quick feedback, focuses on reqts, testable, pace up to dev.

Black-box, white-box

White	Source-code guided
Black	No internal information guidance

Black-box/specification-based

Understand reqts. → explore program → identify partitions → analyze boundaries → devise test cases → automate test cases → augment

White-box/structural

Coverage criteria
Line coverage $\frac{\text{lines covered}}{\text{total lines}}$
Branch coverage $\frac{\text{branches covered}}{\text{total branches}}$
Conditions + Branch coverage $\frac{\text{br covered + cond covered}}{\text{total br. + total cond.}}$
Paths coverage $\frac{\text{paths covered}}{\text{total paths}}$
MC/DC coverage
$\frac{\text{cond. eval to all poss. outcomes affecting decisions}}{\text{total num. of cond. within decision}}$

Mutation testing

Idea: Evaluate quality of existing tests to derive new tests.

Select statement → apply mutation → execute test suite → proceed depending on outcome → undo change and continue until threshold → return mutation score $\frac{\text{killed mutants}}{\text{mutants}}$
Pros: effective in discovering undertested parts
Cons: computationally expensive, equivalent mutants