Lecture 1-9

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

Requirement Specification

1. Create high-level descriptions
2. Distinguish between ‘right’ and ‘wrong’ system
3. Capture ‘WHAT’ not the ‘HOW’ of the solution

Requirements vs Specification

1. Requirements for the user; Specifications for the developer
2. Write the Requirements in the User language
3. Write the Specifications in the system language
4. Make sure that the Specifications meet the Requirements

Non Functional Requirements

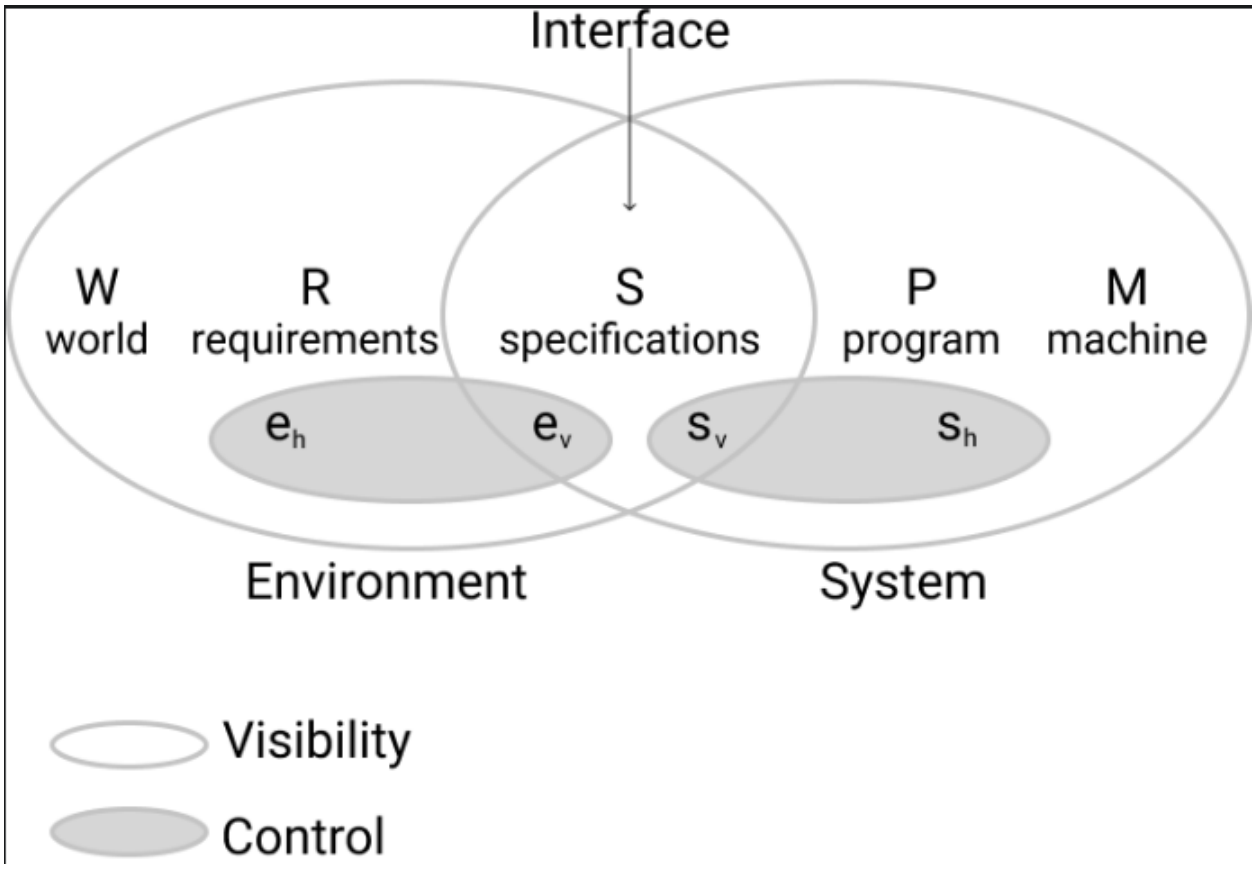
1. It is not about ‘what’ the system will do but rather how the system will perform the behaviors
2. Define system properties and constraints
3. Process requirements
4. Often more critical than functional requirements

WRSPM

1. W — is the world assumption, that we know is true. They have an impact on our system and our problem domain. These things everyone takes for granted, and they are one of the most difficult parts to capture.
2. R — is the requirement. This is the user’s language understanding of what the user wants from the solution. For example, users want to withdraw money. The ATM is the solution.
3. S—is the specification. It is the interface between how the system will meet those requirements. It is written in a system language that says

in plain English what the system will do. The specification is how the system meets the requirements. For example, to withdraw money from the ATM, you have to insert your card, insert your PIN, etc. Those are the things the user doesn't care about. The user wants to get money.

4. P—is the program. It is what software developments will write. The program will meet the specifications to provide the user goal for requirements. The program has all the code, underlying frameworks, etc.
5. M—the machine. It is the hardware specification. For example, it includes the roller for distributing money, the lockbox, etc.

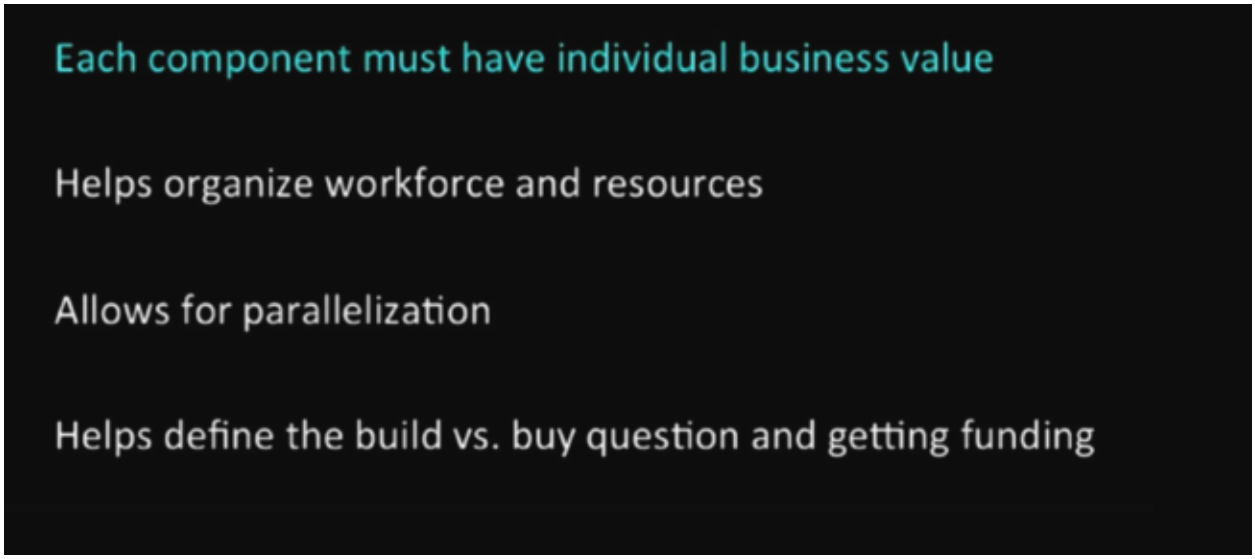


'W' and ' S' imply 'R' and 'P' and 'M' imply 'S' ⇒ Solution correct

Software Architecture

Software architecture is “the structure of the components of a program/ system, their interrelationships, and principles and guidelines governing their design and evolution over time.” – Garlan & Perry

1. Mainly about decomposing the system into components



Models

1. Pipe and Filter

   [A,B,C,D]------:[ ]
   [Final]
   [A,B,E,F]-------:[ ]

   We have to decide the pipeline.
   ABCD or ABEF

2. Blackboard Model

   Components don't talk to each other but to the blackboard

3. Layered Model

   UI
   Data
   Application
   Technical work involved

Each component is independent of the other

4. Client-Server Model
   System ---> Cloud ---> Server
5. Event-Based
   Similar to the blackboard model but in this case the event board in return talks to the other components.

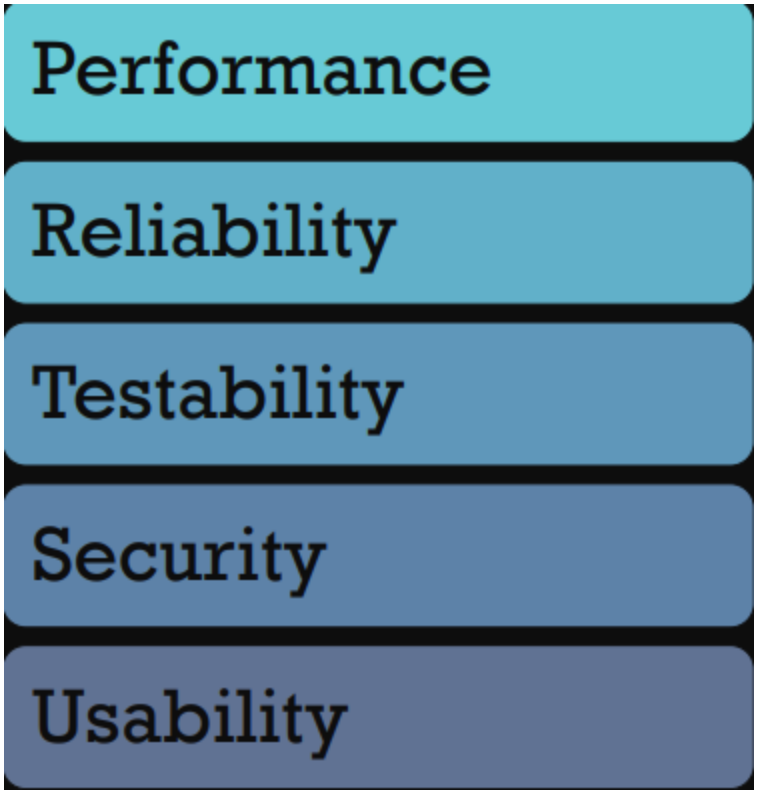
Software Architecture Process

1. Design Process
2. System Structuring
3. Control Modeling
4. Modular Decomposition (maintainability, reliability, security)

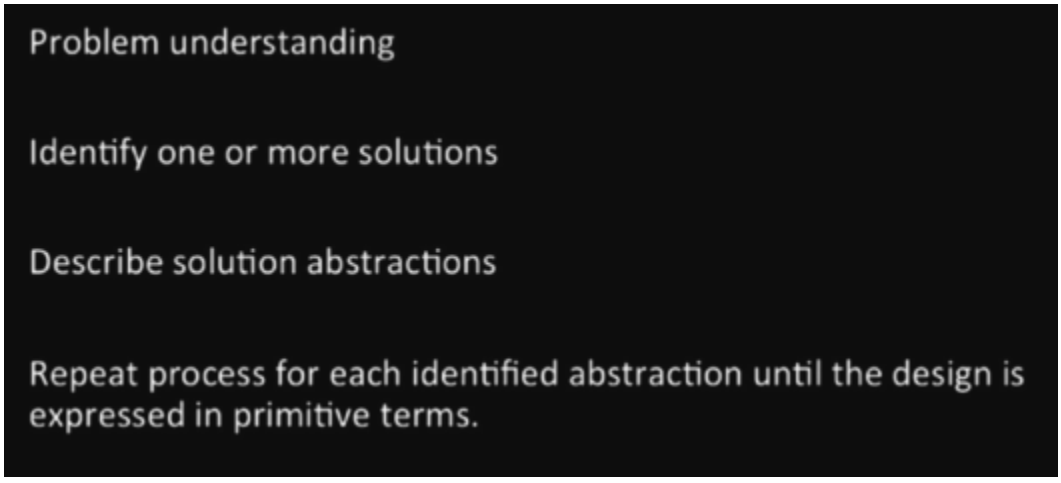
Subsystems vs Modules

1. Sub-System: Independent system which holds the business value
2. Module: Component of a subsystem that can not function as a standalone system

Software Quality Attributes



Stages of Design



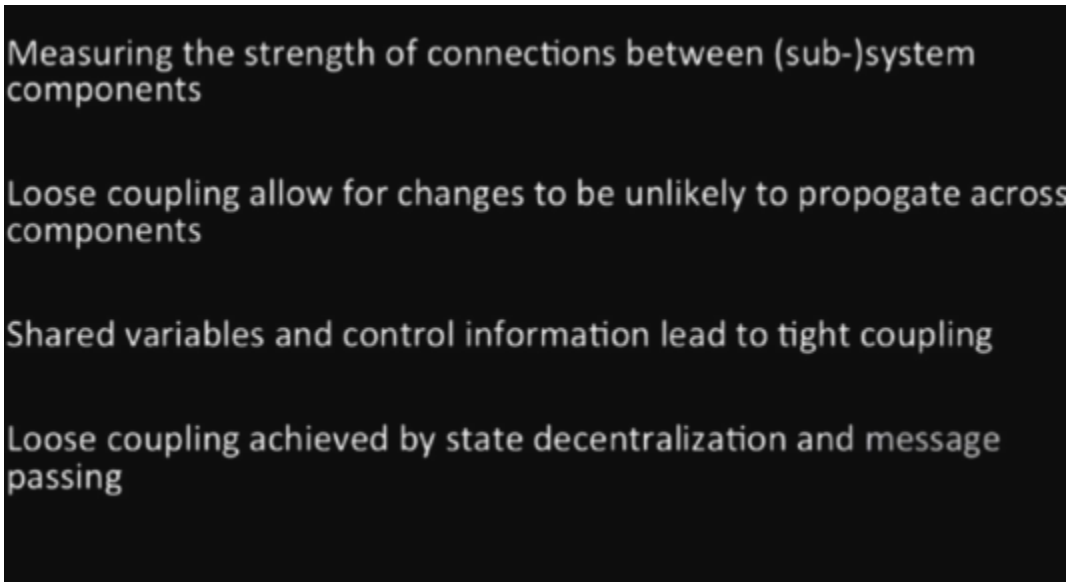
Modularity

1. Complex systems must be broken down into smaller parts
2. 3 main Goals - Decomposability Composability Ease of Understanding

Data Encapsulation

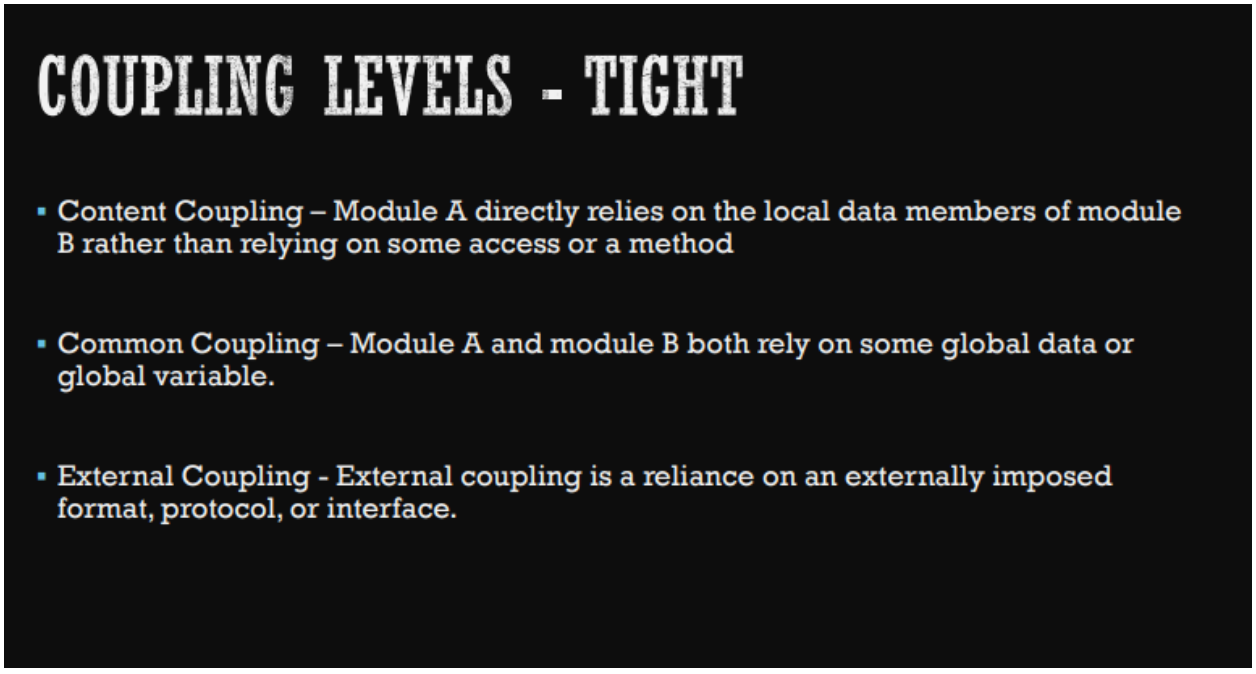
1. Encapsulate the data
2. Help find where problems are
3. Makes design more robust

Coupling

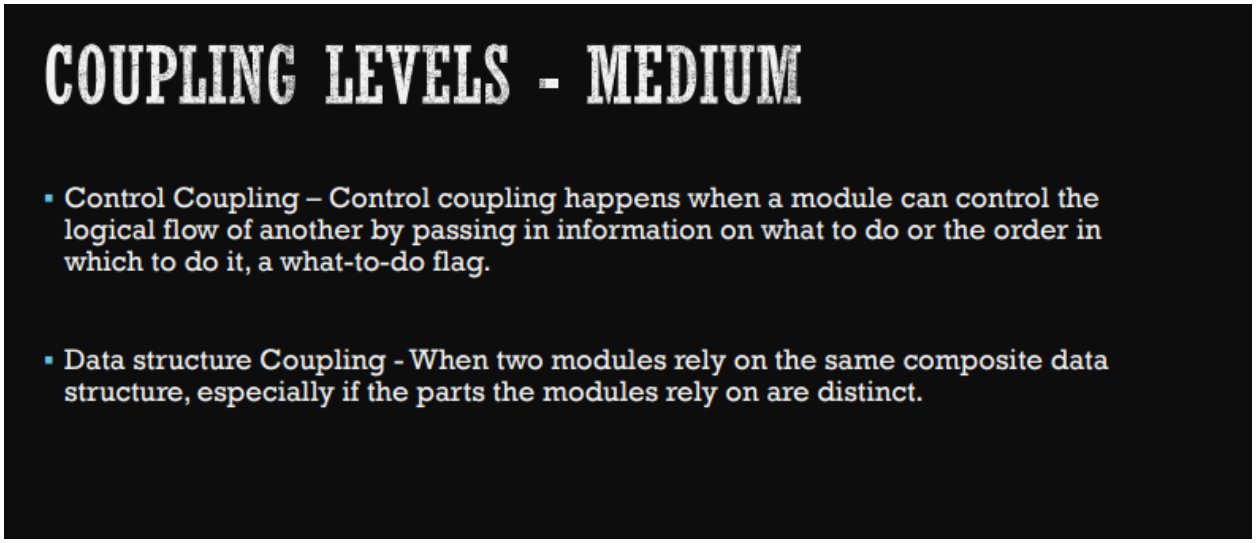


A Trade-off between coupling and cohesion is required. Loose coupling is promoted. (Types - Tight, Medium, Loose)

Tight Coupling



Medium Coupling



Loose Coupling

- Data Coupling – When only parameters are shared. This includes elementary pieces of data like when you pass an integer to a function to compute the square root
- Message Coupling - It's primarily achieved through state decentralization, and component communication is only accomplished either through parameters or message passing
- No Coupling

Cohesion

How well the module's components fit together. Software should be as cohesive as possible. Inheritance will weaken cohesion. Should document why instead of how

Implements a single logical entity or function

Represents a desirable design attribute

Divides into various levels of strength

COHESION LEVEL - WEAK

- Coincidental Cohesion – Just because in the same file
- Temporal Cohesion - Temporal cohesion means that the code is activated at the same time
- Procedural Cohesion - one comes after the other doesn't really tie them together, not necessarily
- Logical Association - components which perform similar functions are grouped

COHESION LEVELS - MEDIUM

- Communicational Cohesion - all elements of the component operate on the same input or produce the same output
- Sequential Cohesion - when one part of the component is the input to another part of the component. It's a direct handoff and a cohesive identity.

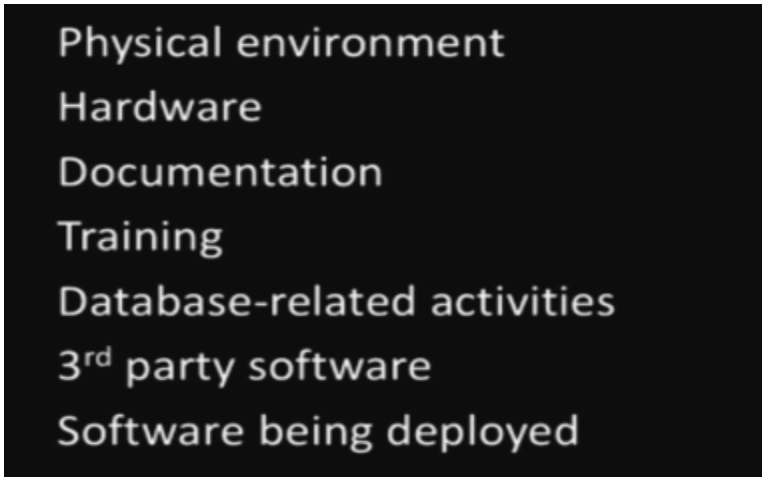
COHESION LEVEL - STRONG

- Object Cohesion - each operation in a module is provided to allow the object attributes to be modified or inspected
- Functional Cohesion - Beyond sequential cohesion to assure that every part of the component is necessary for the execution of a single well-defined function or behavior. So, it's not just input to output, it's everything together is functionally cohesive.

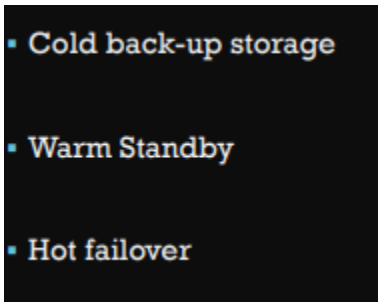
Deployment

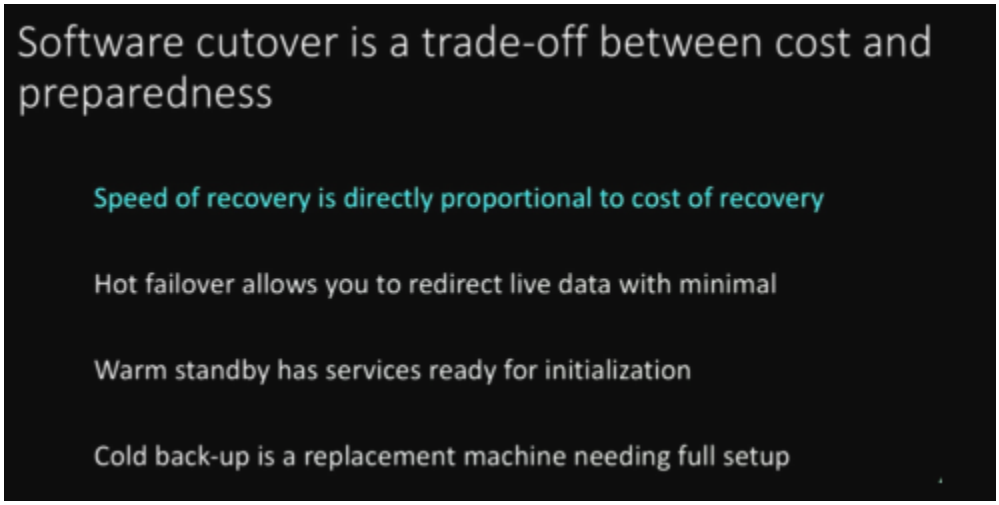
Means production deployment, release to users, Occurs after testing/QA, and others have signed off on the release.

