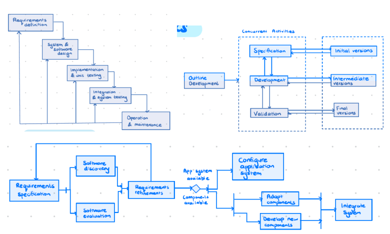
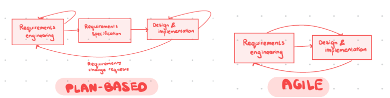
SWE Approaches

3 Process Models



Waterfall inflexible, late issue discovery, errors might need workaround
Incremental lower cost of change, rapid delivery, less measurable, degrading structure
Reuse-Based less software, faster delivery, requirement compromises, less system evolution control

Agile



Lean engineering: **TPS (1988)**
Lean: Focus on creating value and avoiding waste

Key concepts: **Just-In-Time**, **Jidoka** (automation with human touch), **Kaizen** (continuous improvement), 'Andon Cord'

Scrum

Empricism (Knowledge comes from experience) + Lean thinking (reduce waste, focus on essentials)

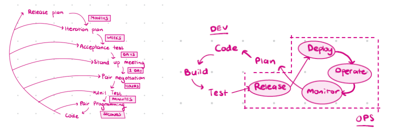
Pillars Transparency, frequent inspection, adaptation

Team Scrum Master, Product Owner (Product Backlog, Goal), Developers (Sprint Backlog)

Kanban

Definition-of-Workflow Shared understanding of flow
WIP Limit on tasks to be worked on at a time

Xtreme Programming &Dev Ops



Xtreme Programming: Collective code ownership, on-site customer, pair programming

DevOps Metrics Change lead time, change fail rate, deployment frequency, failed deployment recovery time
Value Stream Map Reduce ineffeciency by analyse value stream

Requirements Engineering

Difficulty of RE

Wicked problem no stopping rule, one shot operation, unique

Challenges users incomplete understanding, conflicting views, poor understanding of compute capability, analyst knowledge of problem domain, 'obvious' info, jargon, evolution of requirements, ill-defined boundaries, unnecessary design information, untestable vague requirements

Importance

High cost of error during requirement.

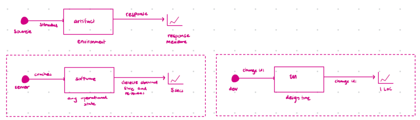
Effect on relations
Customer relations and stakeholders → Communication between stakeholders
Software design → Structure and behaviour
QA → Basis for acceptance
Maintenance, evolution → Dictates quality

Functional vs Non-functional

Functional What system should do decided by stakeholders

Non-functional Not directly concerned with specific services
Defines architecture and tradeoffs between quality attributes

Quality attribute scenario
Source → stimulus → artifact, environment → response → response measure



NFRs:
Usability Task time, # errors, learning time, $\frac{\text{learning time}}{\text{task time}}$, # tasks accomplished, user % satisfaction
Availability % availability (SLA), time where system must be available, time to detect, time to repair, time where system can be in degraded mode, % of prevented faults over time
Performance latency, throughput, % of unsatisfied requests, jitter, usage %
Modifiability cost of effort, #/size/ complexity of artifact, time taken to modify, money, new defects, adaptation time
Deployability cost of deploying, % failed deployments, repeatability, traceability, cycle time
Energy max/avg Kw load, # energy saved, time system must be powered on
Security % compromise, accuracy of attack detection, % vulnerable
Safety %/# entries into unsafe mode, of unsafe states where system can recover, shutdown

Elicitation & Analysis

Interviewing: Closed (predefined set of questions) or open (no predefined agenda).

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

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

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

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

Prototyping Develop simplified model of system (elicitation), develop version of system to check requirements (validation)

Observation discover implicit system reqs. cannot identify new features

Personas Archetype of user group to use in meetings

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.

