Software Engineering
Unit - I

Compiled by
M S Anand

Department of Computer Science

Software Engineering

Introduction

Text Book(s):

1. “Software Engineering: Principles and Practice", Hans van Vliet, Wiley India, 3rd Edition, 2010.
2. “Software Testing – Principles and Practices”, Srinivasan Desikan and Gopalaswamy Ramesh, Pearson, 2006.

Reference Book(s):

1. “Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling” by By Jennifer Davis, Ryn Daniels, O’ Reilly Publications, 2018
2. “Software Engineering: A Practitioner’s Approach”, Roger S Pressman, McGraw Hill, 6th Edition 2005
3. “Software Engineering”, International Computer Science Series, Ian Somerville, Pearson Education, 9th Edition, 2009.
4. “Foundations of Software Testing ", Aditya Mathur, Pearson, 2008
5. “Software Testing, A Craftsman’s Approach ", Paul C. Jorgensen, Auerbach, 2008.
6. IEEE SWEBOK, PMBOK, BABOK and Other Sources from Internet.

Software Engineering
Evaluation guidelines

5 ISAs - Best 4 to be considered for a total of 30 marks (7.5 * 4)

Assignments – 20 Marks

ESA – 100 Marks (scaled down to 50 marks)

After every unit, a case study will be shared with the students to be deliberated in the class and evaluated.

Under weekly evaluations, students work on a project in a **team of four** implementing all principles and concepts of software engineering.

All the details are available in this document.

Software Engineering

Setting the context

MRI Scanning machine

Expectations

Produces detailed images of the inside of the body.
Uses string magnetic fields and radio waves.
The scanner is controlled by a computer kept inside another room

Software Engineering

Setting the context ….

An ATM

Expectations

Electronic banking outlet.

Today's ATMs are technological marvels, many capable of accepting deposits as well as several other banking services.

Software Engineering

Setting the context …

Uber

Expectations

Allows one to book cabs/autos etc.

Booking can be for an immediate ride or much later.

You can do this from the comforts of your home.

Software Engineering

Setting the context ….

A missile launcher

Expectations

Can launch a missile.

Target can be anywhere within the defined range

Can be ballistic missiles (ICBM).

Moving as well as stationary targets

What is the most obvious driving force behind all these systems?

SOFTWARE (Lots of lines of code)

A Boeing 787 has **6.5 million** lines behind its avionics and online support systems. This is much more than the sum of all mechanical parts including nuts and bolts.

Google Chrome (browser) runs on 6.7 million lines of code (upper estimate).

Facebook runs on 62 million lines of code (minus backend).

Impact of a 90 minute outage at Amazon?
$2.8 Million revenue loss and lots of customers lost

Software Engineering

Introduction …..

Any software system should have all the functionalities up and running with minimum downtime.

To build a software of high quality, we need to have the following:

1. Interaction with customers and stakeholders on what exactly is needed
2. A clear understanding of the end users (who are they and how they are going to use the system)
3. Experts in multiple domains
4. Good planning
5. Team work
6. Ability to scale and support

Hence, need for lots of engineers to work in teams and hence teamwork is most important.

Looking at the sizes of commonly used software packages (OS, Facebook, MS Office, etc), engineers have a lot of work to do.

Software Engineering

Terminology

Commonly used terms

Software - A comprehensive definition

Software is not just the programs but also associated documentation and configuration data that is needed to make these programs operate correctly, serving a computational purpose.

What is a software system?

A software system usually consists of a number of separate programs, configuration files, which are used to set up these programs, system documentation which describes the structure of the system, and, user documentation, which explains how to use the system and web sites for users to download recent product information.

Software Engineering

Terminology…

Software product

A software product is software that has been developed and maintained for the benefit of a user base and often to satisfy a need in the market.

Categories (Types) of software

System software and Application Software

Generic software and Custom software

System software

System software is software designed to provide a platform for other software. Examples of system software include operating systems (OS) like macOS, Linux, Android,etc. ,game engines, search engines, industrial automation, and software as a service applications.

Software Engineering

Terminology…

Application Software

Application software is software that allows users to do user-oriented tasks such as create text documents, play or develop games, create presentations, listen to music, draw pictures or browse the web.

Generic software

Stand-alone systems that are developed by a development organization and sold on the open market to any customer who is able to buy them. This can meet many clients' general requirements.

Custom software

Custom software is a bespoke design developed to meet one client's specific needs, based on the budget and requirements predefined by them.

Software Engineering

Terminology …

Engineering

Engineering is the application of science and mathematics to solve problems. While scientists come up with inventions, it is engineers who apply these discoveries to the real world. Engineers innovate.

What is software engineering?

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software.

One more definition

It is also defined as a systematic approach to the analysis, design, assessment, implementation, testing, maintenance and reengineering of software, that is, the application of engineering to software.

Software Engineering

Terminology …

Software Engineering principle drives usage of appropriate tools and techniques depending on the problem to be solved, while considering the constraints and resources available.

Focuses more on techniques for developing and maintaining software that is correct from its inception.

Software Engineering

Introduction ….

When did the term “Software Engineering” first appear?
At the 1968 NATO Software Engineering Conference

Why?
It was meant to provoke thought regarding the perceived "**software crisis**" at the time.

What was Software Crisis?
The term was used to describe the impact of rapid increases in computer power and the complexity of the problems that could be tackled.

What does this mean?
In essence, it refers to the difficulty of writing correct, understandable, and verifiable computer programs.
The roots of the software crisis are **complexity**, **expectations**, and **change**.

Introduction ….

The causes of the software crisis were linked to the overall complexity of hardware and the software development process.

The crisis manifested itself in several ways:

- Projects running over-budget.
- Projects running over-time.
- Software was very inefficient.
- Software was of low quality.
- Software often did not meet requirements.
- Projects were unmanageable and code difficult to maintain.
- Software was never delivered.

An interesting note on state of software projects

Percentage of projects that fail

The statistics for software project delivery show that **only one in three software projects are truly successful**.

According to Standish Group's report, 66% of technology projects (based on the analysis of 50,000 projects globally) end in partial or total failure.

Just one in three software projects are considered truly successful with large projects most at risk of failure

When can a Software Project be defined as a Failure?

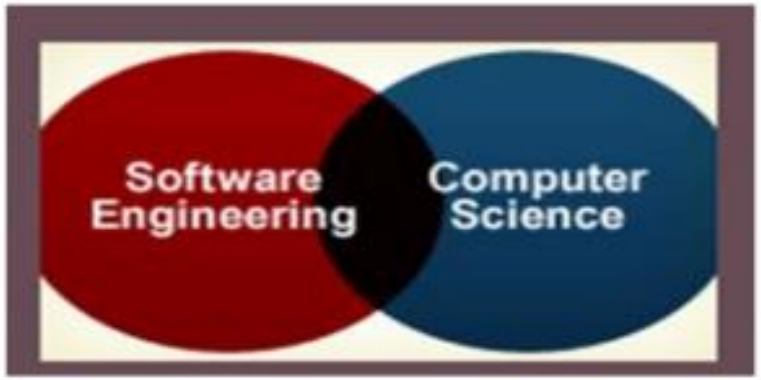
1. The software did not satisfy the requirements of the customer.
2. The software release was later than scheduled (deadline violation).
3. The software had too many bugs.

Software Engineering

Computer Science v/s Software Engineering

Is Computer Science same as Software Engineering?

NO



Computer Science is the study of computation, automation and information. It spans the theoretical disciplines (such as algorithms, theory of computation, information theory and automation) to practical disciplines (including the design and implementation of hardware and software).

Software engineering typically deals with the engineering principles of building, designing and testing software products.

Fundamental drivers of Software Engineering

What are the key factors which drive Software Engineering?

Industrial strength software

Software should

- be operational,
- be maintainable,
- be capable of being moved,
- have elaborate documentation
- have no or minimal number of bugs
- be impactful to the business

Software is expensive

- (Good) Software engineers are at a premium and hence expensive
- Maintenance and rework cost money.

Life-critical and mission-critical software

- Can influence the life or death of a person or a mission.
- Can make or break organizations.

Fundamental drivers of Software Engineering …

Heterogeneity

- Systems should work as distributed systems.

Security and trust

- Software should be trustworthy.
- It should be secure.

Diversity

- Different types of software systems.

Scale

- Software should be scalable.

Business and social changes

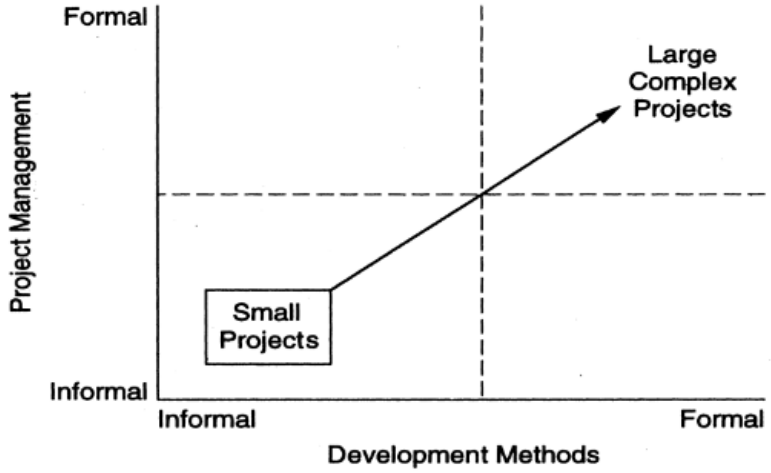
- Ability to change existing software and to develop new software
- The organizations are becoming global.

Fundamental drivers of Software Engineering …

Quality and productivity

Quality measured through **FLURPS** (Functionality, Localizability, Usability, Reliability, Performance and Security) , Portability, Efficiency/Maintainability.

Consistency and repeatability



Late and unreliable.

Typically 35% of the computer based projects are runaway

Software Engineering

Case Study - ARIANE Flight 501

Ariane 5 is a European heavy-lift space launch vehicle developed and operated by Arianespace for the European Space Agency (ESA).
Watch the launch here.

Software Engineering

Case Study - ARIANE Flight 501 ….

•On June 4th, 1996, the very first Ariane 5 rocket ignited its engines and began speeding away from the coast of French Guiana.
•37 seconds later, the rocket **flipped 90 degrees** in the wrong direction, and less than two seconds later, aerodynamic forces ripped the boosters apart from the main stage at a height of 4km.
•This caused the self-destruct mechanism to trigger, and the spacecraft was consumed in a gigantic fireball of liquid hydrogen.

The disastrous launch cost approximately $370m, led to a public inquiry, and through the destruction of the rocket's payload, delayed scientific research into workings of the Earth's magnetosphere for almost 4 years.

The Ariane 5 launch is widely acknowledged as one of the most expensive software failures in history.

Software Engineering

Case Study - ARIANE Flight 501 ….

What went wrong?

The fault was - a software bug in the rocket's Inertial Reference System. The rocket used this system to determine whether it was pointing up or down, which is formally known as the horizontal bias, or informally as a **BH value**. This value was represented by a 64-bit floating variable, which was perfectly adequate.

Problems began to occur when the software attempted to stuff this 64-bit variable, which can represent billions of potential values, into a 16-bit integer, which can only represent 65,535 potential values. For the first few seconds of flight, the rocket's acceleration was low, so the conversion between these two values was successful. However, as the rocket's velocity increased, the 64-bit variable exceeded 65k, and became too large to fit in a 16-bit variable. It was at this point that the processor encountered an operand error, and populated the BH variable with a diagnostic value.

Software Engineering

Case Study - ARIANE Flight 501 ….

At T+37 the BH variable contained a diagnostic value from the processor, intended for debugging purposes only. This was mistakenly interpreted as actual flight data, and caused the engines to immediately over-correct by thrusting in the wrong direction, resulting in the destruction of the rocket seconds later.

It worked on the last device

Several factors make this failure particularly galling.

- Firstly, the BH value wasn't even required after launch, and had simply been left in the codebase from the rocket's predecessor, the Ariane 4, which did require this value for post-launch alignment.
- Secondly, code which would have caught and handled these conversion errors had been disabled for the BH value, due to performance constraints on the Ariane 4 hardware which did not apply to Ariane 5.

Case Study - ARIANE Flight 501 ….

- A final contributing factor was a change in user requirements - specifically in the rocket's flight plan. The Ariane 5 launched with a much steeper trajectory than the Ariane 4, which resulted in greater vertical velocity. As the rocket sped to space faster, there was a higher certainty that the BH value would encounter the conversion error.

Case Study - ARIANE Flight 501 ….

Possible explanations:

1. Inadequate testing
2. Wrong design philosophy
3. Improper reuse

Summary – Why Software Engineering is required

1. Developing large programs
2. Mastering complexity of big programs
3. Efficient development of evolving software
4. Ensuring software process supports users effectively and right choices and decisions are made
5. Supporting large teams and team work
6. Ensuring visibility and continuity

Software Engineering

House construction Life Cycle – Different activities

1. Identify the need, look out for a plot, arrange for financials and buy the plot
2. Prepare a blueprint
3. Get the required permissions from the relevant authorities
4. Site preparation (cleaning, etc)
5. Laying the foundation
6. Building the superstructure
7. Building the roof
8. EXTERIOR AND INTERIOR FITTINGS TASKS WHEN BUILDING A HOUSE
9. FIRST FIX STAGE OF BUILDING A HOUSE
10. DRAINAGE AND EXTERNAL WORKS
11. PREPARING CEILINGS/DRY LINING
12. SECOND FIX CARPENTRY
13. SECOND FIX STAGE ELECTRICS AND PLUMBING
14. DECORATING
15. LANDSCAPING

Software Engineering
Software Project Lifecycle

Software process
A software process is a structured set of activities and associated outcomes (intermediate and final), that produces a software product.

Each activity is defined in terms of:

Entry criteria: What conditions must be satisfied for initiating this phase

Task and its deliverable: What should to be done in this phase

Exit criteria: When can this phase be considered done successful

Who: Who is responsible

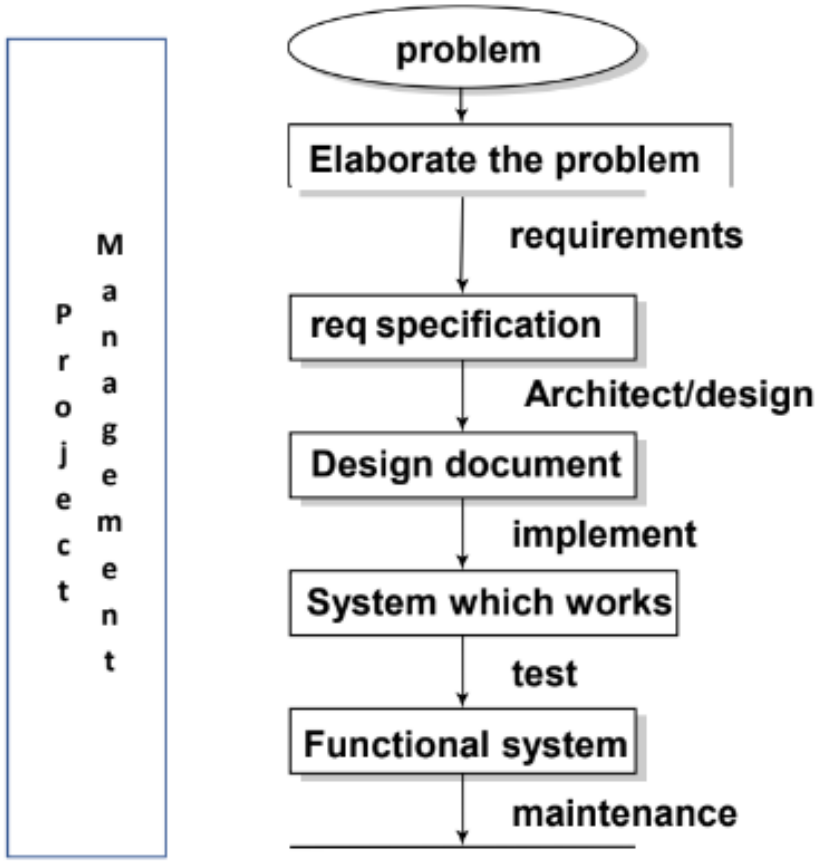
Dependencies: What are the dependencies for this phase ..etc.

Constraints: Time schedule

Software Engineering

Software Project Lifecycle …

Note:
Each step can be a process itself.



Software Project Lifecycle …

There are a few other important activities:

Configuration Management

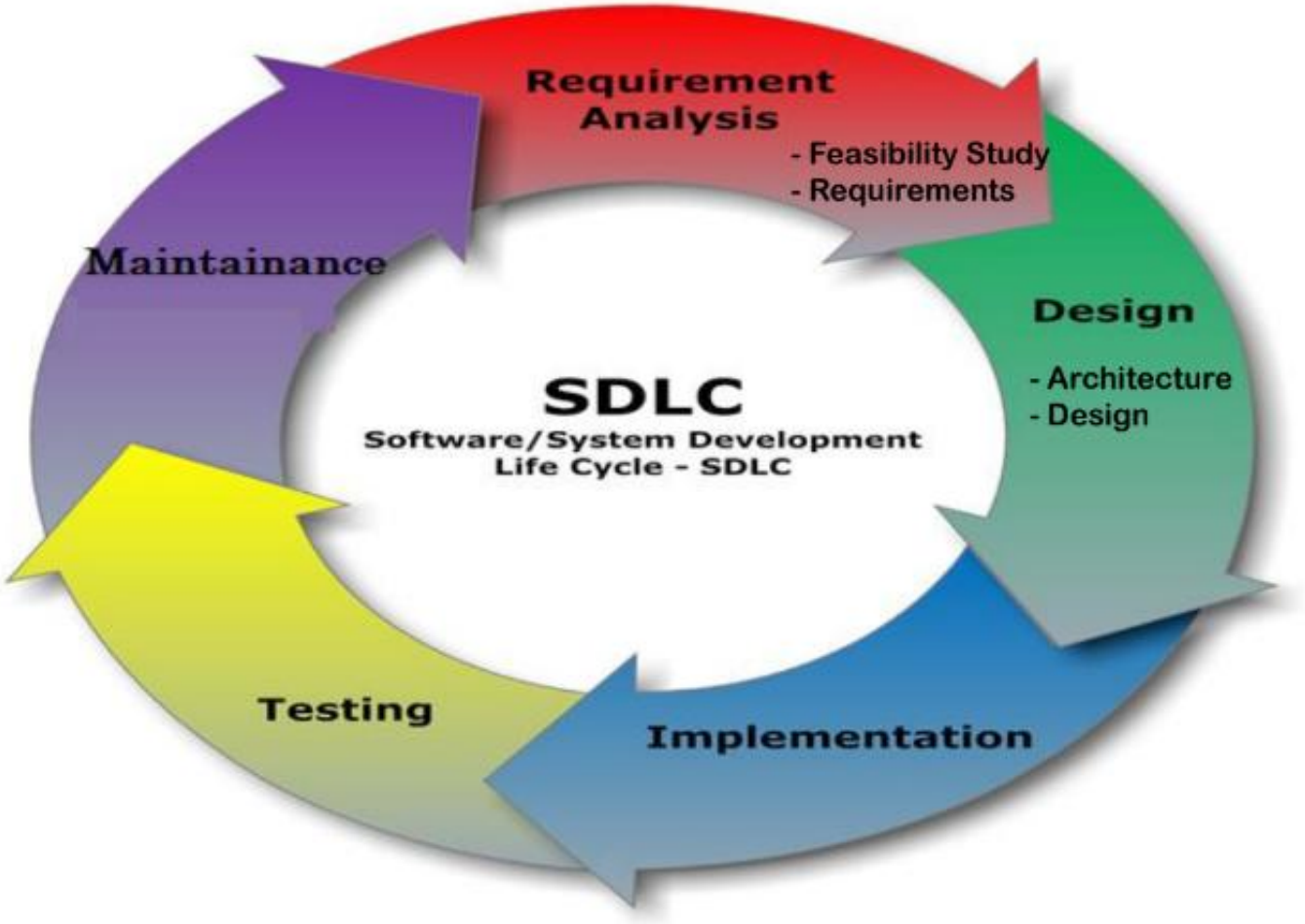
Change management

.
.
.

These form part of the Software Development process.

