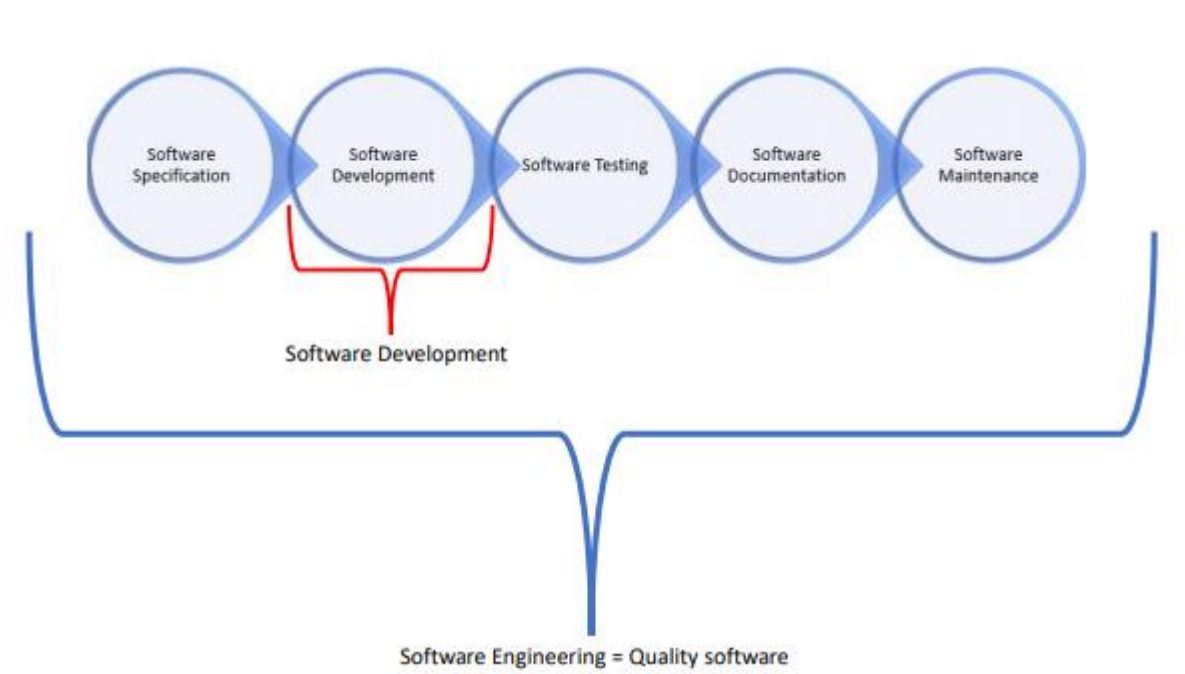
ADVANCED SOFTWARE DEVELOPMENT I & II
CSC3003S

Software Systems Engineering



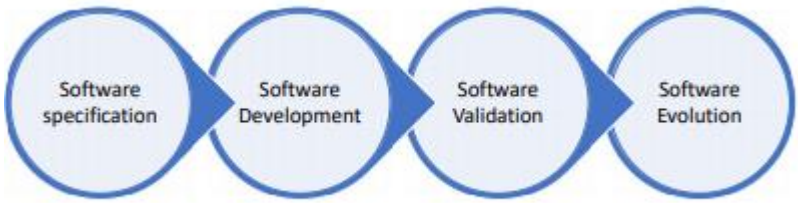
- SW engineering is not just concerned with the technical processes of sw development
    - It includes activities such as software project management, the development of tools, methods, and theories to support software production
- Using sw engineering principles leads to better quality code

Advantages of using Software Engineering Principles

- Maintainability:
    - SW should be written in such a way so that it can evolve to meet the changing needs of customers
    - This is a critical attribute because sw change is an inevitable requirement of a changing business environment
    - i.e. Code written in such a way that it's easy to change if the clients needs change – won't need to refactor/rewrite the sw
- Dependability and Security:
    - SW dependability includes a range of characteristics including reliability, security, and safety
    - Dependable sw shouldn't cause physical or economic damage in the event of system failure
    - Malicious users should not be able to access or damage the system
- Efficiency:
    - SW shouldn't make wasteful use of system resources such as memory and processor cycles
    - Efficiency therefore includes responsiveness, processing time, memory utilization, etc
- Acceptability:

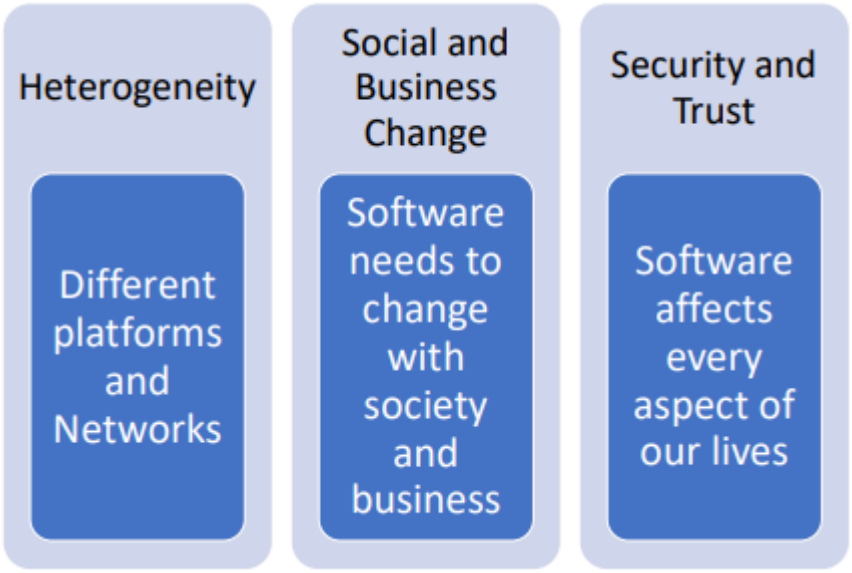
- SW must be acceptable to the type of users for which it is designed
- This means that it must be understandable, usable, and compatible with other systems that they use
- E.g. will design sw for children differently than for adults

Aspects of Software Engineering



- Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation
- Software development, where the software is designed and programmed
- Software validation, where the software is checked to ensure that it is what the customer requires
- Software evolution, where the software is modified to reflect changing customer and market requirements

Software Engineering Diversity



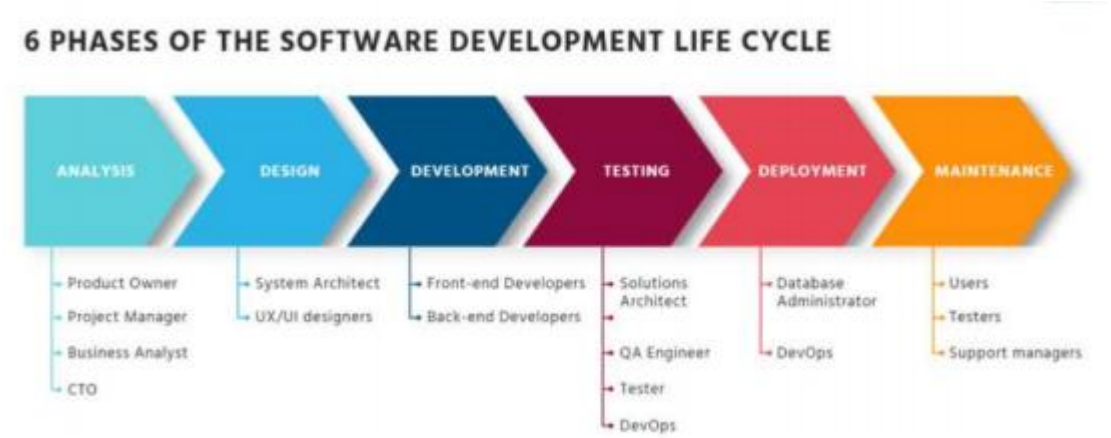
- Heterogeneity:
  - Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices.
    - Also have to operate across a range of OSes
  - As well as running on general-purpose computers, software may also have to execute on mobile phones.

    - You often have to integrate new software with older legacy systems written in different programming languages.
    - The challenge here is to develop techniques for building dependable software that is flexible enough to cope with this heterogeneity
- Business and social change:
    - Business and society are changing incredibly quickly as emerging economies develop and new technologies become available
    - They need to be able to change their existing software and to rapidly develop new software
    - Many traditional software engineering techniques are time-consuming and delivery of new systems often takes longer than planned
    - They need to evolve so that the time required for software to deliver value to its customers is reduced
- Security and Trust:
    - As software is intertwined with all aspects of our lives, it is essential that we can trust that software
    - This is especially true for remote software systems accessed through a web page or web service interface
    - We have to make sure that malicious users cannot attack our software and that information security is maintained

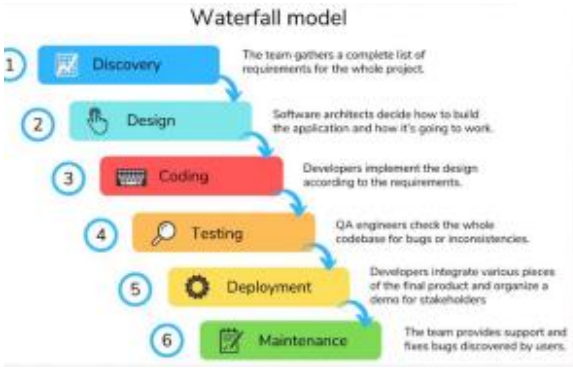
Process

- Use processes to manage heterogeneity, business and social change, as well as security and trust
- Use different software engineering techniques for each type of system because the software has quite different characteristics
- E.g. use different techniques for embedded systems vs entertainment systems
- But, when developing a sw system, one needs to adhere to a specific set of principles
- The following is true for developing any kind of software system:
    - Managed and understood development process
    - Focus on dependability and performance
    - Understand and manage software requirements
    - Effective use/ reuse of resources
- This is achieved by Planning, Communication and People, as well as Methods, Tools and Ethics

Methods



- Also have different sw development methodologies – within this development phase, different methodologies are employed in order to ensure good quality sw is created



Tools

- Case tools – tools that will take you from diagramming to maintenance tools

Ethics

- Confidentiality: respect the confidentiality of your employers or clients
- Competence: you should not misrepresent your level of competence
- Intellectual property rights: you should be aware of local laws governing the use of intellectual property such as patents and copyright
- Computer misuse: you should not use your technical skills to misuse other people's computers

Summary

- Take principles of engineering and apply them to sw development to ensure that good processes are followed in order to deploy quality sw that is maintainable, useable, dependable, acceptable and ensure clients happiness

Project Management

People

- Ensuring software is delivered on time and on schedule and in accordance with the requirements of the organisations developing and procuring the software
- Needed because software development is always subject to the budget and schedule constraints that are set by the organisation developing the software
- Systems development is a complex activity that requires careful project management
    - Why? Lots of dependencies, people and organising
    - Need integration – have different people working on different things
- Inter-dependencies between the artefacts of software development
    - Production has to be planned, monitored and co-ordinated
        - So, software development is efficient, effective and on time
        - This applies to development, testing and deployment
- Often involve many developers, some with specialized skills, who will be required at different times
    - E.g. may only need a graphic artist much later on in the project
- Activities must follow a particular sequence
    - E.g. can only test when have actually developed the components being tested!
    - Test scripts and test harnesses may be prepared earlier in the project
- There is no one technique that fixes the inherent complexity of software design and development
- Difficulties with building software can be divided (Aristotle) into
    - Essence: conceptual structure of the software itself
    - Accidents: difficulties from aspects of the process of realizing the conceptual structure in executable form
        - E.g. a customer forgets to mention a requirement that seems obvious to them
- Lots of projects start out seemingly innocent and straightforward, and then in the dark of the moon they turn into monsters (werewolves)!
    - Only if at least 9/10 of troubles are accidents can shrinking them to zero give you an order of magnitude improvement
        - And they're not
    - Therefore, any silver bullet (anything offering *10 improvement in productivity) must address the conceptual issues
        - May mean dealing with the concepts at a different level

Project Scope

- Ideally set from the beginning of the project
- Functions and features that are to be delivered to end-users
- Data that are input and output

    - i.e. data that is included in or produced by the system
- Content that is presented to users as a consequence of using the software
- The performance, constraints, interfaces and reliability that bound the system
    - Includes non-functional requirements
- How quickly does the project need to be done? Which features MUST be implemented and which are nice-to-haves?

Defining Scope

- Scope usually communicated by client – but essential to document it so that it is clear between your team and the client what is necessary to the system
- Scope defined by either:
    - Narrative description of software scope
        - After communication with stakeholder
    - Set of use-cases
        - Scenario-based description of the user's interaction with the software from the user's point of view
- Performance considerations
    - What are the processing and response time requirements?
- Constraints identify limits placed on the software:
    - Device limitations for end users or expectations about the number of users or transactions the system will take
    - External hardware, available memory, or other existing systems

Determining Feasibility

- Given the scope, determine the resources required
    - Not just about money, but whether the people with the right skills exist, whether the hardware to do it is available to the team and whether there are existing resources you might be able to leverage such as open-source software or even licensed libraries or tools
- Need an answer to the question – do we have the resources we need to build this software?
- Resources are:
    - People
    - HW and SW tools
    - Reusable components

Goals

- Take a step back from features and use cases, and look at overall goals
    - Or step even further back and clearly evaluate the perspectives of all the key stakeholders
- Goals are derived from the needs and expectations of stakeholders
- Stakeholder = someone (in)directly impacted by the project
    - Project sponsor;
    - Customer for the deliverables;
    - Users of the project outputs;
    - Project manager and project team

    - i.e. who are you going to incorporate into the process, and enquire with as you design the system?
- You may find that stakeholders have conflicting goals – and as a result, differing visions of scope and priorities.
    - This frequently needs to be negotiated as well.
    - But just as often – the only stakeholder's opinion that really matters is the one paying you (or giving you a mark)
- Find true needs that create real benefits
    - Prioritize them i.e. from there you can align features and priorities effectively

Smart Goals

- Create measurable goals:
    - S specific, significant, stretching
    - M measurable, meaningful, motivational
    - A agreed upon, attainable, achievable, acceptable, action-oriented
    - R realistic, relevant, reasonable, rewarding, results oriented
    - T time-boxed, time-based, time-bound, timely, tangible, trackable
        - i.e. goals need boundaries, need to be able to track when they're done, not done and partially done

Deliverables

- Deliverables: list of items that have to be delivered to meet the goals
    - Verifiable and specific
    - Can be report/equipment acquisition/an executable code module etc
    - Has a stakeholder who needs it
- Say when it is to be delivered
- Give quality standards, for example:
    - SW runs
    - Document is according to specified format
- Deliverable is finished and there's a clear expectation between all stakeholders as to what the deliverable will look like and what quality it's going to be

Schedule

- The effort required (time-wise)
- The people required (and other resources)
- Update deliverables with this & work schedule out
- If the schedule is unrealistic you can justify some of the following:
    - Renegotiate deadline – delay
    - Additional resources – more expensive
    - Reduce scope – fewer deliverables
- Negotiate a set of must-haves and nice-to-haves with client
    - Conservative schedule for must-haves set – have to ensure that those get done
    - If must-haves can be done, can rework schedule to try and deliver some nice-to-haves

- Don’t set yourself up for failure

Supporting Plans

- Human Resource Plan
  - Name key individuals & organisations: describe roles & responsibilities
    - i.e. what do we need to be able to achieve this set of deliverables?
  - Describe no. and type of people needed
    - Start dates, estimated duration and how to get them
  - What’re the key, required responsibilities and how do they align with the people already available to you?
- Communications and Management Plan
  - Who needs to be kept informed about the project?
  - How they will receive the information?
    - Weekly review meeting
    - Progress reports
    - Revised schedule
- Risk Management Plan
  - ID as many risks as possible
  - Be prepared if something bad happens

The Range of Management Activities

- People
  - Managers—Project Managers—Team Leaders—Software Team—Customers—End Users
- Product
  - Scope and decomposition
  - i.e. what features will be included, how it will be designed and who is doing which parts of the product?
- Process
  - Software Development life-cycle: Initiation →Analysis →Design →Construct →Test →Implement
    - How to monitor this process as one goes
  - Unified process: Inception →Elaboration →Construction →Transition →Production
- Project
  - Size estimation (of project); scheduling (within course of developing and maintaining the system); risk management; tracking

People

- A team needs people with different skills
- Range of duties even in a small project
  - Project management: Strategist, Leader, Politician, Project Facilitator, Administrator
  - Systems analysis: Stakeholder needs, Interaction Designer, Cost estimator
  - User interface designer, user stories
  - Architect: Application overview, performance

        - Middleware —software layer between the operating system and the applications on a distributed computer network
    - Specialists as required: database, games engine, mobile development, …
    - Documentation: e.g. an amanuensis “a literary or artistic assistant, in particular one who takes dictation or copies manuscripts”

13 Essential Roles in Small Software Development Teams

1. Course Developer: Preparation and coordination of training
    a. Co-ordinates training related to project – could be the team itself or for the end-user
2. Database Designer: Essential to the process, mainly due to the specificity of its knowledge
3. Implementer: Programs sub-systems and components that support the desired functionalities
4. Integrator (lead programmer): Responsible for maintaining the implementers’ awareness of the project context, for identifying the tasks to be undertaken and for appointing the person responsible for each one.
    a. Also responsible for the initial definition of the critical dates of the project and for developing a plan for the integration of the sub-systems, to allow the project manager to inform the client when each feature is expected to be available
    b. Can have more than one integrator
5. Process Engineer: Mainly concerned with the management of the development process, its adaptation to the organizational context and monitoring its implementation, in order to identify and implement possible process improvements
    a. IDs bottlenecks, and where the project could potentially be improved
6. Project Manager: Assumes a global overview of the project through a detailed interaction with the internal and external participants
    a. Must create the conditions for the project to achieve success, by ensuring timeliness and fulfilment of all commitments.
    b. Requires: basic knowledge in management; knowledge about the client’s business domain; project management methodologies and negotiation skill
7. Project Reviewer: This role cannot be considered critical, however, due to responsibilities related to the verification and approval of several artefacts produced by other participants, and possible conflict of interests, this person cannot have another role within the project
    a. E.g. examines conflicts of interests between various parties, ensures tests are completed;
    b. i.e. how do we ensure that this project is where people say it is?
8. Software Architect: Responsible for setting the technological foundation on which the project implementation should be based.
    a. The software architect is responsible for managing the technical risks
    b. Manage these risks by developing an architecture that lowers the risk of the system by being well-designed
        i. Using OOP and design patterns
        ii. Builds a maintainable system that meets the project’s goals
9. System Administrator: Focused on ensuring the provision of the infrastructure needs (e.g., PCs for developers, servers, etc.)
10. System Analyst: Scope management.
    a. Identify and document the requirements (functional or non-functional).
    b. Understand the client's business domain and to perceive the real motivations and relevance of the requirements

    c. Evaluates overall system and plan and tries to match it to business domain – checks that system being developed aligns with the real motivations of the client
11. System Tester: Entrusted with very different tasks, like review of documentation and testing behaviour
12. Test Manager: Responsibility is to ensure the product quality by devising a plan for internal quality audits and implementation.
    a. Cannot have other roles, particularly with those roles related to the design and construction
13. User-Interface Designer: The scope of this role in a project varies according to the nature of the artefacts to be developed

Choosing People

- People you know and have worked with before
- Information from candidates about their background and experience (résumé or CV)
    - Best evidence to judge suitability
- Information from interviewing candidates
    - Mainly judge communication and social skills
    - Subjective judgements thus not reliable
    - Can also ask to perform specific exercises
- References and recommendations from people who have worked with them
    - Effective when you can rely on the people making the recommendation

Managing with Different Personality Types

- You need to cater for
    - Backgrounds and personality styles of team members
    - Management styles of customers and developers
- Realize that other people are not necessarily like you
- Different kinds of personalities need different kinds of motivation, recognition and rewards

Management and Team Success

- Most software engineering is a group activity
    - Non-trivial software projects cannot be done by one person
    - People motivated by success of the group and their own personal goal
- Individual success depends on:
    - Ability and interest for work at hand
    - Experience and training with similar applications, development tools, programming languages
- Team success depends on:
    - Ability to communicate and express ideas in the team
        - Be heard or be herded
    - Group interaction is a key determinant of group performance
- Management skills
    - Limited flexibility in group composition – do the best with people available

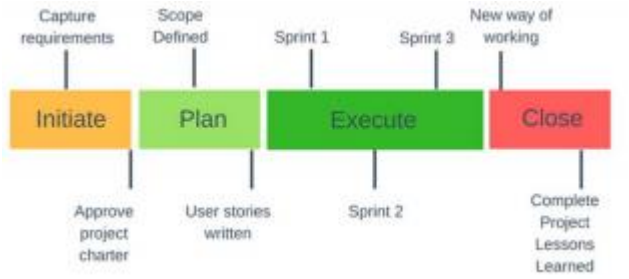
Metrics

Milestones and Deliverables

- Use tools to help with the estimation of how long different parts of the project will take, what resources are required, etc
- Activity: task that takes time
    - Duration
    - Due date
    - Precursor – activity which precedes others that depend on it
- Milestone: mark the completion of an activity
    - May/may not correspond to deliverables
    - Recognisable end-product of a task
        - Hand over system for testing
    - Use them as checkpoints to eval what has been achieved and plan next stages of project
    - Requires a formal, measurable output
- Deliverable: a project result that is delivered (to customer)

Milestone vs Deliverable

- Deliverables = measurable, tangible outcomes of the project
    - Releases, Documentation, Reports
    - Developed in alignment with the goals of the project
- Milestones = checkpoints throughout the life of the project
    - ID when one or multiple groups of activities have been completed ◊ a notable point has been reached in the project



Software metrics

- To plan and manage a software development project
    - Need to estimate the resources required for each of its constituent activities
        - Subjective perceptions of the activity
        - Based upon measurements of size and complexity
            - Activity itself
            - Artefact that's produced
    - Software metric measures some aspect of software development
        - Project level – cost/duration
        - App lvl – size/complexity
- Metrics also need to follow SMART

SMART Metrics

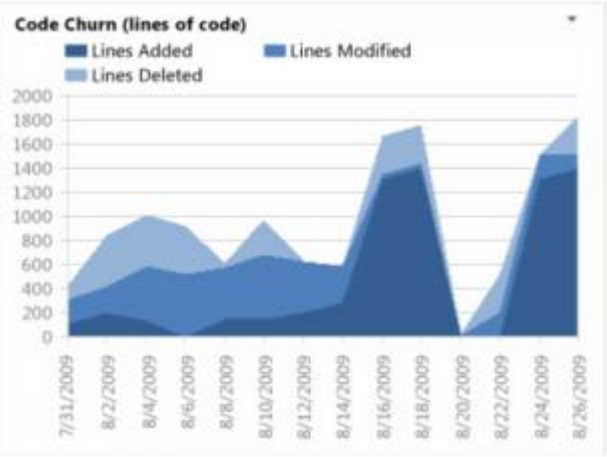
- Key Performance Indicators (KPIS)
- E.g. a KPI of lines of code – could be used to measure activity, but could be out of synch with project progress
    - While you want to measure concepts you actually need to have specific quantitative and objective measures to better track those concepts
    - These measures are often "indicators", rather than direct measures
- Need the right metrics
    - E.g. Consider user satisfaction.
        - You can ask them to rate their satisfaction on a scale of 1-5
        - But what does it mean if everything says it is usable, but no one actually wants to use it?
- Metrics need to be SPECIFIC
    - E.g. 2 sec page load time
- When using success metrics, or monitoring metrics make sure they are specific and clear
- SMART metrics
    - Counts
    - %s
    - Sums/totals
    - Avgs
    - Ratios
    - Eval and see if you've reached these numbers

Software Metric Characteristics

- Process Metrics
    - Measure some aspect of the development process – but they change all the time
        - E.g. project cost to date, time spent so far on project
        - Give some assessment of how far and fast development process is going
- Product Metrics
    - Measure some aspect of the software product
        - Analysis models

        - E.g. number of classes in an analysis class diagram
    - Test plans
    - Program code, etc

Example – Code Chart



- Code churn = measure that tells you the rate at which your code evolves
- Will only have reliable, stable code base when code churn is much lower

Software Metric Characteristics II

- Result Metrics
    - Measure outcomes
        - Current cost of project (even though it'll be modified tomorrow)
    - Aka Control metrics
        - Used to determine how management control should be exercised
            - Measurement of the current level of progress in the project is used to decide whether action is necessary to bring the project back onto schedule
                - E.g. by hiring more people to work on project, shifting people around, etc

Software Metric Characteristics III

- Predictor Metrics
    - Quantify estimates for project resource requirements
        - Class size
            - A crude measure could be – simple count of attributes and operations
            - Predictor because can be used to predict the time that it will take to produce code
    - Also, a measure of some aspect of a software product that is used to predict another aspect of the product or project progress
        - Predict that the system will be difficult to maintain – can then use that info to better allocate resources for maintenance
        - Predict very low levels of reuse – change design to address this
        - Change the design to improve the system

Software Metrics Worth

- Useful for prediction and resource estimation
    - Otherwise, is rather limited – validity of metrics based on idea that accurate predictions can be made e.g. on the time it might take to complete something
- Validity of predictor metrics is based on 3 assumptions
    - You can measure something useful
    - That measure actually predicts something worthwhile
    - This relationship is real and can be expressed in a model or a formula
- Size metrics can be used to estimate the resource requirement for a project provided that appropriate historical data is available to derive and validate the relationship
    - E.g. # of lines of code – not necessarily an indicator of code quality
    - Interpretation of what metrics mean are limited – unless able to contextualise predictions and validate then
- Generally, software developers don't think metrics are important …
    - Focused on the delivery of the product on time
    - Can be used to monitor the performance of the developers
        - This is a cause of concern to developers
        - Ethical issue and a management issue

Metrics for OO Development

- Ability of a package to absorb change is partly depends on ratio of abstract classes to all classes
    - 0 -> package only has concrete classes and is difficult to change
    - 1 -> no concrete classes at all (easy to change!)
- App size
    - # of use cases
    - # of domain classes
    - **X** multiplying factors that reflect the complexity of the user interface
        - i.e as one is added, it multiplies the complexity
- Class size
    - # of attributes
    - # of operations
    - Size of operations
- These concrete numbers help asses project complexity, number of resources that need to be allocated and amount of work required to ensure project completion

Project Scheduling

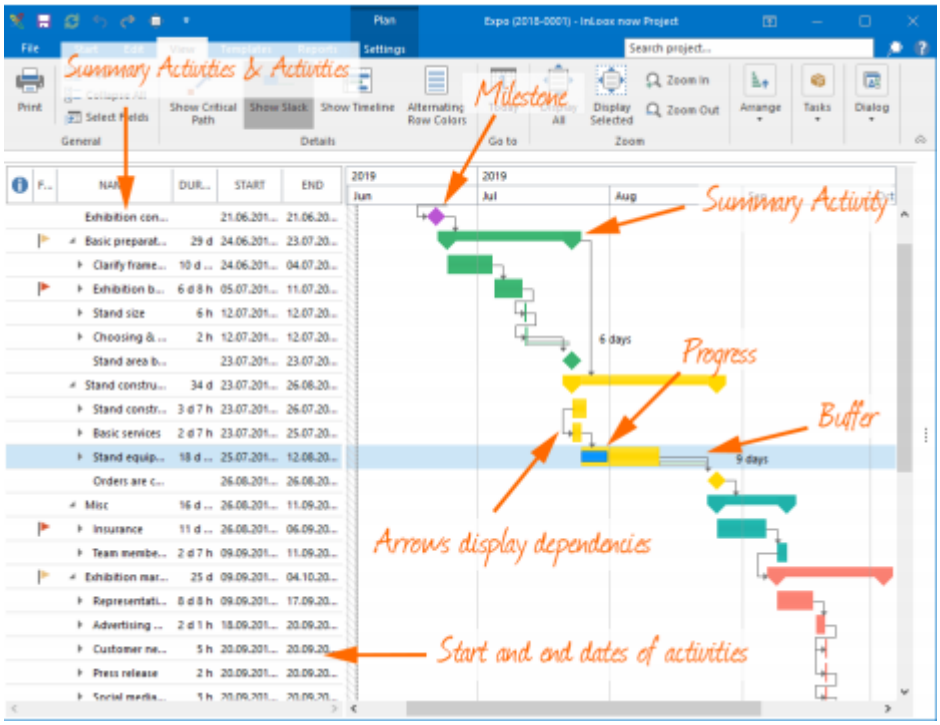
- Split the work in a project into separate tasks
    - Minimize task dependencies
    - Discrete, achievable tasks
    - Clear dependencies where they exist
- Estimate the calendar time needed to complete each task
    - Split up if much longer than 1 week (never more than 2 months)
    - Make tasks concurrent to make optimal use of workforce
- Estimate the effort required

    - Who will work on the tasks?
    - Resources needed to complete each task
    - Estimates can vary be person
- Mostly needs project manager's intuition and experience

Project Scheduling Process

- Linear project scheduling process:
    - (1) As part of the requirements analysis and design, start to identify activities
    - (2) Identify dependencies between the activities, and potential bottlenecks
    - (3) Estimate resources required for each activity
    - (4) Allocate people to the activites
    - (5) And document this in the project charts – the gantt chart or the Kanban board
- Can revise estimates and adjust – the charting activity will help you identify potential bottlenecks and adjust by shifting human resources or finding new ones, or adjusting scope

Gantt Charts

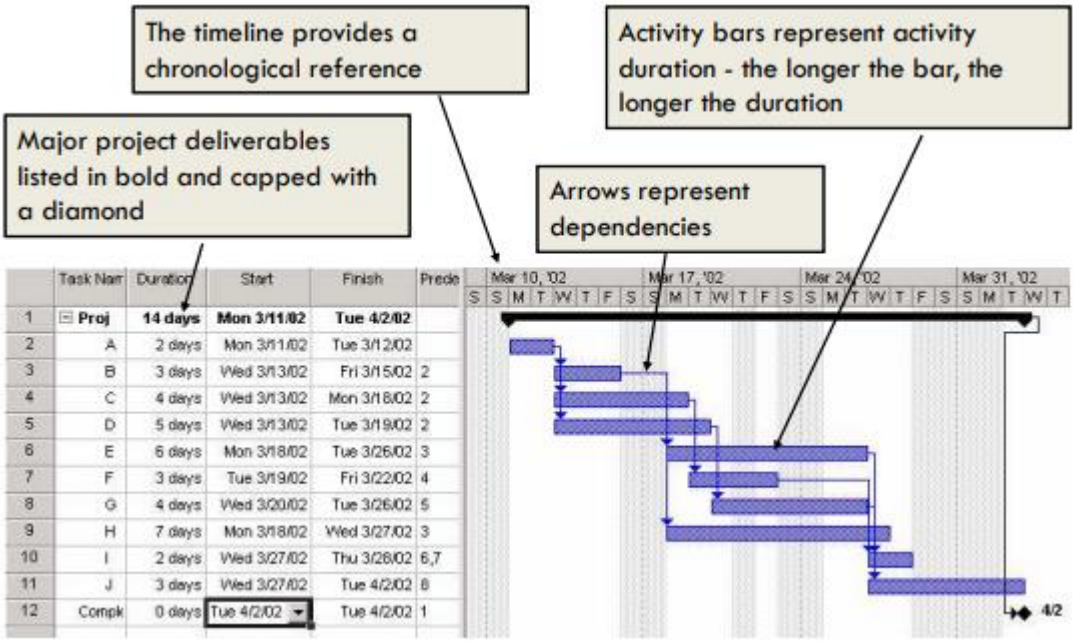


- Horizontal axis represents project time span
- Vertical axis represents project tasks
- Capture:

- Task completion
- Simple dependencies
- Milestones and Deliverables

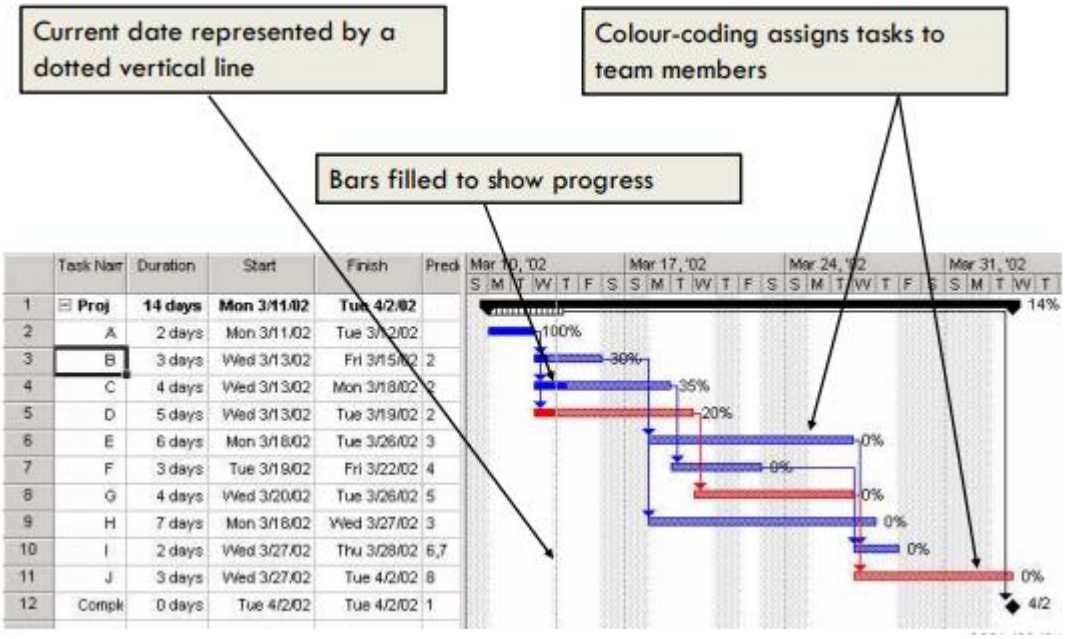
- Can't handle complex task dependencies
- Always accompany your gantt chart in your report with an outline summary of phases, milestones and deliverables

Before Project



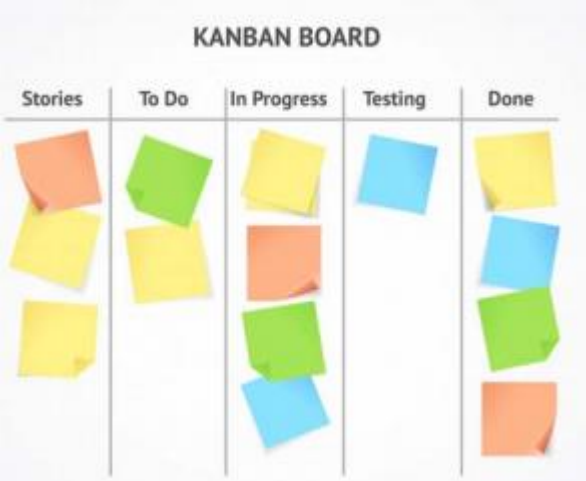
- Planning phase - project details on the left
  - Tasks are often hierarchical, grouped in phases, with a "precedent" column to indicate dependencies on other tasks
- Right side - have visuals to graphically represent the tasks and how they are related.
  - Diamonds indicate the start and end of major deliverables
  - Arrows depict dependencies between tasks

During Project



- More info added - progress for each task indicated by filling the bars, color coding for individual team members, and a vertical dotted line for the current date

Kanban Boards



- Often plan overall project with Gantt Chart, then track individual phases. Iterations/sprints in Kanban board
- Visualize at a glance current project activities
    - i.e. useful for visualizing individual phases or sprints rather than the project as a whole, since the backlog can be more of a pile than a clearly linked set of tasks
- Each task is represented as a single card, usually depicted as a user story
- Cards travel from left to right as they are completed:
    - (1) Stories/Backlog – the full list of tasks
    - (2) To do – tasks planned for current cycle or phase
    - (3) In progress – tasks that have been started
    - (4) Testing – tasks that have been completed and are ready to undergo testing

- (5) Done – tasks that're completed

Kanban Task Management

- Can add or remove columns to suit your workflow
- Some sw also offers time trackers, labels and other features to track dependencies, help assign tasks and estimate time for each
- Kanban strength – can visualize (at a glance) the current state of the project, and that updating status is a simple drag and drop

Network Analysis

Critical Path

- Have list of tasks, dependencies and estimates – can systemically identify your critical path and plan around it
- Related to PERT (Program Evaluation Review Technique) charts
- Min overall duration of the project according to the estimates depends on the critical path(s)
  - Critical path – the sequence of stages determining the minimum time needed for an operation, especially when analysed on a computer for a large organization.
- Critical Path analysis is used to help estimate the minimum overall duration of the project
  - Crit path is important – any delay in any task on this path delays the whole project

Examples – Tasks for a Multi-user 3D Meeting Place

#	Activity	Description	Len	Depend
1	Access	provide access through an account, username, password and access privileges to system.		
1.1	User's status	keep a database of users status (online, available, busy, etc.).	30	
1.2	Persistence	to keep a database of all avatars, objects, their positions and states; thus providing "persistence" in the virtual world for users who come and leave.	180	1.1
1.3	Inventory	keep a database of users virtual inventory and support the addition, removal and exchange of items in the inventory.	10	1.2
2	Logging	log all events in world		
2.1	Log user events	log user status in world	10	1.3
2.2	Log avatar events	log position of avatars	10	3.4
2.3	Log object events	log position of objects.	10	4.3
2.4	Log communication	log all communication that occurs in VE.	100	2.1,2.2,2.3

#	Activity	Description	Len	Depend
3	Avatar	provide user with a virtual representation of themselves in world.		
3.1	Avatar customization	provide user with the opportunity to easily customize avatar.	45	
3.2	Avatar Gesturing	provide user with the ability to express themselves using animated avatar gestures.	30	3.1
3.3	Avatar mood indicator	provide user with the ability to express their moods through visual representations (iconic, color, facial expressions ?).	160	3.2,1.1
3.4	Avatar proxemics	manage optimal proximity and gaze of groups of users engaging in communication.	20	3.3
4	Interactions	provide users with the ability to interact with VE, other users, their avatars and objects in VE.		
4.1	Manipulate objects	provide users with the ability to move, rotate, resize objects in VE in intuitive way.	20	
4.2	Operate Objects	provide users with the ability to operate interactive objects that have been scripted or have a menu interface.	120	4.1,1.1
4.3	Operate on Avatar	provide users with the ability to operate (issue commands) on other avatars through the menu interface.	60	1.2,4.2

- Start with a list of tasks
- Several columns:
  - Task number
  - Activity name
  - Longer description
  - Expected amount of time it will take to complete this (in hours)
  - Dependencies referencing previous task numbers
- Activities and subactivities listed
- For high lvl activities – no length and dependency listed because these are listed in the subtasks

Network Analysis

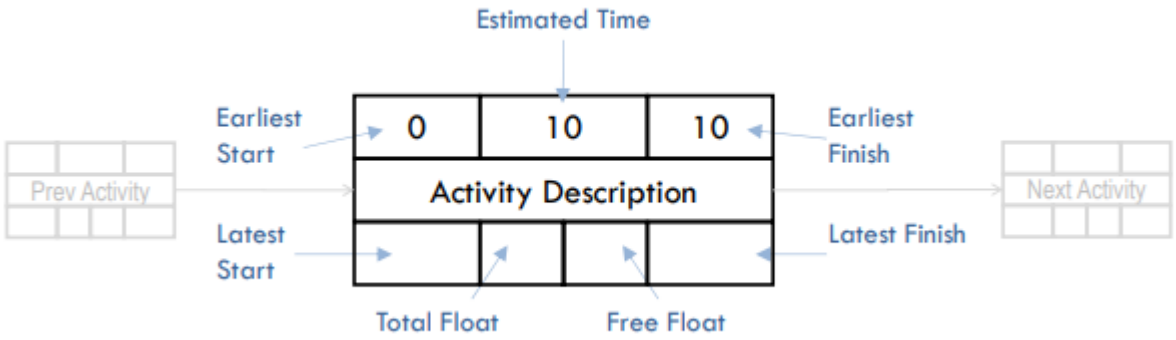
- Project tasks (activities):
  - Are often interdependent
  - But need to be done in parallel for teamwork to be effective
- Task networks are graphical depictions of task dependence
- Network analysis is a project planning method that:
  - Determines the critical path
  - Establishes “most likely” time estimates
  - Calculates boundaries to stop project slippage
- Use nw analysis to establish likely time estimates given parallel streams of tasks
  - Helps understand how much “extra” time we have for tasks to take longer than expected
  - Fuzziness - – due to how difficult it can be to accurately estimate time for tasks

- But the analysis can help us plan better for bad estimates

Terminology

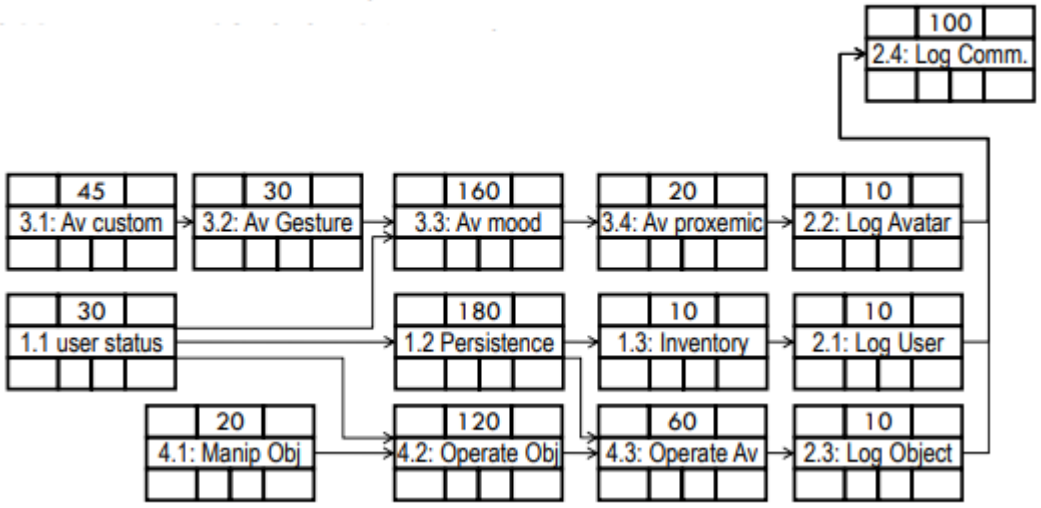
- Earliest Start/Finish
  - Earliest a task can begin/end if all preceding tasks are completed in the shortest time
- Latest Start/Finish
  - Latest a task can begin/end without delaying the minimum project completion time
- Crit path
  - Chain (of tasks) that determines overall project duration
    - Delay in any of these tasks delays the overall project
  - Can be >=1
- Slack (Float): the amount of surplus time or leeway allowed while still maintaining the critical path

Tasks



- Each task in a network analysis is represented as a box, and the boxes will be linked in a nw
  - Sequence constraints are lines connecting the boxes
- Start with activity names and the estimated time

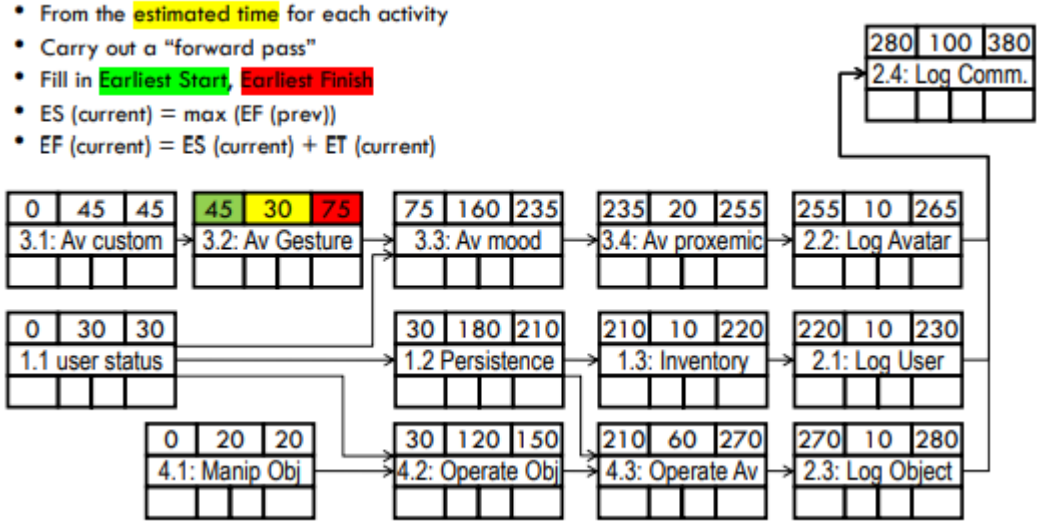
Analysis – Step 1



- Label tasks in order and indicate dependencies
  - One box for each task in table
  - Fill in estimated time
- Arrows will then link all the tasks in order, based on dependencies
- The rows in the table we started with are now represented as a network

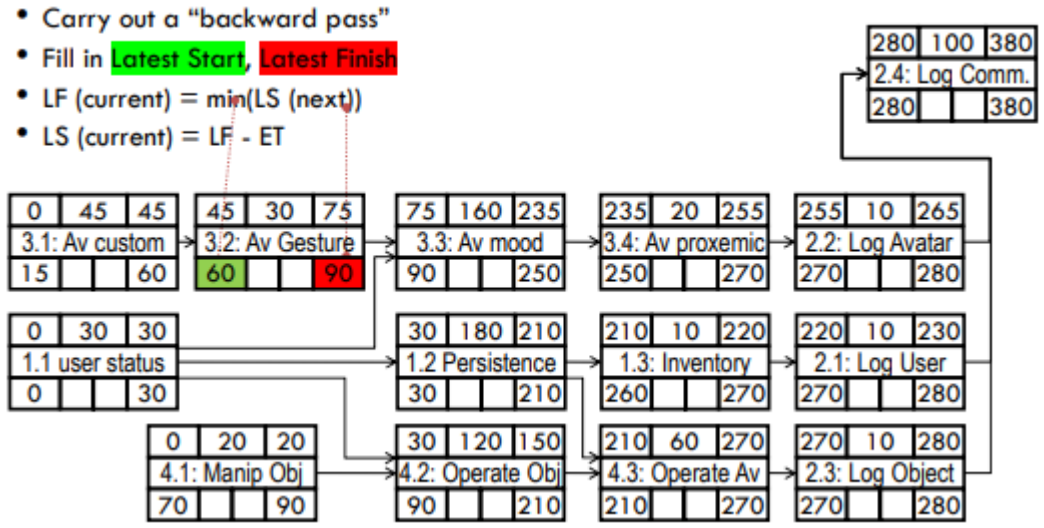
Analysis – Step 2

- From the estimated time for each activity
- Carry out a "forward pass"
- Fill in Earliest Start, Earliest Finish
- ES (current) = max (EF (prev))
- EF (current) = ES (current) + ET (current)



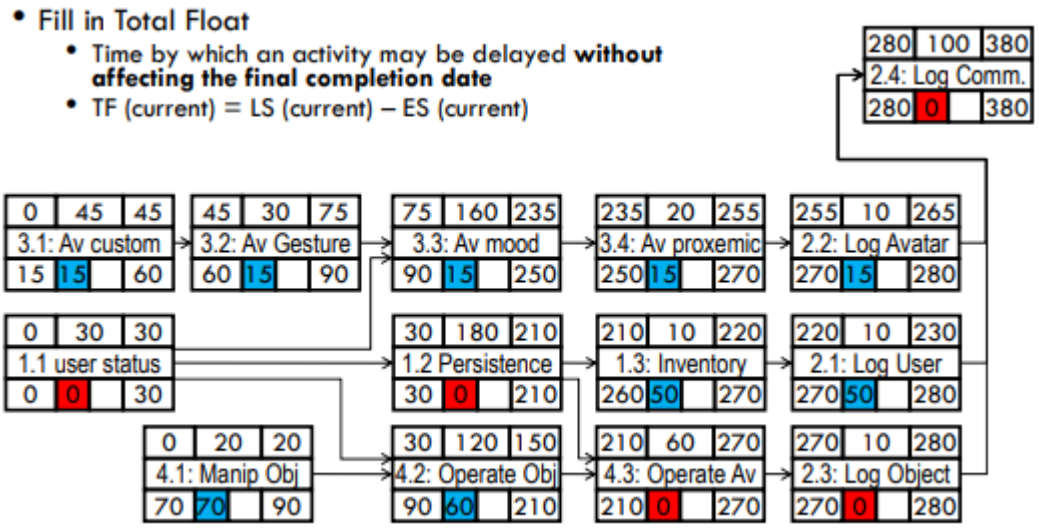
- From the estimated time for each activity
- Carry out a "forward pass" - use the time estimates to calculate earliest start and finish
- Fill in Earliest Start, Earliest Finish
- Start with 0 for tasks with no dependencies - add the estimated time in a given box to the earliest start to calculate the earliest finish for that task
- In the next box, you look at all the precedent tasks and take the latest finish time as the ES
  - Do this for each box
- ES (current) = max (EF (prev))
- EF (current) = ES (current) + ET (current)

Analysis – Step 3



- Start with the final box, the highest "earliest finish" number
- Subtract the time estimate to get the latest start
- That latest start gets filled in to all of the preceding tasks as the latest finish
- If there's more than one preceding task,
  you copy the lowest "latest start" into the preceding task's "latest finish"
- You should end up with at least one task with a latest start of 0.
- Process:
  - Carry out a "backward pass"
  - Fill in Latest Start, Latest Finish
  - LF (current) = min(LS (next))
  - LS (current) = LF – ET

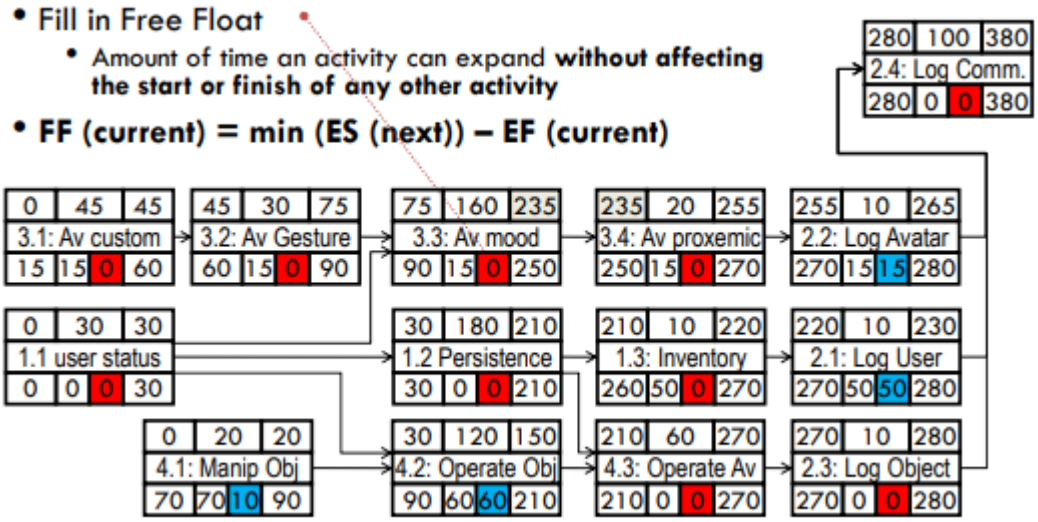
Analysis – Step 4



- Calculate total float, which is the difference between the earliest start and latest start
- Process:
  - Fill in Total Float

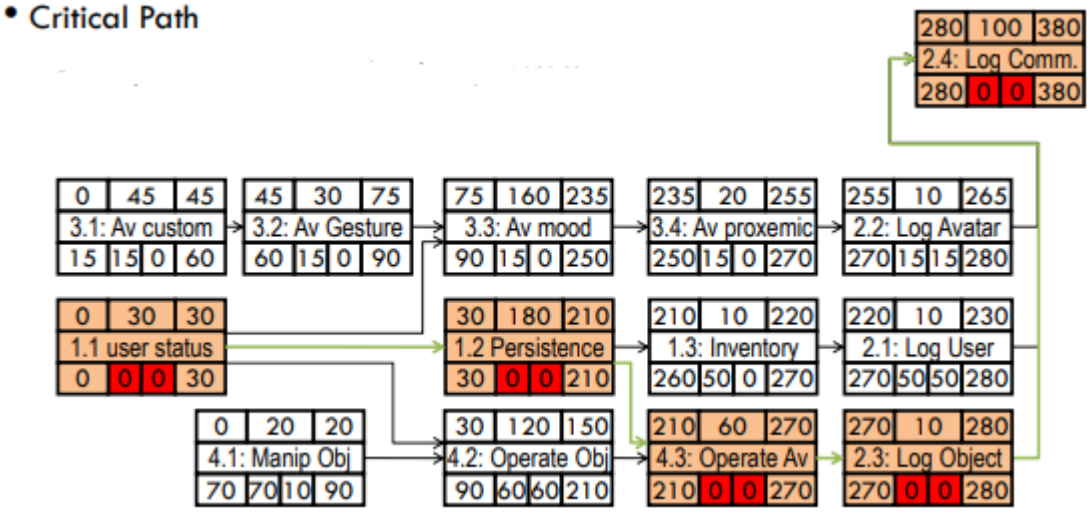
        - Time by which an activity may be delayed without affecting the final completion date
    - TF (current) = LS (current) – ES (current)

Analysis – Step 5



- Free float: the difference between the earliest start of the next tasks, and the latest finish of the current one
    - Basically, the buffer time for that task
- Process:
    - Fill in Free Float
        - Amount of time an activity can expand without affecting the start or finish of any other activity
    - FF (current) = min (ES (next)) – EF (current)

