“To err is human, but to really foul things up you need a computer.” – Paul Ehrlich

- **Software system** = programs + config files + system documentation + user documentation
- **Software product:** Software system for specific user base, made for some target market

Intro

Why did it start?

- 1960s "software crisis" -> hardware was moving fast, software wasn't keeping up -> difficulty in writing correct, understandable and verifiable programs
- Many projects failed due to deadline, not meeting requirements, being incorrect/inefficient, being difficult to maintain
- Even now only 1 in 3 software projects succeed

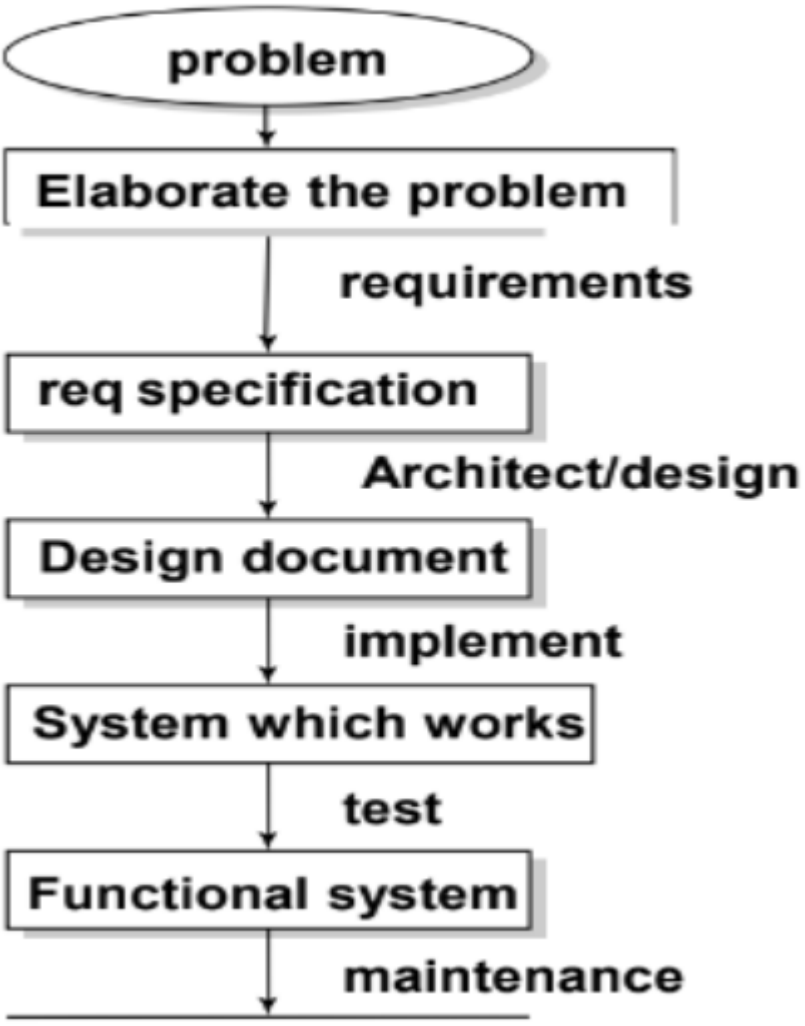
SDLC

Software Process

Any process that contributes to the building of the software product itself. Required characteristics:

- Entry criteria
- Tasks and deliverables
- Exit criteria
- Who is responsible
- Constraints
- Dependencies