Software Engineering

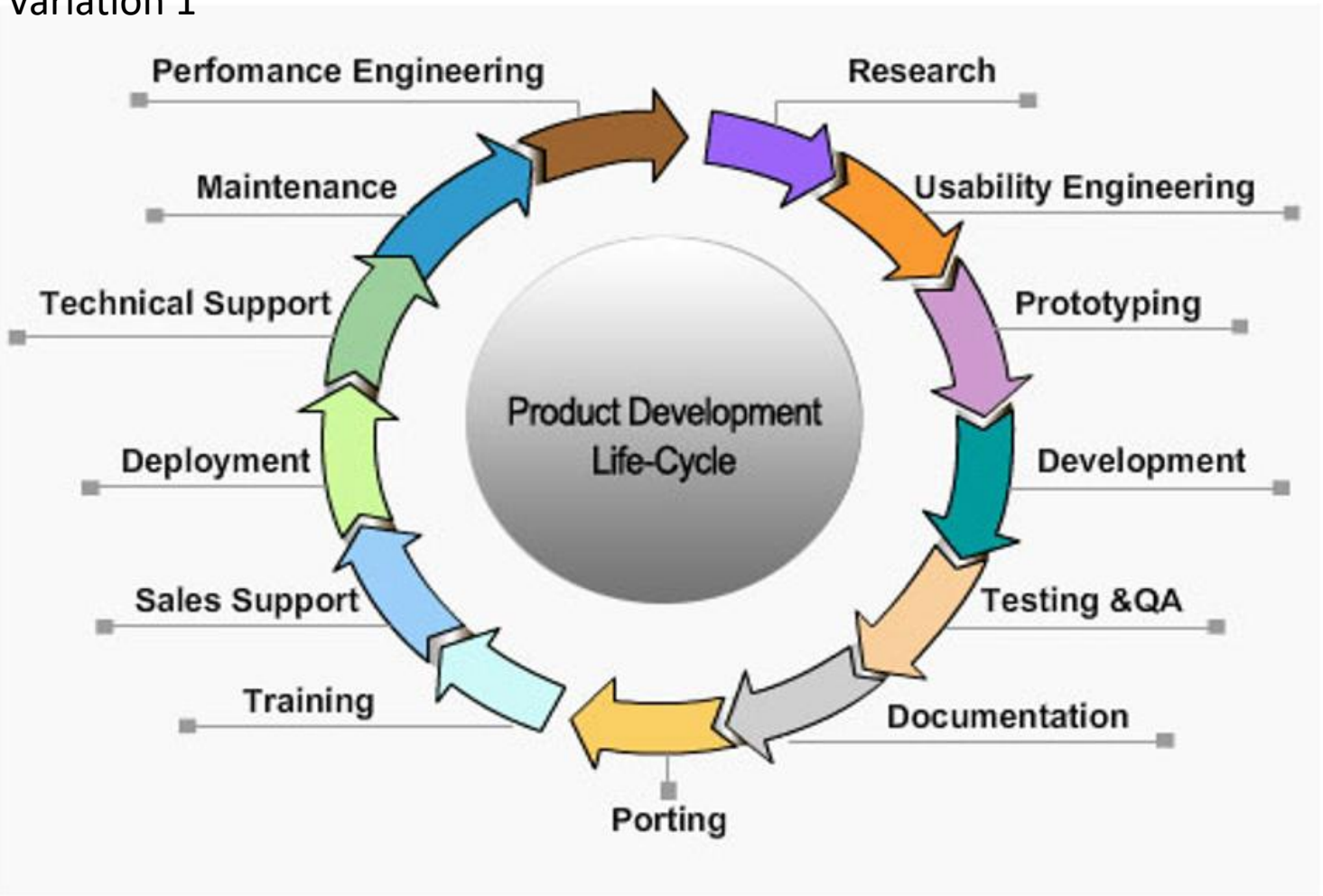
Software Development Lifecycle



Software Engineering

Product Development Lifecycle

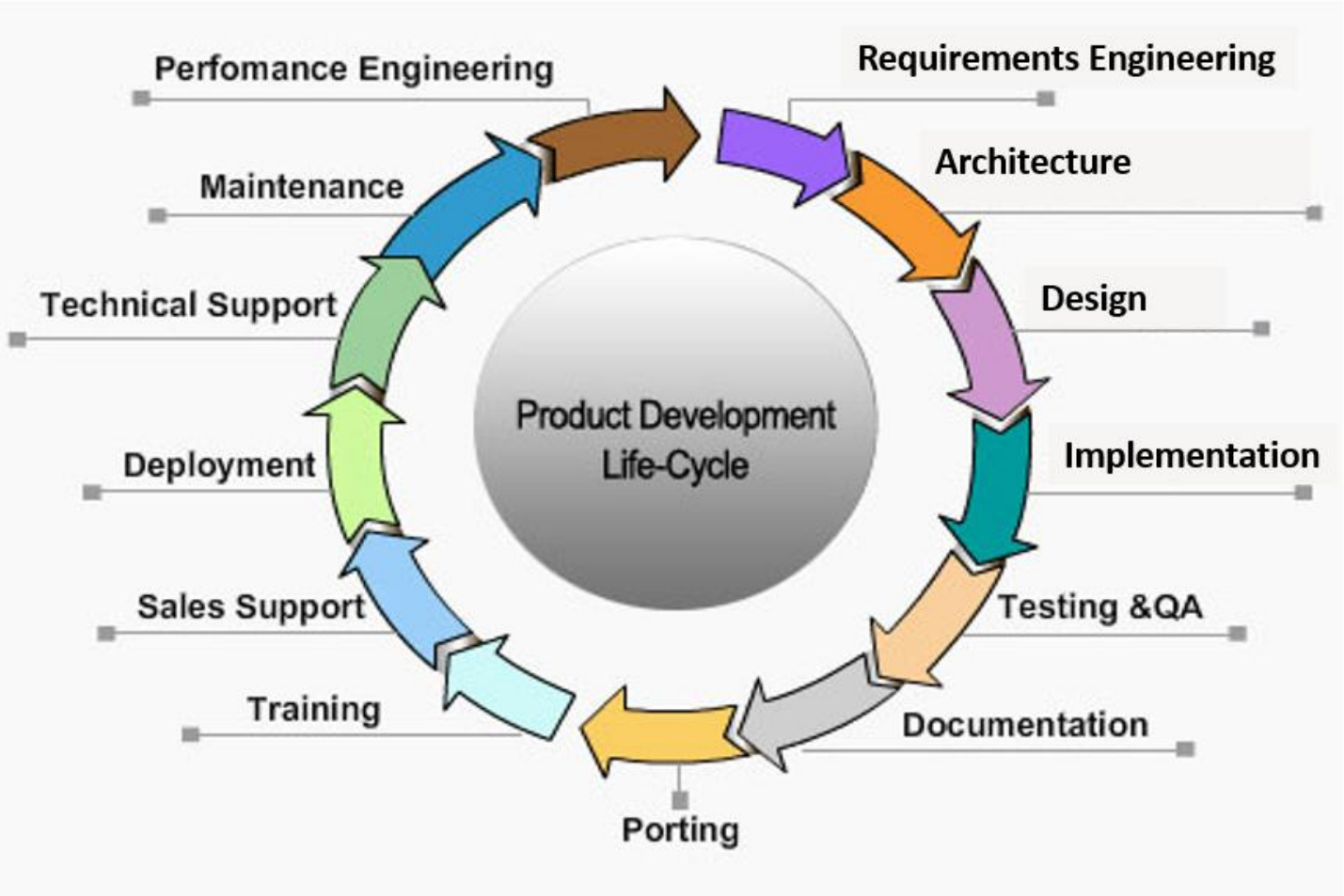
Variation 1



Software Engineering

Product Development Lifecycle

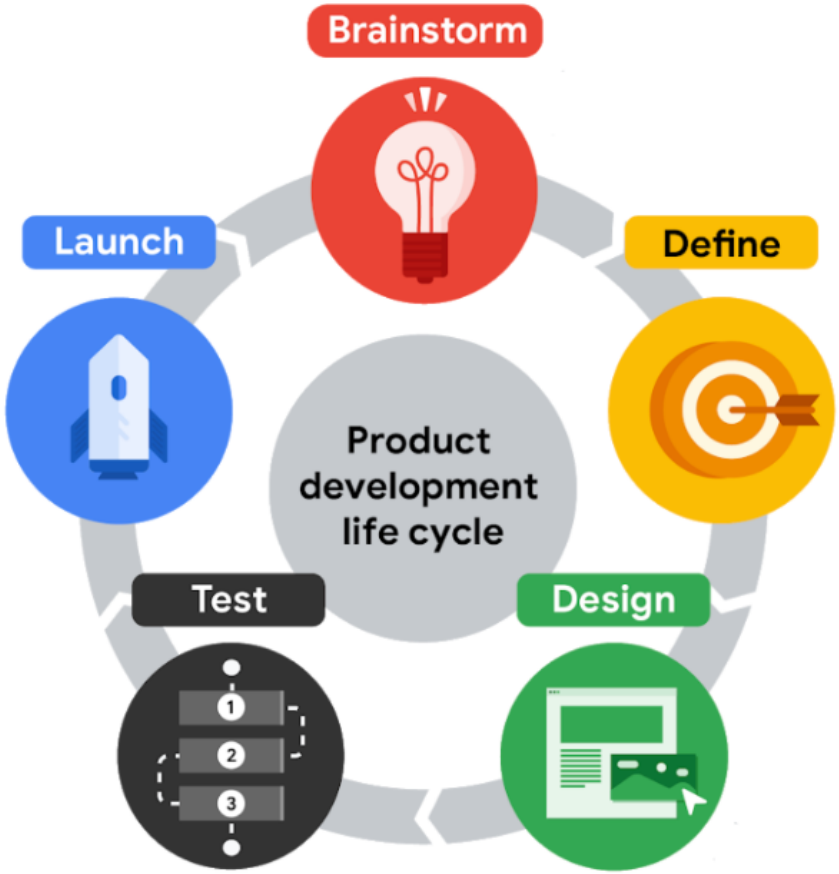
Variation 2



Software Engineering

Product Development Lifecycle

Variation 3



Product Development Lifecycle …

Brainstorm

An active discovery stage that's all about generating ideas about the user and potential needs or challenges that the user might have, it's important to pay attention to the diversity of your team at this stage.

The brainstorming stage is also an ideal time to check out your product's competitors and identify if there are already similar products available in the market, you want your product to fill a gap in the market or solve a problem better than existing products.

note: A UX designer at a large company might not be very involved in the brainstorming stage but a UX designer at a startup or small business could have a big role to play.

Product Development Lifecycle …

Define

This stage brings UX designers, UX researchers, program managers, and product lead to define the product.

The goal is to figure out the specifications for the product by answering questions like who is the product for? what will the product do? and what features need to be included for the product to be successful?

Product Development Lifecycle …

Design

Here the implementation of insights into new designs using various tools is done. At this stage UX designers develop the ideas for the product, generally, UX designers start by drawing wireframes, which are outlines or sketches of the product, then move on to creating prototypes, which are early models of a product that convey its functionality.

UX writers are also involved in the design stage and might do things like write button labels. At this point in the lifecycle, UX designers make sure to include all of the product specifications that were outlined in the define stage and ensure each part of the design fits together.

Software Engineering

Product Development Lifecycle …

Test

Evaluating the product designs based on the feedback of the potential users.

UX designers work with engineers to develop functional prototypes that match the original designs, including details and features that fit the company's brand, like font and color choices. This also means writing the code and finalizing the overall structure of the product.

At this stage, the designs go through at least 3 phases of testing internal tests: within the company reviews with stakeholders, and external tests with potential users.

Alpha testing - The team tests the product internally to look for technical glitches and usability problems.

Beta testing - an external test with potential users. This is the time to figure out whether the product provides a good user experience, meaning its usable, equitable, enjoyable, and useful

Product Development Lifecycle …

Launch

Sharing a finished version of the product with the public.

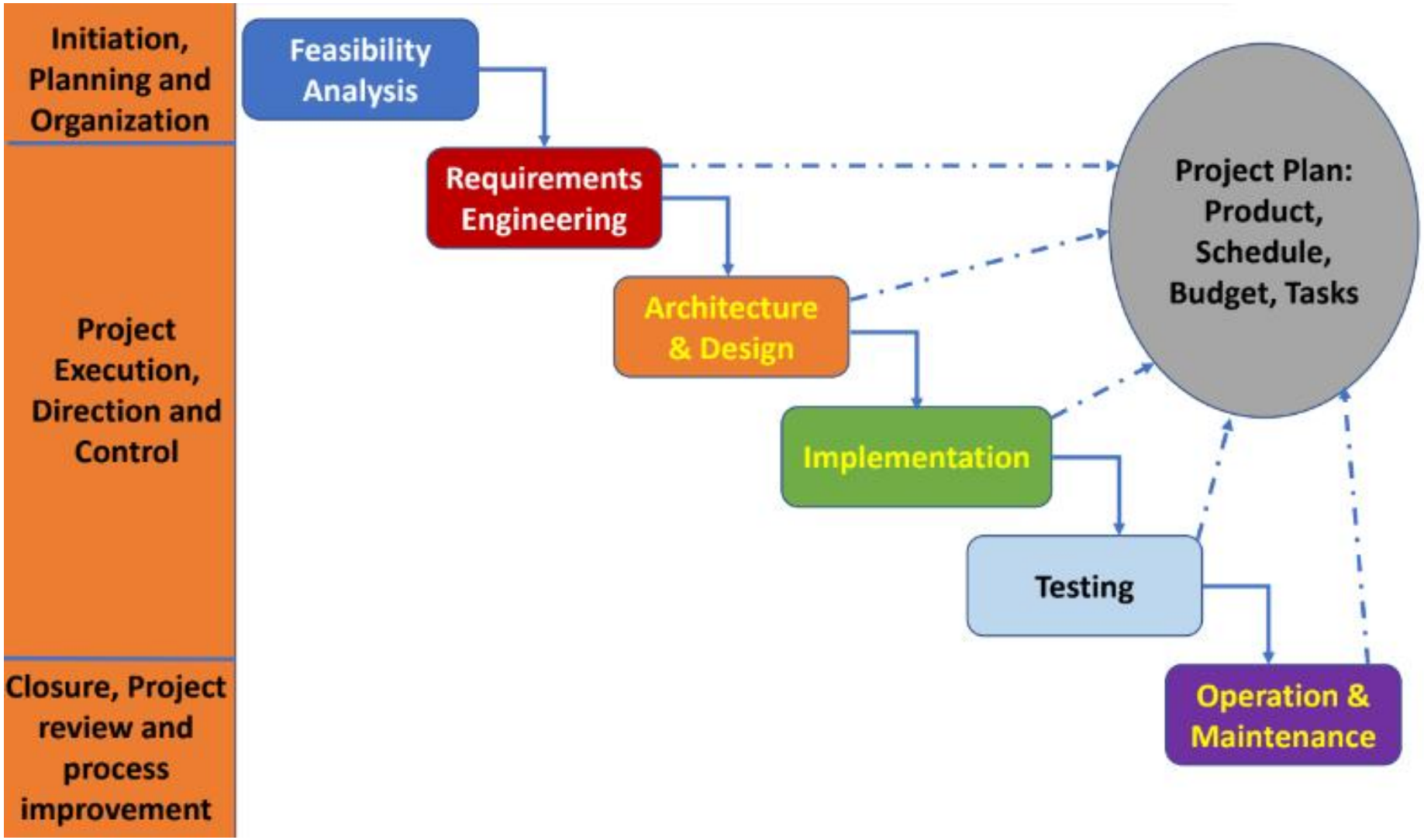
Marketing professionals on your team might post about the new product on social media or publish a press release. The customer support team might get ready to help new users learn how the product works.

Program managers also meet with cross-functional teams to reflect on the entire product development life-cycle and ask questions like what worked and what could be improved? were goals achieved? were timelines met? making time for this reflection is super important since it can help improve the process going forward.

For a physical product, the launch stage might be the end of the product development life cycle. But for a digital product, like an app or website, launching the product to a wider audience provides another opportunity to improve on the UX. So, after the launch stage, teams will often cycle back to the design stage and testing stages to start working on the next version of the digital products.

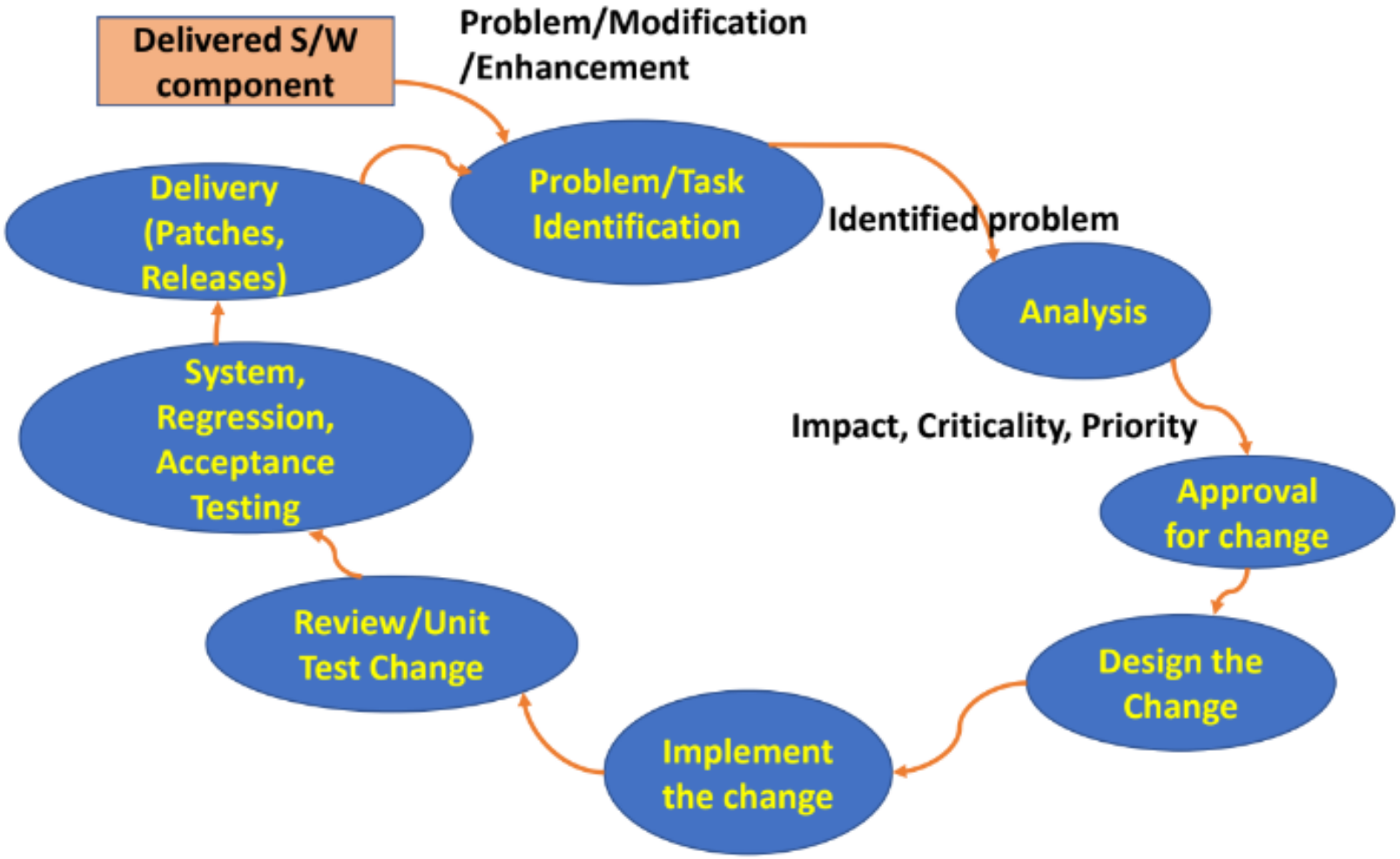
Software Engineering

Project Management Lifecycle (PMLC)



Software Engineering

Software Maintenance Lifecycle (SMLC)



Software Engineering

Product Lifecycle

A product life cycle is **the length of time from a product first being introduced to consumers until it is removed from the market**.

A product's life cycle is usually broken down into four stages; introduction, growth, maturity, and decline.

Introduction: This phase generally includes a substantial investment in advertising and a marketing campaign focused on making consumers aware of the product and its benefits.
Growth: If the product is successful, it then moves to the growth stage. This is characterized by growing demand, an increase in production, and expansion in its availability.
Maturity: This is the most profitable stage, while the costs of producing and marketing decline.
Decline: A product takes on increased competition as other companies emulate its success—sometimes with enhancements or lower prices. The product may lose market share and begin its decline.