Analysis – Step 6

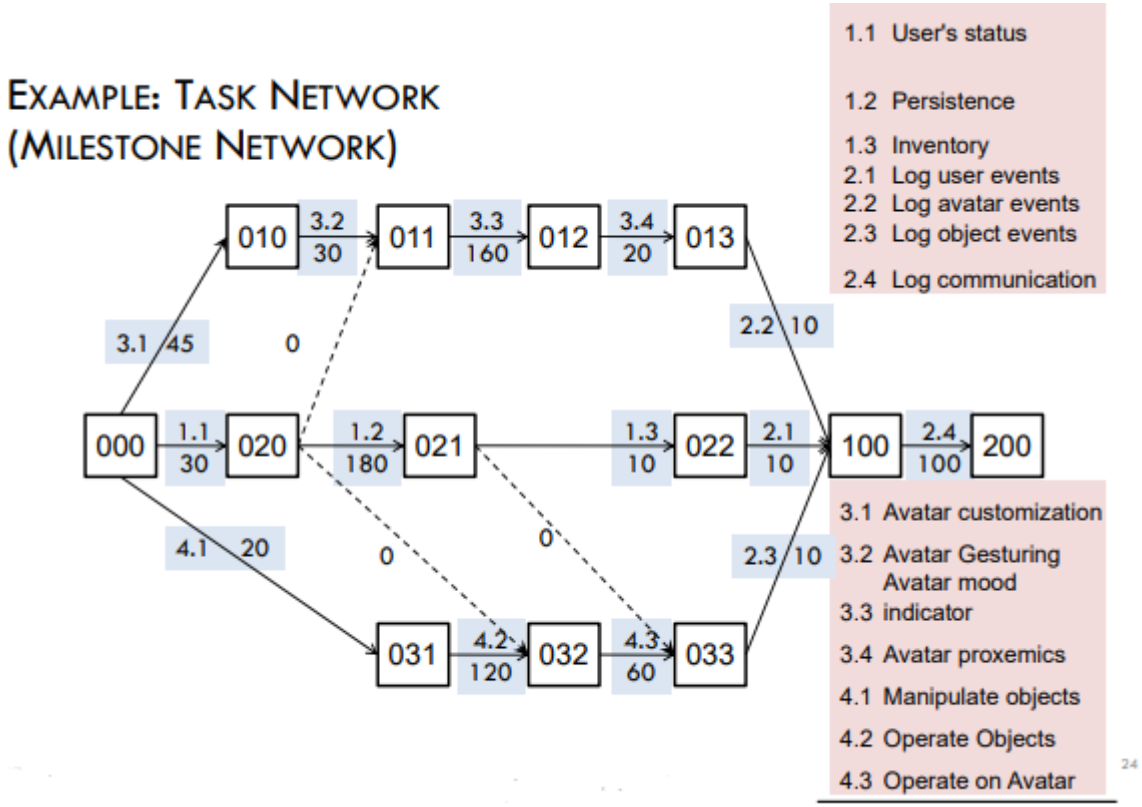


- Once CPA is drawn up, it is possible to see the crit path itself – it's a route through the CPA, which has no spare time (called 'FLOAT' or 'slack') in any of the activities

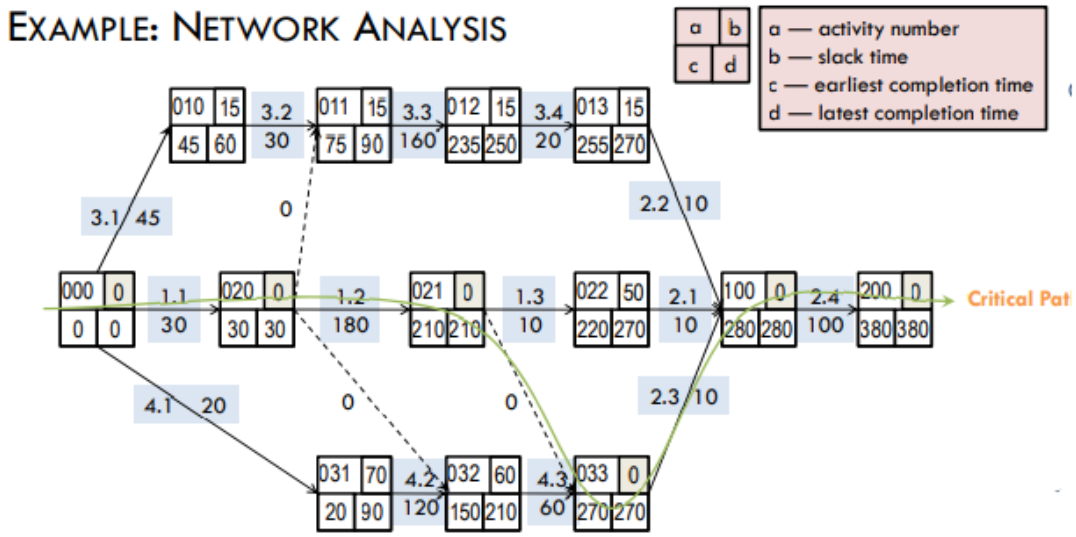
- If there is any delay to any of the activities on the critical path, the whole project will be delayed unless the firm makes other changes to bring the project back on tra ck

- Crit path – this path is the manager's primary concern

Example – Milestone Network (Task Network)



- Another way to depict crit path
- The task # and time estimates are depicted on the arrows
  - Each box represents a milestone
  - Dotted lines depict dependencies across other development paths
- In example, milestone 100 involves potentially a critical integration task



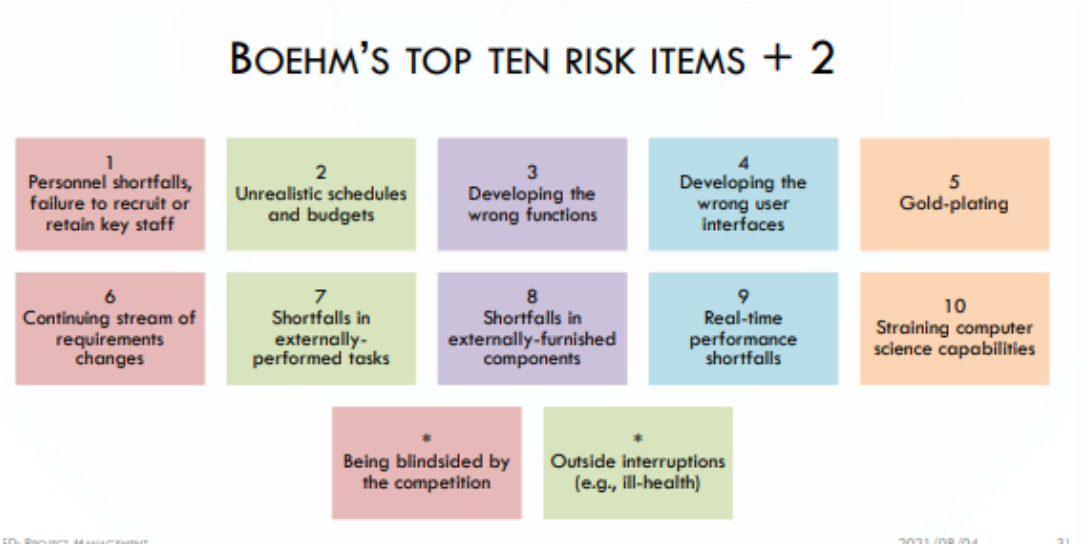
- Splitting each milestone box for network analysis, you can perform the same forwards and backwards analysis to determine slack time, earliest and latest completion times
- Milestones with no dependencies have an earliest completion time of zero
- Earliest completion time of the next box is calculated by adding the estimated time to the max previous earliest completion time
- Latest completion time is calculated backwards the same way

Risk

Managing Risks

- Why?
  - Projects have a high level of uncertainty
  - Better to anticipate problems
- How?
  - Identify -> Analyze -> Rank
  - Plan for monitoring, mitigation, management
  - Revisit continually during project

Boehm's Risk Items



1. Another risk is someone getting sick; someone leaves and a replacement cannot be found
2. Unrealistic schedules and budgets almost guarantees overrunning then
3. Been given the customer spec and requirements docs – but they were incorrect for the problem that needs to be solved
4. /
5. Gold planting – e.g. taking slides and working too hard on how they look, rather than their actual content
6. Scope creep – changing requirements constantly
    a. Mitigated by anticipating change and creating architectures that support adding new features and changing direction
7. Pokemon GO case study – library that routinely crashed
    a. Have dependencies on outside sw – if things aren't up to spec, then this may cause delays
8. /
9. Performance delays – e.g. machine being used to dev is slow
10. /
11. /
12. E.g. Covid

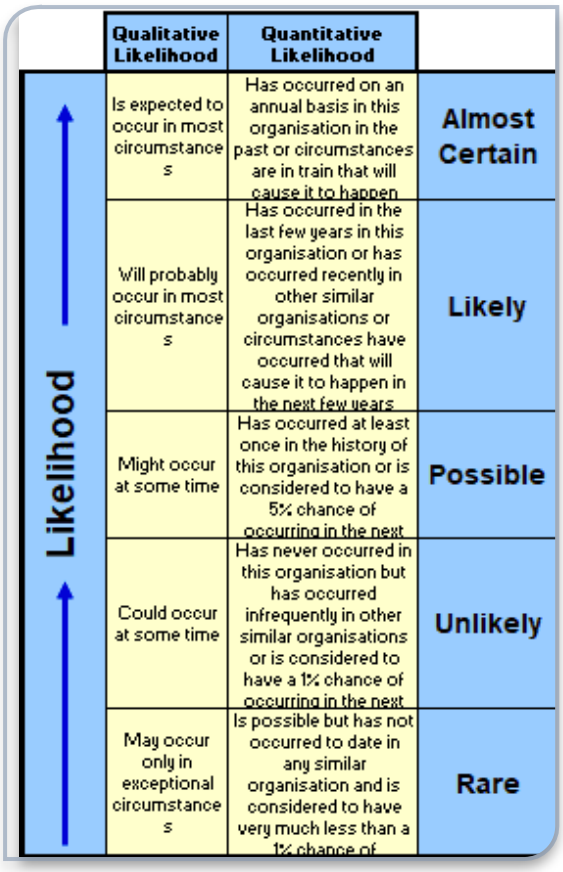
Risk Consequences

	Consequence				
People	Minor skills impact.	Minor impact to capability	Unavailability of core skills affecting services.	Unavailability of critical skills or personnel	Protracted unavailability of critical skills/people.
	Minor injury or first aid treatment	Injury requiring treatment by medical practitioner	Major injury / hospitalization	Single death and/or multiple major injuries	Multiple deaths
Information	Compromise of information otherwise available in the public domain.	Minor compromise of information sensitive to internal or sub-unit interests.	Compromise of information sensitive to this organisation operations.	Compromise of information sensitive to organisational interests.	Compromise of information with significant ongoing impact.
Property & Equipment	Minor damage or vandalism to asset.	Minor damage or loss of <5% of total assets	Damage or loss of <20% of total assets	Extensive damage or loss <50% of total assets	Destruction or complete loss of >50% of assets
Reputation	Local mention only. Quickly forgotten. Freedom to operate unaffected. Self-improvement review required	Scrutiny by Executive, internal committees or internal audit to prevent escalation Short term local media concern. Some impact on local level activities	Persistent national concern. Scrutiny required by external agencies. Long term 'brand' impact.	Persistent intense national public, political and media scrutiny. Long term 'brand' impact. Major operations severely restricted.	International concern, Governmental Inquiry or sustained adverse national/international media. 'Brand' significantly affects organisational abilities.
Financial	1% of Project or Organisational Annual Budget	2-5% of Project or Organisational Annual Budget	5-10 % of Project or Organisational Annual Budget	> 10% Project or Organisational Annual Budget	> 30% of Project or Organisational Annual Budget
Capability	Minimal impact on non-core business operations. The impact can be dealt with by routine operations.	Some impact on business areas in terms of delays, systems quality but able to be dealt with at operational level	Impact on the organisation resulting in reduced performance such that targets are not met. Organisations existence is not threatened, but could be subject to significant review or changed ways of	Breakdown of key activities leading to reduction in performance (eg. service delays, revenue loss, client dissatisfaction, legislative breaches). Survival of the project/activity/organisation is threatened	Critical failure(s) preventing core activities from being performed. The impact threatens the survival of the project or the organisation itself.
	Insignificant	**Negligible**	**Moderate**	**Extensive**	**Significant**

- Assess the magnitude of potential consequences of the risk to project:
    - Timelines
    - Cost
    - Feasibility
    - Quality

Risk Likelihood

- Assess the likelihood of the risk



Likelihood	Qualitative Likelihood	Quantitative Likelihood	
↑	Is expected to occur in most circumstances	Has occurred on an annual basis in this organisation in the past or circumstances are in train that will cause it to happen	Almost Certain
	Will probably occur in most circumstances	Has occurred in the last few years in this organisation or has occurred recently in other similar organisations or circumstances have occurred that will cause it to happen in the next few years	Likely
	Might occur at some time	Has occurred at least once in the history of this organisation or is considered to have a 5% chance of occurring in the next	Possible
	Could occur at some time	Has never occurred in this organisation but has occurred infrequently in other similar organisations or is considered to have a 1% chance of occurring in the next	Unlikely
↓	May occur only in exceptional circumstances	Is possible but has not occurred to date in any similar organisation and is considered to have very much less than a 1% chance of	Rare

Risk Matrix

Likelihood	Consequence: Insignificant	Negligible	Moderate	Extensive	Significant
Almost Certain	6	7	8	9	10
Likely	5	6	7	8	9
Possible	4	5	6	7	8
Unlikely	3	4	5	6	7
Rare	2	3	4	5	6

- Asses overall risk
- Higher likelihood + higher consequence => want to put more effort into mitigation and management

3M's

- Mitigation
  - How can we avoid or reduce the risk?
    - Avoid - change requirements
    - Transfer – e.g. buy insurance
  - Or assume the risk and accept and control it
- Monitoring:
  - What factors can we track that will enable us to determine if the risk is becoming more or less likely?
- Management:
  - What contingency plans do we have if the risk becomes a reality?

Examples of Risk

Risk Condition	Consequence	Cat	Prob	Impact	Mitigation	Monitoring	Management
Competitors duplicate technology rapidly	Competitors products dilute market share	Follow-On	High	Critical	Develop Copy and IP protection, protect trade secrets through NDAs and source code through copyright	Keep abreast of current research and potential competitors	Adjust product pricing, develop upgrades
Product not well targeted to market	Unable to Capture Market Share	Commercialization	Medium	Critical	Create Business Plan and adapt development accordingly	Tracking current and predicting future market trends	Further market research and retargeting of product

Risk Condition	Consequence	Cat	Prob	Impact	Mitigation	Monitoring	Management
Failure to recruit suitable lead developer on schedule	Significant delays (lead developer is on critical path)	Development	Medium	Critical	Advertise widely, offer attractive salary package	Track number and quality of applicants	Second round of recruiting (perhaps internationally), Target specific individual
Failure to acquire international business partner	International marketing and distribution hampered	Follow-On	Medium	Critical	Begin search early, develop business plan and prototypes to promote concept	Keep record of potential partners and commitment status	Delay follow-on until partner recruited or revise business strategy

RISK: COMPETITION IN TECHNOLOGY

Risk Condition	Consequence	Mitigation	Monitoring	Management
Competitors duplicate technology rapidly	Competitors products dilute market share	Develop Copy and IP protection, protect trade secrets through NDAs and source code through copyright	Keep abreast of current research and potential competitors	Adjust product pricing, develop upgrades

Update 9/5/20..	Update 23/10/20..
Virtools and Prof. Kaufman identified as key competitors	3D Games Studio is a competitor, but not Prof. Kaufman

Planning vs Management

- Planning
  - Pre- and post-
  - Network analysis, resourcing, risks, schedule
- Management
  - During
  - Controlling resources and timescales

Software Engineering Methods

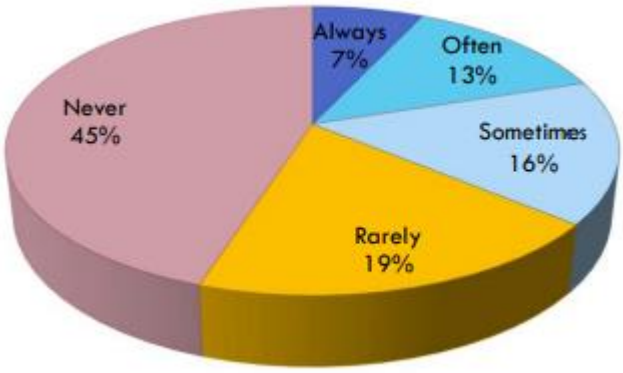
- Most common problem in software systems is not the construction, but the estimation
    - Most frequently, it is discovered too late, or possibly even after the fact, that the targets are wrong
- Software projects fail to meet cost and schedule, because those targets are wrong
    - Costing software = difficult
    - The scope might not be correct, or time estimates for development wildly underestimated
- Know little about accurate estimations so targets are unreasonable
    - Made by people least able to make them
        - e.g., marketers, managers and customers
    - Communication is hard when the ideas are abstract or conceptual

Software Engineering Triangle

- Want to maximise all axes – time, scope and cost
- In a world of limited resources you cannot increase one with increasing at least one other, or compromising on quality
- Reading: https://vula.uct.ac.za/access/content/group/4f18fec6-963a-4b05-a486-4123767ce876/ASD%20Lectures/CHAOSReport2015-Final.pdf

Problems Encountered in Software Engineering

Wasted Effort

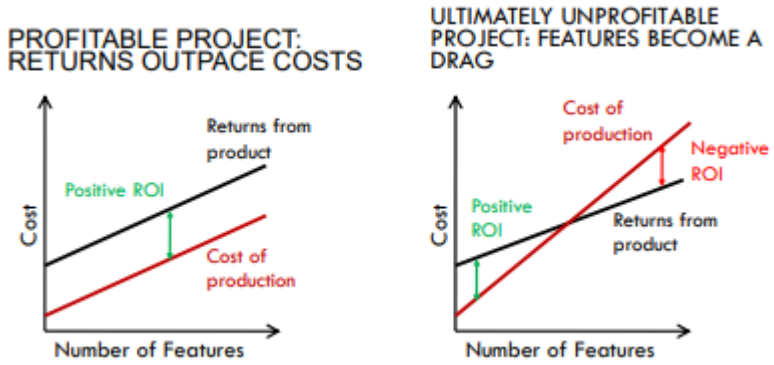


- More than 45% of features are never used, while another 19% are used rarely
    - Almost 2/3 of the features are never or rarely used
    - Stop developing these features and double productivity
- NB the 80/20 rule i.e. 80% of the features are used only 20% of the time
    - By first focusing on the 20% of the features used 80% of the time, you use resources more effectively when they are needed – even when you didn't expect them to be needed

You Ain't Gonna Need It

- Try to prevent speculative development and Gold Plating (aka Bells and Whistles)
- Better is to build only what you need now
- Speculative development adds complexity to code prematurely
    - Bells and whistles don't add very much business value while also introducing another point of failure
- These can be added perhaps when project risk is reduced, after key functionality has been well-developed and tested, AND once you actually have time and resources to properly test these additional features

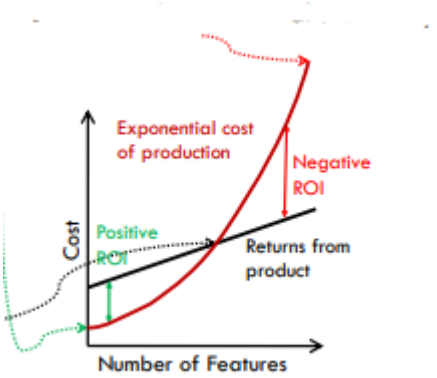
Economies of Adding Features



- Limit to ROI for what that feature adds i.t.o how much money it's going to give you
- Amount of work needed to implement feature often > return on product (when it's a feature that hasn't been prioritised)
- Profitable project – returns always outpace costs

    - For every unit of cost, time added to producing the sw, there is an equivalent increase in returns on the project
- However, if continually increase scope of the project, every new feature only adds a small amount to ROI
    - Can lead to negative ROI

Cost Curve For Software Processes



- The cost curve under most software processes is exponential – as features increase there's increased communication and coordination
    - E.g., Customer and developer must understand each other perfectly
- New projects are successful because the cost curve is still flat, but as costs start increasing the curve quickly overcomes the ROI for new features – the return on investment is linear (or worse) rather than matching increased costs of production

The Crunch

- Crunch is the side effect of other problems and the cause of burnout

Software Entropy, Rot & Geriatrics

- Entropy: a measure of disorder in a physical system
- Software Entropy: measure of code complexity
    - i.e. as you dev sw, there's more disorder – disorder manifests as code complexity
    - Tends to increase over time
    - Speculative development adds complexity at the start
    - Bug fixes and enhancements increase complexity and degrade structure
        - Most software applications grow at annual rates of 5%–10%
- Entropy makes it hard to
    - Make changes and fixes
        - E.g. you made a change here, and then have to add a fix somewhere else because of a strange dependency
    - Understand the code
- Cure for Entropy is:
    - YAGNI at the start and
    - Refactoring as you go along

Yak Shaving Example

1. You want to generate documentation based on your git logs

2. You try to add a git hook only to discover the library you have is incompatible and therefore won't work with your web server.
3. You start to update your web server, but realize that the version you need isn't supported by the patch level of your operating system, so you start to update your operating system
4. The operating system upgrade has a known issue with the disk array the machine uses for backups.
5. You download an experimental patch for the disk array that should get it to work with your operating system: it works but causes a problem with the video driver. …

Stop: what got you started down this road? Try to figure out what shaving a yak has to do with generating documentation for Subversion logs.

- Yak shaving is dangerous because it eats up a lot of time.
    - It also explains why estimating tasks is so often wrong: just how long does it take to fully shave a yak?
    - Always keep in mind what you are trying to achieve and pull the plug if it starts to spiral out of control.

Ultimately

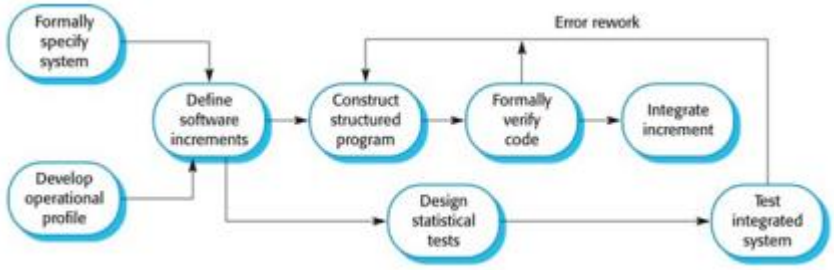
- Have
    - Undefined system
    - Fixed resources
    - Fixed time
    - High quality
- Goal – deliver software product to meet the client's needs, on time and within budget
- Can we develop quality software under these circumstances?
    - If so, how?

Approaches

Code And Fix

- Naïve, first approach
- Little (zero) planning, dive straight into implementation
- Reactive
- End with bugs
    - If bugs multiply too fast to fix: "death spiral" → cancelled
    - To make it you'll have "crunch time"

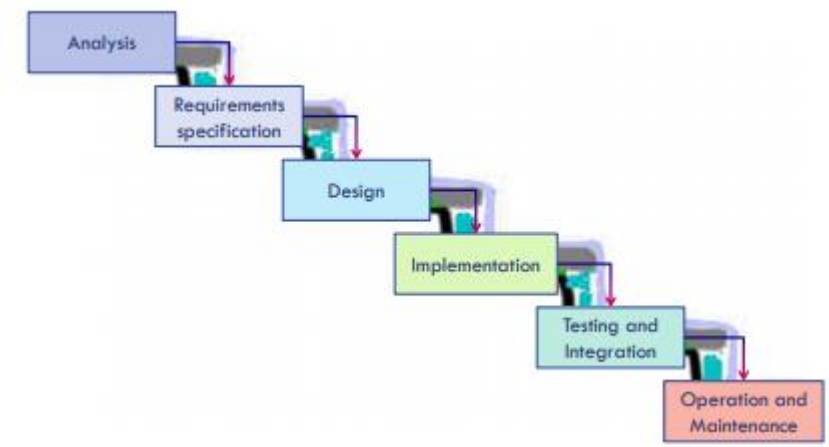
Traditional Methods



- Used for well-defined systems

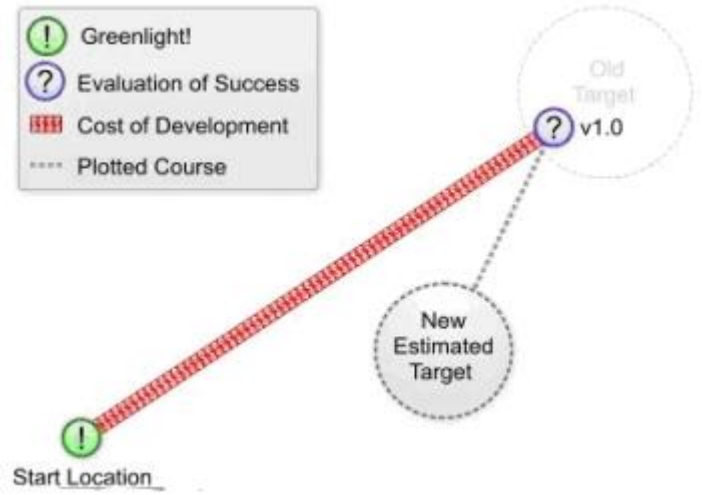
- User can specify the requirements
- Developers can then do the development
- System is finished
- System is launched

Waterfall Method



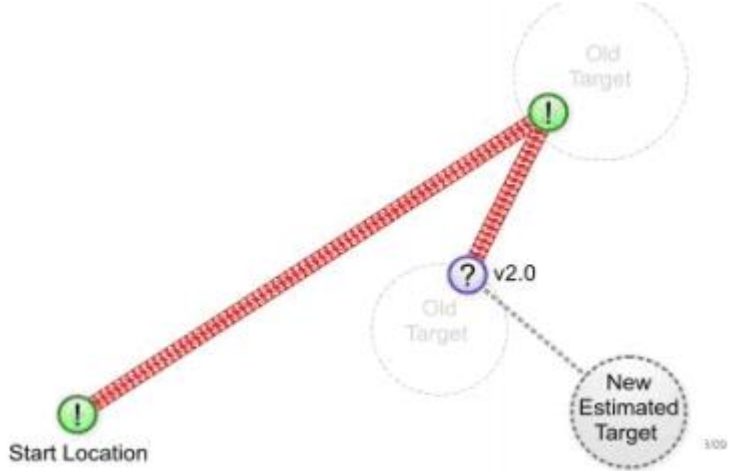
- In this approach software development is a series of stages, in which the input from a previous stage flows into the next
- Linear, sequential
- Teams gather requirements
- Develop the product
- Test it to see if they implemented the spec correctly
- After release they gain insight into what the customer actually desired
- Would be nice if it worked

Versions

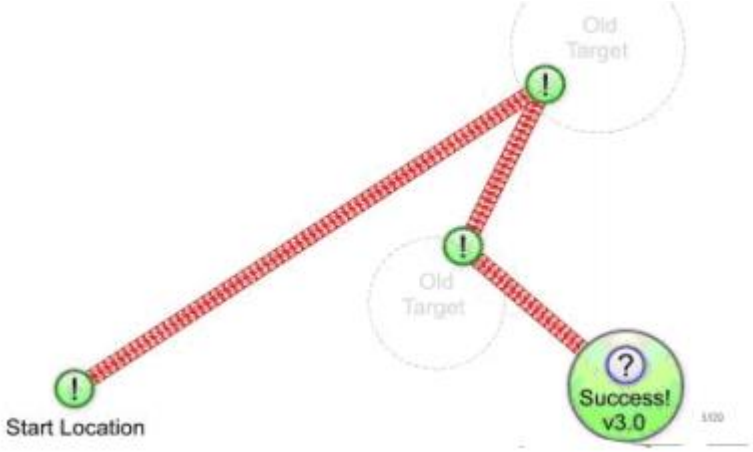


- What actually happens – start project, develop and then present to customer
  - Customer then decides that isn't what they want – target changes
- Target has changed
  - Given more money you can try again

- There a good chance the team learned quite a bit about what their customers actually desired
  - Next rocket has a better chance of landing closer to the actual customer needs



- Customer may change their mind multiple times – but will eventually arrive at what they want, and the product will be completed



Waterfall Concepts

- Software as an Engineering discipline - "Do it right the first time"
- The more design time reduces risk
  - By planning upfront you identify problems early and avoid mistakes
    - The longer you analyse a system, the more edge cases you'll discover
    - Often design elaborate systems for problems that do not really exist

Change and the Waterfall Method

- The cost of change increases exponentially with time – because will have to re-implement the project to come up with new iteration
  - Conservative design decisions motivated by fear of change
    - A change late in the process costs 1000 times as much as a change early in the process
      - 5 minutes to write a spec
      - 2 days to program the feature

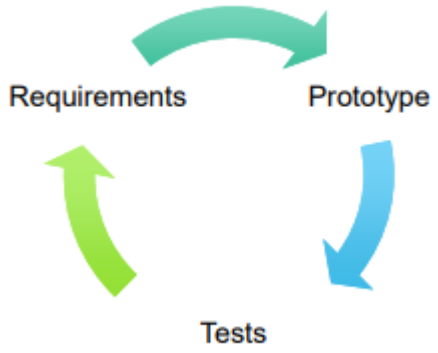
- 2 weeks to test it before deployment
- Month to write a patch that fixes a problem after deployment

Iteration

Modern Processes

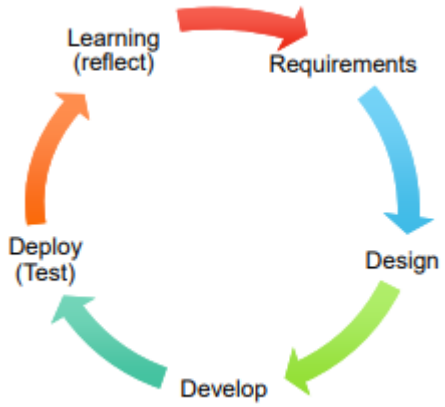
- More lightweight than waterfall
  - Less documentation
  - Fewer procedures
- Don't release only one version at the end.
  - Parallel development
  - Product of prototypes
- Only do what is required
  - No adding in extra requirements
- Design for change
  - Change is inevitable – that's why it's necessary to produce prototypes to get better, more accurate requirements

Alternative Ideas – Prototyping



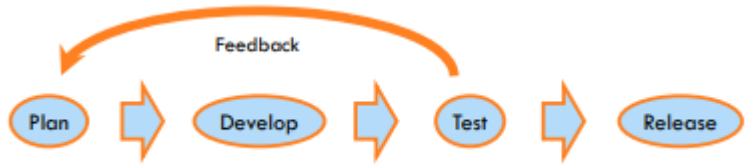
- Start with a set of requirements, then build prototype(s) and then run tests on those prototypes
  - Those tests help inform next set of requirements to prototype and test
  - Iterate until reach a prototype that's close to final product

Rapid Application Development



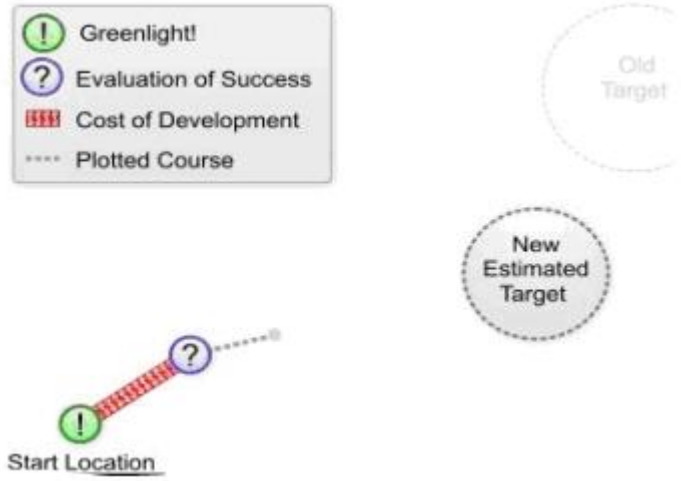
- Expended process to prototyping
- Start with requirements gathering, move into design phase, then develop, deploy and test
    - Examine test results, learn and reflect – that provides feedback into the requirements again
    - Will need to reorient as go through cycles to better match what people need
    - Refine the design over time

Iterative/Incremental Process

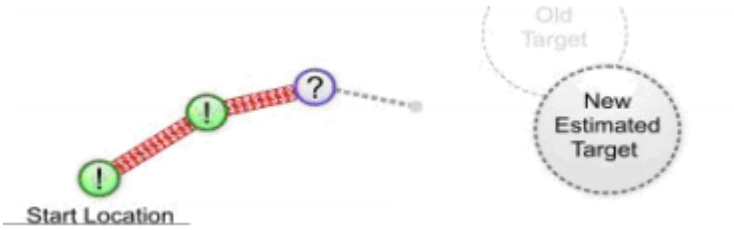


- Plan development
- Undertake development
- Generate a prototype
- Get user feedback
- Develop again
- Iterative software development: a process that reaches the goal in a series of ever improving delivery cycles

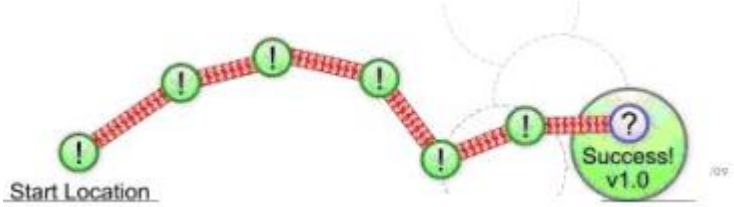
Iterative Process



- Iteration 1 – given a greenlight
    - Go towards target – but don't plan to go all the way there
    - Have a checkpoint – here, assess how the target is looking
    - Gives dev and customer a chance to re-evaluate – may aim for a new estimated target
- This process repeats many times
    - At the end of the evaluation for each checkpoint – able to orient towards a new estimated target



- Eventually reach project success
    - Compared to waterfall which took more time and was more all over the place

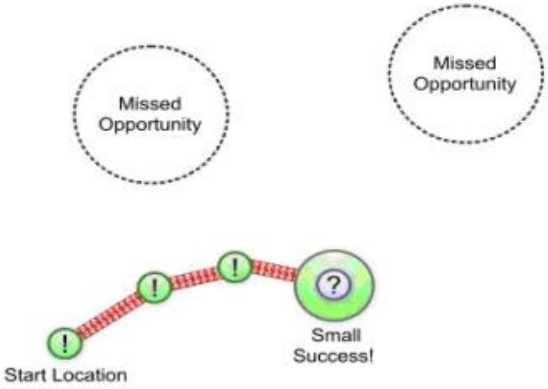


Iterative Process — The Big Difference

- Instead of 2 to 18 months to create and evaluate a concept
    - Build and show a new version to users every 2 to 4 weeks
- Requires
    - Team members close together and close to customer
    - Team members agree on good ideas over a period of hours, not months
    - Teams become experts through intense hands-on problem solving and testing
- Ends up with real systems meeting the users' needs, not their “perceived” needs

Benefits

- Favoured by small start-up companies
    - Greatly reduce the risk of a project failing
        - Only one shot at the target, steer your way to success using information instead of launching blindly into the unknown
    - Long term, iterative development delivers more value sooner, with lower overall risk
- Project is intensely focused
    - In completing only high priority features, many alternative concepts never get explored
        - Good ideas can be lost



Examples of Iterative Software Development Methods

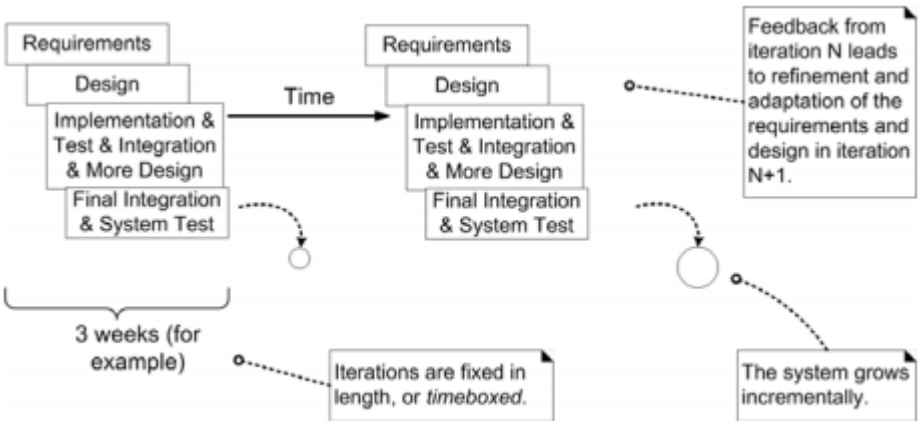
- Agile software development ("Agile Manifesto")
    - Mini software projects; face-to-face communication
- Rapid Application Development (RAD — James Martin)
    - Voice of customer; Collaborative; rigid schedule
- Extreme Programming (XP — Kent Beck)
    - Design on the fly; unit testing of all code, pair programming; refactoring
- Scrum (Takeuchi, Nonaka and, later, Schwaber)
    - Facilitated teams scrum down in short iterations (sprints); empirical process

Unified Process

Ideas

- (Rational) Unified Process (RUP or UP) is a process for building high quality object-oriented systems
- Central Idea: Iterative Development
    - The life of a system stretches over a series of cycles, each resulting in a product release

Iterative Development



- Development as a series of short mini-projects: iterations
- Each iteration gives a tested, integrated & executable system
- An iteration forms a short (2-6 weeks) complete development cycle
    - Requirements
    - Analysis
    - Design
    - Implementation
    - Integration and System Test
- Iterative lifecycle is based on the successive enlargement and refinement of a system
    - Multiple iterations with feedback and adaptation
- System grows incrementally over time, iteration by iteration
    - May not be eligible for production deployment until after many iterations

- Output of an iteration is not an experimental prototype but a production subset of the final system
- Each iteration tackles new requirements and incrementally extends the system
- An iteration may occasionally revisit existing software and improve it

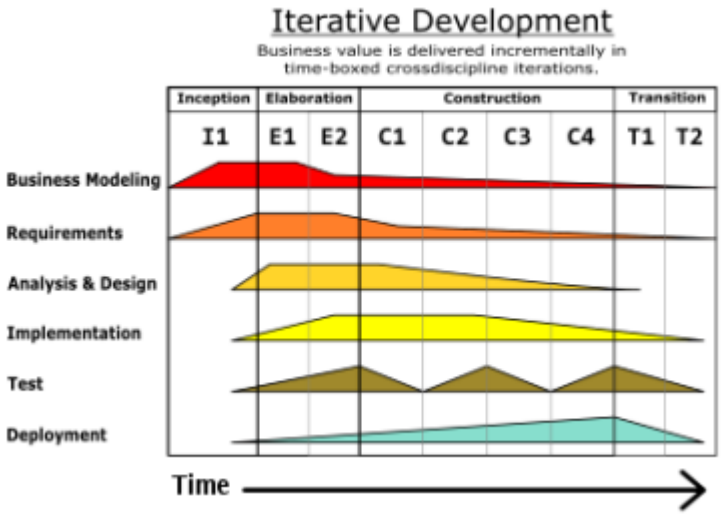
Central Unified Process Central Ideas

- Iterative Development is number one
- Others:
    - Tackle high risk items early
    - Continuous engagement of users
    - Core architecture built in early iterations
    - Continuous verification of quality – test
    - Apply use cases continuously
    - Model software with UML
    - Carefully manage requirements
        - i.e. have a clear idea of what the requirements are at beginning of iteration – and how prototype will be evolved in final deliverable prototype
    - Control changes
        - Factored into the beginning of the next iteration

Phases

Unified Process Phases

- Inception — Define the scope of project
    - Feasibility
- Elaboration — Plan project, specify features, baseline architecture
- Construction — Build the product
    - Refine vision, implement core, resolution of high risks, identify major requirements
    - Several iterations
- Transition — Transfer the product into end user community
    - Deployment, release



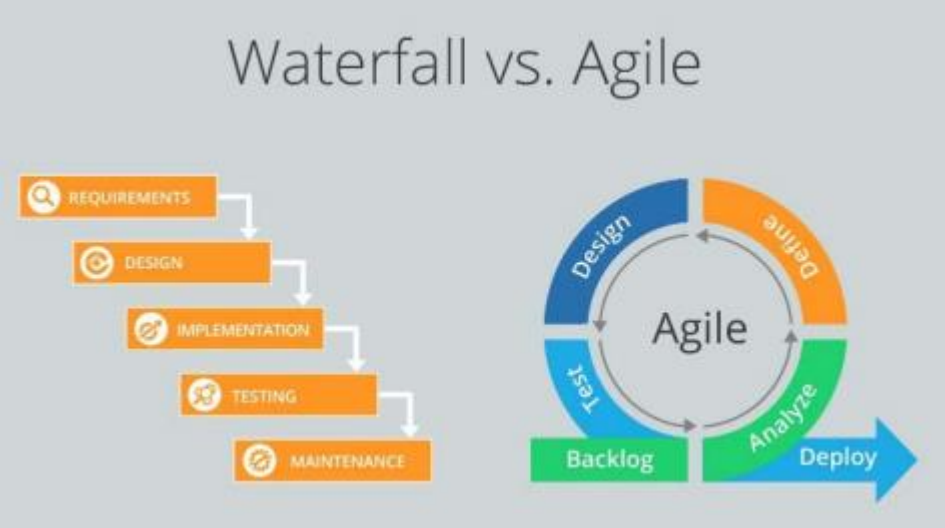
- Can see increased modelling in the beginning but decreases towards the end
- Also, with each iteration, need to fulfil less and less requirements

- Less effort gets put into beginning stages, and more put in towards the end for some phases
    - E.g. deployment

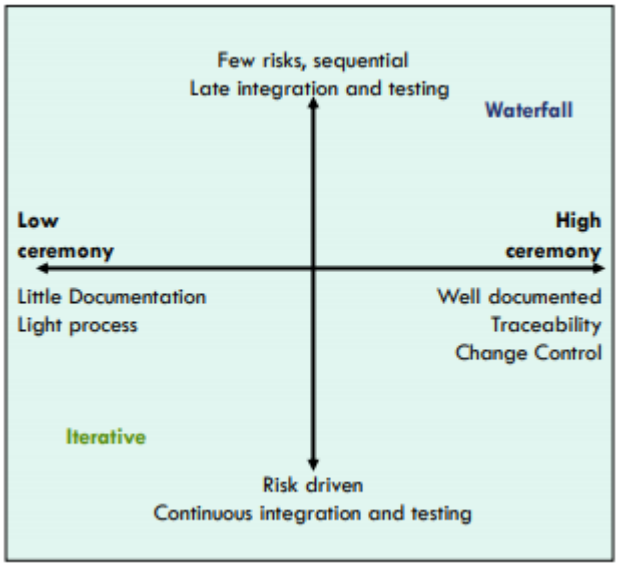
Artefacts

- Docs, diagrams, code, etc. that track our progress
    - UML, interview plans, etc
    - Produce what you need in order to carry out the phase that you're in
- Everything is optional!
- Best kept electronically on website
- Following can start in inception:
    - Use-case model, vision, supplementary specification, glossary, s/w development plan, development case

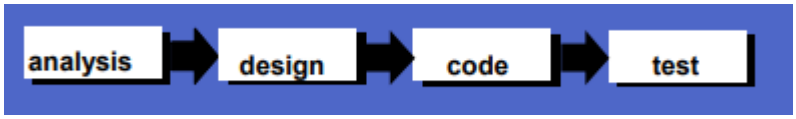
Comparison



Process Comparison

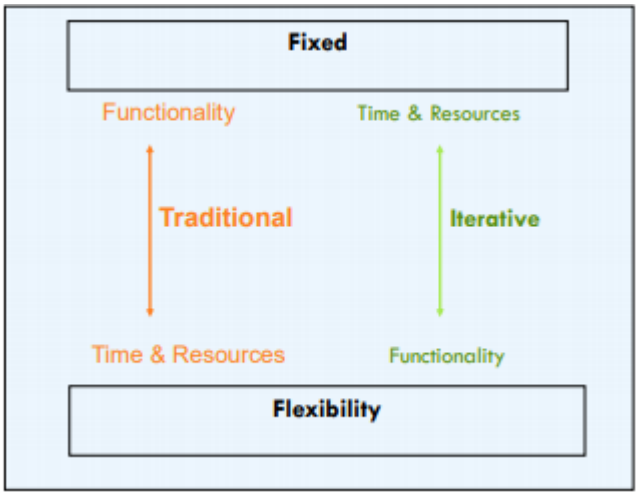


Process Models



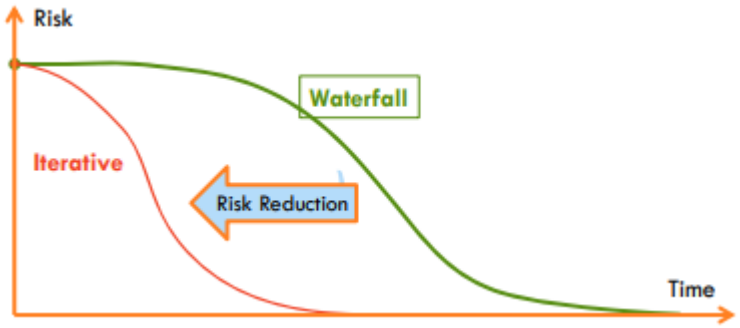
- A framework of tasks applied during software engineering:
  - Linear (Waterfall) – Based on conventional engineering
    - Based on premise that can have all requirements clearly specified upfront and can do it right the first time
  - Prototyping – Build a system to clarify requirements
    - Requires frequent check-ins with customer
  - Rapid Application Development (RAD) – Well-defined 60-90 day projects
  - Incremental – Deliver increasing functionality at each iteration
    - Develop each prototype which incrementally steps towards what the final prototype looks like
  - Spiral (Boehm) – Similar set of tasks applied for each turn of the spiral
  - Component based – Aimed at producing and reusing O-O components
  - Agile – Embrace change and adapt to it and keep things simple.

Process Comparison



- Expectation in traditional process is that functionality is fixed but time and resources are flexible
- Iterative has time and resources fixed, but functionality is flexible

Reduce Risk

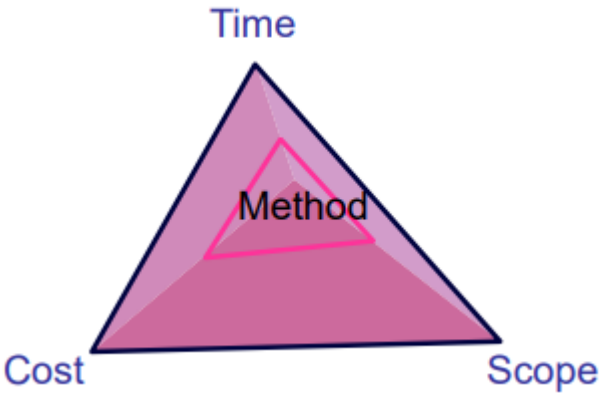


- Iterative methods attempt to reduce risk by bringing versions out early
    - i.e. reducing scope by working in iterations
- By reducing scope, allows one to focus higher value things that are worth the risk to implement, and takes away low priority things

Benefits of iterative Development

- Early reduction of risk – technical, requirements, objectives, usability, etc
- Early visible progress
- Early feedback
    - User engagement, and adaptation
    - Better meets the real needs
- Managed complexity – no very long and complex steps
- Get a robust architecture
    - Architecture can be assessed and improved early
- Handle evolving requirements
    - Users provide feedback to operational systems
    - Responding to feedback is an incremental change
- Allow for changes – System can adapt to problems
- Learn and apply lessons within the development process

Method is a 3rd Dimension



- Can use method to move us into another Cost/ Scope/ Time regime
    - i.e. adjust time/scope/cost more effectively

Conclusion

- Consider only iterative technologies
- Agile techniques
    - Small time cycles
    - Many prototypes
    - Meet user requirements
    - Timescale is adopted by the development team

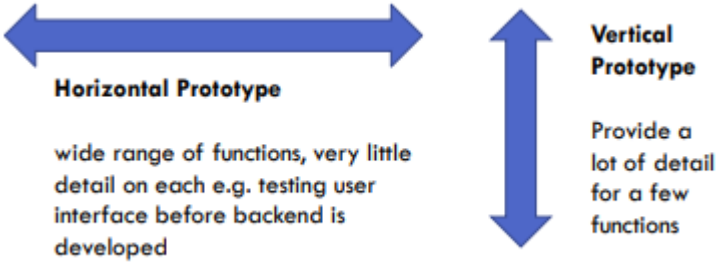
Prototyping

- What Is A Software Prototype? – Depends
  - "Initial, incomplete version of a software system being developed that is used to learn about the problem, explore designs and solution techniques."
- Helps to investigate
  - Technical issues
  - Work-flow, task design
  - Screen layouts, information display
    - e.g. user interface
  - Difficult, controversial, critical areas
  - Match between engineering and customer specification
- Demonstrate proof of concept
- More concrete than a narrative

Why Prototype?

- Enable evaluation and feedback (central to design methodology)
- Improves communication within a team and outside with clients
  - Can help clarify goals, requirements etc
- Testing ideas out – encourages reflection
- Answer questions!
- Explore alternatives

Prototypes Require Compromises



- Slow response, sketchy icons, fake data, limited functionality, limited parameters, etc

Throw-Away Prototypes

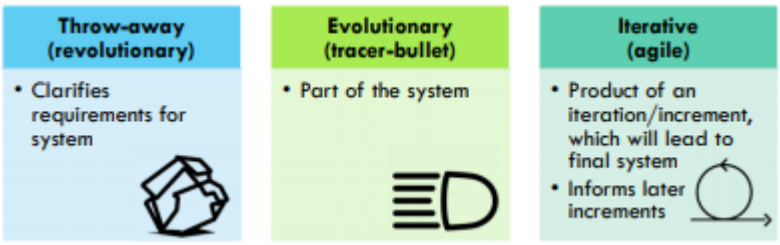
- Address high-risk issues
  - Uncertainty in requirements
  - User interface design
  - Alternative implementation strategies
  - Technology platform
- Only enough effort to help address specific issues
  - Focus only on the issue, ignore all others
  - No unit tests
  - Too much effort will make you hesitant to throw it away

- Great for trying alternative ideas

Evolutionary Prototypes

- Intended to be early, not necessarily release-able version of the actual software…
  - … will evolve into the final product
- Quality is important (unit tests are back!)
- Can be put to limited use (e.g. demo, constrained env)
- Implementing and validating well-understood requirements -> providing opportunity for change and reorienting if necessary
- Potential Weakness – customers/testers may be hesitant to criticize underlying problems in something that seems heavily invested/developed!

Prototypes Vs The Final System



The Demo

- How to beat Murphy:
  - Make sure your system works
  - Make sure your system works
  - Also practice and rehearse the demo
- How To Make Sure The System Works
  - No last-minute code changes
  - Demo stable version with fewer features rather than an untested version with more
  - Rehearse the entire demo!
  - No, really resist the temptation to tweak the demo!
  - Test any peripherals (e.g. speaker, projector, scanner)

Preparing For The Demo

- List (and test) the tasks you will demonstrate
- Plan what to say:
  - (Very) short overview of the system
  - Explain what the demo will show
- Plan what to say during each test, work out the steps involved
- Ensure that each team member participates and can demonstrate contribution to knowledge
- Rehearse and time the demo
- Think of possible questions – prepare answers
  - What has been left out of the demo and why?
  - What are the roles of the team members?

Before the Demo

- Arrive early
- Make sure the system works!