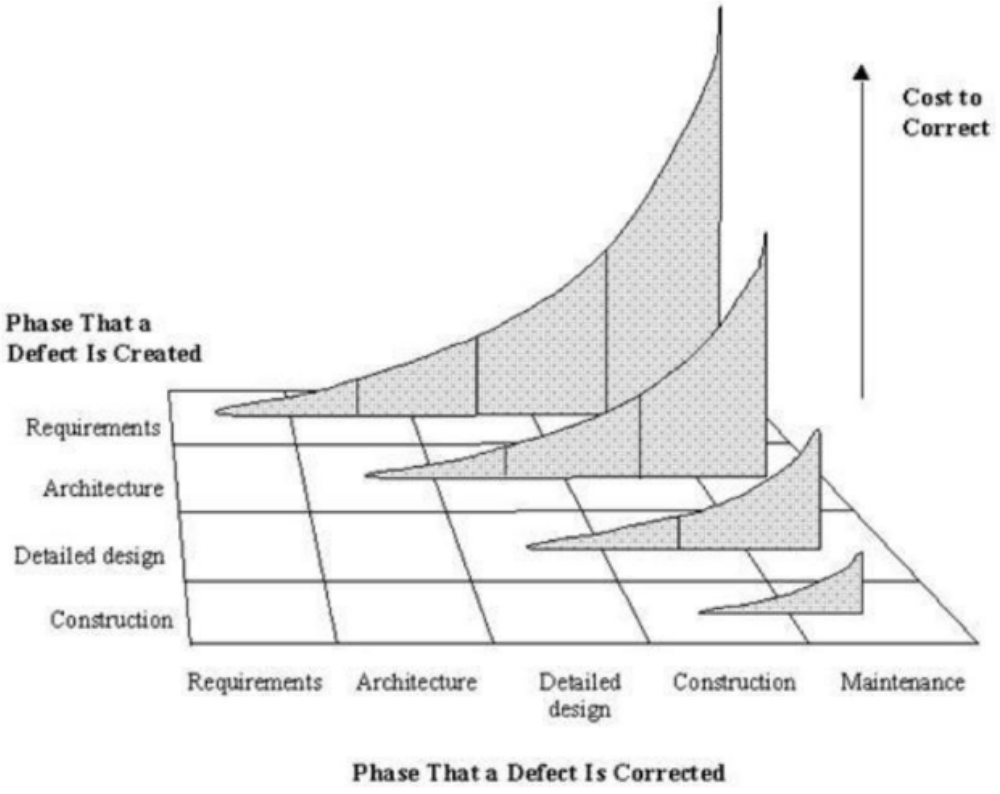
Basic SDLC



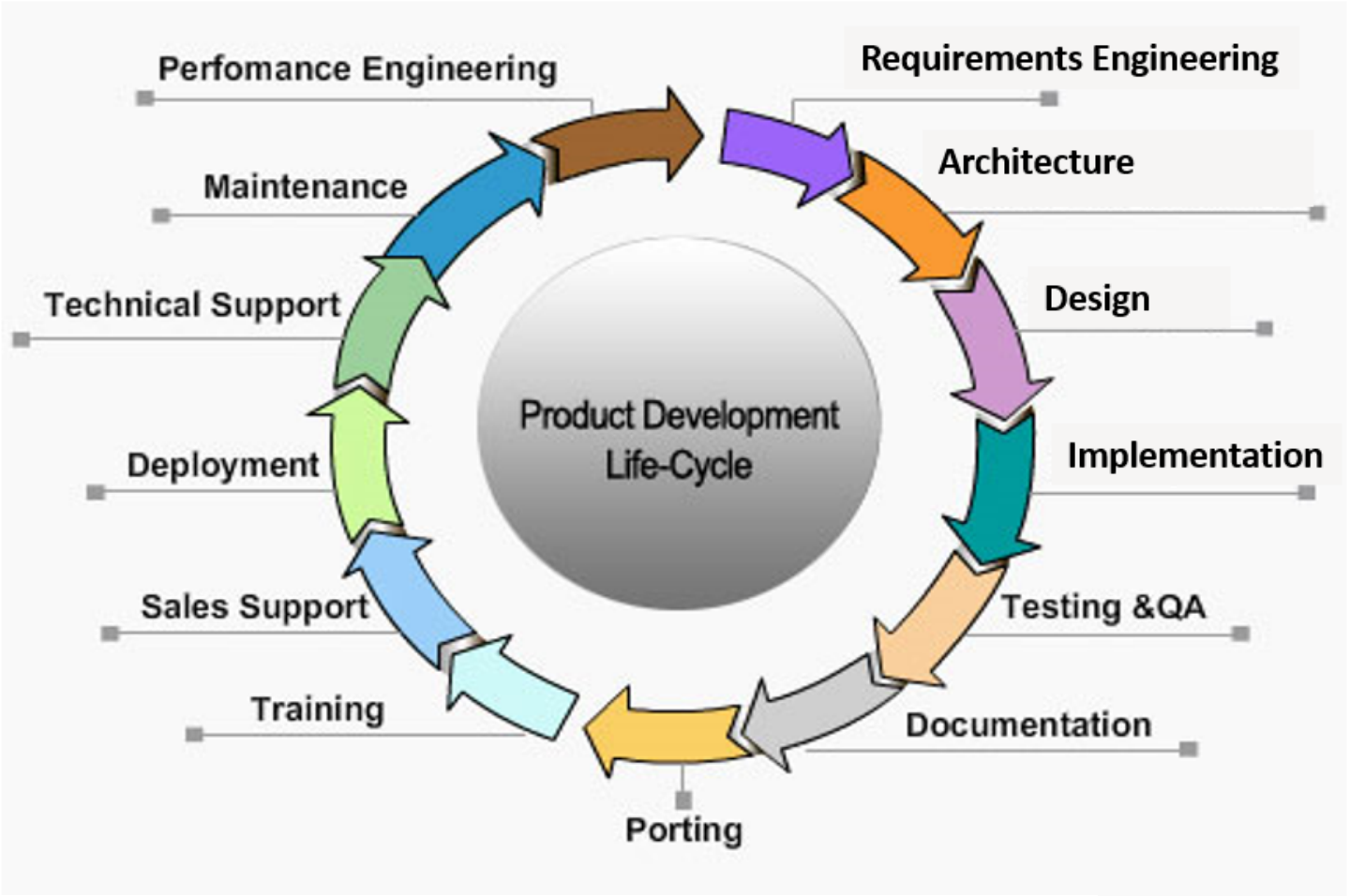
Requirements -> Design/Architect -> Implementation -> Testing -> Maintenance

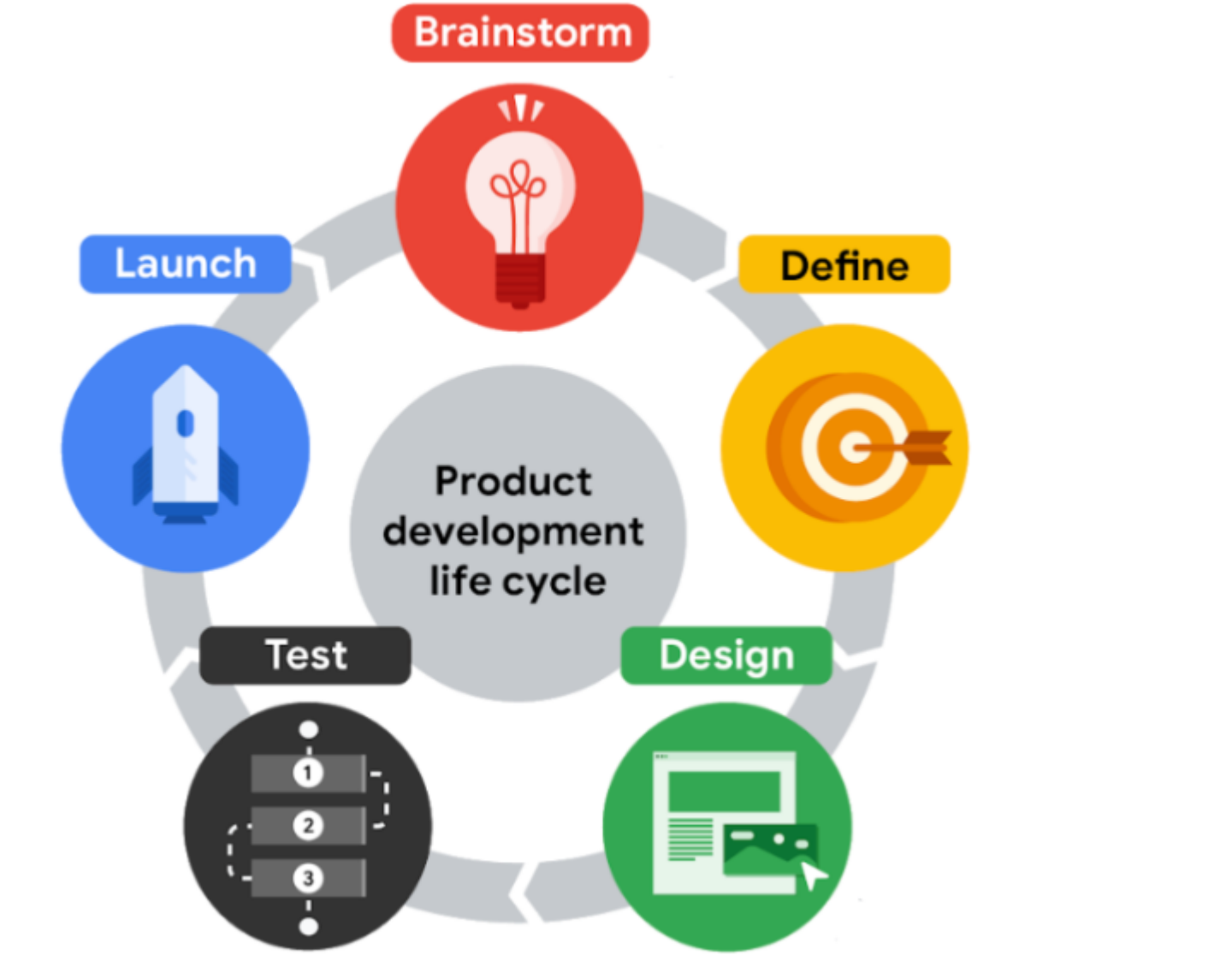
- Requirements: high level what
- Architecture (high level design): mid level what, high level how
- Design (low level design): low level what, mid level how
- Code: low level how