Software Engineering

Product Lifecycle

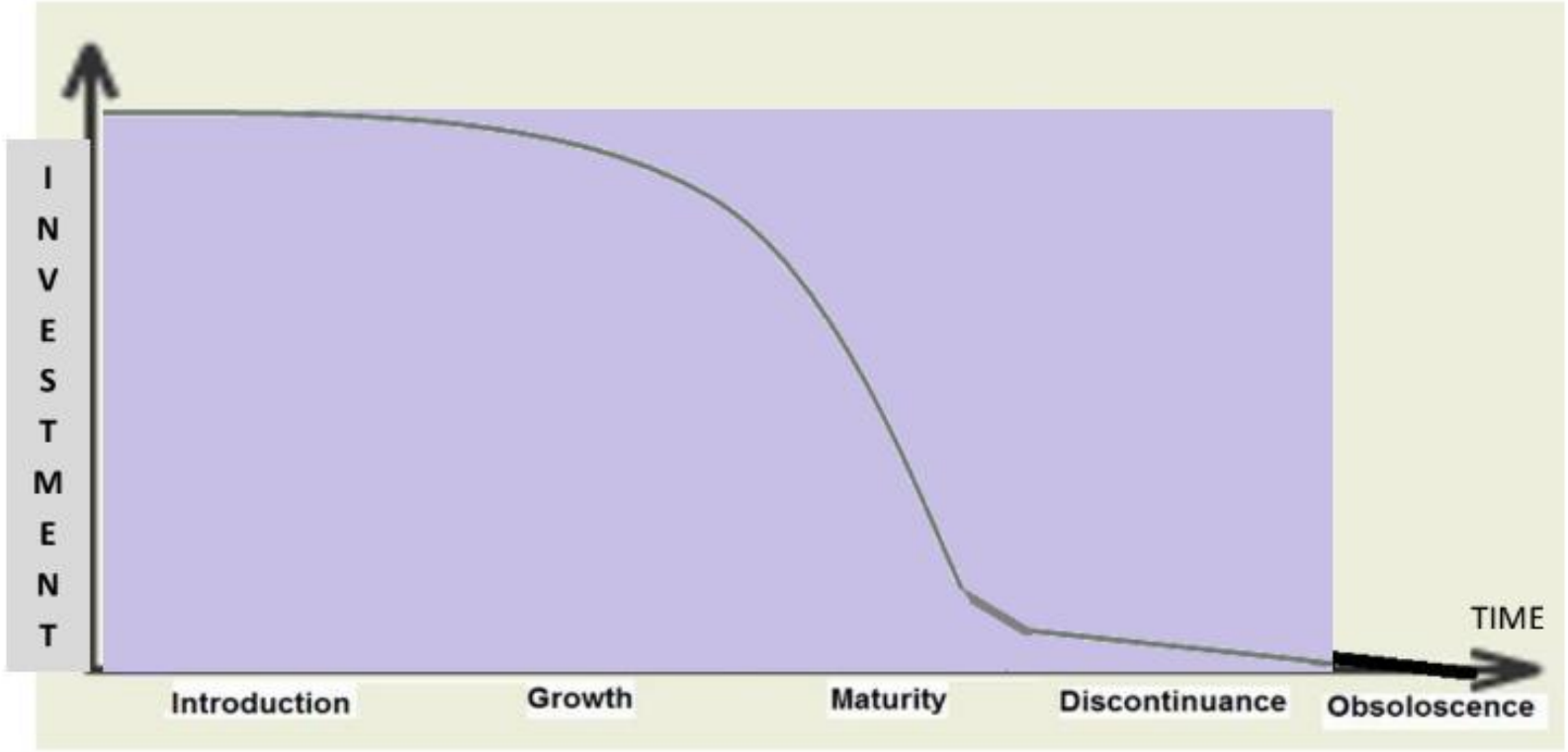
Attributes to Consider

1. Market capitalization
2. Sales
3. Investment
4. Competition
5. Profit
6. Support

Product built using a SDLC is initially released into market

Software Engineering

Product Lifecycle characteristics

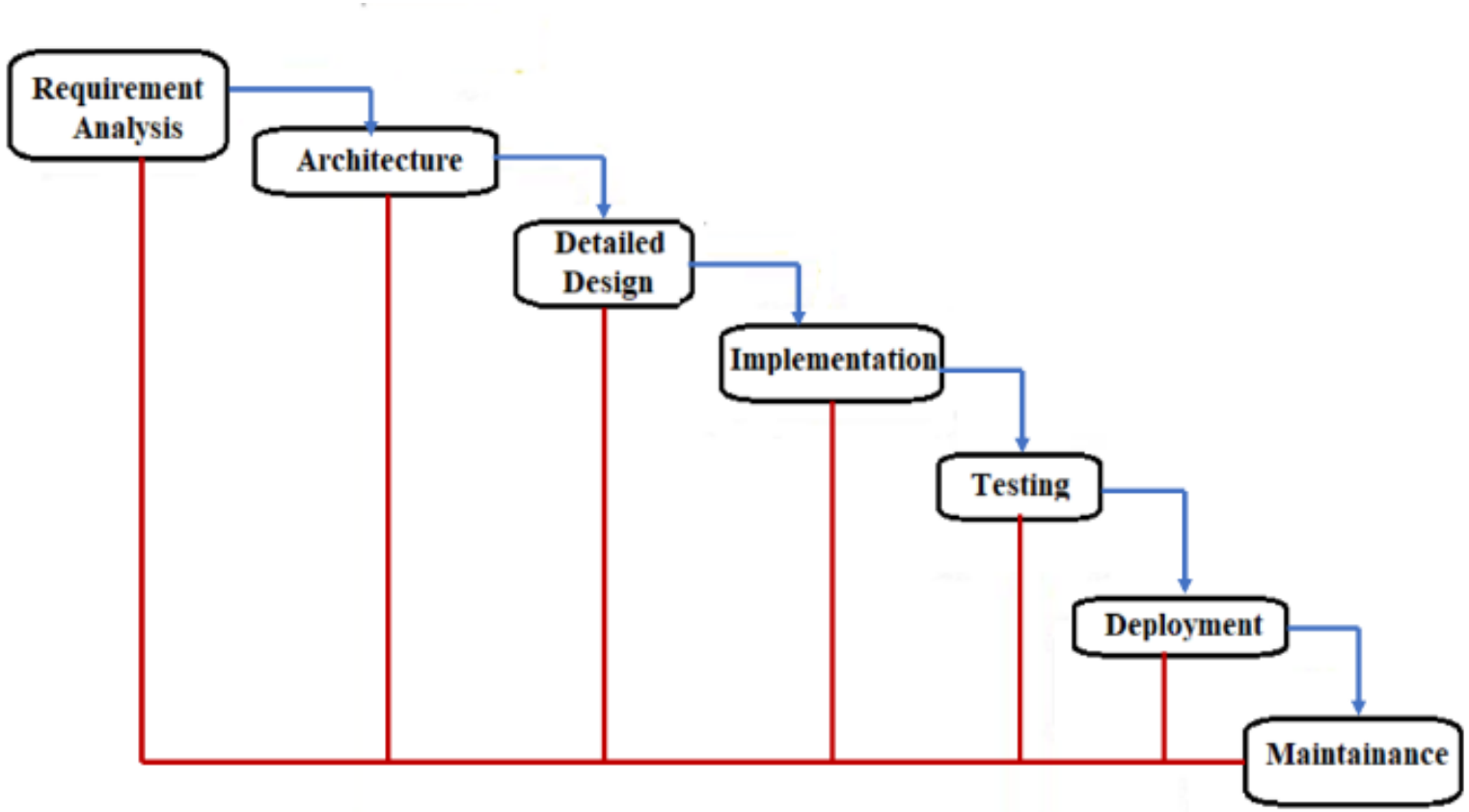



PES
UNIVERSITY
ONLINE

Software Engineering

Legacy Software Development Life Cycles (SDLC)

Waterfall model



Waterfall model

ADVANTAGES	DISADVANTAGES
Simple	Assumes requirements are frozen
Clear identified phases	Difficult to change & sequential
Easy to manage due to rigidity	Poor model for long projects
Each phase – specific deliverables + reviews	Big Bang approach
Easy to departmentalize and control	High risk + Uncertainty

Legacy Software Development Life Cycles (SDLC)

When can you use this model?

Pure form: Short projects where requirements are well known

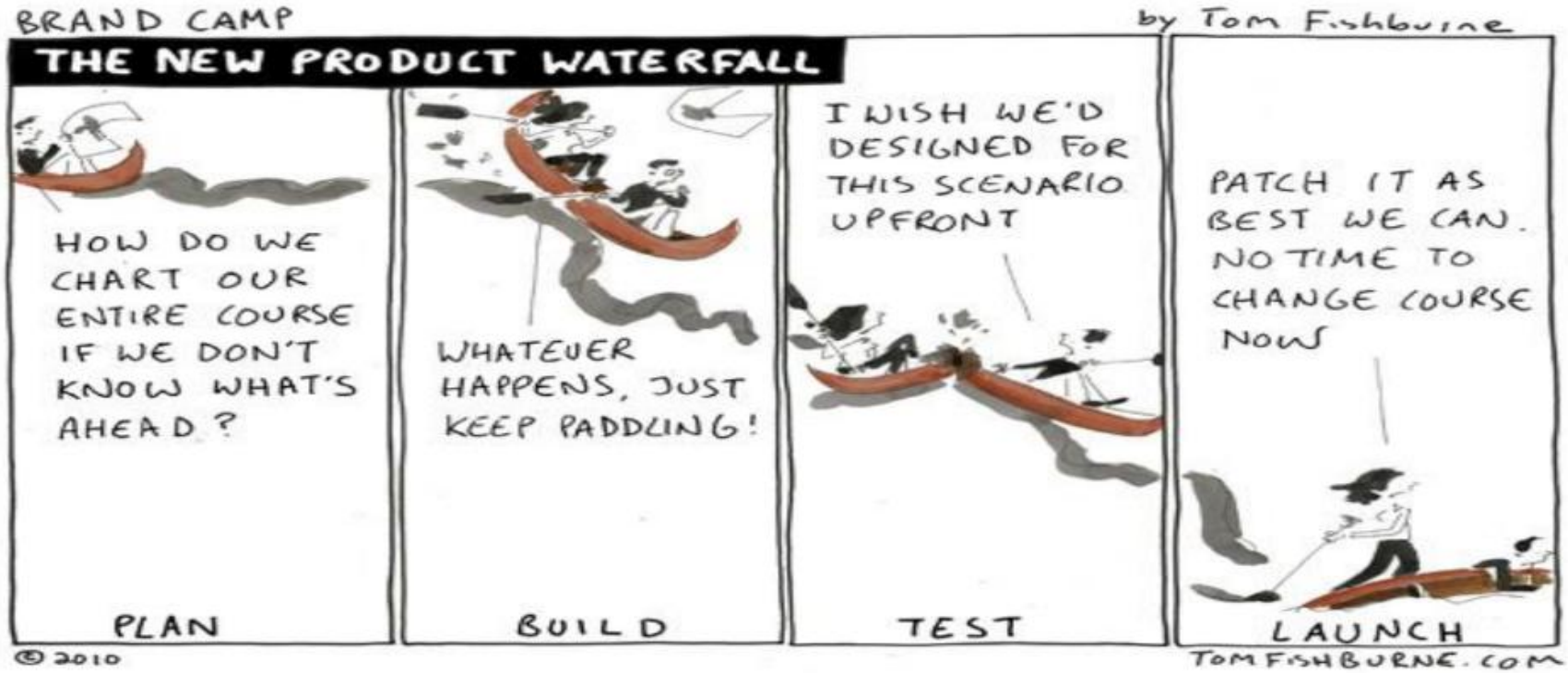
Product definition is stable & technology is understood

Variant form: High level in long projects

Software Engineering

Legacy Software Development Life Cycles (SDLC)

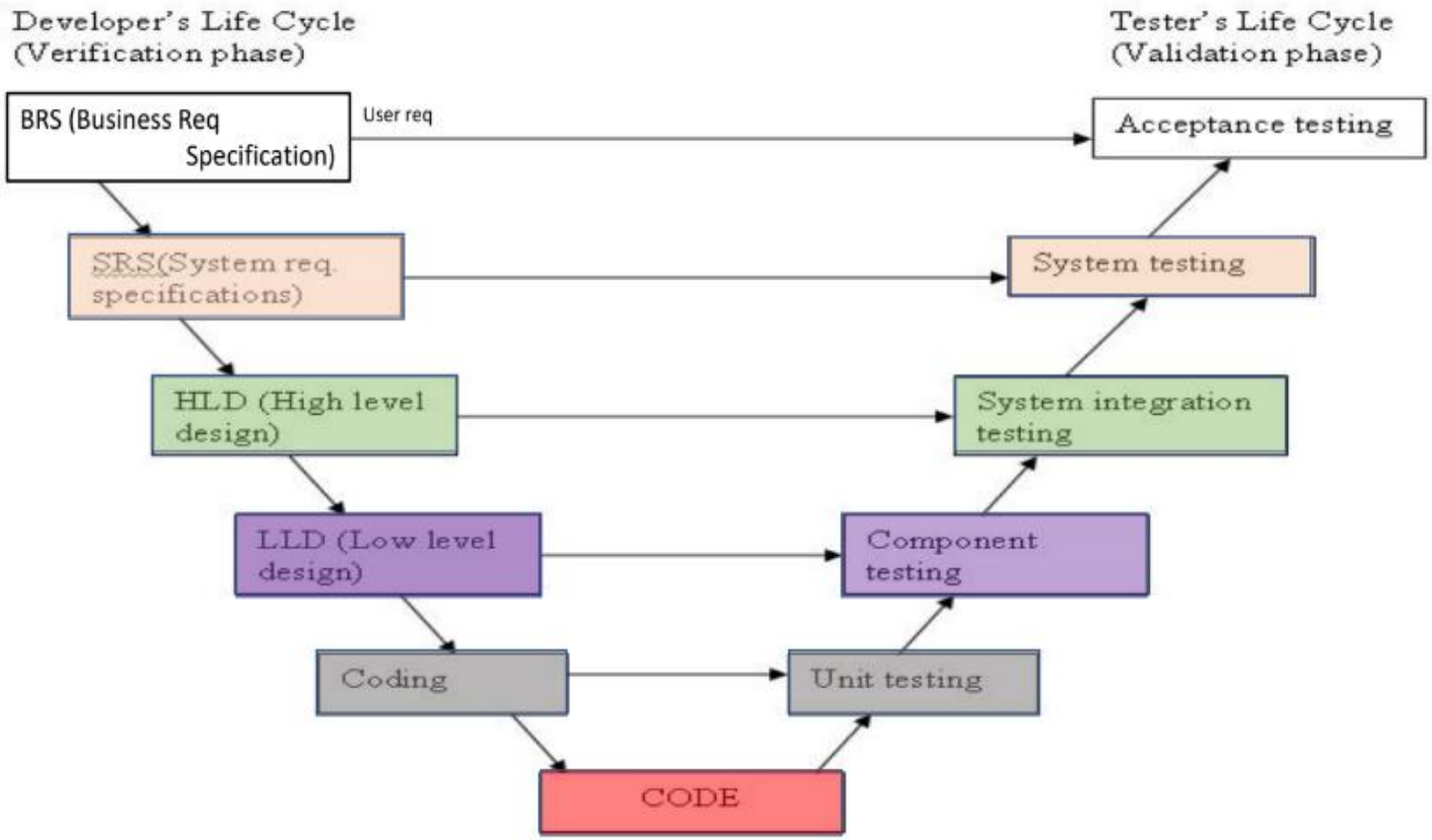
Waterfall model



Software Engineering

Legacy Software Development Life Cycles (SDLC)

The V model



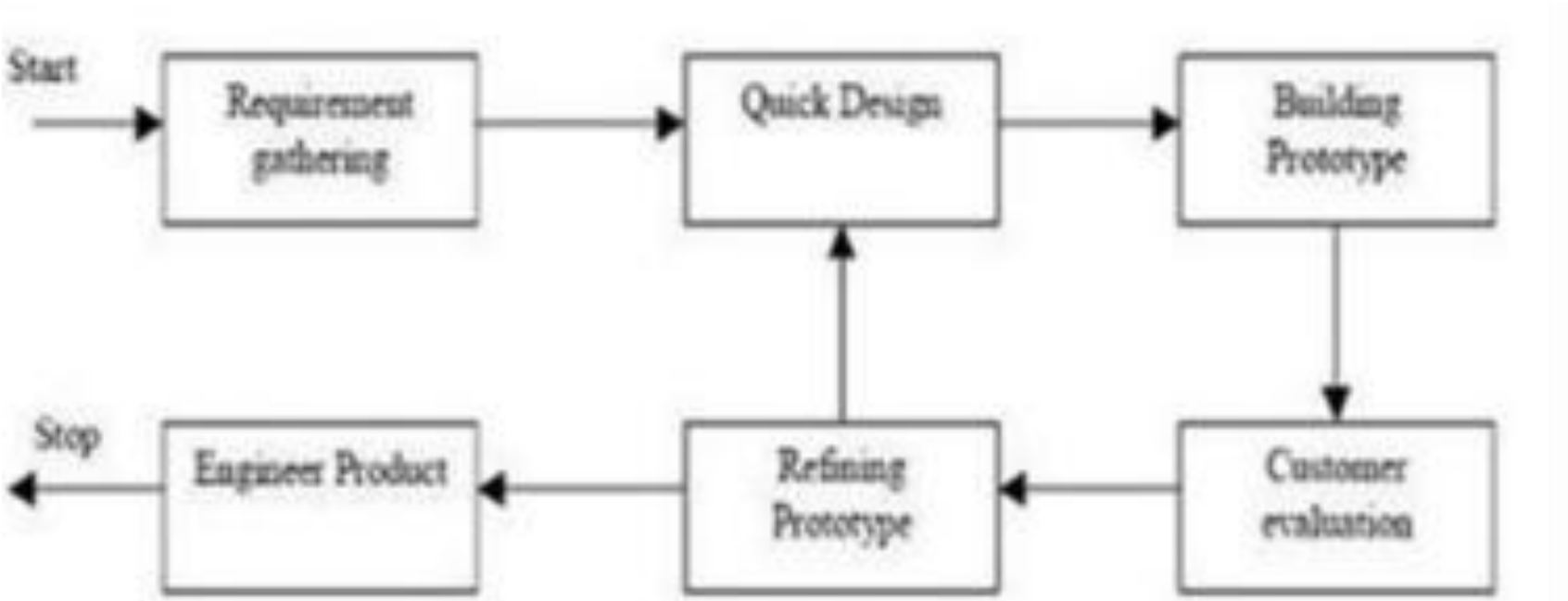
The V model

ADVANTAGES	DISADVANTAGES
Similar to Waterfall model	Similar to Waterfall model
Test development activities can happen before formal testing cycle	No early prototypes of software
Higher probability of success + Increased effectiveness of usage of resources	Change in process => change in test documentation

Usage: Similar to waterfall model

Legacy Software Development Life Cycles (SDLC)

The Prototype model



- Cheap
- Entire system prototype is built to understand the requirements
- Types: Throw-away and Evolutionary

The Prototype model – advantages and disadvantages

ADVANTAGES	DISADVANTAGES
Active involvement of users	May increase complexity of system as scope of system may expand beyond original plans
Better risk mitigation, Reduced time and cost, Resulting system is full featured, More stable system	Performance of resulting system may not be optimal

Usage:

- When requirements are not clear
- Users are actively involved

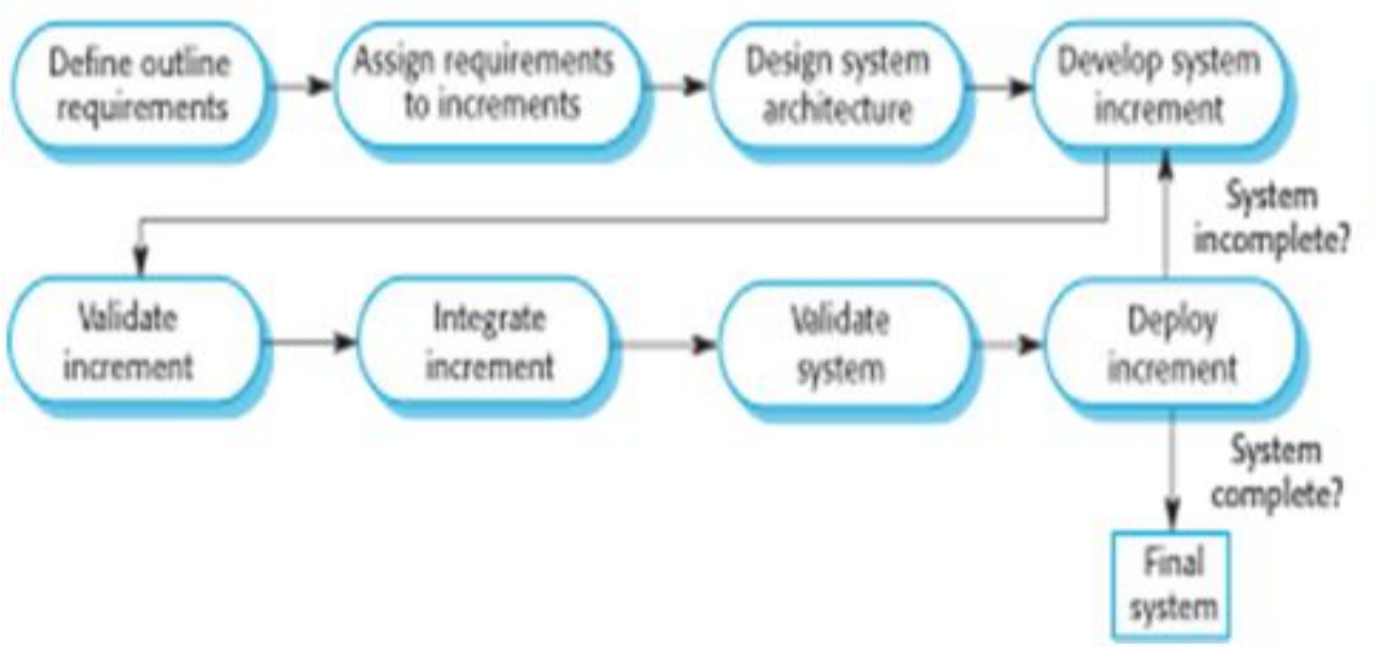
Legacy Software Development Life Cycles (SDLC)

Incremental model

- Requirements are partitioned
- Working software in first increment(module).
- Each subsequent release adds functionality to previous module
- Continuous integration is done until entire system is achieved

Legacy Software Development Life Cycles (SDLC)

Incremental model



- Partitioned requirements can have a development lifecycle
- Models like waterfall can be used for each partition

Legacy Software Development Life Cycles (SDLC)

Incremental model

ADVANTAGES	DISADVANTAGES
Customer value and more flexible	Needs good planning and design
Easier to test and debug	Needs clear and complete definition of whole system
Easier to manage risk	Total cost is higher than waterfall
Continuous increments rather than monolithic	Hard to identify common functionalities across increments
Reduces over functionality	Management visibility is reduced

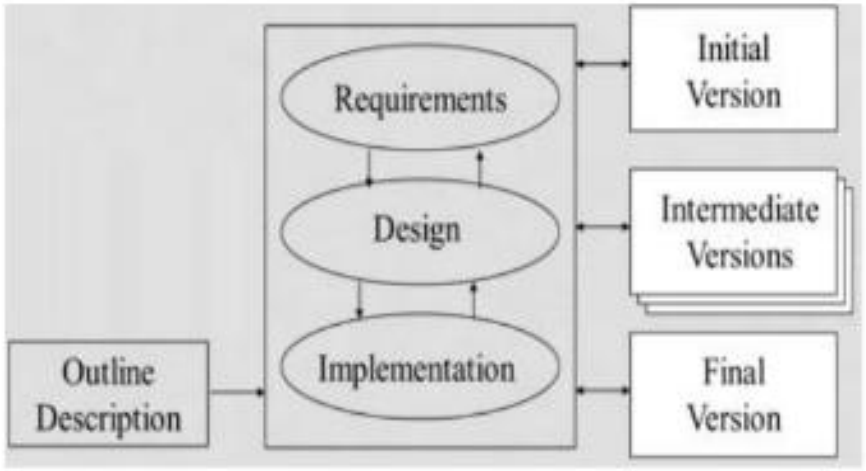
Incremental model
When to use?