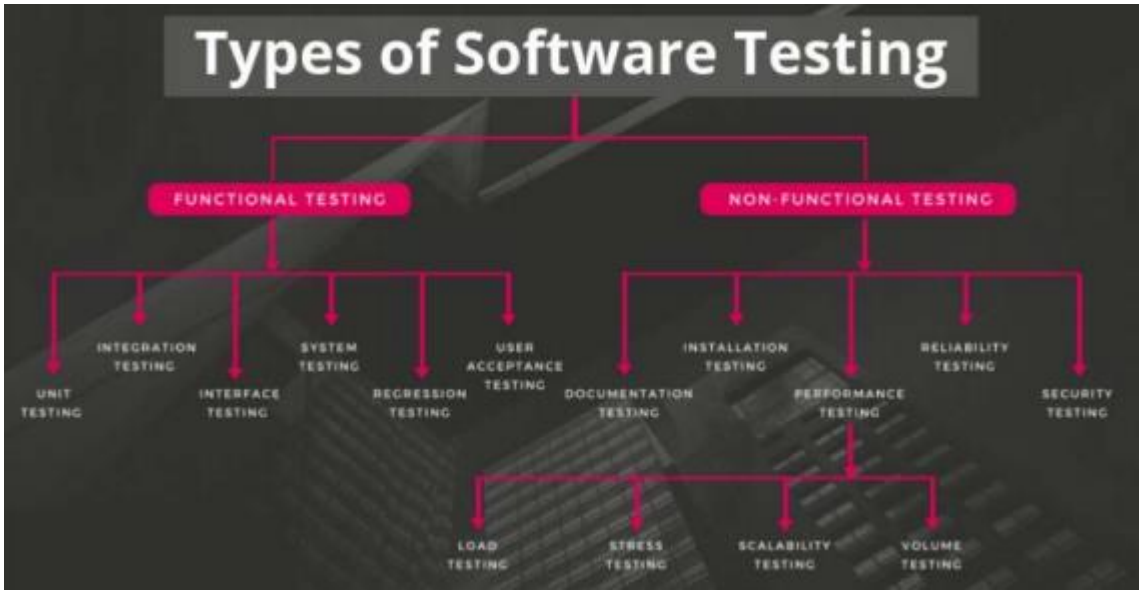
During the Demo

- Be prepared to change direction in response to questions
- Allow and encourage the client to try things (if you are prepared for this and it's safe)
- Be prepared to show code or data or documentation
- Be prepared to answer questions

UH OH, SOMETHING BAD HAPPENED

- Stay calm
- Don't make effusive apologies if anything goes wrong
  - Fix the problems quietly
  - Be able to restart the system without recompilation
  - Or have a backup system ready
  - Let one person go on explaining the system
- Don't argue with or contradict team members

Software Testing



- Focus on Functional Testing – including security testing

Unit Testing

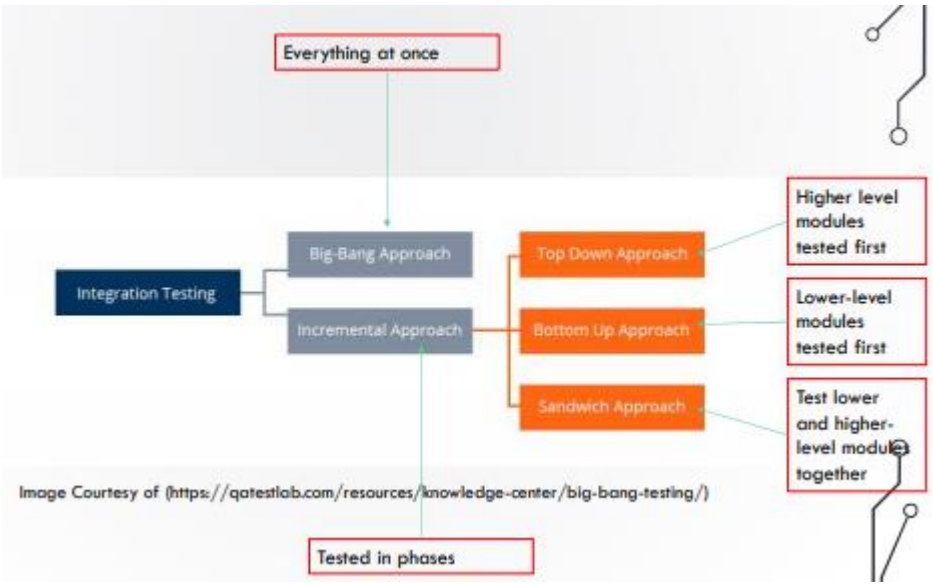
- Usually, the first level of testing done
- Develop, then test feature
- Test a single unit/component of code – e.g. a class, method etc
- White Box testing – since can see the code

Integration Testing

	executed	failed
iterations	1	0
requests	4	0
test-scripts	4	0
prerequest-scripts	0	0
assertions	4	0
total run duration: 650ms		
total data received: 1.91KB (approx)		
average response time: 80ms [min: 10ms, max: 265ms, s.d.: 106ms]		

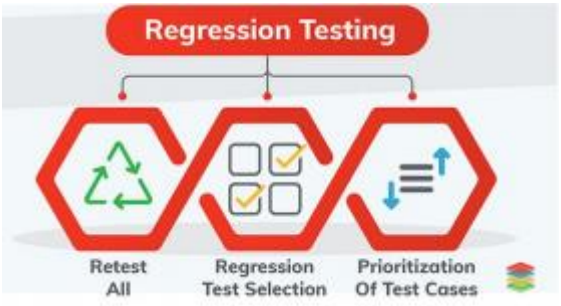
- Tests how different software modules integrates with each other and makes sure that the functionality remains the same and correct

Types of Integration Testing



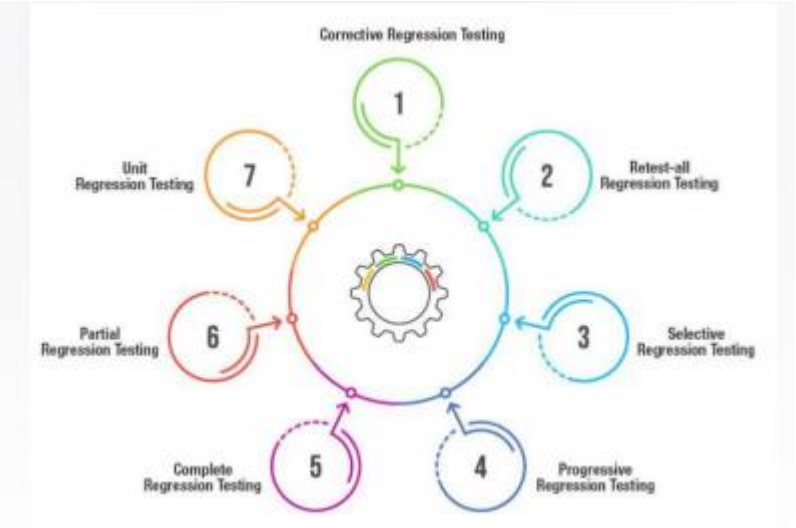
- Big bang: integrate all the features and test all at the same time
    - Very risky
- Incremental: test in phases
    - Bottom up - test the lower level modules such as connecting to the database before we test the login method that reads from the database
    - Top down – higher level modules tested first
        - First test the login method, and then test if we can connect to the database
    - Sandwich approach – choose sections to test together
- Different types work for different implementations

Regression Testing



- Conducted as we add new features to ensure that the addition of these new features did not break previously developed features

Types of Regression Testing



1) Corrective Regression Testing: used when there are no changes introduced in the product's specification with existing test cases that we can reuse to conduct the test.
2) Retest-all Regression Testing: very tedious and tends to waste a lot of time.
    a. The strategy involves the testing of all aspects of a particular product as well as reusing all the test cases.
3) Selective Regression Testing: is done to analyse the impact of new code added to the already existing code of the software.
    a. When this type of regression testing is conducted, a subset from the existing test cases is used, to reduce the effort required for retesting and the cost involved.
4) Progressive Regression Testing: works effectively when there are certain changes done in the program specifications as well as new test cases are designed.
    a. Conducting this testing helps in ensuring that, there are no features that exist in the previous version that has been compromised in the new and updated version.
5) Complete Regression Testing: best to be used in case there are multiple changes that have been done to the already existing code.
6) Partial Regression Testing: done to test issues when new codes are added to already existing code.
    a. The idea behind partial regression testing to make sure that a system is performing as it is supposed to be after addition of new code.
7) Unit Regression Testing is the most important part of unit testing.
    a. Mostly conducted in isolation, mainly focused on code unit and all the dependencies and interactions will be blocked at the 15 time of test

User Acceptance Test

- Final phase of testing to ensure that the software is ready for production by:
    - Making sure it is what the user wanted
    - Is usable and user friendly
- Alpha Testing
    - Often conducted in software house or with “super users”
    - Software not production ready at this stage
- Beta testing
    - Conducted with actual users of the system
    - Software production ready

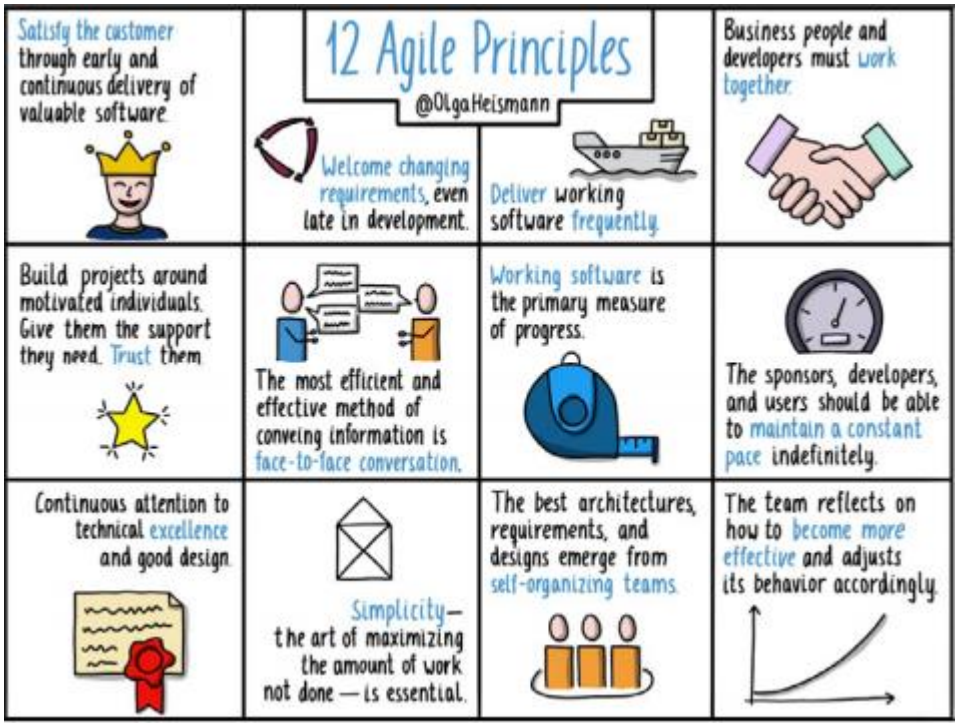
Security Testing

Agile Software Processes

Agile Principles

- Agile Software Development
  - Put the software being developed first
  - Acknowledge that user requirements change
  - It is agile because it can respond quickly to the users' changing and emerging needs
  - Advocates frequent and regular, software releases
    - Users can respond quickly to these releases, changing requirements
- Agile Manifesto

  - While there is value in the items on the right, we value the items on the left more
  - This clearly distinguishes Agile from more documentation-focused processes like rational unified process and waterfall



Team Empowerment

- Teams must be empowered
- The agile project team is core to the success of agile
    - They must be trusted, feel ownership over the product and customer needs, and be able to work together effectively
    - These teams benefit from individuals that are agile, that can shift between roles as necessary, and work well together
- The project team must have sole responsibility to deliver the product
- Any interference with the project team is disruptive and reduces their motivation to deliver
- The team must together
    - Establish and clarify the requirements
    - Prioritise them together
    - Agree to the tasks required to deliver them
    - Estimate the effort involved
- It ensures the buy-in and commitment from the entire project team from the outset
- When challenges arise, the team feels a real sense of ownership

Bare Requirements

Pareto's Law

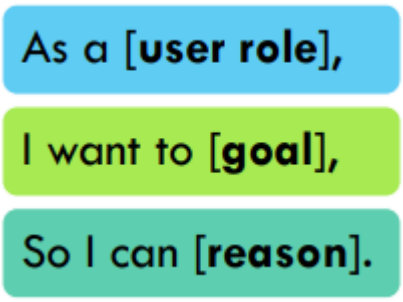
- Typically, 80% of your results may actually come from only 20% of your efforts!
    - Try to apply the 80/20 rule, and focus on the important 20% of effort that gets the majority of the results
- The difficult question is:
    - Can you see initially which 20% is the important 20%?
        - The 20% that will deliver 80% of the results
        - In very many cases, the answer is NO

- Caveat – if you target the 80% population, consider who you are excluding
    - Bells and whistles often unnecessarily exclude bandwidth-constrained communities

Agile Requirements

- Agile requirements are barely sufficient
- Capture requirements at a high level and on a piecemeal basis
    - Just-in-time for each feature to be developed
    - Barely sufficient – the minimum to enable development and testing
        - Minimise the time spent on anything not part of product
- Understand enough to determine (and prioritize) the scope and for high level budgetary estimates
- Captured in collaborative workshops so that all team members understand the requirements
    - Allows everyone to contribute, challenge and understand what's needed and why

User Stories



- Most Agile teams represent each requirement as a user story
    - Similar to Use Cases but lightweight and simpler
    - Simple statement about what a user wants to do with a feature
    - Should be written in business language understandable to all
- Should focus on the who, what and why of a feature, not how

On a job listing site, two high-level User Stories might be:

- As a job seeker, I want to search for a job, so I can advance my career.
- As a recruiter, I want to post a job vacancy, so I can find a new team member.

- At the start of a project, capture an initial list of User Stories up-front
    - Useful for estimating and planning
- Defer capturing the details until the story is prioritised and due to be developed
- Users often tell stories
    - About the failings of their current system
    - How they see things working better in future
    - Capture these stories as User Stories – as they are told
- In traditional development projects, these stories are captured in a lengthy analysis process and available in a lengthy document;

- Not user friendly

Recording User Stories

- Written on postcard size cards in 3 parts:
  - Heading
    - Name/description of the user story, reference numbers, estimated size, etc
  - Conversation (on the front of the card)
    - Info about the user story + what system is meant to do
      - Sketch/diagram of the feature,
      - Notes about how it should function
  - Acceptance Criteria (on the back of the card)
    - Test cases to help identify scenarios that users, developers and/or analysts may not have thought of
- Writing User Stories on a card ensures requirements are broken into small, manageable pieces of functionality
  - Individual features
- Cards can be supported by documentation, but keep it to the bare minimum to allow a feature to be developed, and always in very small units
- Requirements should be broken into tasks of <=16 hours. Preferably 8 hours, so progress can be measured daily
- All items are deliverables not activities or tasks
  - You can see a deliverable to judge its quality and completeness
  - A task you cannot

Incremental Design

- Agile does not follow a top-down design method
  - Top-down design says time in design is worth it to save cost of reworking the design many times
- Agile design is always the same size as the system
  - "You can't possibly anticipate the problems and alternatives that will arise once you start coding"
- If a new feature comes along that requires major changes then that is the trade-off for the flexibility it allows
  - Perhaps this feature was not even known at the beginning anyway
  - Or it might have gone away if we knew of it at the start

Fixed Timescale

- In agile we allow the scope to be flexible in order to ensure that the project has a fixed cost and timescale
- In Agile Development, requirements evolve, but timescales are fixed
- Contrast to traditional development:
  - Capture all known requirements
  - Changes are subject to change control
  - Users are told it's much more expensive to change or add requirements during or after the software is built

        - It becomes imperative to include everything they can think of, everything they ever dreamed of!
- Normally
    - Users may actually use only 20% or less of the product
    - Many projects start with a bloated scope
        - No-one is sure at the outset which 20% they will use
    - It is impossible to think of everything, things change, and things are understood differently
- Agile Development assumes that requirements emerge and evolve
    - However, much analysis and design you do, you cannot really know what you want until you see and use the software
    - In the time spent analysing and reviewing requirements and designing a solution, external conditions could change

Fixed Budget

- What does business expect from development teams?
    - Deliver the agreed business requirements
    - On time and within budget
    - To an acceptable quality
        - In Agile Development, it is the scope that is variable, not the cost and timescale
- For this to work, it's imperative to start development with the core, highest priority features
    - Delivered in the earliest iterations
- As a result
    - Business has a fixed budget,
    - Based on affordable resources and
    - Can make plans based on a certain launch date
- Fixed timescale, fixed budget means project itself is predictable from budgetary sense, and that it will meet a minimum requirement to be delivered, with the core features

Working Product

- Key is to have a working product at all times

Agile Development Cycle

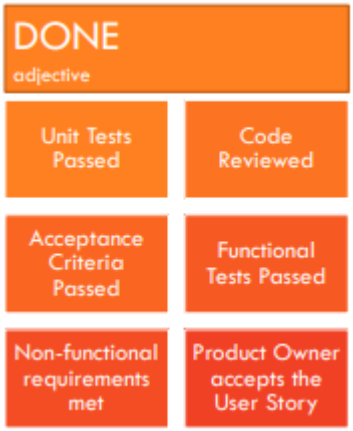
- The cycle is Analyse, Develop, Test; Analyse, Develop, Test; and so on
    - Doing each step for each feature, one feature at a time
- Advantages of this approach include:
    - Reduced risk – doing a small thing and testing it thoroughly
    - Increased value – can prioritise some things, delivering some benefits early
    - More flexibility/agility
    - Better cost management
- Each feature must be fully developed, to the extent it can be shipped
- Develop features in priority order

Frequency

- How frequent is frequent

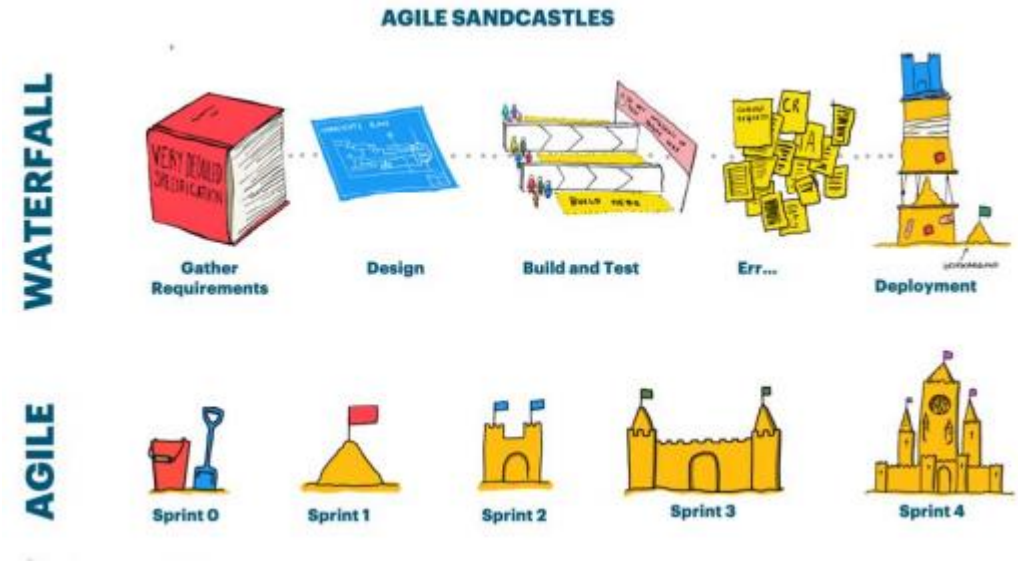
- Competitors won't wait
- Speed-to-market – a significant competitive edge
- The value of first-mover advantage is enormous
    - Research shows 80% of first to market end up market leaders
- So how frequent is frequent enough?
- There is no right or wrong answer
    - Decide what's appropriate; stick to a regular release cycle
    - Allows you to plan
    - Allows your infrastructure and ops teams to plan
    - Allows your business colleagues to plan
    - Allows launch events, marketing campaigns, etc to be planned
- BUT – frequent releases of buggy software can really irritate customers

Done Means Done

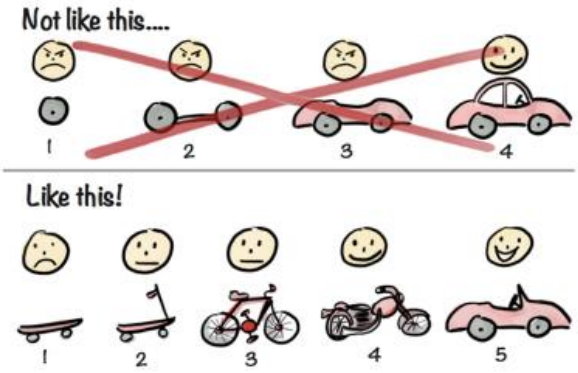


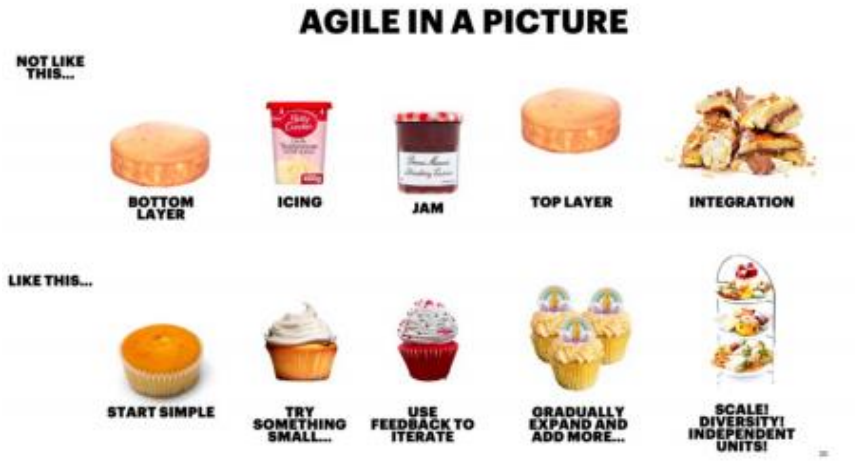
- Features developed in an iteration, should be 100% complete by the end of the iteration
    - Ideally, each iteration results in a release
- In Agile development, “DONE!” means shippable
    - In practice a feature may rely on other features being completed before the product could really be shipped.
    - But the feature on its own merit should be shippable
- Completing each feature before moving on to the next ensures the system is not in a state where multiple features are 90% complete or untested, as in traditional developments

The Working Product



- What does it mean to have a working product at all times?
    - Meaning 1 – a software product should always be in a working state
        - Not always functionally complete, just that it works and has high quality
    - Meaning 2 – the emphasis is on producing a working product and shipping it
        - Not on producing documentation that might lead to a product
- The best way to get user feedback is to give a product even if it is only work in progress.
- Prototypes are better than a document
- Effort spent getting the product back to a working state is a missed opportunity to be doing valuable work





Prototypes

- Prototype solutions to risky problems helps to increase the chance of having a working product
- Prototypes: an inexpensive way to try out ideas so that as many issues as possible are understood before the real implementation
- Two main classes of prototypes
    - The true prototype
        - Test implementation to understand a problem before it is implemented for real
    - Tracer bullets
        - Prototype that is intended to gradually turn into the final solution

Continuous Integration

- An important discipline is to continuously integrate changes
    - Frequent integration helps to ensure that modules will fit together;
    - And also that the product continues to work with all the changes
- Developers have the bad habit of checking out a number of files and not checking them in again until their work is done
    - Developers should integrate their work daily.
    - This gradual introduction of changes ensures that integration problems or regressions are caught early

Nightly Builds

- Software should be completely rebuilt from scratch daily.
    - The result of the build should be an installable product image.
- The build should include as many automated tests as possible to catch integration problems early
    - If the build or tests fail, fix the problems first thing
    - Don't let anyone integrate any additional work until after the build succeeds again
    - There is a risk of multiple bad changes accumulating that will jeopardize the quality of the product

Performance

- Don't Neglect Performance
    - Some people feel that code clarity is more important and that you should get the code clarity right first and then optimize the 1% to 3% of code that needs it.
    - Others feel that you should code for performance first, because if you don't, your code will always be slow

Pair Programming

- Programmers work in pairs, sit together to write every line of code
    - 2 programmers + 1 computer = atomic unit of XP code development
    - One person at the keyboard, other supporting
    - Pairs are created dynamically
    - Egoless development

Advantages

- Informal review process: each line of code is looked at by at least two people
- Common ownership of code
    - Individuals are not held responsible for problems with the code
- Collective responsibility for the system
    - Team has collective responsibility for resolving problems
- Spreads knowledge across the team
    - Reduces risk if someone leaves
- Motivates refactoring as the whole team will benefit from it.
    - Refactoring is effort expended to get to where you started but in a better position for later

Productivity and Pair Programming

- Productivity is similar to that of two people working independently …
    - Fewer false starts and less rework
    - Errors avoided due to informal inspection -> less time repairing bugs
- Pairing experienced with inexperienced developers helps with training
- However – for experienced developers there may be a significant loss of productivity, for some quality benefits
- Pair for Complex Tasks, but not necessarily for the straightforward ones

Challenges

- Working on with someone else's schedule can be exhausting!

- Constant communication requires empathy and interpersonal skills
- Different skill levels makes it difficult for pairs to estimate contributions and pace
- Power dynamics (e.g. gender, seniority, race, degree)
- No time for yourself (important for introverts)
- Rotations lead to context switching

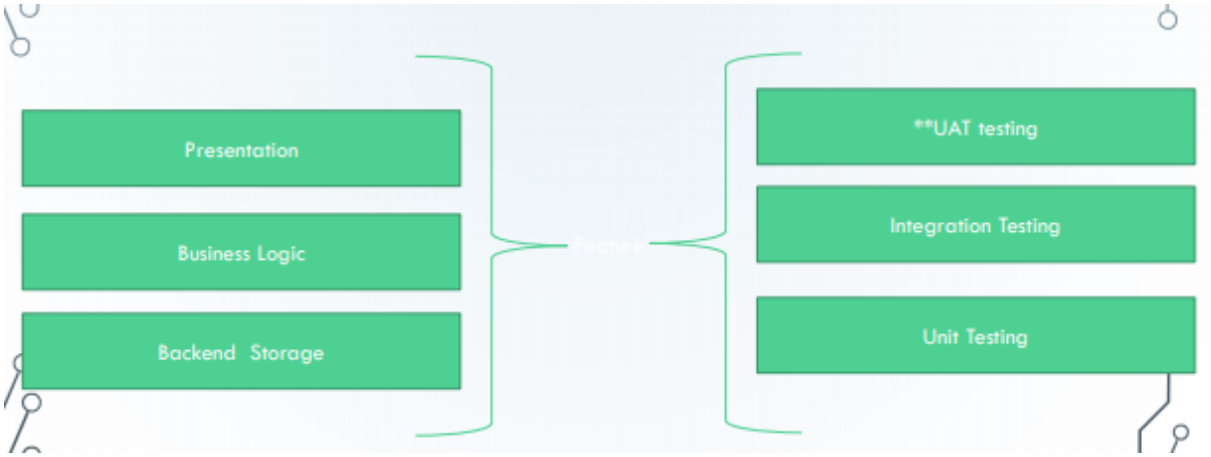
Code Review vs Pair Programming

- A good code review process also ensures that every change is reviewed by at least one other
- The challenge is:
    - Sloppy or superficial reviews
    - Over-reliance of developer on the code reviewer to catch minor problems
    - Sunk cost – we can be reluctant as code reviewers to tell the developer to change something they thought was done

Agile Testing

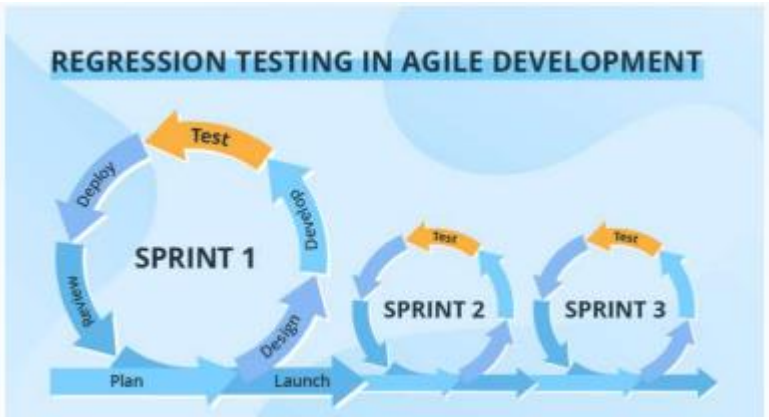
Testing in Agile Development

- Testing is integrated throughout the lifecycle;
    - Testing the software continuously throughout development
- Agile development does not have a separate test phase
- Developers write automated repeatable unit tests
    - Testing done as part of the build
    - Ensures all features are working each time a build is produced.
        - Builds should be regular, at least daily
    - Integration is done as you go too
- These actions keep the software in a releasable condition throughout the development,
    - Can be shipped whenever appropriate
- For each feature developed, have to follow a tiered module approach:



    - E.g. for a login – need to see that data can be pulled from database
        - Then can client use the business logic that has been coded in to log in
        - ….etc

Regression Testing in Agile Development



- In sprint 1 we test, ensure all tests pass, then move to sprint 2

- In sprint 2 we run sprint 2's tests, but also sprint 1's tests, and so on, as you move through the various sprints

Test-First Development

TEST CASE DESCRIPTION FOR CHICKEN BURGER

TEST CASE PLANNING AND EXECUTION TEMPLATE

TRY smartsheet for FREE

TEST TITLE	PRIORITY	TEST CASE ID	TEST NUMBER	TEST DATE
CheeseBurger	Urgent	CHB01	CHB001	5 Aug 2021
TEST DESCRIPTION		**TEST DESIGNED BY**	**TEST EXECUTED BY**	**EXECUTION DATE**
Testing the addition of the chicken burger feature		Sarina Till	Sarina Till (not good)	17 Aug 2021

TEST DESCRIPTION	TEST DEPENDENCIES	TEST CONDITIONS	TEST CONTROL
Chicken Burger Test	Burger class		Cheese /Mushroom B

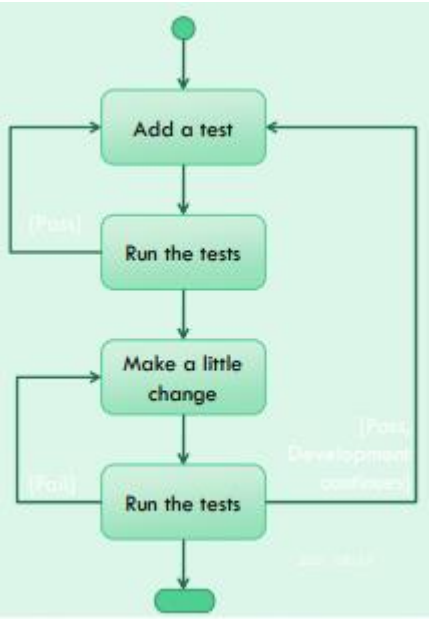
STEP ID	STEP DESCRIPTION	TEST DATE	EXPECTED RESULTS	ACTUAL RESULTS	PASS / FAIL	ADDITIONAL NOTES
1	Bun returns Gluten Free	17 Aug 2021	String "Gluten Free"			
2	Patty returns chicken	17 Aug 2021	String "Chicken"			

- Writing tests before code clarifies the requirements* to be implemented
- Tests are programs rather than data
  - Executed automatically
  - Usually with a testing framework such as Junit
- All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors

Involvement

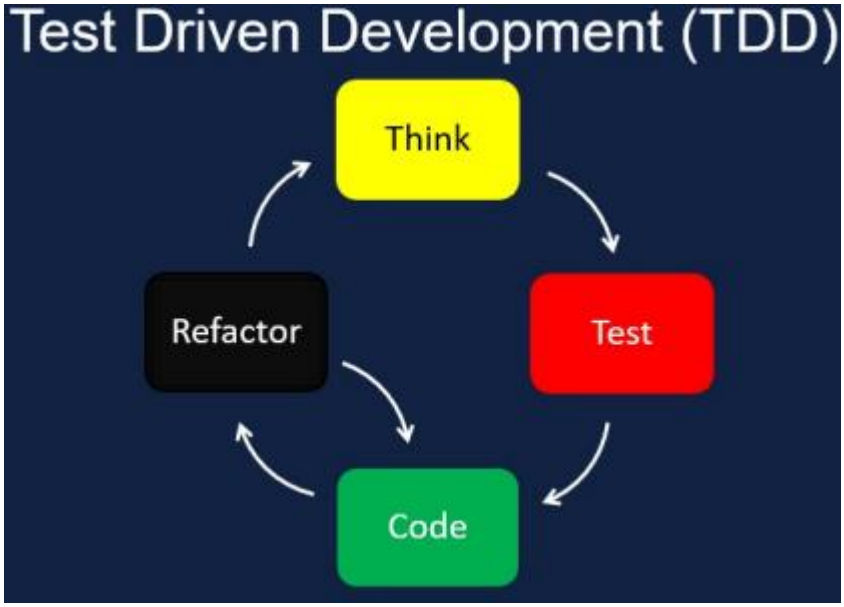
- Role of the customer in testing is to help develop acceptance tests for the stories implemented in the next release of the system
- All new code is therefore validated to ensure that it is what the customer needs
- Customers have limited time available
  - Cannot work full-time with the development team
  - May feel that providing requirements was enough of a contribution
  - May be reluctant to get involved in the testing process

Test-First Development



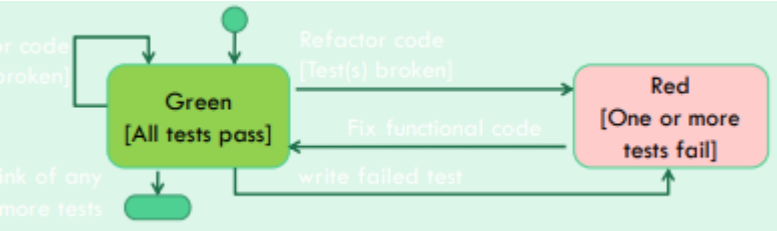
- Quickly add test for new feature
  - Just enough code to fail
- Run your tests
  - The complete test suite or
  - For speed – a subset, to ensure that new test does in fact fail
- Update the functional code to make it pass the new tests
- Run the tests again
  - If they fail update the functional code and retest
- Once the tests pass start over
  - Possibly refactoring any duplication out of the design

Test Driven Development



- TDD can be described as:

    - TDD = TFD + refactoring
- TDD turns traditional development around
    - Instead of writing functional code first and then your testing code as an afterthought
    - You first write your test code before your functional code
- Also, you do so in very small steps
    - One test and a small bit of code at a time
- With TDD a developer refuses to write a new function unless there is a test that fails because that function isn't present
    - Refuse to add even a single line of code until a test exists for it
- Once the test is in place, do the work required to ensure that the test suite now passes.
- Once your code works, refactor it to ensure that it's remains of high quality
- The diagram shows how you perform test driven development:



Rules of Test-Driven Development

- (1) Write new code only when an automated test fails
- (2) Eliminate any duplication – generates complex individual and group behaviour
    - Some technical implications are:
        - You design organically, with the running code providing feedback between decisions.
        - You write your own tests because you can't wait 20 times per day for someone else to write a test.
        - Your development environment must provide rapid response to small changes
        - Your designs must consist of highly cohesive, loosely coupled components
            - This makes evolution and maintenance of the system easier

Unit Tests

- Implication: developers need to learn how to write effective unit tests
- Experience is that good unit tests:
    - Run fast
        - They have short setups, run times, and break downs
    - Run in isolation
        - You should be able to reorder them
    - Use data to make them easy to read and understand
    - Use real data when they need to
        - Copies of production data.
    - Represent one step towards your overall goal

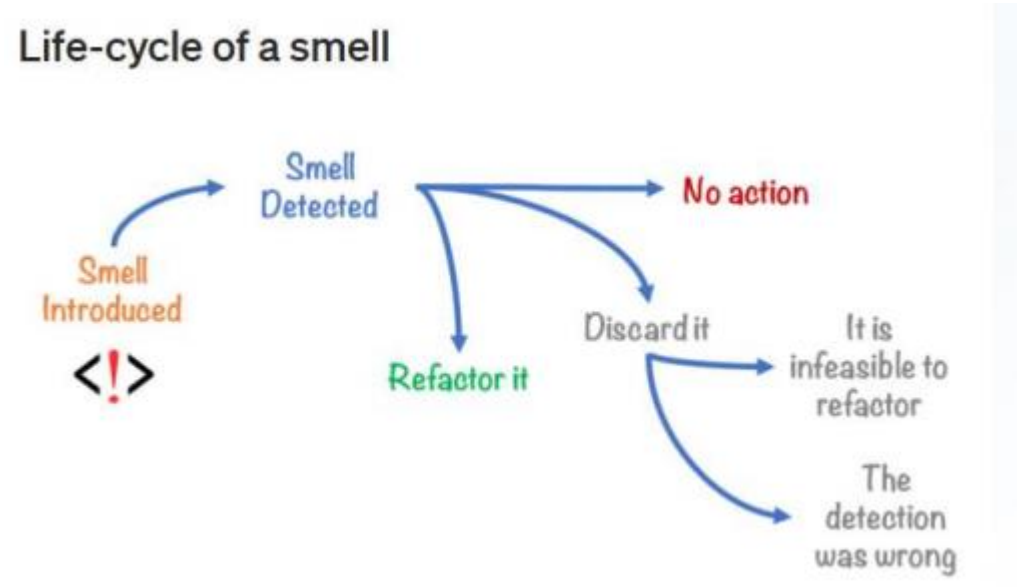
Refactoring

Technical Debt

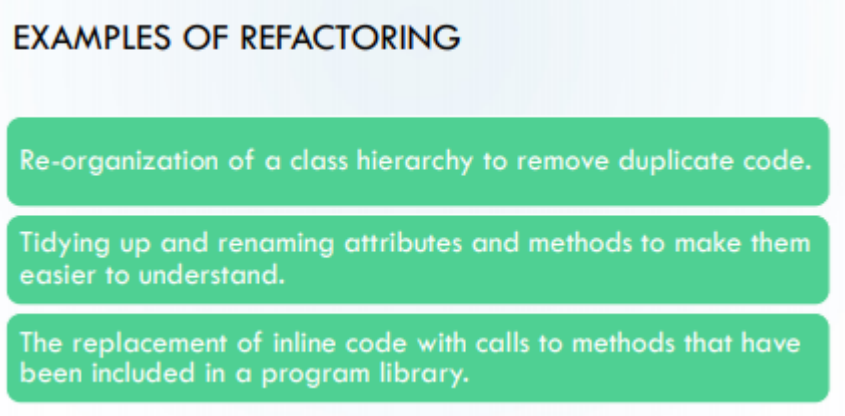
- "Technical debt (also known as tech debt or code debt) describes what results when development teams take actions to expedite the delivery of a piece of functionality or a project which later needs to be refactored. In other words, it's the result of prioritizing speedy delivery over perfect code."

Code Smells

- "Code smells are not bugs or errors. Instead, these are absolute violations of the fundamentals of developing software that decrease the quality of code."



Refactoring



- "Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design"
    - Just fix the code that is currently there
- A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour
- Adding functionality does not change existing code, it only adds new capabilities
    - Measure progress by adding tests and getting the tests to work
- Refactoring does not add functionality, you only restructure the code
    - Don't even add any tests — only restructure the code
- These software improvements are made before there is an immediate need for them

Advantages

- Improves the design of sw
    - Deals with Software rot, decay and loss of structure
    - Refactoring is like tidying up the code.
        - Regular refactoring helps code retain its shape
        - You refactor code that works but is not ideally structured
    - An important aspect of improving design is to eliminate duplicate code
        - Ensure the code says everything once and only once
        - Mode code => harder to mod correctly => more code to understand
        - "Change this bit of code here, but the system doesn't do what is expected because you didn't change that bit over there that does much the same thing in a slightly different context."
    - Changes are easier to make because the code is well-structured and clear
- Makes sw easier to understand
    - Improve the understandability and readability of the software
        - Reduces the need for documentation
        - Good programmers write code understandable by human beings!
    - After code is written it has to be maintained
        - Someone will try to read the code to make changes
        - It matters if it takes a programmer a week to make a change that would have taken an hour if she had understood your code
    - When you are trying to get the program to work, you are not thinking about that future developer
        - It takes a change of rhythm to make changes that make code easier to understand.

        - Refactoring leads to higher levels of understanding that would otherwise be missed during development
- Helps find bugs:
    - By clarifying the structure of the program, you clarify certain assumptions you've made
        - To the point at which even you can't avoid spotting the bugs.
- Helps you program faster

    - Able to move through these stages much faster

Refactoring Categories

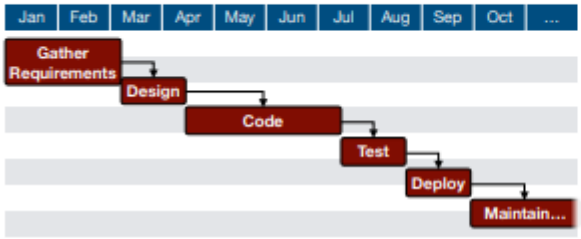
- Composing methods
    - These refactorings serve restructurings at the method level
- Moving features between objects
    - These refactorings support the moving of methods and fields between classes
- Organizing data
    - These refactorings restructure the data organization
- Simplifying conditional expressions
    - These refactorings simplify conditional expressions
- Making method calls simpler
    - These refactorings simplify method calls
- Dealing with generalization
    - These refactorings help to organize inheritance hierarchies

Refactoring to Patterns

- Refactoring to Patterns is the marriage of refactoring with patterns
    - Patterns ≡ classic solution to recurring design problems
- Use patterns to improve an existing design
    - Better than using patterns early in a new design

Scrum and Agile

Historical "Waterfall" Process



"I do my part, then hand off to you"

- Works well if requirements are fully known and documented at the outset (When does that ever happen!?)
- Compartmentalized—requires little interaction or iteration among teams (Who doesn't need to communicate today!?)

No longer fits our connected, fast-paced, iterative world

Negatives

- Strong code ownership; maintainability can suffer if a key developer leaves
- No back-and-forth communication between teams at each phase of project
- Development cycles are long
- Customer feedback is left to the end
- Failures are costly and difficult to predict

Agile

- Umbrella term for the mindset behind many kinds of modern iterative software development methodologies
- Hallmarks include small teams doing small chunks of work, customer-oriented design, and collaboration among teams
- Agile practices include Scrum, Kanban, etc
- Can only estimate so far with so much accuracy – agile capitalises on this

Customer-Oriented Design

- Learn about your customer so you can think from their perspective ("customer empathy").
  - What does the customer want?
- Keep an open "conversation" with customers (forums, feedback, fix bugs)
- Prioritize work that solves important customer-facing problems
- Limit work that does not impact the customer; fit it in when it fulfills a larger customer-impacting effort
- Speed and timing matter; perfect architecture / perfect code doesn't exist
  - Always try to ship!

Example

- Large problem—break it down.
  - Want to estimate time to solve the whole thing—might take weeks or months if we're honest
- Break into high-level features, then into small tasks that can be completed in a week or less

- Prioritize important pieces first.
    - When one is completed, move on to the next most important piece
- Include some friends (work on a team) to reach the goal sooner.
    - Agree on goals for each piece before starting any work, then check on progress daily
- Every couple weeks, talk about how it's going, demo your progress, and take time to redistribute the work if necessary.

Stories, Points, & Epics

- Epic - a high-level feature;
    - A product owner creates and breaks down each into a collection of User Stories during Backlog Grooming
- User Story - a small, user-facing feature
    - Follows the formula: "As a user, I want …"
- Spike - a story only for the purpose of learning how (not) to do something
    - A prototype
- Acceptance Criteria / Definition of "Done" (DoD) - the conditions that must be satisfied for a User Story to be "done"
- Story Points - a rough estimate of effort to complete a user story; incorporates known effort as well as uncertainty
    - Assigned using Fibonacci numbers (1, 2, 3, 5, 8, 13, …)

Scrum

- Scrum: an agile methodology conducted in short periods of work called "sprints," after each of which software is expected to be "shippable."
- Sprint - each lasts a fixed number of weeks (usually 1-4) and is marked by several ceremonies or rituals
- Helps focus effort and keep out distractions, improving progress toward completing the goal

Scrum Roles & Responsibilities

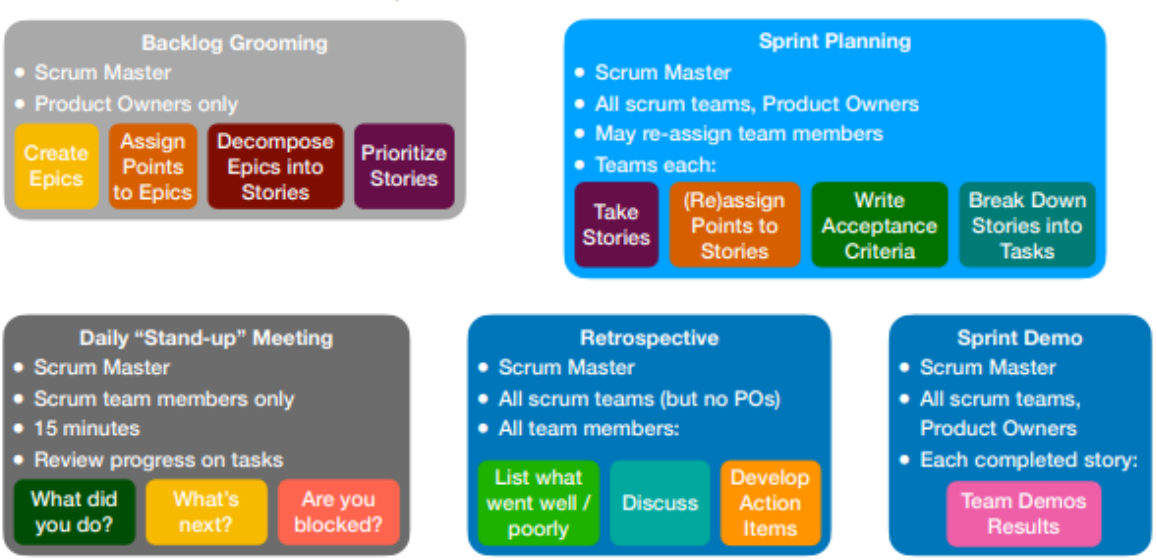
- Scrum Master - facilitates all ceremonies
    - Ensures that what need to happen during a sprint happens
- Product Owner - ultimately responsible for owning what the customer wants and what the team ships
- Scrum Team - made up of individual contributors; engineers, designers, testers, etc
    - Develop Acceptance Criteria with input from Product Owner / Designers / Testers
    - Break down User Stories into tasks
    - Implement tasks (lots of ways: in teams, as individuals, in pairs)
    - Verify tasks are "done"

Scrum Ceremonies

- Once per sprint a Scrum Master facilitates the following meetings:
    - Backlog Grooming - POs prioritize and break down items in the backlog
        - E.g. take an Epic and break it down into subcomponents, user stories and prioritise them
            - E.g. as an HR user, I want to be able to delete employees from database

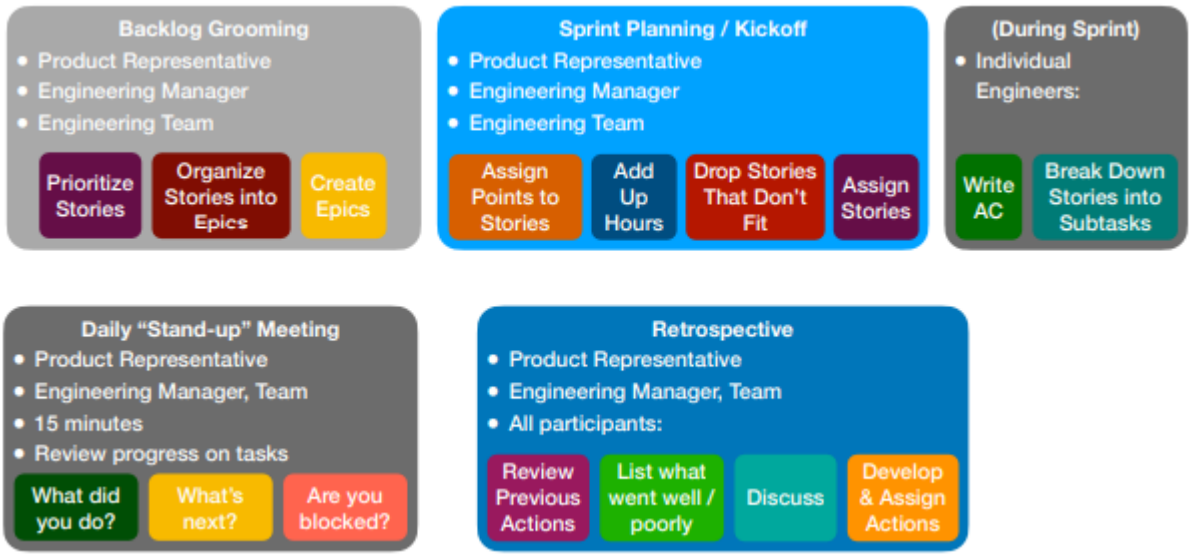
        - Assign duration of time that you expect a given user story to take – i.t.o story points, not days
            - E.g. team can do 20 story points this sprint
    - Sprint Planning - POs and team members select items from the backlog, agree on Acceptance Criteria, do task breakdown, and assign tasks
    - Retrospective (Retro) - Team members (without POs) discuss successes as well as areas to improve; formulate plans to enact said improvements
    - Demonstration (Demo) - Teams or team members show the results of their work to POs

Many Scrum Teams



- Larger engineering departments or companies
    - May have a Scrum Master or dedicated facilitator of Agile ceremonies
    - Have more personnel with overlapping skillsets, allowing for more scrum teams with cross-disciplinary membership
        - Each team able to have multidisciplinary compositions with duplicates
            - E.g. each term has a server expert, UI expert, etc
        - This composition allows for the completion of an entire user story, from end to end
    - Often Design, Testing, or Documentation personnel serve across all of the scrum teams, rather than become members of a particular team

Fewer/Smaller Scrum Teams



- Smaller engineering departments or companies
    - May not have a Scrum Master or other specialized roles (Design, Testing, Documentation), leaving those roles to engineers or managers
    - May have fewer teams or teams with fewer members
    - May have teams or team members dedicated to a specialized area, rather than teams of generalists (e.g. audio or mobile expertise)
    - Divide user stories among people, not teams
    - Volatility with planning – if someone gets stuck, there's no one with similar expertise to bail them out
        - User stories might then be dropped

Non-Scrum Agile (Kanban)

- Some feel the sprint boundary of Scrum is artificial and unhelpful

    - Rather want to operate in a continuous loop of taking on user stories
- It is possible to measure completed work continuously as a velocity (story points completed/day) instead of tallying this only at the end of a sprint
    - Want to keep a velocity under a certain amount for a team
- Measuring velocity tells POs how far through the backlog a team will get by a certain date

Conclusion

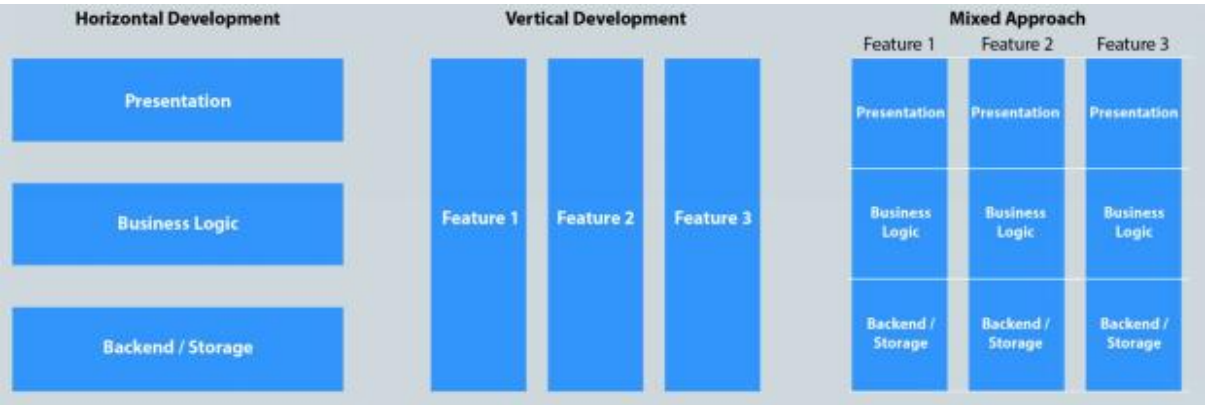
- Agile is user-focused and uses Epics/Stories, Points, Acceptance Criteria
- Scrum is one kind of Agile methodology that uses fixed duration sprints
- Scrum has roles, ceremonies, and sprints that aim to keep out distractions
- Other Agile methodologies manage work and progress differently (velocity)
- Goals: Improve quality and predictability – build what the customer wants today, not what they thought they wanted months or years ago

Scrum Development

Being a Scrum Master

- Key – Scrum teams need to be cross-functional, need to grow and work
    - Scrum master encourages team to grow and develop cross functional skills

Vertical Features



- How a scrum team measure their delivery
- Horizontal Development

    - Different layers built independently
    - In a waterfall approach – build storage, then business logic, then presentation
- Scrum uses the mixed approach
    - Try and build all the layers in parallel
-

Scrum Artifacts

- Increment: the collection of the Product Backlog Items that meet the team's Definition of Done by the end of the Sprint.
    - The Product Owner may decide to release the increment or build upon it in future Sprints.
- Definition of Done: a team's shared agreement on the criteria that a Product Backlog Item must meet before it is considered done
    - Need to have feature "client-ready"
    - Differs from team to team

Key Scrum Roles

- Product Owner
    - Managing the Backlog (Prioritization)
- Scrum Master
    - Servant Leader - ensure the team follow the values and is fully functional
- Development Team
    - Do the work!
    - Build code, test code, deliver product features

Scrum Processes

- Product Backlog – domain of product owner
    - Get to add to backlog, decide what to prioritise
    - Backlog grooming session – team gathers with product owner to get more detail, ask questions about features on backlog
        - Try to get the right number of features into a sprint
- Sprint planning meeting – team gets together

    - Start from top of backlog and work down
    - Team commits to deliver features for product owner
- Sprint backlog – used for period of the sprint
    - 1-4, typically 2 weeks
- Have a daily scrum
    - Allows one to see interdependencies between the team, as well as any blockers
- At the end of the sprint
    - There's a sprint review – demo working features to product owner
    - There's a retrospective – talk about what worked, what didn't, what needs to change

Software Quality Attributes

Runtime

- See as a user of the system

Performance

- Focuses on how a system running the system performs under a particular load
- Load testing - checks the application's ability to perform under anticipated user loads.
    - The objective is to identify performance bottlenecks before the software application goes live

- Stress testing - involves testing an application under extreme workloads to see how it handles high traffic or data processing.
    - The objective is to identify the breaking point of an application
- Endurance testing - is done to make sure the software can handle the expected load over a long period of time.
- Spike testing - tests the software's reaction to sudden large spikes in the load generated by users
- Volume testing - Under Volume Testing large no. of. Data is populated in a database and the overall software system's behavior is monitored.
    - The objective is to check software application's performance under varying database volumes
- Scalability testing - The objective of scalability testing is to determine the software application's effectiveness in "scaling up" to support an increase in user load. It helps plan capacity addition to your software system
- Smoke Testing is a software testing process that determines whether the deployed software build is stable or not. Smoke testing is a confirmation for QA team to proceed with further software testing.
    - It consists of a minimal set of tests run on each build to test software functionalities. Smoke testing is also known as "Build Verification Testing" or “Confidence Testing.”