Cost of Defects



Product DLC



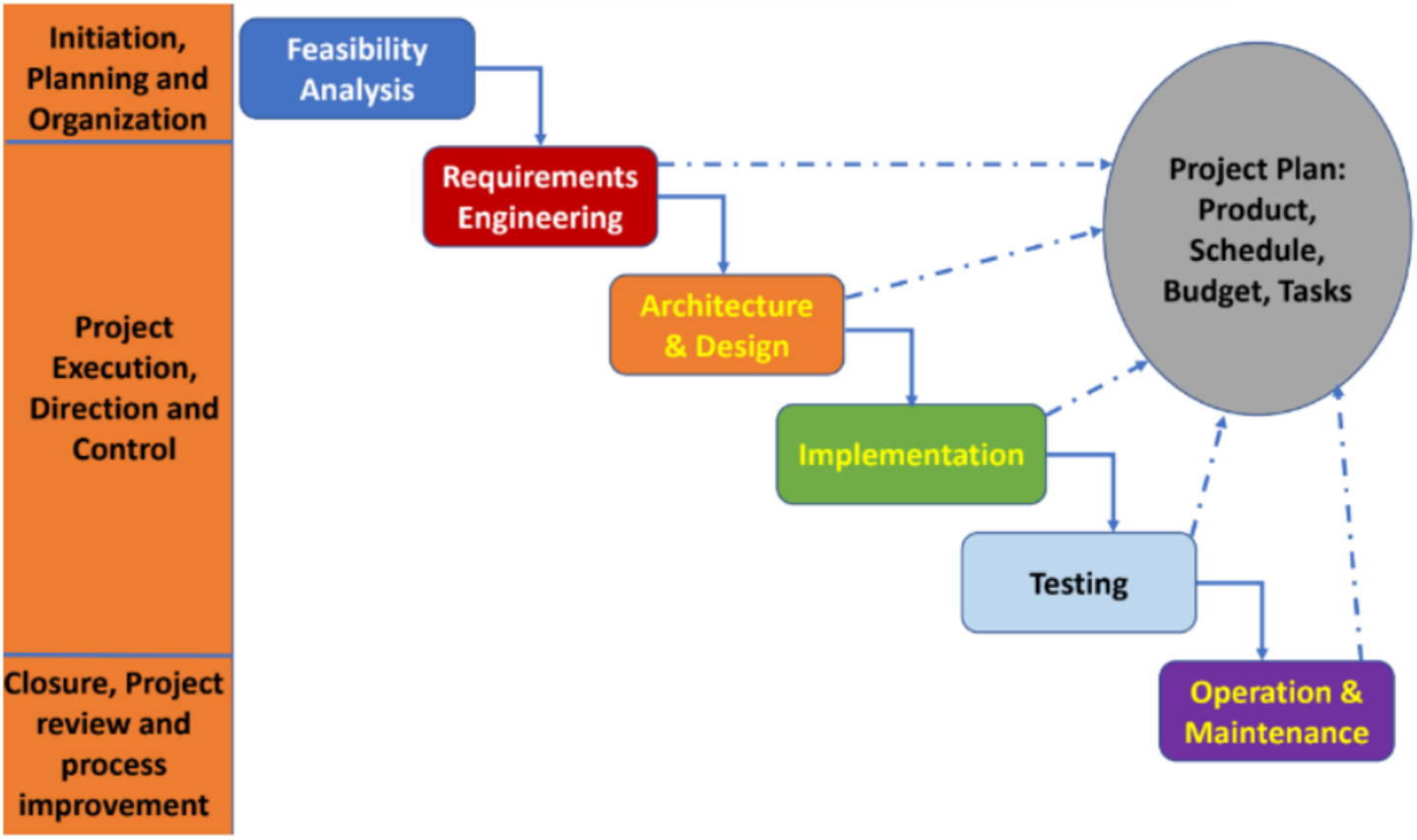


Brainstorm: Identify problems customers could be facing to solve, competitor analysis

Define: Figure out the product specifications (who is it for, what needs to be there for it to succeed etc.)

Design: Wireframes -> prototypes. Translate specification into design
Test: Write and test code.
Internal (stakeholders) -> alpha (internal) -> beta (external)
Launch: Share with all customers. Review what went right, what went wrong. Iterate if needed.