Deployment plan concerns



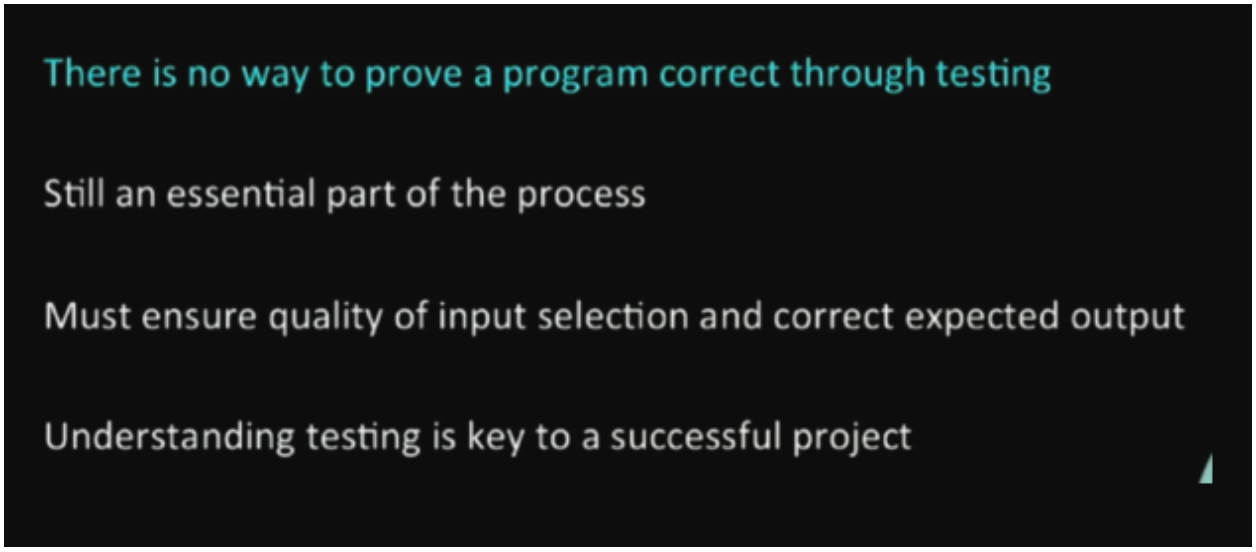
Cut Over Strategies



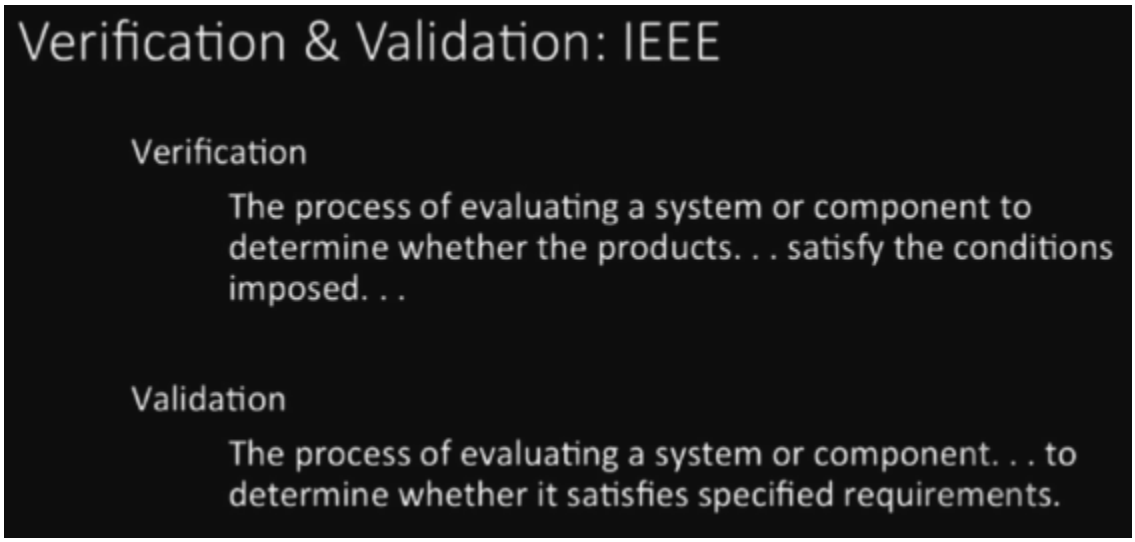


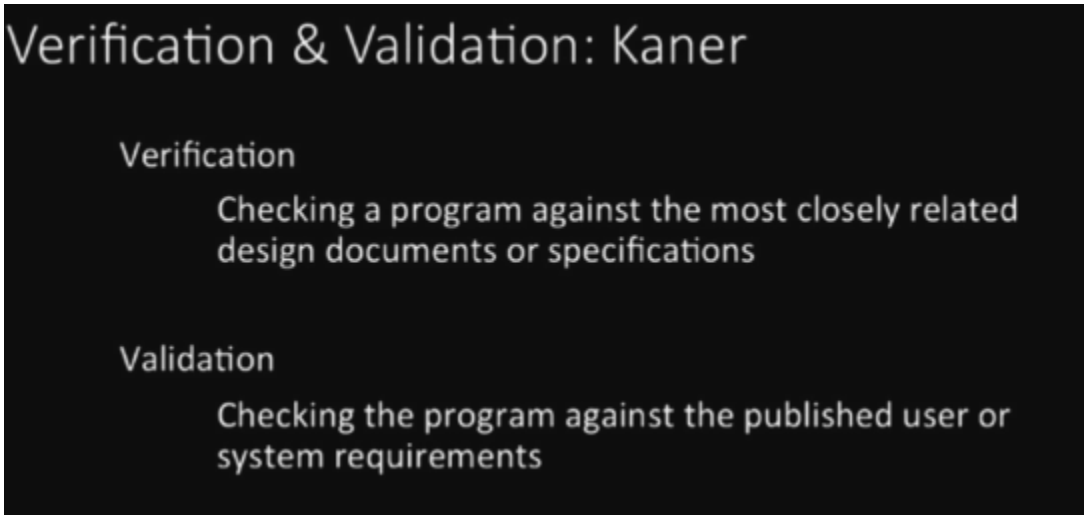
Testing

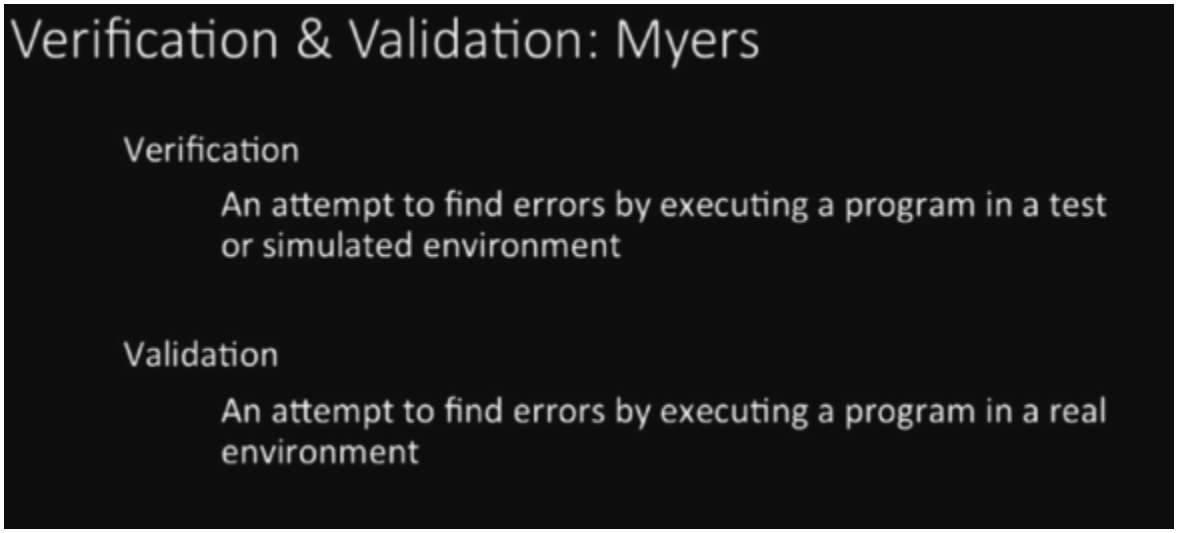
1. A process to find and fix defects in a system's implementation.



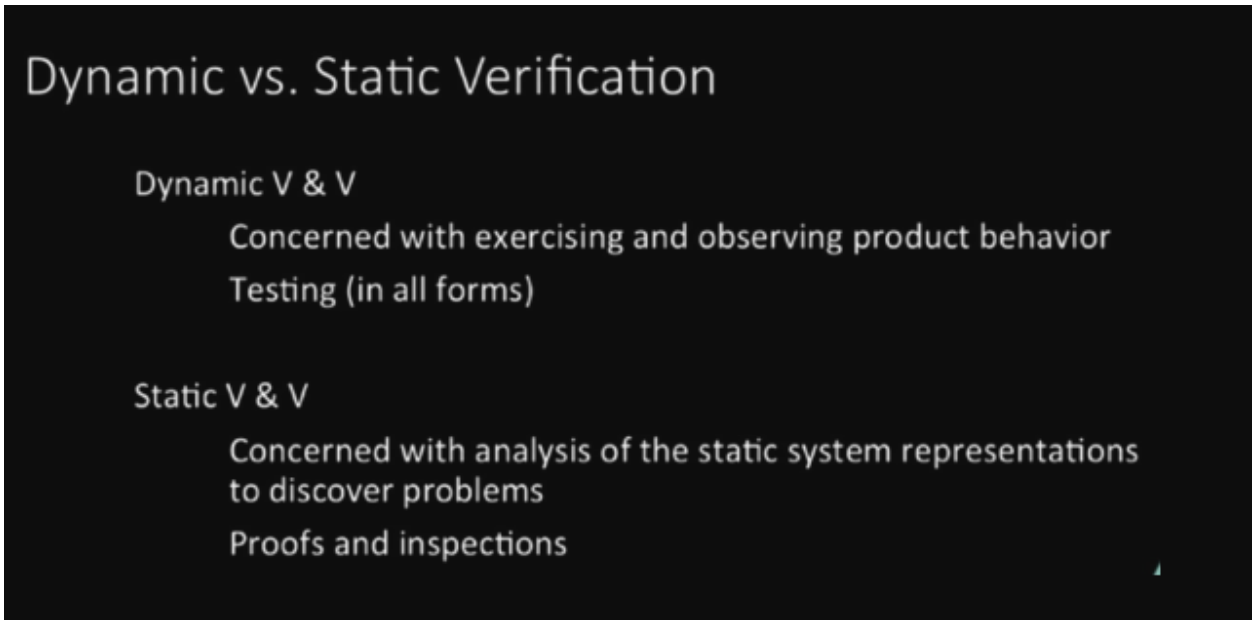
Verification vs Validation

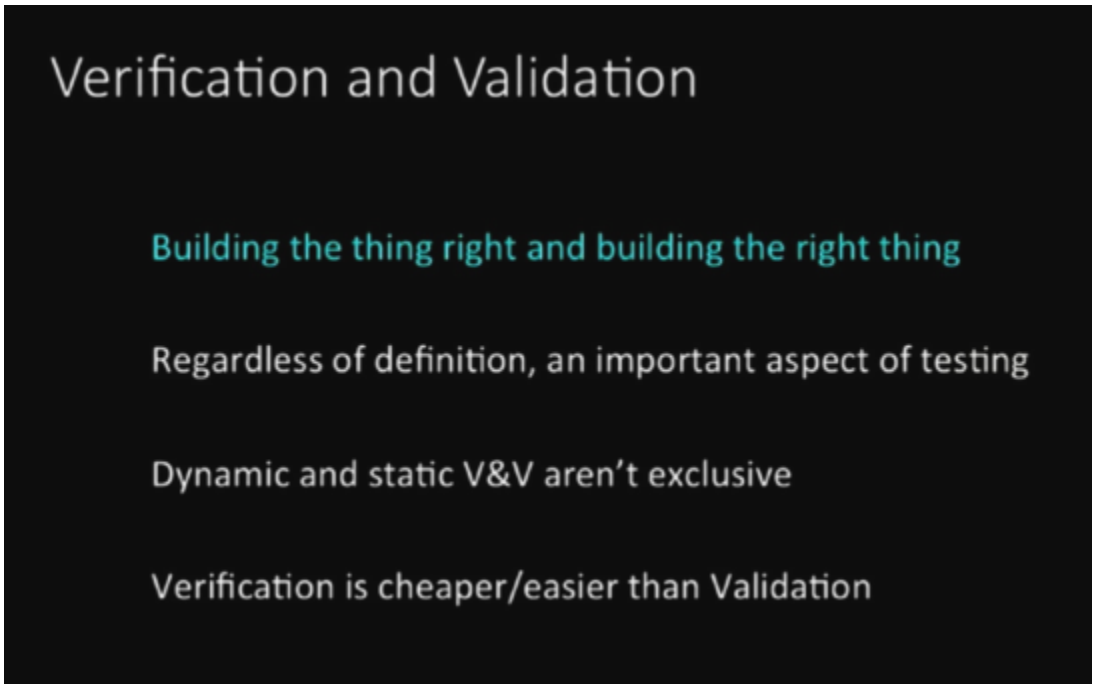






In simple words
Verification - Confirms that the software performs and conforms to its specifications. Are we building the thing right?
Validation - Confirms that the software performs to the user's satisfaction. Are we building the right thing?





Verification is cheaper than validation

Software Models

Predictive vs Adaptive

Predictive

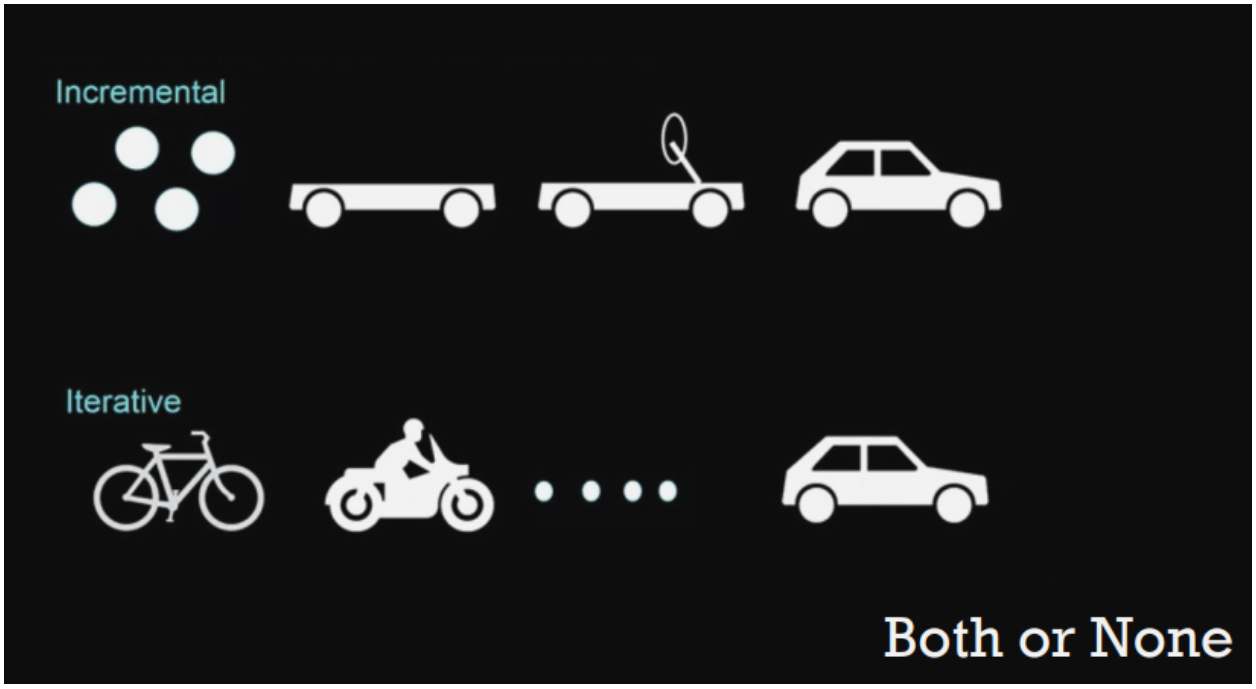
1. It is a software development process in which the model is being designed, executed and analysis is done step by step till the product reaches its end and satisfies all its requirements. The predictive approach concentrates on making strategies and analyzing the project for its better development and predicting any risk. Predictive development is also named as Iterative Waterfall method. Waterfall methodology just follows the direction where the first flow goes or those who lead the direction, just like that this methodology also depends on the previous step of analyzing and further it follows the above phase and then processes it.

Adaptive

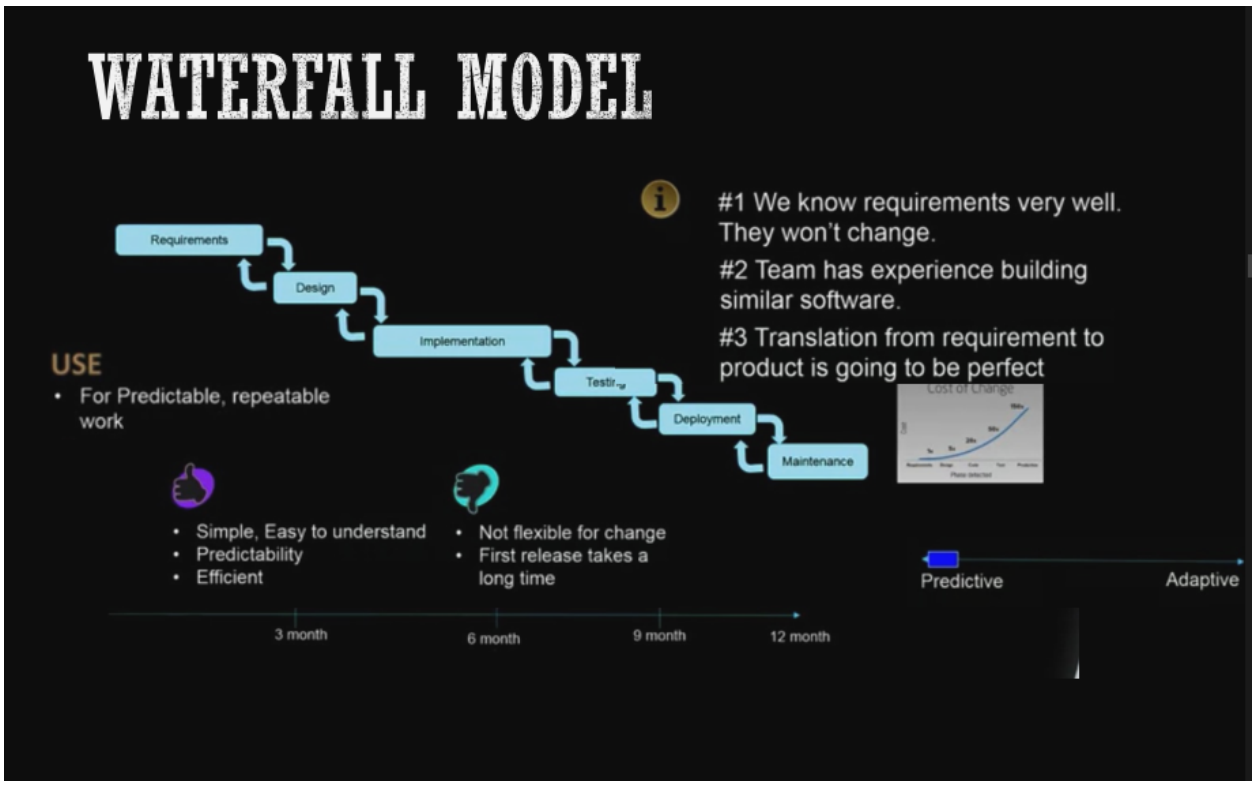
1. Adaptive development is also known as Agile methodology. Adaptive development altered the previous traditional method that is Waterfall method by a contiguous series of reflection, partnership, and grasp cycles. This cycle provides us the gaining continuous knowledge and adaptation to the developing stage of the project. The main focus of the Adaptive development life cycle is to be concentrated on projects, identified, repeated, time-boxed, courageous, and change liberally.

Incremental vs Iterative

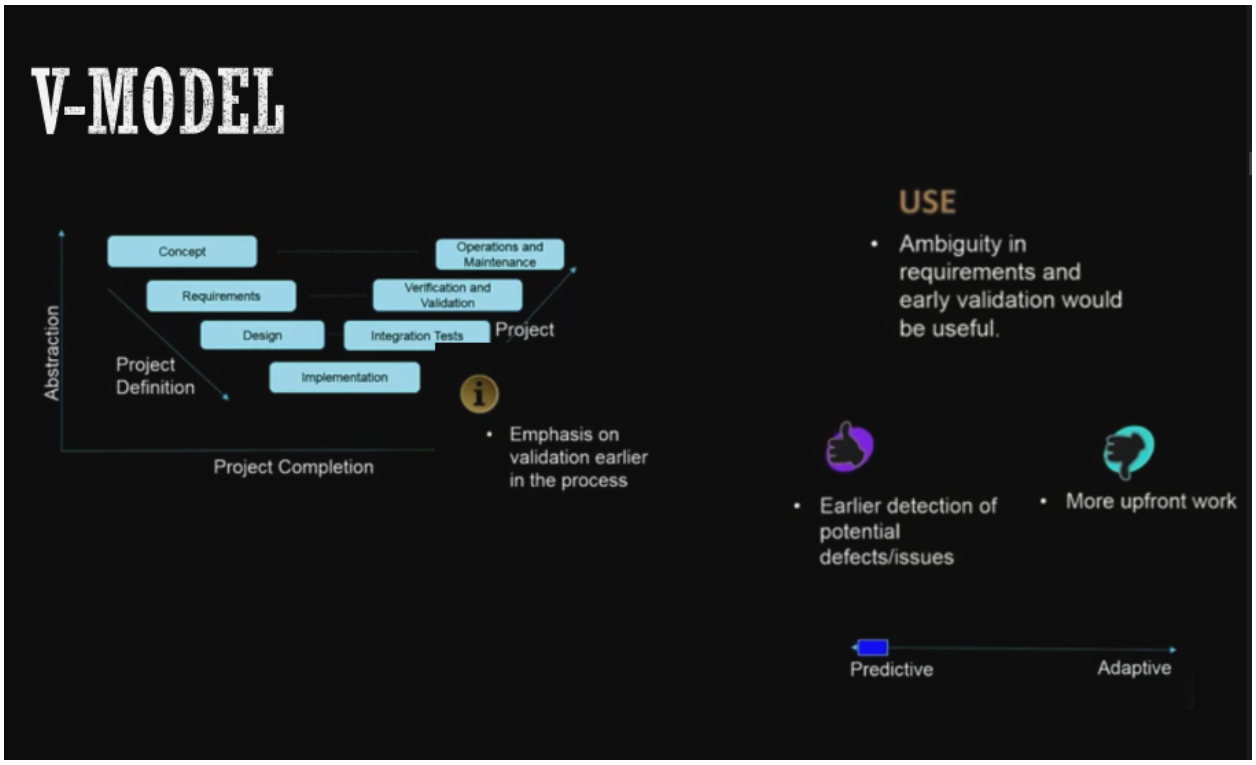
1. The incremental model is where you build the entire solution in parts, but at the end of each phase or section, you have nothing to review or feedback on. You need to wait until the final stage of the incremental process to deliver the final product.
2. The iterative model is where we iterate the idea and improve as we iterate over versions. When you move from one version to another, you decide (based on feedback) what you need in the new version as a better choice and what you need to discard.