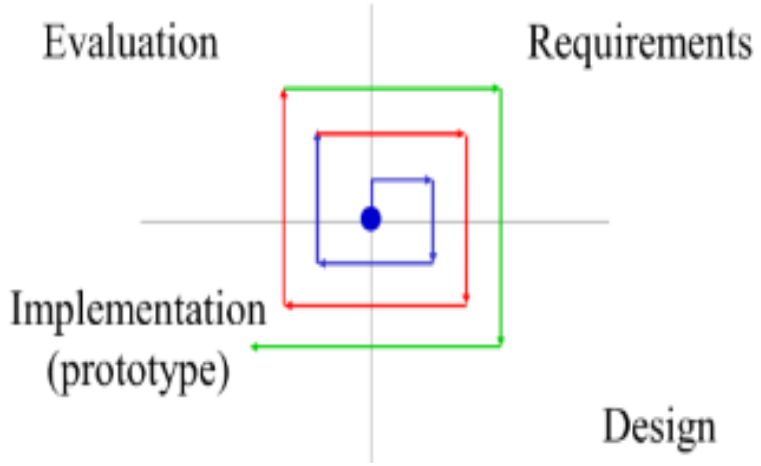
- Major requirements are defined
- Product needs to get to market early
- New technology is used
- High risk features and goals
- Resources with required skill set unavailable

Software Engineering

Legacy Software Development Life Cycles (SDLC)

Iterative model





Legacy Software Development Life Cycles (SDLC)

Iterative model

- Initial implementation starts from a skeleton of product
- This is followed by refinement through user feedback & evolution
- Built with dummy modules
- Rapid prototyping
- Successive refinement

Legacy Software Development Life Cycles (SDLC)

Iterative model

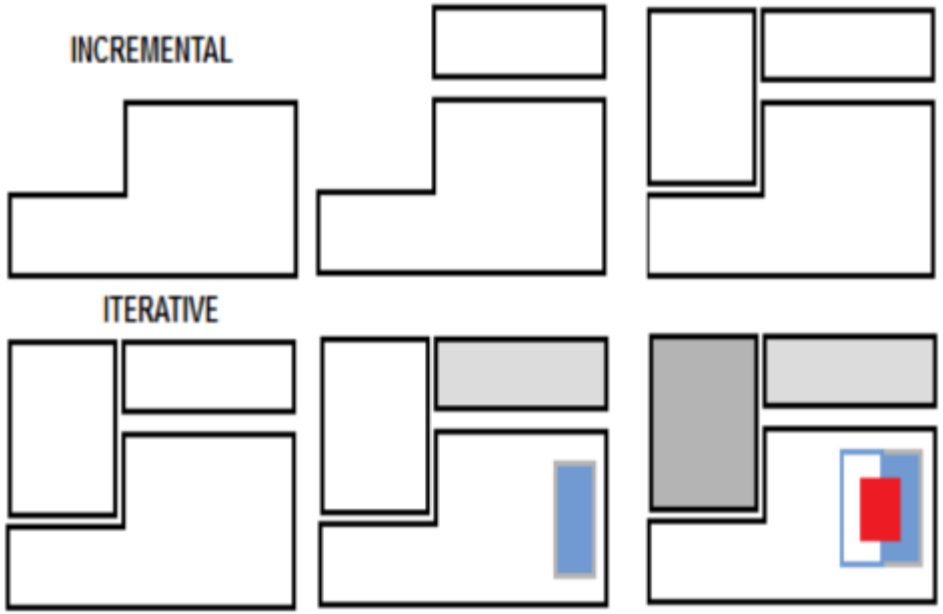
ADVANTAGES	DISADVANTAGES
Help identify requirement & solution visualization	Each phase is rigid with overlaps
Support risk mitigation, rework is reduced, incremental investment, feature creep, increased customer engagement	Costly system architecture may arise

USAGE: Large projects which may get extended

Software Engineering

Iterative model vs Incremental model

ITERATIVE MODEL	INCREMENTAL MODEL
Revisit and refine every thing	No need to go back and change delivered things
Focus on details of things	Focus on things not implemented yet
Leverage on learnings	Does not leverage on experience or knowledge




PES
UNIVERSITY
ONLINE

Limitations of most legacy lifecycle models

1. Predictive software development methods
2. Upfront planning
3. Do not facilitate periodic customer interaction
4. Suited for large complex projects
5. Regulatory perspectives
6. Suited for global and distributed organizations
7. Product lifecycle and its ecosystem
8. People and skill perspective
9. Suitable for projects with clear definition
10. Suitable when things are not changing too fast

Software Engineering

Agile philosophy

Agile is an umbrella term used to describe a variety of methods

Agile methods encourage:

1. Continual realignment of development goals with needs and expectations of the customer
2. Reducing massive planning overhead to allow fast reactions to change

Agile philosophy …

Agile is not a process.

It is a set of values or philosophy.

AGILE key words

- Rapid
- Iterative
- Cooperative
- Quality driven
- Adaptable

Software Engineering

Agile manifesto

Individual and interactions	over	Process and tools
Working software	over	Comprehensive documentation
Customer collaboration	over	Contract negotiation
Responding to change	over	Following a plan

While there is value in the items on the right, we value the items on the left more.

Software Engineering

Agile principles

The **principles** of agile methods:

Customer involvement

Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.

Incremental delivery

The software is developed in increments with the customer specifying the requirements to be included in each increment.

People not process

The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.

Embrace change

Expect the system requirements to change and so design the system to accommodate these changes.

Maintain simplicity

Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

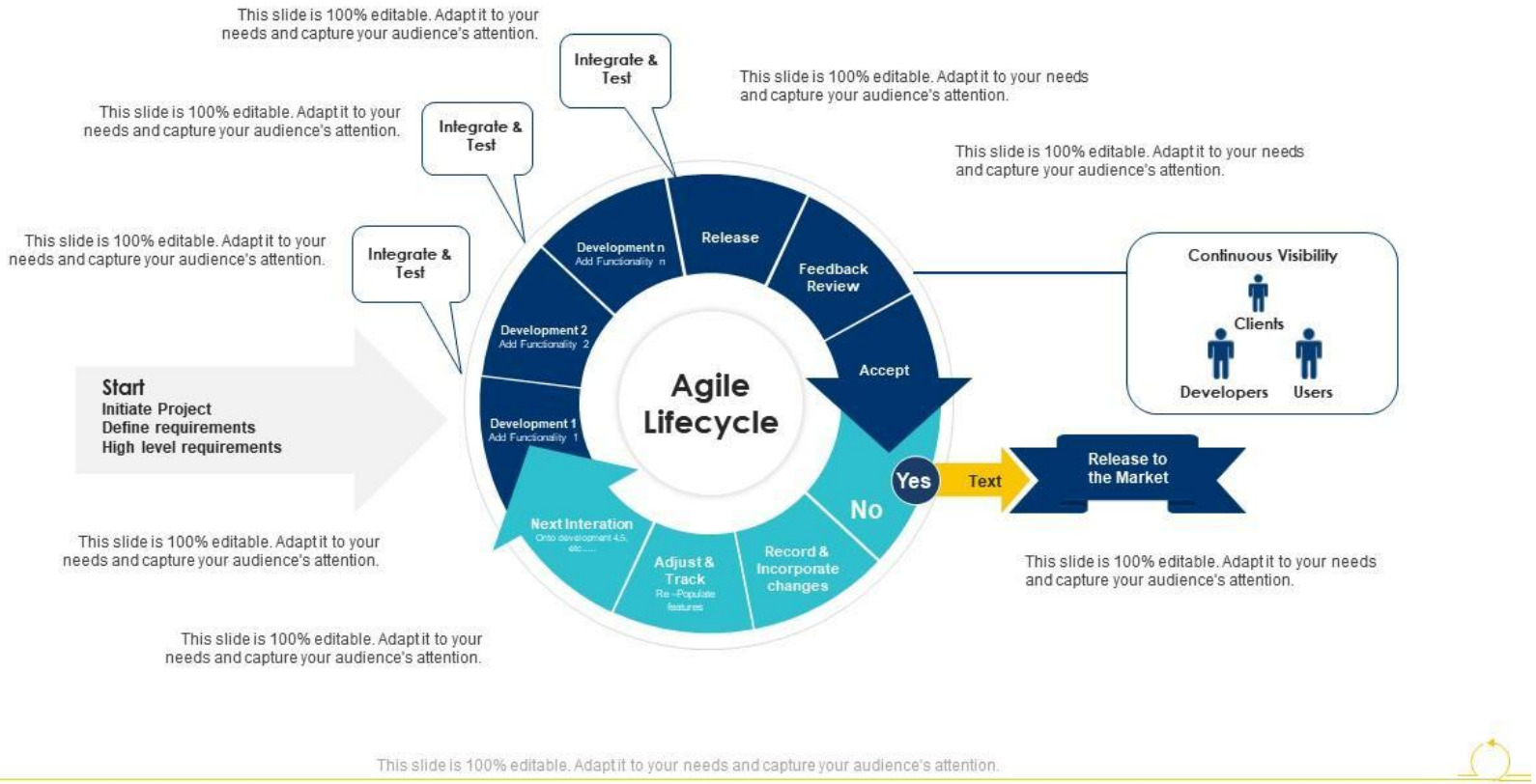
Software Engineering

Agile methodology

Pros	Cons
Is a very realistic approach to software development	Not suitable for handling complex dependencies.
Promotes teamwork and cross training.	More risk of sustainability, maintainability and extensibility.
Functionality can be developed rapidly and demonstrated.	An overall plan, an agile leader and agile PM practice is a must without which it will not work.
Resource requirements are minimum.	Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines.
Suitable for fixed or changing requirements	Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction.
Delivers early partial working solutions.	There is very high individual dependency, since there is minimum documentation generated.
Good model for environments that change steadily.	Transfer of technology to new team members may be quite challenging due to lack of documentation.
Minimal rules, documentation easily employed.	
Enables concurrent development and delivery within an overall planned context.	
Little or no planning required	
Easy to manage	
Gives flexibility to developers	

Software Engineering

SDLC in Agile



Software Engineering

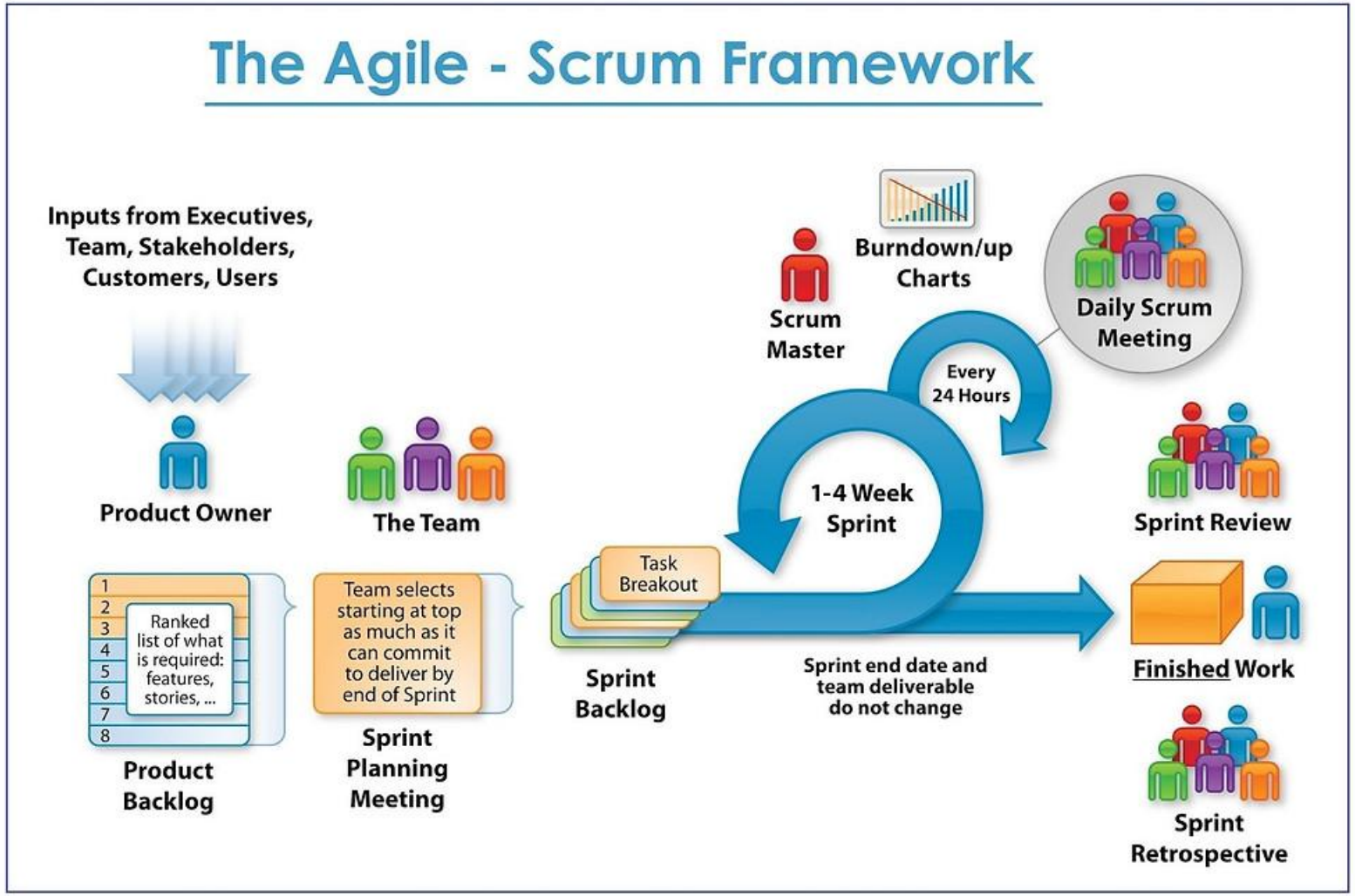
Agile methodologies - Scrum

Scrum is a framework of **rules**, **roles**, **events**, and **artifacts** used to implement Agile projects. It is an iterative approach, consisting of sprints that typically only last one to four weeks. This approach ensures that your team delivers a version of the product regularly.

Scrum was designed using a software model that follows a set of roles, responsibilities, and meetings. It can be used for any complex project but works best when the result is a concrete product rather than a service.

Software Engineering

Agile methodologies - Scrum



Software Engineering

Agile methodologies – Scrum in brief

Roles

Product Owner (PO): *Responsible for the product success*

- Envisions the product
- Is the only one responsible for the Product Backlog (items and prioritization)
- Is responsible for the product 's profitability (ROI)
- Decides on release date and content
- Accepts or rejects work results
- Collaborates with both the team and stakeholders

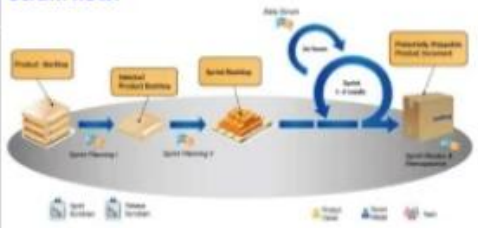
Team: *Responsible for delivering product functionalities*

- Self-organizing
- Cross-functional with no roles
- Seven plus or minus two
- Responsible for meeting their commitments
- Authority to do whatever is needed to meet commitments

ScrumMaster: *Responsible for the success of Scrum*

- Enforces the Scrum Rules
- Facilitates all the Scrum meetings
- Shields the team from external interference
- Leads the team to be self-organizing and to continuously improve
- Coaches the PO on his role
- Serves the team and PO
- Removes obstacles
- Is a change agent

Scrum flow:



Meetings

Sprint Planning Part I: *Define "What" to do*

- PO presents top priority Product Backlog items
- Team selects the amount of Backlog for the upcoming Sprint
- Acceptance criteria are negotiated and clarified
- Sprint Goal is defined

Sprint Planning Part II: *Define "How" to do*

- Team participates while PO's available
- Team breaks items into tasks to form the Sprint Backlog
- Involves detailed design
- Team makes commitment for the Sprint

Daily Scrum: *Inspection and adaptation meeting for the Sprint*

- 15 minute Daily status meeting
- Same place and time every day
- Three questions for everyone
  - What have you completed since last meeting?
  - What will you complete before next meeting?
  - What is in your way?
- Team updates the Sprint backlog and Sprint Burndown chart
- Open meeting for all, but only Scrum Team members can talk

Sprint Review: *Inspection and adaptation meeting about the product*

- Team presents the "Done" work and "Undone" work
- Get feedback from the Product Owner and Stakeholders
- Update Product Backlog and release Burndown chart

Sprint Retrospective: *Inspection and adaptation meeting about the process*

- Scrum Team inspects the last sprint regarding people, relationships, processes and tools
- Scrum Team identifies possible improvements and agrees on the measures for next Sprint
- Scrum Team may update its own working agreement

Artifacts

Product Backlog: *List of desired product features*

- Is Detailed, Emergent, Estimated, Prioritized (DEEP)
- More details on higher priority backlog items
- Maintained by the Product Owner but anyone can contribute
- One list per product
- Needs to be groomed every Sprint

Sprint Backlog: *Tasks to turn Product Backlog items into working product functionality*

- The selected Product Backlog items for the sprint do not change during the sprint
- Made and maintained by the team throughout the Sprint
- Any team member can add, delete or change a task the Sprint Backlog
- Team members sign up for tasks, they aren't assigned
- The size of a task should be less than 1 day
- Estimated work remaining is updated daily

Sprint Burndown chart: *Shows remaining work in a Sprint*

- Calculated with the number of remaining tasks or story points
- Updated daily by the team

Release Burndown Chart: *Shows remaining work in a release*

- Calculated in story points
- Maintained by product owner
- Updated every Sprint

The 3 pillars of Scrum:

Transparency

Inspection

Adaptation

Glossary

Timebox: A period of time of fixed length which cannot be exceeded.

Scrum Team: The Team, the PO and the ScrumMaster form the Scrum Team.

Definition of "Done" (DoD): List of development activities required to consider an increment of functionality as "Done".

Sprint Taskboard: A board containing the team's Sprint goals, Sprint Backlog and the Sprint Burndown chart. Physical white boards are recommended.

Velocity: The rate at which team converts items to "DONE" in a single Sprint. It is usually calculated in Story Points.

User Story: a short description of a behavior of the system in the point of view of the user.
User Story template:
As a <User>, I can <function> so that <desired result>.

Story Points: a relative measure of the size of the user stories. Can have different scales, typically Fibonacci sequence as in Planning Poker.

The 5 Scrum values:

Commitment

Focus

Openness

Respect

Courage

Software Engineering

Agile scrum

A burndown chart **shows the amount of work that has been completed in an epic or sprint, and the total work remaining**. Burndown charts are used to predict your team's likelihood of completing their work in the time available. They're also great for keeping the team aware of any scope creep that occurs.

Software Engineering

Scrum in brief

SCRUM CHEAT SHEET

Roles

Scrum Team

- Team is cross-functional and consists of 5-9 people
- There are no set project roles within the team
- Team defines tasks and assignments
- Team is self-organizing and self-managing
- Maintains the Sprint Backlog
- Conducts the Sprint Review

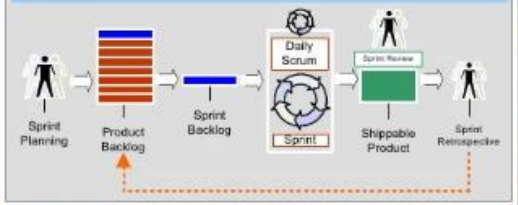
Product Owner (PO)

- Accountable for product success
- Defines all product features
- Responsible for prioritizing product features
- Maintains the Product Backlog
- Insures team working on highest valued features

Scrum Master (SM)

- Holds daily 15 minute team meeting (Daily Scrum)
- Removes obstacles
- Shields the team from external interference
- Maintains the Sprint Burndown Chart
- Conducts Sprint Retrospective at the end of a Sprint
- Is a facilitator not a manager

Process



Tools

Task Board

- White Board containing teams Sprint goals, backlog items, tasks, tasks in progress, "DONE" items and the daily Sprint Burndown chart.
- Scrum meeting best held around task board
- Visible to everyone

Artifacts

Product Backlog - (PB)

- List of all desired product features
- List can contain bugs, and non-functional items
- Product Owner responsible for prioritizing
- Items can be added by anyone at anytime
- Each item should have a business value assigned
- Maintained by the Product Owner

Sprint Backlog – (SB)

- To-do list (also known as Backlog item) for the Sprint
- Created by the Scrum Team
- Product Owner has defined as highest priority

Burndown Chart – (BC)

- Chart showing how much work remaining in a Sprint
- Calculated in hours remaining
- Maintained by the Scrum Master daily

Release Backlog – (RB)

- Same as the Product Backlog. May involve one or more sprints dependent on determined Release date

"DONE"= Potentially Shippable!