Project Management Lifecycle

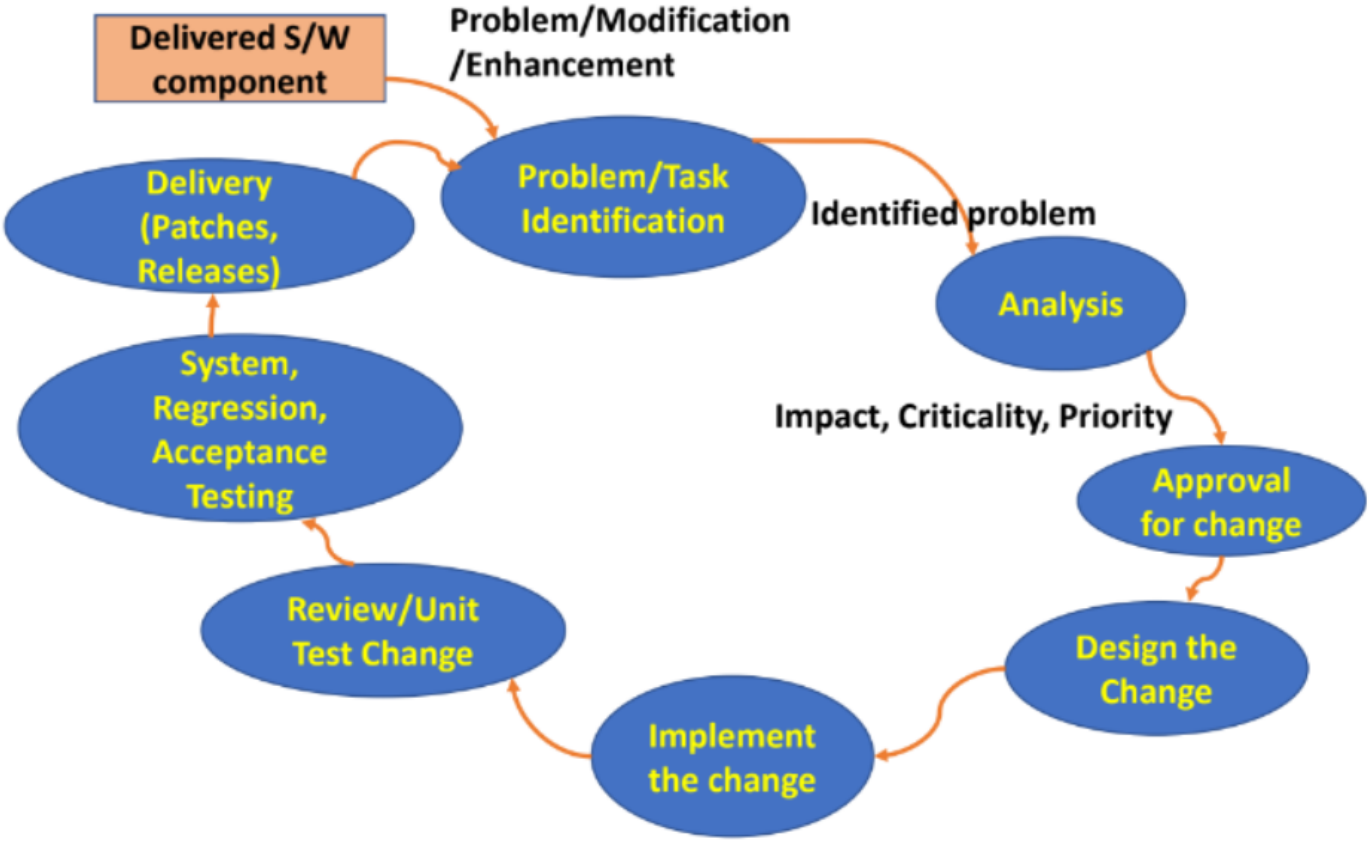


Initiation -> Planning -> Controlling execution, direction -> closure and review

Software Maintenance Lifecycle

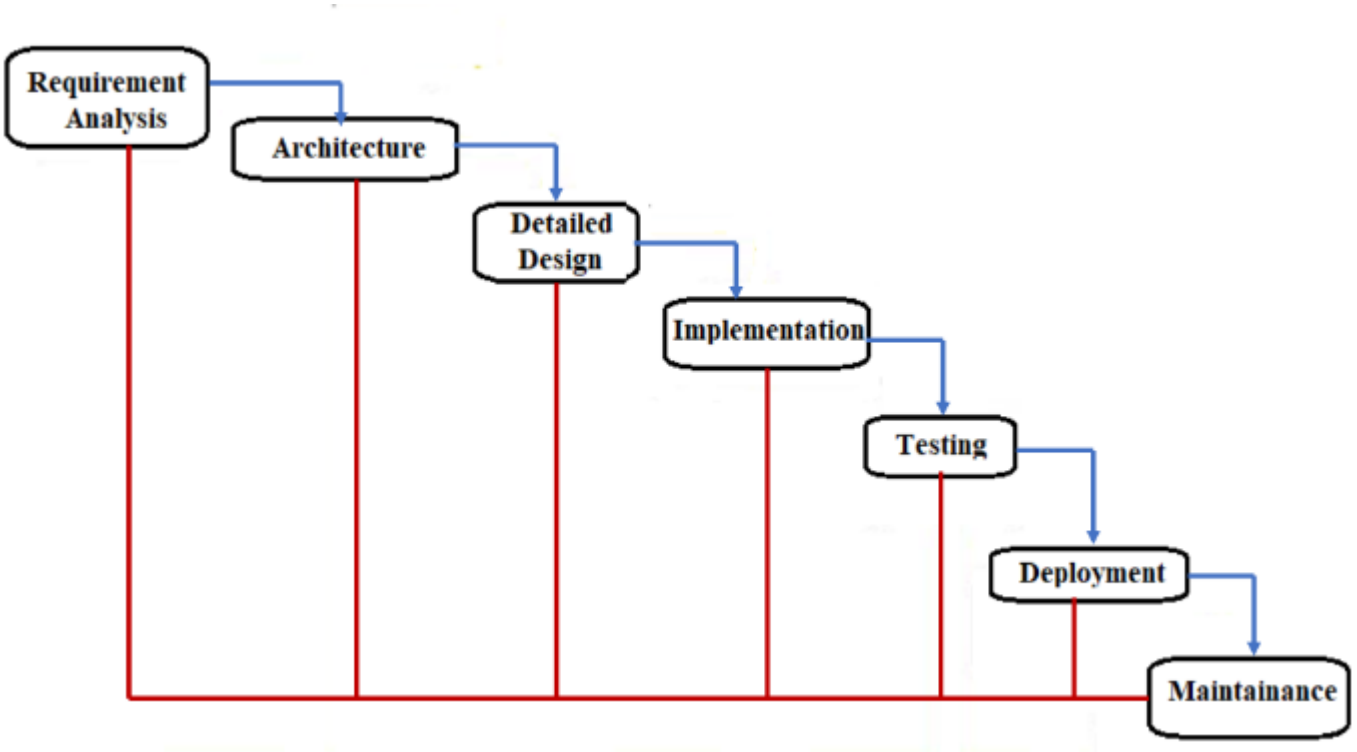
Issue raised -> Problem identification -> Risk/Impact analysis -> Request for change -> Approval for change -> Design change -> Implementation of change -> Testing -> System regression/acceptance

testing -> Deliver change



Waterfall Model

- Sequential model, every phase frozen and signed off on before moving to the next
- Most steps result in a final doc
- **When to Use:** Small projects with a clearly defined set of requirements; tech being used is understood well



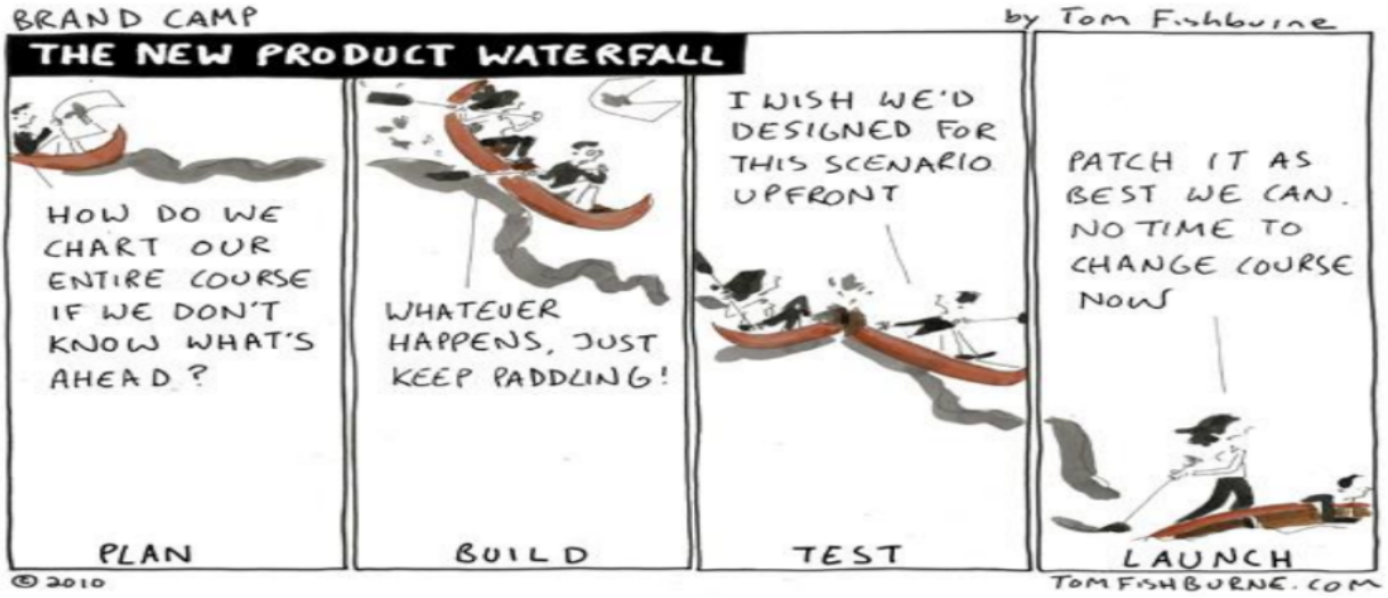
Advantages

- Simple, clearly identified phases

- Easy to manage
- Each phase -> specific deliverables, reviews
- Easy to departmentalise