Security

- Vulnerability Scanning: This is done through automated software to scan a system against known vulnerability signatures.
- Security Scanning: It involves identifying network and system weaknesses, and later provides solutions for reducing these risks.
    - This scanning can be performed for both Manual and Automated scanning
- Penetration testing: This kind of testing simulates an attack from a malicious hacker.
    - This testing involves analysis of a particular system to check for potential vulnerabilities to an external hacking attempt
- Risk Assessment: This testing involves analysis of security risks observed in the organization. Risks are classified as Low, Medium and High.
    - This testing recommends controls and measures to reduce the risk
- Security Auditing: This is an internal inspection of Applications and Operating systems for security flaws.
    - An audit can also be done via line by line inspection of code

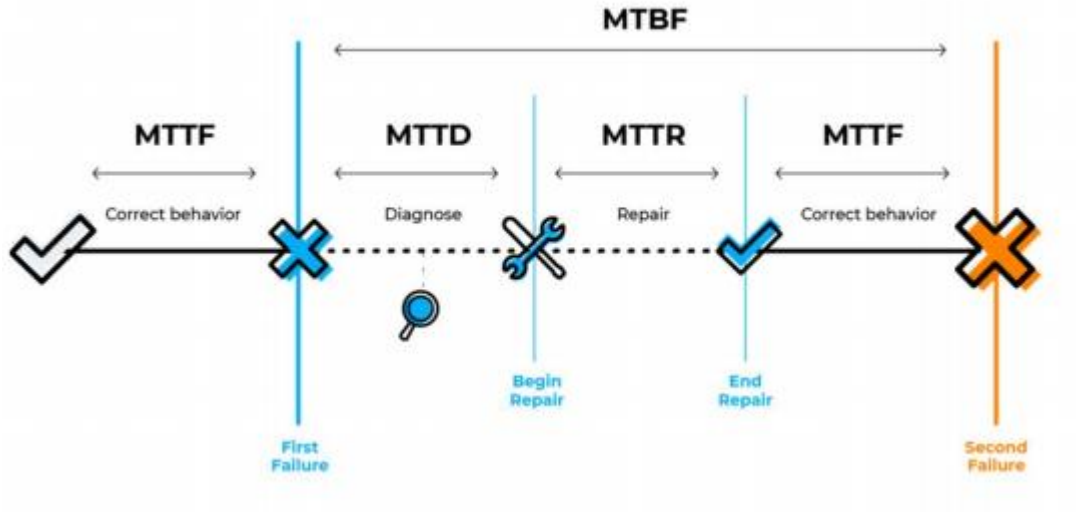
- Ethical hacking: It's hacking an organization's own software systems.
    - Unlike malicious hackers, who steal for their own gains, the intent is to expose security flaws in the system
- Posture Assessment: This combines Security scanning, Ethical Hacking and Risk Assessments to show an overall security posture of an organization

Availability

Rule of nines	
99%	*3 days, 15 hours and 40 minutes downtime per year*
99.9%	*8 hours, 46 minutes downtime per year*
99.99%	*52 minutes, 36 seconds downtime per year*
99.999%	*5 minutes, 15 seconds or less of downtime in a year*
99.9999%	Moving towards the dream

- The probability that the system is operating properly when it is requested for use.
    - In other words, availability is the probability that a system is not failed or undergoing a repair action when it needs to be used
- Dream is 5 9's

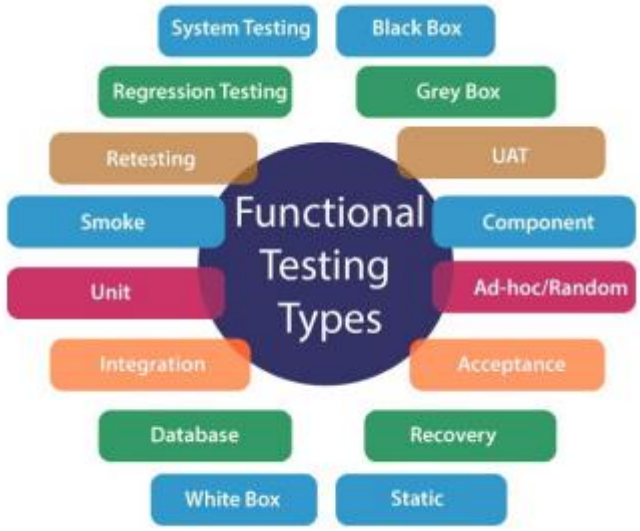
Measuring Up-time



- MTBF (mean time between failures): the average time between repairable failures of a technology product.
    - The metric is used to track both the availability and reliability of a product.
    - The higher the time between failure, the more reliable the system.
- MTTR (mean time to repair): the average time it takes to repair a system (usually technical or mechanical).
    - It includes both the repair time and any testing time.
    - The clock doesn't stop on this metric until the system is fully functional again
- MTTD (mean time to diagnose): once the system is down, how long it takes to diagnose what is wrong

- MTTR (mean time to recovery or mean time to restore) is the average time it takes to recover from a product or system failure.
    - This includes the full time of the outage—from the time the system or product fails to the time that it becomes fully operational again.
- MTTF (mean time to failure) is the average time between non-repairable failures of a technology product
    - For example, if Brand X's car engines average 500,000 hours before they fail completely and have to be replaced, 500,000 would be the engines' MTTF

Functionality



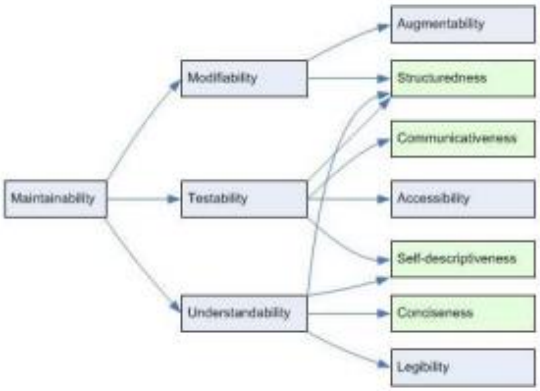
- Good quality sw will work as it is intended

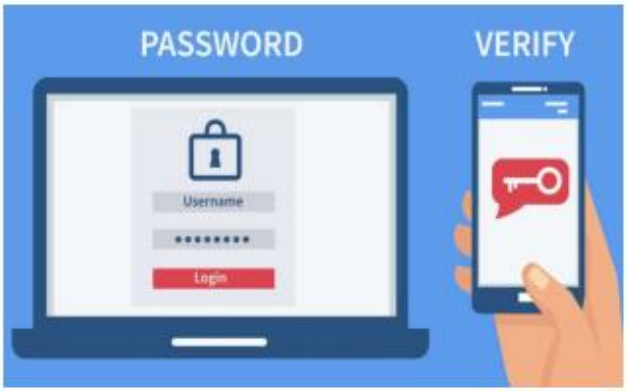
Usability

- Want to have good apps with good ux, that it fits its user-base

Non-Runtime

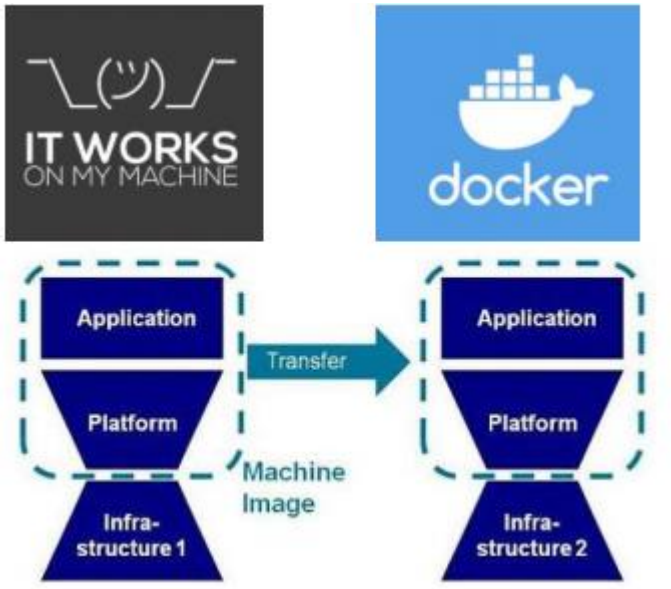
Modifiability/Maintainability





- A quality attribute of the software architecture that relates to “the cost of change and refers to the ease with which a software system can accommodate changes”
- E.g. good sw is such that can easily switch to two-factor authentication without having to refactor the whole code case

Portability

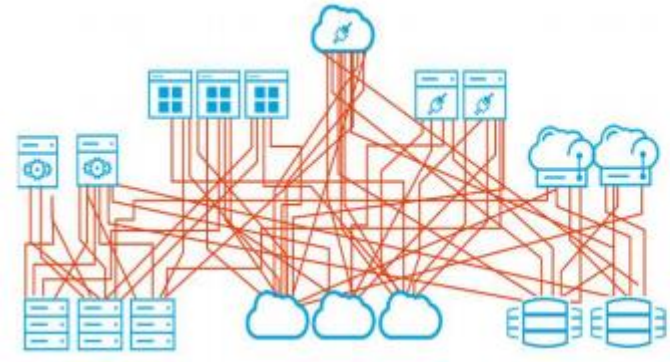


- The possibility to use the same software in different environments.
    - It applies to the software that is available for two or more different platforms or can be recompiled for that system

Reusability

- Modularity: Dividing code up into re-usable sections
- High Cohesion: Your modules should be functionally related to perform a task
    - Have sections of code that each do one job – when put together, enables the functioning of an entire system
- Low Coupling: Low level of inter dependence between modules
    - So, if have to remove one module/class, won't break entire app

Interoperability/Integrability



- Interoperability: the real-time data exchange between different systems that speak directly to one another in the same language
- Integration: the process of combining multiple applications to function together as one uninterrupted system, often involving the use of middleware

Testability

- Software needs to be written in such a way that it is easy to run any type test (unit, integration UAT) against the software
- Use principles of test-driven development to ensure that have highly testable sw

Teamwork for Software Quality

Code Reviews

Advantages

- 1. Consistent design and implementation
    - Code peer review can enforce a consistent coding style throughout a project, thereby making source code readable by anyone who might be introduced to the project at any given time during development
- 2. Minimizing your mistakes and their impact
    - This might seem like the most obvious advantage to the code peer review process, but it's also one of the most important.
    - When you're working with the real pressures of time and budget, it's easy to skip this step.
    - Sure, you're confident in your work, but even the best coders can go crossed-eyed from looking at their own work too long
- 3. Ensuring project quality and meeting requirements
    - The scope of any given software project and its requirements might run through the hands of several developers.
    - The code review process can serve as a check and balance against different interpretations of that scope and requirements compared to the code that ends up being delivered.
    - The second set of eyes can ensure you don't fall into the "pit" you created based on your own understanding
- 4. Improving code performance
    - Due to the lack of experience, some younger developers might be unaware of optimization techniques that could be applied on their code
    - The code review process provides an opportunity for these developers to acquire skills and boost the performance of their code
- 5. Sharing new techniques

- During a code review, developers can also share new technologies and techniques with each other
- Sharing techniques transcends seniority, where every developer is equally able to share, cooperate and improve their skills

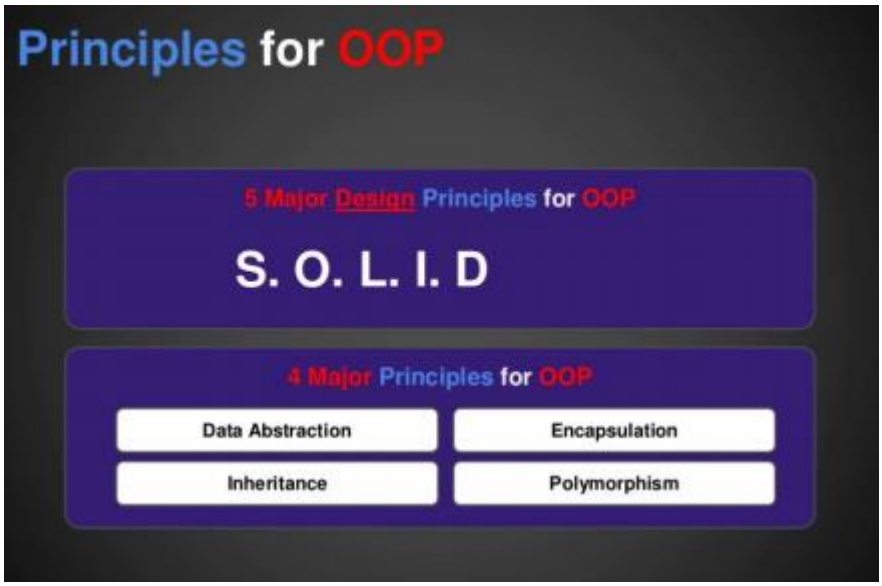
Software Quality Metrics

- Number of lines, files, etc.
  - How do your file sizes affect your software?
  - What is the function of the code lines?
  - Are your numbers maintainable?
- Field bugs
  - What are the problems in the already running software?
  - How many bugs did you find in production?
  - Is your software reliable?
  - How many times did it fail over a set period of time?
- Code churn
  - Why is one part of the code churning more than others?
  - Why is it error-prone?
  - Is anything in the completion rate standing out? Is the code usable once it has churned?
- Static analysis findings
  - What is consistent about the software?
  - How long does it take to fix code?
  - Is the software secure in its current standing vs. what has changed?
- Bug arrival rate
- How are you finding bugs?
  - When did a bug show up?
  - Why did you think the software was ready?
  - What are the rates at which bugs are coming in?
  - How many software releases happened during a period?

- Performance
    - Does the software code last during updates?
    - Is it performing as it should?
    - Why isn't it performing in load, stress, or response testing?
    - Do users enjoy it?
- Test failures
    - Automated and manually, what tests are failing?
    - Was a test working and now it is failing?
    - What is your failure balance?
    - How can testability be improved with technologies

SOLID Design Principles OOP

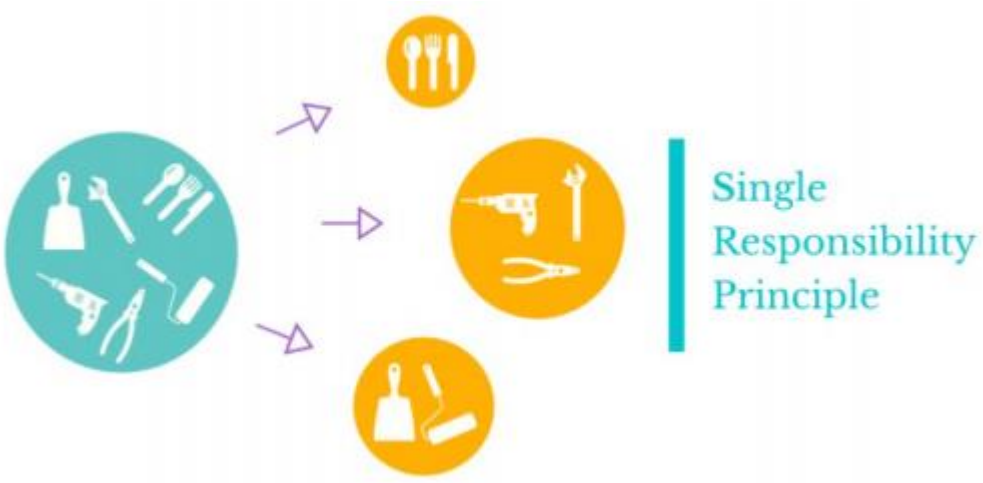
https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/



Single Responsibilities

- The concept is of a Class doing one specific thing (responsibility) and not trying to do more than it should
    - aka High Cohesion
- Only one potential change (database logic, logging logic, and so on) in the software's specification should be able to affect the specification of the class
    - Eg. If a class is a data container, like a Book class or a Student class, and it has some fields regarding that entity, it should change only when we change the data model
- Classes don't often start with low cohesion, but typically after several releases and different developers adding onto them, the class may be become a monster/god class
    - Class should be refactored
- The single responsibility principles is a programming principles that states that every module, class or function should have responsibility over a single part of the functionality provided by the sw, and that responsibility should be entirely encapsulated by the class
- It is necessary because:
    - Many many different teams can work on the same project and edit the same class for different reasons, this could lead to incompatible modules
    - It makes version control easier

S.O.L.I.D



Example

```
class Person {
    public name : string;
    public surname : string;
    public email : string;
    constructor(name : string, surname : string, email : string){
        this.surname = surname;
        this.name = name;
        if(this.validateEmail(email)) {
          this.email = email;
        }
        else {
            throw new Error("Invalid email!");
        }
    }
    validateEmail(email : string) {
        var re = /^([\w-]+(?:\.[\w-]+)*)@((?:[\w-]+\.)*\w[\w-]{0,66}
        return re.test(email);
    }
    greet() {
        alert("Hi!");
    }
}
```

Open Closed

- Open/closed principle states that “sw entities (classes, modules, functions, etc) should be open for extension but closed for modification”
    - That is, such an entity can allow its behaviour to be extended without modifying its src code
- Basically - write your code so that you will be able to add new functionality without changing the existing code
- Most “important” principle of OO-design

Liskov's substitution

- E.g. A penguin "is a" bird – so Penguin class should inherit from the Bird class
- The "is a" technique of determining the inheritance relationships is simple and useful – but occasionally results in bad use of inheritance
- Liskov Substitution Principle is a way of ensuring that inheritance is used correctly

Example

- Bird class has setAltitude method – but not all birds fly
- End up with empty method signatures for all classes that inherit from the parent Bird class
    - Leads to messy code
- Try to avoid creating this situation
- If want to make a parent class and put methods in that class – need to be sure that *every* class that *ever* in the app's lifetime will use every single method in that class
    - Otherwise, don't put the method in
    - Parent abstract class should be very lean – done to prevent forcing an inheriting class to implement something it can't use

```
class Bird {
public:
    virtual void setLocation(double longitude, double latitude) = 0;
    virtual void setAltitude(double altitude) = 0;
    virtual void draw() = 0;
};
```

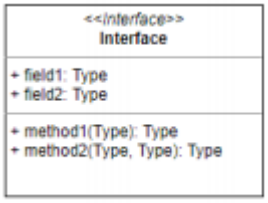
```
void Penguin::setAltitude(double altitude)
{
    //altitude can't be set because penguins can't fly
    //this function does nothing
}
```

```cpp
//Solution 3: Proper inheritance
class Bird {
public:
    virtual void draw() = 0;
    virtual void setLocation(double longitude, double latitude) = 0;
};

class FlightfulBird : public Bird {
public:
    virtual void setAltitude(double altitude) = 0;
};
```

Interface Segregation

Interfaces and their Implementation



- In general, an interface is a device or a system that unrelated entities use to interact.
- An Interface is a type, just as a class is a type.
    - Like a class, an interface defines methods.
    - Unlike a class, an interface never implements methods; instead, classes that implement the interface implement the methods defined by the interface.
    - A class can implement multiple interfaces.
- When a class implements an interface, the class agrees to implement all the methods

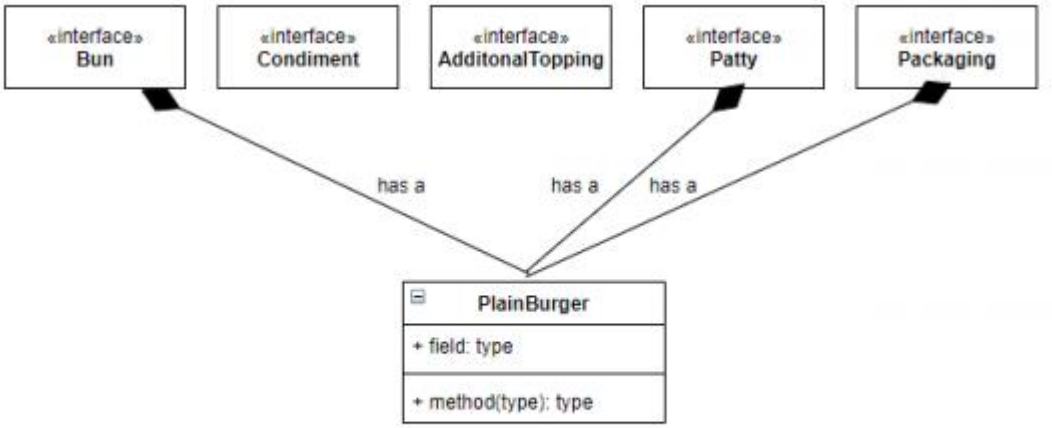
Interface Use

- Capturing similarities among unrelated classes without artificially forcing a class relationship
- Declaring methods that one or more classes are expected to implement
- Revealing an object's programming interface without revealing its class

- Modelling multiple inheritance, a feature that some object-oriented languages support that allows a class to have more than one superclass
- Enabling polymorphic behavior
- NB Java does not support multiple inheritance – things get too complicated too quickly

Example – Inheritance or Composition

- Look at “is it” vs “has a”



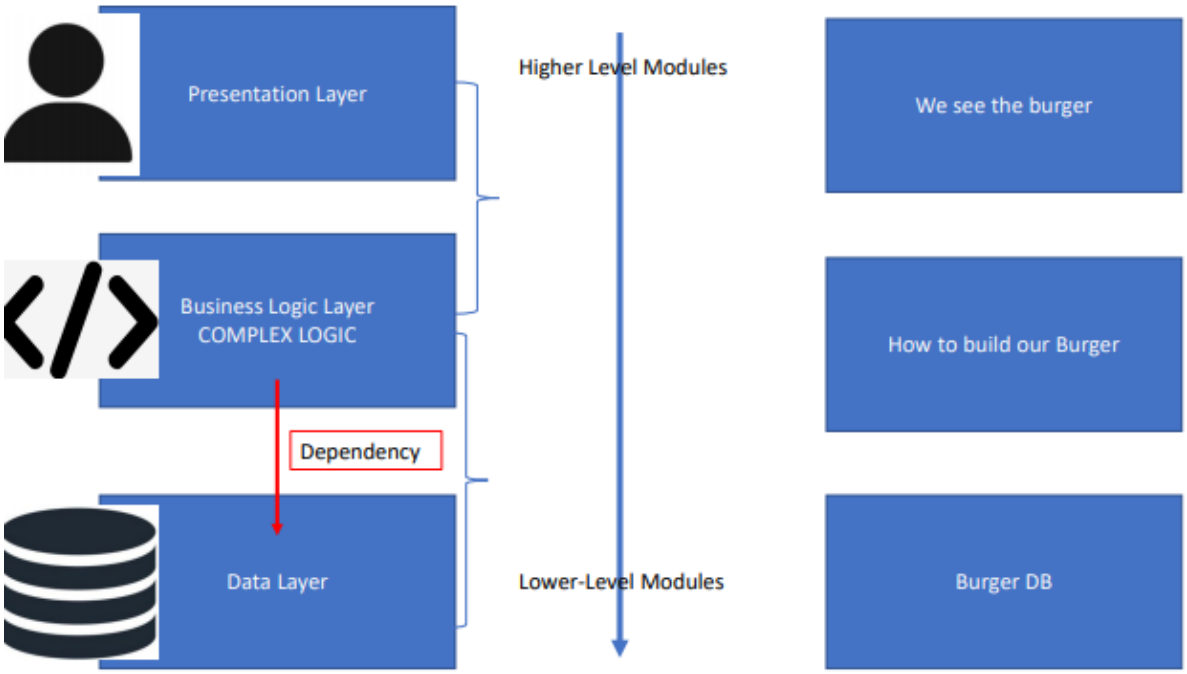
- Above shows the single responsibility principle
  - Have an interface for buns – only deals with buns, etc
- Use these interfaces to compose a burger – composition over inheritance
  - Designing for capabilities rather than identity

Interface Segregation

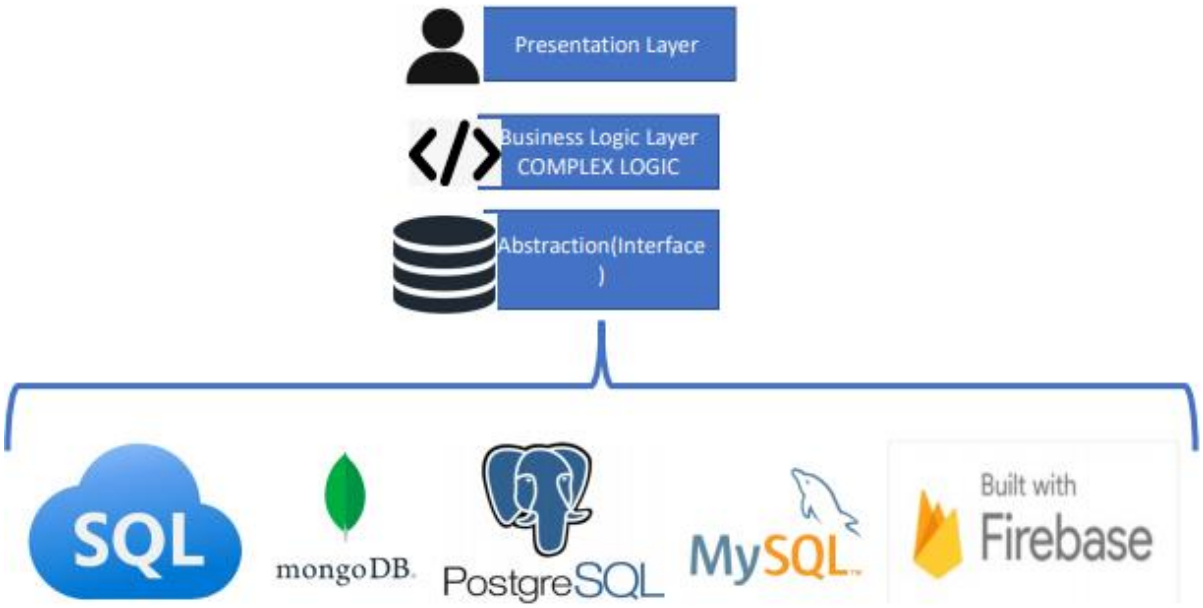
- Interface Segregation Principle (ISP) states that clients shouldn't be forced to depend on methods they don't use
- Interfaces should belong to clients, not to libs/hierarchies
- App devs should favour thin, focused interfaces to "fat" interfaces that offer more functionality than a particular class/method needs
- Basically – if this interface has a method that not all classes that implements this interface is going to use, that method belongs in a new interface

Dependency Inversion

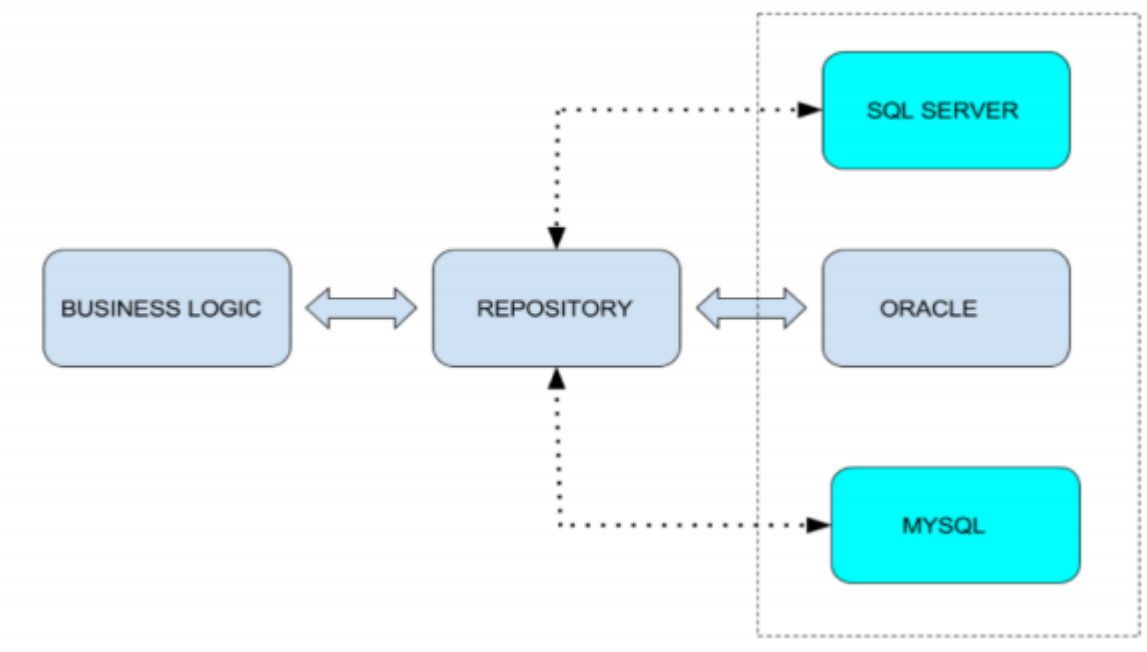
- Based on the Open/Closed Principle and the Liskov Substitution Principle
- High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level module



- Diagram above shows how we model sw
- Higher level modules at the top, lwr towards the bottom
- Data Layer – persist layer in dtb
    - Move to business logic layer – then pull data out of dtb and manipulate it in order to present to user in presentation layer
- When sw built badly, a dependency between the data and business layer is created



- Ideal is this above diagram
- At the bottom – have diff db sw
    - If one of these dtbs stops being developed/company moves to different service
    - Should write sw in such a way that these can be plugged in directly into application – achieved by using an interface (instead of directly wiring in the dtb into the app)
- Create an interface/abstraction that hides from the app what dtb is used – app then implements the interface
    - Thus, business and data layer won't be tightly coupled



- Result is above figure – have repo interface with a dtb
    - These dtbs are hidden from app – business logic class then talks to repo, rather than

SOLID Summary

- SRP – a class has only one job
    - Single Responsibility
- OCP – extend classes, don't change them
    - Open/Closed Principle
- LSP – use abstraction correctly
    - Liskov's Substitution Principle
- ISP – create "thin" specific interfaces
    - Interface Segregation
- DIP – Decouple lower logic from higher up, more complex logic
    - Dependency Inversion

Gang Of Four Design Patterns

- Design patterns represent the best practices used by experienced object-oriented software developers.
    - Design patterns are solutions to general problems that software developers faced during software development.
    - These solutions were obtained by trial and error by numerous software developers over quite a substantial period.

Patterns and Description

- Creational Patterns: provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using the “new” operator
- Structural Patterns: these design patterns concern class and object composition
    - Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
- Behavioural Patterns
    - These design patterns are specifically concerned with communication between objects

Creational Patterns

Pattern Name	Description
Singleton	The singleton pattern restricts the initialization of a class to ensure that only one instance of the class can be created.
Factory	The factory pattern takes out the responsibility of instantiating a object from the class to a Factory class.
Abstract Factory	Allows us to create a Factory for factory classes.
Builder	Creating an object step by step and a method to finally get the object instance.
Prototype	Creating a new object instance from another similar instance and then modify according to our requirements.

- Focus on Singleton and Factory

Structural Patterns

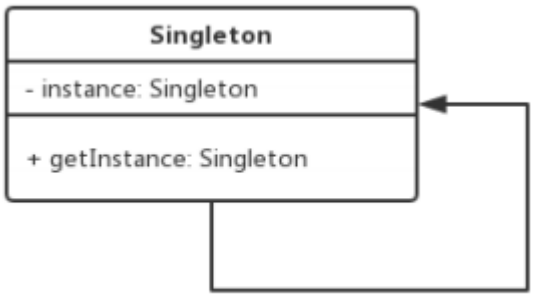
Pattern Name	Description
Adapter	Provides an interface between two unrelated entities so that they can work together.
Composite	Used when we have to implement a part-whole hierarchy. For example, a diagram made of other pieces such as circle, square, triangle, etc.
Proxy	Provide a surrogate or placeholder for another object to control access to it.
Flyweight	Caching and reusing object instances, used with immutable objects. For example, string pool.
Facade	Creating a wrapper interfaces on top of existing interfaces to help client applications.
Bridge	The bridge design pattern is used to decouple the interfaces from implementation and hiding the implementation details from the client program.
Decorator	The decorator design pattern is used to modify the functionality of an object at runtime.

- Focus on Adaptor and Composite Patterns

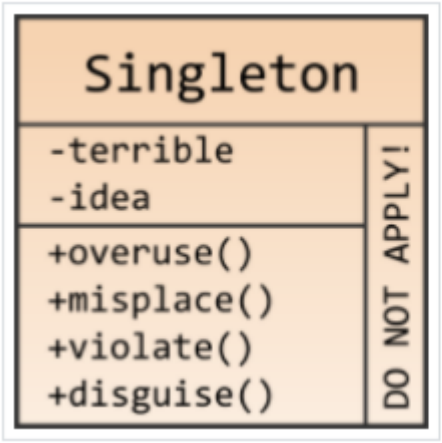
Behavioural Patterns

Pattern Name	Description
Template Method	used to create a template method stub and defer some of the steps of implementation to the subclasses.
Mediator	used to provide a centralized communication medium between different objects in a system.
Chain of Responsibility	used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them.
Observer	useful when you are interested in the state of an object and want to get notified whenever there is any change.
Strategy	Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.
Command	Command Pattern is used to implement lose coupling in a request-response model.
State	State design pattern is used when an Object change it's behavior based on it's internal state.
Visitor	Visitor pattern is used when we have to perform an operation on a group of similar kind of Objects.
Interpreter	defines a grammatical representation for a language and provides an interpreter to deal with this grammar.
Iterator	used to provide a standard way to traverse through a group of Objects.
Memento	The memento design pattern is used when we want to save the state of an object so that we can restore later on.

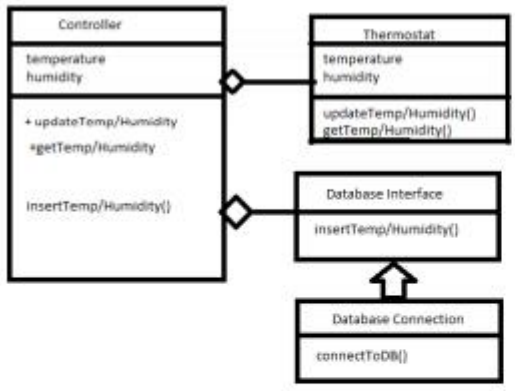
Singleton Design Pattern

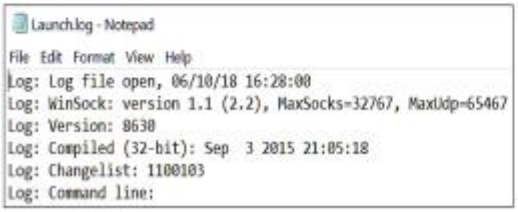


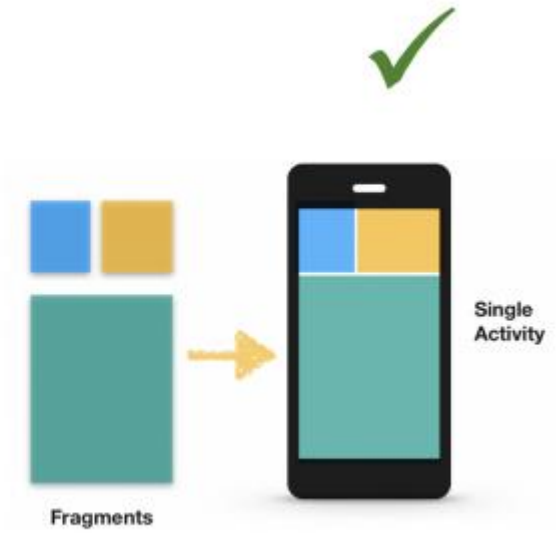
- Restricts initialisation of a class to ensure that only one instance of the class can be created
    - Not always good practice



- Can be used in the wrong place and then violate rules of SOLID by having tight coupling in your application
- When then is singleton used?



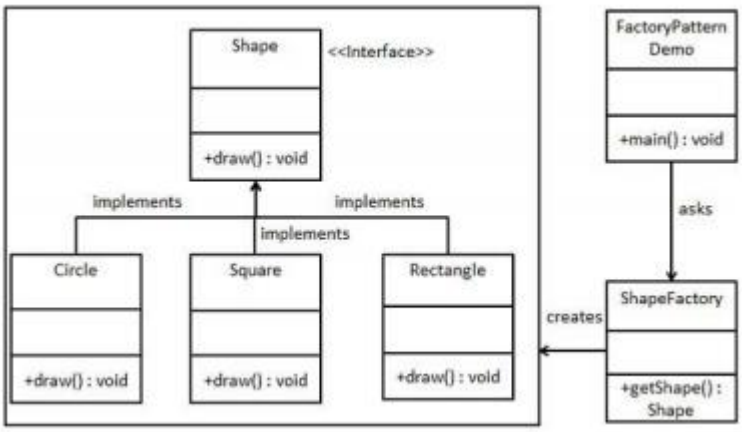




- E.g. dtb connection – only want one connection to the dtb; also when writing log files
  - Don't want multiple instances of a class writing entries to your log file – just want one instance of a class to write logs for your app
- Also used when coding fragments in Android Development

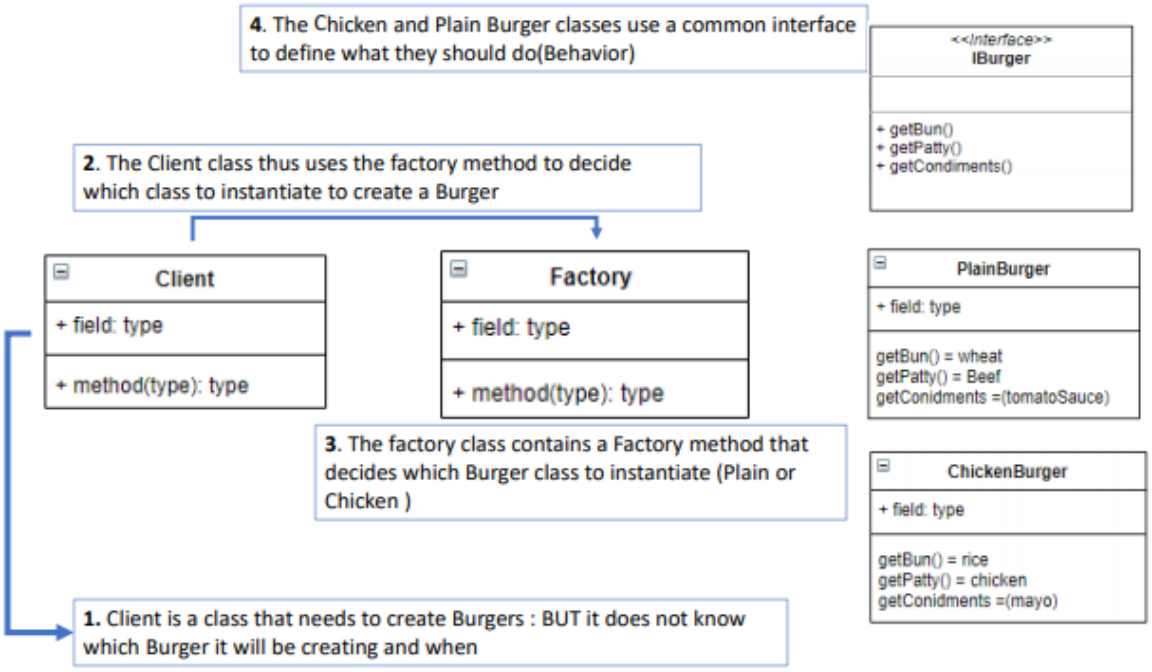
- Derived Exception = Sealed keyword
  - https://www.oracle.com/technical-resources/articles/java/singleton.html

Factory Design Pattern

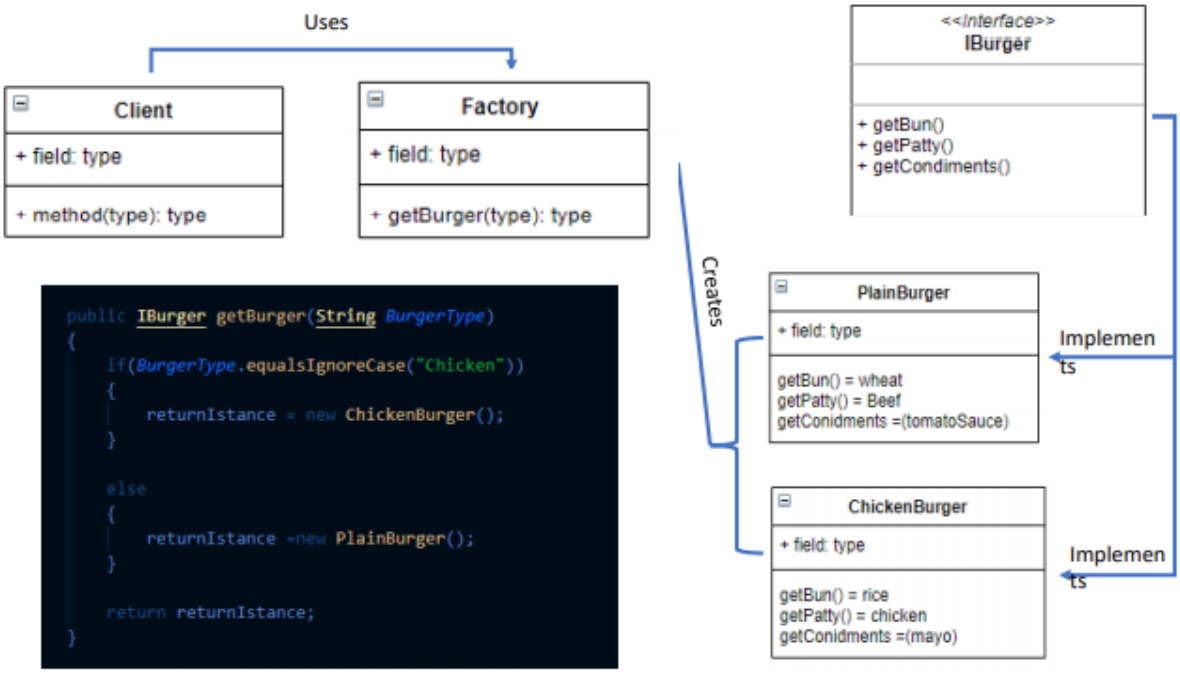


- The Factory pattern is one of the most used design patterns in Java
  - This type of design pattern falls under creational patterns as this pattern provides one of the best ways to create an object
- In the Factory pattern, we create an object without exposing the creation logic to the client and refer to the newly created object using a common interface

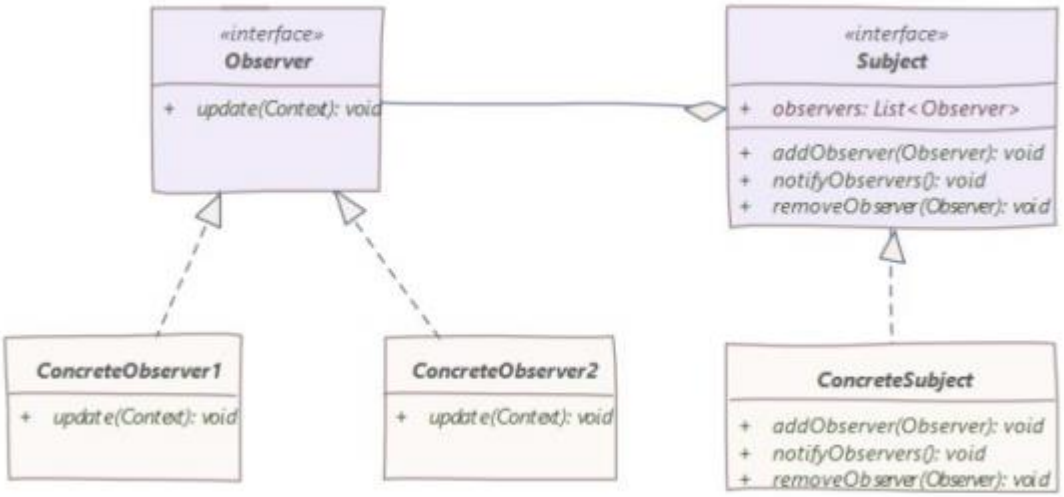
Example



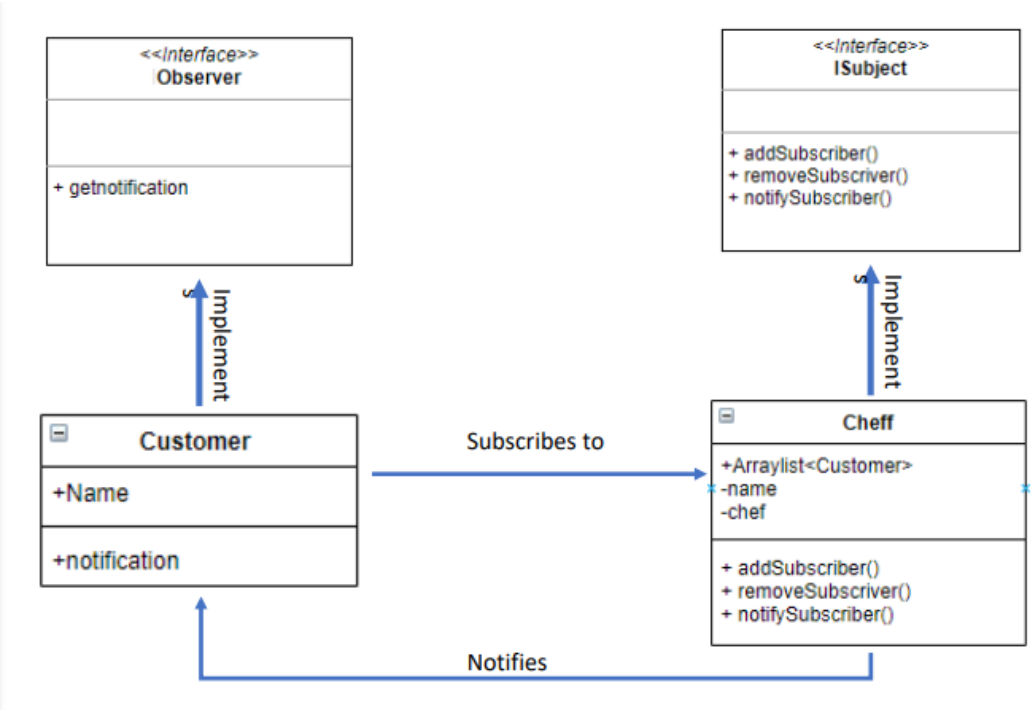
- Client is a class which needs to instantiate classes to create burgers – but doesn't know which burger it's going to create or when
    - Need to be able to create any burger, any time
- So, client uses which class to instantiate to create chicken/plain burger – client defers instantiation to the factory.
    - Factory contains factory method to decide which burger to create
- Chicken and burger classes use interface to define their behaviour
    - i.e. they implement interface and inherit all methods



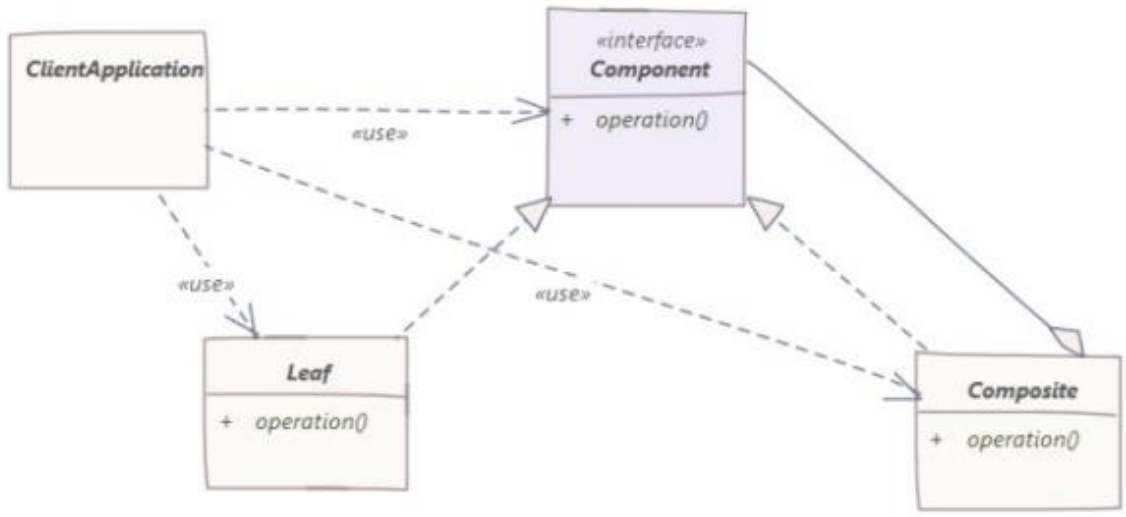
Observer Pattern



- The Observer Design Pattern maintains one-to-many dependency between Subject (Observable) and its dependents (Observer) in such a way that whenever state of Subject changes, its dependents get notified
- The Observer Design Pattern is a design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods
- The Observer Design Pattern is used when we like to have notification upon changes in the objects state
- The Observer Design Pattern is mainly used to implement distributed event handling, in "event driven" system
- In such systems, the subject is usually called a "source of events", while the observers are called "sink of events"

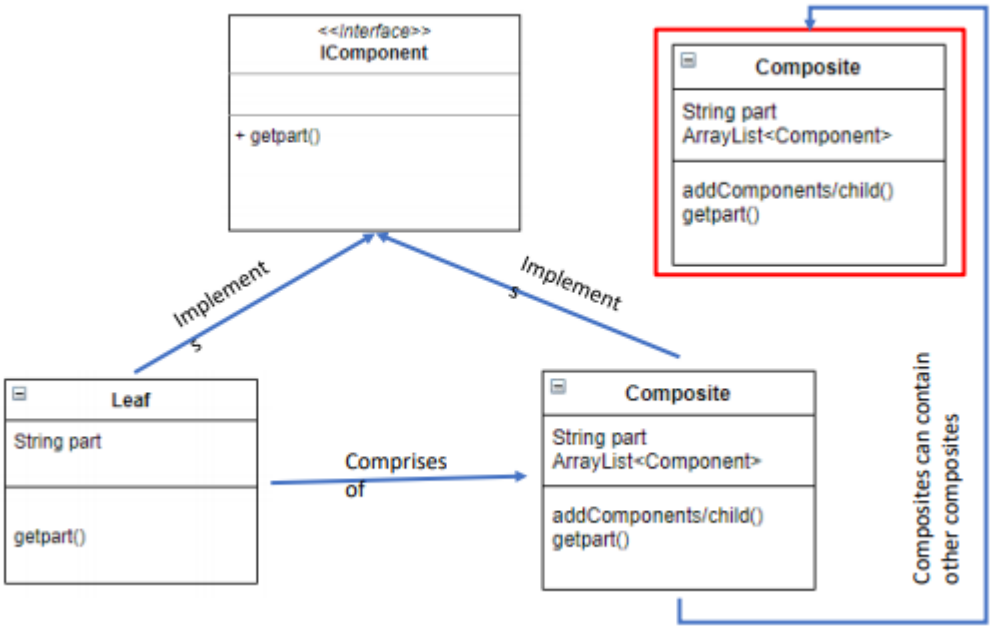


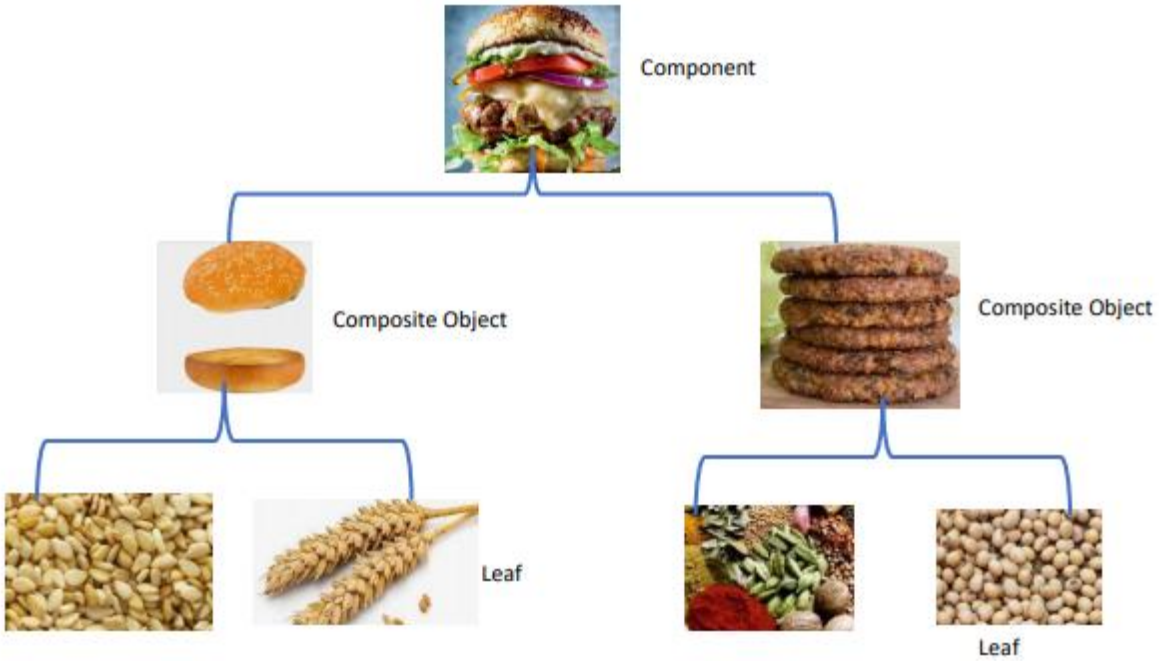
Composite Pattern



- The Composite Design patterns describe groups of objects that can be treated in the same way as a single instance of the same object type
- The Composite pattern allows us to "compose" objects into tree structures to represent part-whole hierarchies
- In addition, the Composite patterns also allow our clients to treat individual objects and compositions in the same way
- The Composite patterns allow us to have a tree structure for each node that performs a task
- In object-oriented programming, a Composite is an object designed as a composition of one-or-more similar objects, all exhibiting similar functionality.
    - This is known as a “has-a” relationship between objects

- Diagram above:
  - A leaf has no children – a composite is an obj that can contain leaves
    - One composite can contain many leaves or many composites
  - Component is the final product

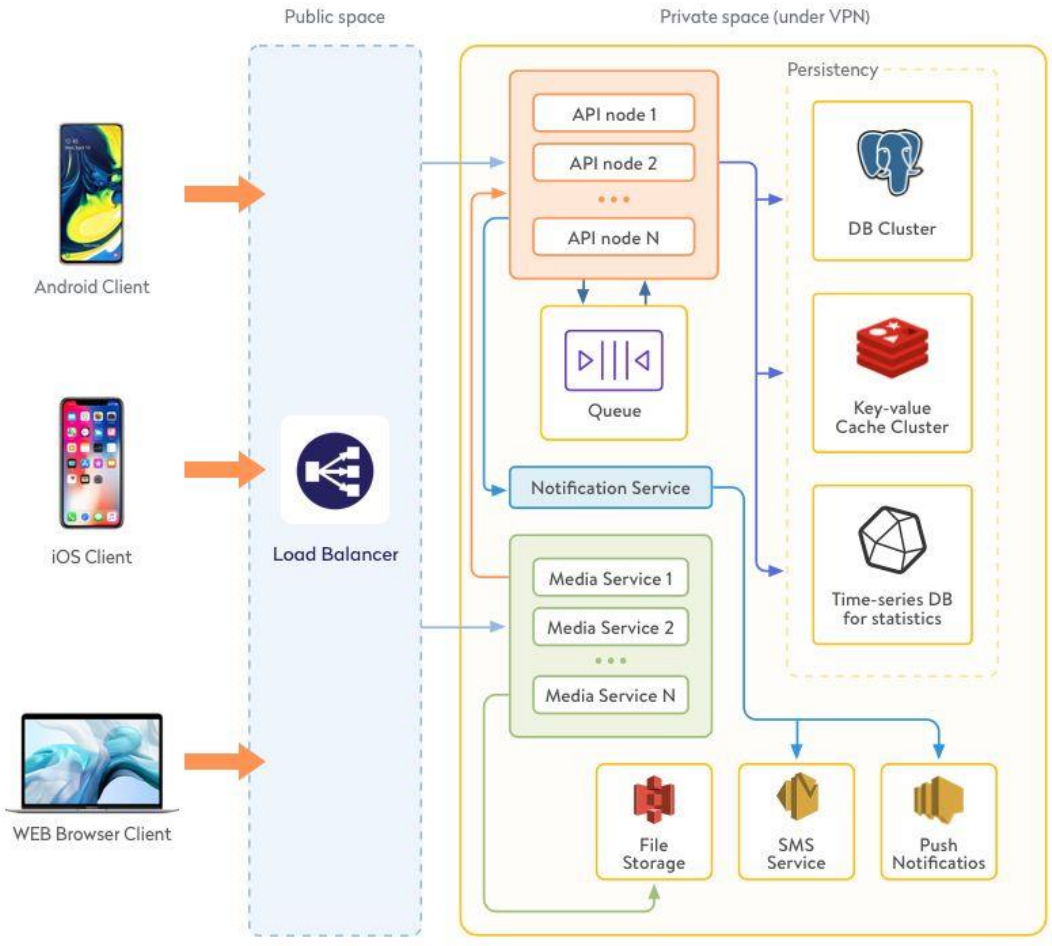




Architecture

- Think Big, Act Small, Fail Fast, Learn Rapidly
    - Slogan from Lean Software Development (another Agile method)
        - Has the principle of “See the whole” (amongst others)
        - =/= do the whole design early!
            - Rather do just enough design at a high lvl to have a good enough idea of how the pieces will interlock together
- Seeing the whole when building a single system – this is architectural thinking
    - It is the translation from the problem domain to the solution concepts
        - Concepts at a high lvl
        - Technology with purpose
- Another idea is development as a cycle of experiments with a test at the end of each cycle.