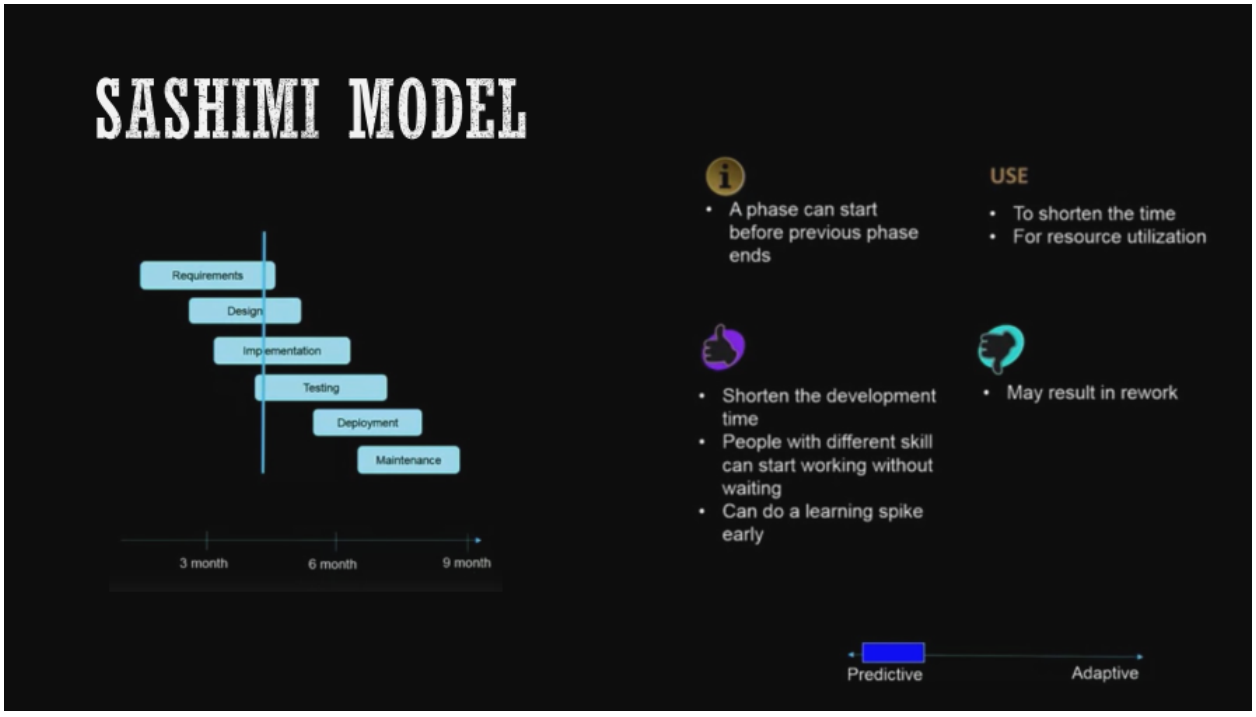
Waterfall Model



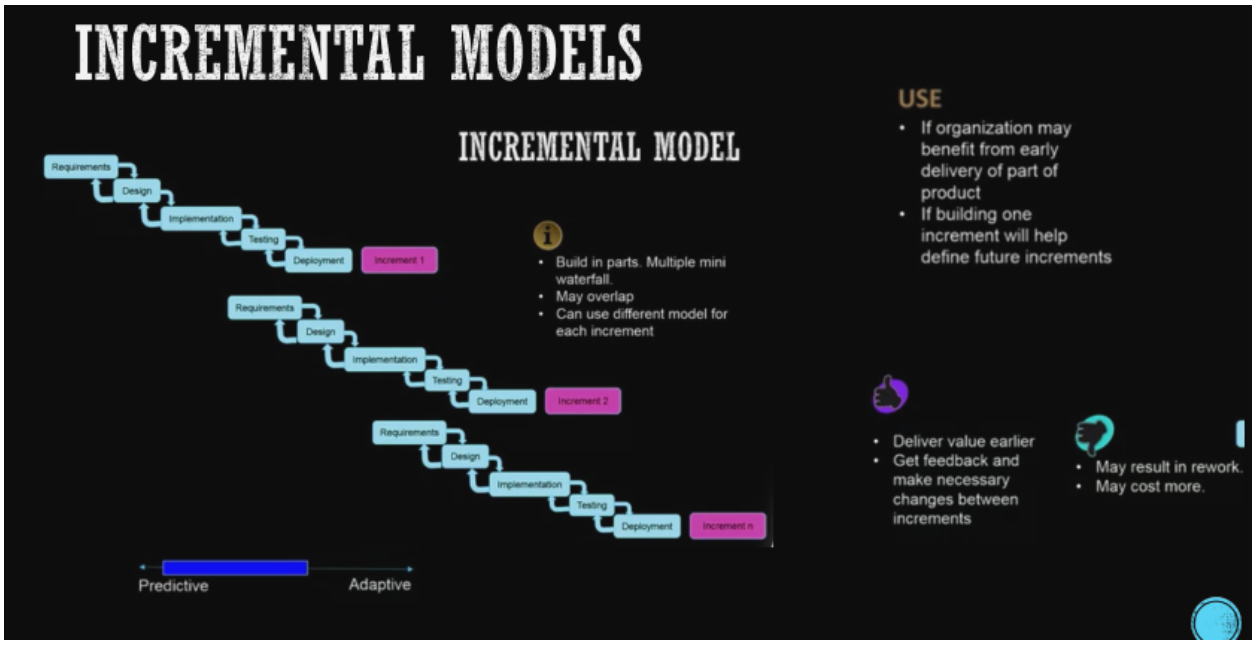
V Model



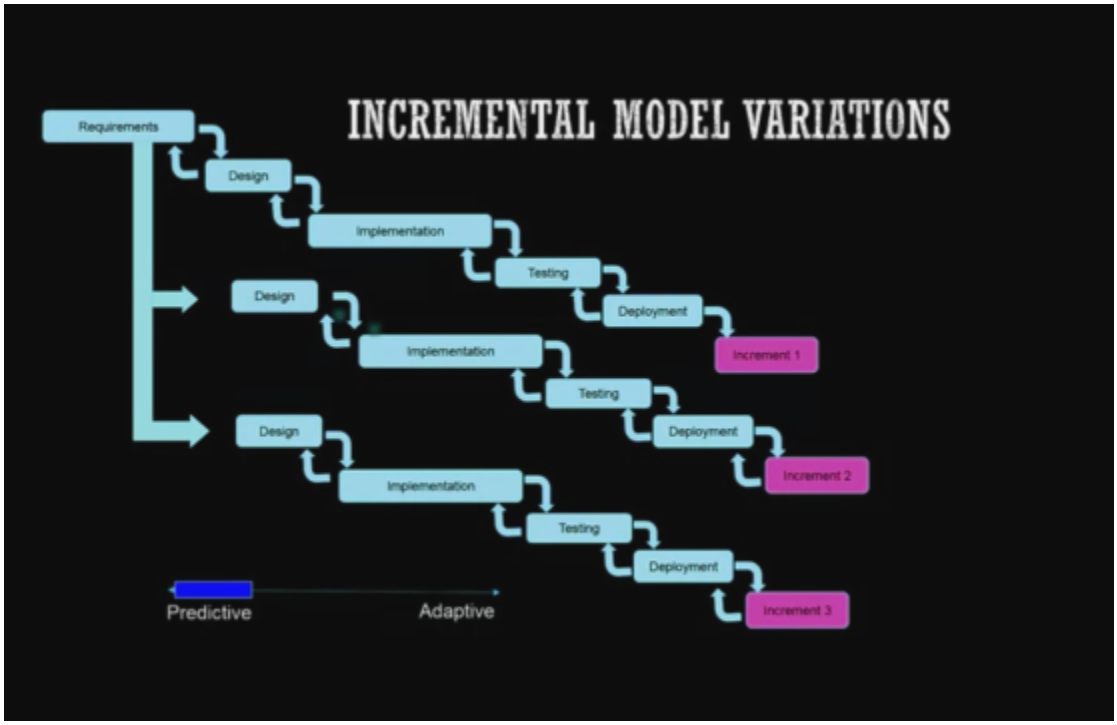
Sashimi Model

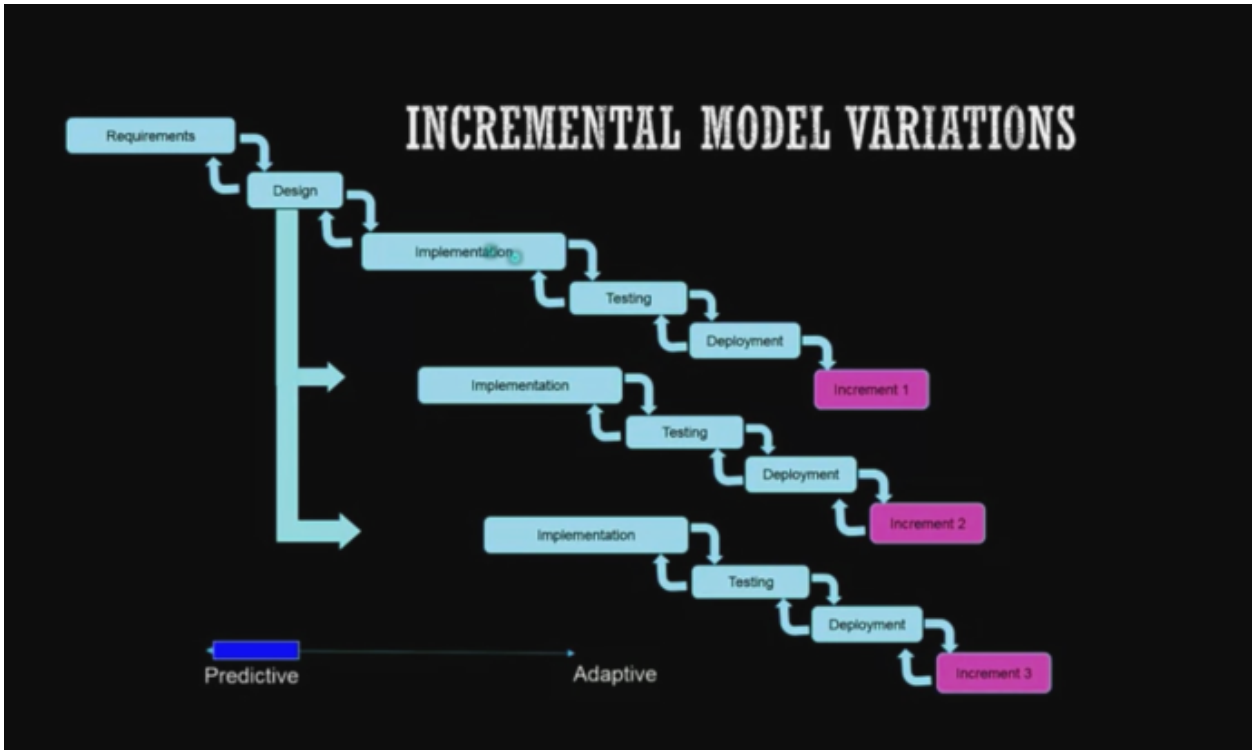


Incremental Model

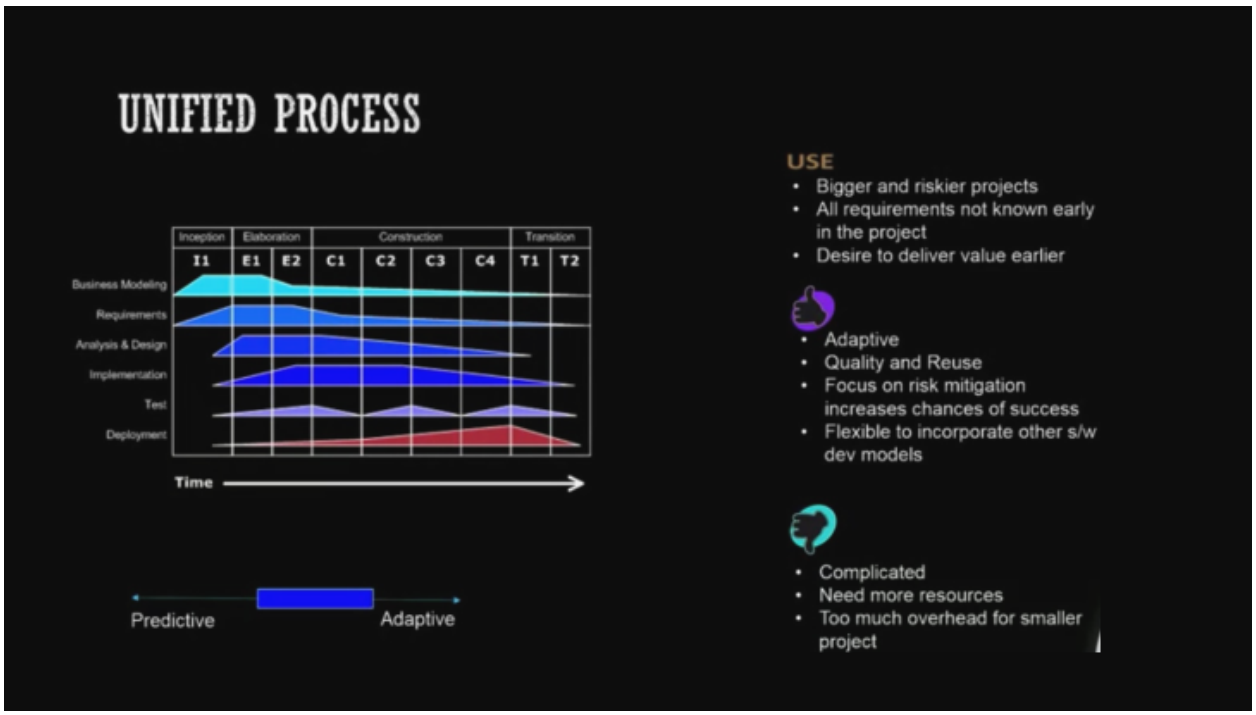


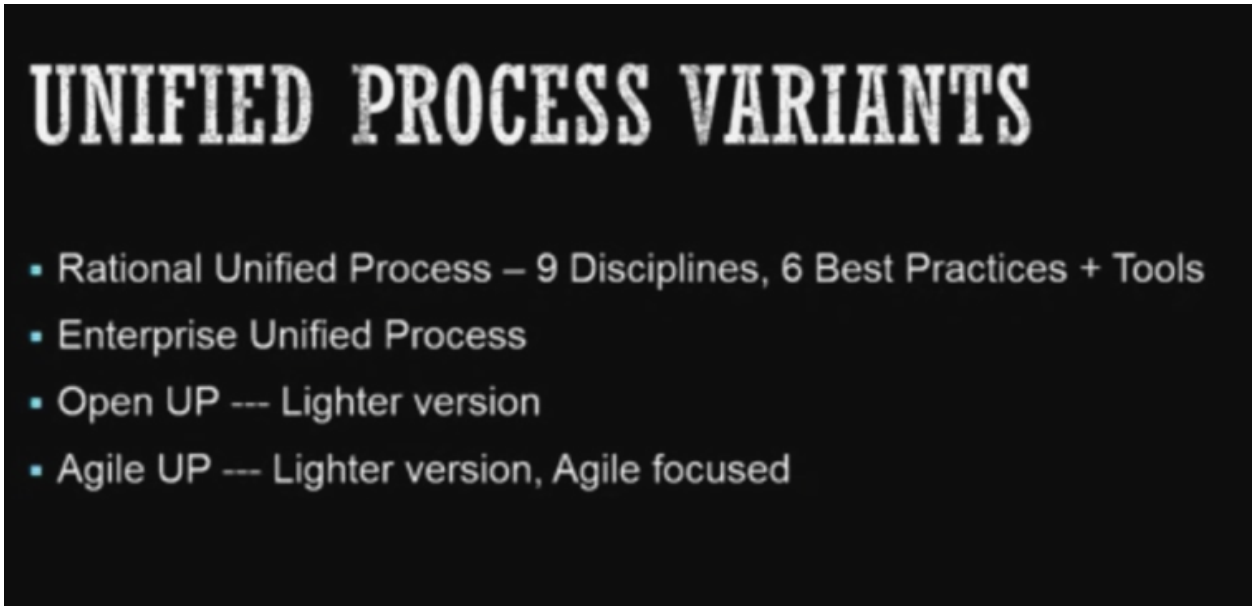
Variations Incremental Models



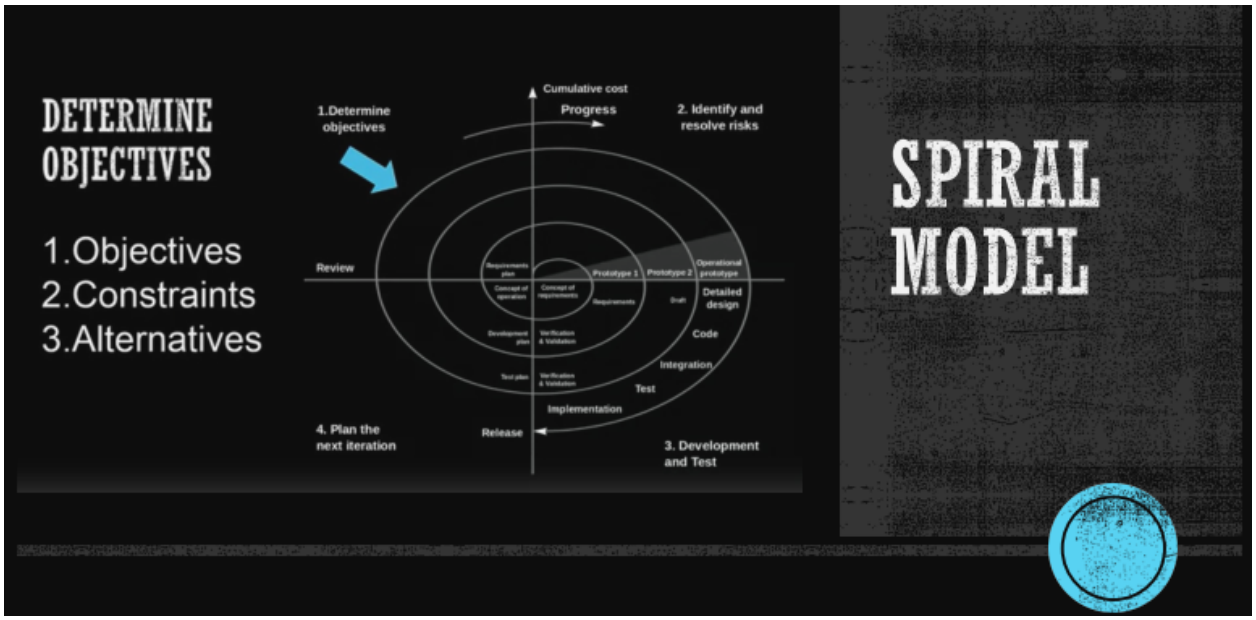


Iterative Models





Spiral Model



Identify and resolve risks
Dev and tests - Work done to meet objectives
Plan for next iteration - Review work done and commitment for the next iteration.

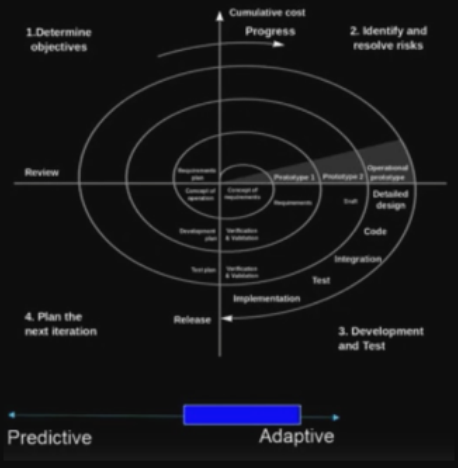
SPIRAL MODEL

USE

- Very large High risk projects

- Adaptive
- Risk focus increases chances of success
- Flexible for using any model
- Minimizes waste.
- Options for go/no-go

- Complicated
- Cost more to manage
- Need stakeholder engagement



Example Modeling

EXAMPLE 1

- Company X need to install a well known HR Management System at a big retailer's HQ.
- Company X has done many such installs on other big retailers before.

Ans - Waterfall model, Sashimi, V

EXAMPLE 2

- Very large hospital with locations all over the world wanted to automate their processes
- Hired a company that is expert in this area and has done similar automation but not at this scale.
- Hospital management want consistency across the globe but not sure about the nuances in different part of the world.
- Few of the places can benefit greatly from the immediate automation

EXAMPLE 2 – ANALYSIS & RECOMMENDATIONS

- Fairly known problem
- Fairly known solution except the scale
- Benefit from phased delivery
- May tweak a bit based on early iterations
- Wants consistency

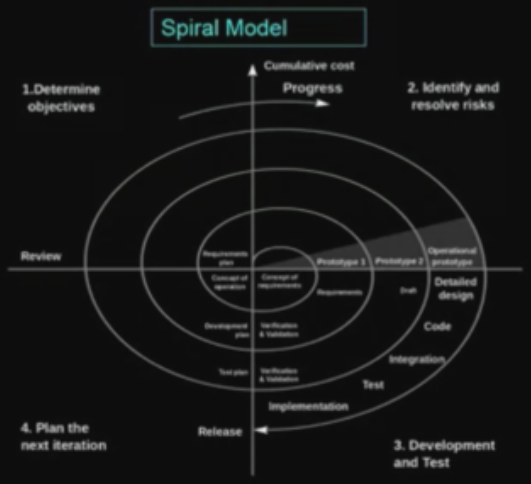
Ans - Incremental Model

EXAMPLE 3

- Defense organization of a country did a recent study and the research recommends new capability the country should build to keep the country protected from potential conflicts in the region.
- The system required to be build has never been attempted and no literature exists for such system.
- It is fairly big and complex system and potentially can take a decade to build.
- Scientists have vague ideas but no concrete plan exists.
- There are lot of Organization stakeholders and constraints that will impact this initiative.

EXAMPLE 3 – ANALYSIS & RECOMMENDATIONS

- Fairly unknown needs and outcomes
- Very very risky
- Very large and complex project

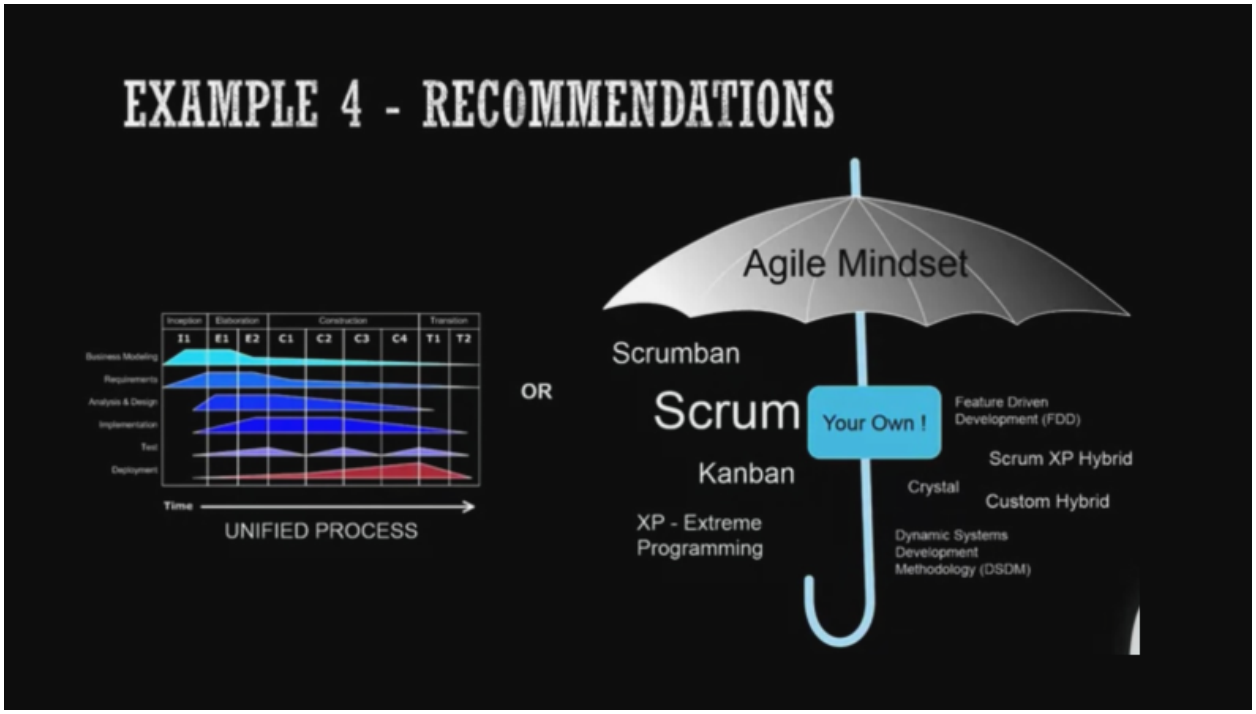


EXAMPLE 4

- A fairly big organization has a product for processing medical prior authorizations submitted by insurance members. Product has been in use for years. This Organization mostly follow traditional project mgmt.
- The current system has several limitations and has not been keeping up to date with current industry trends. It cannot fulfill functionality expected by clients.
- Organization wants to build a whole new system that will satisfy client needs and setup the organization for future.
- The potential team who will be working on this has fairly good domain knowledge and knows the base technology pretty well. There is still lot of new technology they will be working with.
- The current system is fairly complex and it won't be easy to migrate existing workflows to new system. Expected duration of this migration is around 2 years.
- Lot of stakeholders (including leaders, managers, potential users) will be impacted by this change. This means significant training, change management, stakeholder mgmt is needed to be successful.