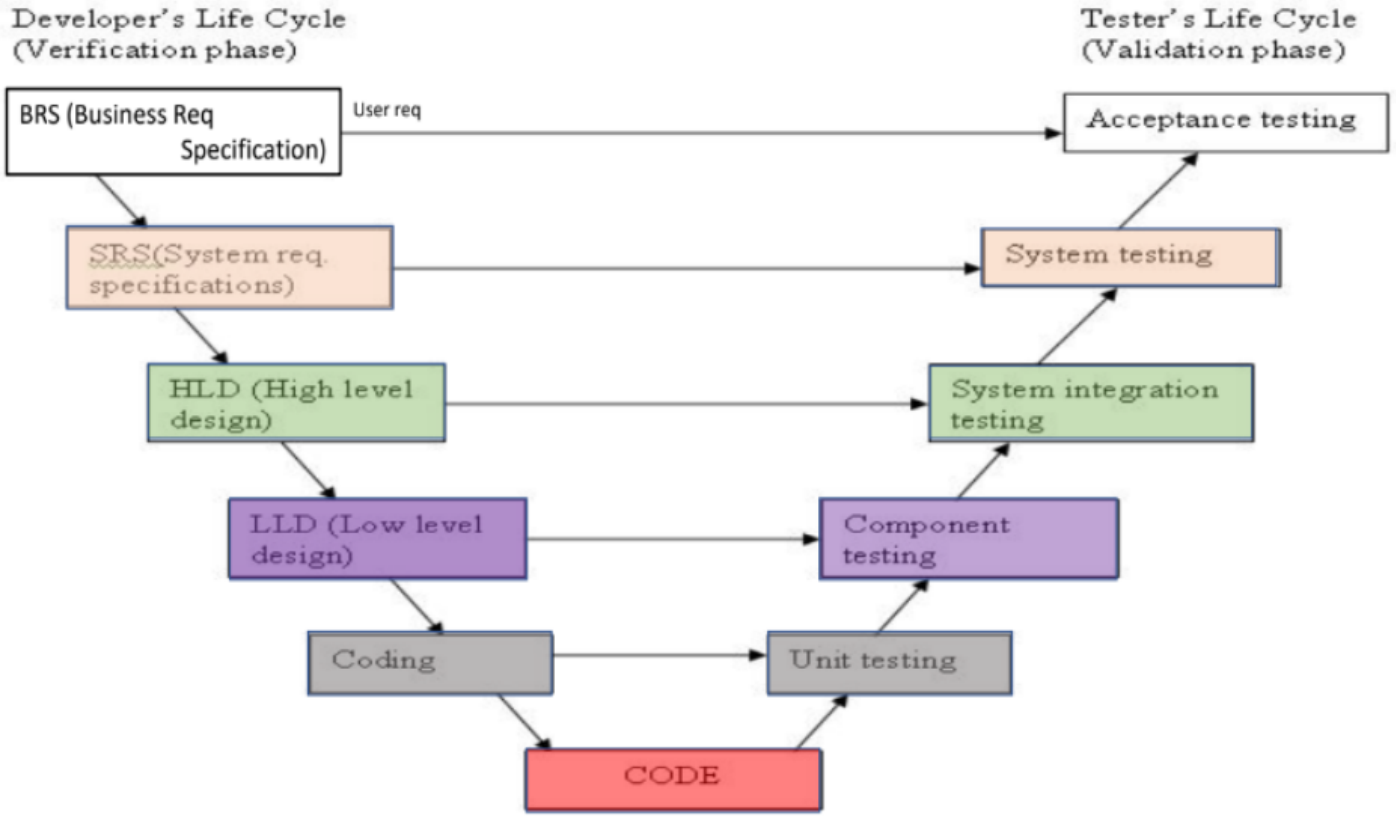
Disadvantages

- Requirements are frozen
- Difficult to change once started
- Poor model for long projects
- No prototypes of software available until testing, high risk



The V Model

- Similar to waterfall but integrates testing activities into every phase
- Leads to higher probability of success and efficient use of resources



Advantages

- Higher probability of success
- No need to wait until formal testing cycle for testing activities
- Similar to waterfall model stuff

Disadvantages

- Similar to waterfall model stuff (no prototype, rigid structure etc)
- Test documentation also has to change in case of any change in the process

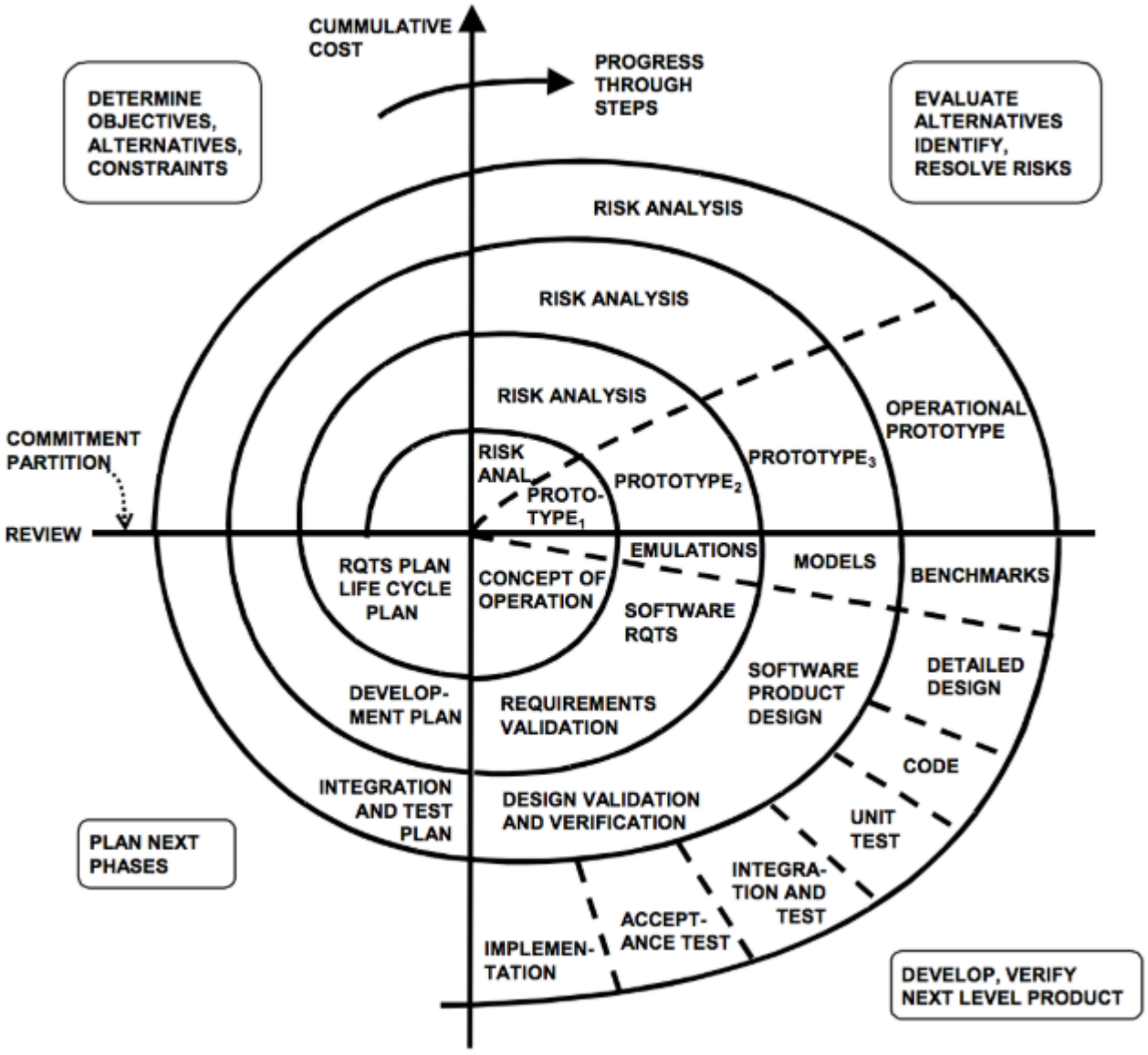
Iterative vs Incremental Models

Iterative	**Incremental**
Revisit and refine everything	Deliver and done, no need to go back
Focus on details of components	Focus on components not delivered yet
Leverage past learnings	Does not leverage past learnings

Spiral Model

- Iterative + Incremental: combines the waterfall model with prototyping
- Each loop represents a phase of the development process
- Every iteration is called a spiral, and contains these:
  - Determine objectives
  - Risk analysis
  - Develop and test
  - Plan next delivery

- Phased reduction of risk: address high risk early to minimise cost



Advantages

- Risk reduction as cost increases
- Prototyping integration
- Early indication of issues/pivots
- Allows for changes

Disadvantages

- Requires much more planning to execute correctly
- More complex to manage
- Requires proper risk assessment

SecDLC - Security Development Lifecycle

Maps security activities to every phase of the SDLC

Pre-SDL

- Basic security training given to all employees

- Topics like threat modelling, secure coding, data privacy

Requirements

- Security Risk Analysis (SRA) document to define project security requirements
- Which parts of the project will require in depth risk analysis and security design reviews?
- Which parts of the project will require external pen-testing?