FAQ

- ***Who decides when a Release happens?*** At the end of any given Sprint the PO can initiate a Release.
- ***Who is responsible for managing the teams?*** The teams are responsible for managing themselves.
- ***What is the length of a task?*** Tasks should take no longer than 16 hours. If longer then the task should be broken down further.
- ***Who manages obstacles?*** Primary responsibility is on the Scrum Master. However, teams must learn to resolve their own issues. If not able then escalated to SM.
- ***What are two of the biggest challenges in Scrum?*** Teams not self-managing, Scrum Master managing not leading.

Meetings

Sprint Planning – Day 1 / First Half

- Product backlog prepared prior to meeting
- First half – Team selects items committing to complete
- Additional discussion of PB occurs during actual Sprint

Sprint Planning – Day 1 / Second Half

- Occurs after first half done – PO available for questions
- Team solely responsible for deciding how to build
- Tasks created / assigned – Sprint Backlog produced

Daily Scrum

- Held every day during a Sprint
- Lasts 15 minutes
- Team members report to each other not Scrum Master
- Asks 3 questions during meeting
- *"What have you done since last daily scrum?"*
- *"What will you do before the next daily scrum?"*
- *"What obstacles are impeding your work?"*
- Opportunity for team members to synchronize their work

Sprint Review

- Team presents "done" code to PO and stakeholders
- Functionality not "done" is not shown
- Feedback generated - PB maybe reprioritized
- Scrum Master sets next Sprint Review

Sprint Retrospective

- Attendees – SM and Team. PO is optional
- Questions – What went well and what can be improved?
- SM helps team in discovery – not provide answers

Visibility + Flexibility = Scrum

Glossary of Terms

- **Time Box** - A period of time to finish a task. The end date is set and can not be changed
- **Chickens** – People that are not committed to the project and are not accountable for deliverables
- **Pigs** – People who are accountable for the project's success
- **Single Wringable Neck** – This is the Product Owner!

Estimating

User Stories

- A very high level definition of what the customer wants the system to do.
- Each story is captured as a separate item on the Product Backlog
- User stories are NOT dependent on other stories
- **Story Template**:
- "As a <User> I want <function> So that <desired result>
- **Story Example**:
- As a user, I want to print a recipe so that I can cook it.

Story Points

- A simple way to initially estimate level of effort expected to develop
- Story points are a relative measure of feature difficulty
- Usually scored on a scale of 1-10. 1=very easy through 10=very difficult
- **Example**:
- "Send to a Friend" Story Points = 2
- "Shopping Cart" Story Points = 9

Business Value

- Each User Story in the Product Backlog should have a corresponding business value assigned.
- Typically assign (L,M,H) Low, Medium, High
- PO prioritizes Backlog items by highest value

Estimate Team Capacity

- Capacity = # Teammates (Productive Hrs x Sprint Days)
- Example – Team size is 4, Productive Hrs are 5, Sprint length is 30 days.
- Capacity = 4 (5 x30) = 600 hours
- **NOTE**: Account for vacation time during the Sprint!

Velocity

- The rate at which team converts items to "DONE" in a single Sprint – Usually calculated in Story Points.

Software Engineering

Agile methodology – Scrum in-class exercise

How is Scrum aligned to the Agile principles?

Individuals and interactions over Processes and tools - Scrum addresses this with Cross functional teams, Scrum Meetings, Sprint reviews

Working software over Comprehensive documentation - Periodic customer experienceable deliverables at the end of every sprint which can be reviewed and experienced

Customer collaboration over Contract negotiation - Having customers to experience the sprint outcomes and participate in sprint reviews to ensure they can visualize and ensure that product meets their needs

Responding to change over Following a plan - User stories which is picked at the beginning of every sprint which can ensure requirement changes can be factored in and prioritized unlike a plan which needs to be followed

Focus on Simplicity in both the product and the process by keeping the process simple, planning is short term focused and hence simple with more interactions and minimal documentation

Software Engineering

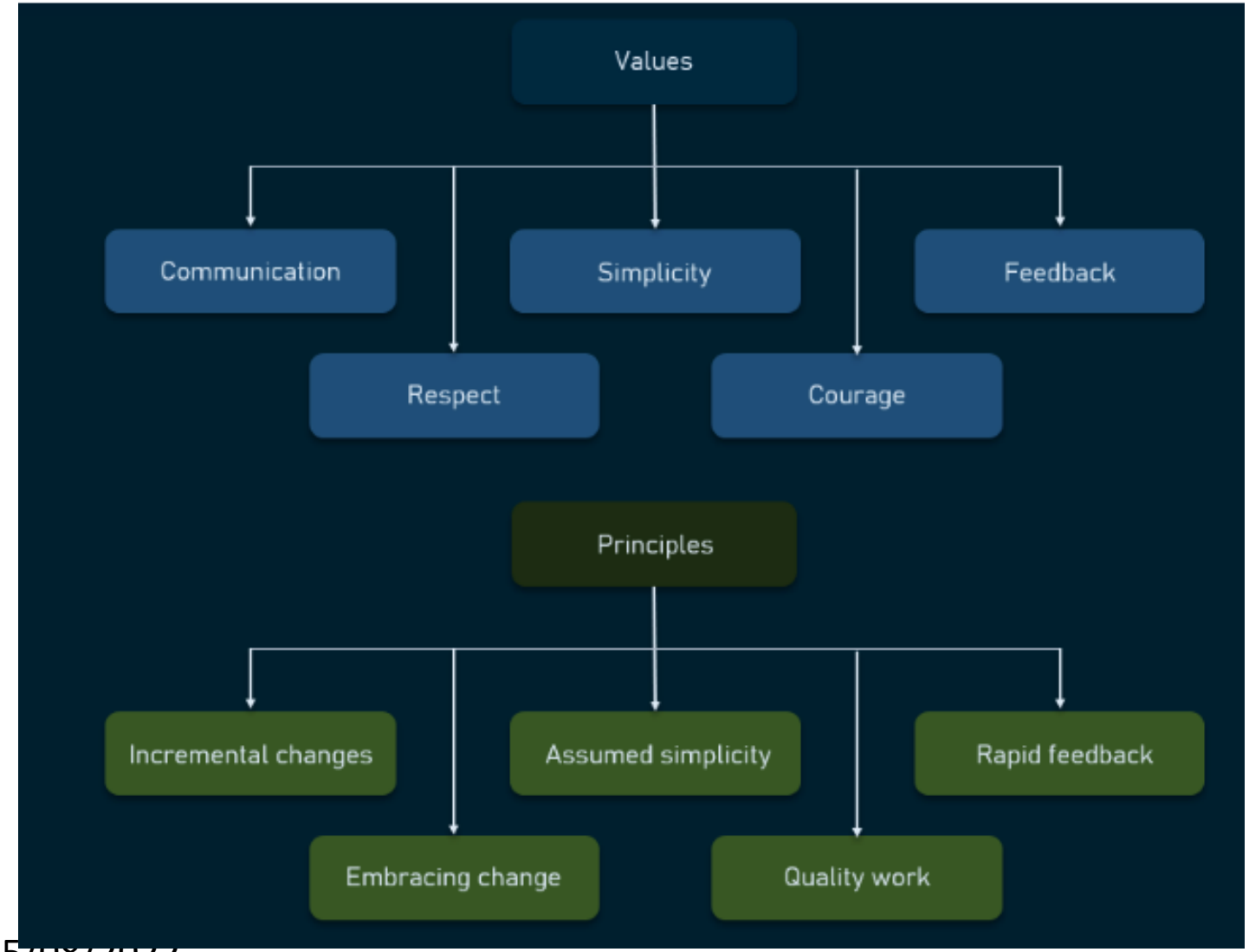
Agile methodologies – eXtreme Programming (XP)

Extreme programming is a software development methodology that's part of what's collectively known as agile methodologies.

XP is built upon **values**, **principles**, and **practices**, and its goal is to allow small to mid-sized teams to produce high-quality software and adapt to evolving and changing requirements.

Software Engineering

XP values and principles



XP core practices

#1 Planning Game

In XP the main planning process is called Planning Game. There are 2 levels of plans in XP; level one is release planning and level 2 is iteration planning.

The first phase is release planning. The team and stakeholders/customers collaboratively decide what are the requirements and features that can be delivered into production and when. This is done based on priorities, capacity, estimations and risks factor of the team to deliver.

In Iteration planning the team will pick up the most valuable items from the list and break them down into tasks then estimates and a commitment to delivering at the end of the iteration.

XP core practices …

#2 Simple design

In XP, the team will not do complex or big architecture and designs upfront; instead the team will start with a simple design and let it emerge and evolve over a period of iterations. The code is frequently refactored so that it will be maintainable and free of technical debt. Simple designs make the 'definition of done' easier.

XP teams conduct a small test or proof-of-concept workout called **spike**. The outcome of the spike helps the team to understand the validity of the hypothesis, gauge the complexity of the solution and feel assured to estimate and build something primarily based on the test.

#3 Test-Driven Development (TDD)

In XP, developers write the unit test cases before starting the coding. The team automates the unit tests and it helps during the build and integration stage. The main benefit of TDD is that programmers have to only write the code that passes the tests.

Basics steps of TDD,
Write the unit test case first
Write a minimal amount of code to pass the test.
Refactor it by adding the needed feature and functionality, while continuously making sure the tests pass.

XP core practices …

#4 Code Standard

Organizations want their programmers to hold to some well-described and standard style of coding called coding standards. It is a guideline for the development team as in XP. Since there are multiple programming pairs at play, coding standards are very useful to make consistency in code, style, naming conversion, exception handling and use of parameters.

These standards must be defined and agreed before the team starts the coding.

It will make the code simple to understand and helps detect the problem or issues quickly and also increases the efficiency of the software.

XP core practices …

#5 Refactoring

Refactoring, as the word suggests, is restructuring or reconstructing existing things. In XP, over a period of time, the team produces lots of working code which increases the complexity of the whole code. To avoid this, we should consider the points mentioned as under:

- Ensure code or functions are not duplicated.
- No long functions or methods
- Removing unnecessary variables
- Proper use of access modifiers etc.

By refactoring, the programmers look to improve the overall code quality and make it more readable without altering its behavior.

#6 Pair programming

Pair programming consists of two programmers operating on the same code and unit test cases, on the same system (one display and one keyboard). One programmer plays the pilot role and focuses on clean code, and compiles and runs. The second one plays the role of navigator and focuses on the big picture and reviews code for improvement or refactoring.

Every hour or given a period of time this pair is allowed to switch roles so that the pilot will play the role of navigator and vice versa.

The pairs of pilots and navigators are also not fixed and they are frequently swapped, the main benefits of that over a period of time is that everyone gets to know about the code and functionality of the whole system.

XP core practices …

#7 Collective code ownership

By following pair programming practices the XP team always takes collective ownership of code. Success or failure is a collective effort and there is no blame game. There is no one key player here, so if there is a bug or issue then any developer can be called to fix it.

#8 Continuous Integration

In XP, Developers do pairs programming on local versions of the code. There is a need to integrate changes made every few hours or on a daily basis so after every code compilation and build we have to integrate it where all the tests are executed automatically for the entire project.

If the tests fail, they are fixed then and there, so that any chance of defect propagation and further problems are avoided with minimum downtime.

XP core practices …

#9 Small release

A cross-functional team in XP, releases Minimum Viable Product (MVP) frequently. Small releases also help to breakdown complex modules into small chunks of code. This helps the developer team as well as the on-site customer to demonstrate the product and focus only on the least amount of work that has the highest priority.

#10 System Metaphor

This is mainly related to user stories; the stories must be simple enough to be easily understood by user and developers and to relate it with code.

It could be a naming conversion practice used in design and code to have a shared understanding between teams. For example Order_Food() is easily explained -- this will be used to order food.

It easy for the team to relate to the functionality of the particular component by just looking at its name.

#11 Onsite Customer

This is a role similar to Product Owner in Scrum. Onsite customer plays a major role here and is responsible for crafting the vision, defining user stories and acceptance criteria, the definition of "done" and release planning.

They are the experts who know the domain or product and know how to generate a return of investment (ROI) by delivering the minimum viable product (MVP).

If the onsite customer role is not full time, the role can be filled with product managers, product owners, UI-UX designers and business analysts who are called proxies.

The word "on-site" implies that the customers or their representatives sit with the rest of the team to ensure that communication flows freely.

XP core practices …

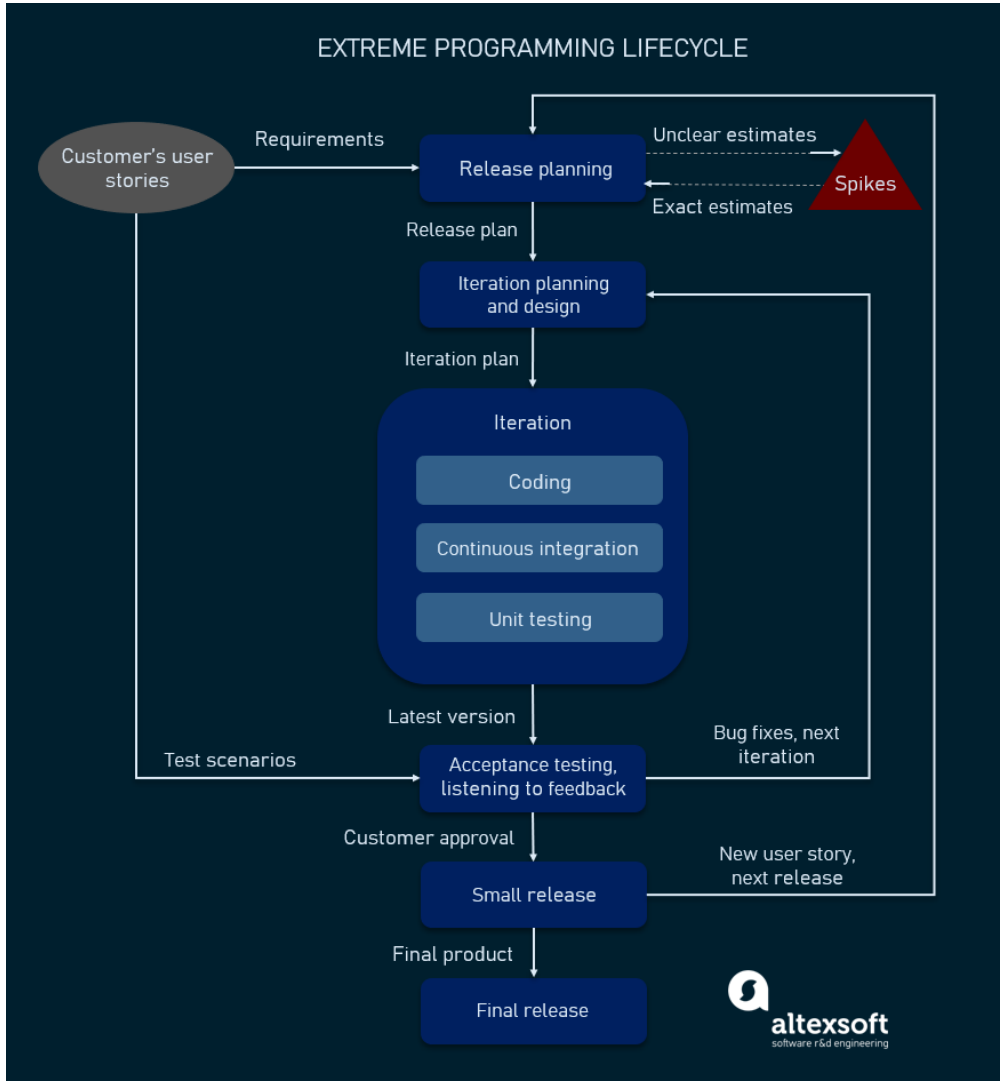
#12 Sustainable pace

This is a people-centric practice. In XP practices like TDD, continuous integration and refactoring of code help to proactively improve the quality and stability of the working software.

XP maintains a sustainable pace by introducing down time during the iteration. The team is not doing actual development at this time but acts as a buffer to deal with uncertainties and issues. Teams can use the slack time to pay down technical debt by refactoring code or do research to keep up the pace.

Software Engineering

XP Lifecycle



Lean Agile

Lean-Agile is a set of principles and practices for working that aims to minimise waste whilst maximising value.

This enables organisations to make quality a priority in their products and services.

Lean Agile principles

Eliminate waste

What's waste? It's anything that doesn't "add value to a product, value as perceived by the customer." Waste is anything produced or sitting around that doesn't get used. It can be "[when] developers code more features than are immediately needed." The time lost shifting development from one group to another? That's waste too.

Amplify learning

Tackling a software solution as a lot like trying to put together a new recipe for a top restaurant dish. A chef will iterate and learn from the variations produced. Software development is more complex, and teams can be big, but when you amplify learning, it makes this discovery process possible.

Lean Agile principles …

Decide as late as possible

In something like embedded software development, there can be many unknowns. Which processor is going to fit the product and users' demands best? Should we develop in x, y or z language? Is this screen better than that screen? Functionality likely to change due to user or market feedback? There's no value in speculating. Instead, delay decisions until you have the facts.

Deliver as fast as possible

"Design, implement, feedback, improve," and repeat. "Rapid development" is entirely plausible; its speed enables you to make informed decisions with real feedback. Keeping this cycle short, amplifies learning as well as making sure "customers get what they need now, not what they needed yesterday."

Lean Agile principles …

Empower the team

The people doing the work are the ones who understand the details the best. Want “excellence”? Make sure your technical team are involved in “the details of technical decisions”. Enable these teams to use “pull techniques” and “local signalling mechanisms” (like Kanban) to schedule and complete work whilst letting everyone know what’s coming up. So long as a leader guides the team, “they will make better technical decisions and better process decisions than anyone can make for them.”

Lean Agile principles …

Build integrity in

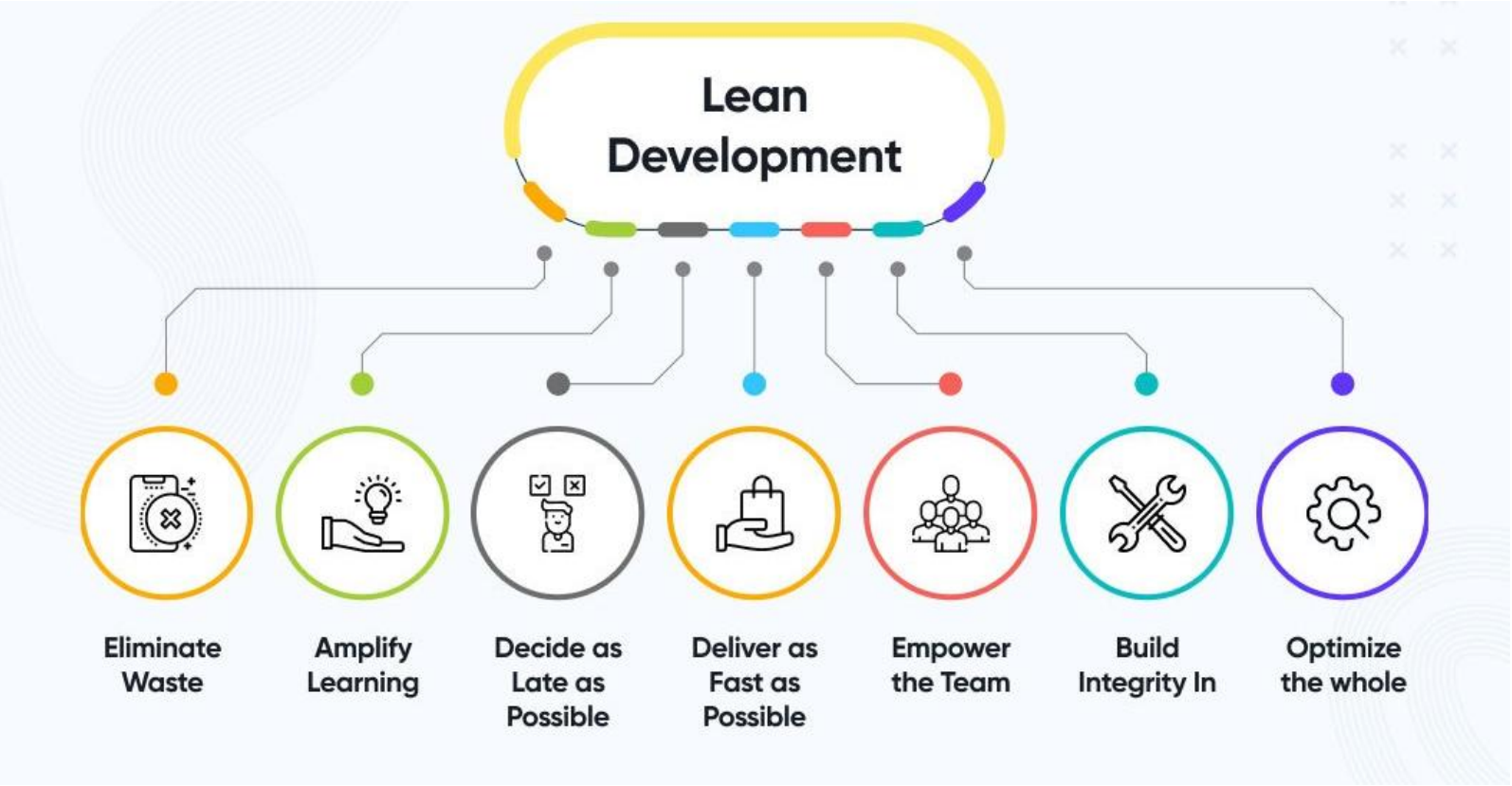
Here the focus is on ensuring quality. This is more than just customers being happy with the final product, its “perceived integrity”. And it’s going further than “conceptual integrity” where the whole works seamlessly together. For software, building integrity in means it can evolve smoothly over time, because it has “coherent architecture, scores high on usability and fitness for purpose, and is maintainable, adaptable, and extensible.” For the team, integrity comes “from wise leadership, relevant expertise, effective communication, and healthy discipline”.