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

Agile vs Plan-driven

Agile No complete process model
Plan-driven System Requirements Specification (SRS) Document.
User reqts: External behaviour
System reqts: Complete and detailed specifications

Spiral model
Elicitation → Specification → Validation
Business reqts → User reqts → System reqts

Modeling & Architecture

Perspectives
External: context, environment of system
Interaction: interaction between system/env or components
Structural: model organisations of system/structure of data
Behavioural: dynamic behaviour of systems, and response

Key modelling languages
Use block diagrams
UML Various levels, widely known, rich tool support Structural: class, Behavioural: sequence, activity, state diagram, External: use case
C4 Context, Container, Component, Code. Notation independent

Attribute Specific Requirements Requirements with measurable impact on architecture

Attribute-driven design
Ensure ASRs → establish iteration goal → choose existing structure to improve → select multiple design satisfying ASRs → instantiate patterns/tactics to context → record design decisions → analyse partial design to see if iteration goal is addressed

Conway's Law Organisations which design systems are **constrained to produce designs** which are copies of the communication structures of these organisations

Architectural Patterns

Layered portability, reusability, modifiability, performance penalty
Pipe & filter modifiability, reconfigurability, evolution, fixed format, performance (parse into format)
Model-centered independent, consistent, single point-of-failure, hard to distribute
MVC highly modifiable, can be persisted and managed, concurrent, significant upfront complexity, coupling
Microkernel modifiability, extensibility, testability, security vulns, privacy threats
Client-server low coupling, scalable, evolvable
Monolith single deployable unit
Service-oriented deployability, testability, reliability
Microservices collection of independently deployable services

Architectural Tactics

Decisions that influence 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

Pesticide paradox Running the same tests repeatedly can become less effective at finding new bugs

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.

Unit tests: fast, least prone to flakiness, least complex, least realistic
Integration tests: multiple components
System tests: most realistic, slow, most prone to flakiness, most complex

Test flakiness Tests that might non-deterministically pass/fail due to 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
Quick feedback, focuses on reqts, testable, pace up to dev.

Black-box/specification-based

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

Combine tests: test exceptional cases only once, not combine them & Test valid input at least once in a positive test case

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 Evaluate quality of existing tests to derive new tests. effective in discovering undertested parts, computationally expensive, equivalent mutants

Select statement → apply mutation → execute test suite → proceed depending on outcome → undo change and continue until threshold → return mutation score $\frac{\text{killed mutants}}{\text{mutants}}$

Debugging

Debugging is done during dev and test phases (development) or after user reports (commit, integration, prod)

Terminology
Mistake Human act/decision resulting in error
Defect Error in program code
Fault Location in which error triggered
Failure Fault is externally visible

Rubberducking Explain problem to someone else

Debugging types

printf simple, intuitive, language and tool agnostic, can be confusing, unclear, left in code, require recompilation of code
Logging more systematic than printf Multiple levels (Debug, Info, Warn, Error, Fatal)
Debugger systematic, display additional info, breakpoints
Scientific Formulate question → hypothesis → prediction → test (Repeat)

Tracking origins First state where origins are all good but there is a fault
Program slicing Identify subset of program that could have influenced variable/returned value (Dynamic slicing: removes part of code not executed) (Backward vs forward slicing: What influenced a value (backward) vs What statements are influenced by the value (forward)

Dependencies Data, control

Statistical fault localisation Utilise multiple execution for localising faults
Tarantula

$$Suspiciousness(line) = \frac{\frac{failed}{\#failed}}{\frac{failed}{\#failed} + \frac{passed}{\#passed}}$$

Small-scope hypothesis A high proportion of errors can be found by testing a program for all test inputs within some small scope
Test case reduction Specify oracle in executable way. Binary search cannot be used as a bug might be triggered by two but not by the two individually
Delta debugging Check subset (divide by 2), if both are ok, check complement, if both are good, increase granularity to max(2n, total subset size), repeat

Regression bugs Feature that worked before stopped working
Use binary search (git bisect: exit 0 good, exit 1 bad, exit 125, undetermined)

SWE Dev. Practices

Source Control Centralised vs decentralised, dependency management, scalability, dev tool support, expertise of team
Single source of truth, consistency, small, focused changes, clarity, appropriate tags & branches, toolset to manage dependencies

Code styles Consistency over optimality, ensure code is clear and unsurprising, practicality over pretty/clever, use autoformatting

Software versioning
Semantic Major.Minor.Patch - major for incompatible API changes, minor for backward compatible functionality add, patch for backward compatible hotfixes
Hashing In commits. automatic, no extra information, not every commit is a release
Assigned identifiers Code names, version increments. easier to remember, branding, inferred ordering

Hyrum's Law With a sufficient number of users of an API, it does not matter what you promise in contract, all observable behaviour of your system will be depended on by somebody