Design

- Describe how to implement all functionalities in a secure manner
- Reduce attack surface area
- Increase defense depth (multiple layers of defense)
- Risk analysis of components that have significant security risk

Implementation

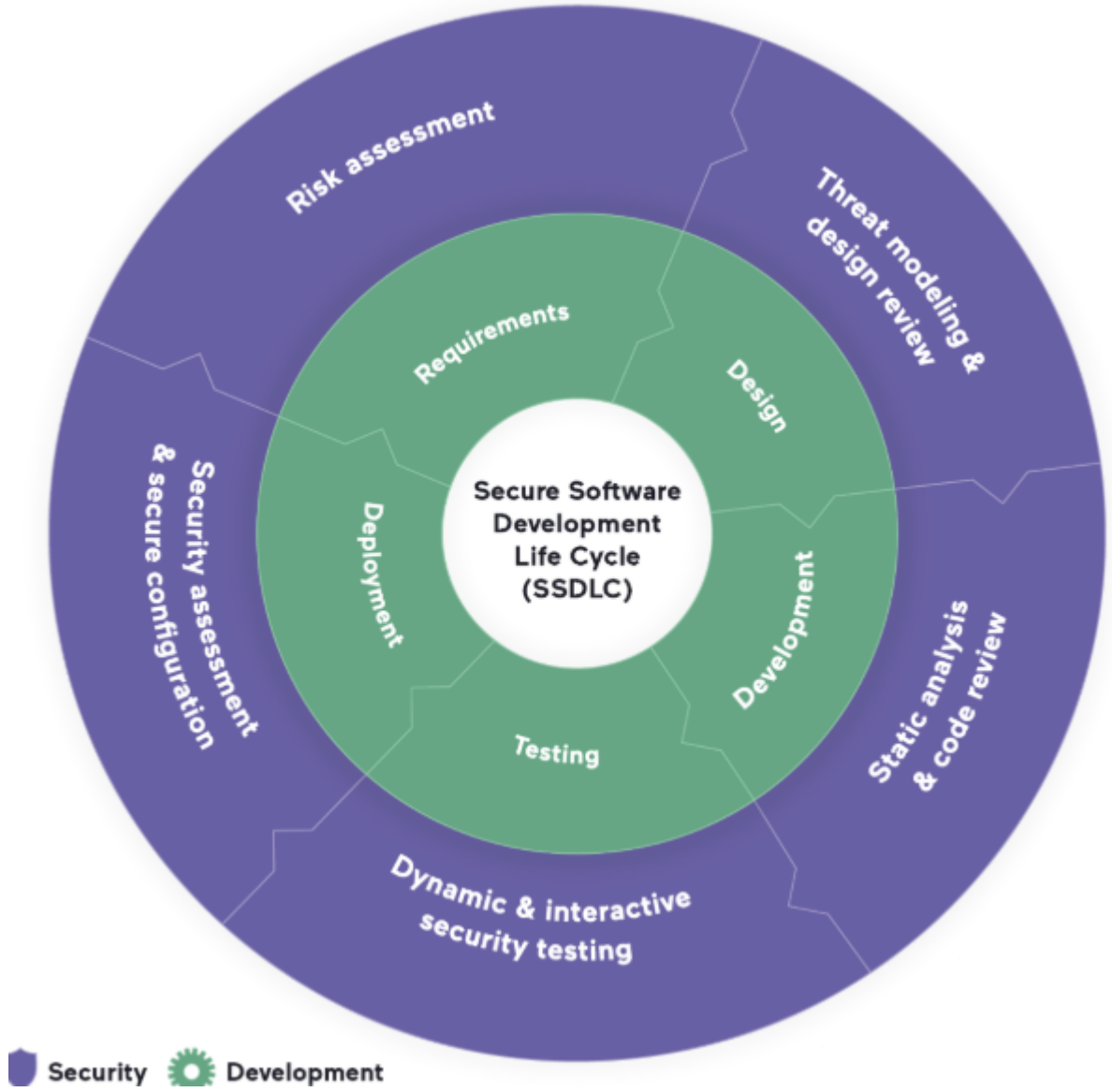
- Publish a list of approved tools and associated security checks
- Analyse APIs and other external services that can be used for the software -> prohibit those with security concerns, including open source ones
- Scan all code for any usage of prohibited APIs/services
- Static security analysis: scan for vulnerabilities without executing code

Testing/Validation

- Dynamic program analysis
- External pen-testing
- Fuzz testing: deliberately introduce malformed inputs during dynamic analysis
- Update threat analysis accordingly and account for any implementation changes

Release

- PoC for any security emergency
- Contacts with decision making authority 24/7
- Security servicing plans for code from:
  - Other teams in the organisation
  - Third parties
- **Final Security Review (FSR):** examination of threat analysis, tool output, performance against quality gates