EXAMPLE 4 - ANALYSIS

- Some unknowns and risks
- Since team has good domain knowledge and base technical expertise, it is not totally new space for the team.
- Medium size project
- Complex deployment and change management
- Architecture is key to setup the organization for future. Need some work to get this right.
- Looking at complexity and number of stakeholder, it will be beneficial to roll this out incrementally to get some feedback and tweak the system.
- Most of the projects in the Organization are managed using Traditional approaches and leaders prefer certainty, milestones and solid plans.



Agile Manifesto

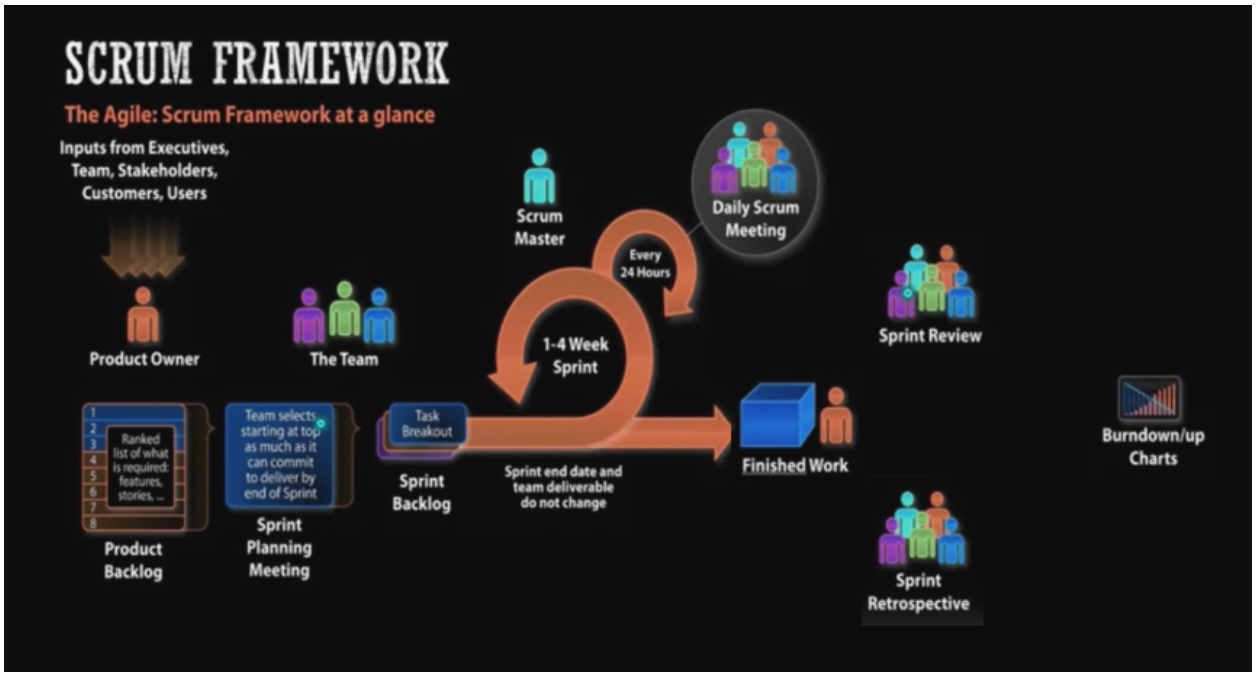
PRINCIPLES BEHIND AGILE MANIFESTO

1. Our highest priority is to **satisfy the customer through early and continuous delivery** of valuable software.
2. **Welcome changing requirements**, even late in development. Agile processes harness change for the customer's competitive advantage.
3. **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. **Business people and developers must work together** daily throughout the project.
5. Build projects around **motivated individuals**. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation.**

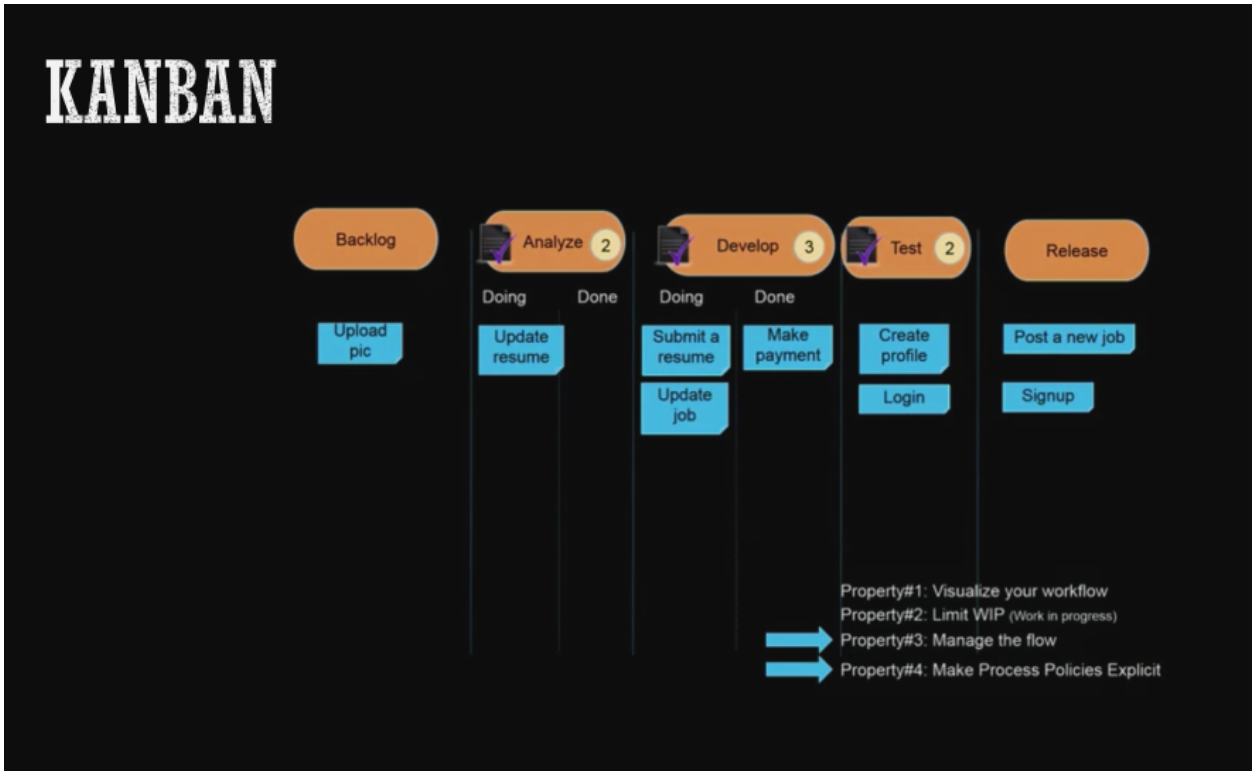
PRINCIPLES BEHIND AGILE MANIFESTO

7. **Working software is the primary measure of progress**.
8. Agile processes promote **sustainable development**. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to **technical excellence** and good design enhances agility.- cost of exploration
10. **Simplicity**–the art of maximizing the amount of work not done–is essential.
11. The best architectures, requirements, and designs emerge from **self-organizing teams**.
12. At regular intervals, the **team reflects** on how to become more effective, then tunes and adjusts its behavior accordingly.

Scrum



Kanban



KANBAN PRINCIPLES

1. Start with what you do know
2. Agree to pursue incremental, evolutionary change
3. Respect the current process, roles, responsibilities & titles

KANBAN PROPERTIES

1. Visualize the workflow
2. Limit WIP (work in progress)
3. Manage flow
4. Make process policies explicit
5. Improve Collaboratively

When to use scum (agile mindset)

AGILE AND LEAN

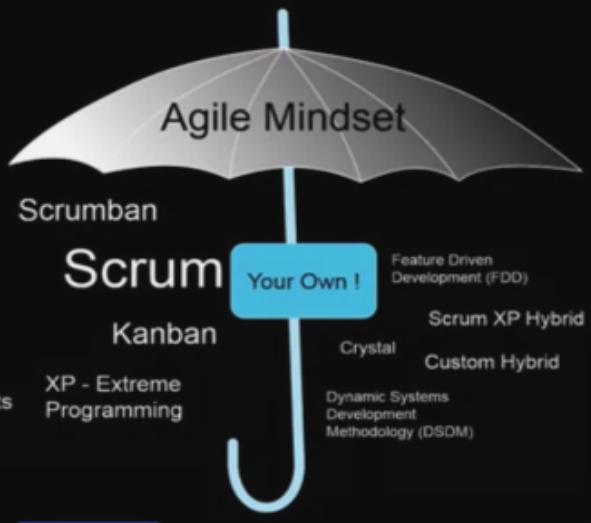
- Mindset not process/ model
- Incremental and Iterative
- Very short dev cycle

- Flexible to change increases chances of building the right product
- Speed to market

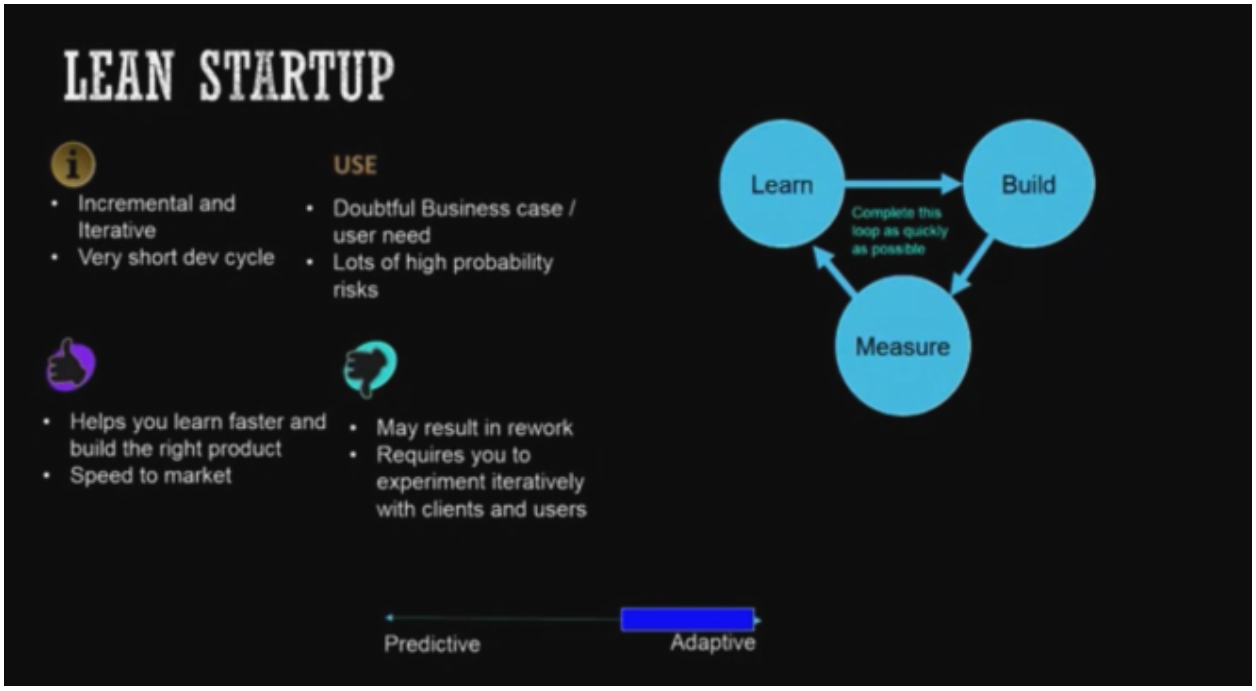
USE

- Where requirements may change
- Team using unproven technology

- May result in rework
- Requires close collaboration with clients and users



Predictive — Adaptive



Lecture 10 - 16

<to be pasted>

Lecture 17 - 30

Design Pattern

Q. What is a design pattern?
Ans. The solution to a set of common programming problems. Technique to make code more flexible. To achieve a particular purpose. A high-level programming idiom. Connects different components. Eg. Encapsulation, disadvantages of encapsulation - Interface may not provide all desired operations. Eg. Subclass
Problems

1. Reversibility (solved by inheritance)
2. Extensibility (solved by polymorphism)

Disadvantages - Reduce understandability, the overhead for dynamic dispatch

Types of patterns

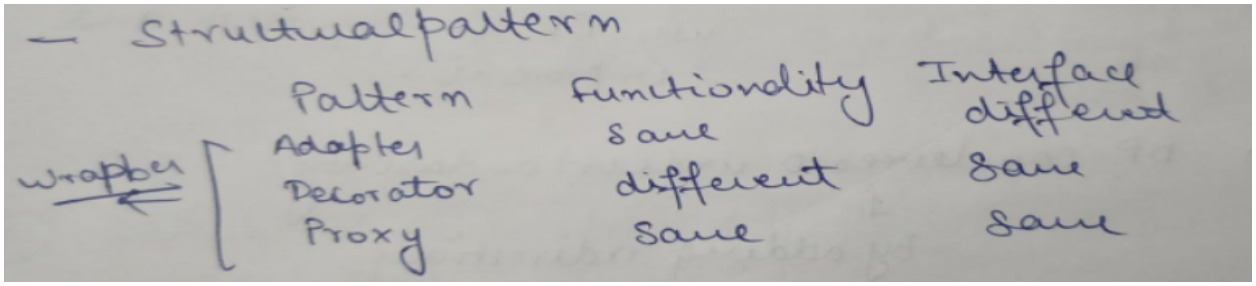
1. Creational Pattern - How do we create objects, where do we create objects, how many objects do we create, and should we clone an object or create a new one. Eg. Abstract Factory, builder, Factory methods, Factory Object, Prototype (clone object), singleton (single object).
2. Structural Pattern - How system components are related to each other. Eg. Adapter, Bridge, Composite, Flyweight, Facade, Proxy, Decorator.
3. Behavioral Pattern - iterations between objects. Eg. Chain of responsibility, Iterator, Command, Observer, Mediator, Visitor, Null Object, Template Methods.

Q. When to use design patterns?
Ans. Delay - Don't use design patterns prematurely. The best approach is to implement something, ensure it's working, and use a design pattern if required to improve it. Design patterns can decrease understandability in the following ways: By adding indirections. An increasing amount of code. Design patterns can increase understandability in the following ways: Improve modularity, better separation, and easier description.

Creational Pattern

1. Factory - Creates objects by some create methods (described previously)
2. Prototypes - Clone methods
3. Observer Pattern - Maintains a list of states which need to be notified. Add register and remove methods. ‘Push’ structure sends all the information. ‘Pull’ structure simply sends the notification whenever a change occurs.

Structural Pattern



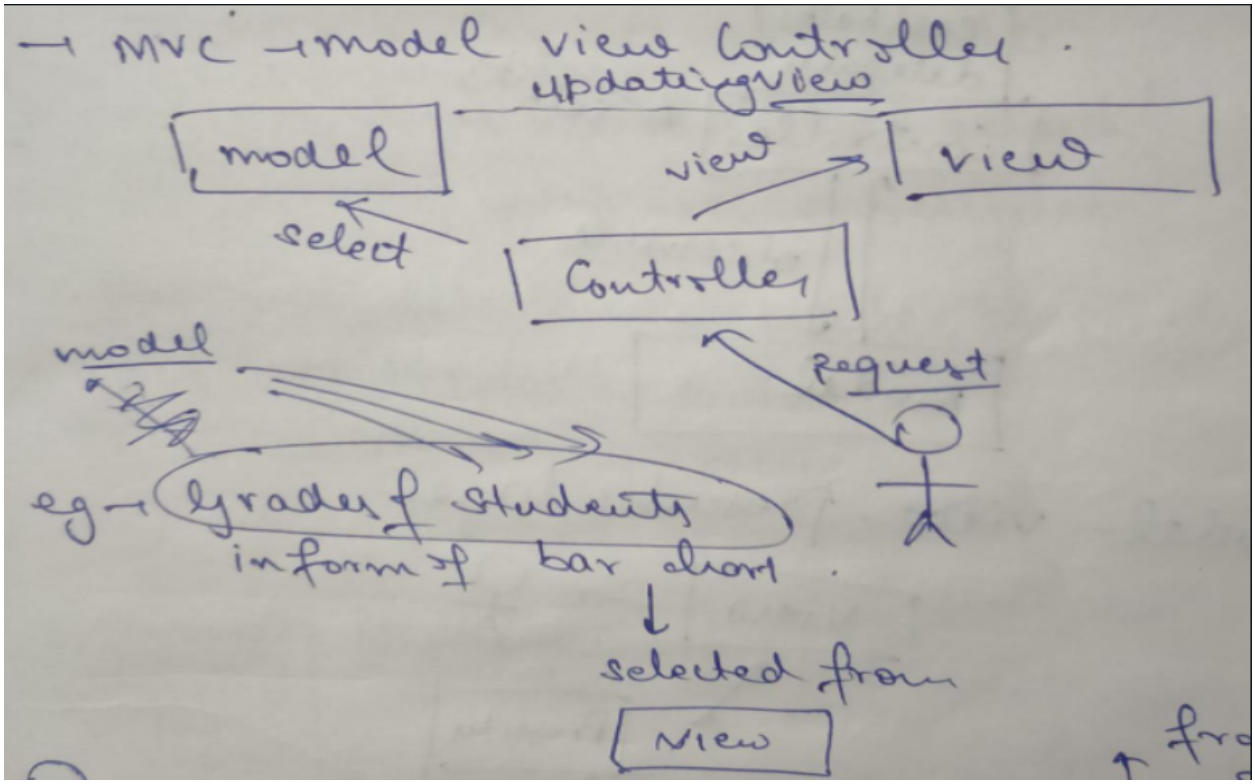
Dependency Injection (DI) - Related to inversion of controls.
Interface resolved in run time.
CRUD - Create, Read, Update, Delete.
Two ways to inject:

1. Through configuration file
   <inject src = “Istore”>
   implement -> “student oracle”
2. Service locator close

UI Design patterns

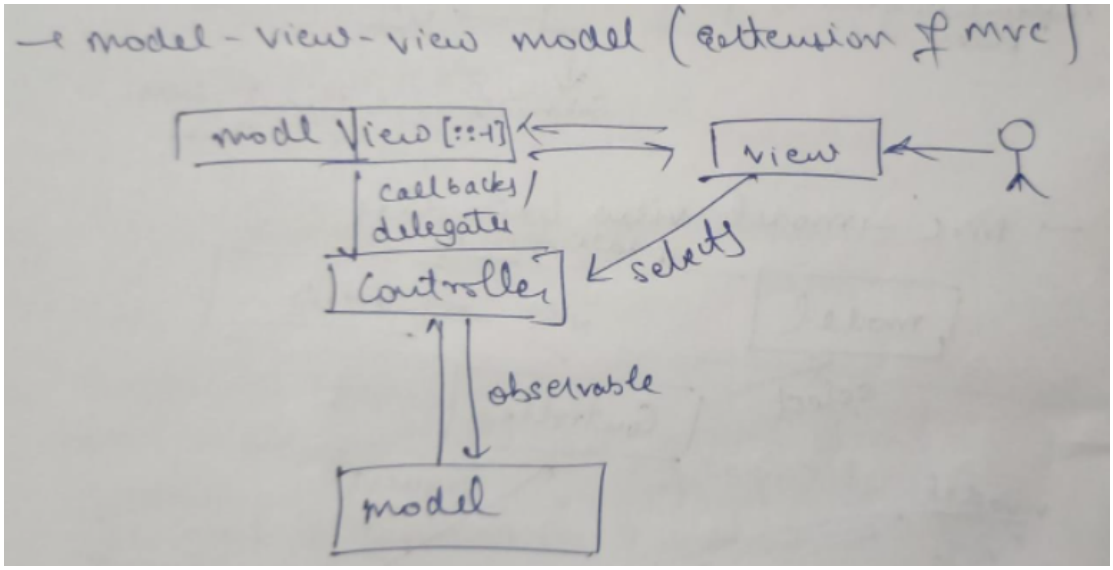
1. 3 layers architecture
   ..CRUD [ ] Data Access layer
   ..Checking constraints [ ] Business logic layer
   Implementing business rules
   ..Submit forms [ ] User interface (UI)
   Client side checks

2. MVC (Model View Controller)

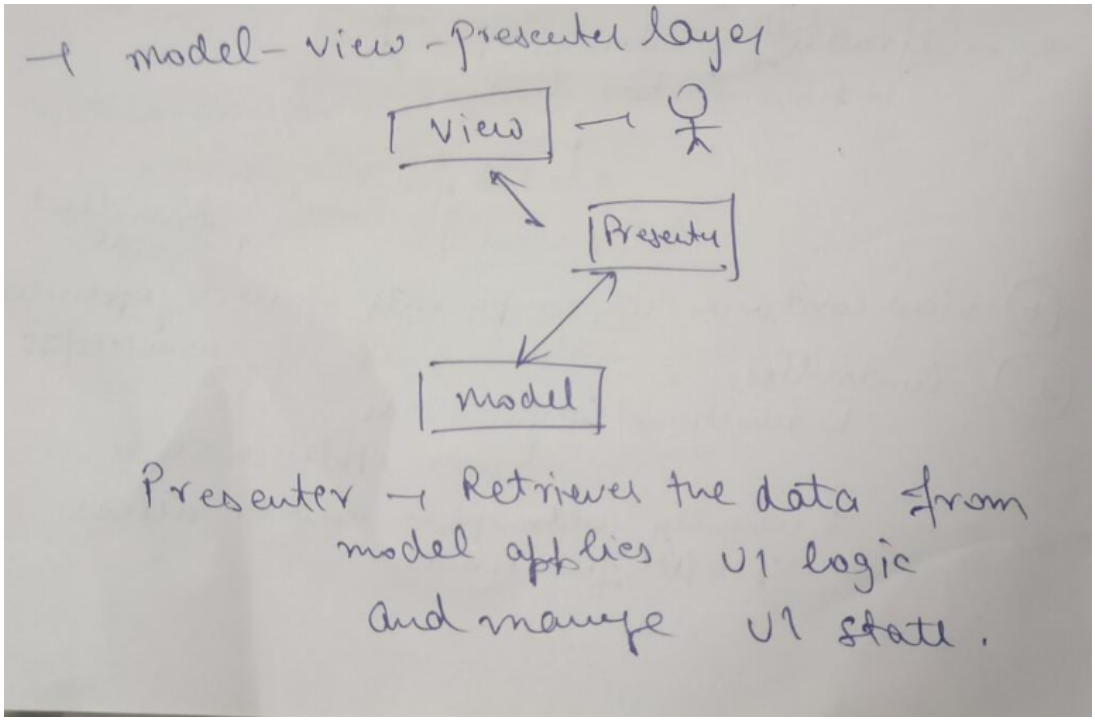


View contains UI components (HTML/CSS/JavaScript)
Controller functions/compile code (Web application's server), usually, a URL mapper is used to access specific functions.
Model - Equivalent to an entity, store the data in runtime, encapsulate application states, handles state quires, and updates, and exposes application functionality.