Dependency hell Two components use different versions of same component.
Dependency management Use package manager. **Small/critical** Keep in same library **Medium** Keep own repo **Large** submodule **Runtime** Lockfile & containers

Deprecation Code as a liability. Difficult because new system is different, emotional attachment, difficult to fund, migration (to new system) is expensive
Advisory deprecation warning prevent new uses, rarely leads to migrations, can overwhelm users
Compulsory deadline for removal
During design planned decommisioning

Code review feedback on code design, teaching/demoing coding methods and design, awareness of state of product, improve maintainability
Good practices
Author keep reviews small, write good desc. and annotations, choose right reviewers, beware of inertia, think carefully
Reviewers push back on overly broad/poorly described changes, attack code, not person, consider not just normal case, be responsive

Static Analysis

Examination of code without executing it. early bug detection, improve code quality, reduce debugging time, enhance maintainability

Types linters, style checkers, data-flow and control-flow analysis, type checkers

Lambda Calculus

$(Terms) M, N := x | \lambda x.M | M.N$
$(Types) \tau, \sigma := T | \sigma \to \tau$

$\alpha - equivalence : \lambda x.M = \lambda y.M[y/x]$ where y fresh
$\beta - reduction : (\lambda x.M)N \to M[n/x]$

Reduction rules
$\beta - reduction$
$\frac{M \to M'}{M\,N \to M'\,N}$
$\frac{N \to N'}{M\,N \to M\,N'}$
$\frac{M \to M'}{\lambda x.M \to \lambda x.M'}$

Free and bound variables
Bound (placeholder, can be renamed), free (cannot be renamed)

Reduction strategies
Normal-order Choose left most outermost redex first, always finds normal form
Applicative-order Choose left most innermost redex first

Language constructs
$True \stackrel{def}{=} \lambda x.\lambda y.x$
$False \stackrel{def}{=} \lambda x.\lambda y.y$
$not \stackrel{def}{=} \lambda b.b\ False\ True$
$or \stackrel{def}{=} \lambda b.\lambda b'.b\ True; b'$
$and \stackrel{def}{=} \lambda b.\lambda b'.b\ b'; False$
$b?M : N \stackrel{def}{=} b\ M\ N$
Church numerals $n \stackrel{def}{=} \lambda f.\lambda x.f^n x$
$succ \stackrel{def}{=} \lambda n.\lambda f.\lambda x.f(nfx)$
$succ' \stackrel{def}{=} \lambda n.\lambda f.\lambda x.nf(nfx)$
Tuple $(M_1, ..., M_n) \stackrel{def}{=} \lambda f.f M_1 ... M_n$
$\pi_i \stackrel{def}{=} \lambda p.p(\lambda x_1 ... \lambda x_n.x_i)$
Product and Sum Types Product types must have both types (chosen by user), while sum types has either or (chosen by value)
Soundness Never accepts program that can go wrong (no false negatives)
Completeness Never rejects program that can't go wrong (no false positives) **SLTC** Sound (reduction of well-typed term diverges or terminates in a value of the expected type) - Preservation and progress. Not complete

Advanced Software Testing

Automated testing for complex systems, components to find bugs more creativity needed to realise tests

Approaches

Property-based (Quickcheck, jkwik) Specify properties for tested component and let framework find counterexample to break
Differential (Black-box) Use different SUTs to test if output is the same simple, effective small overlap in functionality, non-deterministic, multiple systems can be wrong, difficult to generate valid input
Metamorphic Use source test-case to generate follow up Avoid differential weaknesses by using 1 system, simple and effective Difficult to find bug-revealing relation
EMI Compile P(I), mutate unexecuted of P into P', verify P'(I)
Fuzzing
Black-box send random input simple, efficient unlikely to uncover deep bugs if input format complex (use grammar based fuzzing)
Mutation-based mutate existing inputs Higher quality input Based on seed
White-box (SAGE) leverage knowledge of program to fuzz diverse test inputs heavyweight & costly

Symbolic execution

Execute program on symbolic input to represent all input (KLEE).

Path condition A set of conditions that make a path in the execution.

SAT Solver Takes in propositional formula and outputs a model that satisfies the formula
SMT Solver Solves formula based on a given theory (for non-Boolean formula)

If path cannot be reached, code is redundant or code will never be executed (bug).