Architecture



[App architecture]

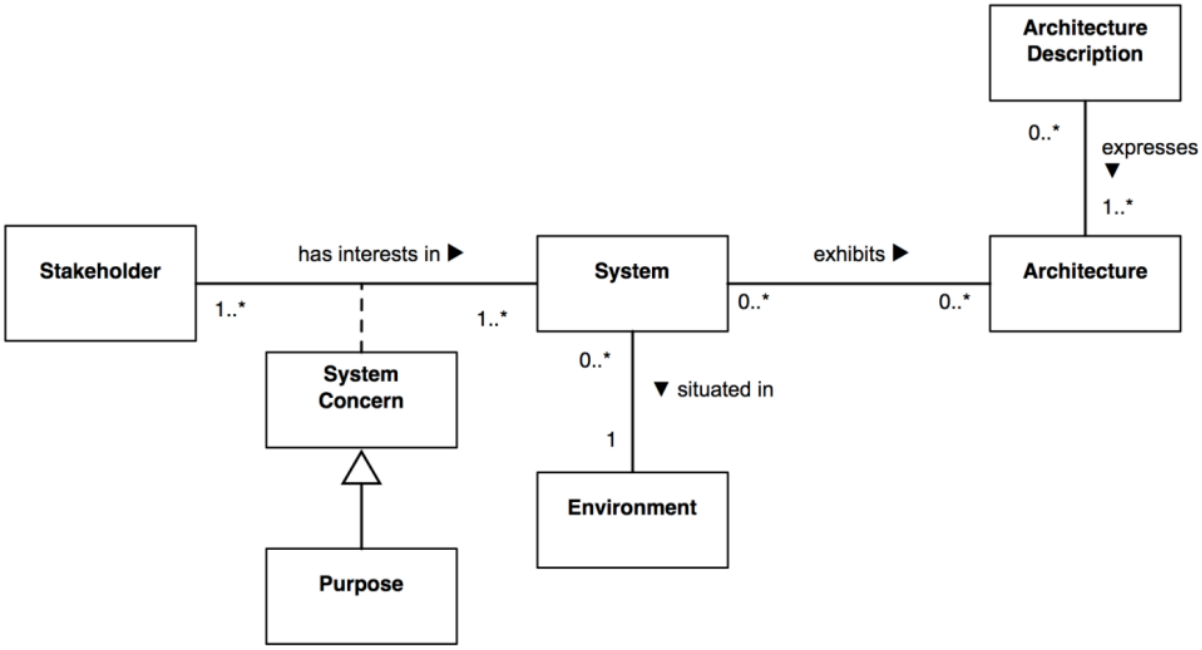
- SW architecture: the set of principal design decisions about the system
    - The heart software system
- Well-engineered sw ==> good software architecture
    - A good set of design decisions
    - NB: this is very different from the other meaning of Architecture in CS (hardware and the associated abstractions)

Definition: Architecture

- Basic idea – it is about the BIG picture; the large scale:
    - Motivations
    - Constraints
    - Organisation
    - Patterns
    - Responsibilities
    - Connections of a system (or a system of systems)
- Architecture is about how all of the code components fit together, how pieces of code talk to one another, and how the overall system functions at a high level
    - Many different design patterns interleaving to support an overall system – and the pattern choices we make will be governed by lots of factors:
        - Motivations of the various stakeholders, constraints on the system and the users, organization of the software as well as physical organizations and dependencies on other products, design patterns, responsibilities, and the connections of the system.
- "An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behaviour as specified in the collaborations among those elements, the composition of these structural and behavioural elements into progressively larger subsystems, and the architectural style that guides this organization—these elements and their interfaces, their collaborations, and their composition."
    - Booch, Rumbaugh, and Jacobson, The UML User Guide, 1999
- The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.
    - Architecture is concerned with the public side of interfaces; private details of elements—details having to do solely with internal implementation—are not architectural
    - By "externally visible" properties, we are referring to those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on.
    - The intent of this definition is that a software architecture must abstract away some information from the system (otherwise there is no point looking at the architecture, we are simply viewing the entire system) and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction.
    - Bass, Clements, and Kazman. Software Architecture in Practice, 2003
- Think of a component – when looking at it from a high lvl architectural perspective, it is a black-box

- It doesn't matter how anything is implemented inside the box – so long as it's clear what the inputs and outputs are and how its communicating with the rest of the system

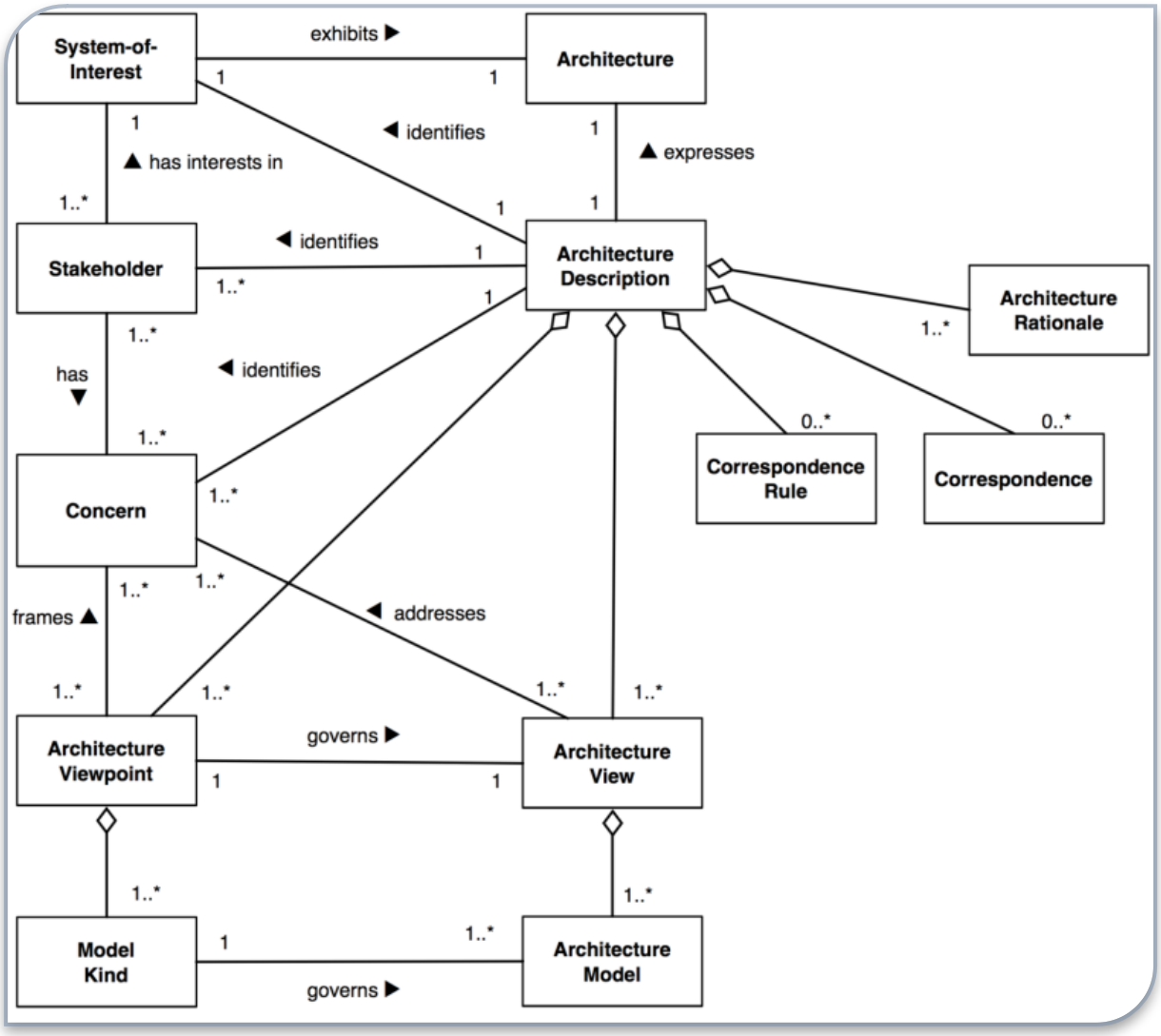
ISO/IEC/IEEE 42010:2011: Architecture Description



- International Standard: ISO/IEC/IEEE 42010:2011
  - www.iso-architecture.org/ieee-1471/index.html
  - Recommended practice for architectural description of software-intensive systems
    - Recommendation that explains what are the different things that need to be aware of in a system and what do we need to document so that it's clear to anyone else that looks at the documentation, that you agree on the same things
- Diagram:
  - Systems exist.
    - A System is situated in its Environment.
    - That environment could include other Systems
  - Stakeholders have interests in a System; those interests are called Concerns.
    - A system's Purpose is one very common Concern
  - Systems have Architectures.
    - An Architecture Description is used to express an Architecture of a System. System: The Standard takes no position on the question, What is a system?
  - In the Standard, the term system is used as a placeholder – e.g., it could refer to an enterprise, a system of systems, a product line, a service, a subsystem, or software.
    - Systems can be man-made or natural.
    - Nothing in the Standard depends upon a particular definition of system.

        - Users of the Standard are free to employ whatever system theory they choose.
        - The premise of the Standard is, For a system of interest to you, the Standard provides guidance for documenting an architecture for that system.
    - Environment: Every System inhabits its Environment.
        - A System acts upon that Environment and vice versa.
        - A system's Environment determines the range of influences upon the system.
        - In the Standard, Environment is intended in the widest possible sense to include developmental, operational, technical, political, regulatory, and all other influences which can affect the architecture.
        - These influences are categorized as Concerns. Architecture: Systems have architectures.
        - In the Standard, the architecture of a system is defined as: "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution".
        - The definition was chosen (1) to accommodate the broad range of things listed above under System: the architecture of X is what is fundamental to X (whether X is an enterprise, system, system of systems, or some other entity); and (2) to emphasize (via the phrase "concepts or properties") that a system can have an architecture even if that architecture is not written down.
    - Architecture Description: An Architecture Description (AD) is an artefact that expresses an Architecture.
        - Architects and other system stakeholders use Architecture Descriptions to understand, analyse and compare Architectures, and often as "blueprints" for planning and construction. ADs are the primary subject of ISO/IEC/IEEE 42010

Core of Architecture Description



- Contents of an AD:
  - The relations between content items when applying the Standard
  - Produce an AD to express an Architecture for some System of Interest.
- **Architecture Description:** An Architecture Description is a work product used to express the Architecture of some System Of Interest.
  - The Standard specifies requirements on ADs.
  - An AD describes one possible Architecture for a System Of Interest.
  - An AD may take the form of a document, a set of models, a model repository, or some other form (AD format is not defined by the Standard).
- **Stakeholder:** Stakeholders are individuals, groups or organizations holding Concerns for the System of Interest.
  - Examples of stakeholders: client, owner, user, consumer, supplier, designer, maintainer, auditor, CEO, certification authority, architect.
- **Concern:** A Concern is any interest in the system.
  - The term derives from the phrase “separation of concerns” as originally coined by Edsgar Dijkstra.

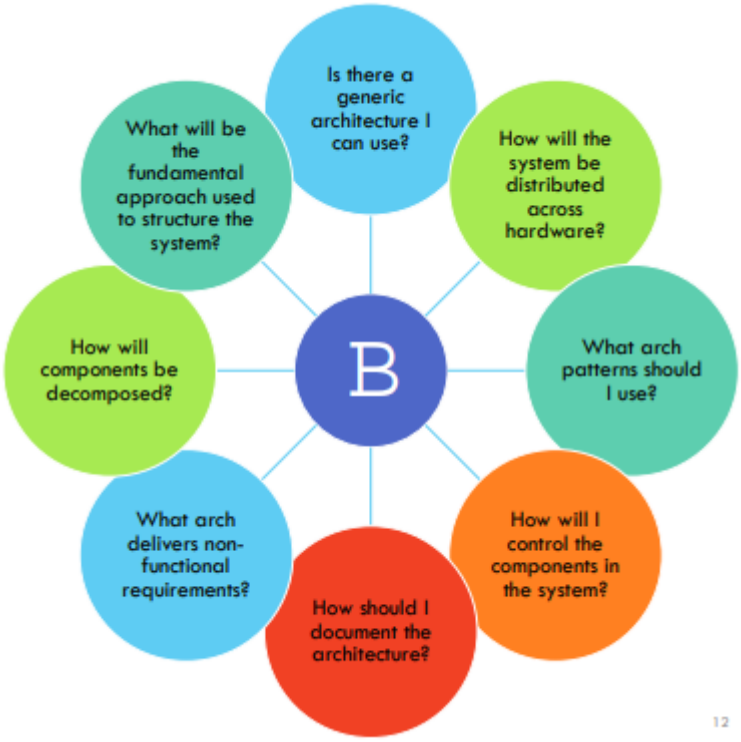
    - Examples of concerns: (system) purpose, functionality, structure, behaviour, cost, supportability, safety, interoperability.
- **Architecture Viewpoint:** An Architecture Viewpoint is a set of conventions for constructing, interpreting, using and analysing one type of Architecture View.
    - A viewpoint includes Model Kinds, viewpoint languages and notations, modelling methods and analytic techniques to frame a specific set of Concerns.
    - Examples of viewpoints: operational, systems, technical, logical, deployment, process, information.
- **Architecture View:** in an AD expresses the Architecture of the System of Interest from the perspective of one or more Stakeholders to address specific Concerns, using the conventions established by its viewpoint.
    - An Architecture View consists of one or more Architecture Models.
- **Architecture Model:** A view is comprised of Architecture Models. Each model is constructed in accordance with the conventions established by its Model Kind, typically defined as part of its governing viewpoint.
    - Models provide a means for sharing details between views and for the use of multiple notations within a view.
- **Model Kind:** A Model Kind defines the conventions for one type of Architecture Model.
- **AD Elements and Correspondences:** Architecture Descriptions are comprised of AD Elements.
    - Correspondences capture relationships between AD Elements.
    - Correspondences and Correspondence Rules are used to express and enforce architecture relations such as composition, refinement, consistency, traceability, dependency, constraint and obligation within or between ADs.

How ISO/IEC/IEEE 42010 defines Architecture

- http://www.iso-architecture.org/ieee-1471/defining-architecture.html
    - The disjunction, concepts or properties, was chosen to support two different philosophies without prejudice.
    - These philosophies are:
        - Architecture as Conception: an architecture is a concept of a system in one's mind
        - Architecture as Perception: an architecture is a perception of the properties of a system
    - Under either philosophy, an architecture is abstract — not an artefact.
        - The Standard uses another term, architecture description, to refer to artefacts used to express and document architectures.
    - The point of this documentation and description is to attain agreement between all of the different stakeholders as devs, clients and users about what is being developed
        - So that you can agree that everything is progressing in the right direction and that the right things have been prioritised i.t.o the requirements
    - Note that principles of its design and evolution may be formulated by an architect, “reverse engineered” from an existing system, or even discovered in nature — depending on the system.
        - While for man-made systems the architecture often reflects an intentional stance, this intention is not part of the definition.

- The fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution
- In the standard an architecture is abstract — not an artefact
    - Architecture description – artefacts used to express & document architectures
- What is fundamental to a system may take several forms:
    - Elements – the constituents that make up the system
    - Relationships – both internal and external to the system
    - Principles of its design and evolution
- Different architecture communities place varying emphases
    - SW architecture: focused on software components as elements and their interconnections as a key relationship
    - System architecture emphasizes sub-system structures and relationships such as allocation.
    - Enterprise architecture emphasizes principles.

Architectural Design Decisions

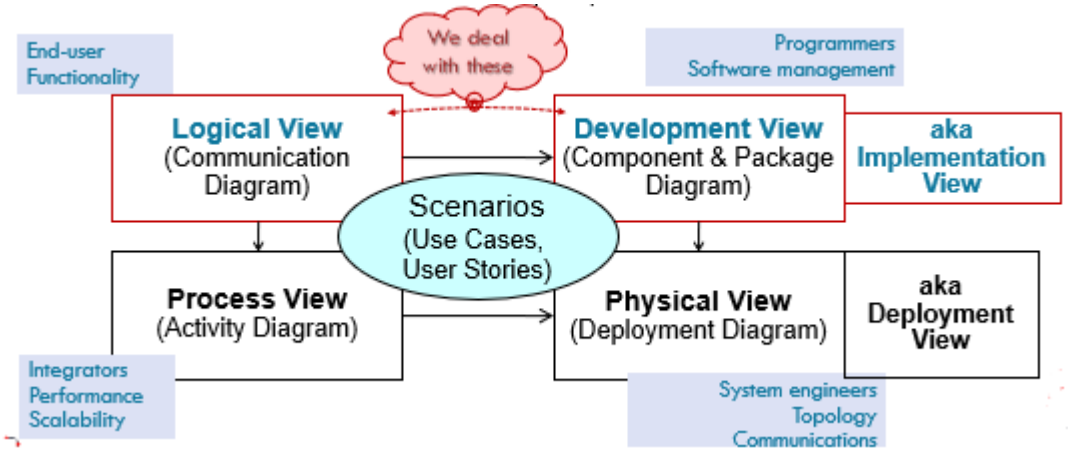


- Generic architecture – what is the one that can be applied to the system
    - Could be several architecture patterns that will be incorporated together to meet several different non-functional requirements
- How components are decomposed – could postpone this until later
    - Decompose component just to be able to do the next step
- System distribution across hw – important to know for client-server systems where you will need to scale (for load-balancing)
- …..etc

UML views

- How to use UML to represent architectures

Rational Unified Process "4+1" View of Architectures



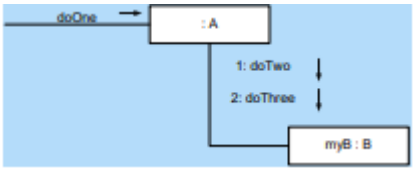
- A logical view, which shows the key abstractions in the system as objects or object classes.
- A process view, which shows how, at run-time, the system is composed of interacting processes.
- A development view, which shows how the software is decomposed for development.
- A physical view, which shows the system hardware and how software components are distributed across the processors in the system
    - Becomes important if have more than one server and they'll be interacting – i.e. which pieces are on which devices
- Related using use cases or scenarios (the +1) – used to provide an overall view of what's happening in the system
- **Conceptual View** is more of a domain level view, initial steps in architectural design.
    - This view is an abstract view of the system that can be the basis for decomposing high-level requirements into more detailed specifications, help engineers make decisions about components that can be reused, and represent a product line rather than a single system.
    - In practice, conceptual views are almost always developed during the design process and are used to support architectural decision making.
    - They are a way of communicating the essence of a system to different stakeholders.

Why so many views and diagrams?

- Seems like there's a lot of overhead from diagrams
- Because so many different stakeholders are interested in the overall design ≡ Architecture
- Viewpoints of the different stakeholders may lead to different views of the same system:
    - These views have to be communicated and represented and then integrated
    - Together they form the complete architectural description of the software system being designed
- Architectural representation has two objectives:
    - To be able to accommodate different views based on the requirements

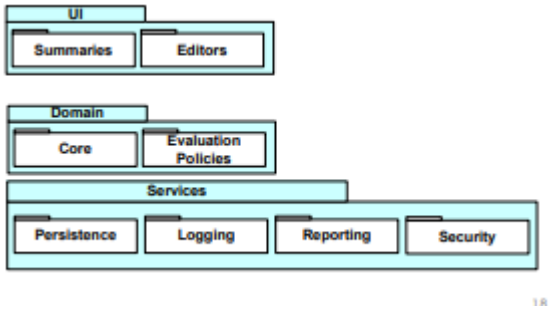
- Integrating of these different views to form the complete architectural representation

Logical View (Module View)



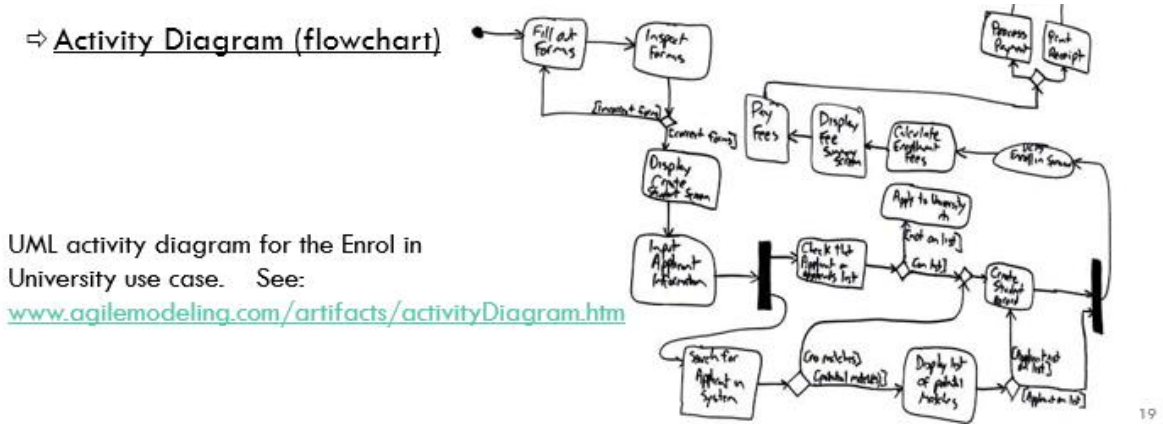
- The logical view is concerned about the output(s) of the system and how it will affect the end users
- The logical view splits the system into a set of abstractions, or modules
- This decomposition serves two purposes:
  - It enables functional analysis
  - It helps in identification of common mechanisms and design elements that are common across the system
- Communication diagrams
- What are the different modules and how are they going to communicate with one another?

Development View (Allocation View)



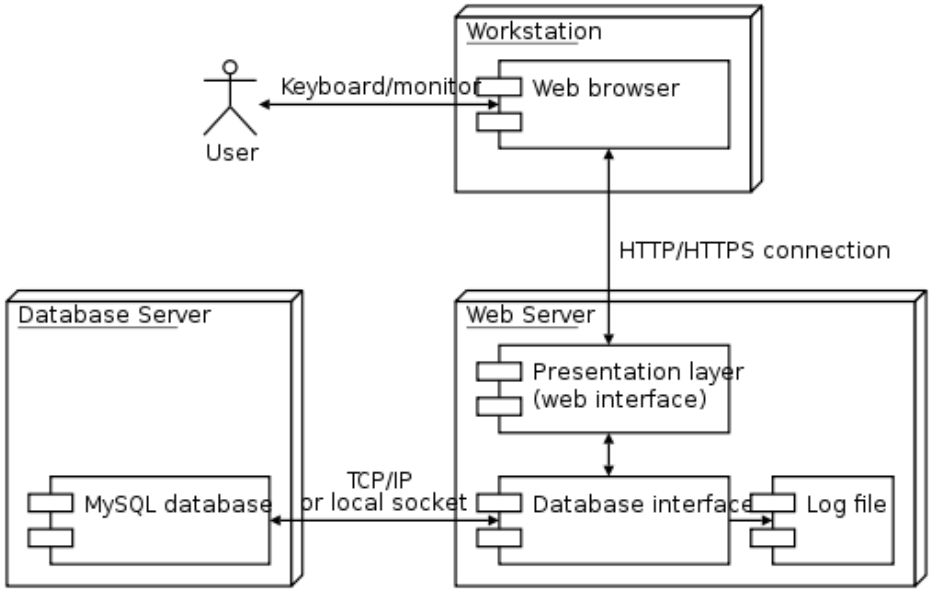
- This view describes the static organization of the software in its development environment
- It deals with modules, work allocation, costs and planning.
- It also involves monitoring of project progress, software reuse, and security
- Needed for purposes of project management and organising development of the sw, rather than organising the sw itself and how they're communicating with one another
- Component & Package Diagrams
  - See how all of the components will be organised into separate packages that will talk within one another

Process View (Component-and-Connector view)



- This view deals with concurrency and distribution, system integrity, and fault tolerance
- It explains which components interact, and how they do so
    - And the dynamic connections between different components at runtime
- Activity Diagram (flowchart)
- Emphasis now on runtime – how different things conflict when they're operating, and how to ensure they work well together without creating a state/condition that system can't recover from

Physical View (Deployment View)



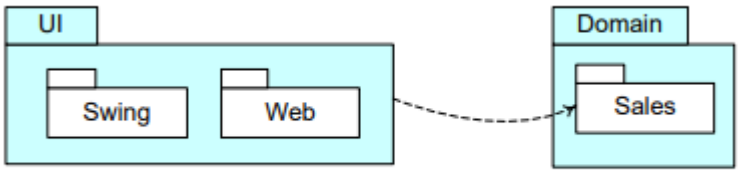
- This view describes how the software maps onto the hardware
- It shows networking and distribution
- It considers system requirements like reliability and performance
- Deals with the elements identified in the previous three views
- Deployment Diagram

Summary

- Need to be able to look at an architecture diagram and assess whether this is a physical view, logical view, process view, development view

    - And understand what is the purpose of the diagram – is it intended to help plan the logical organisation of the system, how development will be managed, hardware and testing environments, etc?
- Lots of different views together produce a whole to give a broader idea of what that architecture is

UML Packages



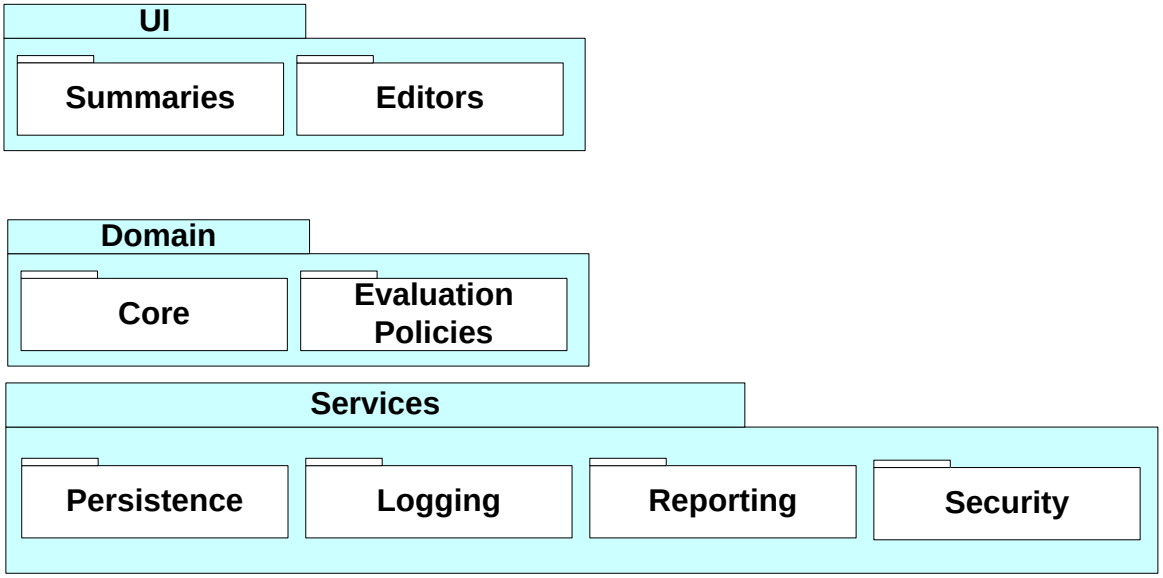
- UML package diagrams are often used to illustrate the logical architecture of a system-the layers, subsystems, packages (in the Java sense), etc
    - A layer can be modelled as a UML package; for example, the UI layer modelled as a package named UI
- A UML package diagram provides a way to group elements
    - A UML package can group anything: classes, other packages, use cases, and so on
    - Nesting packages is very common
    - A UML package is a more general concept than simply a Java package or .NET namespace, though a UML package can represent those and more
- The package name may be placed on the tab if the package shows inner members, or on the main folder, if not
- It is common to want to show dependency (a coupling) between packages so that developers can see the large-scale coupling in the system
    - The UML *dependency line* is used for this, a dashed arrowed line with the arrow pointing towards the depended-on package
- UML packages:
    - Packages group elements
        - For example, groups of classes in a single namespace
    - Drawn as a rectangle with a smaller tab at the upper left
        - If members are shown within the package, name the tab
    - Used to show the high-level organization of a project
    - A dashed arrow between packages indicates a dependency

Logical Architecture

- Shows large-scale organization of software classes, grouped by
    - Layers (coarse-grained)
    - Packages
    - Subsystems (finer-grained)
    - Together, provide cohesive responsibility for a major aspect of the system
- “Logical” => independent of actual deployment decisions
- Aiming for high cohesion
- Place elements together that:

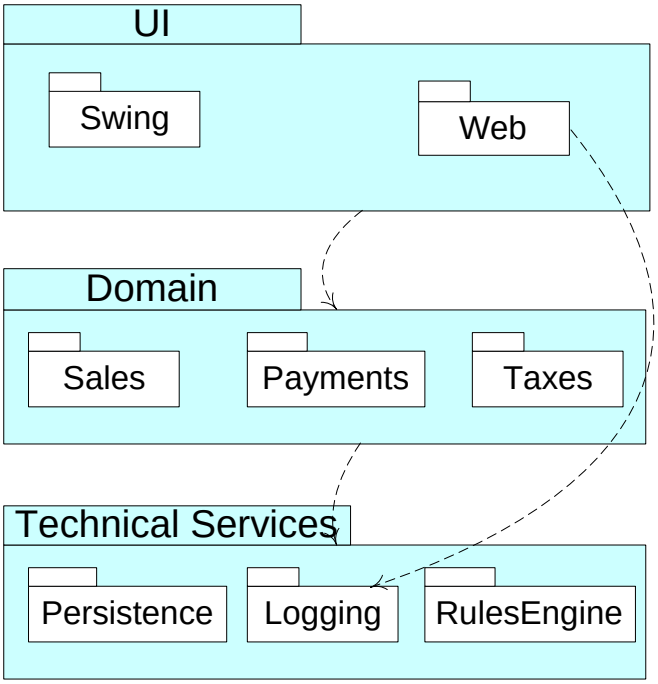
- Are in the same subject area – closely related by concept or purpose
- Are in a class hierarchy together
- Participate in the same use cases
- Are strongly associated

Package Diagrams for Logical Architecture



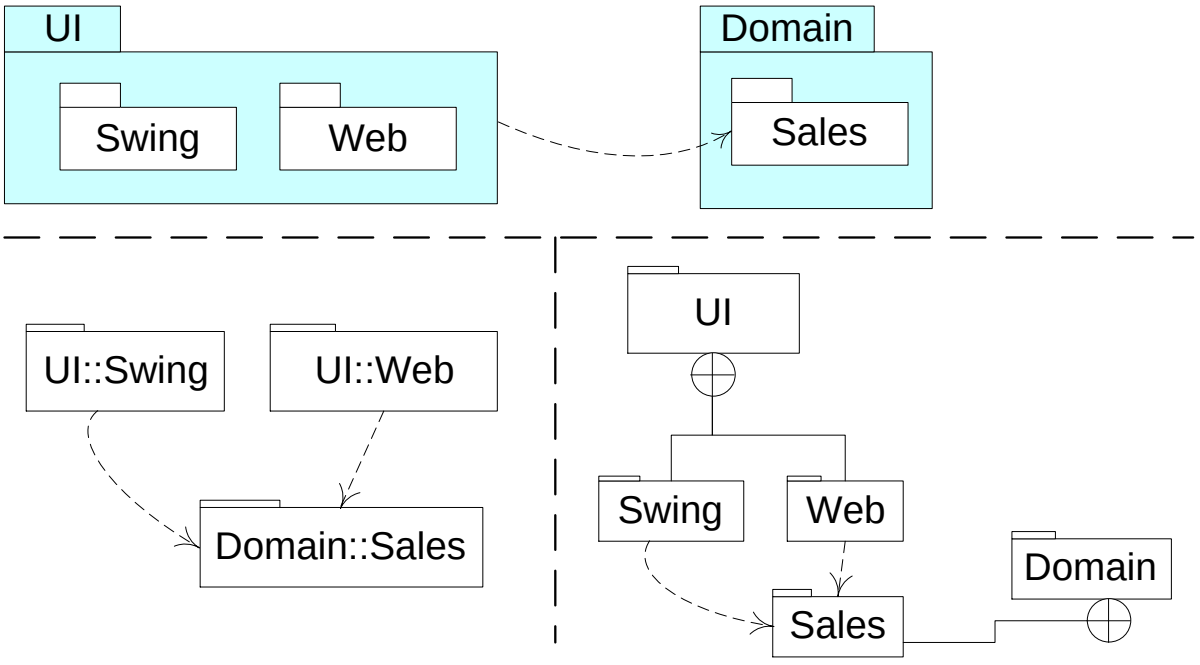
- On UML logical partitioning is illustrated with package diagrams.
- Group elements within packages
  - Classes
  - Other packages
  - Use cases, etc.
  - Grouped according to their responsibilities as it relates to the overall arch
- E.g. can abstract away persistence, logging, reporting, security by referring to the “services” package (refer to their shared traits)
- Allows for easier communication

Layers shown with UML Package Diagram



- General rule – communication happens between two different layers
  - For strict layering – communication happens between any given layer, but not across multiple layers
    - E.g. UI to Domain to Technical Services
  - For less stricter layering – can have communication across multiple layers

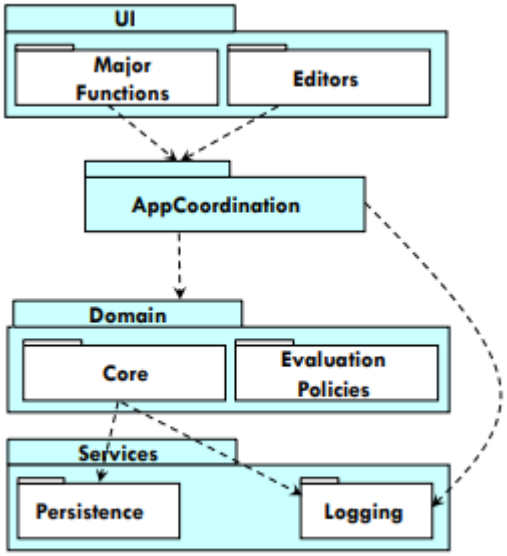
Various UML Notations for Package Nesting



- A UML package represents a namespace
- E.g. a Date class may be defined in two packages.

    - If you need to provide *fully-qualified names*, the UML notation is, for example, _java::util::Date_ in the case that there was an outer package named "java" with a nested package named "util" with a _Date_ class.
- The UML provides alternate notations to illustrate outer and inner nested packages
    - Sometimes it is awkward to draw an outer package box around inner packages
        - So can use alternative representations to capture that idea of an outer package that encloses a lot of inner packages

Package Dependencies



- In a strict layered architecture, a layer only calls upon the services of the layer directly below it.
    - This design is common in network protocol stacks, but not in information systems, which usually have a relaxed layered architecture, in which a higher layer calls upon several lower layers.
- Dependency line indicates coupling of packages
    - Arrow points to the depended upon package
    - Implies change to depended upon package likely impacts dependent package
    - Robust architectures minimize dependencies

Ordering Work

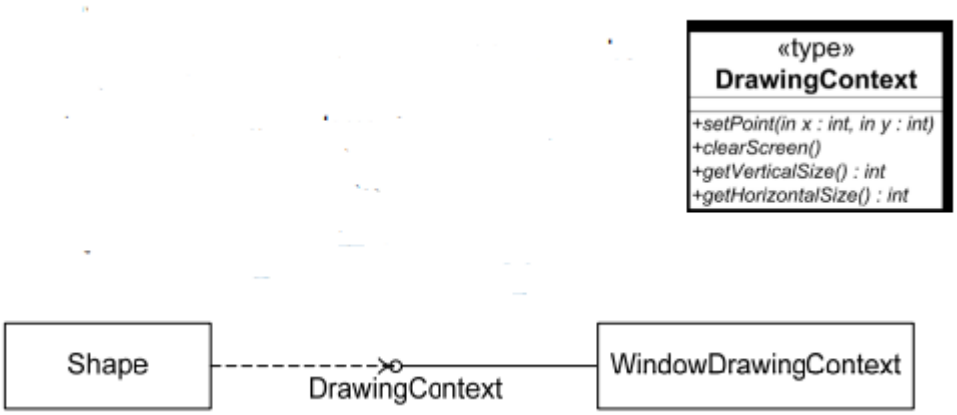
- Logical view with packages helps to order work:
    - What can we start with?
    - What can we do in parallel?
        - How?
            - Developers can work independently on different layers simultaneously

Type and Lollipop

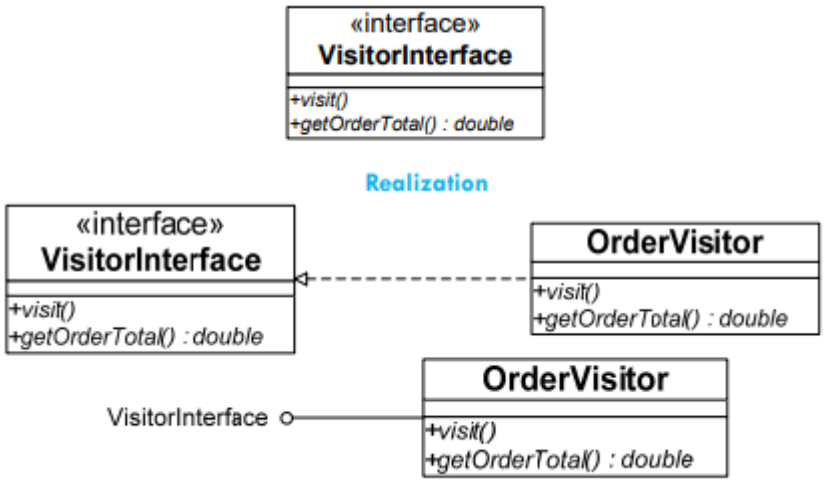
- Interface/type/abstract class allows programmers to work and progress independently

- Note: interface is similar but not the same as an abstract class.
  - It depends on the language.
  - In Java an interface is not a class and an abstract class has no code.
  - In other languages an interface has methods with no code, just the signature.
  - An abstract class may then have a few basic methods with code and instance variables that are specialized by subclasses.

- The «type» stereotype indicates that the class is an interface
  - It has no member variables, and all of its member functions are pure virtual
- A shortcut for «type» classes is the "lollipop" notation to represent an interface
  - Shape depends on DrawingContext as shown by the dashed arrow (as usual)
  - The class WindowsDrawingContext is derived from, or conforms to, the DrawingContext interface



- Can use «interface» instead of «type»



- Design Patterns in Java (Interface = abstract superclass)
  - An interface specifies the externally visible operations of a class, but not the actual implementation of those operations.
  - An interface often specifies only a part of the behaviour of an actual implementer class.

    - An interface can be drawn using a class- like rectangular setup, with the text "interface" above the name of the interface.
    - Figure above shows an interface named VisitorInterface.
- Realisation – depicts the relationship between an interface and a class that provides the actual implementation.
    - This can be drawn in two ways depending on how the interface is depicted.
        - 1. Using a closed, hollow arrowhead pointing from the implementing class to the interface with a dashed line
        - 2. With a line and a circle, where the circle represents the interface (with the name of the interface kept near the circle) and the line can be drawn pointing to the class that implements the interface represented by the circle.
- Dependency – depicts the relationship between a source and a target component, when there is a dependency relationship between the two.
    - It means, when there is a change in the target, the source element undergoes a necessary change but not vice versa

Component – a Design Level Perspective

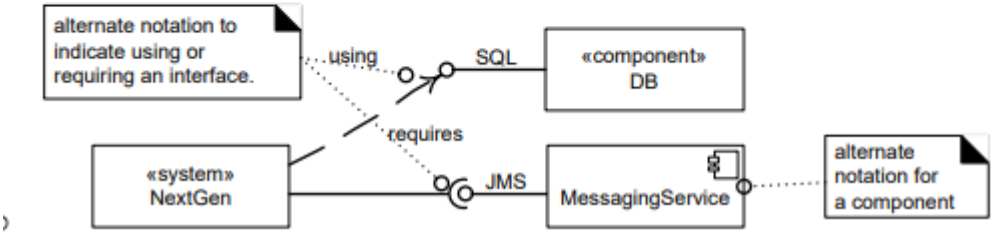
- A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.
    - Defines its behaviour in terms of provided and required interfaces
    - Serves as a type defined by these provided & required interfaces
    - Can be composed of multiple classes, or components
- Intent of using components is to emphasize
    - That the interfaces are important, and
    - It is modular, self-contained and replaceable
        - It is a (relatively) stand-alone module
- Components does not represent concrete software
    - Can map to concrete artefacts such as a set of files
- Analogy for software component modelling is a home entertainment system
    - We expect to be able to easily replace the DVD player or speaker
    - They are modular, self-contained, replaceable, and work via standard interfaces
- Components are a slightly fuzzy concept in the UML, because both classes and components can be used to model the same thing … more a matter of intent …
    - What are you communicating by labelling something as a component?
    - By dividing bits and pieces of sw into logical components

Components – UML Example

- At a large-grained level, a SQL database engine can be modelled as a component

⇒ any database that understands the same version of SQL and supports the same transaction semantics can be substituted.

- At a finer grained level, any solution that implements the standard Java Message Service API can be used or replaced in a system.



Architecture Diagrams

- The Architecture Diagram provides a graphical view of the major components in the system, and the relationships between them
- Conceptual architecture diagram communicates with various stakeholders (e.g., management, project managers for team/individual work assignments, developers and customers or users)
    - Provides a high-level view useful to non-technical audiences
    - Summarizes the entire system for technical audiences
    - i.e. Abstract enough that someone non-technical will know what's going on BUT also specific enough that can analyse overall arch of the system, plan and have expectations of what we're going to implement from a technical standpoint
- Use any appropriate UML subset (even class diagrams)
- Remember the point is capturing and conveying the information; not providing perfect UML
    - Make the architecture understood

Case Study - Subscription-Based Sensor Collection Service

- The "hello world" equivalent of an architecture description conforming to ISO/IEC 42010
    - www.iso-architecture.org/ieee-1471/docs/SBSCS-AD-v02.pdf

Version:	**v02**
Date of issue and status:	15 April 2010, approved
Issuing organization:	Dunder Mifflin and Associates, Inc.
Change history:	Version v02 was updated to reflect requirements and numbering changes between WD4 and CD1 of ISO/IEC 42010.
Summary:	This architecture provides a subscription-based service of providing access to a widely-distributed set of sensors.
Scope:	Includes only weather sensors. Does not consider acquisition or maintenance issues.
Context:	Gore and Associates commissioned this architecture study.
Glossary:	Not applicable.
Results from evaluations:	The SBSCS AD was reviewed on 6 Nov 2009 and 14 February 2010. The results of evaluations can be obtained at: https://dunder-mifflin.com/sbscs-eval
References:	Technical Memo, SCS Architecture Study, 12 March 2010

System Stakeholders and Concerns

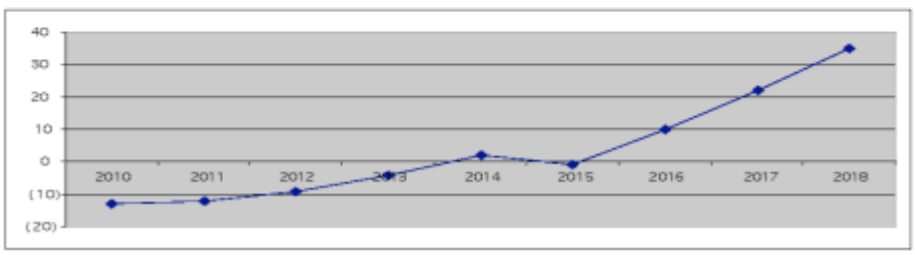
- The following stakeholders were considered and identified:
  - Users of the system
  - Operators of the system
  - Developers of the system
- The system concerns were considered, and the following concerns were identified for SBSCS:

System Concerns	Stakeholders
Return on investment	Operators
Timely delivery of sensor data	Users
Understanding of interactions between system elements	Developers

- Architecture Description uses three viewpoints: a financial viewpoint (FVP), an operational viewpoint (OVP) and a system viewpoint (SVP)
  - FVP helps operators understand return on investment
  - OVP helps users understand what is going on with timely delivery
  - SVP helps devs understand interactions between system elements

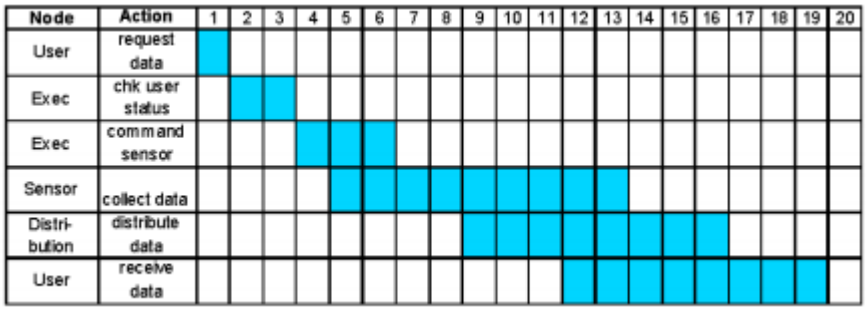
SBSCS — Financial View

Year	2010	2011	2012	2013	2014	2015	2016	2017	2018
Income	0	2	4	6	8	12	14	16	18
Expenses	(13)	(1)	(1)	(1)	(2)	(15)	(3)	(4)	(5)
Profit	(13)	1	3	5	6	(3)	11	12	13
ROI	(13)	(12)	(9)	(4)	2	(1)	10	22	35



- Overview: This view projects that SBSCS will achieve breakeven after five years of system operation
- Models:
    - Model ID: SCS profit statement; Version: v1.1; Model kind: cash flow statement. Shown in figure 1
    - Model ID: SCS profit statement; Version: v1.1; Model kind: cash flow statement. Shown in figure 1

SBSCS — Operational View

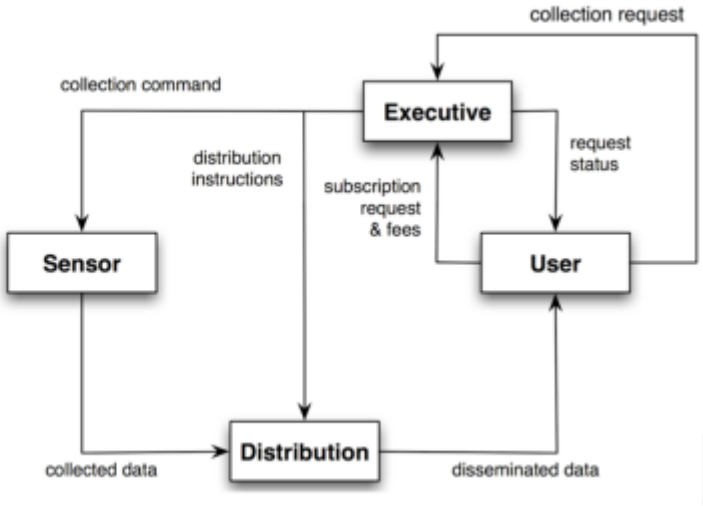


- This view shows that a typical user request will be satisfied within 20 seconds
- Model ID: Collection TLD; Version: v2.4; Model kind: Timeline diagram.

SBSCS — System View

NodeCheck: Each node in a dataflow diagram must appear at least once in the timeline diagram with its corresponding response time for that node.

Assessment of **NodeCheck**: This rule holds true for this SBSCS AD. All nodes in SCS Dataflow are present in Collection TLD. Each entry in Collection TLD specifies a response time.



- This view shows system nodes and dataflow between nodes
- Main components:
    - Sensor, Executive, User, Distribution
- Have relationships between these components
- Model ID: SCS Dataflow; Version: v0.5; Model kind: Dataflow diagram

Guidelines: Architectural Analysis

- Start architectural analysis before the first cycle
    - Can start early iterations before architectural analysis is complete
- It is mainly concerned with non-functional requirements
    - Quality attribute

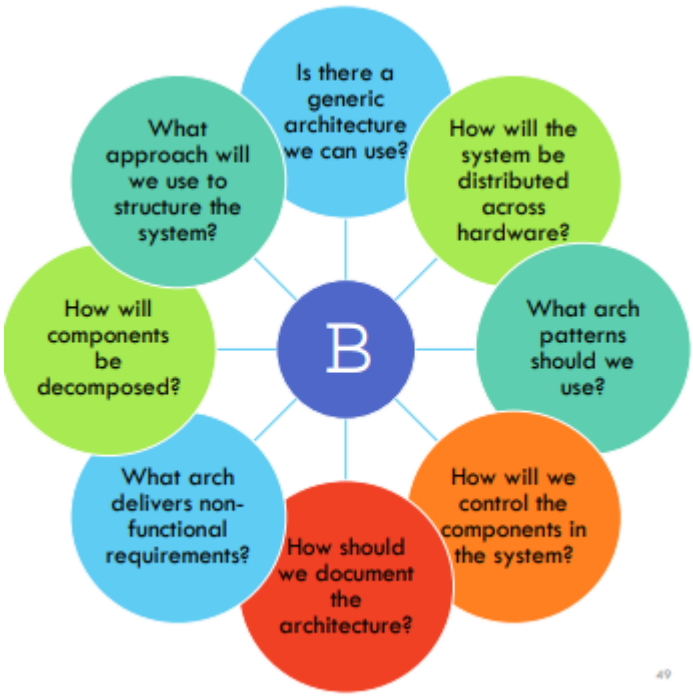
    - e.g., security
    - Not concerned with whether something works but rather the quality with which it works
- Within the context of the functional requirements
    - e.g., processing sales
- Examples of issues to be identified and resolved:
    - How do reliability and fault-tolerance requirements affect the design?
    - How do the licensing costs of purchased subcomponents affect profitability?
    - How do the adaptability and configurability requirements affect the design?

Common Steps In Architectural Analysis

- 1. Identify and analyse the non-functional requirements that have an impact on the architecture
    - Architectural factors (or drivers)
- 2. For those requirements with a significant architectural impact, analyze alternatives and create solutions that resolve the impact
    - Architectural decisions
    - Thinking through all possible solutions and analysing – how do they weigh against each other ito all the functional requirements (to resolve impact)
        - Some solutions are harder to implement at first, but will save time in the long run – other may be too much overhead e.g. for the amount of fault tolerance you need

Identification and Analysis of Architectural Factors

- Quality Scenarios
    - Form: <stimulus> <measurable response>
    - Record non-functional architectural factor in a measurable form
- Example:
    - When the completed sale is sent to the remote tax calculator to add the taxes, the result is returned within 2 seconds “most” of the time, measured in a production environment under “average” load conditions
    - When a bug-report arrives from a test volunteer, reply with a phone call within 1 working day
- No point in describing scenarios that will never be tested before shipping
- When defining quality requirements during architectural factor analysis, quality scenarios, are recommended
- Comes back to Architectural Design Decisions
    - i.e. there are questions that can be asked that can help you hone in on what is the appropriate architecture for all stakeholders involved



The Basic Architectural Design Principles

- Low coupling (separate components are not densely linked)
    - It is low coupling between applications, subsystems, or process rather than between small objects
- High cohesion (elements within a given component are strongly related)
- Separation of concerns and localization of impact
    - One could design persistence support such that each object also communicated with a database to save its data
    - The concern of persistence is then mixed in with the concern of application logic
    - And same with security, etc.
        - Cohesion drops and coupling rises
    - Recommend: factor out persistence, security, …
        - Object with application logic just has application logic
        - Persistence subsystem focuses on the concern of persistence
        - Security subsystem doesn't do persistence

Architecture Patterns

Architecture Centred Approach

- Place an emphasis on design
    - Design pervades engineering activities
    - Important aspect of Computer Science
- An incorrect interpretation of the lean documentation approach of Agile development leads to inadequate levels of architectural design information
- Architecture Decisions are system-wide Strategic Decisions
    - Use Object-Orientation
    - Choose overall Architectural pattern
    - Choice of API
- In waterfall, much of the design is done up front, and this can lead to brittle systems that aren't resilient to changes in requirements.
    - It's tempting, then to think that agile approaches require an abandonment of early design.
    - However, this is incorrect – in agile we want design done early at a high level.
        - Object-orientation generally ensures clearer code that is easier to refactor if necessary.
        - Patterns help us to make systems-wide decisions that account for strategic needs without a lot of time spent designing solutions from scratch.
        - Clear APIs will enable developers working on separate components to work independently.

Patterns and Architecture

- Patterns come from real architecture
- "Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use this solution a million times over without ever doing it the same way twice."
- Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution."
- Definition: a pattern is a solution to a problem in a context.