3. 'Model view view' model (Extension of MVC)



4. ‘Model view Presenter’’ layer



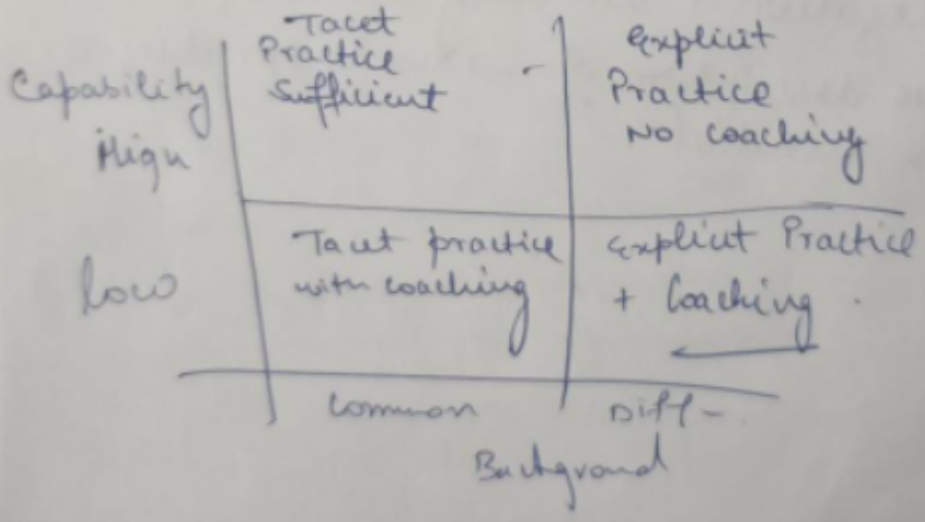
Kernel (Essence)

Kernel → Essence

idea
- methods are composition & practices
- → There is a common ground (Kernel) among all the methods.
- → Focus on essential needed
- → Engaging user-experience

How much details we need to have while defining practices?

1. Capability of the team
2. Background of the team

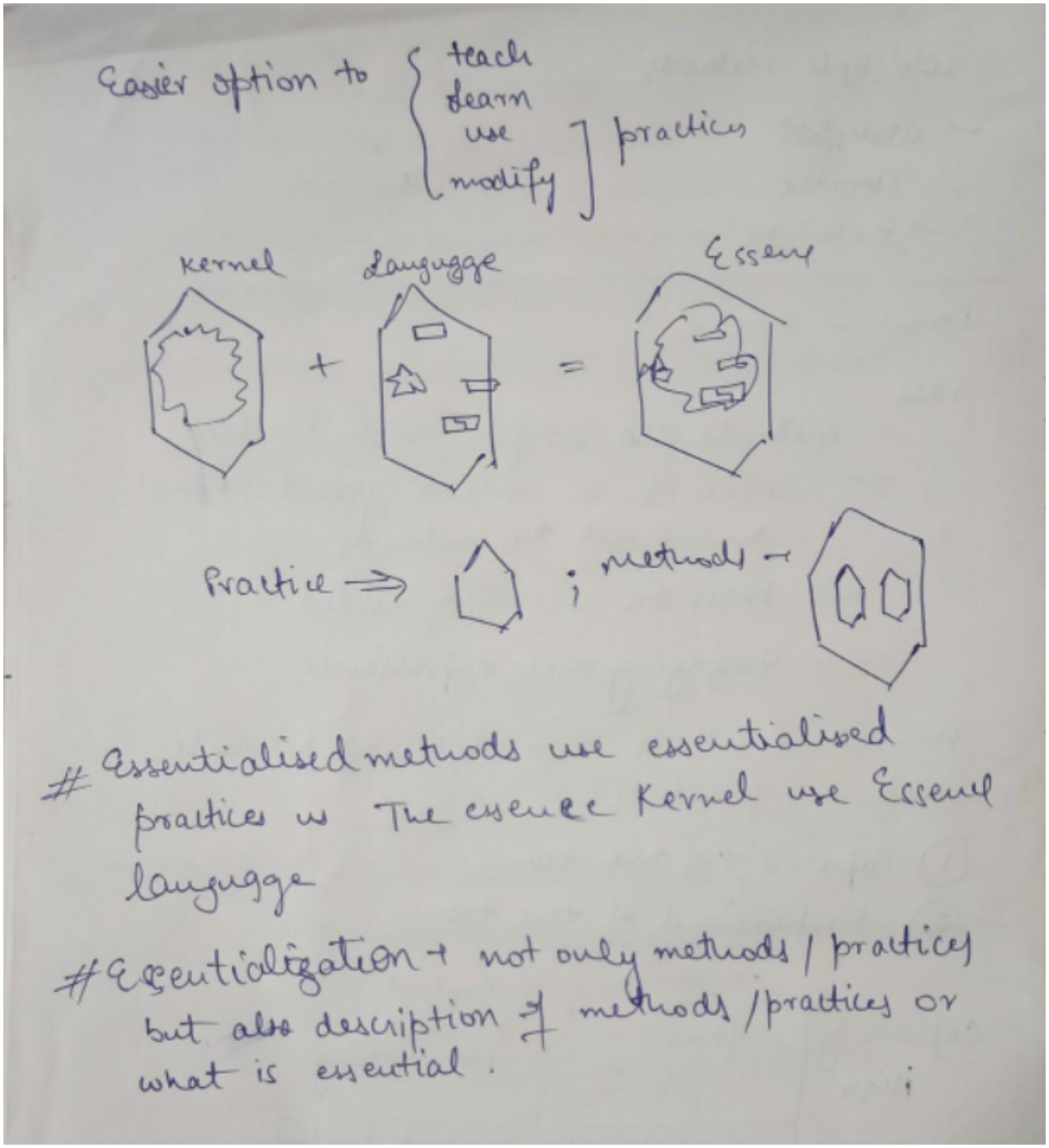


Life Cycle methods:

1. Waterfall
2. Iterative
3. Prototype

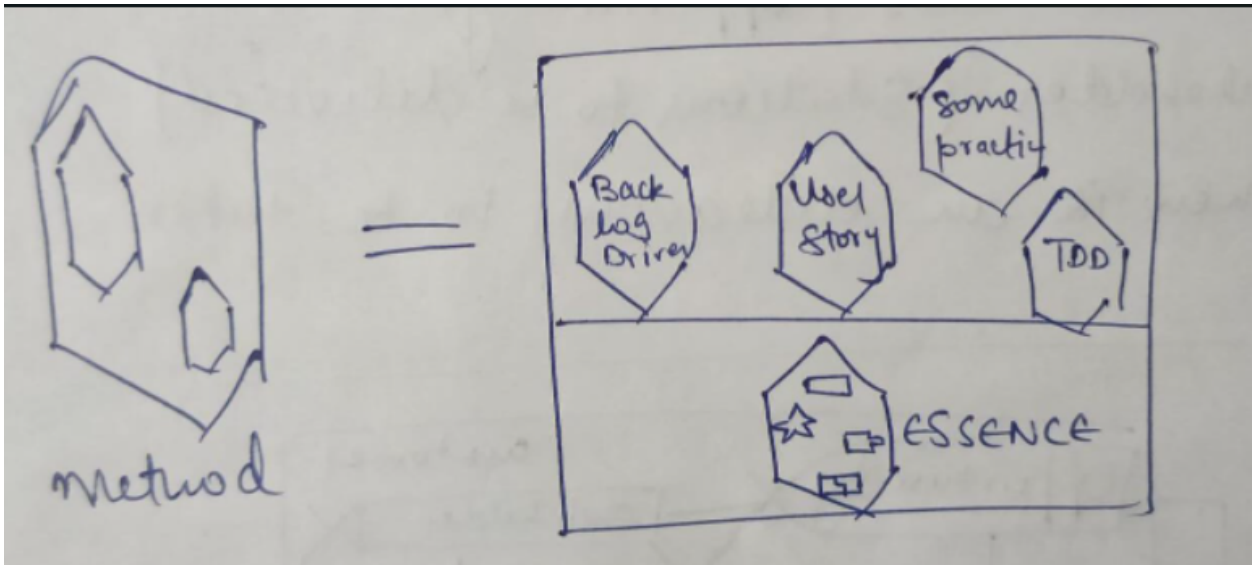
Essence Diagrams

Define Essentialization, Essentialized methods



Easier to build our own methods based on requirements.

1. Building new practices with the essence
2. Mix practices from the library.
   Method Diagram



3. Focus only on essentials
4. Engaging user experience - Essence provides a hands-on, tangible user experience, and supports software professionals.

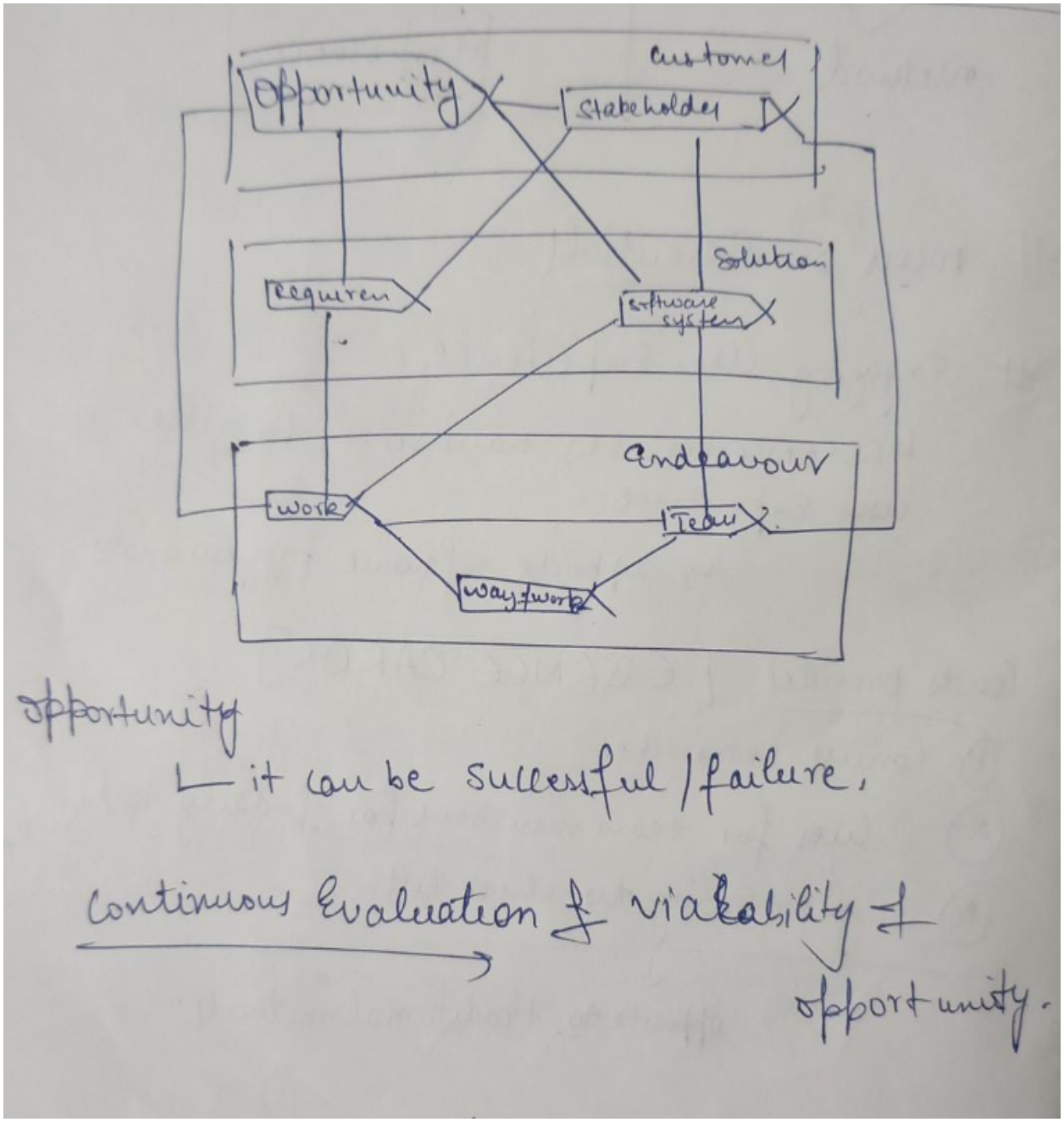
Benefits of Cards:

1. Concise Reminder
2. Cues for team members of daily tasks
3. Provides the checklist opposite to traditional methods
4. Cards increase the understandability
5. Helps the kernel to be more digestible

Kernel dependency

1. Kernel depends on customer needs (Opportunity)
2. Stakeholders (Solutions to be delivered)
3. Endeavor to be taken

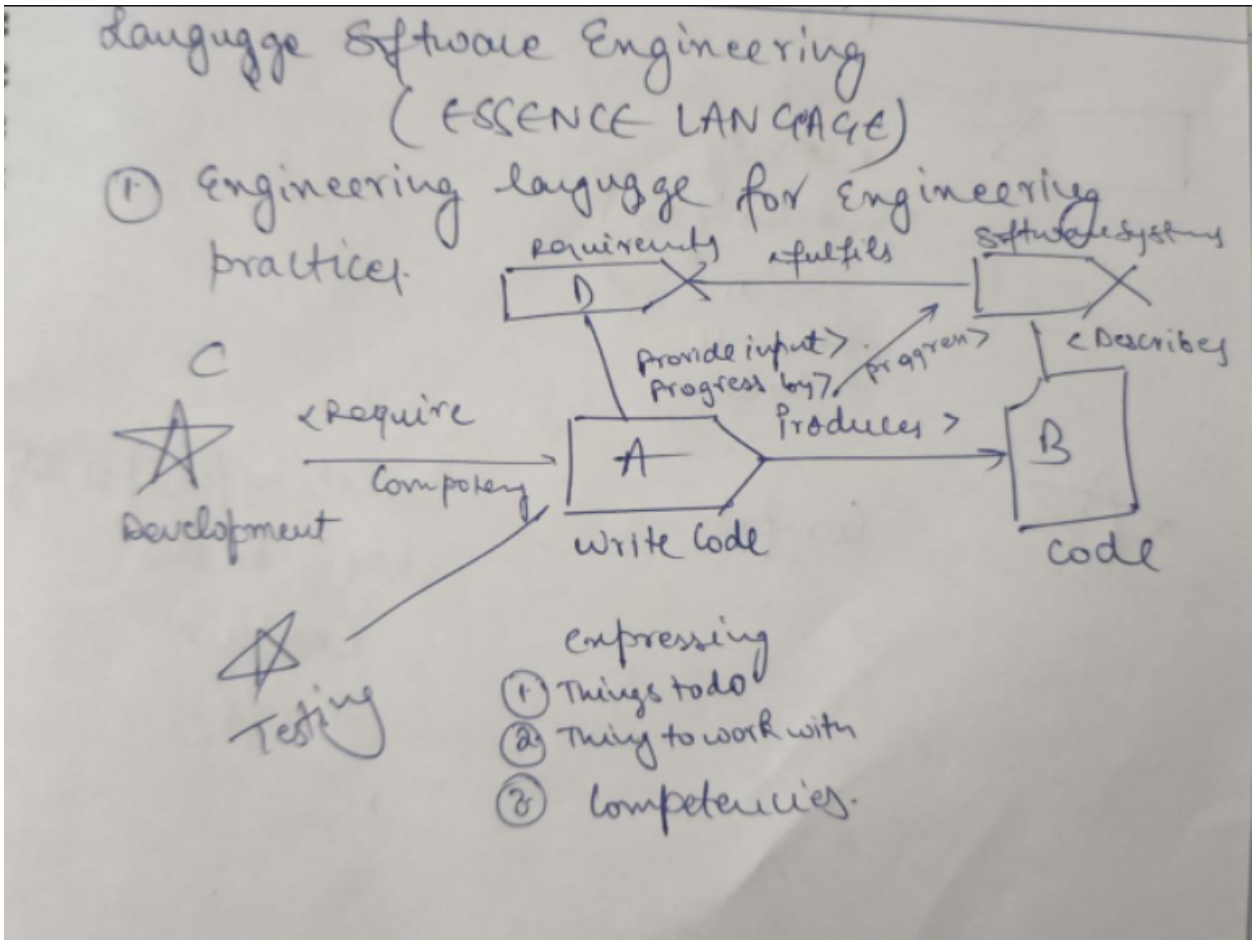
Diagram showing dependency



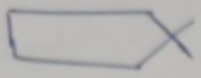
Requirements good software

1. Functionality -> Requirements
2. Quality -> Context Dependent -> Requirements
3. Extensibilty -> Functionality -> Expressed through requirements

Essence Language



Diagrams



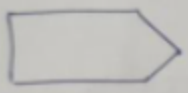
Alpha
intangible

- Things we want to track.
- Health and progress of the endeavour.

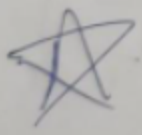
work product
Tangible

produce from diff. activities of SE Endeavour

Req → SRS
↑ work product
Alpha.

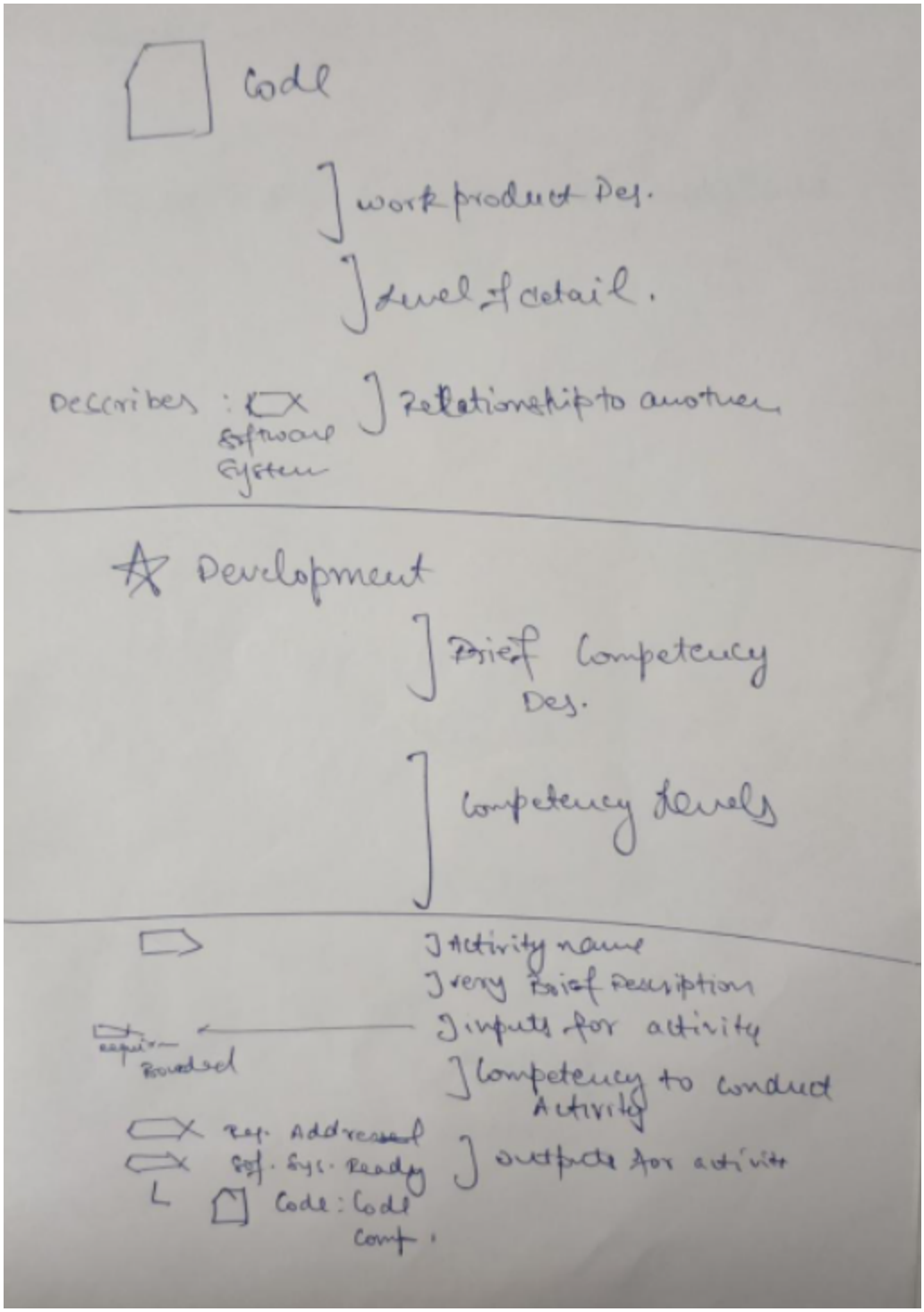


Activity
(Something to do
like write code)

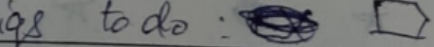
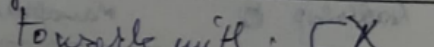
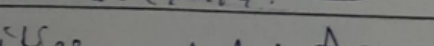
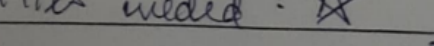