Optimize(See) the whole

Don’t just look at one area of a project. Step back and look at all areas involved. “Quite often, the common good suffers if people attend first to their own [specialised] interests.” Instead of measuring based on people’s “[specialised] contribution”, look instead to the overall performance of the project.

Software Engineering

Lean Agile - principles



Component Based Software Engineering (CBSE)

Component-based software engineering approach is a reuse based approach to define, implement or select off-the-shelf components and integrate/compose loosely coupled independent components into systems.

Why CBSE?

- Increase in complexity of systems
- Reuse rather than re-implement and hence shorten development time.

Advantages of CBSE

- Increases productivity and reduces cost.
- Reduces development time
- Black-box usage of components
- Improves quality

Component Based Software Engineering (CBSE)

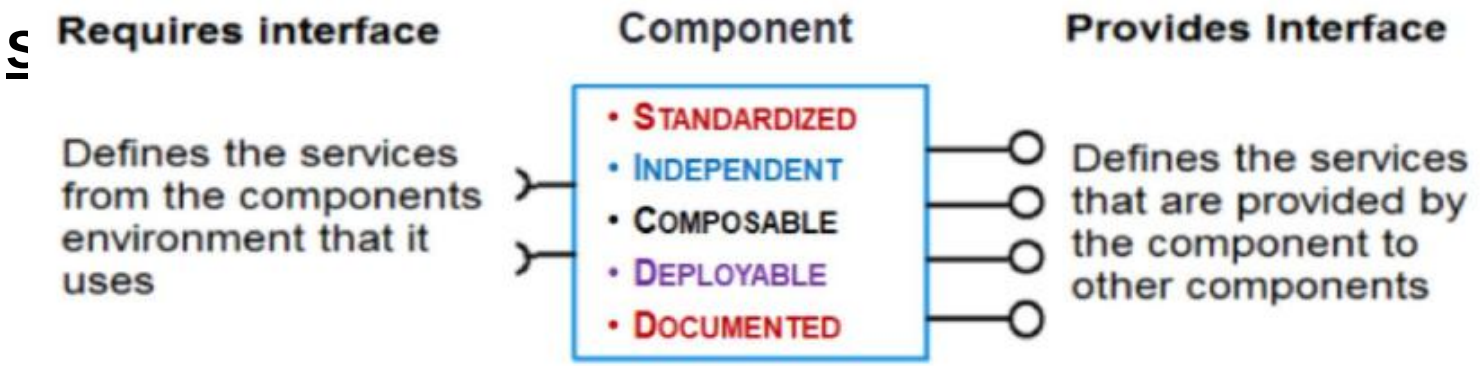
Disadvantages of CBSE

- Requirement trade-off
- Component certification
- Emergent property prediction
- Trusting components

Software Engineering

Essentials of CBSE

- Independent components that are completely specified by the public interfaces
- Component standards that facilitate the integration of components
- The component interface is published and all interactions are through the published interface.
- Middleware that provides software support for component integration
- Development process that is geared up to CBSE

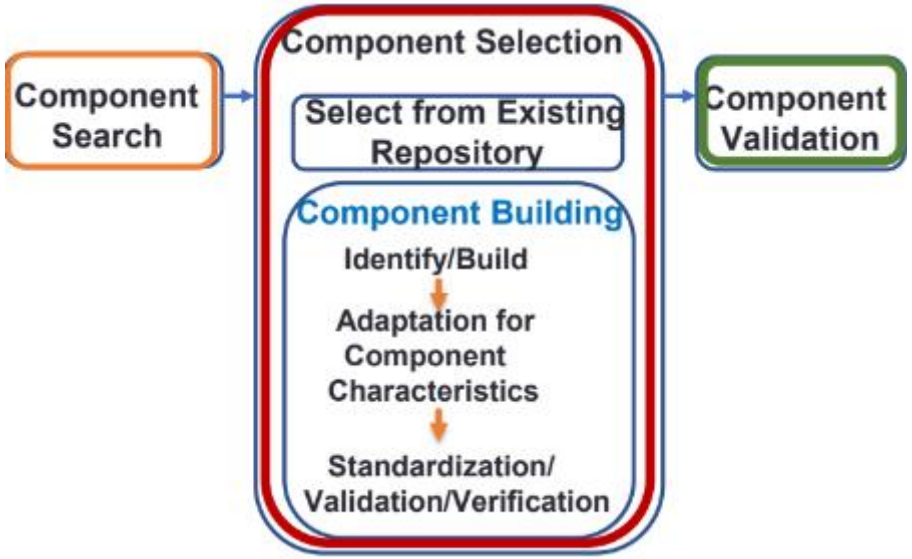


Essentials of CBSE

"A software component is a *unit of composition* with contractually specified *interfaces* and explicit *context dependencies* only. A software component can be deployed independently and is subject to composition by third parties.".

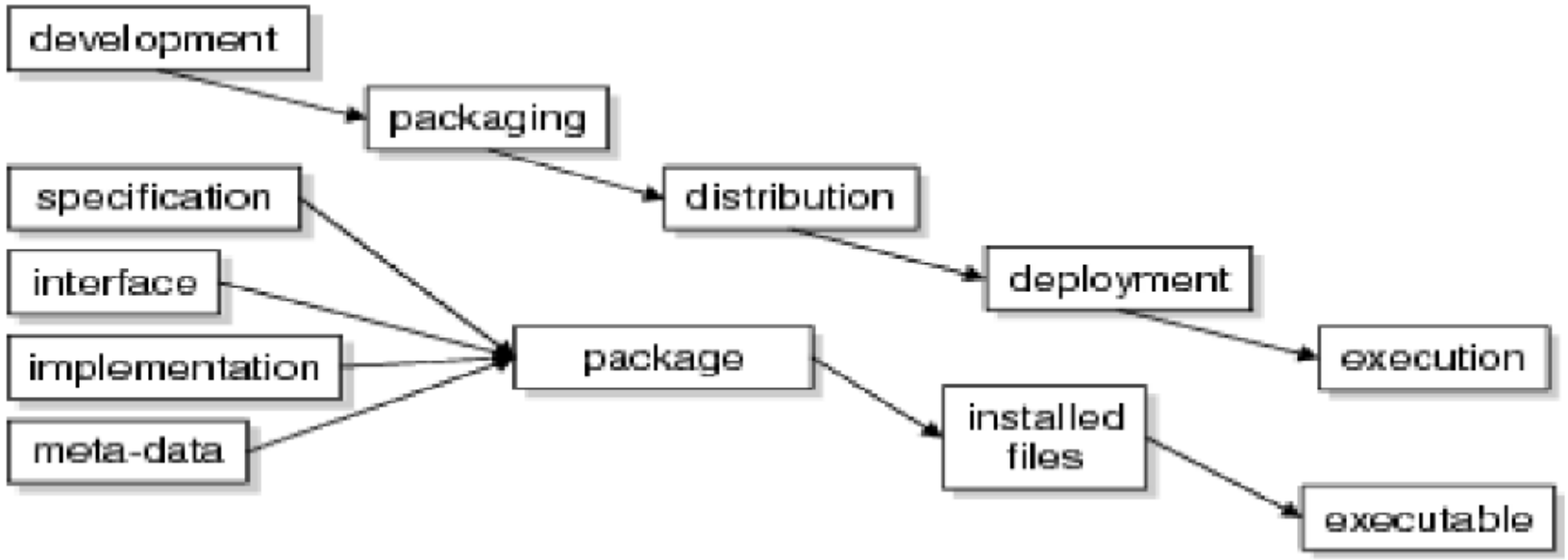
Identifying a component

Search for component -> Select Component -> Compose component from existing ones -> Adapters or “glue” to reconcile different component interfaces -> Validate the component



Software Engineering

Component Development Stages



Different forms of component representation:
During development using UML
When packaging - .ZIP
In the execution stage – blocks of code and data

Software Engineering

Component characteristics

Component characteristic	Description
Standardized	Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment.
Independent	A component should be independent—it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a 'requires' interface specification.
Composable	For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes.

Software Engineering

Software Product Lines

A product line is **a group of connected products marketed under a single brand name by the same company**. Firms sell multiple product lines under their various brand names, often differentiating by price, quality, country, or targeted demographic.

When similar products are developed, we may hope to reuse elements from earlier products during the development of new products. Such is not the habit in software development though. In many an organization there is no incentive to reuse elements (code, design, or any other artefact) from another system since that is not what we are being paid for. Similarly, there is no incentive to produce reusable elements, since the present project is all that counts.

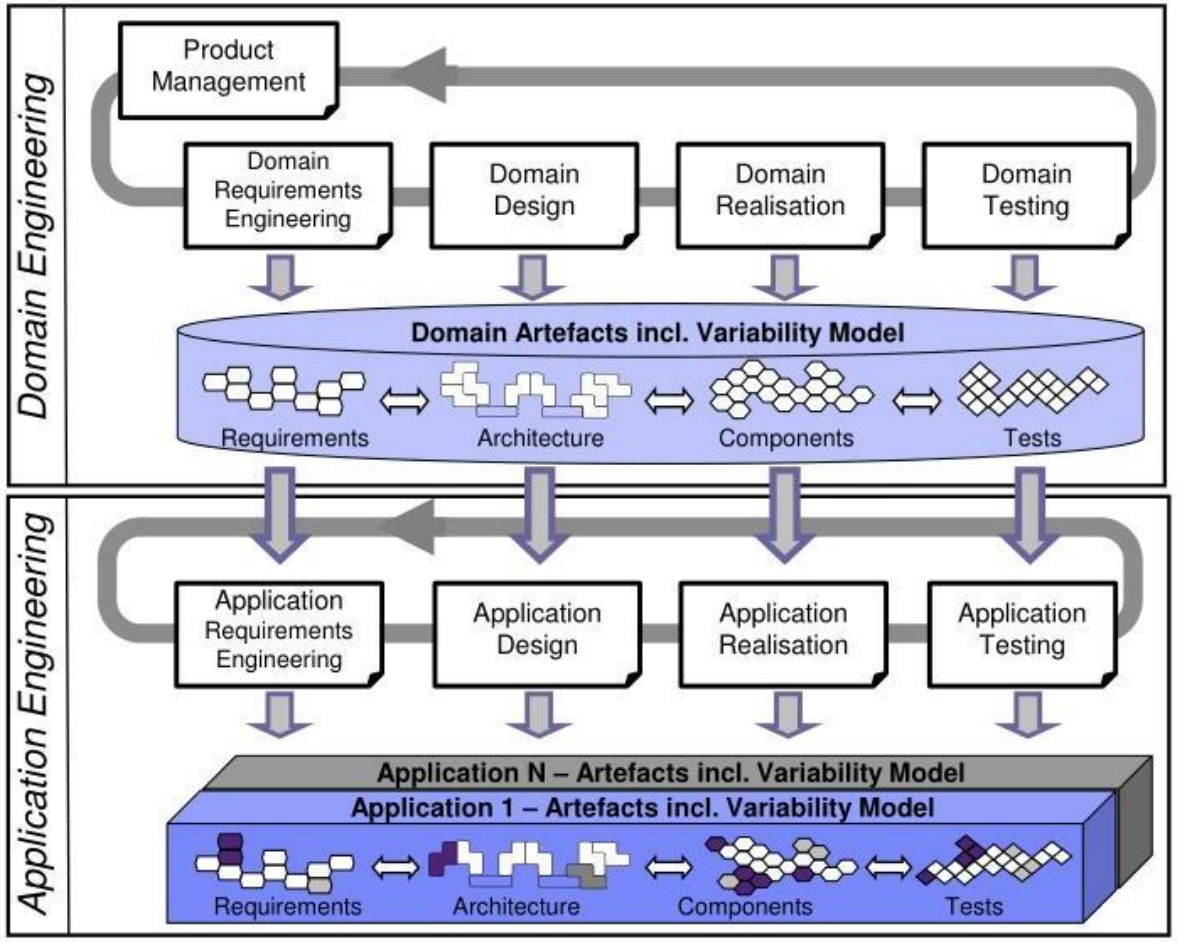
As an alternative, we may conceive of the notion of a *software product line*, a set of software systems that share elements. In a software product line, reuse is planned, not accidental. To keep the scope within reasonable boundaries, this planned reuse is tied to a given domain.

Software Engineering

Software Line Engineering Framework

Domain Engineering:
Define and realize
the commonality
and variability.
The goal is to establish
a reusable platform

Application Engineering:
Reuse domain artifacts,
exploiting variability to build
a product.
The goal is to derive a
product from the platform
established in the Domain
Engineering phase



Software Engineering

Requirements Engineering

Requirement is the property which must be exhibited by software developed/adapted to solve a particular problem.

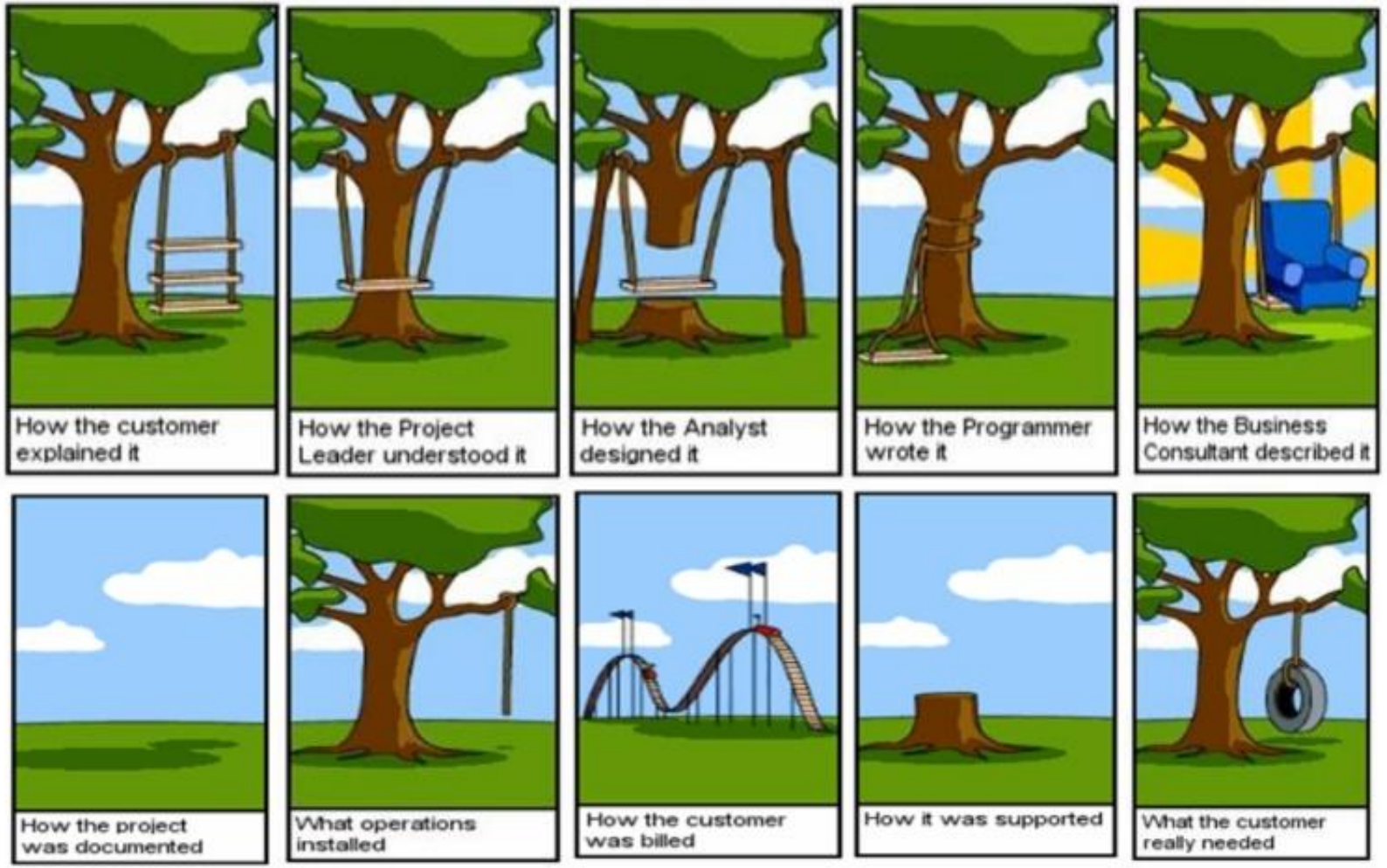
Requirement should specify the externally visible behavior of what and not how.

Requirement engineering is the first step in any software intensive development lifecycle irrespective of model

- Difficult, error prone and costly
- Critical for successful development of all down stream activities
- Requirement errors are expensive to fix

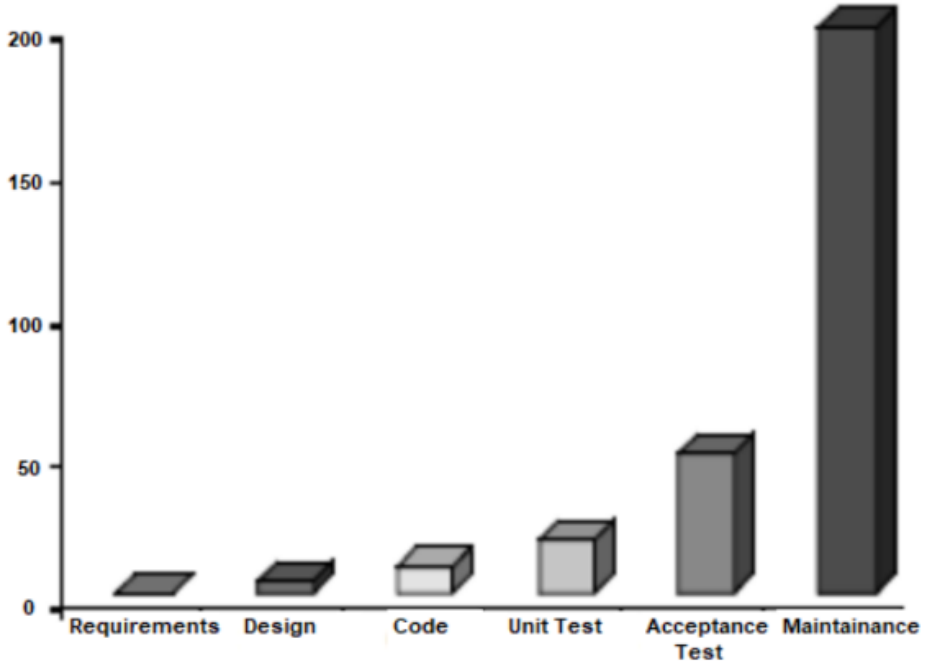
Software Engineering

Parody !