What's different about Software Architecture

- Software is not embedded in space
    - Often no constraining physical laws – so not subject to physical laws and constraints beyond the capabilities of the hardware running it.
    - Software is (infinitely) malleable
    - Computing power and memory can be limitations but they can be solved with clusters
- No obvious representation
    - E.g., no familiar geometric shapes

    - Software is more conceptual – there's no clear physical representation of underlying structures
- Software does stuff (it is active) – it can change the environment around it and even change itself

Aims of Software Patterns

- The aim is to enhance reusability of object-oriented code
    - Well-structured object-oriented systems have recurring patterns of classes and objects
    - Makes code easier to analyse and read
    - Knowledge of the patterns that have worked in the past allows:
        - A designer to be more productive and
        - The resulting designs to be more flexible and reusable

Design Pattern Benefits

- Capture expertise and make it accessible to non-experts in an encapsulated design pattern
- When systems are being implemented by large teams of developers the design pattern forms a basis for communication between them
    - Help communication amongst developers by providing a common language
        - Improve design understandability
- Make it easier to reuse successful designs and avoid alternatives that diminish reusability
- Facilitate design modifications
    - The design is more easily understood
    - The system is designed using the pattern and as a result is better understood by the user, developers and implementation team
- Improve design documentation
    - The system documentation starts with the UML design pattern
    - The design pattern forms the basis of the documentation of the system

Architectural Patterns

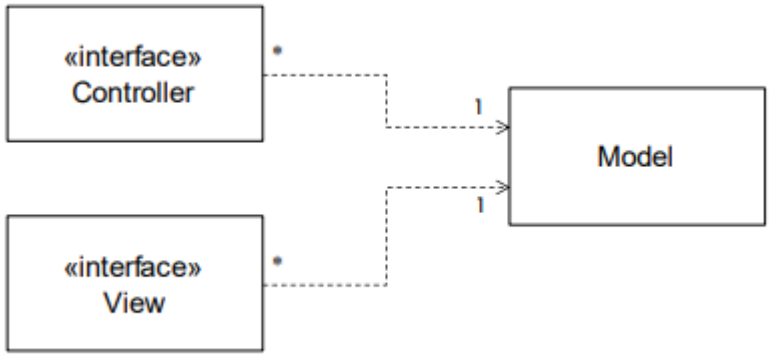
- An architectural pattern is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears
- These are incorporated into the model, visualisation and view
- Architecture Model: an artefact documenting some or all of the architectural design decisions about a system
- Architecture Visualization: a way of depicting some or all of the architectural design decisions about a system to a stakeholder
- Architecture View: a subset of related architectural design decisions
- Styles vs Patterns
    - Many people make a distinction between Patterns and Styles
        - Styles provide a vocabulary (or language) to discuss a design
        - Styles require less domain knowledge and have a wider range of applicability than patterns
    - Treat Styles as high-level coarse grained Patterns (this course)

Key Patterns

Model-View-Controller Pattern

- Problem: used when there are multiple ways to view and interact with data.
    - Also used when the future requirements for interaction and presentation of data are unknown.
    - i.e. have an idea of what the data is but want to be able to represent the data in different ways – separate the presentation and the interaction from sys data
- Solution: separates presentation and interaction from the system data
    - The system is structured into three logical components that interact with each other
    - The Model component manages the system data and associated operations on that data
    - The View component defines and manages how the data is presented to the user
    - The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model
- Pro: allows the data to change independently of its representation and vice versa.
    - Supports presentation of the same data in different ways with changes made in one representation shown in all of them
- Con: can involve additional code and code complexity when the data model and interactions are simple

MVC Essential Dependencies Diagram



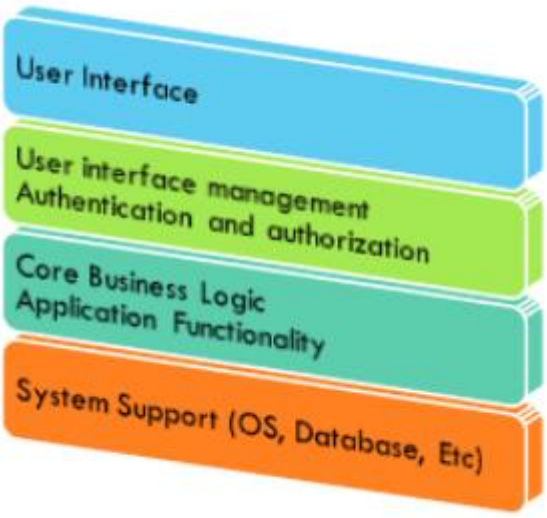
- View actually can link to controller (when window gets exposed etc)
- One-to-many relationship between Model and Controller, Model and View
    - For a given model, can have many possible different views and ways to control that model
- View and Controller are actually «interfaces» and so in reality need concrete classes to realize them

Layered Architecture Pattern

- Problem: used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
- Solution: organizes the system into layers with related functionality associated with each layer.

    - A layer provides services to the layer above it (and can receive permission from the layer below it) so the lowest-level layers represent core services that are likely to be used throughout the system
- Pro: allows replacement of entire layers so long as the interface is maintained.
    - Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system
- Con: in practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it.
    - Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer

Layered Architecture Diagram



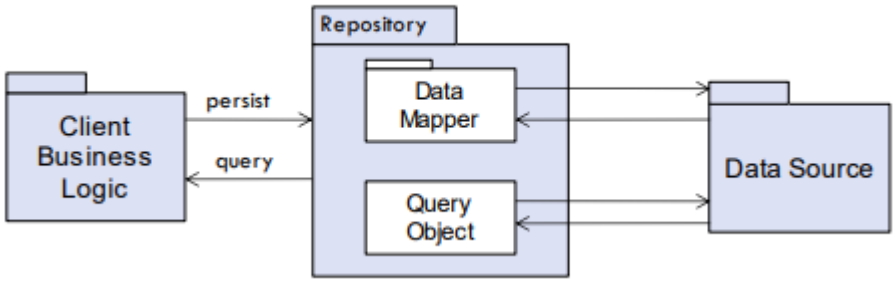
- Above diagram – a generic layered architecture
- Keep all these concerns separated ensures code remains cohesive

Repository Pattern

- Problem: you should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time.
    - You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
- Solution: all data in a system is managed in a central repository that is accessible to all system components.
    - Components do not interact directly, only through the repository
- Pro: components can be independent—they do not need to know of the existence of other components.
    - Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
- Con: the repository is a single point of failure so problems in the repository affect the whole system.
    - May be inefficiencies in organizing all communication through the repository.

- Distributing the repository across several computers may be difficult (depending on the nature of the diagram)

Repository Diagram



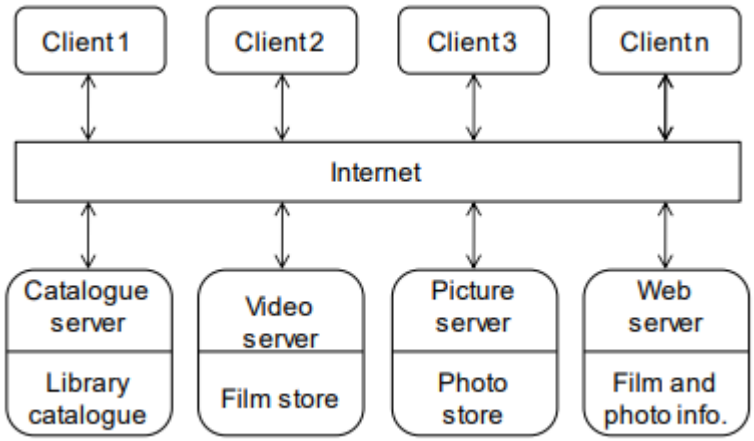
- Repo is central – has the sole relationship with that data source
- Clients can interact with repo – can have many clients that are implemented differently interacting with the central repo

Client-Server Pattern

- Problem: used when data in a shared database has to be accessed from a range of locations.
  - Because servers can be replicated, may also be used when the load on a system is variable.
- Solution: in a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server.
  - Clients are users of these services and access servers to make use of them.
- Pro: the principal advantage of this model is that servers can be distributed across a network and servers can added or upgraded with minimal disruption.
  - General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all.
- Con: each service is a single point of failure so susceptible to denial of service attacks or server failure.
  - Performance may be unpredictable because it depends on the network as well as the system.
  - May be management problems if servers are owned by different organizations.

Client-Server Diagram

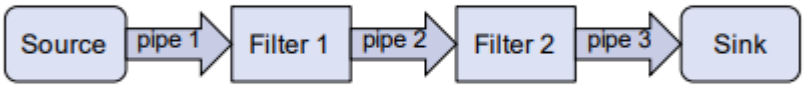
- For a Film Library

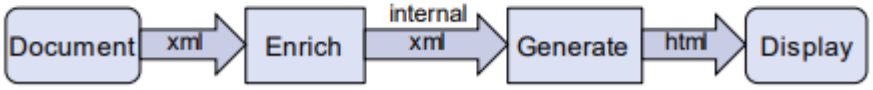


Pipe and Filter Pattern

- Problem: commonly used in data processing applications (both batch- and transaction based) where inputs are processed in separate stages to generate related outputs.
- Solution: the processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation.
    - The data flows (as in a pipe) from one component to another for processing.
- Pro: easy to understand and supports transformation reuse.
    - Workflow style matches the structure of many business processes.
    - Evolution by adding transformations is straightforward.
    - Can be implemented as either a sequential or concurrent system.
- Con: the format for data transfer has to be agreed upon between communicating transformations.
    - Each transformation must parse its input and unparse its output to the agreed form.
    - This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures

Pipes and Filters to Break Processing into Steps





- Pipe and filters are useful for data processing – have a particular source and you're going to run it through filters

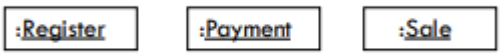
Key Aims

- Key aim of these patterns for design
- Cohesion - degree to which communication takes place within the module
- Coupling - degree to which communication takes place between modules
- Minimize coupling while maximizing cohesion

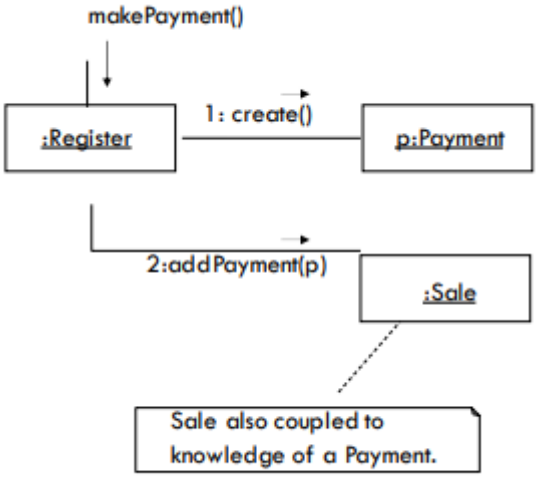
Low Coupling

- Coupling: a measure of how strongly one element is connected to, has knowledge of, or relies upon other elements
- A class with high coupling depends on many other classes (libraries, tools)
- Problems because of a design with high coupling:
    - Changes in related classes force local changes.
    - Harder to understand in isolation; need to understand other classes
    - Harder to reuse because it requires additional presence of other classes
- Problem: How to support low dependency, low change impact and increased reuse?
- Solution: Assign a responsibility so that coupling remains low

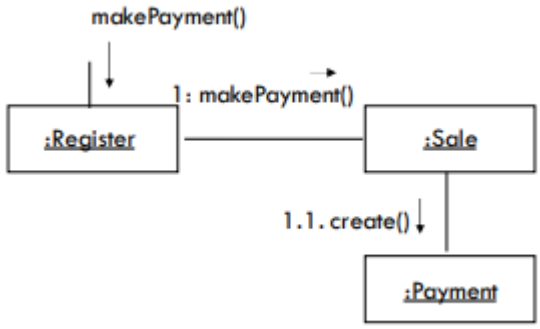
Example



- Assume we need to create a Payment instance and associate it with the Sale.
- What class should be responsible for this?
- Register is a candidate.



- Register could then send an addPayment message to Sale, passing along the new Payment as a parameter.
- The assignment of responsibilities couples the Register class to knowledge of the Payment class



- An alternative solution is to create Payment and associate it with the Sale
- No coupling between Register and Payment.

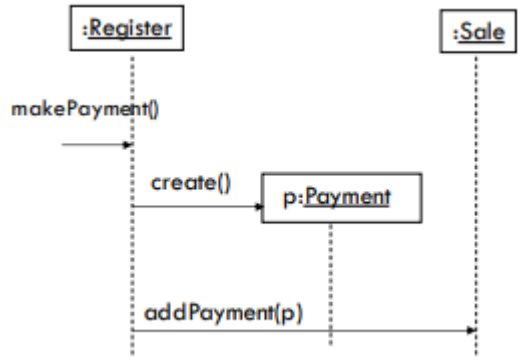
Low Coupling Continued

- Some of the places where coupling occurs:
  - Attributes: X has an attribute that refers to a Y instance
  - Methods: e.g. a parameter or a local variable of type Y is found in a method of X
  - Subclasses: X is a subclass of Y
  - Types: X implements interface Y
- There is no specific measurement for coupling, but in general, classes that are generic and simple to reuse have low coupling
- There will always be some coupling among objects, otherwise, there would be no collaboration

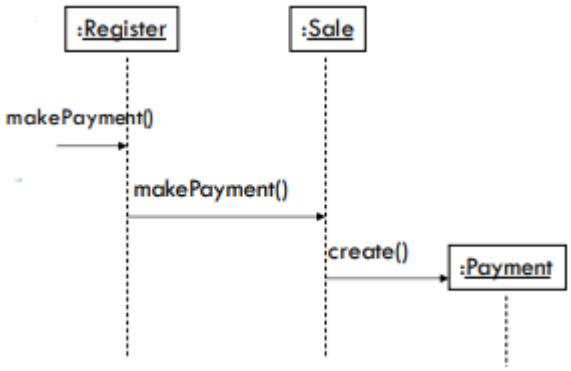
High Cohesion

- Cohesion: it is a measure of how strongly related and focused the responsibilities of an element are.
- A class with low cohesion does many unrelated activities or does too much work
- Problems because of a design with low cohesion:
  - Hard to understand
  - Hard to reuse
  - Hard to maintain
  - Delicate, affected by change.
- Problem: How to keep complexity manageable?
- Solution: Assign a responsibility so that cohesion remains high.

Example



- Assume we need to create a Payment instance and associate it with Sale. What class should be responsible for this?
- Register is a candidate
    - Register may become bloated if it is assigned more and more system operations



- An alternative design delegates the Payment creation responsibility to the Sale, which supports higher cohesion in the Register.
    - This design supports high cohesion and low coupling

High Cohesion Continued

- Many scenarios that illustrate varying degrees of functional cohesion
- Very low cohesion: class responsible for many things in many different areas.
    - e.g.: a class responsible for interfacing with a data base and remote-procedure-calls
- Low cohesion: class responsible for complex task in a functional area
    - e.g.: a class responsible for interacting with a relational database
- High cohesion: class has moderate responsibility in one functional area, and it collaborates with other classes to fulfil a task.
    - e.g.: a class responsible for one section of interfacing with a database
- Rule of thumb: a class with high cohesion has a relative low number of methods, with highly related functionality, and doesn't do much work. It collaborates and delegates

Non-Functional Requirements

- Low Coupling and High Cohesion are high-level aims that help us to assess architectural options.
- But the key influencers of software architecture are the non-functional requirements.
- Performance
    - Localize critical operations and minimize communications.
    - Use large rather than fine-grain components.
    - Allow for concurrency.
- Security
    - Use a layered architecture with critical assets in the inner layers.
- Safety
    - Localize safety-critical features in a small number of subsystems.
- Availability
    - Include redundant components and mechanisms for fault tolerance.

    - Allow replacement of components without stopping the system.
- Maintainability
    - Use fine-grain, replaceable components
    - Avoid global/shared data structures.
- Some recommendations will conflict with one another – you want large components for performance – but fine-grain ones for maintainability
    - Some of our decisions entail tradeoffs

Purpose of Architecture – Beforehand

- Software architecture can serve as a design plan for the negotiation of system requirements
- A means of structuring discussions with clients, developers, and managers.
- It is an essential tool for complexity management.
    - It hides details and allows the designers to focus on the key system abstractions

Afterwards

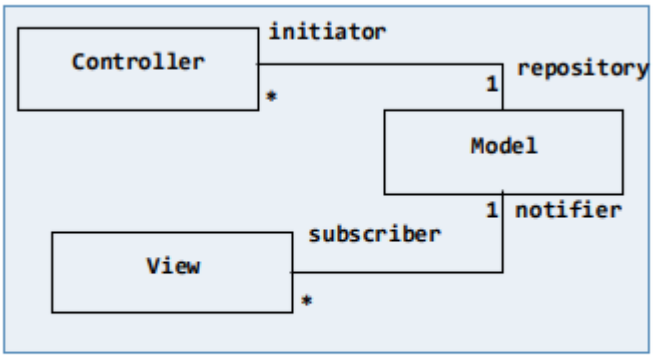
- Documenting an architecture that has already been designed
    - The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.
    - Easier to understand and evolve system
- If you need it:
    - Use a notation with a well-defined semantics, such as UML
    - Start with automated documentation for lowlevel components
- Probably not worth it unless you are producing (safety-, mission-) critical systems

Architecture Diagrams

- Simple Block Diagrams:
    - Simple, informal block diagrams showing entities and relationships
    - Most frequently used method
    - Good for communicating at a high level of abstraction
- Diagrams with Rich Semantics
    - Do not show the types of relationships between entities
    - Do not show visible properties of entities (interfaces)
    - Costly to produce

Model-View Controller

MODEL-VIEW-CONTROLLER (MVC) TRIAD



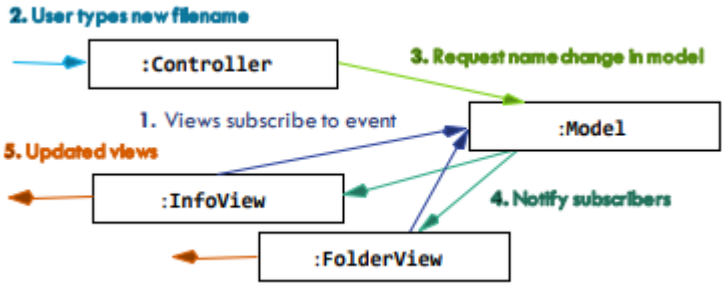
Model	representation of the application
View	way information is displayed
Controller	mapping from user actions to operations on model

- Architecture for interactive applications.
    - A model (or application) is created independently of the user interface

Subsystems

- Subsystems are classified into 3 different types
    - Model subsystem: Responsible for application domain knowledge
    - View subsystem: Responsible for displaying application domain objects to the user
    - Controller subsystem: Responsible for sequence of interactions with the user and notifying views of changes in the model.
- MVC is a special case of a repository architecture:
    - Model subsystem implements the central data structure, the Controller subsystem explicitly dictate the control flow
- The forces one to think of the application in terms of these three modules
    - Model – the core of the application
        - This maintains the state and data that the application represents
        - When significant changes occur in the model, it updates all of its views
    - Controller – the user interface presented to the user to manipulate the application
    - View – the user interface which displays information about the model to the user
- This application architecture is very similar to a client/server model, except that all the components are bundled into one application

Example – MVC in File System



- Consider file system with a folder (directory) view and an information (properties) view

- Sequence of events
- A model in MVC can have several views
    - Two examples are the rows and columns view of a spreadsheet and the pie chart view of some column in the same spreadsheet

Components

Model

- In MVC the model is the code that carries out some task
- It is built with no necessary concern for how it will "look and feel" when presented to the user
- It has a purely functional interface, meaning that it has a set of public functions that can be used to achieve all of its functionality
- Some of the functions are query methods that permit a "user" to get information about the current state of the model
- Others are mutator methods that permit the state to be modified
- However, a model must be able to "register" views and it must be able to "notify" all of its registered views when any of its functions cause its state to be changed.
- In Java a Model consists of one or more classes that extend the class java.util.Observable
- This superclass will provide the register/notify infrastructure needed to support a set of views

Views

- A view provides graphical user interface (GUI) components for a model. It gets the values that it displays by querying the model of which it is a view
- When a user manipulates a view of a model, the view informs a controller of the desired change
- In Java the views are built of AWT or SWING components
    - However, views must implement the java.util.Observer interface

Controller

- Views in MVC are associated with many controllers that update the model as necessary when a user interacts with an associated view.
- The controller can call mutator methods of the model to get it to update its state
- Of course, then the model will notify ALL registered views that a change has been made and so they will update what they display to the user as appropriate.
- In Java the controllers are the listeners in the Java event structure

Model Implementation

- First you write a Model that extends java.util.Observable.
- You give the class accessors to get information about its current state
- You write mutators to update the state
- Each mutator should call setChanged() and one or the other of notifyObsvers() after it has actually changed the state
- notifyObservers will send an update message to each registered observer (View)
- There are versions of notifyObservers that let you pass additional information about the change as well
- Each view must implement the java.util.Observer interface and hence the update method

- Second parameter will receive additional information if passed

```
interface Observer {
    void update(Observable t, Object o);
}
```

- View implements the update method by querying the model (Observable t) for the changes
    - Then makes appropriate changes to the view itself
- View registers with the Model it observes by sending addObserver
- Model remembers all views so as to notify (update) them later
    - Parts of a model can be separately Observable, with own Observers
- The reason that a Model has to extend java.util.Observable is that the Observable class provides all of the register/notify infrastructure needed by a model, so you don't have to build any of this and can concentrate on the functionality of your application
    - A model can have several views - MVC was created specifically to permit this
    - Also, a view can register with several models and get updates from each of them.
- It is not necessary to build a Model so that it is a single Observable.

Simple Example

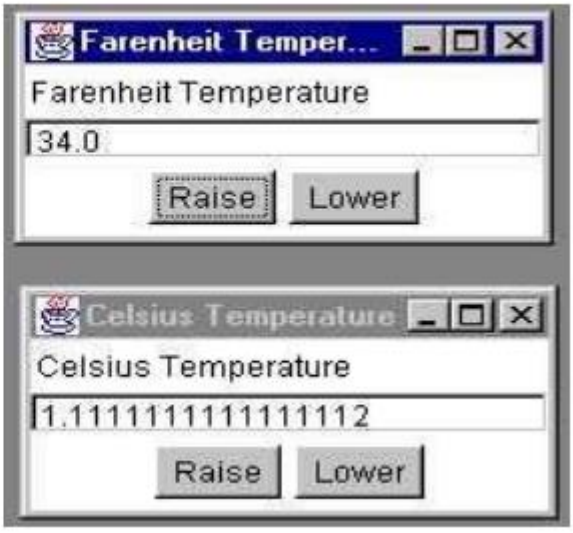
```
public class TemperatureModel extends java.util.Observable {
   public double getF() {return temperatureF;}
   public double getC() {return (temperatureF - 32.0) * 5.0/9.0;}
   public void setF(double tempF) {
     temperatureF = tempF;
     setChanged();
     notifyObservers();
   }
   public void setC(double tempC) {
     temperatureF = tempC*9.0/5.0 + 32.0;
     setChanged();
     notifyObservers();
   }
   private double temperatureF = 32.0;
 }
```

- It encapsulates the notion of a temperature in either Fahrenheit or Celsius
    - Model that is actually too simple to really be built as a class.
- We can ask for its value in either type of unit and we can likewise set its value using either kind of unit
    - Notice that there are no GUI elements here, but there is some infrastructure, namely the setChanged and notifyObserver calls in the mutators.

Simple GUI for Model

EXAMPLE: MULTIPLE VIEWS

- Designed the system to have two different views
  - One for Fahrenheit
  - The other for Celsius
  - simultaneously attached to the model.
- When you manipulate either of them, the other is also automatically updated.



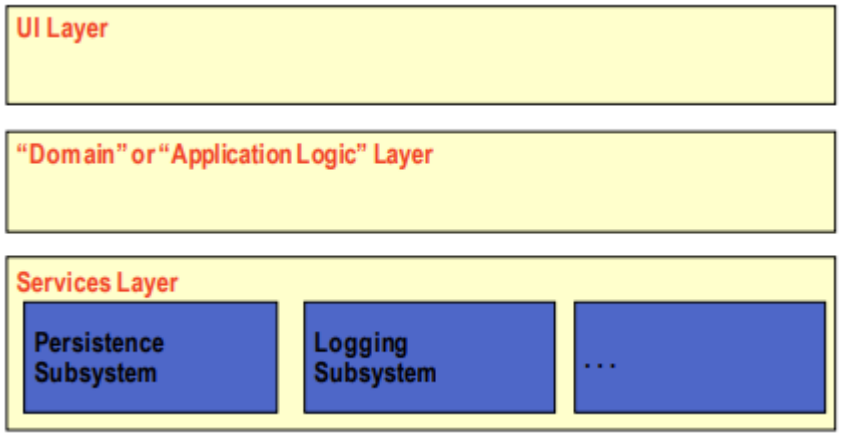
- We want a simple textual view of the temperature along with two buttons, one to increase and the other to decrease the temperature by one degree.
- The user can also put a number into the text field and hit the "Enter" key.
- We can actually have two different GUIs with these properties, one for Fahrenheit and the other for Celsius temperatures.
- It is important to note that these are separate GUIs
  - Also, both are connected to the same model

Abstract Class for Temperature Views

```
abstract class TemperatureGUI implements java.util.Observer {
   TemperatureGUI(String label, TemperatureModel model, int h, int v) {
    this.label = label;
    this.model = model;
    temperatureFrame = new Frame(label);
    temperatureFrame.add("North", new Label(label));
    temperatureFrame.add("Center", display);
    Panel buttons = new Panel();
    buttons.add(upButton);
    buttons.add(downButton);
    temperatureFrame.add("South", buttons);
 etc ...
```

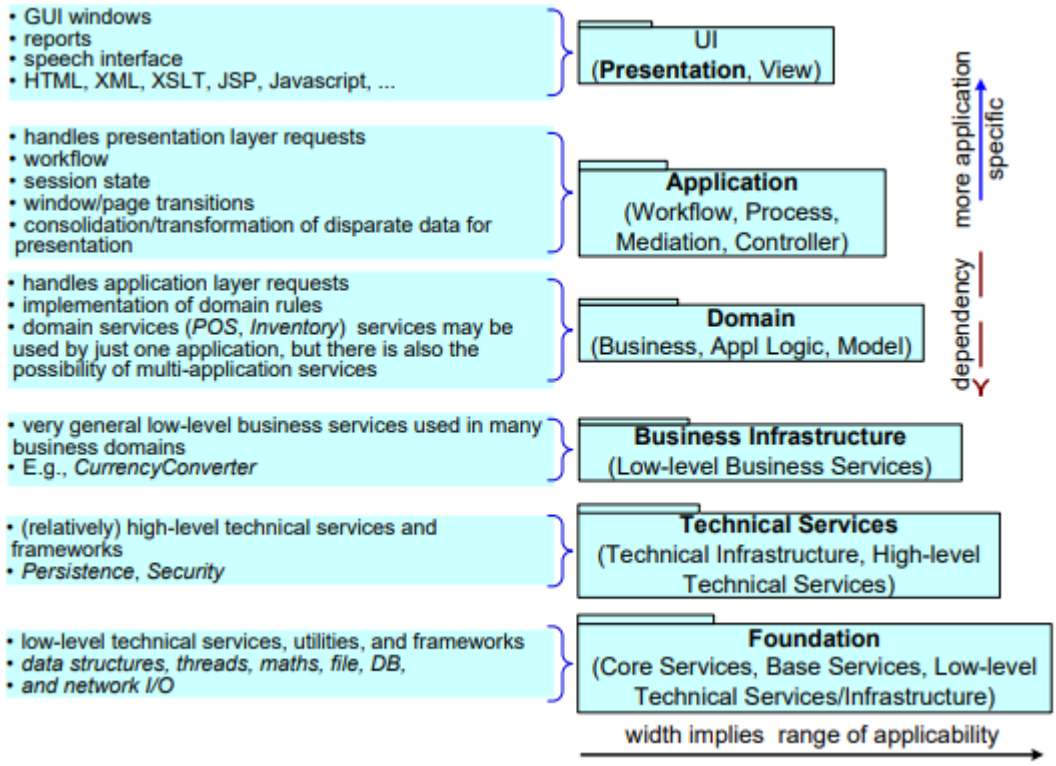
- Since these two GUIs are so similar, they are built as subclasses of a general TemperatureGUI class
- This has all of the functionality, but has no listeners for its buttons and field
- It is abstract and doesn't implement the required update method of the Observer class
- The view does not hold any information internally about the current temperature in the model
- This way it is never "out of date"
  - It always gets the temperature by querying the model

Layered Architecture

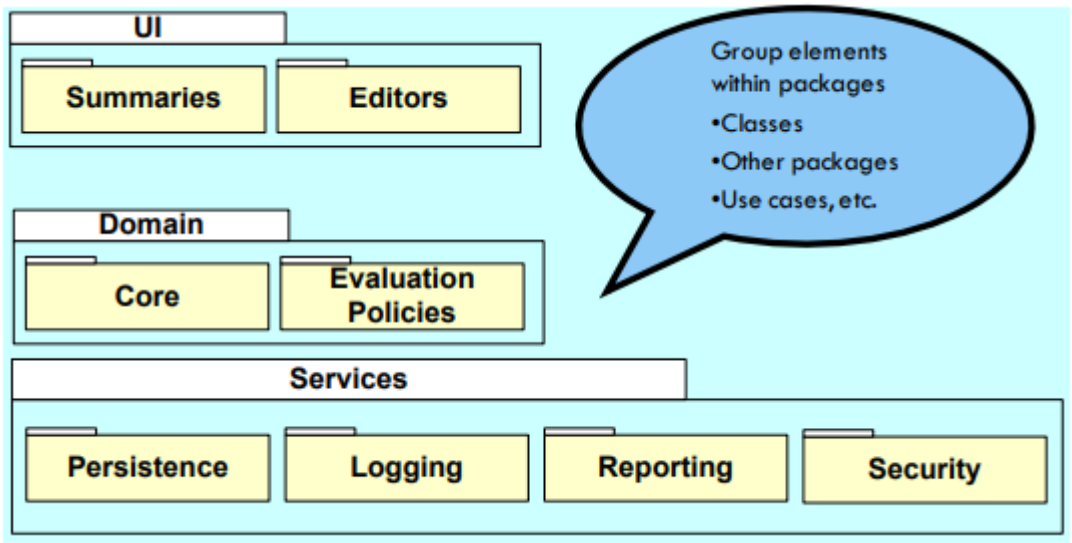


- If have multi-tiered layered archs – separate presentation and application logic, and other areas of concern
    - So if want to e.g. change UI layer – can pull it out without changing any of the application logic
- Logical architecture organised into layers
    - Layer: very coarse-grained grouping of classes (or packages or subsystems) with cohesive responsibility for a major aspect of the system.
- Relationship between layers:
    - "Higher" layers call services of "lower" layers
    - Strict Layered Architecture – layer uses only layer directly below
    - Relaxed – can use several lower layers
        - Potential problems with coupling – if change the lwr layer, then could change all the other layers interfacing with it
- Typical layers in an OO system:
    - UI
    - App Logic and Domain Objects
    - Technical Services
        - Application-independent, reusable across systems

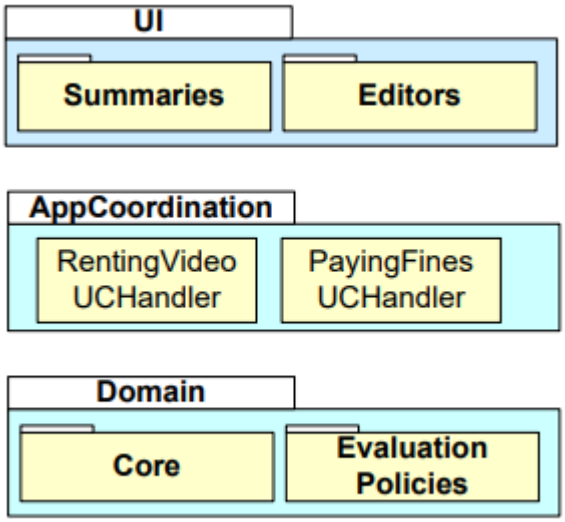
Layering Scheme



Example



- In the UML, the logical partitioning is illustrated with package diagrams
- Each layer have sub-packages/sub-elements that're grouped logically and then stacked on top of each other



- Consider an "application coordination layer" whose objects represent use cases.
    - Use cases will integrate with domain
    - They may also hold session state

Model-View Separation Principle

- Tiers may seem similar to the model-view-controller (MVC) concept; however, topologically they are different
    - A fundamental rule in a three-tier architecture is the client tier never communicates directly with the data tier; in a three-tier model all communication must pass through the middle tier.
    - Conceptually the three-tier architecture is linear.
    - However, the MVC architecture is triangular: the view sends updates to the controller, the controller updates the model, and the view gets updated directly from the model.
- Model ≡ domain layer
- View is the User Interface (UI) layer
    - Model objects shouldn't have knowledge of View objects
    - Don't assign domain responsibility to View objects
- Better separation of concerns with lower coupling
    - When domain layer independent of UI layer, domain layer can be used with another UI layer
- Facilitates multiple views of a given Model
    - Example: Application and web UIs for a given application

Benefits of Layered Architecture

- System easier to comprehend: each layer has specific purpose
    - i.e. easier to, given a functionality, find out which layer it is in
- Facilitates 'high-cohesion' within layers and 'low-coupling' between layers
- Prevents source code changes from rippling throughout system
    - Code changes should be confined within a specific layer
- Minimizing coupling makes it easier to swap out a technical service in a lower layer and replace it with another technology

    - Replacing RDBMS without major code re-write in application logic
    - RDMS=Relational Database management system
    - E.g. want to switch out MySQL for another dtb
- Lower layers contain more reusable functionality

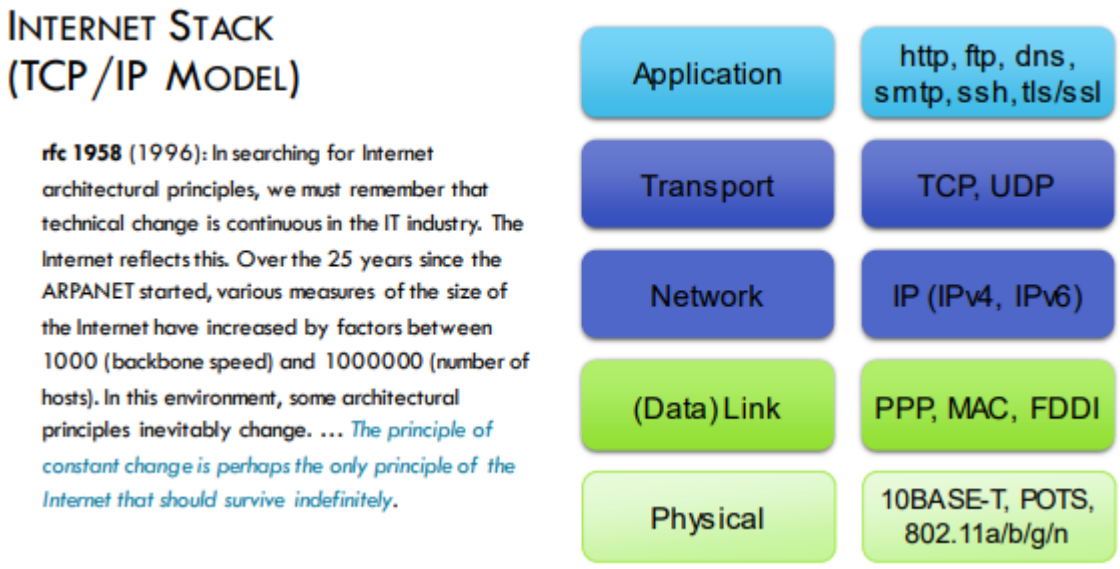
Reference Architectures

- Idealized way of discussing and comparing domain-specific architectures
- They do not represent actual real systems
- Reference architecture features make them easier to describe and understand
    - A reference model provides a vocabulary for comparison
    - It acts as a base against which systems can be evaluated

OSI Model for Distributed Systems

- OSI stack is seven-layer model for open systems interconnection
    - Lower layers: physical interconnection,
    - Middle layers: data transfer
    - Upper layers: semantically meaningful application information.
- Allow conformant systems to communicate with each other
- Each layer should only depend on the layer beneath it
    - A layer could be transparently reimplemented
    - E.g. swap out mobile conn for wi-fi conn and app won't break
- Performance problems with this layered approach
    - Vast physical differences between networks
- Well defined functional characteristics but not the non-functional
- Developers implement their own higher-level facilities and skip layers in the model

TCP/IP Model



- Highly resilient architecture

Harms of Layering

- Networking layering => functions of each layer are carried out completely before protocol data unit is passed to the next layer
    - Optimization of each layer has to be done separately
    - Hides information that lower layers may need to optimize performance
    - Layered model (TCP/IP & ISO OSI) causes conflict:
        - Layer N may duplicate lower-level functionality (hop-h
        - Layers may need the same information (time stamp)
        - Layer N may need layer N-2 information (lower layer packet sizes)
- Increased layering ⇒ increased complexity (via inter-layer dependencies)
    - "It is always possible to agglutinate multiple separate problems into a single complex interdependent solution. In most cases this is a bad idea."
- Conclusion: horizontal separation may be more cost-effective and reliable than vertical

Example - Basic Web 2.0 Reference Architecture Diagram

Resource tier: Capabilities or backend systems (data or processing) that support services consumed over the Internet:

Service tier: packages resources as a service and controls what goes in and out.

- Application servers deploying SOAP, EJB, PHP, Rails, ASP, …

Connectivity: means to reach service

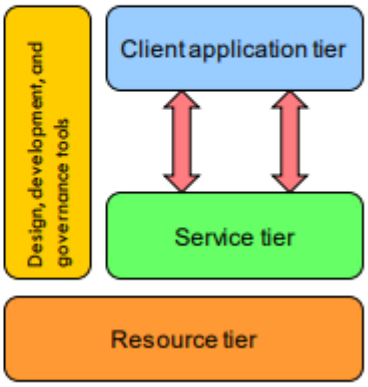
- Standards/protocols like XML over HTTP

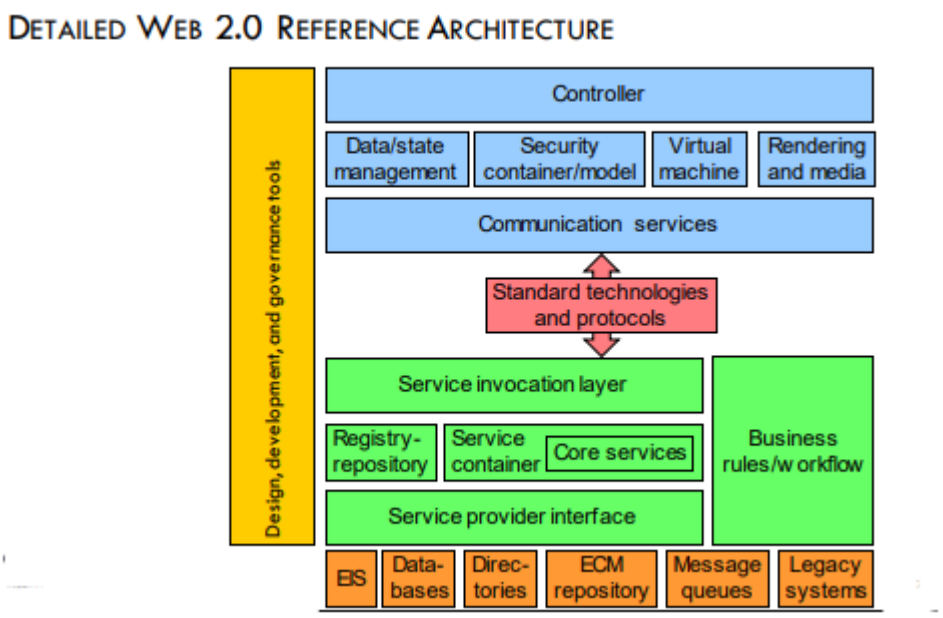
Client tier: user view of services

- Web browsers, Flash, Acrobat, iTunes, …

Design, development, and governance tools to build web apps

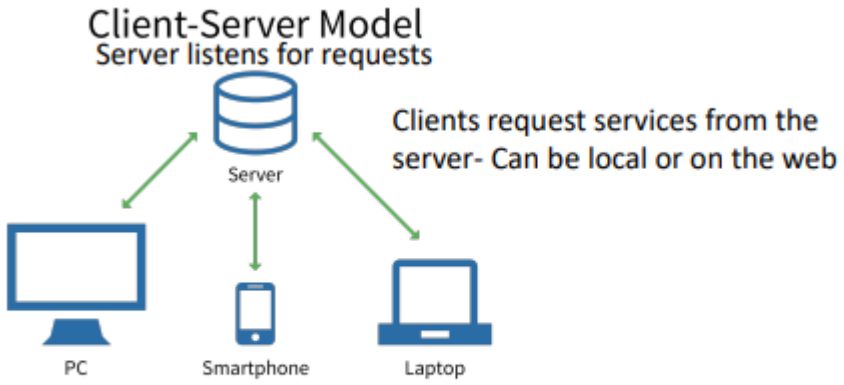
- Adobe Dreamweaver and Apple's tools (xCode, DashCode), other IDEs





Architecture Patterns II

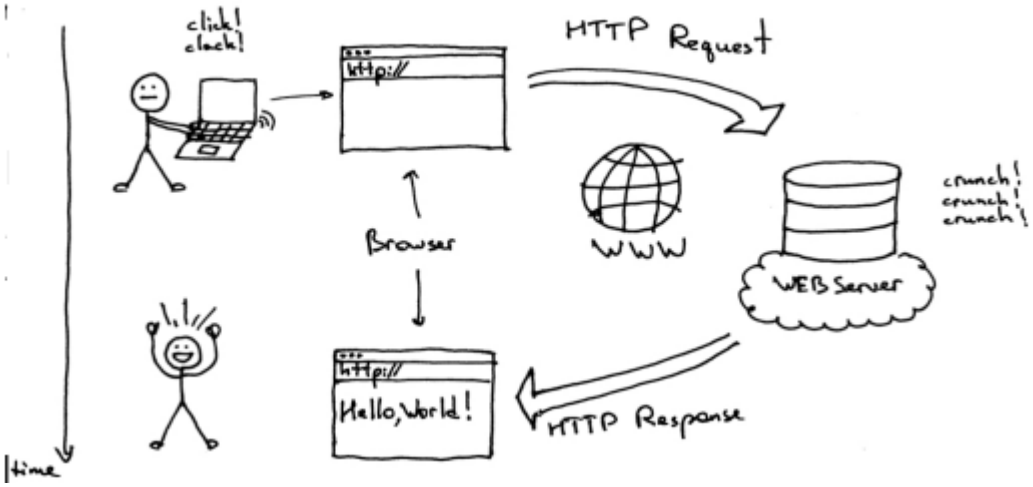
Client Server



Consists of a server and multiple clients

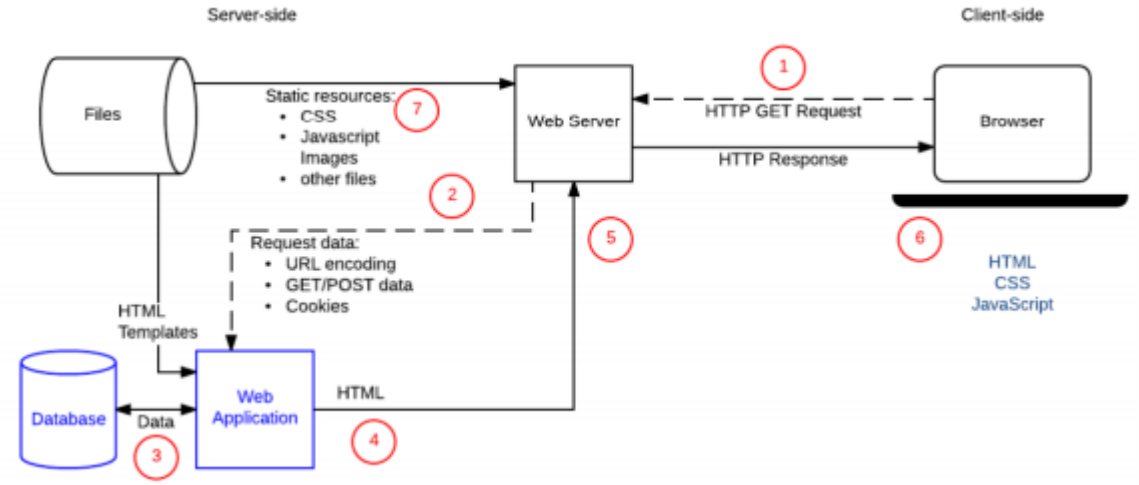
Often used for online services

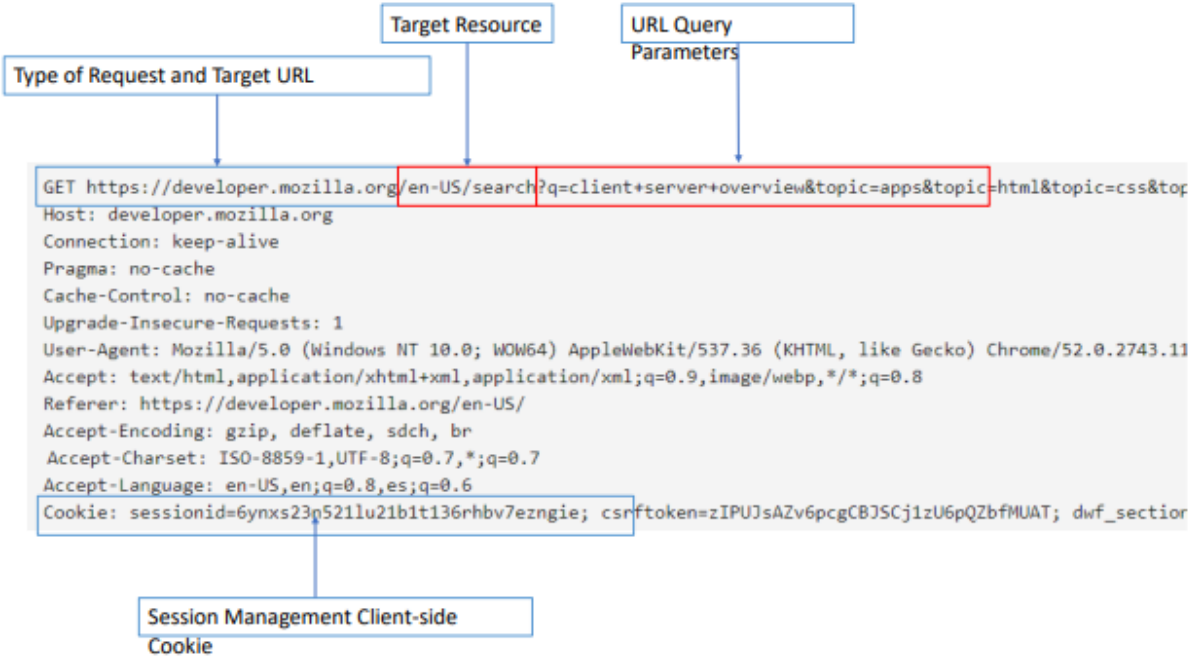
- Client-server model used the most for online services
- Example – you sent request for a web page to browser (client)



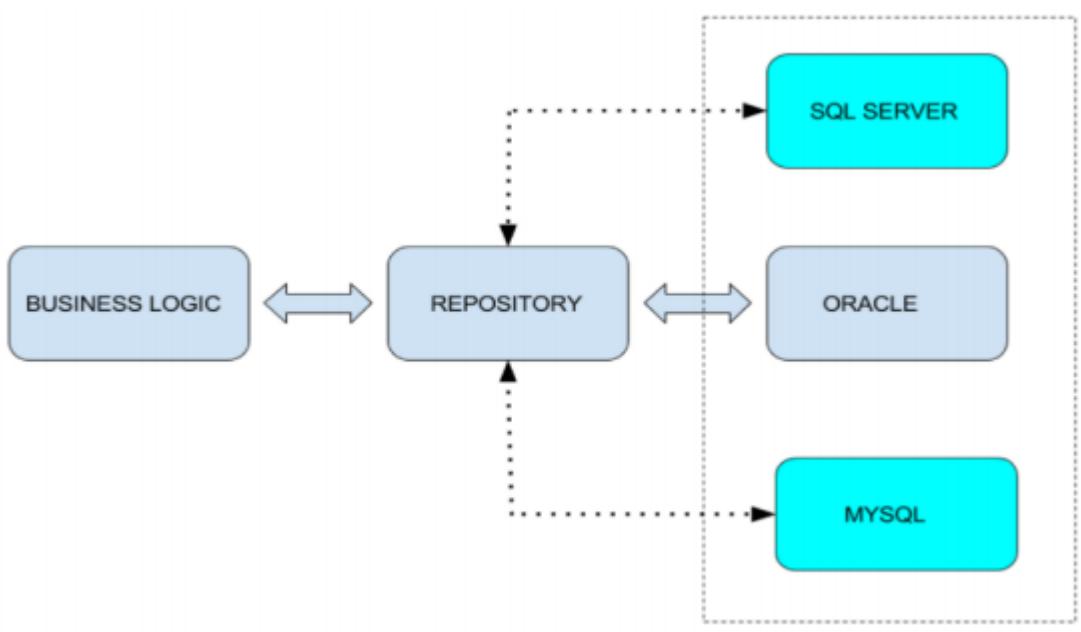
- Browser sends http request to web server
  - Web server is a machine that hosts the web app
- Server processes request and sends a response

Case Study – Web Requests



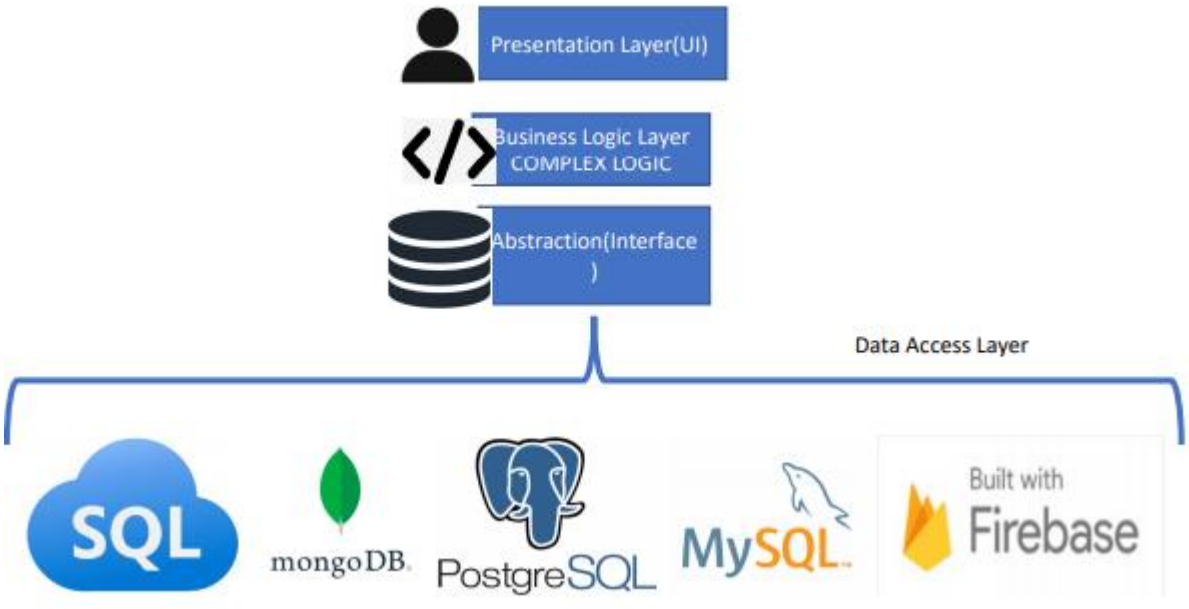


Repository

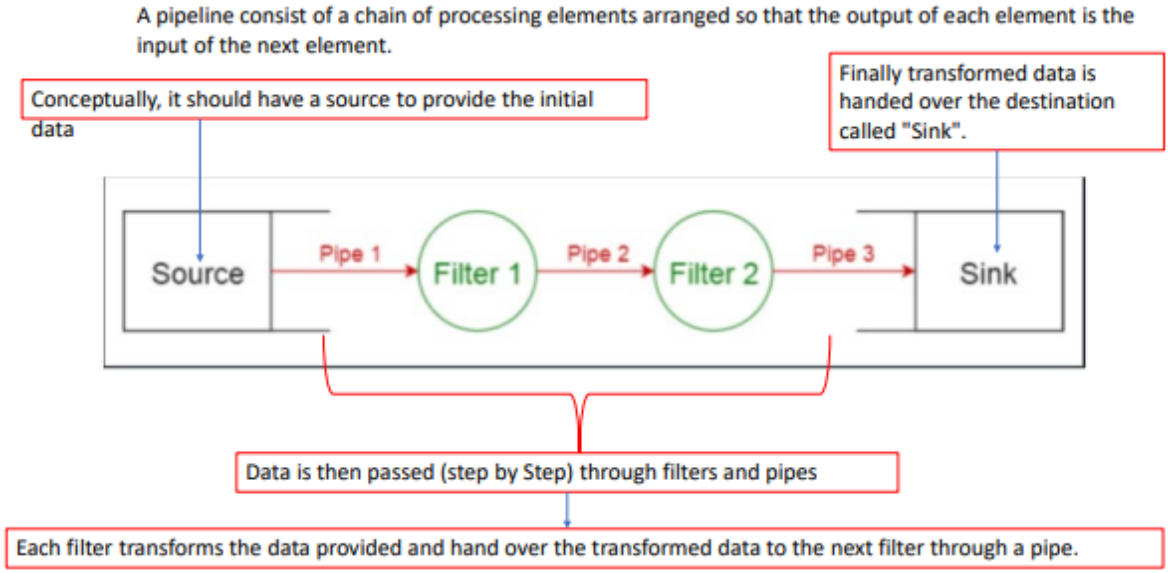


- Loosely couples Business Logic and dtb
  - Data Access Layer – SQL server, Oracle, etc
- Usually, a class that encapsulates dtb logic and handles CRUD operations on behalf of the dtb
  - CRUD = create, read, update, delete
  - Business Logic doesn't know about data access layer – the Repo handles it
- Allows for the abstraction of the dtb provider
- Example:

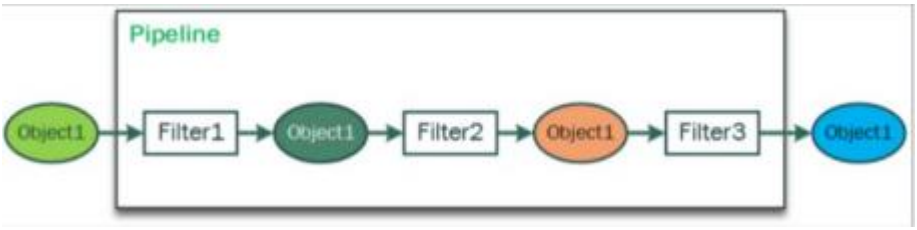
- Should be able to swop out any element in the data access layer – doesn't matter to Business Logic layer
  - Logic layer "speaks" to interface that handles CRUD for it