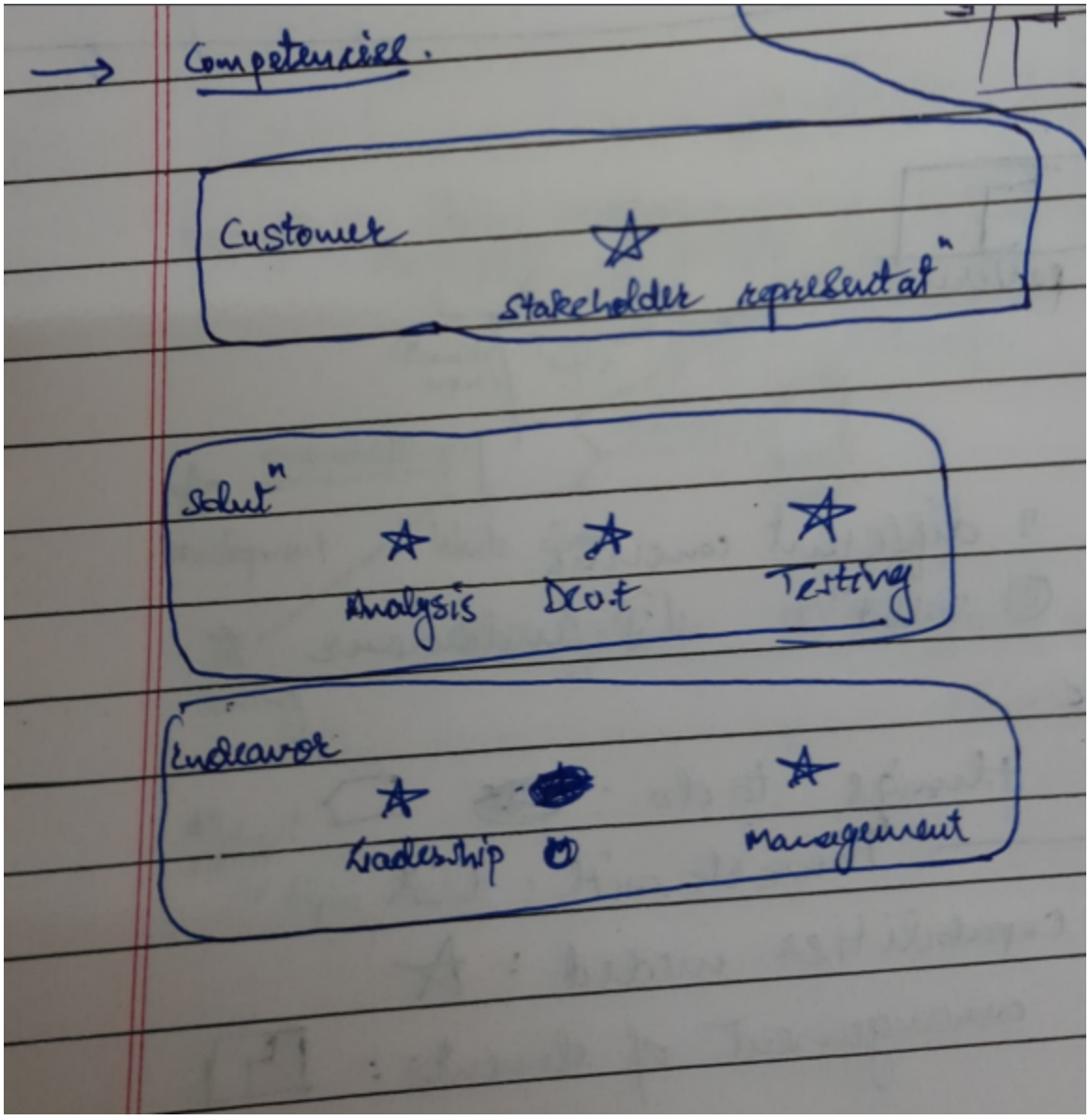
Competency

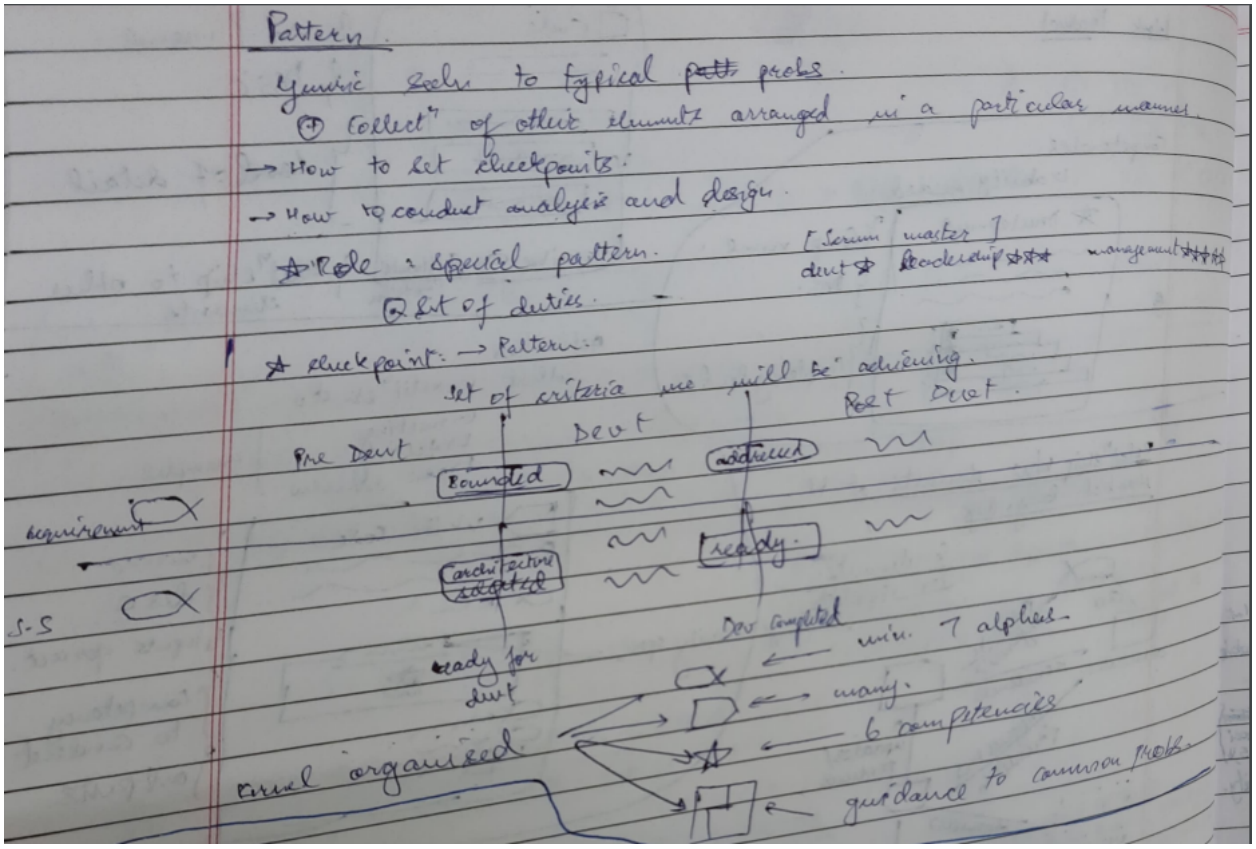
ability / capability req. to do certain things.


Code
work product Des.
level of detail.
Describes :
Software System
Relationship to another
Development
Brief Competency Des.
Competency levels
Activity name
Very brief Description
Inputs for activity
acquire-Bounded
Competency to conduct Activity
Req. Addressed
Soft. Sys. Ready
Code : Code Comp.
outputs for activity

Kernels

- organised within 3 different concerns
  - ① customer ② solut^{n} ③ endeavour
- 4 kinds of elements.
  - ① The essential things to do : 
  - ② —11— towards with : 
  - ③ —11— capabilities needed : 
  - ④ —11— arrangement of elements : 


Competencies.
Customer
Stakeholder representation
Solution
Analysis
Dev.t
Testing
Endeavor
Leadership
Management


Pattern
Generic refers to typical probs.
"Collect" of other elements arranged in a particular manner.
→ How to let checkpoints.
→ How to conduct analysis and design.
Role : special pattern.
Set of duties.
[Scrum master]
dent leadership management
checkpoint: → Pattern:
Set of criteria we will be achieving.
Pre Dest
Dest
Post Dest.
Bounded
abstract
requirement
Architecture adapted
ready.
S-S
ready for dest
Dev completed
min. 7 alphas.
many.
6 competencies
guidance to common probs.
Proud organised

Games with cards.

→ Serious games (called) Purpose :- • Entertainment • Team integrity. • Achieve goals.
↳ Cooperative (not competitive) • Skill development opportunity (Percept", attentiveness, decision making)

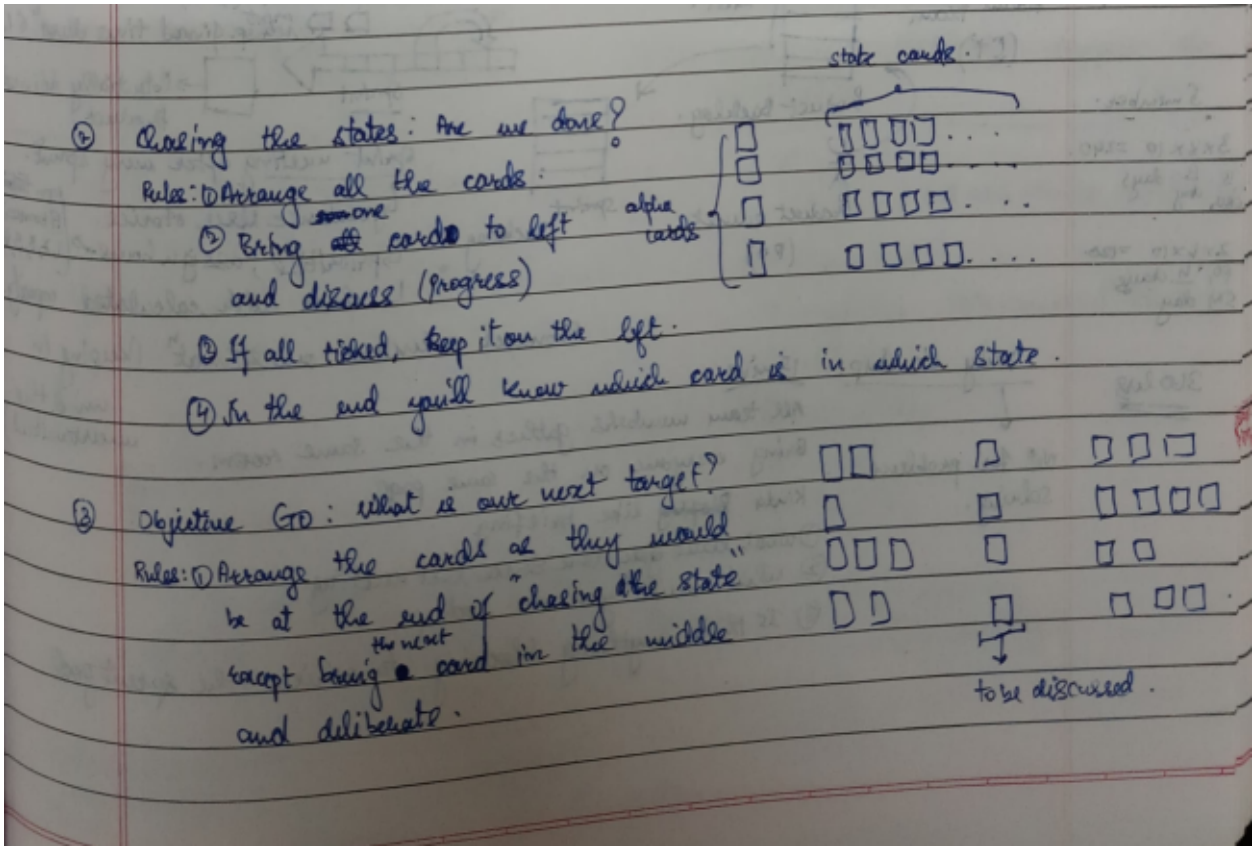
① Progress Poker ② Chose the statel ③ objective go ④ Checkpoint count.

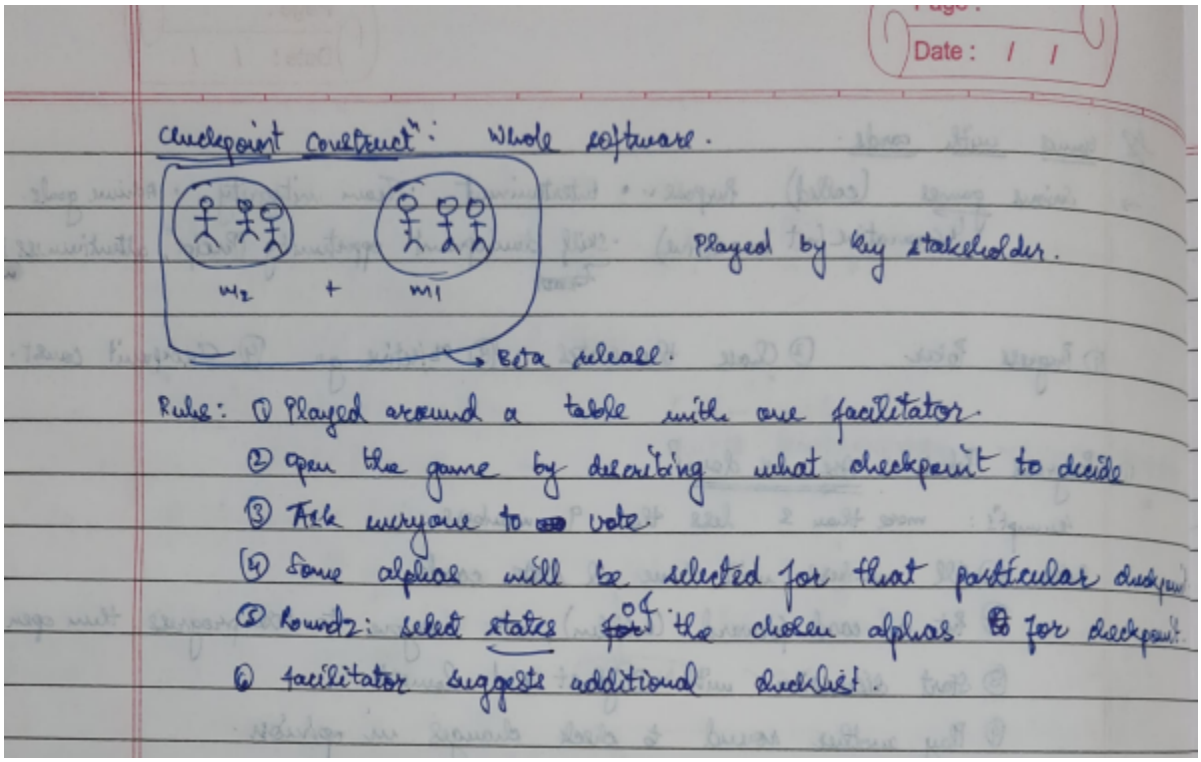
① Progress Poker : Are we done?

Prompt's : more than 3 less than 9 members.

Rules: ① All members will have all state cards.
② Put a card forward (hidden), ask everyone to tell progress. then open.
③ Start discussion with highest and lowest.
④ Play another round to check changes in opinion.

Benefits: ① Everyone is forced to participate. Since everyone will be forced to think on every aspect.
② Explain thinking ③ Increase understanding
④ Ensure checklist items completed.





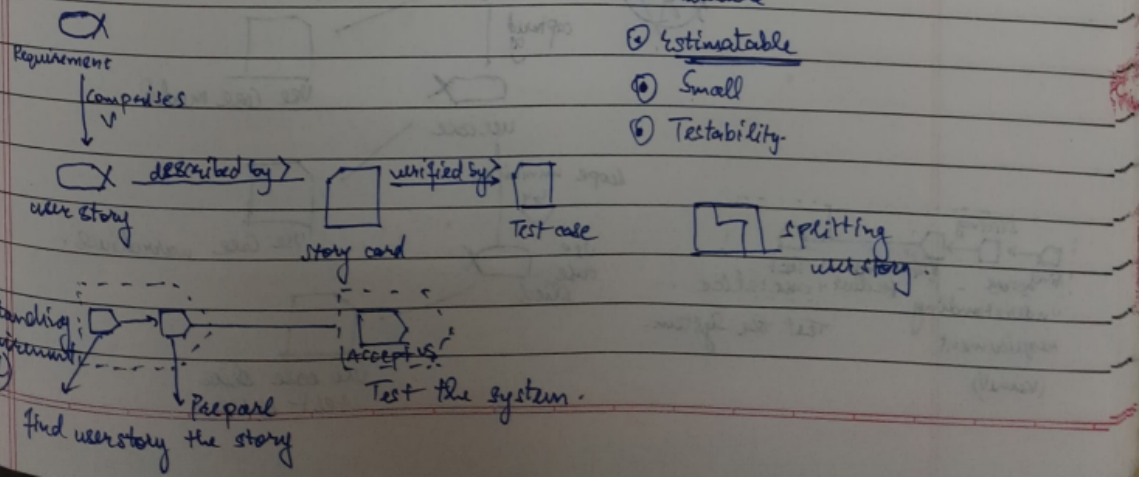
Example Essence

team.

→ User stories through essence.

As a <role, type of user>
I want to <list of funct°>
so that <list of objective>

INVEST (all user stories should follow)
- Independent.
- Negotiable (like conversat° starter)
- Valuable
- Estimatable
- Small
- Testability.



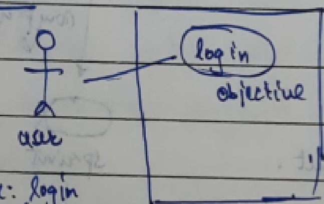
Test Case
↳ Acceptance criteria captured. (parameters etc.)

Scripted
↳ Retain a test script. Tests the software.
not everything can be scripted.

Automated
↳ scripts can be run manually / automated

LITE

Use Case



use case model.

use case: login
Actors: Student, teacher System
pre-condit^n: account created

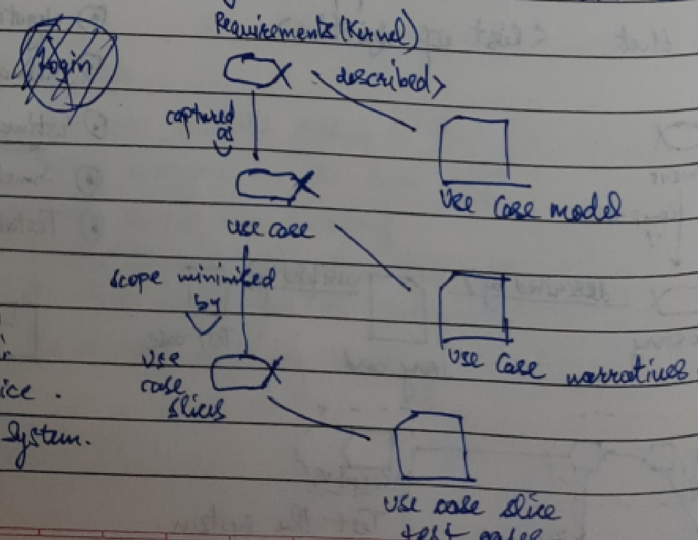
post condit^n: My account opens.

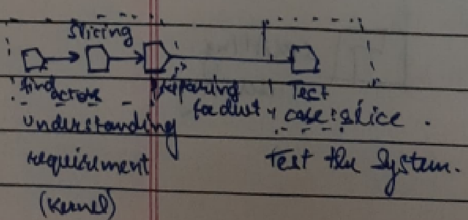
Basic flow: 1. User enters username
2. —"— password
3. System verifies.
4. Land on account page.

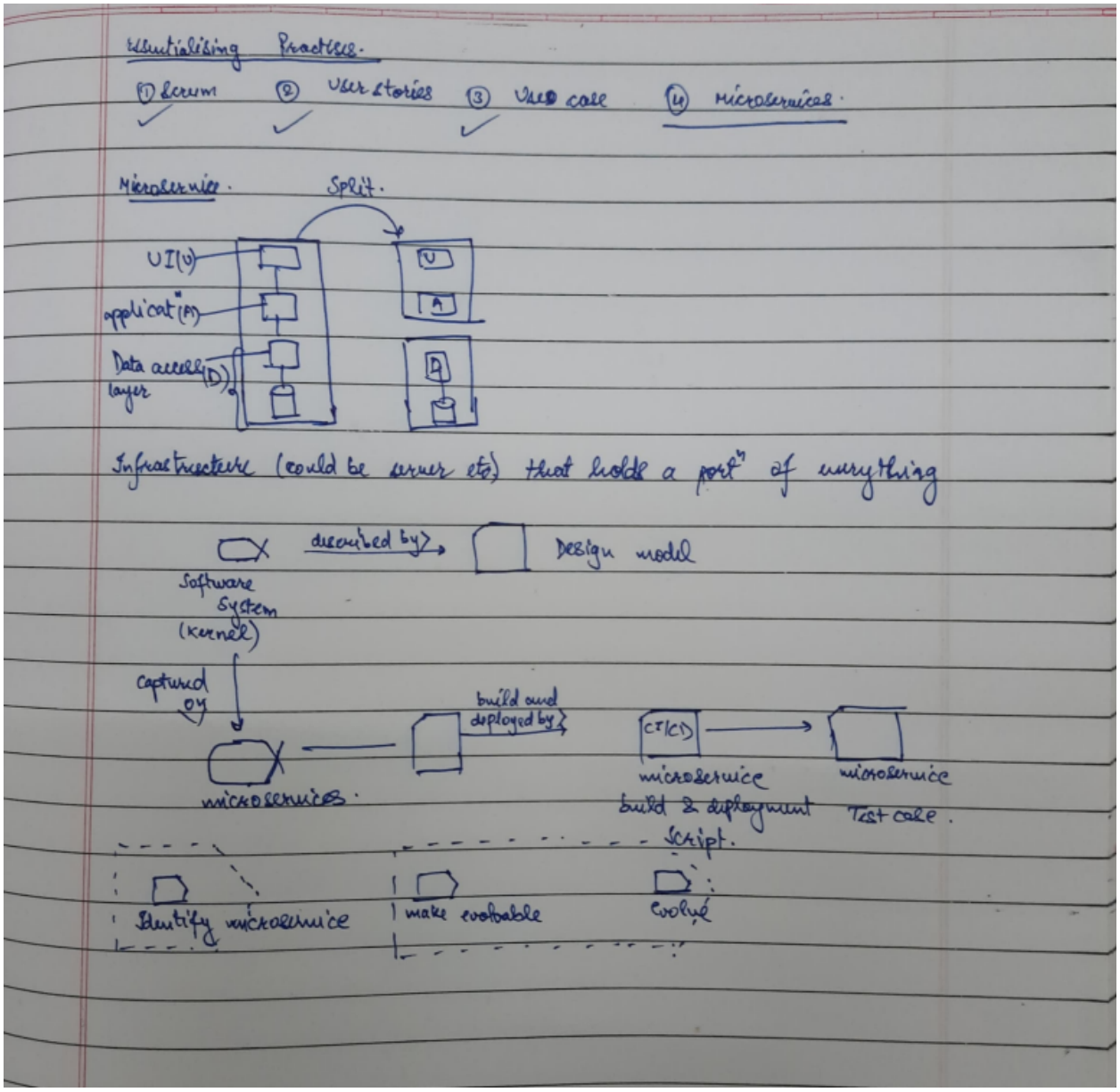
Alternate flow
A1 : Password incorrect
A2 : username incorrect.

user story: For experienced team, error tolerance is there.
Use Case: Even low experience can understand, no error tolerance

→ Slicing Use case: Breaking down a use case into smaller use c







Lecture 31 -38

Software Engineering

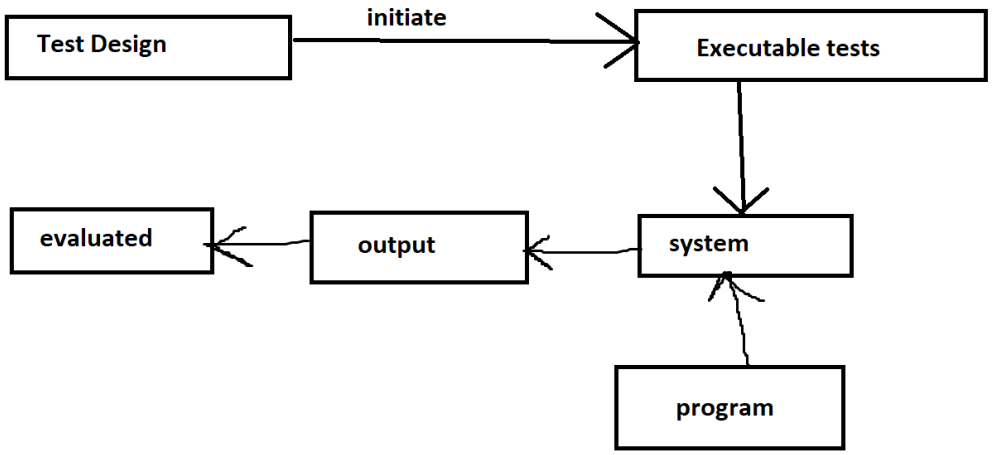
Class 31: Testing

Software needs to be tested to find the errors in it.

Testing is pretty costly, it takes up about 50% of the software development life cycle.

Automated testing is needed when the software is going to be regularly updates, otherwise manual testing is sufficient.

Unit testing is the role of a Software Developer. We develop unit tests first before writing the function. We write the test case, and then write a function to satisfy the test case. This is known as test-driven development and for this purpose developer needs to be a tester as well.



Test cases are like: for one variable (input) x we can pass an empty list (one test case), or we can pass a string (another test case), or an array of integers (another test case), etc.