Software Engineering

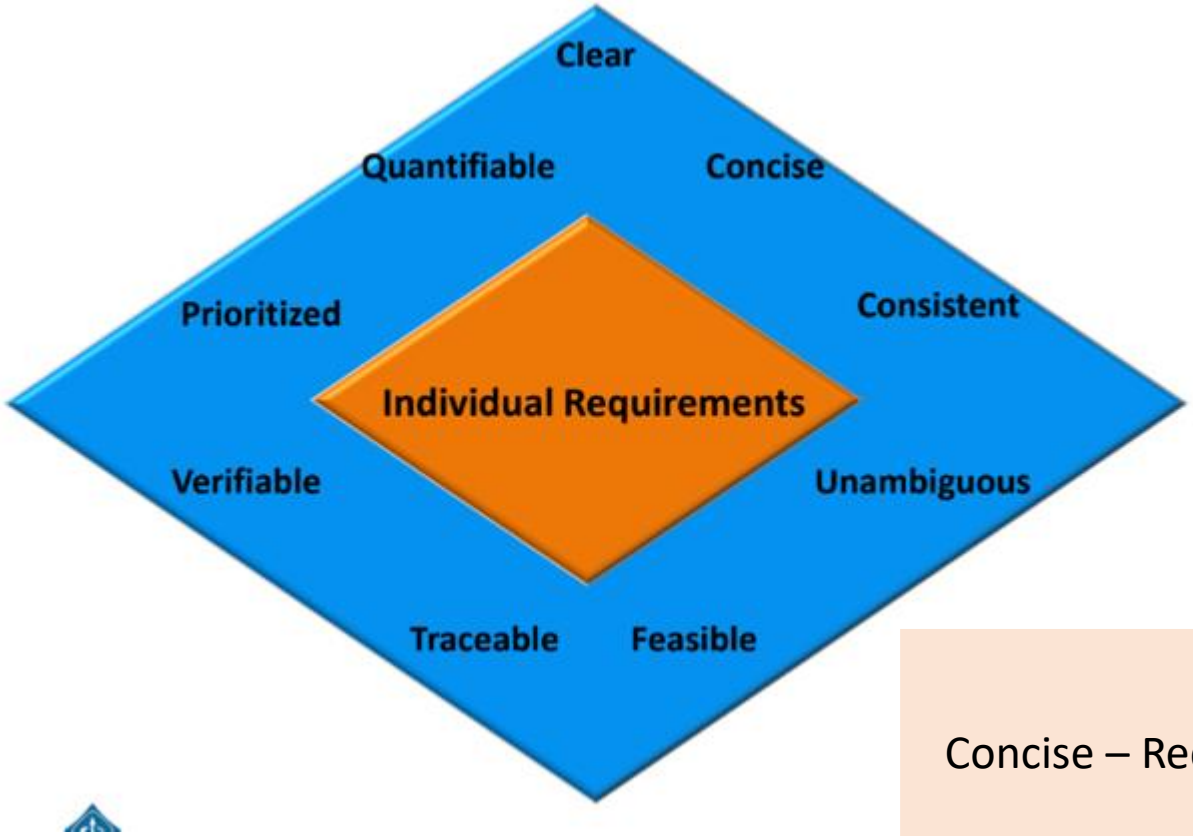
Cost of repair as a function of time



Life Cycle Stage	Relative cost of Repair
Requirements	0.1 to 0.2
Design	0.5
Coding	1
Unit test	2
Acceptance test	5
Maintenance	20

Software Engineering

Properties of requirements



Concise – Requirements should describe a single property

See more details here:
https://www.informit.com/articles/article.aspx?p=1152528&seqNum=4

Software Engineering

Requirements Engineering – In-class exercise

Use the properties of requirement to transform the given sentences into requirements.

All screens must appear quickly on the monitor

When the user accesses any screen, it must appear on the monitor within 2 seconds (Clear, Concise, Unambiguous, Verifiable, Measurable)

The replacement control system shall be installed with no disruption to production

The replacement control system shall be installed causing no more than 2 days of production disruption (Feasible)

The system must generate a batch end report and a discrepancy report when a batch is aborted

The system must generate a batch end report when a batch is completed or aborted

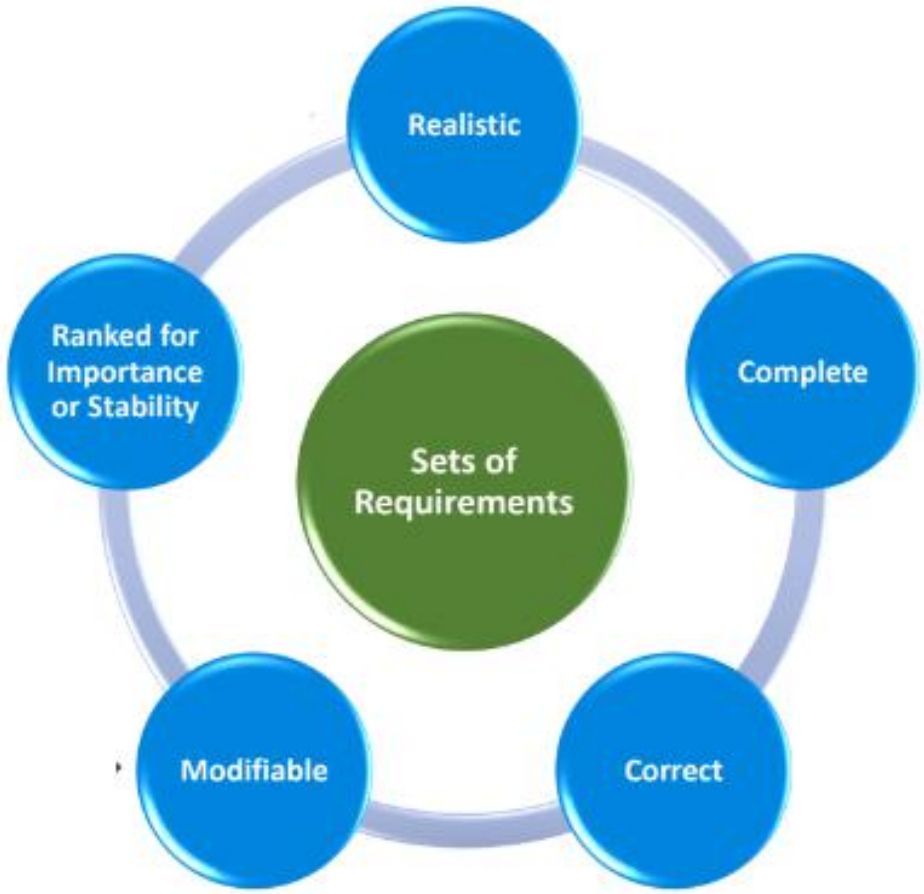
The system must generate a discrepancy report when a batch is aborted (Traceable)

The system must be user friendly

The user interface shall be menu driven. It shall provide dialog boxes, help screens, radio buttons, dropdown list boxes, and spin buttons for user inputs (Verifiable)

Software Engineering

Properties of a set of requirements



When there are many requirements but limited time or budget, choices must be made about which to include or exclude. Factors such as:

- changes in customer needs
- improved developer understanding of the products
- changes in organizational policy

will affect the stability of requirements.

Software Engineering

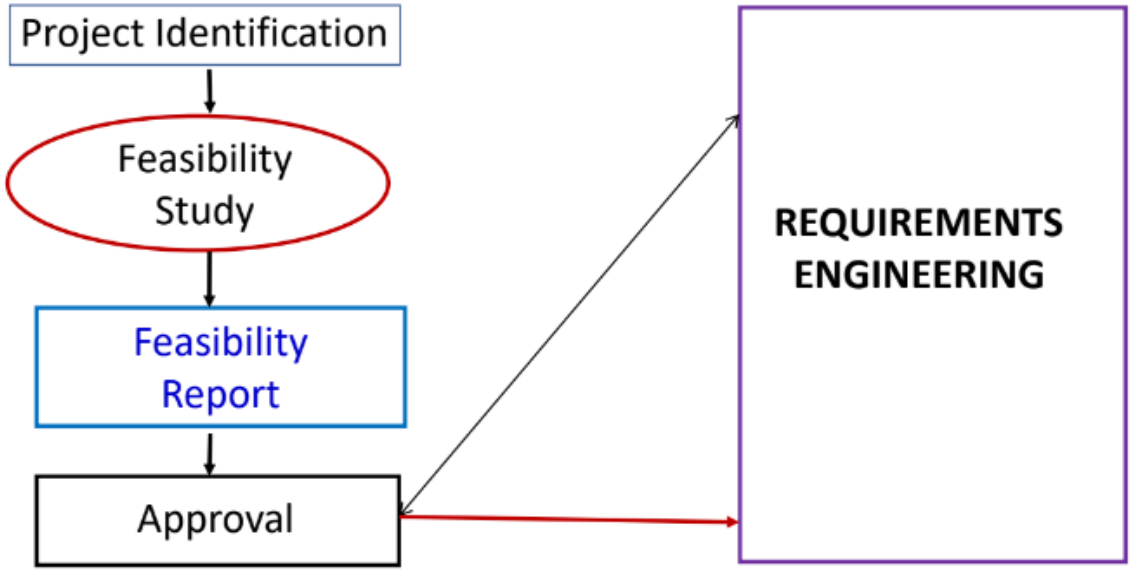
Feasibility study

What is Feasibility Study?

Short, low-cost study to asses the practicality of the project and whether it should be done.

When is Feasibility Study conducted?

Mostly done before beginning a project



Software Engineering

Activities in feasibility study

ACTIVITIES IN FEASIBILITY STUDY

- Figure out the client or the sponsor or the user who would have a stake in the project
- Find the current solution to the problem
- Find the targeted customers and the future market place
- Potential benefits
- Scope
- High level block level understanding of the solution
- Considerations to technology
- Marketing strategy
- Financial projection
- Schedule and high level planning and budget requirements
- Issues, assumptions, risks and constraints
- Alternatives and their consideration
- Potential project organization

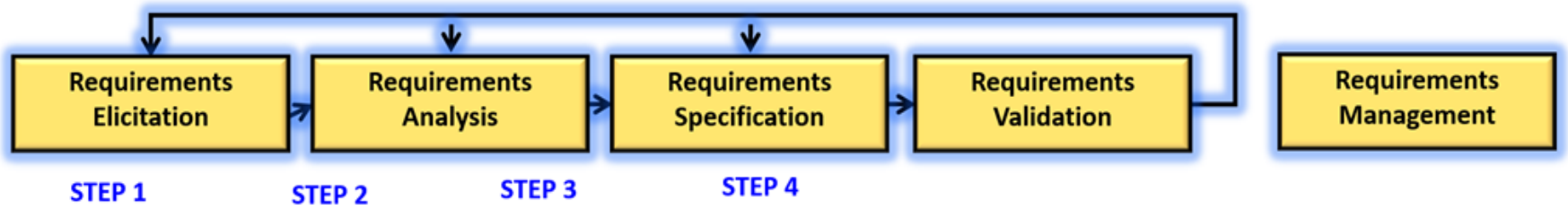
Ends with GO or NO-GO

Software Engineering

Requirements Engineering process

A “four + one” set of activities to produce specifications or requirements

It is an iterative process



Requirements Validation – Helps ensure the right requirements are realized

Software Engineering

Requirements Elicitation

It is the process of working proactively with all stakeholders gathering their needs, articulating their problem, identifying and negotiating potential conflicts thereby establishing a clear scope and boundary for a project.

It involves:

- Understanding the problem
- Understanding the domain
- Identifying clear objectives
- Understanding the needs
- Understanding constraints of the system stake holders
- Writing business objectives for the project

Software Engineering
Elicitation techniques

Approach is based on:

- Nature of the system being developed
- Background and experience of stakeholders

Elicitation techniques – **Active** and **Passive**

<u>Active elicitation techniques</u>

Ongoing interaction between the stake holders and users.

- Interviews
- Facilitated meetings
- Role-playing
- Prototypes
- Ethnography
- Scenarios

Software Engineering

Elicitation techniques …

Passive elicitation tecniques

Infrequent interaction between the stake holders and users.

- Use cases
- Business process analysis & modeling
- Workflows
- Questionnaires
- Checklists
- Documentation

Software Engineering

Process of Requirements Analysis

1. Understand requirements in depth
2. Classify requirements into coherent clusters
3. Model the requirements
4. Analyze requirements using fish bone diagram
5. Recognize and resolve conflicts
6. Negotiate requirements
7. Prioritize requirements – MoSCoW (**Must have, Should have, Could have, Won't have)**
8. Identify risks
9. Decide on build or buy – COTS solution.

Process of Requirements Analysis …

Understand requirements in depth.

Classify requirements into coherent clusters:

1. Functional requirements
2. Non-functional requirements
3. User requirements
4. System requirements
5. Domain requirements

Process of Requirements Analysis …

Functional requirements

Functionality or services, the system should provide with different inputs, and expression on how the system should behave in particular situations.

Non-functional requirements

Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards.

User requirements

Statements in natural language plus informal context diagrams system/sub-system and their interconnections and operational constraints. Written for/by customers.

Process of Requirements Analysis …

System requirements

A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor. Also called Software requirements or Functional specifications

Domain requirement

Constraints on the system from the domain of operations.

Identify the class of requirement

System shall assign a unique tracking number to each shipment

Functional requirement

With 100 concurrent users a database record shall be fetched over the network in less than 3ms

Non-functional

Software Engineering

Process of Requirements Analysis …

Model the requirements

A model is a representation of a system in some form.
A is a model of B if A can be used to answer questions about B.

Primary goals of Modeling

- Providing an understanding of the existing System
- Communicating the requirements in terms of who, what and interpreting it in the same way

Models could be ***Structural Models*** and ***Behavioral models***

Structural Models

- Captures static aspects of system
- What entities exist in the system?
- How are they related?

Example: Class diagram

Behavioral Models

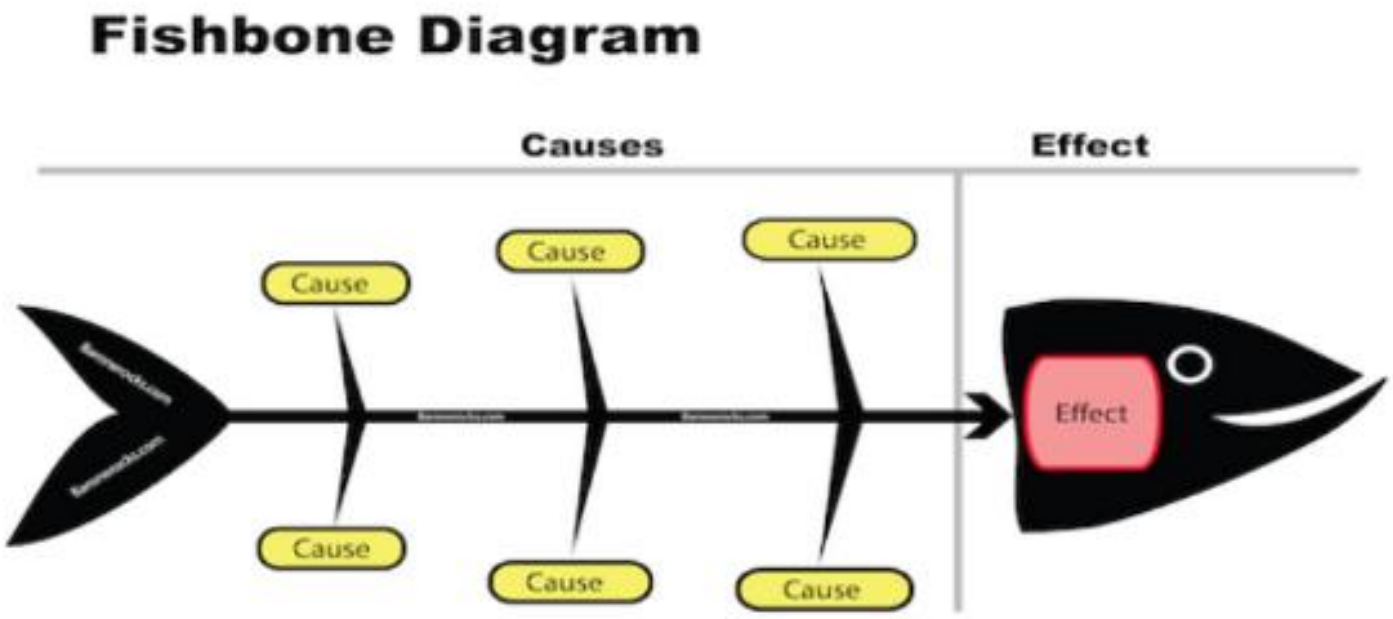
- Captures dynamic aspects of the system
- How do the entities interact in response to a stimulus?

Example: Use Case diagram

Process of Requirements Analysis …

Use-case models are the most popular models used during analysis

Analyze requirements using fish bone diagram
List out all the reasons/causes on why the requirement (effect) has come in and become relevant.



Process of Requirements Analysis …

Recognize and resolve conflicts
Functionality vs Cost vs Timelines

Negotiate requirements

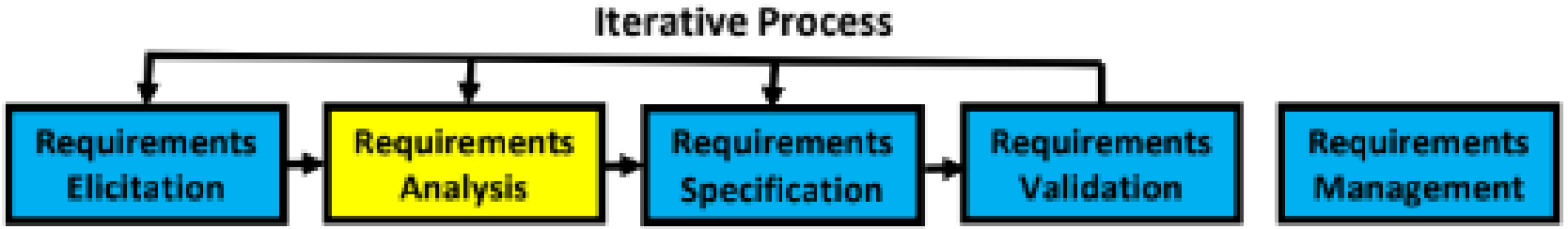
Prioritize the requirements (MoSCoW -Must have, Should have, Could have, Won't have)
Pareto Analysis (80-20 to focus on vital few to trivial many)
Vital Few (Prioritize and needed critically) – Trivial Many (Lower priority) 80-20 Rule

Identify risks

Decide on build or buy – COTS solution
Commercial Off The Shelf solution – COTS

Software Engineering

Process of Requirements Analysis …



Unified Modeling Language (UML)

What is UML?

UML (Unified Modeling Language) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of complex software systems.

UML plays an important role in defining different perspectives of a system

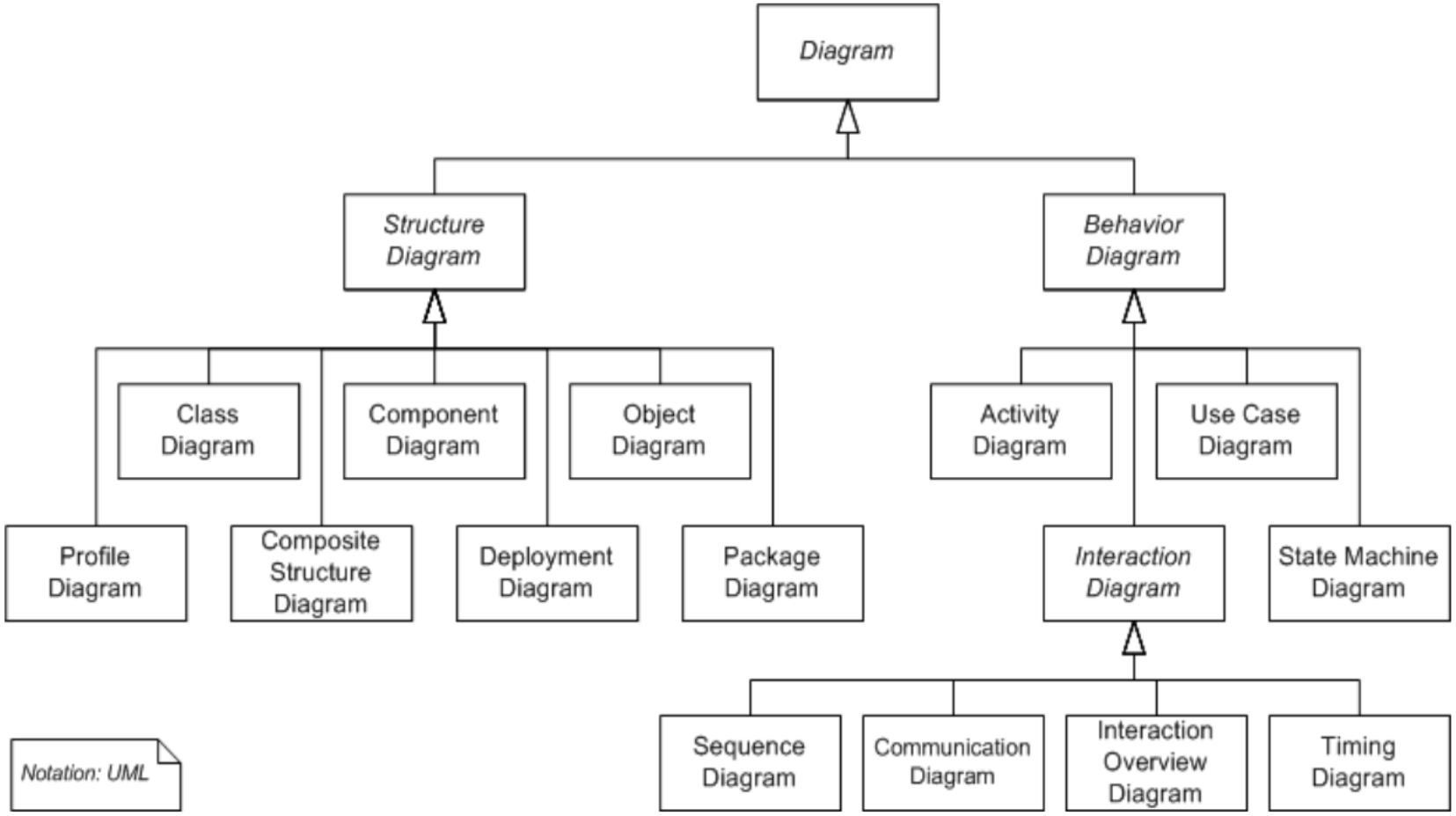
Requirements Design Implementation Deployment

Why is UML used?

UML Use-case models are predominantly used with Modeling Systems, to discuss the dynamic behavior of the system when it is running/operating. Its often used to used to gather the requirements of a system including internal and external influences.