Pipe and Filter

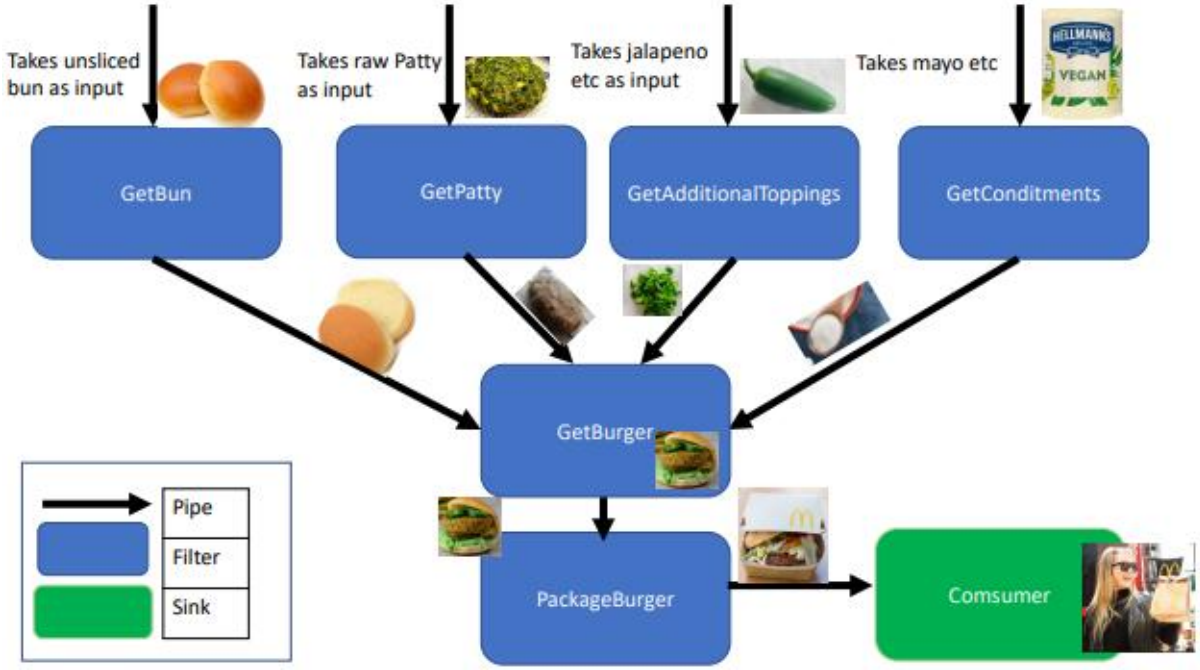


- A pipeline consists of a chain of processing elements arranged so that the output of each element is the input of the next element
- Architecture pattern often used for video streaming
- Pipe and Filter is very reusable



- E.g. have a green object – goes through a series of filters and becomes blue
    - Could not become orange if it wasn't green before, nor blue if it wasn't orange before

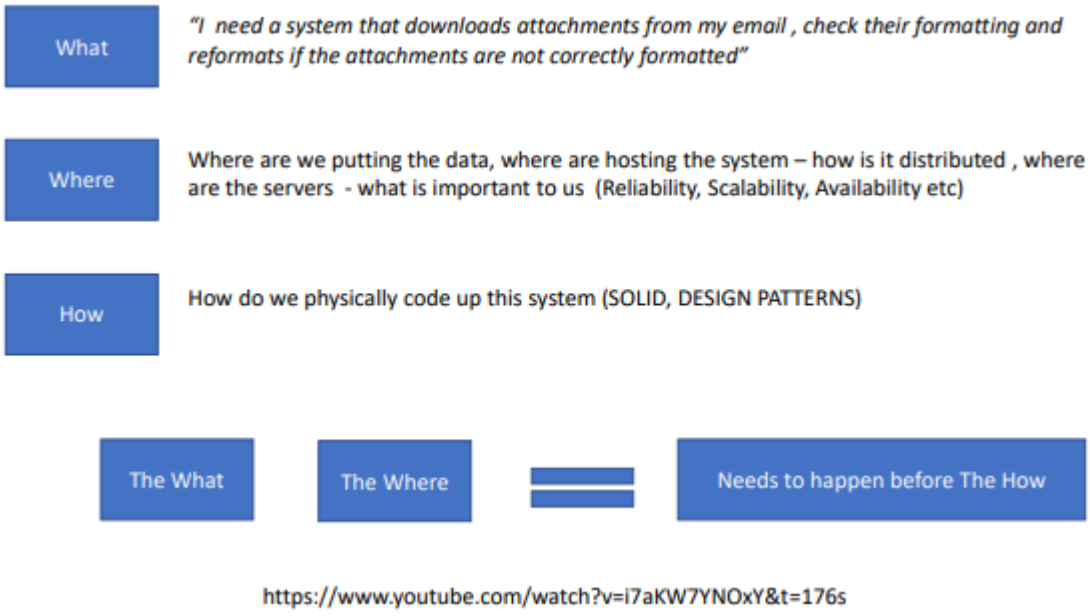
Example



- Want to make a Burger and have products that need to be filtered
    - Have filter that takes unsliced bun and turns it into a sliced bun – sticks it on the pipe
    - A raw patty is cooked and stuck onto the next pipe
    - A jalapeno is chopped, stuck onto a pipe ….and so forth
- These are all sent into the next single filter which assembles the burger
- Then the assembled burger is put onto the pipe and passed to PackageBurger
    - It then gets sent to the "sink" i.e. the consumer

System Design vs System Architecture

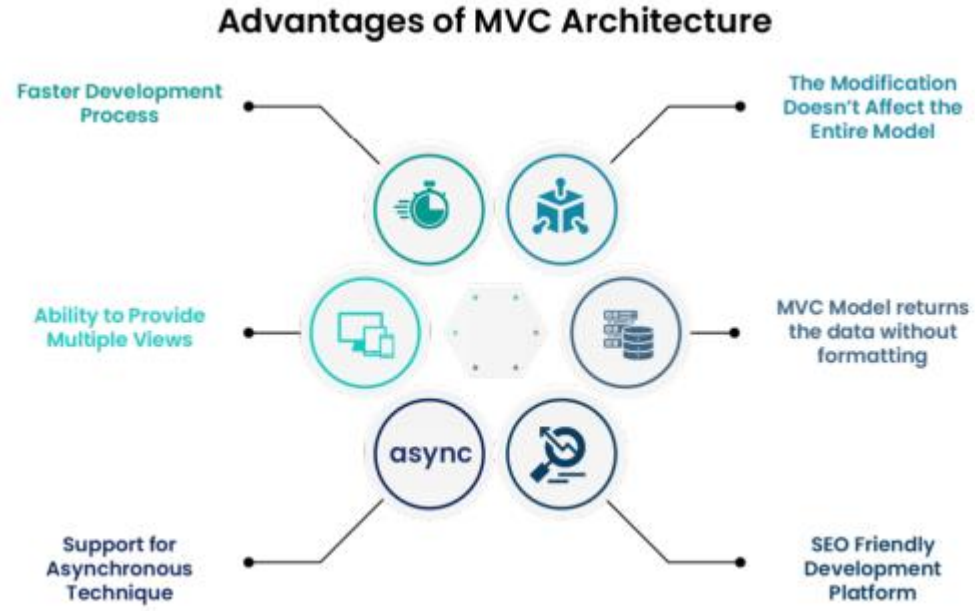
What, Where and How - Example

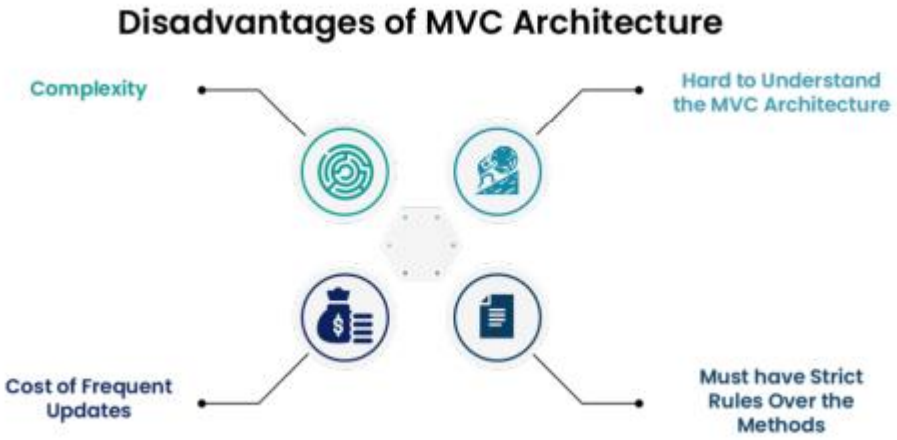


https://www.youtube.com/watch?v=i7aKW7YNOxY&t=176s

- What and the Where *needs* to happen before the How

MVC Architecture

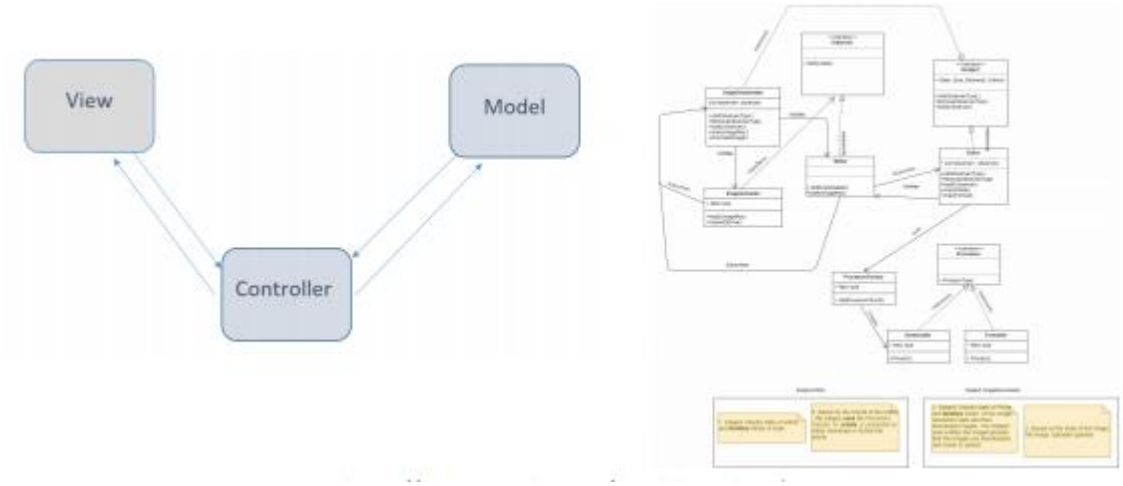




- Have to look at pros and cons of system and see if it meets non-functional requirements
- Software architecture focuses on developing the skeleton and high-level infrastructure of software
    - Big-picture design
    - Non-functional requirements exist in architecture and help make the decision for which architecture to use
- Software design, on the other hand, concentrates on the code level design
    - Class, features designed to be highly maintainable, reusable and reliable

Example

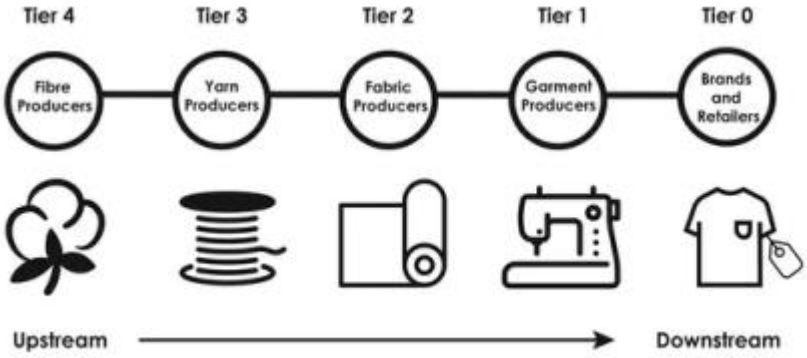
- Reliability == NNB to the client and evs
- With MVC pattern, if one view/model breaks – the others will still work
    - => more robust, reliable system
- Tradeoff with complexity (for reliability)



- Under this architecture – sw is created
- How this sw is coded up is also really important
    - Have to choose the right big picture and code level design

Application Architecture Patterns

- Application systems are intended to meet a business or organizational need
- All businesses have much in common—they need to hire people, issue invoices, keep account
- These commonalities have led to the development of software architectures that describe the structure and organization of particular types of software systems
- Application architectures encapsulate the principal characteristics of a class of systems
- For example, a system for supply chain management can be adapted for different types of suppliers, goods, and contractual arrangements



Uses

Good starting point if you are unfamiliar with architecture patterns

As a starting point for architectural design.

Compare your design with known designs

As a design checklist

Contains stable structures and features that can be designed in parallel

As a way of organising the work of the development team.

Compare the components you want to reuse with generic structures to see whether there are comparable structures

As a means of assessing components for reuse.

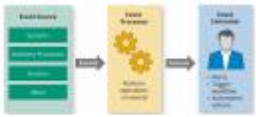
As a vocabulary for talking about application types

If you are discussing a specific application or trying to compare applications of the same types, then you can use the concepts identified in the generic architecture to talk about the applications.

Types

Data processing applications
Data driven applications that process data in batches without explicit user intervention during the processing.
-Billing systems; Payroll systems

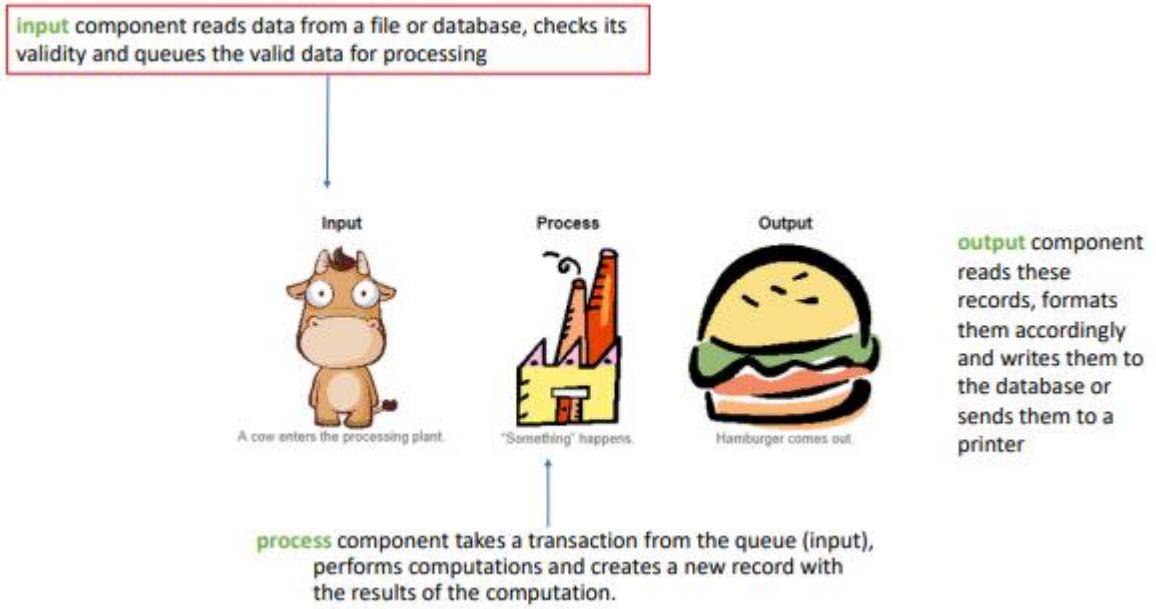
Transaction processing applications
Data-centred applications that process user requests and update information in a system database.
-E-commerce systems; Reservation systems



Event processing systems
Applications where system actions depend on interpreting events from the system's environment.
-Games; Word processors; Real-time systems

Language processing systems
Applications where the users' intentions are specified in a formal language that is processed and interpreted by the system.
-Compilers; Command interpreters

Data Processing Systems



- Largely happens without direct interaction with the user
- E.g. Payroll looks through system, pulls name J Smith
    - Inputs J Smith – PAYE is calculated
    - Then salary is paid to J Smith and salary slip is printed

Transaction Processing System

- Process user requests for information from a database or requests to update the database
- From a user perspective a transaction is: any coherent sequence of operations that satisfies a goal
    - For example - find the times of flights from London to Paris
- Users make asynchronous requests for service which are then processed by a transaction manager
    - E.g. book and pay for flights

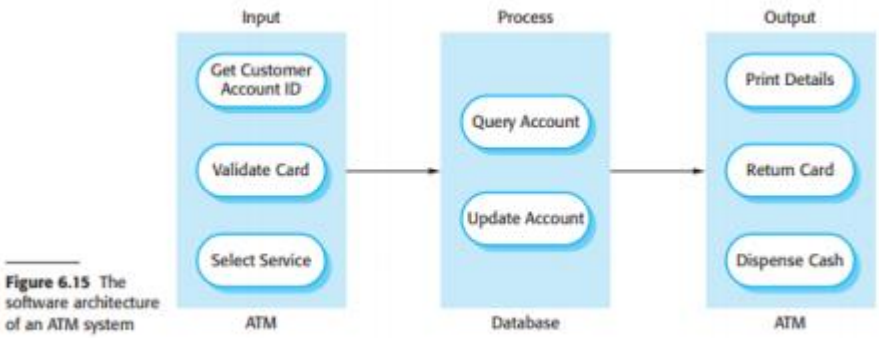
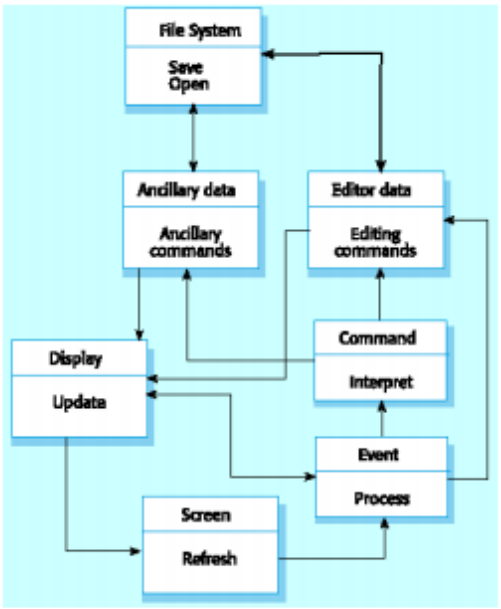


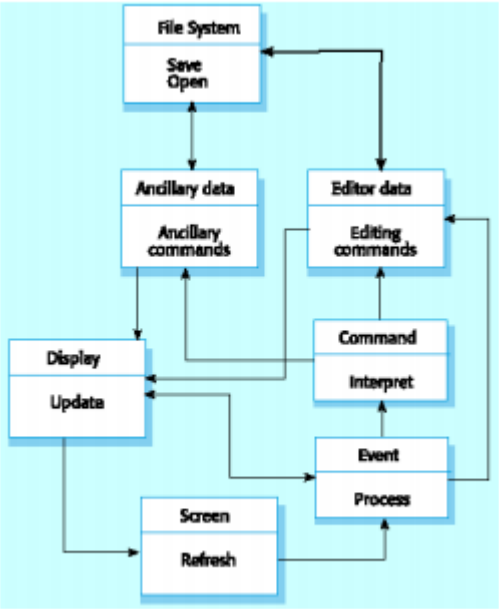
Figure 6.15 The software architecture of an ATM system

- ATMs use this pattern

Event Processing Systems

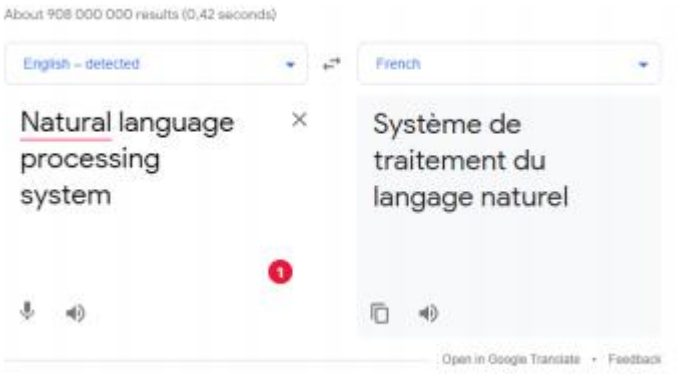


Editing systems are naturally object-oriented:
Screen: monitors screen memory and detects events; (Click and Drag)
Event: recognises events and passes them for processing; (Highlights Text)
Command: executes a user command; (User Selects Format Text to Bold)
Editor data: manages the editor data structure; (Processes Command)
Ancillary data: manages other data such as styles & preferences;
File system: manages file I/O;
Display: updates the screen display(Screen shows Text in Bold)

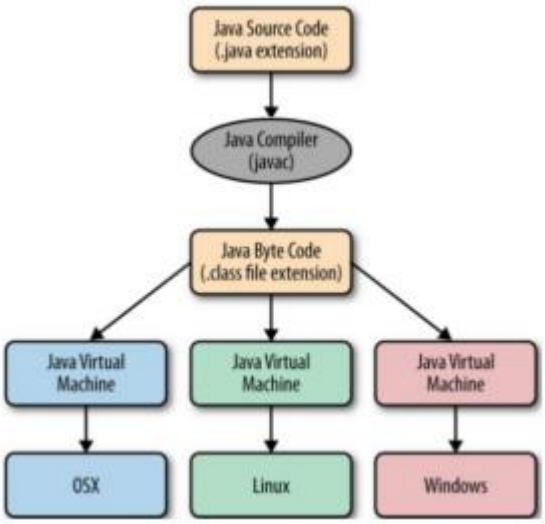


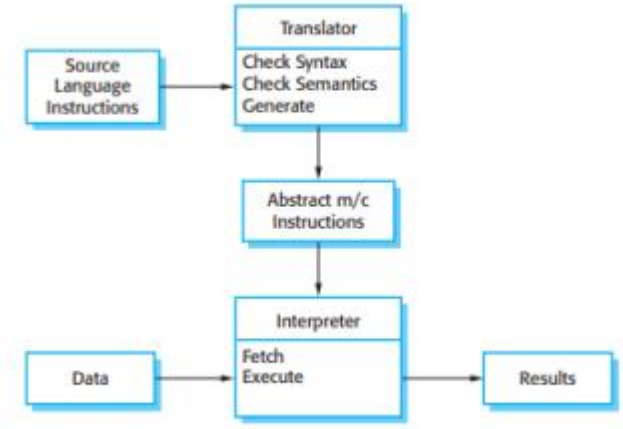
- Screen waits – there's an event
    - Processes the event
    - Interprets the cmd
    - Once interpreted – the output is edited and displayed
    - The screen is refreshed

Language Processing Systems

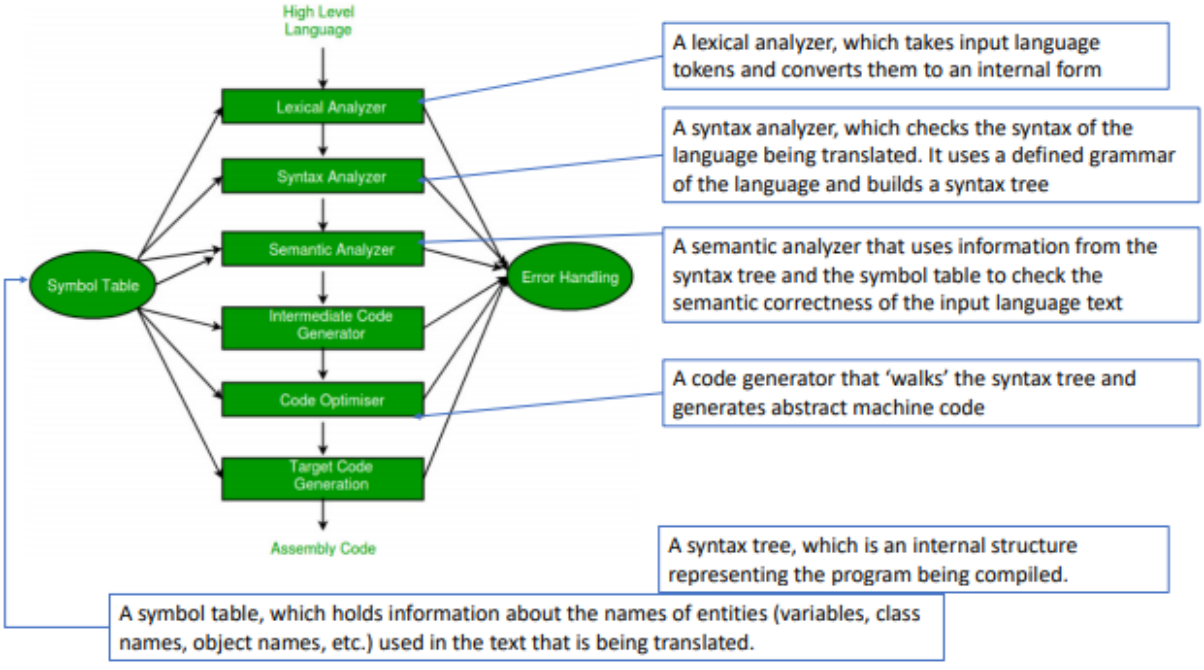


- Used for translation, but better known for compilers

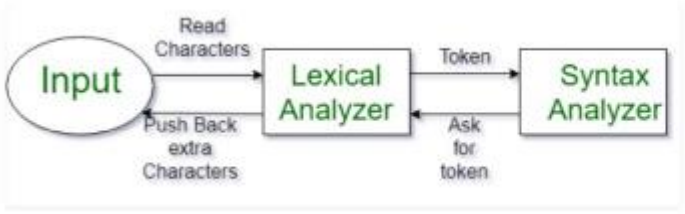

Java Source Code
(.java extension)
Java Compiler
(javac)
Java Byte Code
(.class file extension)
Java Virtual Machine
Java Virtual Machine
Java Virtual Machine
OSX
Linux
Windows


Translator
Source Language Instructions
Check Syntax
Check Semantics
Generate
Abstract m/c Instructions
Interpreter
Data
Fetch
Execute
Results

Example


High Level Language
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Symbol Table
Error Handling
Intermediate Code Generator
Code Optimiser
Target Code Generation
Assembly Code
A lexical analyzer, which takes input language tokens and converts them to an internal form
A syntax analyzer, which checks the syntax of the language being translated. It uses a defined grammar of the language and builds a syntax tree
A semantic analyzer that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text
A code generator that 'walks' the syntax tree and generates abstract machine code
A syntax tree, which is an internal structure representing the program being compiled.
A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.

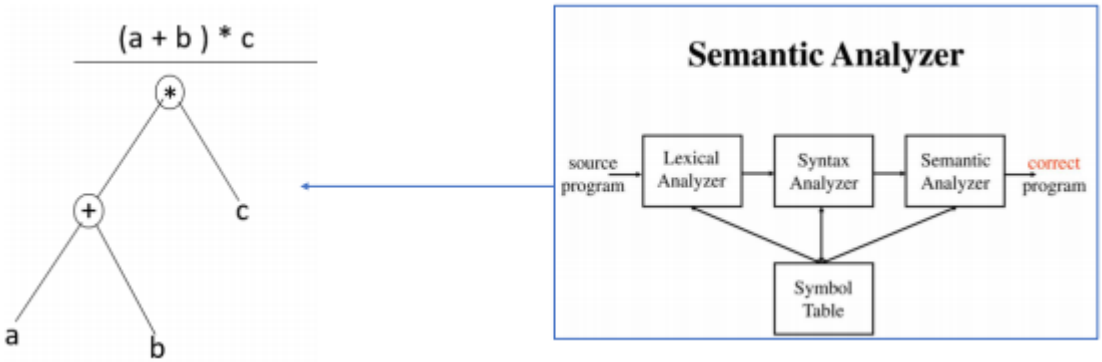
- When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scans the input and divides it into tokens) to target code generation
- Lexical Analysis is the first phase of the compiler also known as a scanner.
    - It converts the High level input program into a sequence of Tokens
- The output is a sequence of tokens that is sent to the parser for syntax analysis
- The output is a sequence of tokens that is sent to the parser for syntax analysis

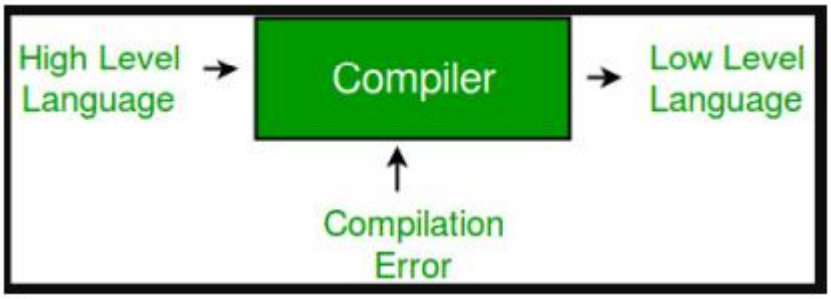


Example of tokens:
- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

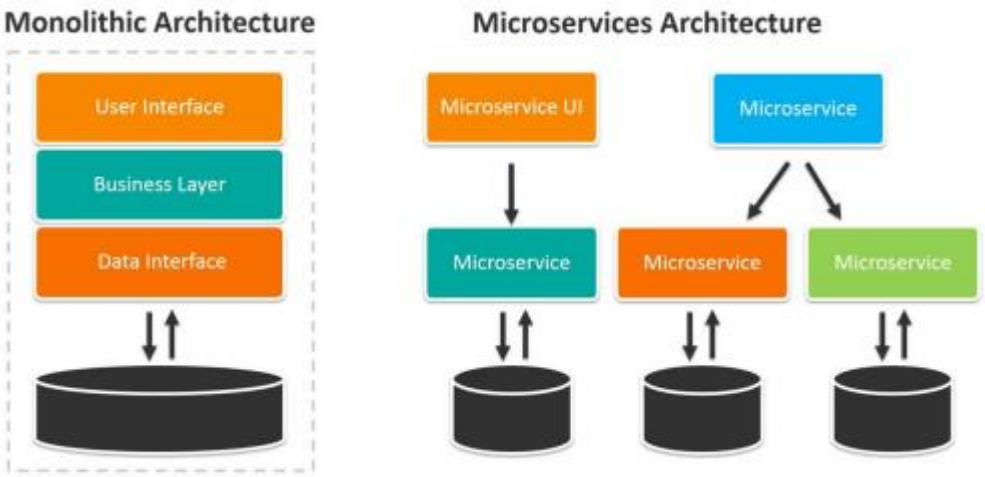
- The lexical analyzer errors with the help of the automation machine and the grammar of the given language on which it is based like C, C++, and gives row number and column number of the error

- 1-5 in above diagram are all tokens
- Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis.
    - It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not.
    - It does so by building a data structure, called a Parse tree or Syntax tree.
- The parse tree is constructed by using the pre-defined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. if not, error is reported by syntax analyzer

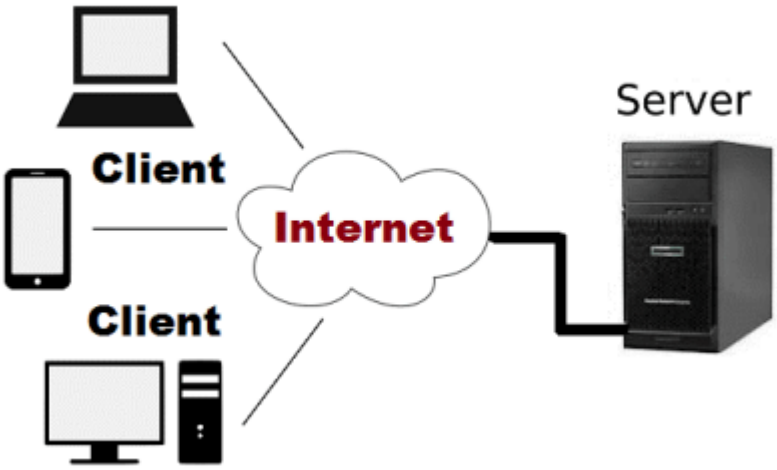



High Level Language
Compiler
Low Level Language
Compilation Error

Service Oriented Architecture



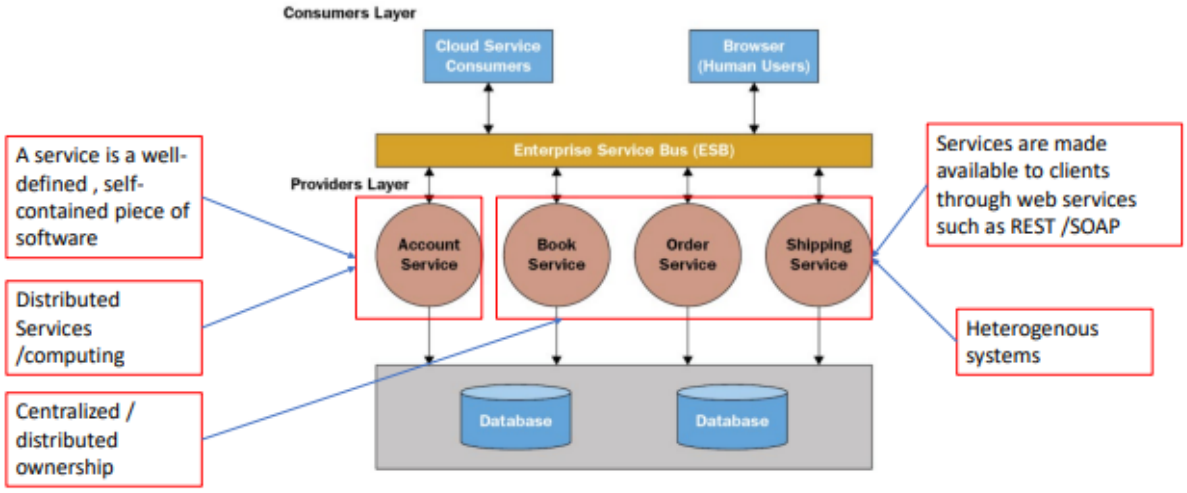
- Monolithic Architectures are coded as single apps – this gives a single Point of Failure
    - Complexity of applications didn't work well or serve industries as they grew
- Take a monolithic architecture and break it down into self-contained, smaller apps called services



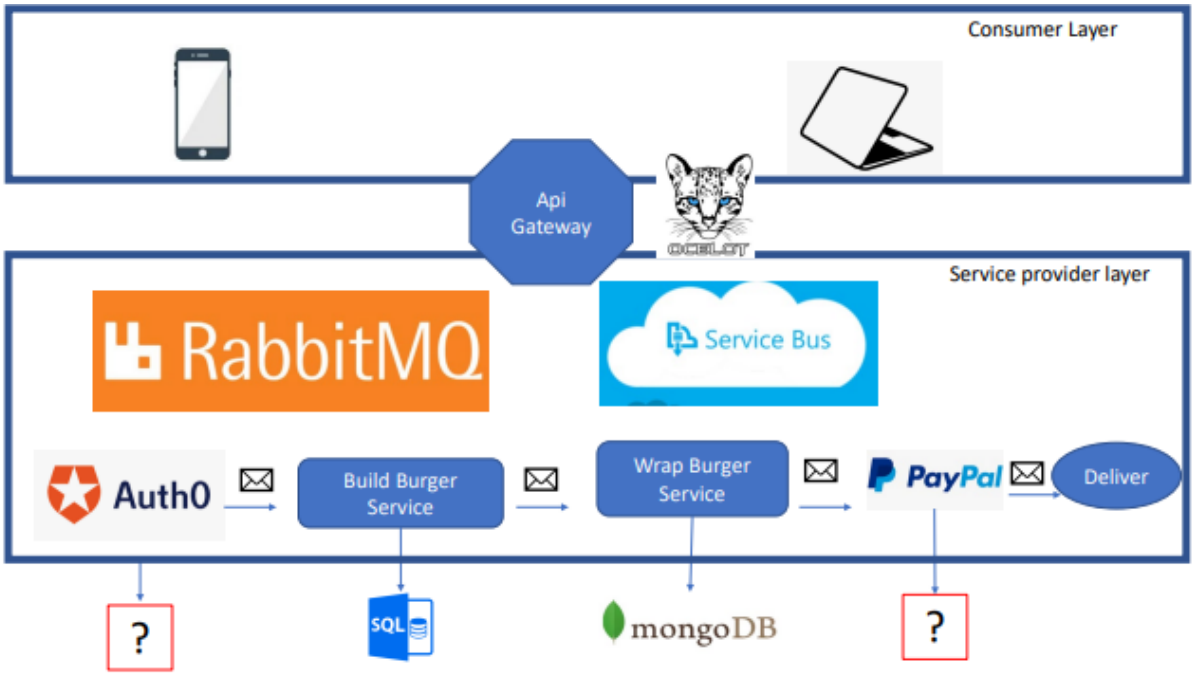
- Monolithic Architecture had server(s) with clients that connected to the servers
- Simple architectures could no longer scale large complex applications
    - Distributed computing was born – break app up and put onto different services
    - Web services allowed for this
    - Web services (Cloud) such as REST and SOAP led to SOA
- Service Oriented Architecture: an approach for designing, developing, deploying and maintaining software systems that are based on distributed resources (Services)
    - Gives Scalability, Flexibility over time
- NB: These services can be controlled by a single owner.. OR NOT… You can make use of predeveloped marketed paid for services as well (centralized or distributed ownership)

- SOA as an architectural pattern focusses on the building of software systems based on loosely coupled service components that can represent business/operational procedures

Example

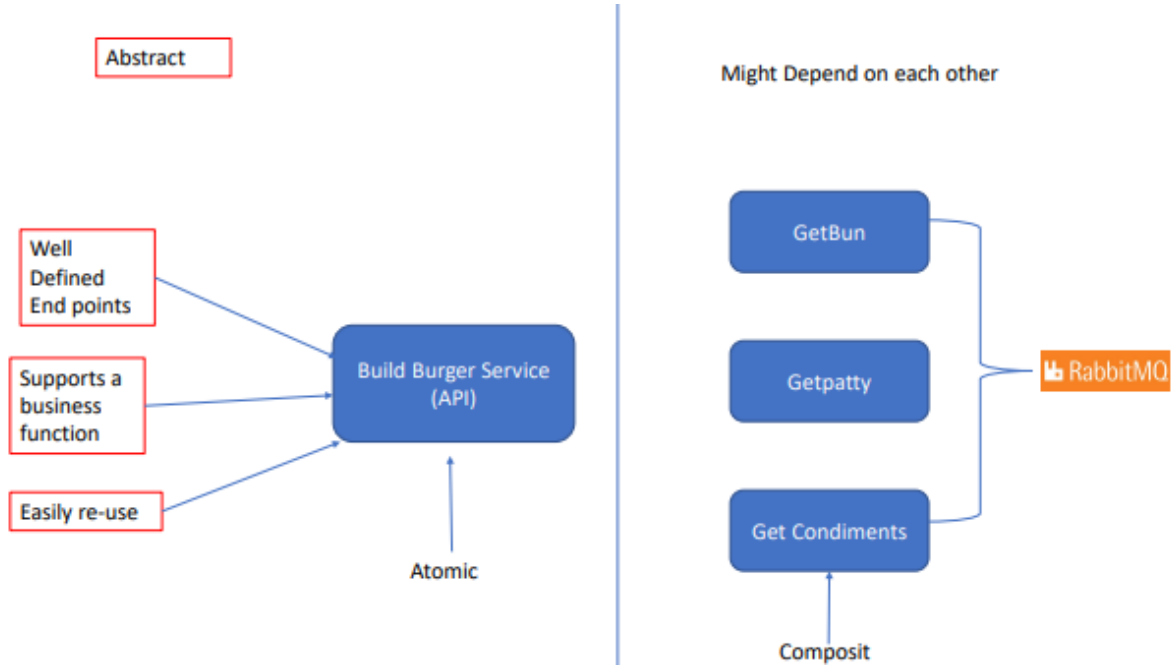


- Service – fully-fledged app that can run on its own, self-contained
    - E.g. Account Service
    - Create different services – they provide services to the big, overall app
    - If one service dies, everything else still lives
    - Allows one to add/drop services as one needs them
- Heterogenous Systems:
    - Heterogeneity can be seen in below diagram in the fact that two different databases are being used but the client is not aware of it
        - Build Burger Service speaks to SQL dtb instance whilst the Wrap Burger Service speaks to a mongoDB instance
    - E.g. purchase PayPal Service because don't have devs to handle payment authentication and transactions

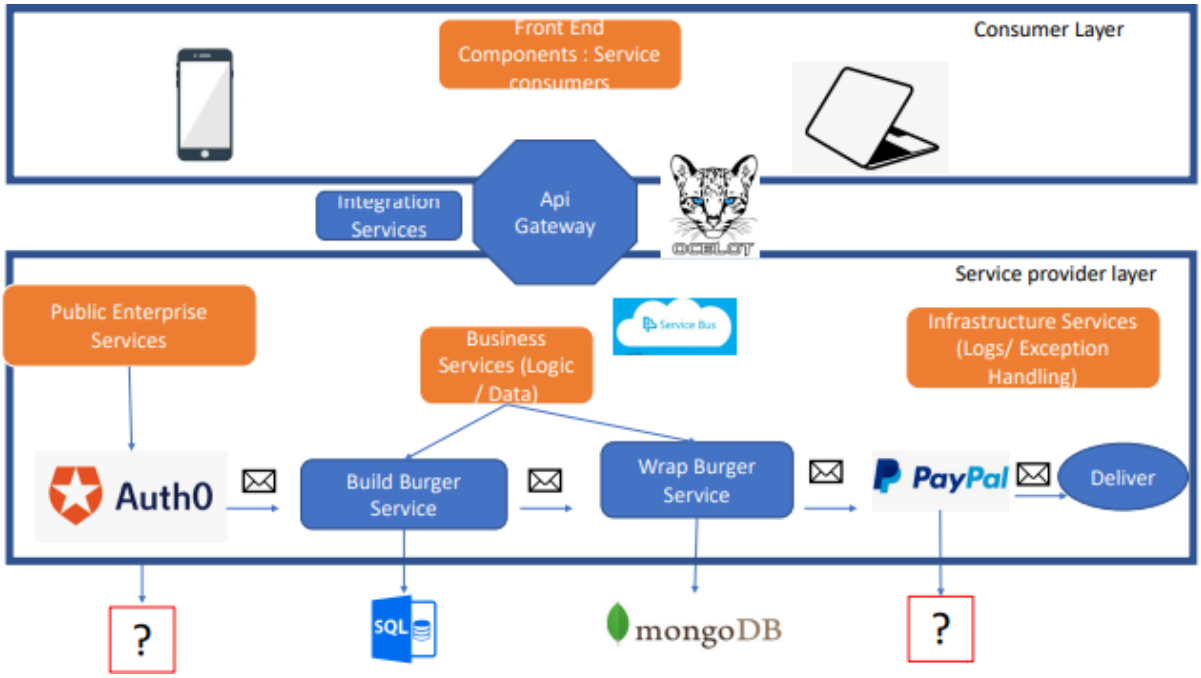


- Services need to be able to talk to each other
  - E.g. PayPayl needs to tell delivery that payment was successfully made
  - All of this is largely handled by e.g. Enterprise service bus
- Client needs to be able to connect to multiple services – thus an API gateway is necessary
  - E.g. client needs to log-on to Burger-API gateway
    - Gateway knows on the app that log-in has been clicked
  - Check that you're authenticated – sends yes to Build Burger Service
  - And so the process continues until you receive your burger

Services

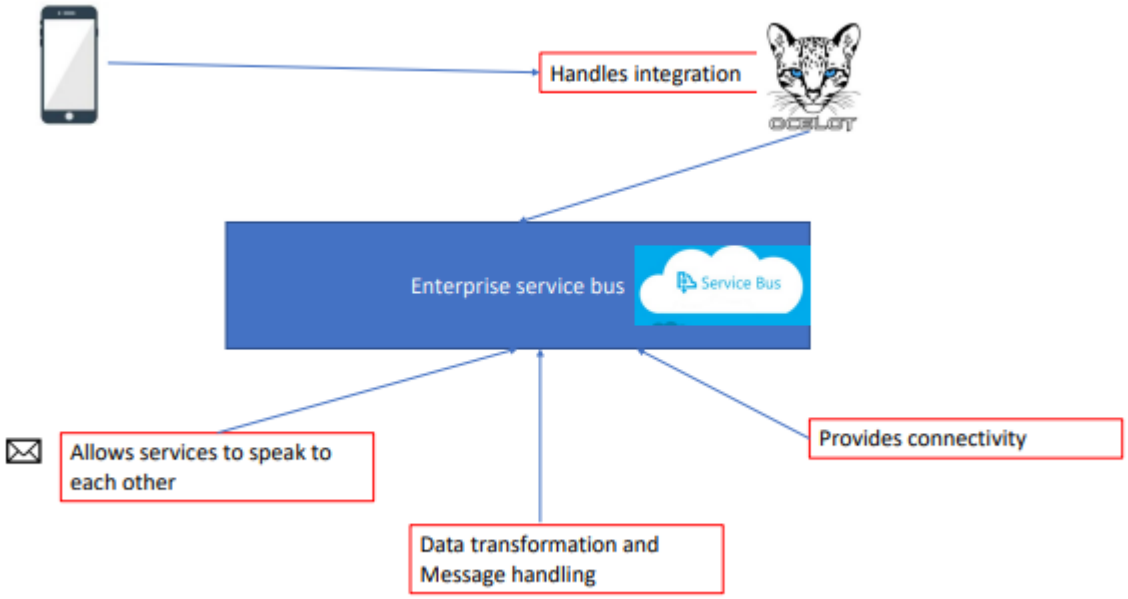


- Services are abstract – consumer of the service never sees the services logics
- Well defined end point – whole apps by themselves
- E.g. if create a good burger service, could then be used by multiple burger vendors
- Atomic services – just does one thing
- Composite – take burger service and have smaller services combined to deliver services
    - Combo of atomic services – need to be able to talk to each other through a msging queue



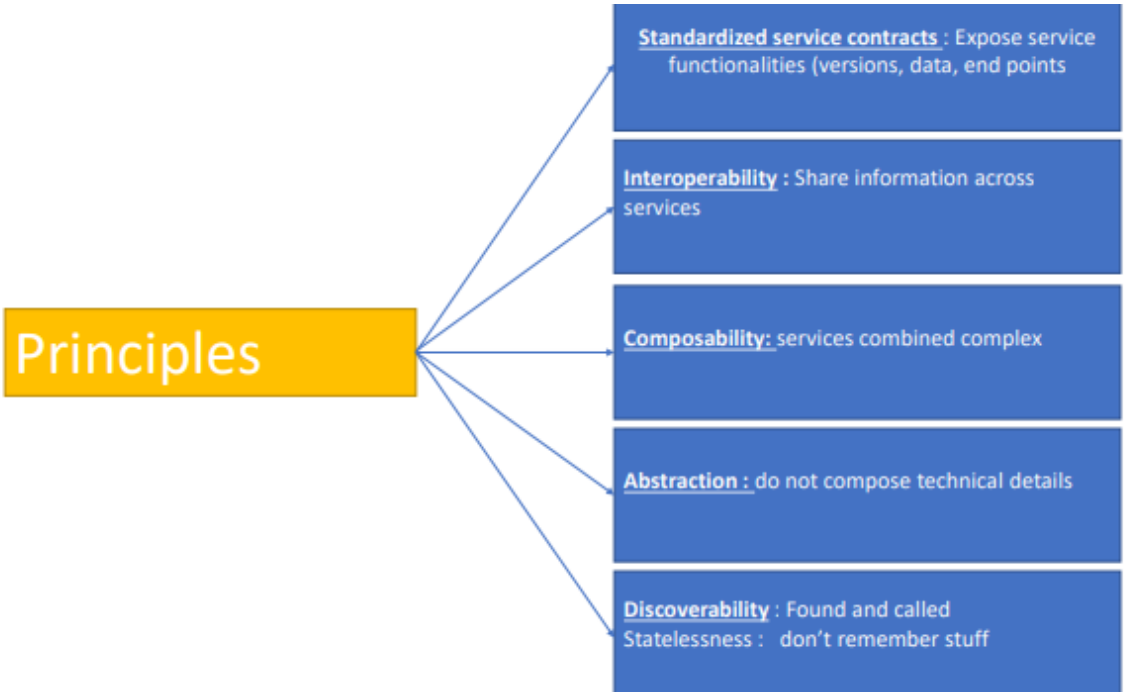
- Components are normally divided into different kinds of services
- Front-end components services:
- Integration services API gateway
- Business logic services – pull data from dtb, use business logic to package it like how it needs to be and gets it ready to be displayed to client
- Public/Enterprise services – this service can be used throughout entire enterprise or by others
    - E.g. authentication
- Infrastructure Services – handles logging and exception handling
    - A single service that handles all this for all the other services

Enterprise Service Bus



- Assists in making the services interoperable and allows them to talk to each other
    - Ensures that msg sent from one service is readable by another service – if data is sent with that msg, ensures data is sent in the correct form

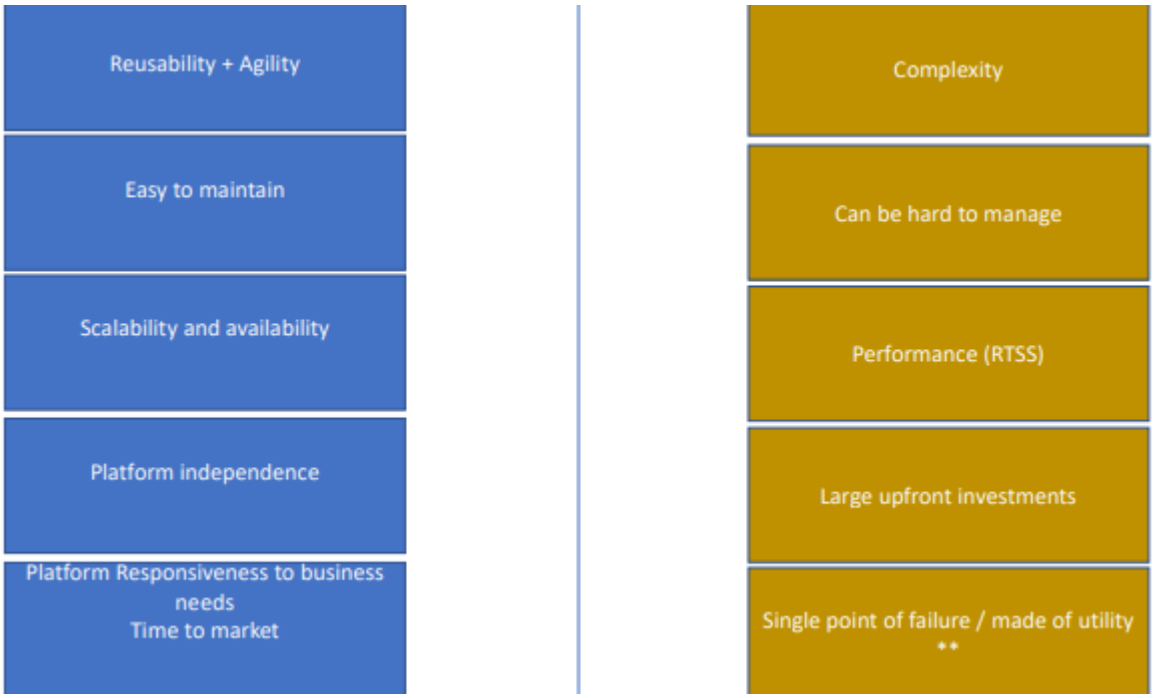
SOA Principles



- Rules for breaking down a monolith into small, self-contained services
- Standardized service contracts : Expose service functionalities (versions, data, end points
    - E.g. OpenAPI – API is the rule i.e. what version being used, what data will be delivered etc

    - Services are often delivered as APIS – services need an endpoint
        - E.g. www.cheeseburger.com/**wrapburger** - bold is an endpoint
            - i.e. where you end up with that service and are able to accomplish what is needed
- Interoperability: Share information across services
    - Use Enterprise Service Bus and msging queues
- Composability: services combined complex
    - Atomic and composite services
- Abstraction: do not compose technical details
    - E.g. client only sees the whole app – not all the complexity
    - One service doesn't know what the other does – just passes the msg onto it
- Discoverability: Found and called
- Statelessness: don't remember stuff

Advantages and Limitations



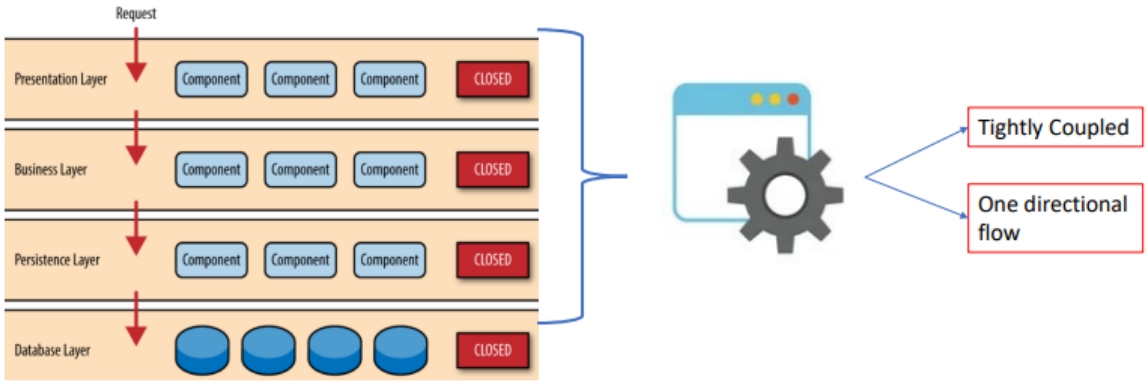
- Advantages:
    - Reusability + Agility:
        - E.g. can easily add another service
    - Easy to maintain
    - Scalability and availability
        - Can grow app
        - E.g. if burger site goes down, can still order drinks, etc
    - Platform independence
    - Platform Responsiveness to business needs
    - Time to market
- Limitations:
    - Complexity
    - Can be hard to manage

- Performance (RTSS)
- Large upfront investments
- Single point of failure / made of utility