Classification of Tests

- Testing levels based on Software Activity:
    - Acceptance Testing — This is the final test on the whole system which is accepted by the client

- System Testing — It tests if the system when replicated on another machine, can function properly or not. This way the errors that can arise due to communication/configuration challenges. It is against the architectural design.
- Integration Testing — We integrate different modules and test them.
- Module Testing — It is Detail Design. Here we test the modules to create stubs.
- Unit Testing — It is against the Code file. This is usually done by the developer.

The order of these testing levels is bottom up. We first test the code of one module, then test different modules, then integrate different modules and test them and so on.

Beta testing — when we do not have a definite client then we can go with beta testing.

> Regression Testing — This is needed for maintenance. When we fix some bugs, then we need to test that the software works as intended.

- Testing levels based on Test Maturity
  - Level 0 — no difference between testing & deploying. This is simple testing of the code by providing different inputs and seeing if the output is as expected or not. We do not usually do exhaustive testing, n or do we do it in a systematic manner.
  - Level 1 — Testing done to show that software is working. We need to do it with systematic test cases BUT we do not do exhaustive testing. Our goal is to show that for a general input, the software works, i.e. there MAY be test cases for which the software does not give correct output.
  - Level 2 — Here we try to explore the possibility of the software not working, i.e. some test case for which the software breaks or crashes or has bugs that need to be identified.
  - Level 3 — To reduce the risk of using the software (domain specific like healthcare, flight system, etc). Even if the software goes into a bad state then the software shouldn’t be affected too much.

    - Level 4 — It is related to mental state that produces high quality software. **(not really part of the course)**

Validations and Verifications

Validation — “Are we building the right product?”. Product should comply with the requirements. Happens at the end of the product’s life cycle. Depends on the domain knowledge. It is against the domains.

Verification — “Are we building the product right?”. It is against the requirements.

Faults vs Errors vs Failures

- Software Faults — A static defect in the software.
- Software Errors — An incorrect state in the software. It need not be reflected in the output, i.e. incorrect behaviour.
- Software Failures — It is incorrect behavior.

> Coverage is how many test cases we have tried for testing our code

Testing

It is evaluating software by observing executions

Test Failure

Execution that results in failure

Debugging

Process of finding the fault by looking into the failure

Bug report

- What is the failure?
- How to reproduce this failure?
- System Info on which the failure occurred

R.I.P

R ⇒ Reachable. Code must be reachable where the fault is occurring.

I ⇒ Infection. It should be in an error state.

P ⇒ Propagation. If the software is an error state then it should be propagated to the output.

- These three properties must be there to identify the failure.

Input to a Test Case is called Test Value and the expected output is called Expected Value.

Prefix value is needed to bring the software into a certain state, it is required to start execution. E.g. user login.

Postfix value is what happens to the output once execution is finished.

Test Case is a combination of test values, prefix values, postfix values and expected values.

Test Set is a set of Test Cases. We need this to have the entire code *coverage*.

Class 32: Graph Coverage Criteria

How to generate the test sets?

Test Requirements

Specific elements of a software artifact (code file, design file, whole software, etc). These requirements must be satisfied by the test case. E.g. variable x needs to be an integer. Every test requirement need not be satisfied by every test case. Test requirements can also be a set.

Coverage Criteria

Impose test requirement on a test set. It dictates how to develop a test set from the test requirements. These criteria define that if these are the test requirements, then you can develop test set which can satisfy the requirements.

Coverage

Based on the coverage criteria, there should be atleast one $t \in T$ which satisfy some $tr \in TR$. Here, T is the test set, TR is the test requirements set.

Coverage Level

Coverage Ratio = $\frac{|tr_{\text{satisfied}} \in TR|}{|TR|}$, i.e. number of test requirements satisfied divided by number of test requirements to be satisfied.

Black box testing

We do not know anything about the internal working of the application. While testing we can only provide some input and receive output, without any knowledge of how the output was generated.

White box testing

We test the internal information/source code of the software.

Static testing

We do not run the code

Dynamic testing

We run the code with real inputs

Graph Coverage

$N = \{\text{nodes}\}; N_o = \text{set of starting nodes}, N_o \subseteq N; N_f = \text{set of final nodes}, N_f \subseteq N; E = \{\text{edges}\}$

Each subgraph of a graph has its own set of N_o and N_f. All graphs considered here are **directed graphs**.

Test path is the execution of the test case. E.g $n_0 \rightarrow n_1 \rightarrow n_2$

Reachability: (syntactically) a node n (or edge) is reachable from n_i if there exists a path from n_i to n. (semantically) if it is possible to execute atleast one of the paths with some input.

Coverage Criteria

- Structural Graph Coverage Criteria (Control Flow)

- Data Flow Coverage Criteria

For this course we will focus on the first one

Structural Graph Coverage Criteria

Test requirements:

1. Visit a node
2. Visit an edge
3. Touring a particular path

Coverage Criteria:

1. Node Coverage Criteria — TR should contain each reachable node
2. Edge Coverage Criteria — TR contains each reachable path of length upto 1 inclusive in G (edge)

SOFTWARE TESTING PERSPECTIVES

Black-box and White-box Testing

Black-box Testing
- Designed without knowledge of program's internal structure
- Based on functional requirements

White-box Testing
- Examines the internal design of the program
- Requires detailed knowledge of its structure

AXIOMS OF TESTING

As the number of detected defects in a piece of software increases, the probability of the existence of more undetected defects also increases

Assign your best programmers to testing

Exhaustive testing is impossible
- You cannot test a program completely
- Even if you do find the last bug, you'll never know it
- It takes more time than you have to test less than you'd like
- You will run out of time before you run out of test cases

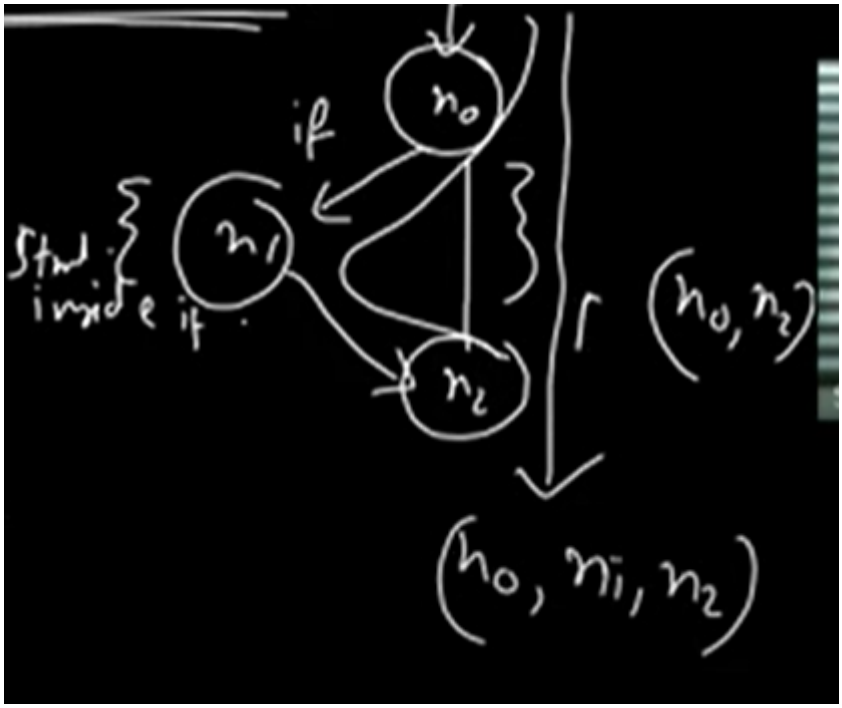
maximum length simple path

→ prime path, no need to test all simple path if it is a subset of another path

Definition → A path from a to b is a prime path if it is a simple path and it does not appear as a proper sub graph to any other simple graph.

- Criterion 3 → Prime path coverage (PPC)

→ TR contains each prime path in G.

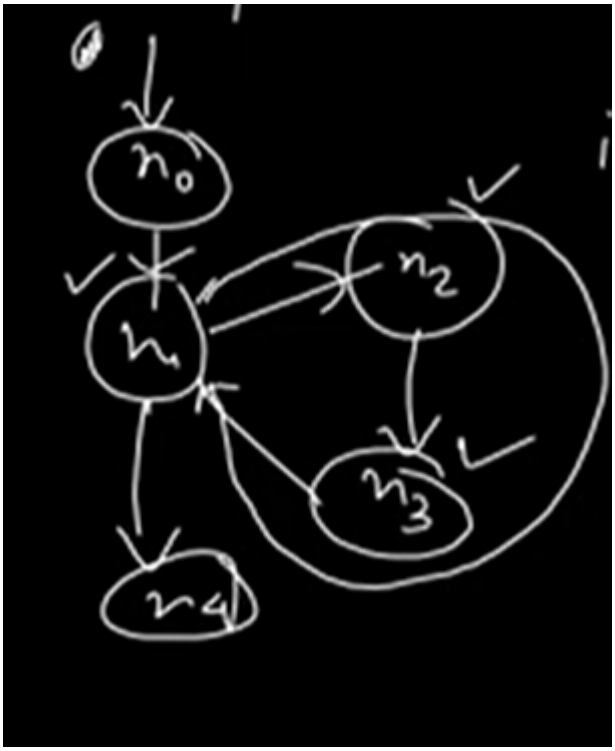


Two prime path in above figure (n0, n2) and (n0, n1, n2)

Loops

→ Round Trips → prime path of non zero length that starts and ends in single node.

Example → done below



→ Simple round trip coverage (SRTC)

- TR contains at least one round trip path for each reachable node in G that begins and ends a round trip path.

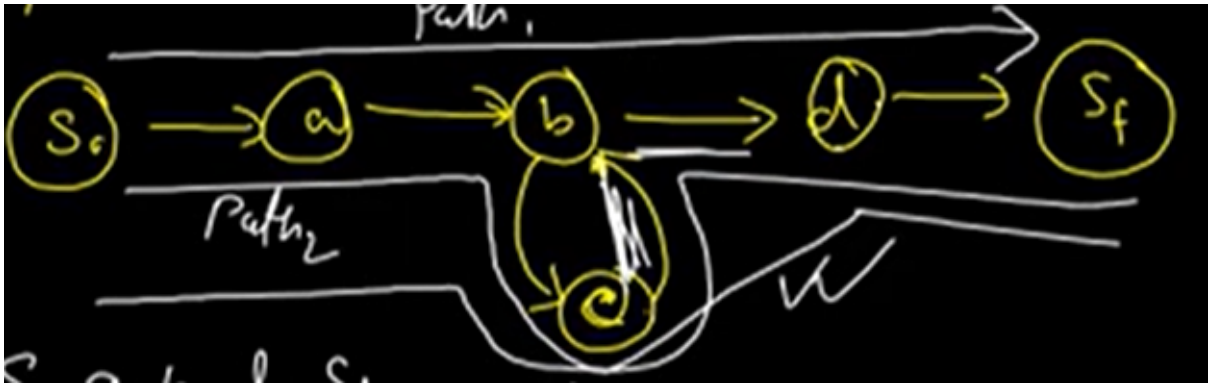
→ Complete Round Trip (CRTC)

- TR contains all round trip paths from each reachable node in G.

→ **Impratical** → **Complete** Path coverage (CPC)

Tour with side trip

→ can go direct So to Sf (straight) and also go from So to Sf via node c, but all the basic requirements needs to be installed, so prime path should be followed.



→ node : represents a block of code,(sequence of lines executed).

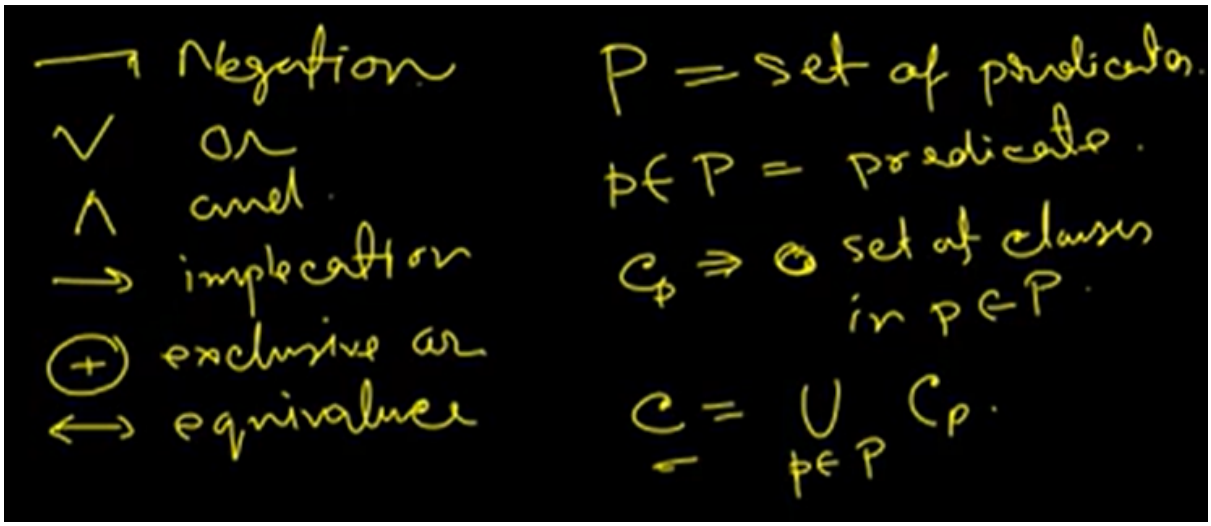
→ if else statements are like unweighted directed graph, while switch statements are weighted directed graph,

- Logic Coverage

  → Predicate : Expression that evalutes to boolean value.

  → clause : a predicate without logical operations.

  → notations



- Predicate Criteria

for each p belongs P , TR contains two requirements . Evaluate(p) ← True, Evaluate(p) ← False

- Clause Criteria

for c belonging to C , TR contains two requirements. Evaluate(c) ← True, Evaluate(c) ← False

More comprehensive criteria → Combinational coverage

→ for each p belonging to P, TR has test requirements for the classes in Cp to evaluate to each possible combination of truth table values.

p = (a v b) ^ c

draw truth table

→ better to have more clauses.

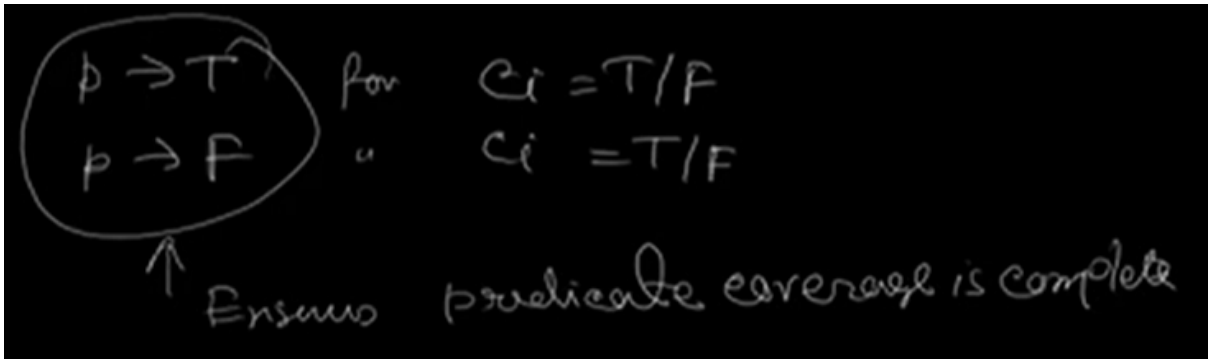
→ Let we have multiple clauses so we need to determine to main clause by keeping other clause as minor clauses.

so I need to determine value such that c1 determine p.

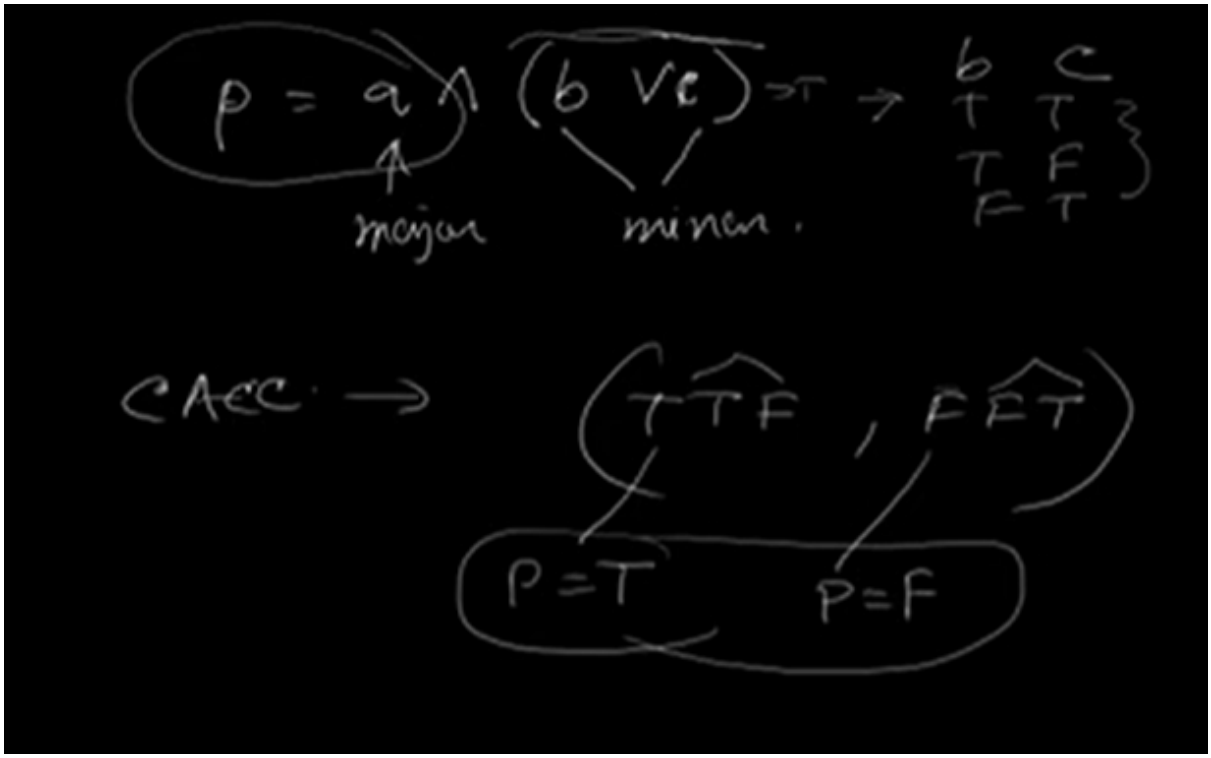
and this is **Active clause coverage (ACC) → for each p belonging to P and major clause ci belongs to Cp, choose minor clause Cj so that Ci determines P, TR has two requirements for each ci , evaluating to true/ false.**

Three different version of ACC:

- General ACC (GACC)→ the value chosen for minor clause(Cj) do not need to be same for Ci = True and Ci = F.
- Correlated ACC (CACC)→ value of minor clause Cj must set p to true for a value of Ci, and p to false for a value of Ci.



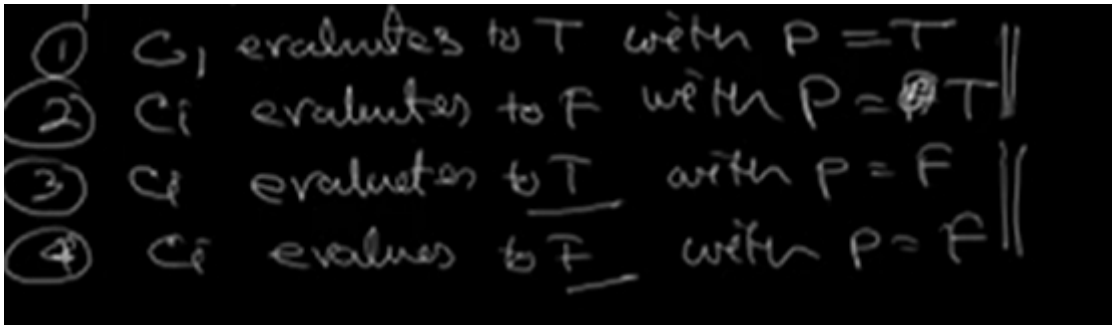
example →



- Restricted ACC (RACC) → must be same for both Ci = True or Ci = False.

→ CACC/ RACC → subspace predicate coverage.

- Inverted ACC(IACC) → choose cj such that ci does not determine p. Then we have a TR requirement where



Challenges → How to set the variables

→ Reachability → evaluating local variables in terms of global variables/ user inputs.
→ prefix in test case.

Input Space Partitioning

→ used in manual testing

graphs and logics → automated testing.

for each parameter determine the domain → domains = blocks → combine for all parameters.

- Two properties of the input space are:

→ Disjoint

→ Cover the entire domain.

General Strategy

- At least one valid value likewise add(int a, int b) a = 5, b = 7
- At least one invalid values like a = 5.5 b = 3.7
- use sub partitioning

  → likewise checking 0 - 100, 100 - 150
- Boundary values → most critical
- missing pattern → union of the partition is covering the domain.