Agile

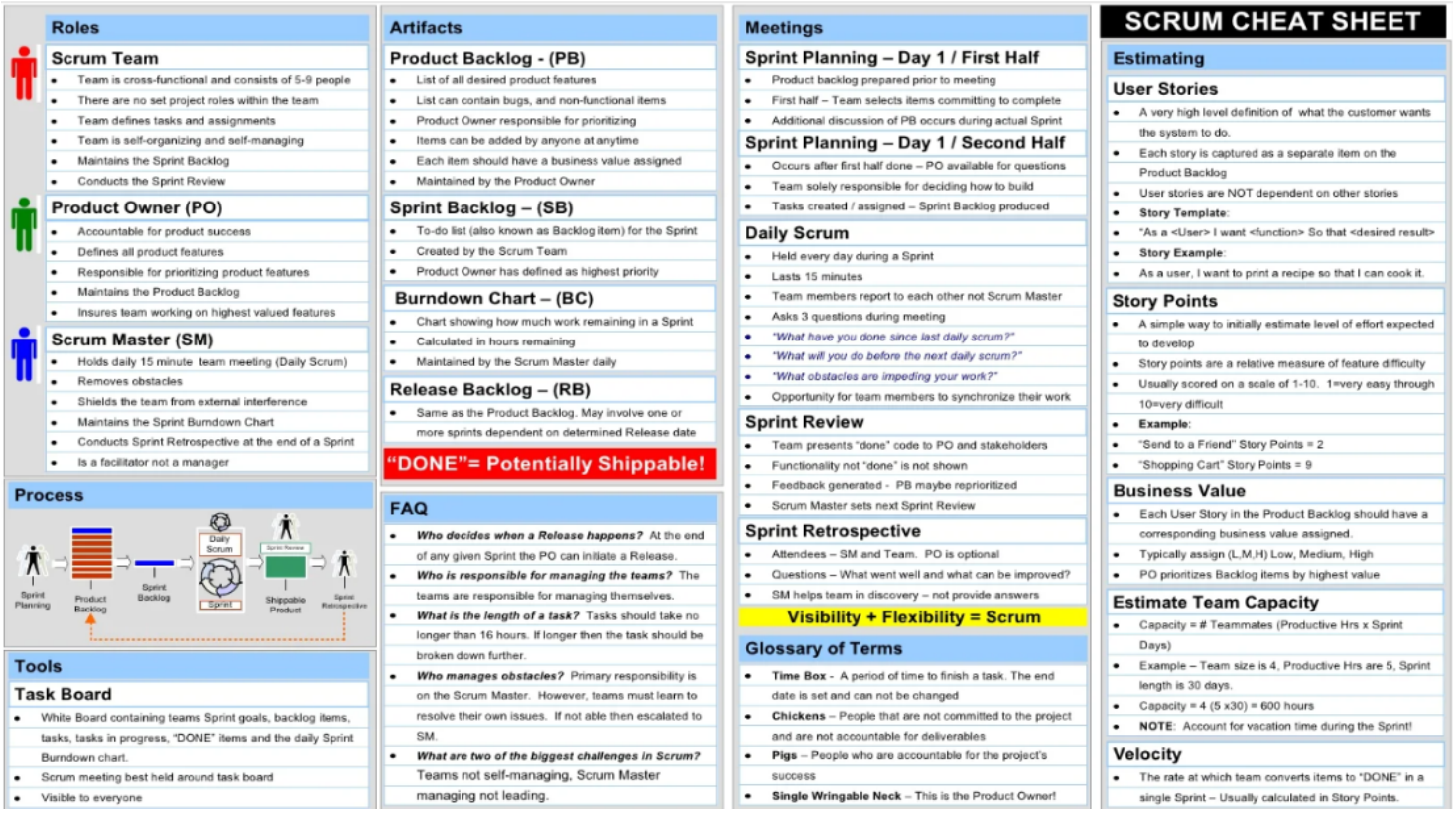
- Focuses on rapid development centred around the customer's requirements
- Open to making changes in between
- Functional code over comprehensive documentation; working software primary measure of success
- Incremental software delivery
- Short iterations -> quickly identify risk
- Frequent testing and integration
- Customer closely involved throughout development
- Focus on maintaining simplicity; actively remove complexity wherever possible
- **Cons:** High individual dependency, no comprehensive documentation -> makes it harder for newer team members to join in, not suitable for handling complex dependencies

Methodology	Type	Key Focus	Best For	Highlights
Agile (Manifesto)	Framework / Philosophy	Flexibility, collaboration, iteration	All types of projects	Umbrella term for adaptive, incremental development
Scrum	Agile Method	Iterative sprints, roles, ceremonies	Product development, teams <10	Fixed-length sprints, roles (PO, Scrum Master)
Kanban	Agile Method	Visual workflow, continuous delivery	Ops, maintenance, support	WIP limits, visual boards, no fixed roles
RUP	Process Framework	Structured, phase-based, iterative	Large, complex enterprise systems	4 Phases: Inception → Transition; use cases, architecture focus
DSDM	Agile Framework	Business needs, timeboxing, governance	Projects needing predictability + agility	MoSCoW prioritization, strong user involvement
XP (Extreme Programming)	Agile Method	Engineering practices, code quality	High-change, high-quality code projects	TDD, Pair Programming, Continuous Integration
Crystal	Agile Method Family	Team size, criticality-based tailoring	Small to medium teams with varying needs	Lightweight, adaptable per project (e.g., Crystal Clear)
FDD (Feature-Driven Dev)	Agile Model	Features and modeling	Large, structured teams	Domain modeling, short iterations, feature-centric

RUP = Rational Unified Framework

DSDM = Dynamic System Development Method

Scrum



Sprint Planning

- 30-60 min, biweekly
- Create SB from PB

Daily Scrum

- 15 min, daily
- Questions:
  - What did I do today?

- What will I be doing tomorrow?
- Am I facing any obstacles in completing my work?

Sprint Review

- With PO
- 30-60 min biweekly
- Present results of sprints, features developed

Sprint Retrospective

- 30-60 min biweekly
- Look back on sprint and identify what went well, what could have gone better

User Stories

Good story -> follows INVEST

Independent

- Story should not be dependent on others
- Allows dev team to reorder the implementation as they see fit

Negotiable

- Should not be like a contract; it is a starting point for back and forth between the devs and the stakeholders
- Eg: "As a user, I want to receive notifications" -> allows dev team to interpret the technical details of said notifications

Valuable

- Should convey a genuine need of the user
- Allows dev team to deliver tangible value with every feature

Estimable

- Dev team should be able to estimate amount of effort needed to implement feature
- Should not be vague
- Max estimate for a task is 16 hours; if beyond that, it should be broken down further

Small

- Should be concise enough to complete in a single iteration
- Easier to estimate, test and deliver

Testable

- Defined success criteria that can be used to validate task

Extreme Programming (XP)

Planning Game

- Level 1: Release planning
  - Decide on features required in production and when
  - Decide on priorities for different features
- Level 2: Iteration planning (kinda like sprint)
  - Pick most valuable items from list, break down into tasks
  - Estimate effort
  - Commit to delivering at end of iteration

Simple Design

- No complex architecture or design; start simple and let it build over iterations
- Easier to make functional changes
- **Spike:** PoC/test to understand validity of hypothesis. Outcome of spike lets the team know if they can proceed with a particular dev direction or not

Test Driven Development (TDD)

- Write unit test cases first, and then write minimal code to satisfy said test cases
- Refactor code while still making sure it passes the unit cases

Code Standard

- Certain practices to be followed while writing the code
- Makes it simple to understand and debug, helps with pair programming, consistency etc

Refactoring

- Modify/restructure code to make it simpler and align with standards
- Improve overall code quality, prevent repetitions/redundancies etc

Pair Programming

- Two programmers share one device, one screen when working on a feature
- Two roles:
  - Pilot: focuses on clean code, compiles and runs. Low level focus
  - Navigator: keeps the big picture in mind and reviews the same code for refactoring. High level focus
- Roles are switched, pairs are switched

Collective code ownership

- Everyone works on everything, no blame game
- Team is responsible

Continuous Integration

- In addition to local test cases, tests are run overall on the entire project when new changes are integrated
- If tests fail, they are fixed then and there

Small Release

- Cross functional team releases MVP regularly
- Helps break down complex models into smaller chunks of code

System Metaphor

- Stories must be simple enough to understand and map to a feature
- Naming convention for tasks from stories for understandability

Onsite Customer

- Expert in domain, know how to generate ROI on an MVP
- Sit with the rest of the team to ensure comms flow freely

Sustainable Pace

- Down time during iteration: fix bugs, refactor code or do research to maintain the pace