Microservices Architecture

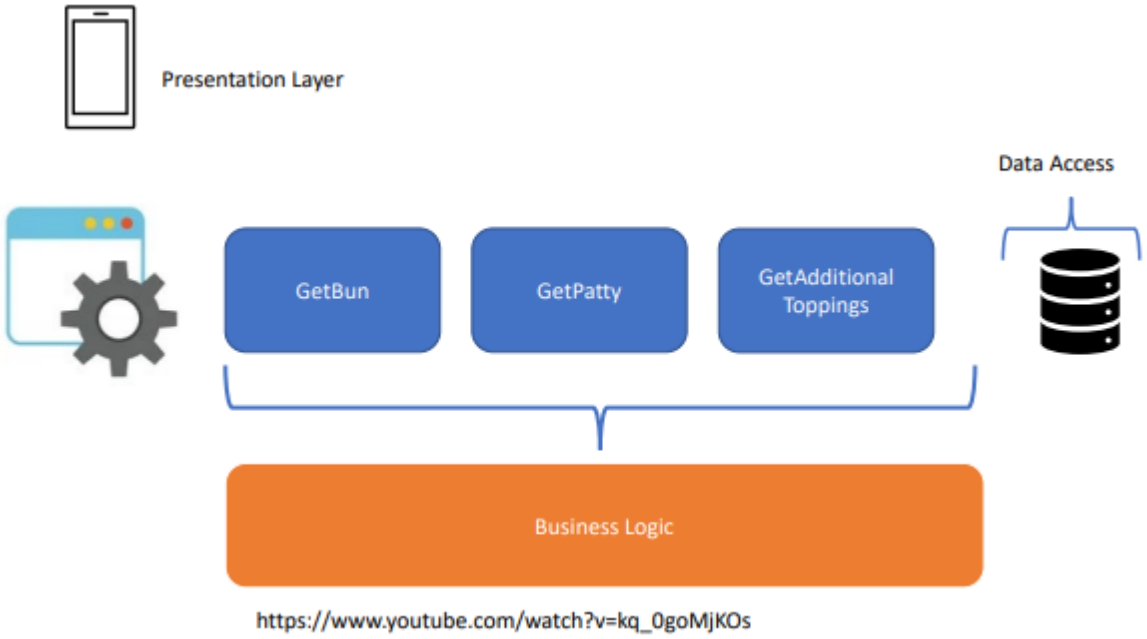
REST

- REST API call – client app connects to REST API
  - Does so by sending HTTP request which could be GET, POST, PUT, DELETE
  - REST API talks to rest server and will return whatever info client wants – returned as JSON usually, sometimes XML



MicroServices

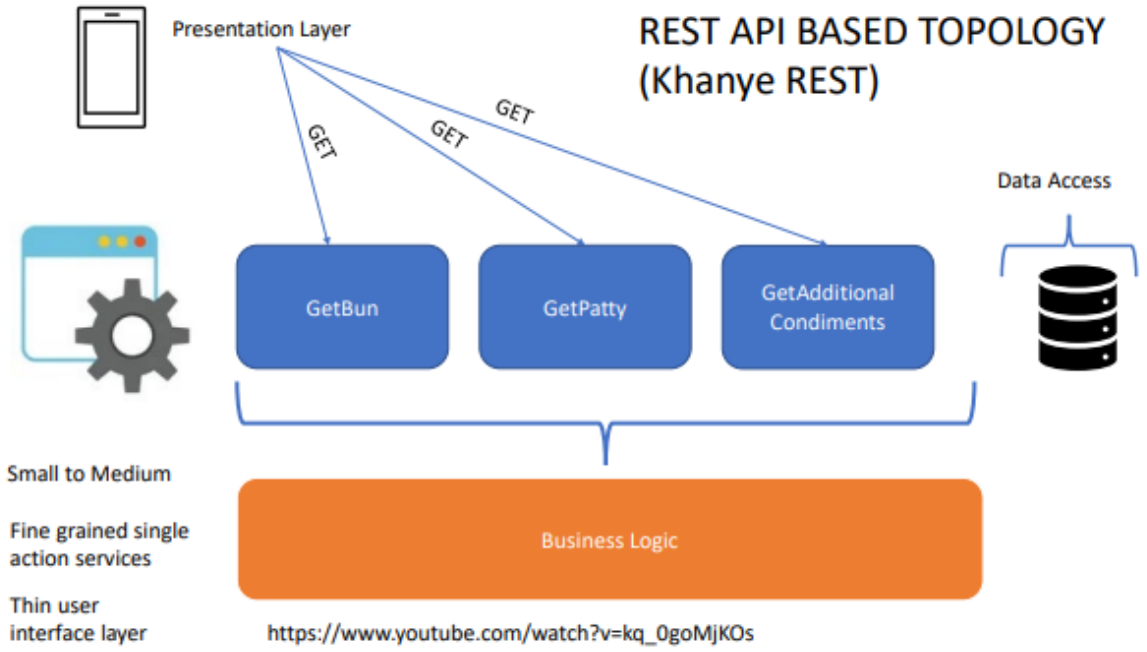
- Above diagram shows tight coupling and if one layer breaks, everything breaks – not good for scalability
- Microservices are *not* good for high performance systems
- With microservices one consider the app and breaks it down



- Take layered architecture and break it up
    - Have presentation layer and services – services are more scaled down than SOA because the single responsibility principle is applied

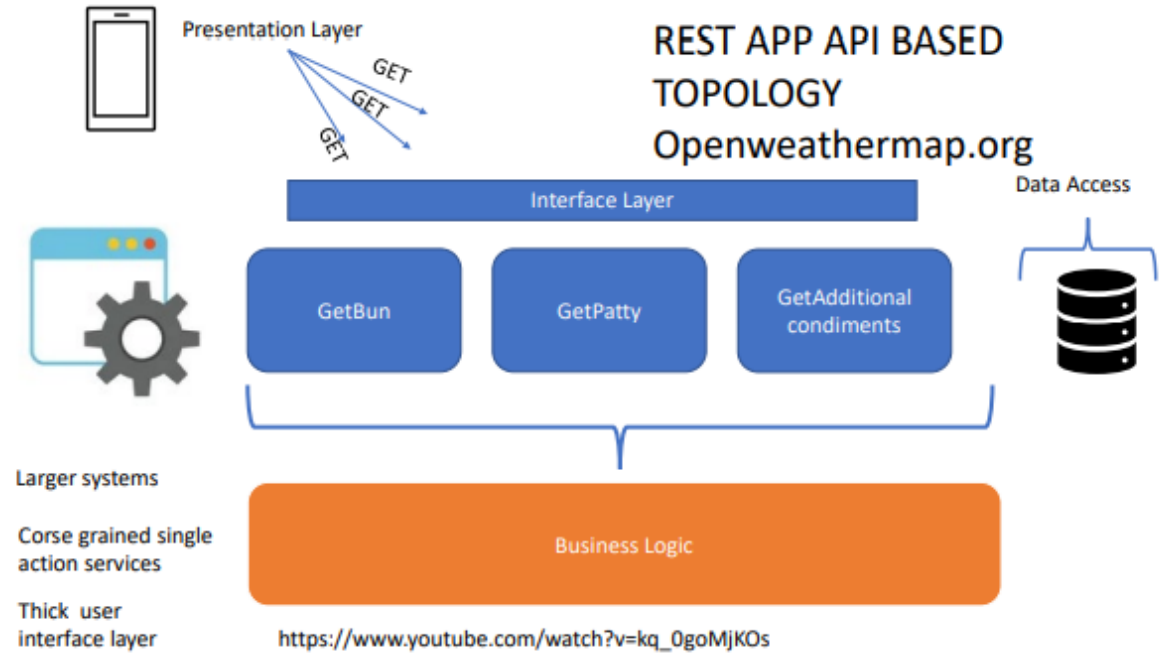
Microservices Topology Approaches

REST API Based Technology



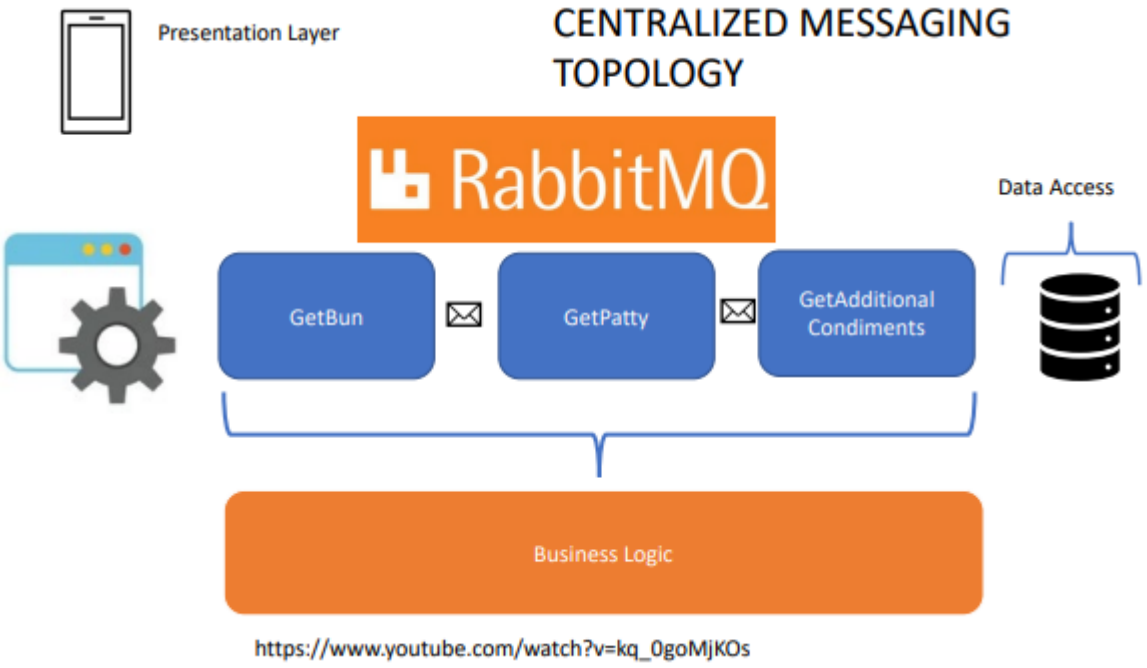
- Small to medium institutions
- Fine grained single action services
- Thin user interface layer

REST App API Based Technology



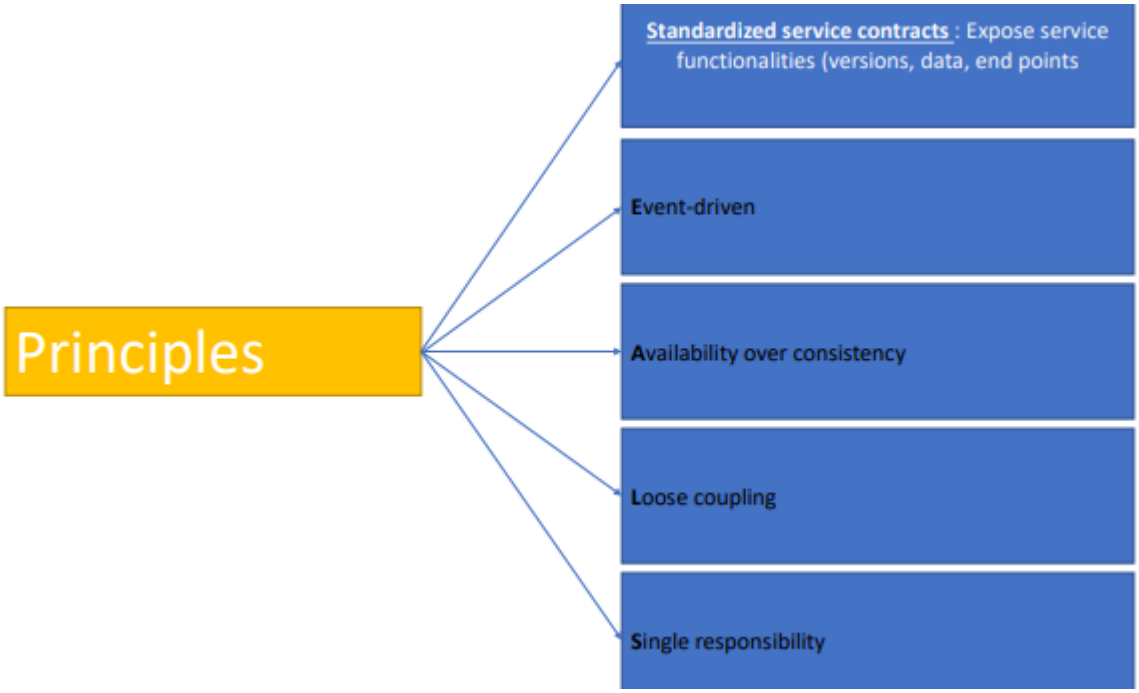
- Larger systems
- Corse grained single action services
- Thick user interface layer

Centralised Messaging Topology

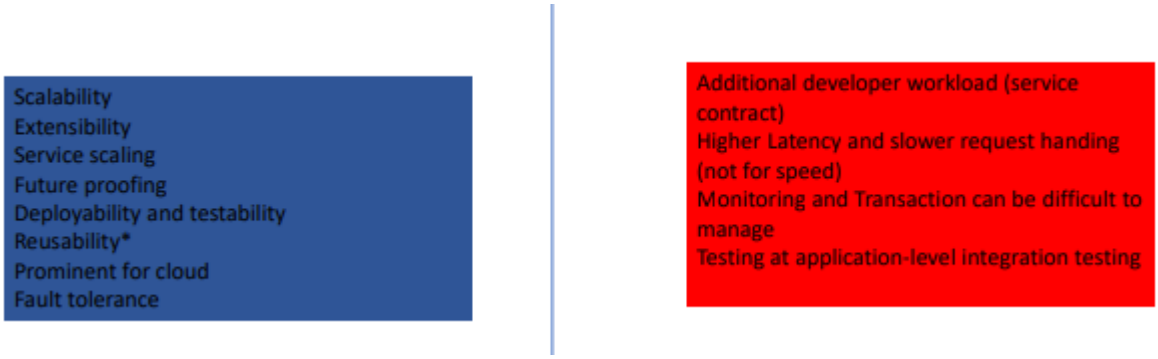


- Used a lot in the banking sector
- Messaging queue in the middle and services communicate with each other

Principles



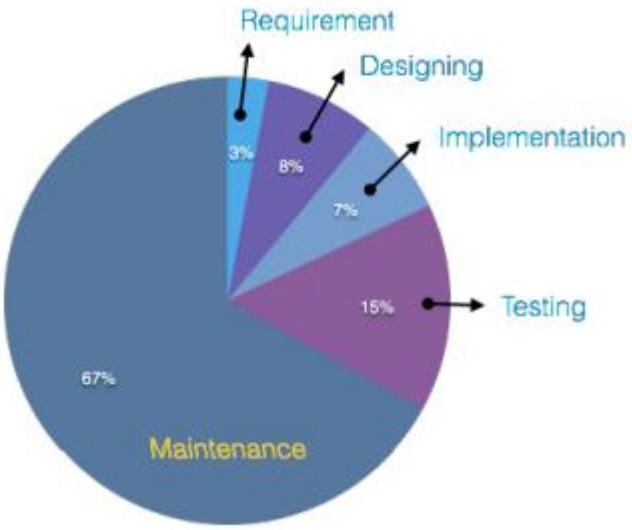
Advantages and Disadvantages



- Advantages:
  - Scalability
  - Extensibility
  - Service scaling
    - E.g. if the Bun service is not getting used but the Patty Service is being used more – can give more resources to the patty service (only – don't have to give more resources to all services)
  - Future proofing – can add services as they're needed
  - Deployability and testability
  - Reusability* - not what the aim of microservices is though
  - Prominent for cloud
  - Fault tolerance
- Limitations:
  - Additional developer workload (service contract)
    - E.g. takes longer to setup
  - Higher Latency and slower request handing (not for speed)
    - Don't use if performance is an important factor

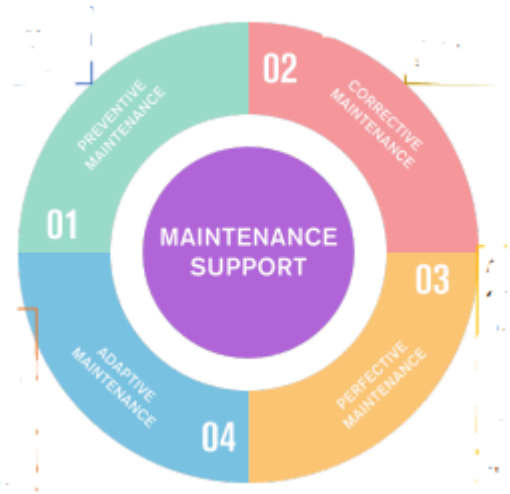
- Monitoring and Transaction can be difficult to manage
- Testing at application -level integration testing

Software Maintenance



- Software maintenance is the process of changing, modifying, and updating software to keep up with customer needs
- Software maintenance takes place after the product has launched for several reasons including improving the software overall, correcting issues or bugs, to boost performance
- Software maintenance is a crucial part of remaining competitive and relevant

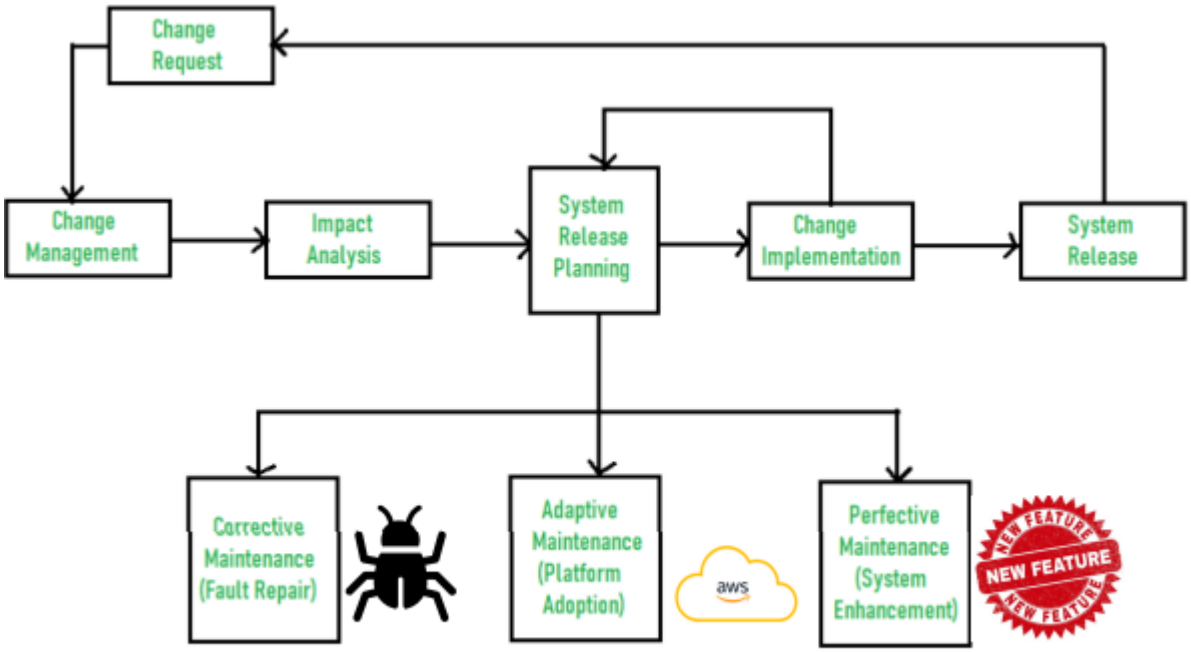
Types of Software Maintenance



- Preventative software maintenance is looking into the future so that your software can keep working as desired for as long as possible
    - This includes making necessary changes, upgrades, adaptations and more
- Corrective software maintenance is the typical, classic form
    - Maintenance is necessary when something goes wrong in a piece of software including faults and errors

- As with any product on the market, once the software is released to the public, new issues and ideas come to the surface
    - Users may see the need for new features or requirements that they would like to see in the software to make it the best tool available for their needs
- Adaptive software maintenance has to do with the changing technologies as well as policies and rules regarding your software
    - These include operating system changes, cloud storage, hardware, etc

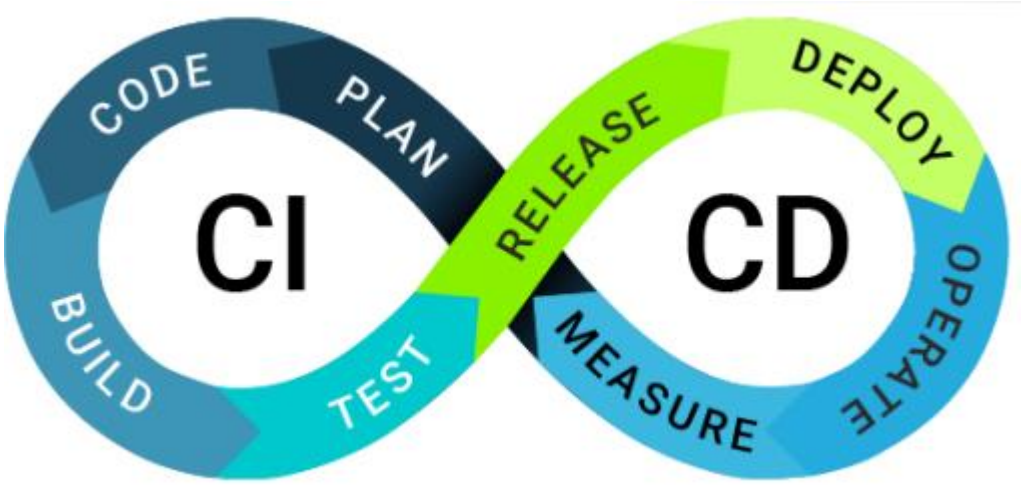
The Software Maintenance Process



MAINTENANCE PROCESS

- Starts with a Change Request – goes to Change Management
- Change Management conducts an Impact Analysis – greatest constraints usually are cost and time
- If feature deemed possible – do System Release Planning
    - Decide what kind of step – is it corrective maintenance, adaptive maintenance, or perfective?
- Implement the change and release it

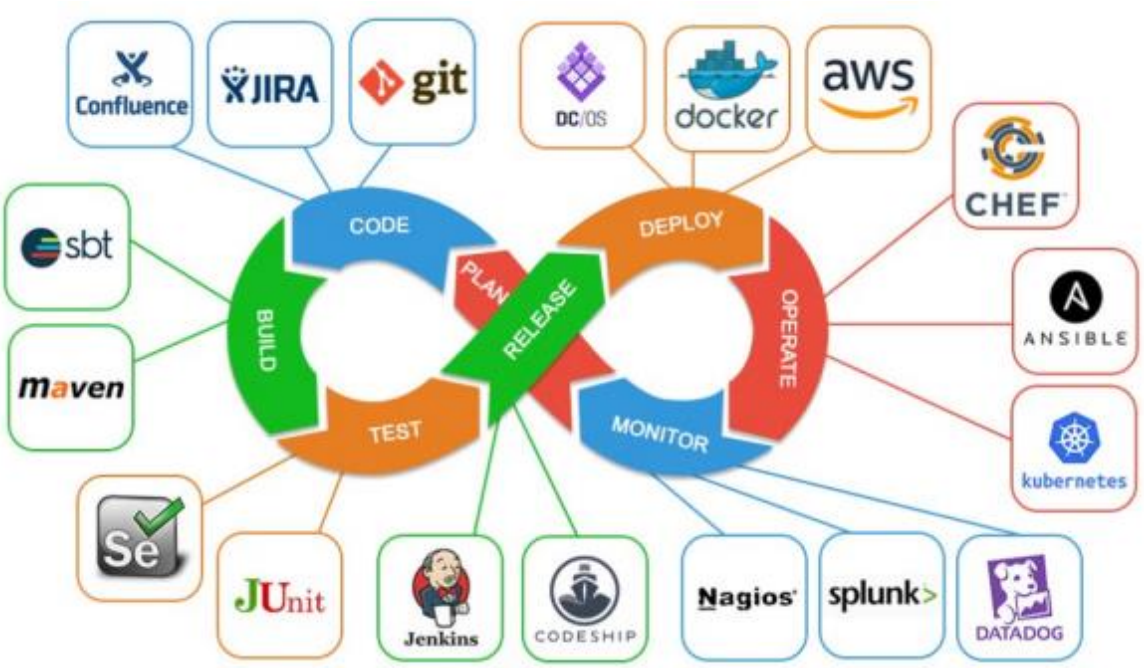
Continuous Integration / Continuous Deployment



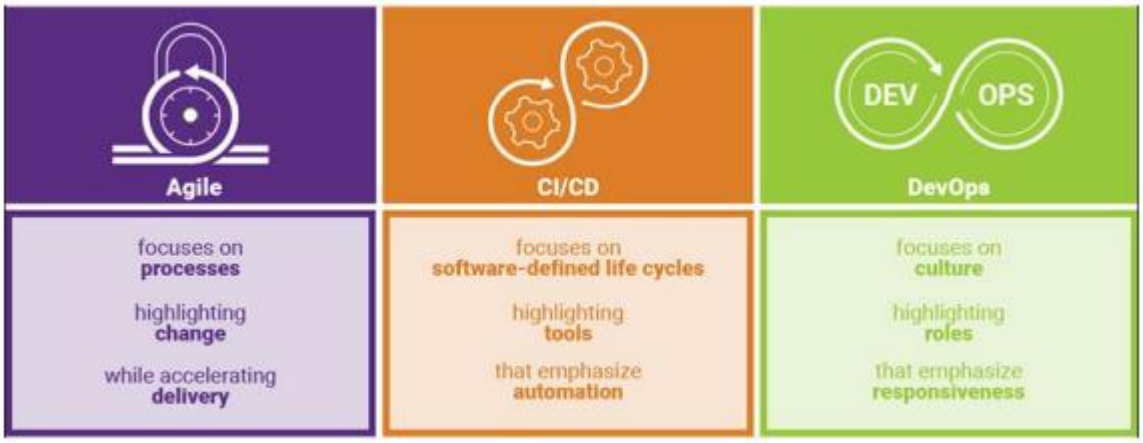
- The Plan phase often combines practices from Scrum and Agile to enable frequent micro-incremental releases.
- The Code phase focuses on core development tasks from within IDEs and appropriate sandboxing and frameworks
- The Build phase rapidly and incrementally merges code commits with some testing and security validation
- The Test phase focuses on automated verification of enhancements, often incorporating test-driven deployment practices
- The Release phase is centered around repository commits and adequate documentation of the changes
- The Deploy phase is the actual update to the codebase, with special thought given to issue and error avoidance
- The Operate phase occurs once the code is made live, and consists of monitoring and orchestration
- The Monitor & Optimize(Measure) phase takes place parallel to the Operate phase and consists of data collection, analysis, and feedback to the start of the pipeline and to other phases as needed

DevOps

- Devs – build, code, plan test
- Ops – a separate team to devs
- Now have DevOps engineers- involved in all the phases (not just dev or ops)
    - Allows for faster deployment and shorter time to market

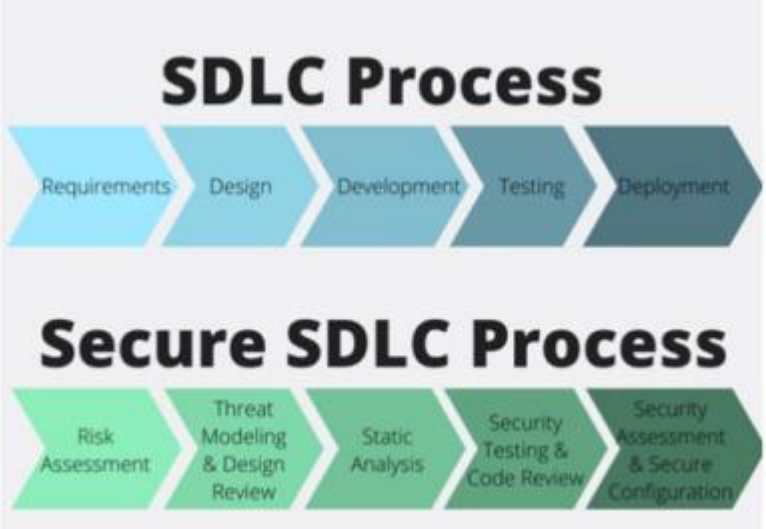


- DevOps involves a lot of tools at each step of the way


Agile
focuses on processes
highlighting change
while accelerating delivery
CI/CD
focuses on software-defined life cycles
highlighting tools
that emphasize automation
DEV
OPS
DevOps
focuses on culture
highlighting roles
that emphasize responsiveness

Software Security

Secure Software Development Lifecycle



- Security is an important part of any application that encompasses critical functionality
    - Security is a functional requirement
- Security applies at every phase of the software development life cycle (SDLC) and needs to be at the forefront of your developers' minds as they implement your software's requirements.
- Secure SDLC is a collection of best practices focused on adding security to the standard SDLC
- With dedicated effort, security issues can be addressed in the SDLC pipeline well before deployment to production
- Implementing SDLC security affects every phase of the software development process
    - It requires a mindset that is focused on secure delivery, raising issues in the requirements and development phases as they are discovered.
    - This is far more efficient—and much cheaper—than waiting for these security issues to manifest in the deployed application
    - Secure software development life cycle processes incorporate security as a component of every phase of the SDLC

- Risk Assessment – identifying risks from requirements
    - E.g. Login = access to only allowed data
- Threat Modelling and Design Review – functional requirements state what should happen, thread modelling state what should NOT happen
- Static Analysis – static security testing is done on the application – checks for secure coding guidelines
- Security Testing and Code Review – automated Security testing
- Security Asessment and Secure Configuration – continuously patching code and looking for zero days

Defensive Development

10 Best Coding Practices

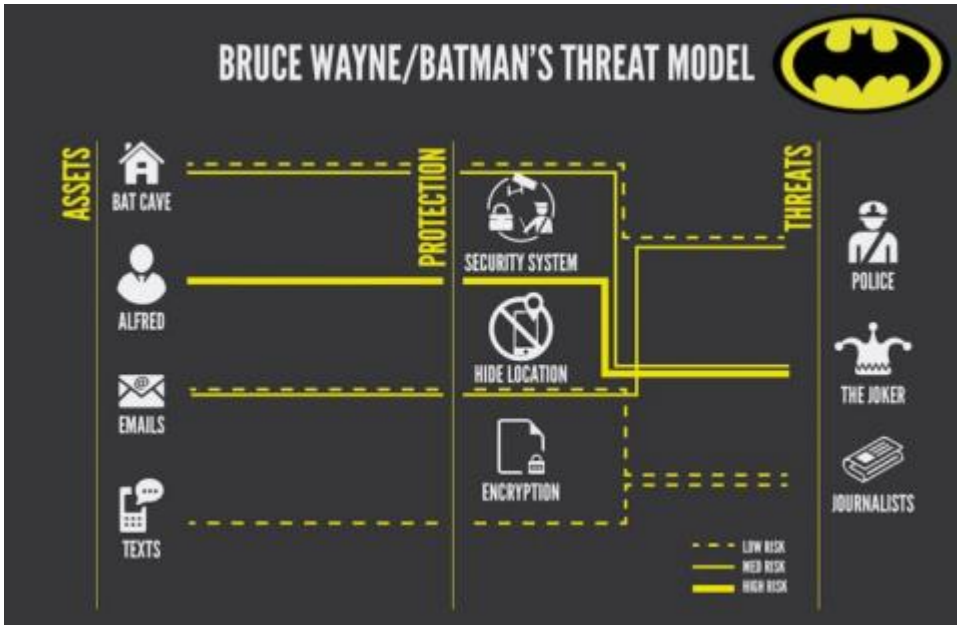
- Defensive programming is when a programmer anticipates problems and writes code to deal with them.
- Validate input: Validate input from all untrusted data sources
- Heed compiler warnings: Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code
- Architect and design for security policies: Create a software architecture and design your software to implement and enforce security policies.
    - For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems
- Keep it simple: Keep the design as simple and small as possible
    - Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use.
- Default deny: Base access decisions on permission rather than exclusion.
    - This means that, by default, access is denied
- Adhere to the principle of least privilege: Every process should execute with the least set of privileges necessary to complete the job.
    - Any elevated permission should only be accessed for the least amount of time required to complete the privileged task
- Sanitize data sent to other systems: Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components

- Practice defense in depth: Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability
- Use effective quality assurance techniques: Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities.
    - Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program
- Adopt a secure coding standard: Develop and/or apply a secure coding standard for your target development language and platform

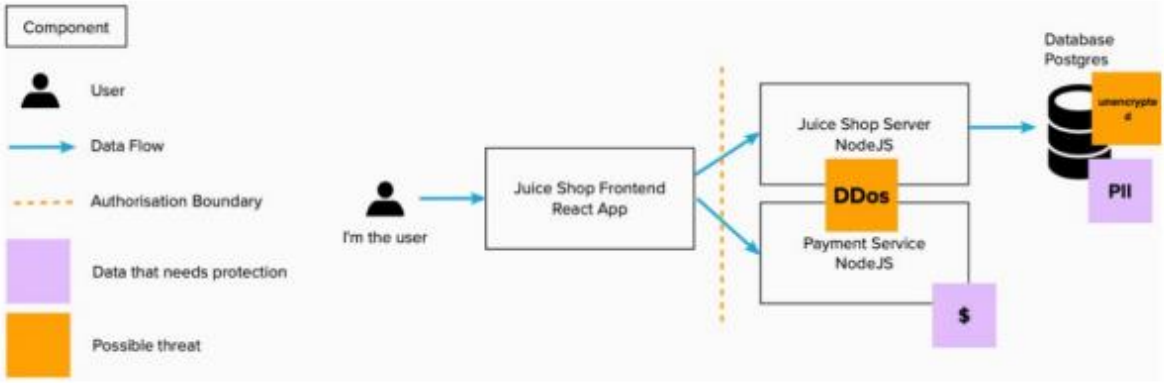
OWASP

- Open Web Application Security Project

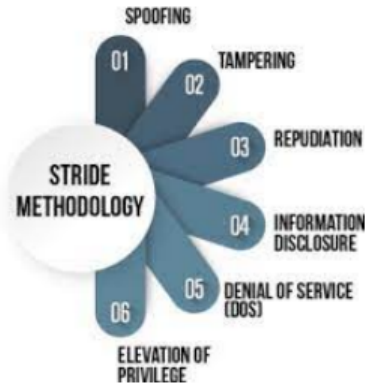
Threat Modelling



- Threat modeling is a structured process with these objectives: identify security requirements, pinpoint security threats and potential vulnerabilities, quantify threat and vulnerability criticality, and prioritize remediation methods
- Threat modeling methods create these artifacts:
    - An abstraction of the system
    - Profiles of potential attackers, including their goals and methods
    - A catalog of threats that could arise



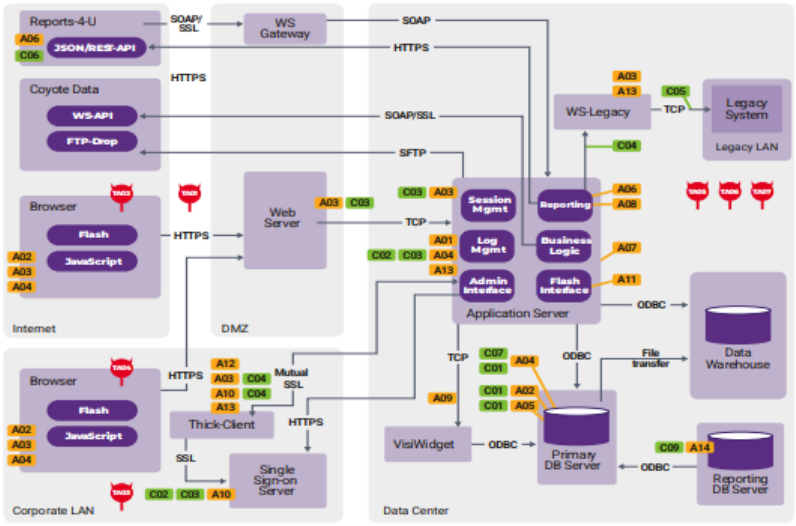
- STRIDE evaluates the system detail design.
    - It models the in-place system.
    - By building data-flow diagrams (DFDs), STRIDE is used to identify system entities, events, and the boundaries of the system.
    - STRIDE applies a general set of known threats based on its name, which is a mnemonic, as shown in the following table:



	Threat	Property Violated	Threat Definition
S	Spoofing identify	Authentication	Pretending to be something or someone other than yourself
T	Tampering with data	Integrity	Modifying something on disk, network, memory, or elsewhere
R	Repudiation	Non-repudiation	Claiming that you didn't do something or were not responsible; can be honest or false
I	Information disclosure	Confidentiality	Providing information to someone not authorized to access it
D	Denial of service	Availability	Exhausting resources needed to provide service
E	Elevation of privilege	Authorization	Allowing someone to do something they are not authorized to do

Table 1: STRIDE Threat Categories

Example of Stride Model



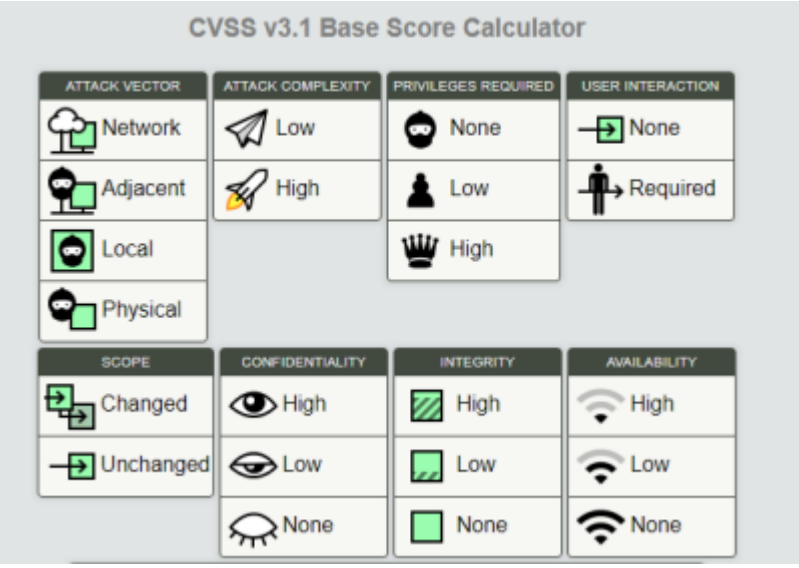
- **Diagram.** What are we building?
- **Identify threats.** What could go wrong?
- **Mitigate.** What are we doing to defend against threats?
- **Validate.** Have we acted on each of the previous steps?

Assets
A01 : Primary DB credentials
A02 : Sensitive application data
A03 : Session tickets
A04 : Application username and password
A05 : Application event tables
A06 : Report data
A07 : Coyote Data SFTP credentials
A08 : Reports-4-U REST credentials
A09 : VisiWidget API
A10 : Active Directory username and password
A11 : Special Flash functionality
A12 : Client certificate for Thick-Client
A13 : Application log file and event data
A14 : Cached reporting data

Threat Agents
TA01 : Unauthorized external user
TA02 : Authorized external application user
TA03 : Unauthorized internal user
TA04 : Authorized internal application user
TA05 : Authorized internal system admin
TA06 : Authorized internal DB admin
TA07 : Authorized developer

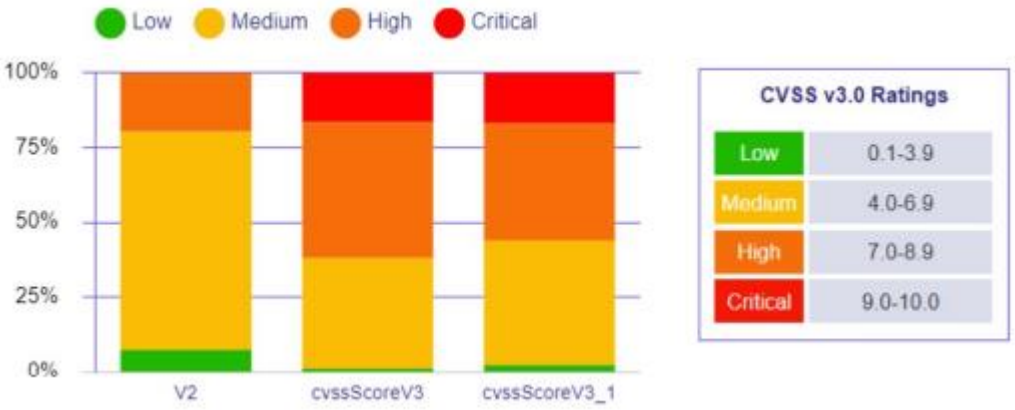
Contols
C01 : Primary DB authentication and schema
C02 : External application user authentication
C03 : One-way SSL
C04 : Mutual SSL
C05 : IP address restrictions
C06 : Token validation (for Reports-4-U)
C07 : Salted SHA-256 hash of user passwords
C08 : Internal user Flash SWF file
C09 : 90-day limit for report data

Common Vulnerability Scoring System



- The Common Vulnerability Scoring System (CVSS) captures the principal characteristics of a vulnerability and produces a numerical severity score.
- CVSS was developed by NIST and is maintained by the Forum of Incident Response and Security Teams (FIRST) with support and contributions from the CVSS Special Interest Group.

- The CVSS provides users a common and standardized scoring system within different cyber and cyber-physical platforms



SAST and DAST

- Static versus Dynamic security tests

Open Source

- Advantages
  - Speed of development & time savings
  - More people fixing bugs & security flaws
  - Open Standards & Standard Interfaces
    - Enables the use of different solutions
    - Less likely to have vendor lock

Open-Source Basics

- Basic open-source project components
  - Community of software developers
  - A software license
  - Tools for hosting the project and communicating
  - A governance model
- Open-Source Foundations
  - Goal – help software developers establish opensource projects
  - Have different approaches and objectives
  - E.g. Apache Foundation, Linux Foundation, Mozilla foundation

Open-Source Software Licenses

- A license protects contributors and users
- Outlines the terms of use
  - How the software can be used, modified, and distributed
- Can limit liability of people who wrote the sw
- Important to understand legalities
- Some popular licences:
  - Apache License 2.0, BSD 3-Clause "New" or "Revised" license, GNU General Public License (GPL) , MIT license, Common Development and Distribution License
- Many different licenses exist
- Before using open-source code or binaries make sure you understand the license terms
- Some licenses are incompatible with other licenses
- “Copyleft” – requires derived code to inherit license terms
- “Permissive” – provides more freedom for code reuse

Open-Source Governance Models (some examples)

- "Do-ocracy" – people who invest the most in the project often have the most say
- Founder-leader – project leaders maintain control of the project
- Self-appointing council or board – issue is that it becomes insular
  - Only end up appointing people who agree with you
- Electoral – who gets a vote?
- Corporate-backed – employees control most of the commits
- Foundation-backed – uses umbrella organisation to establish legal relationships

Open-Source Culture

- Every good work of software starts by scratching a developer's personal itch
- Good programmers know what to write.
    - Great ones know what to rewrite (and reuse)
- Plan to throw one [version] away; you will, anyhow
- If you have the right attitude, interesting problems will find you
- When you lose interest in a program, your last duty to it is to hand it off to a competent successor
- Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging
- Release early. Release often. And listen to your customers.
- Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.
- Smart data structures and dumb code works a lot better than the other way around
- If you treat your beta-testers as if they're your most valuable resource, they will respond by becoming your most valuable resource
- The next best thing to having good ideas is recognizing good ideas from your users.
    - Sometimes the latter is better
- Often, the most striking and innovative solutions come from realizing that your concept of the problem was wrong.
- Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away.
- Any tool should be useful in the expected way, but a truly great tool lends itself to uses you never expected
- When writing gateway software of any kind, take pains to disturb the data stream as little as possible — and never throw away information unless the recipient forces you to!
- When your language is nowhere near Turing-complete, syntactic sugar can be your friend
- A security system is only as secure as its secret. Beware of pseudo-secrets
- To solve an interesting problem, start by finding a problem that is interesting to you.
- Provided the development coordinator has a communications medium at least as good as the Internet, and knows how to lead without coercion, many heads are inevitably better than one

Running A Free Software Project

- Do need to get the culture and motivating forces right
    - People should feel that their connection to a project, and influence over it, is directly proportional to their contributions
    - Every good work of software starts by scratching a developer's personal itch.
- Additionally, complexity due to open-source culture
- Most free software projects fail: ≈ 90–95%
    - For some familiar reasons: unrealistic or vague specifications, poor management, insufficient design
    - Then there are also problems unique to open source

Open-Source Issues

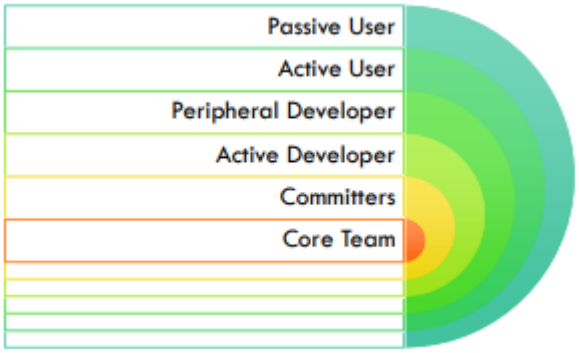
- Unrealistic expectations about the benefits of open source
    - Too few active developers who volunteer their time to your project

    - Open code has to be made comprehensible to strangers
    - Needs a development web site and communication tools
- Barriers to entry for new developers and new users due to poor presentation and packaging.
    - Need developer and user documentation
- Fallacy that little or no project management is required in open source
    - Management practices used for in-house development do not work
- Management in an open source project isn't always very visible
    - Active, often informal, subtle, and low-key
    - Coercive techniques don't work

Open-Source History and Culture

- Shared culture and etiquette for participation
- To work in Open-Source you must be aware of the issues and polemics
- Free software is a volunteer community with very light "investment"
    - If people are unhappy in one project, they just wander off to another one
    - Most people involved never actually meet face-to-face
    - They simply donate bits of time whenever they feel like it
- People should feel that their connection to a project, and influence over it, is directly proportional to their contributions

The Onion Model of Open-Source Community Structure



- Concentric rings represent decision-making power with respect to contributions
    - At the innermost core, decision making is most concentrated.
    - The next three outward circles consist of developers.
    - The last two circles represent users.
- The core team consists of one or more developers who make decisions about the software's direction and contributions to the code base
    - Smallest team with greatest decision-making power
- The committers have commit rights that allow admission of code into the main code base
- Active developers regularly contribute to the source code repository
- Peripheral developers submit code as patches, meaning their code does not enter the main code base until a review from the core team.
- Active users contribute documentation, localization, supporting users in discussion forums, or filing bug reports.
- Passive users download and use the software without connecting to the community.

Open-Source Projects

What A Project Needs

- Web site
  - One-way information from the project out to the public
  - Interface for other project tools
- Mailing lists
  - Usually most active communications forum in the project
  - "medium of record"
- Real-time chat
  - Quick, lightweight discussions; question/answer exchanges
- Version control
  - Enables developers to manage code changes conveniently
  - Enables everyone to watch what's happening to the code
- Bug tracking
  - Enables developers to keep track of what they're working on, coordinate with each other, and plan releases.

Transform Your Private Vision Into A Public One

- Choose a Good Name – check against existing names & trademarks
- Register the domain (even if you use a canned host)
  - Canned Hosting: sites that offer free hosting and infrastructure for open source projects
  - Web area
  - Version control
  - Bug tracker
  - Download area
  - Chat forums
  - Regular backups
- Set up the web site

Home Page

- For example, www.openoffice.org/: OpenOffice.org

  "To create, as a community, the leading international office suite that will run on all major platforms and provide access to all functionality and data through open-component based APIs and an XML-based file format."

- Project Name
- Then – a Clear Mission Statement
  - Prominent short description so people can decide if they are interested in learning more.
- State that the Project is Free and what the licensing is

Communication

- Communications Channels to humans involved with the software
  - Mailing lists

    - Chat rooms
    - IRC channels
    - Other forums for interaction with developers
- Developer Guidelines
    - Social rather than technical – explain how the developers interact with each other and with the users
    - Instructions on how to report bugs and submit patches
    - Some indication of how development is usually done
        - Project is a benevolent dictatorship, a democracy, or something else (project leadership)
    -

Features, Requirements and Status

- Features:
    - Answers "What does it do?"
    - Brief list of the features the software supports
        - Put “planned” or “in progress” where that is so
- Requirements
    - Computing environment required to run the software
- Computing environment required to run the software
    - And they can consider getting involved as developers
- Development Status
    - List the project's near-term goals and needs
        - e.g., looking for developers with particular expertise
        - Gives an idea of how big community is
    - History of past releases, with feature lists
        - Gives an idea of progress and speed

Accurate Accounting of Deficiencies

- Norm in the open-source world
    - Don't exaggerate the project's shortcomings
- Identify issues scrupulously and dispassionately in context
    - In the documentation
    - In the bug tracking database
    - On a mailing list discussion
- This is not defeatism on the part of the project
- This is not a commitment to solve the problems by a certain date
    - Unless the project makes such a commitment explicitly
- People will discover the deficiencies anyway
    - It’s much better for them to be psychologically prepared
    - Shows a solid knowledge of how things are going

Distribution

- Downloads
    - Source code in standard formats
    - Executables will come later
- The distribution mechanism should be as convenient, standard, and lowoverhead as possible

- Important to standardize build and installation procedures
    - Otherwise, you discourage developers who might otherwise have contributed to the code
    - Someone visits a web site, downloads the software, tries to build it, fails, gives up and goes away

Documentation: Minimal Criteria

- Tell the reader clearly how much technical expertise they're expected to have.
- Describe clearly and thoroughly how to set up the software (with simple diagnostic test to confirm that it is set up correctly.
    - Startup documentation is in some ways more important than actual usage documentation
    - When people abandon, they abandon early
- Give one tutorial-style example of how to do a common task.
    - If time is limited, pick one task and walk through it thoroughly
    - If someone sees that the software can be used for one thing, they'll start to explore what else it can do on their own
- Label areas where the documentation is known to be incomplete
- Align yourself with your users' point of view
- Availability of documentation
    - On web
    - Downloadable distribution
- Developer documentation
    - This is technical not social
    - Do not delay a release to write as long as developers are there to answer questions
    - If you have to choose, write for the users first
        - Developers can use this too
    - A wiki works if you have active editors
- Screenshots are useful for GUI's and graphics
- FAQ: Base it on real questions and keep up to date

Choosing A License and Applying It

- "Do Anything": If you're OK with your code being used in proprietary programs, e.g., MIT/X-style license:
- The GPL: If you don't want your code to be used in proprietary programs,
    - GNU General Public License is most widely recognized
- How to Apply a License to Your Software
    - Tell the public how you intend to license on project web page:
        - Give the name of the license, and make it link to the full license text
    - For legal purposes the software itself must contain the license
        - Full license text in a file called COPYING (or LICENSE),
        - A short notice at the top of each source file, naming the copyright date, holder, and license, and where to find the full text of the license

Announcing

- Tell the world!

- Ongoing debate in the free software world about whether it is necessary to begin with running code, or whether a project can benefit from being opened even during the design/discussion stage.
    - But: Running code is still the best foundation for success

Open-Source Practices

Dynamics Of A New Project

- Choosing a mailing list address is easy; ensuring that the list's conversations remain on-topic and productive is another matter entirely
- Stability in a project does not come from formal policies, but from a shared, hard-to-pin-down collective wisdom that develops over time
- This effort is aided by the fact that people generally show up expecting and looking for social norms

Setting The Tone

- Avoid private discussions – no more inner circle
- Public discussion also
    - Helps train and educate new developers
    - Train you in the art of explaining technical issues to people who are not as familiar with the software as you are
    - Discussions and its conclusions will be available in public archives forever after
- Nip rudeness in the bud $\Rightarrow$ never let bad behaviour slide by unnoticed

Practice Conspicuous Code Review

- Heart of open source is to get people looking at each other's code
    - Commit emails – the log message and diffs for every change
    - Reviews are public
- Easy to see in diffs:
    - Security vulnerabilities
    - Memory leaks
    - Insufficient comments or API documentation
    - Off-by-one errors
    - Caller/callee discipline mismatches
    - And other problems that require a minimum of surrounding context to spot
- With practice – larger-scale issues such as failure to abstract repeated patterns

Version Control

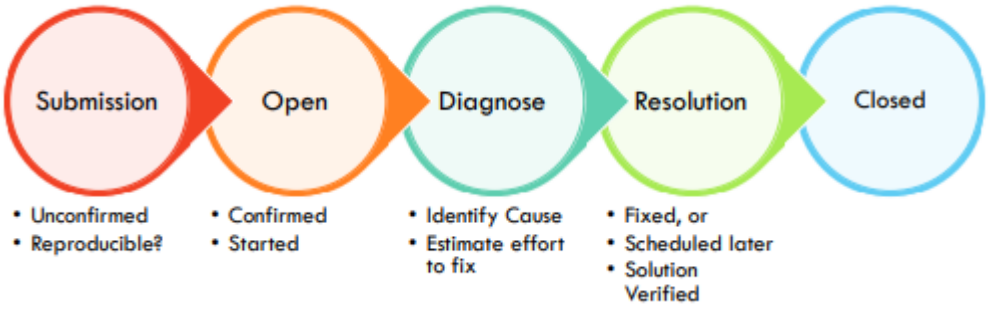
- Addresses the key area that needs managing in Open Source: coordination of developers, in virtually every aspect of running a project:
    - Inter-developer communications
    - Release management
    - Bug management
    - Attribution and authorization of changes by particular developers
    - Code stability
    - Experimental development efforts

- A version control system (or revision control system) is a combination of technologies and practices for tracking and controlling changes to a project's files:
    - Source code
    - Documentation
    - Web pages
- People won't take the project seriously if it doesn't use version control with at least minimal competence
- Nightly source snapshots are not fine-grained enough for development
- People need real-time access to the latest sources
    - Allows team members work simultaneously on a project's source code without stepping on each other's toes, or crunching each other's code
    - Shows both users and developers that this project is making an effort to give people what they need to participate

Bug Or Issue Tracker

- Tracker is part of the public face of the project
    - Anyone may file an issue,
    - Look at an issue,
    - Browse the list of currently open issues
- The size and skill of the development community determines the rate at which issues can be resolved
- Try to acknowledge each issue the moment it appears
    - The tracker must be connected to its own mailing list, such that every change to an issue, including its initial filing, causes a mail to go out describing what happened.
    - Should (try to) capture the reporter's email address, so she can be contacted for more information
- Examples: Bugzilla, and GNATS

Issue Life Cycle



Accessible Well-Maintained Bug Database

- Bug Tracker is useful to developers
- Signifies that a project should be taken seriously
    - Bugs are responded to promptly
    - Duplicate bugs are unified, etc.
- The higher the number of bugs in the database, the better the project looks!
    - Number of bugs recorded really depends on three things:
        - (1) Absolute number of bugs present in the software,
        - (2) Number of users using the software,

        - (3) Convenience with which those users can register new bugs.
    - Of these, 2 & 3 are more significant than 1.

Conclusion: FOSS4Dev

- Creating Free and Open-Source Software for Development requires the methods and skills that we advocate.
- FOSS depends on access to source-code
    - Need local Software Engineering skills to use and modify code appropriately
- Significant lock-in to proprietary software in the developing world due to a lack of skills in exploiting FOSS
    - Bridges.org: “Specific software applications that could make computers more useful to local communities — such as putting ICT to work to improve healthcare and education, and designed with cultural factors in mind — are still missing”
- We must address such issues and take ownership of FOSS4DEV