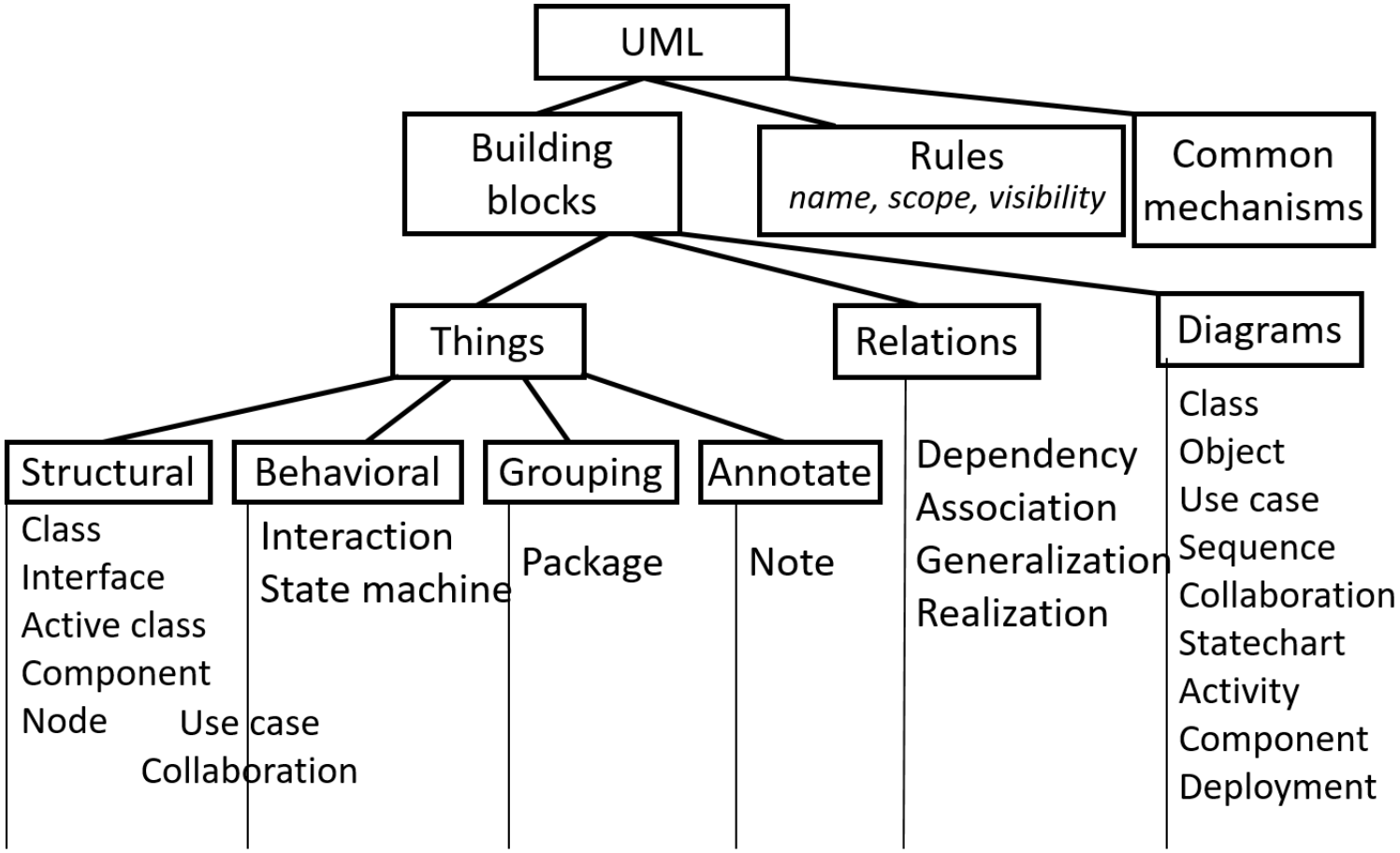
Software Engineering

UML



Software Engineering

Conceptual UML model



Software Engineering

Using Use Case diagram

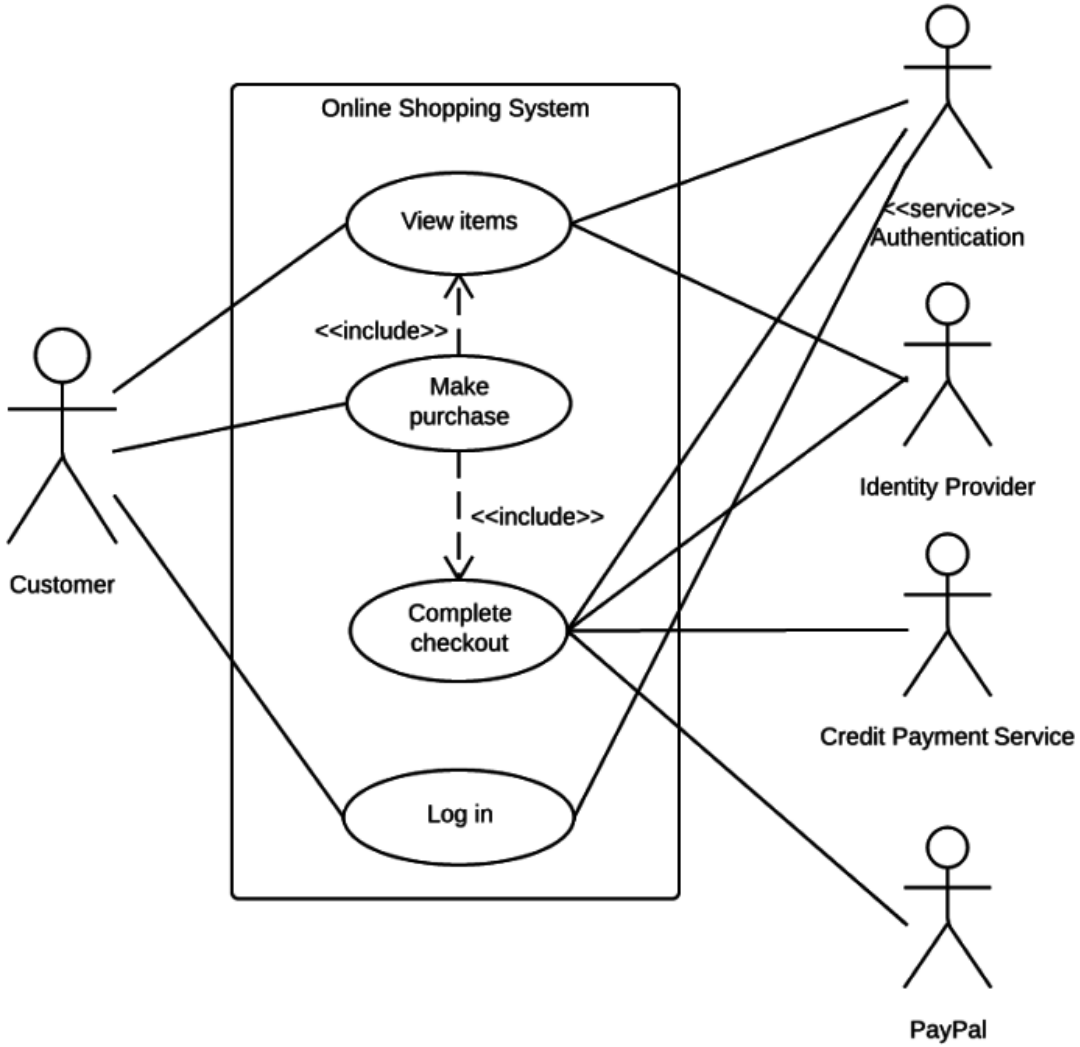
Use case from a user's point of view outlines how the proposed system will perform a task expected to be performed, while responding to a request or task of a role/actor/user.

Use case diagrams are used to visualize, specify, construct, and document the (intended) behavior of the system, during requirements capture and analysis. Used by developers, domain experts and end-users.

Actor is someone (can be a human or other external system) interacting with the use case (system function), named by noun but is not part of the system.

Software Engineering

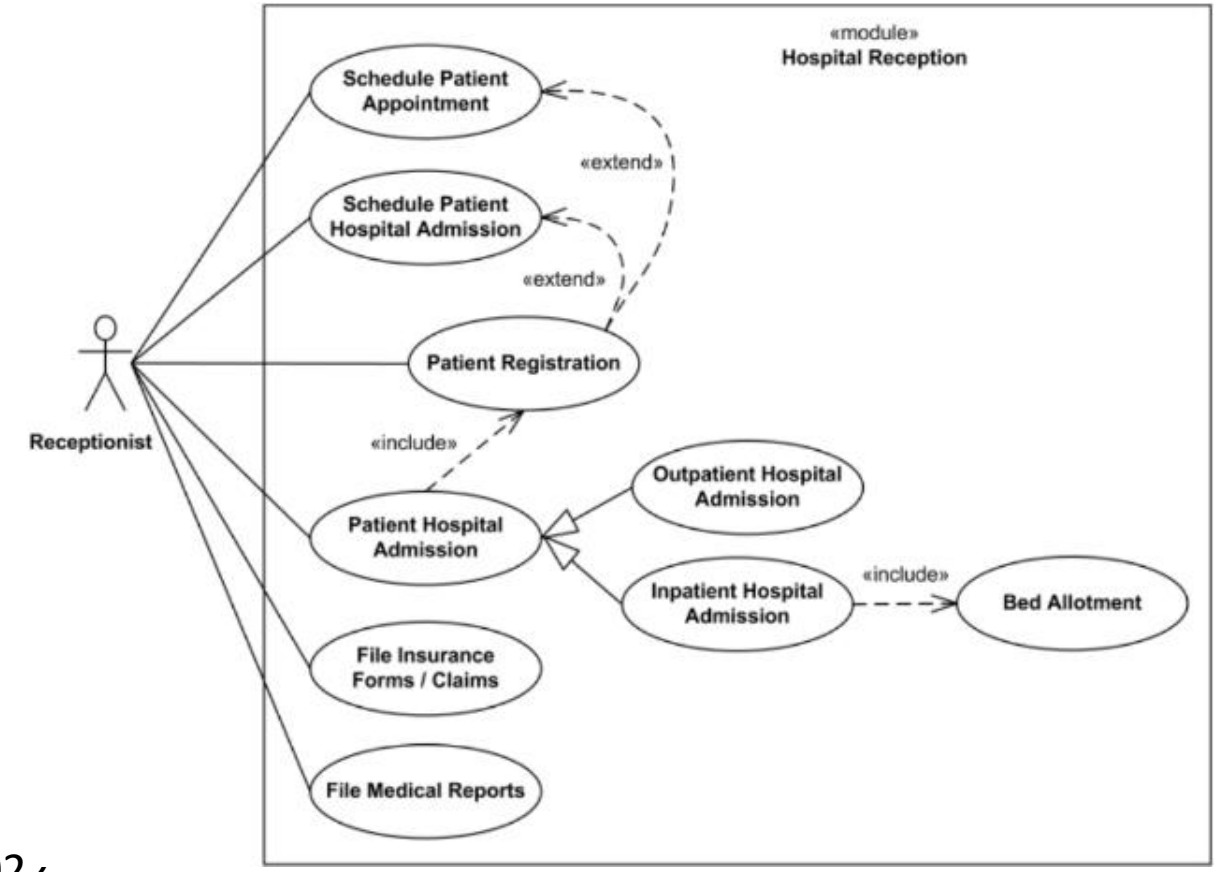
A sample Use Case diagram



Software Engineering

Use Case diagram – In-Class Exercise

Using Use Case diagram, to depict the job of a hospital receptionist. Include scheduling appointments, admissions, bed allotment, filing insurance and filing medical reports as some of the use cases.



Software Engineering

Requirement Specification

After elicitation and analysis, we need to specify the requirements.

Requirements specification is the documentation of a set of requirements that is reviewed and approved by the customer and provides direction for the software construction activities in the next stage of the life cycle.

The **software requirements specification (SRS)** document is the basis for customers and contractors/suppliers agreeing on what the product will and will not do. It describes both the functional and nonfunctional requirements

Software Requirement Specification (SRS)

Functionality: What is the software supposed to do?

External interfaces: How does the software interact with people, the system's hardware, other hardware, and other software?

Non Functionality: This includes all of the Quality criteria which drive the functionality. Example: Performance, Availability, Portability etc.

Design constraints imposed on an implementation:

- Required standards in effect
- Implementation language
- Policies for database integrity
- Resource limits
- Security
- Operating environment(s) etc.

Software Engineering

An SRS template

ToC recommended by IEEE for SRS (IEEE Std. 830-1998)

1. Introduction
1.1 Purpose
1.2 Scope
1.3 Definitions, acronyms, and abbreviations
1.4 References
1.5 Overview
2. Overall description
2.1 Product perspective
2.2 Product functions
2.3 User characteristics
2.4 Constraints
2.5 Assumptions and dependencies
3. Specific requirements
3.1 External Interface
3.2 Functional Requirements
3.3 Non-Functional Requirements
3.4 Design Constraints
Appendixes
Index

Software Engineering
Requirements validation

The purpose of requirements is to help ensure that the requirements do what the customer wants. This is an important phase because repairing requirement errors in downstream phases can be expensive.

VALIDATION & VERIFICATION

Validation determines whether the software requirements if implemented, will solve the right problem and satisfy the intended user needs

Verification determines whether the requirements have been specified correctly

(Reviews are used for both validation and verification)

Software Engineering
Requirements validation

Requirement reviews

Software Engineering
Requirement validation

Prototyping

Prototype facilitates user involvement during requirements engineering phase and ensures engineers and users have the same interpretation of the requirements.
Prototyping is most beneficial in systems – With many user interactions
Example: Design of online billing systems

Systems with little or no user interaction may not benefit as much from prototyping.
Example: Batch processing

Model Validation

- Ensuring that the models represent all essential functional requirements
- Demonstrating that each model is consistent in itself
- Usage of the Fish Bone Analysis technique for validation

Acceptance Criteria

To check if there are requirements matching with that the Acceptance criteria.

Requirements Management

Requirements specification is the baseline on which the future lifecycle phases will need to build upon.

Can you think of reasons why requirements might change?

- Better understanding of the problem
- Customer internalizing the problem and solution
- Evolving environment and technology landscape

Facets of Requirements Management

1. Ensuring that the requirements are all addressed in each phase of the lifecycle
2. Ensuring that the changes in the requirements are handled appropriately

Software Engineering

Requirements Traceability Matrix (RTM)

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

Requirements are traced across the SDLC using the requirement traceability matrix (RTM)

- Forward Tracing
- Backward Tracing

Every phase of the SDLC progressively fills the RTM

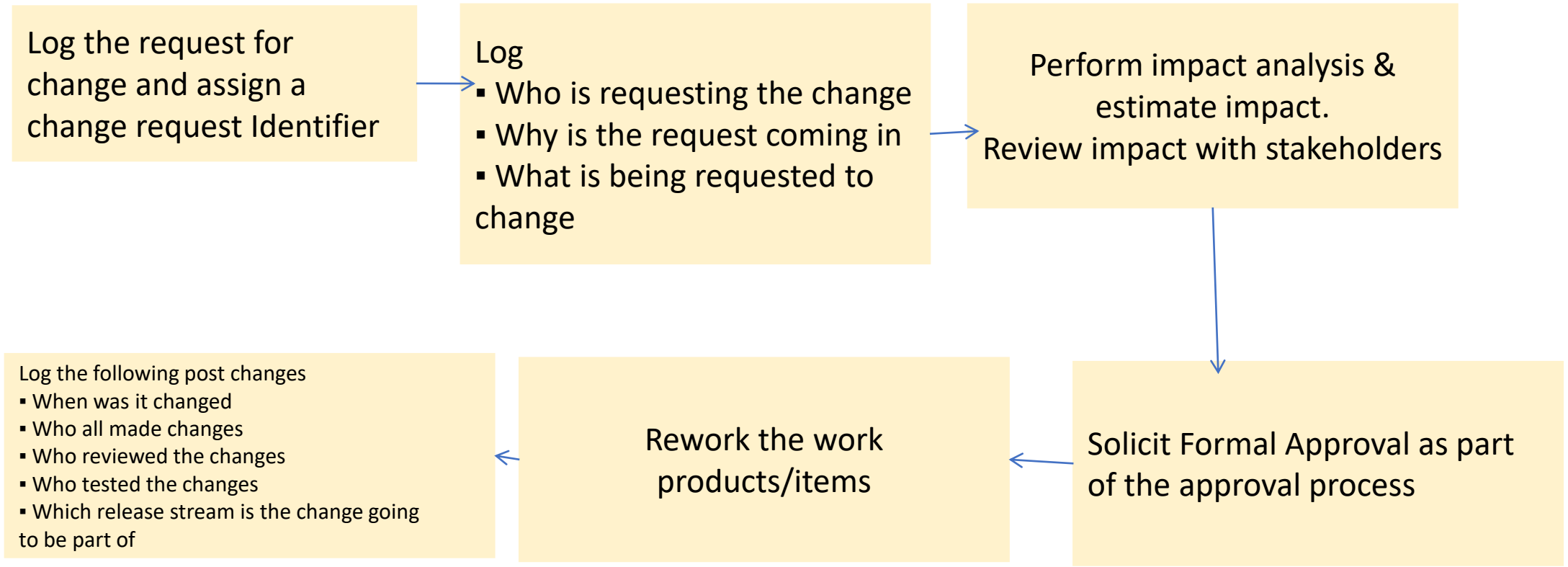
Requirements Change Management

Change in the requirements have impacts on plans, work products etc.

Uncontrolled changes can have a huge adverse impact on project in terms of cost, schedule, quality and expectations.

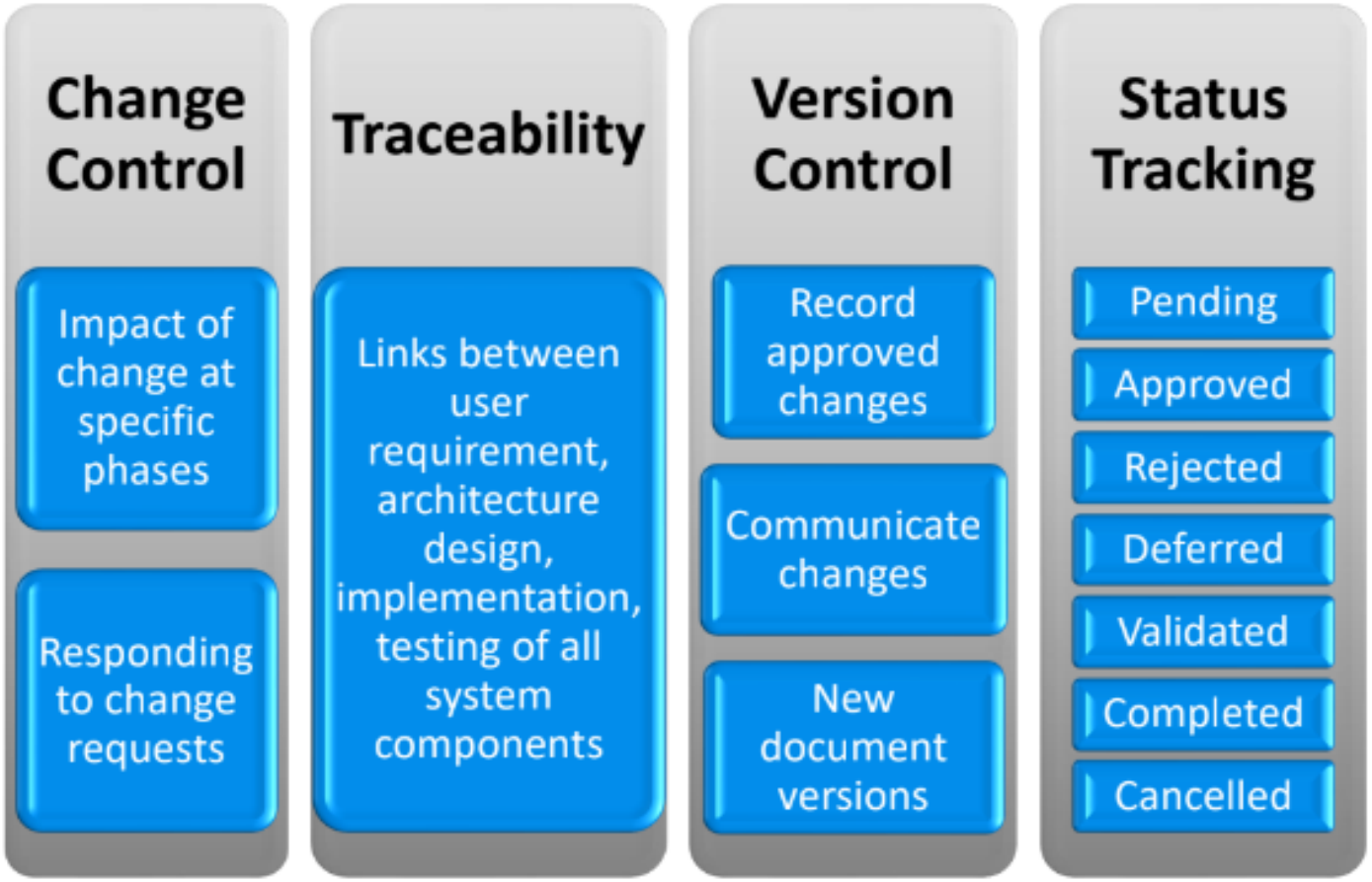
In the perspective of managing the changes, change requests go through a formal change management process.

REQUIREMENT CHANGE PROCESS



Software Engineering

Requirements Change Management



THANK YOU

M S Anand
Department of Computer Science Engineering

anandms@yahoo.com