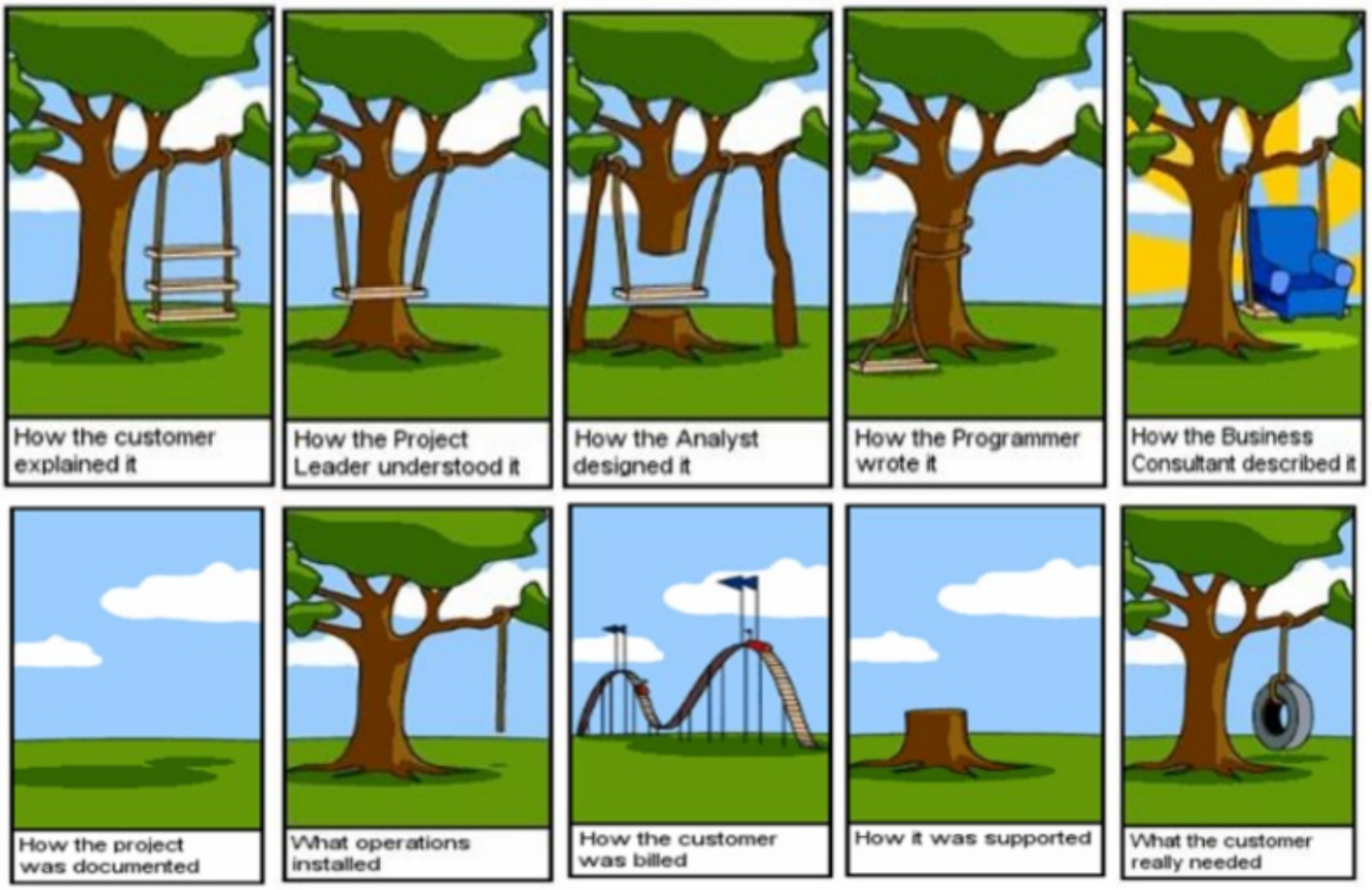
Lean Agile

Focus: Maximise value, minimise waste
Waste = anything that does not add value from a customer's perspective

Principles

- Eliminate Waste – Remove anything that doesn’t add value.
- Amplify Learning – Encourage continuous learning and improvement.
- Decide as Late as Possible – Keep options open until the best choice is clear.
- Deliver as Fast as Possible – Provide value quickly to get feedback sooner.
- Empower the Team – Give teams autonomy and trust to make decisions.
- Build Integrity In – Ensure quality is part of the process, not an afterthought.
- Optimize the Whole – Focus on improving the entire system, not just parts of it.

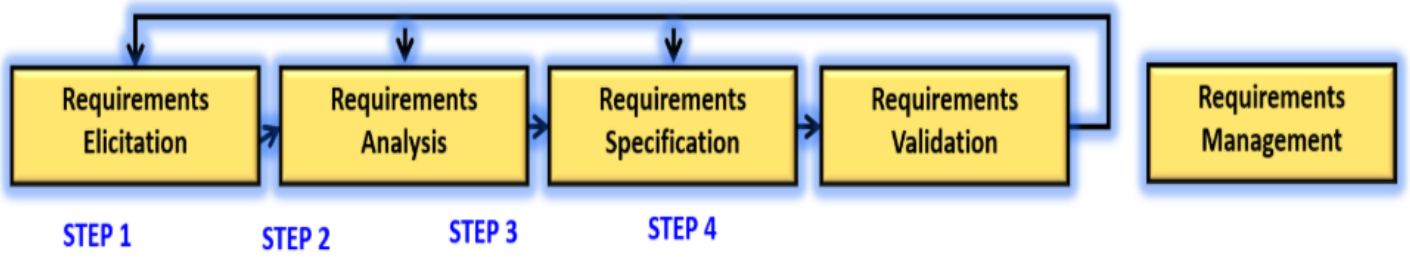
Requirements Engineering



- Process of eliciting, analysing, documenting and maintaining requirements -> requirements engineering
- Focus on completeness and consistency:
  - Complete: Should have descriptions of all facilities required
  - Consistency: No contradictions in descriptions of system facilities

Step 0: Feasibility Study

- Short, low-cost study to analyse whether the project should actually be done or not
- Approval from the feasibility study's report is required to move on to actual requirement engineering



Step 1: Elicitation

- Talk to stakeholders to determine what requirements are
- Establish clear scope and objective for project
- Two types of techniques:
  - Active: Ongoing interaction with stakeholders. Eg: Interviews, prototype testing, meetings etc.
  - Passive: Infrequent interaction. Eg: Questionnaires, documentation etc.

Step 2: Analysis

- Classify requirements:
  - Functional
  - Non-functional
  - User
  - System
  - Domain
- Prioritise: MoSCoW -> Must have, Should have, Can have, Won't have
- Decide whether to build or buy (COTS solution)

Functional Requirements

- How system should react in different scenarios
- Actual functionality of project

Non-Functional Requirements

- System properties and constraints
- May influence implementation and functional constraints since they apply to system as a whole
- May generate functional requirements or restrict existing requirements
- Classification:
  - Product
  - Organisational
  - External

Step 2.5: Security Requirements

- SRA - Security Requirements Analysis -> Based on results of initial analysis
- Identify attack paths and potential attackers
- Categorise risk:
  - Low: high cost of exploit, low damage, low probability
  - Medium: medium everything
  - High: low cost of exploit, high damage, high probability
- Convert each high priority risk into a measurable objective -> non-functional requirement

Step 3: Specification

- SRS: Software Requirement Specification doc
- Specifies functional, non-functional and design constraints; also specifies external interfaces, scope, etc.

Step 4: Validation

- Validation: If requirements implemented, will they meet users' needs?
- Verification: Have the requirements been specified correctly?

- Prototyping:
  - Helps to ensure both stakeholders and engineers are on same page
  - Helps mainly with systems with user interactions

Step 5: Management

- Requirement Traceability Matrix (RTM): **captures the complete "life story" of a requirement**, providing a clear audit trail from its origin through to its final testing and deployment.

Req ID	Architectural Section	Design Section	File/Implementation	Unit Test ID	Functional Test ID	System Test ID	Acceptance Test ID

- Change management: Uncontrolled requirement changes can have huge impacts