- normal use

Test driven development

Testing in agile world

→ developing feature 1, 2, 3 doing testing on each traditional approach

→ In agile test-fast

- testing starts as soon as development
- write code, write test case
- developer and testers work together
- agile focuses on automatic testing → automatic integration → unit testing → regression testing

Agile testing principles

- Test provide feedback

  if testing started late then late feedback

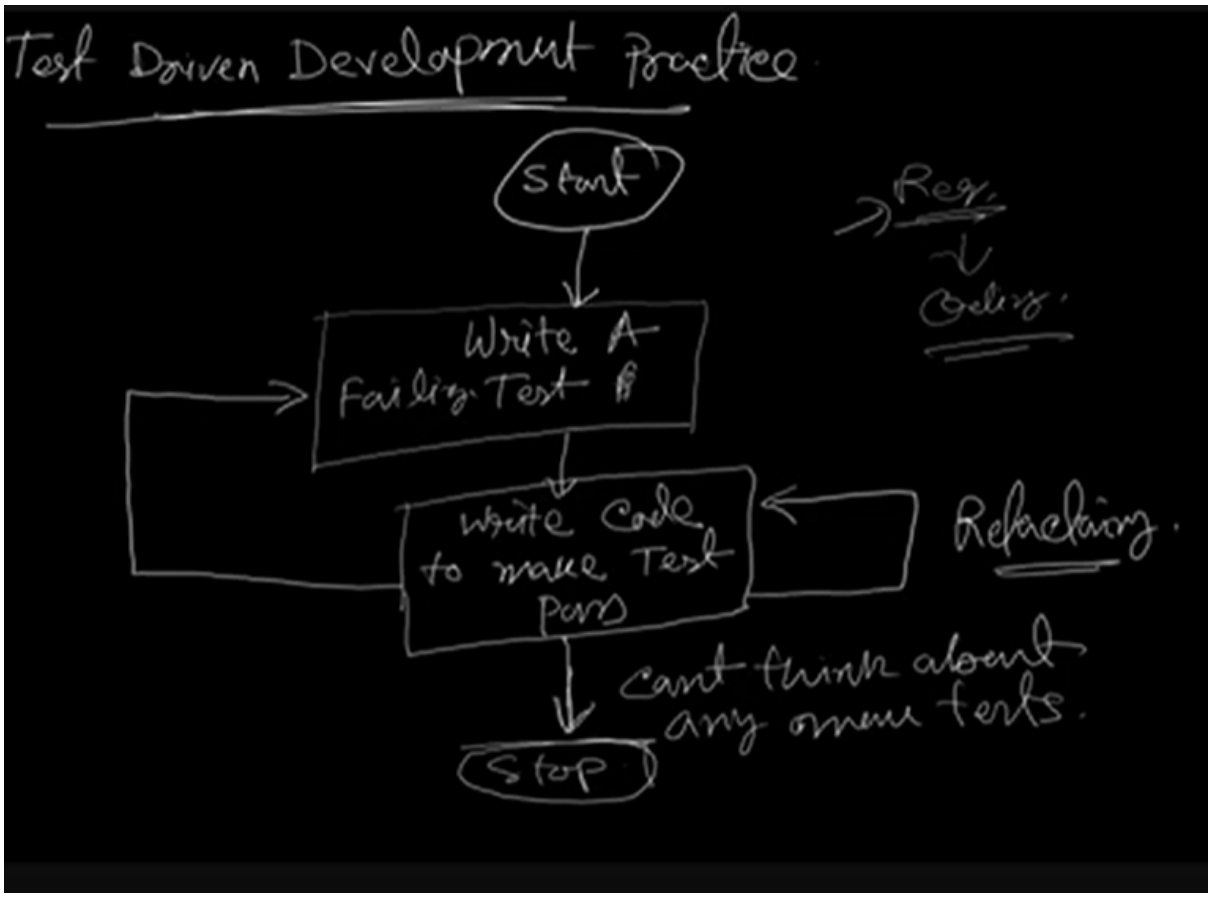
- continuous testing

  frequently during the sprint

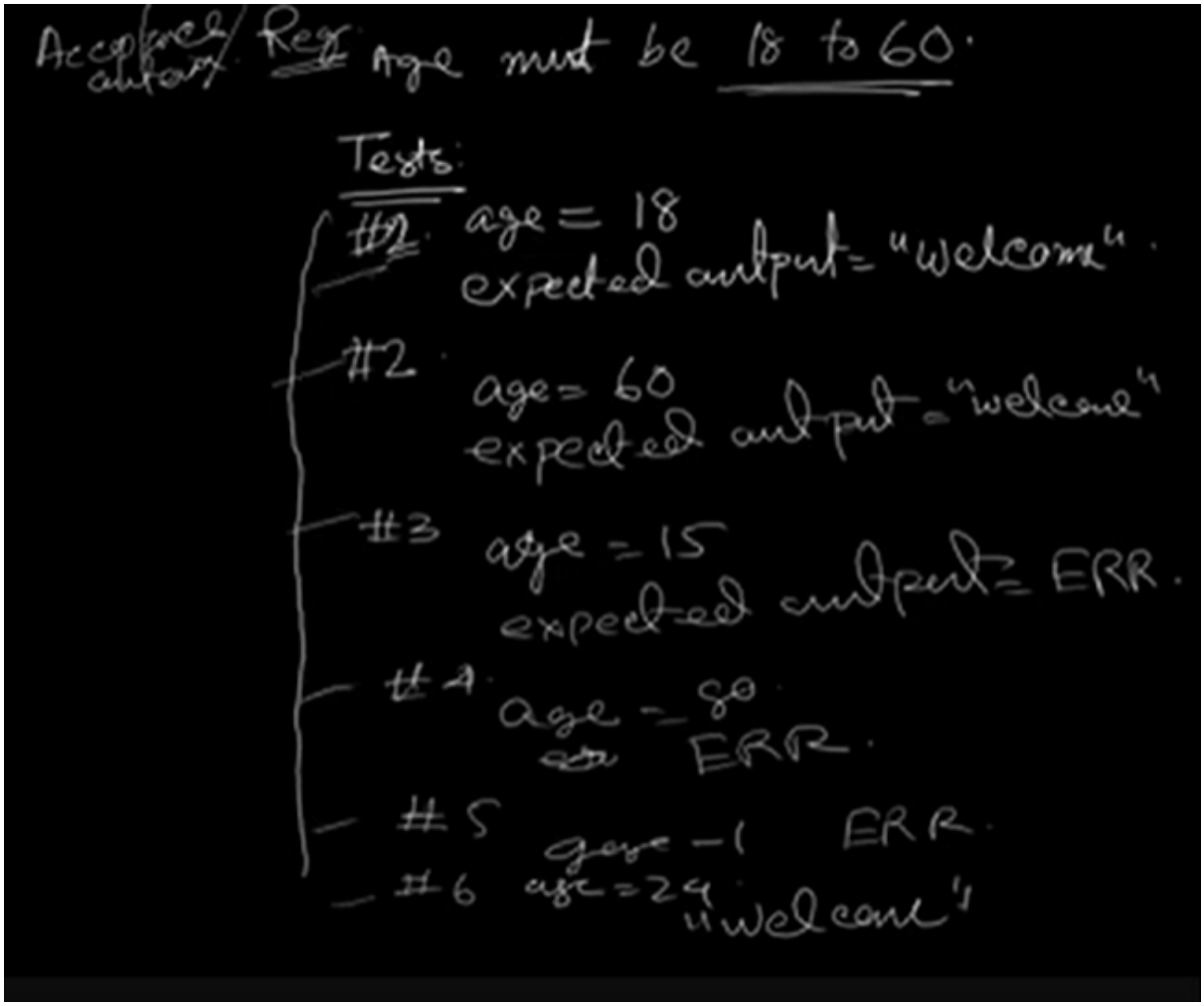
- testing by entire team
- Rapid feedback loop → based on test early/ test after
- **Important** → Clean code
- Lightweight documentation
- Done means Done → feedback is implemented if tested
- Test Driver

Test Driven Development practices

→ identify requirements → starts coding → write test → should fail → write code to pass test → do refactoring (do modification in code/ optimization and still passes the test, and figure out another test). → stop



Accepting test → age must ne less than 60 and threshold age = refactoring



Levels of refactoring

- Low Refactoring → renames variables, function
- High Refactoring → changes class design and class objects (interfaces).

→ Tests are the first clients → which gives the rough of the design → getting feedback fast.

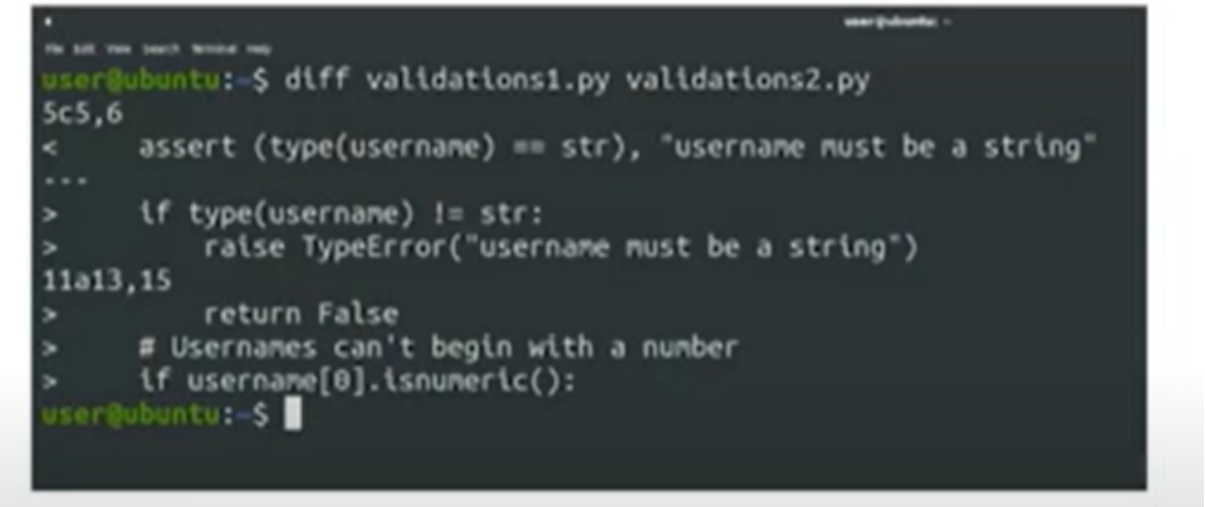
Software Versioning

→ manage all the changes after deployed , new feature required, change in environment variables, inevitable changes, track of all code history.

→why need to imp? many people working, many modification at same time, all this need to be tracked, so need to known who made those changes and how?

→ send code to other machine , code not working , solving bugs, maybe revert back to older version, so with software versioning this all can be done easily.

→ so instead of sending the whole file again and making copy of file so I would simply see where the modification are done and do that only.



command is → “diff -u”

→ if changes are very large patch your new file with old file.

→ how to fix the bug:

- Help fix a bug in the code –
  - Make copy of the file
  - Fix the bug in ‘copy version’
  - Generate the diff
  - Test the patched original file with ‘patch’ command
  - If it works, send the diff file to the colleague

`

Version Control System

→ does collaboration, which here is git.

→ git has distributed architecture.

Continuous Integration

- CICD (continuous integration continuous deployment)

Benefits of continuous integration

→ need to be done quickly and of high quality

- Continuous Integration Environment is key to having a good software delivery process that gives us confidence to build and prepare our software assets for delivery to our customers.

- Continuous Integration means we can build our software with confidence and move with better agility with the work products we're creating to deliver value.

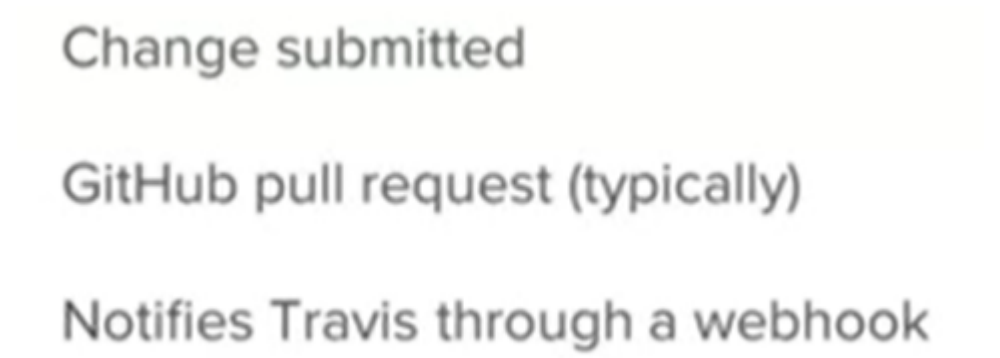
Travis - Enable Continuous Integration in GitHub

→ write travis.yml file , this file is based on the languages and tools you are using.

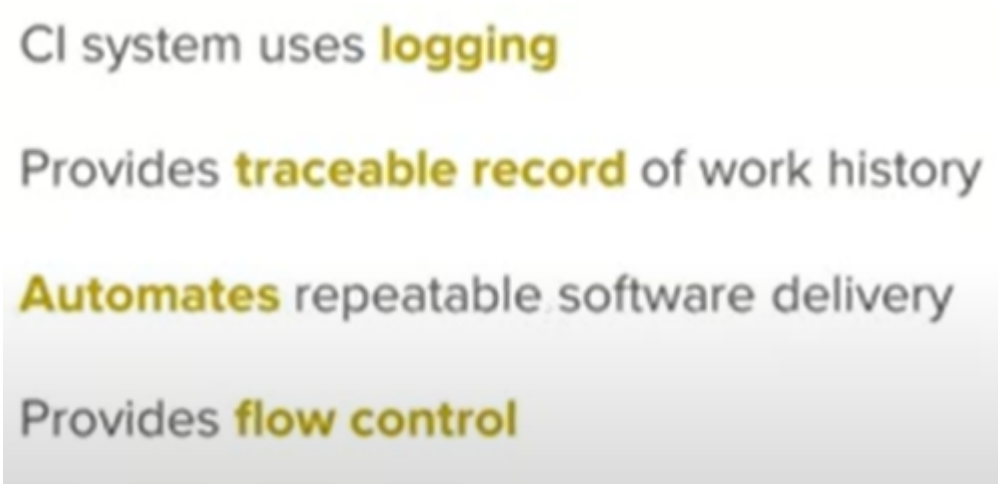
Travis.yml file to define:

- the build environment,
- the pre-installed language support,
- the sequence of steps,
- sensitive information and
- most importantly the scripts you designed to validate a ship your application

→ Primary method to trigger a job on Travis

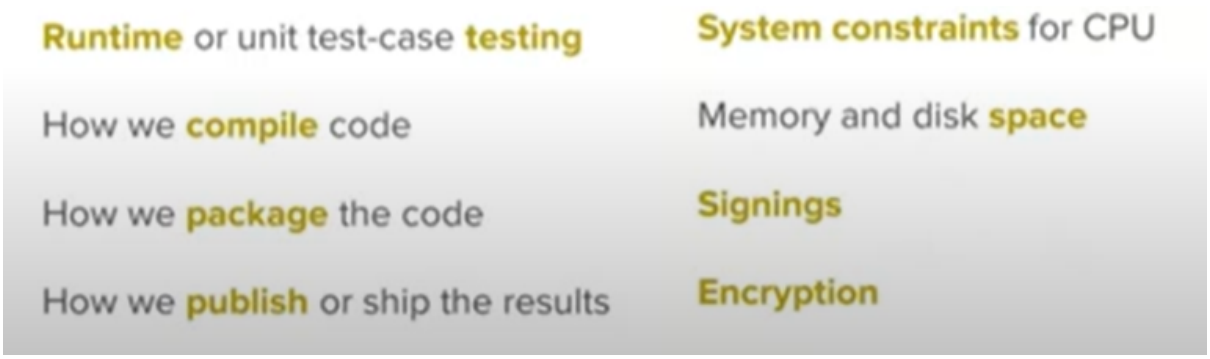


→ Basis to achieve higher quality software delivery



so it helps to achieve reliability, reusability

How CI +travis.yml file helps?



Docker → type of container

→ software having prerequisites so have docker so that one not have any tension to add / import modules

Introduction to project management

→ production , project is a temporary endeavor.

→ good project management addresses:

- quality
- cost
- schedule
- programmatic alignment

→ consequences if not met the above needs:

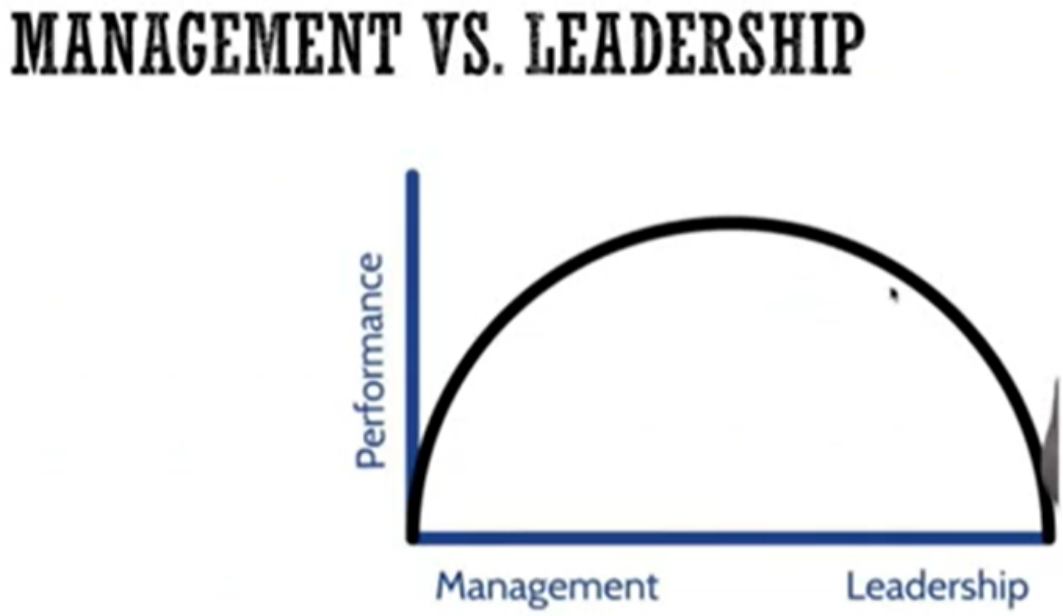
- contract penalty
- reputation damage
- resources waste
- higher costs

Project management process

→ Key elements → Management, Leadership

Management → blocking/ tackling/ technical elements of running project

Leadership → creating a vision and motivating people



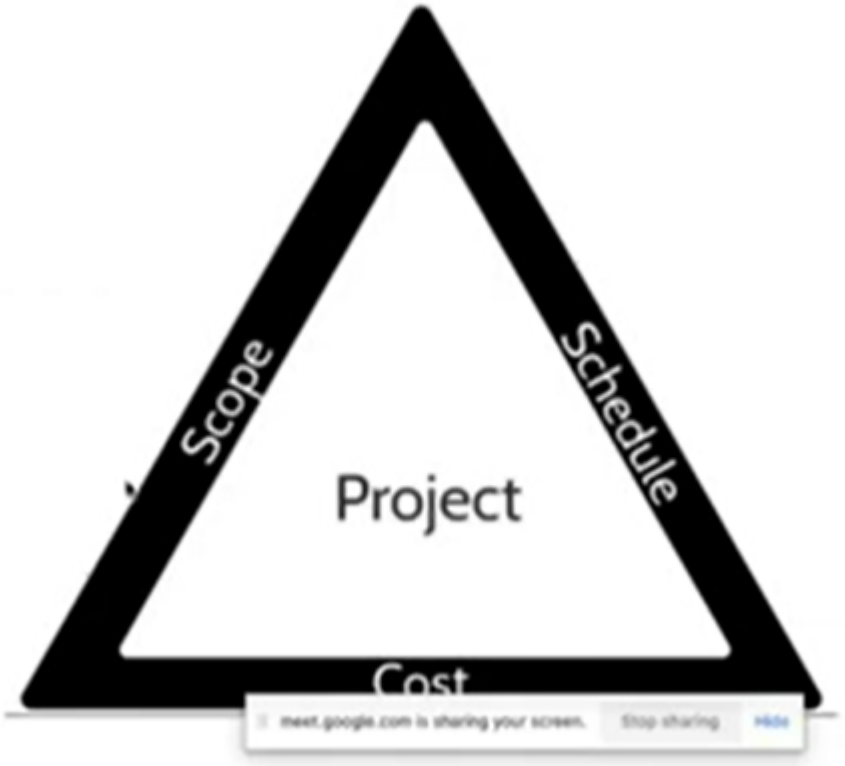
for best performance we need to have both management and leadership qualities.

→ Leadership Styles

→ For a successful project manager

PMI Talent triangle is important

→ Managing Iron Triangle

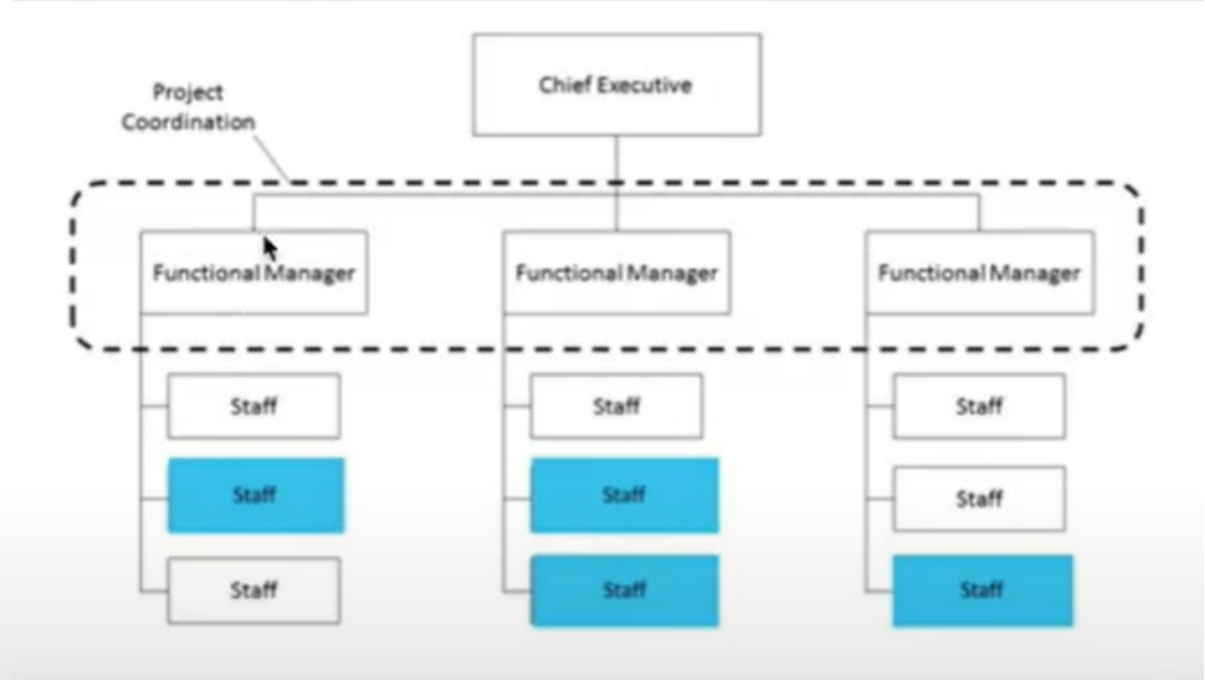


→ equilateral triangle, if any dimension changed whole triangle needs to be changed.

Project Organization

- Functional based
- Matrix Based
- Project Based

1 . Functional based



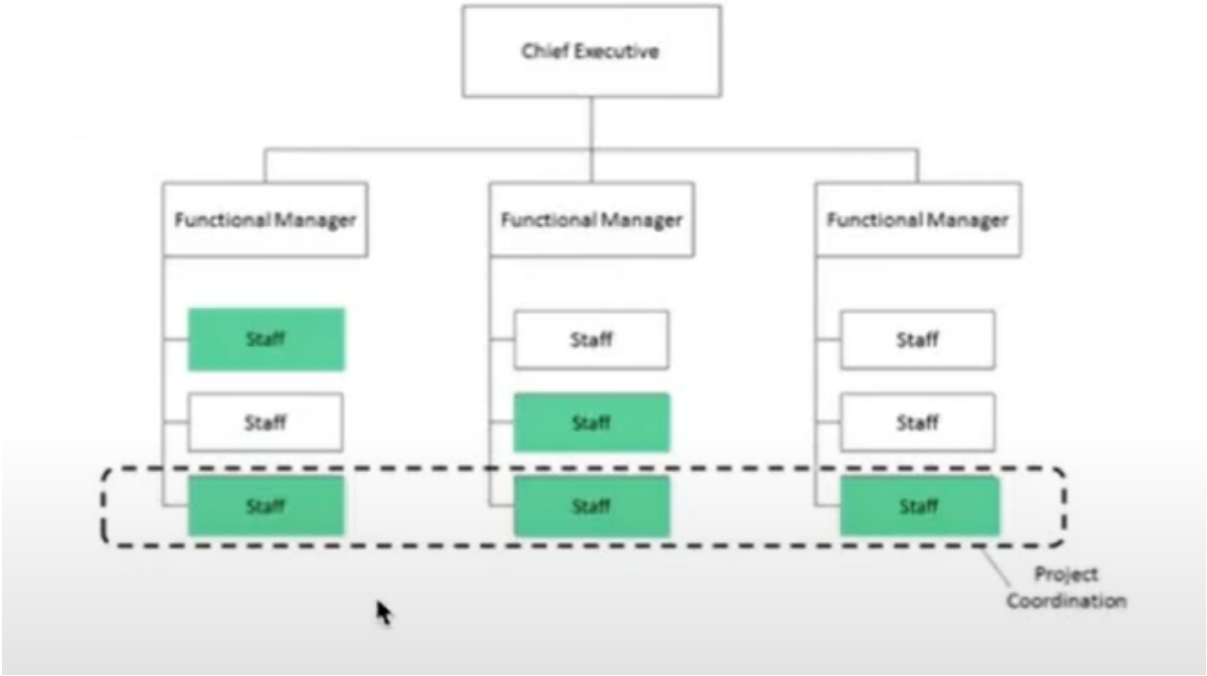
Pros

- Resources are optimized
- Improved technical control
- Communication and procedures established
- Job security for team members

Cons

- No one person is responsible
- Coordination is more complex
- No customer focal point
- Functional priorities govern over project priorities
- Project activities can easily get siloed
- Distributed budget and schedule authority

2. a) Weak matrix based



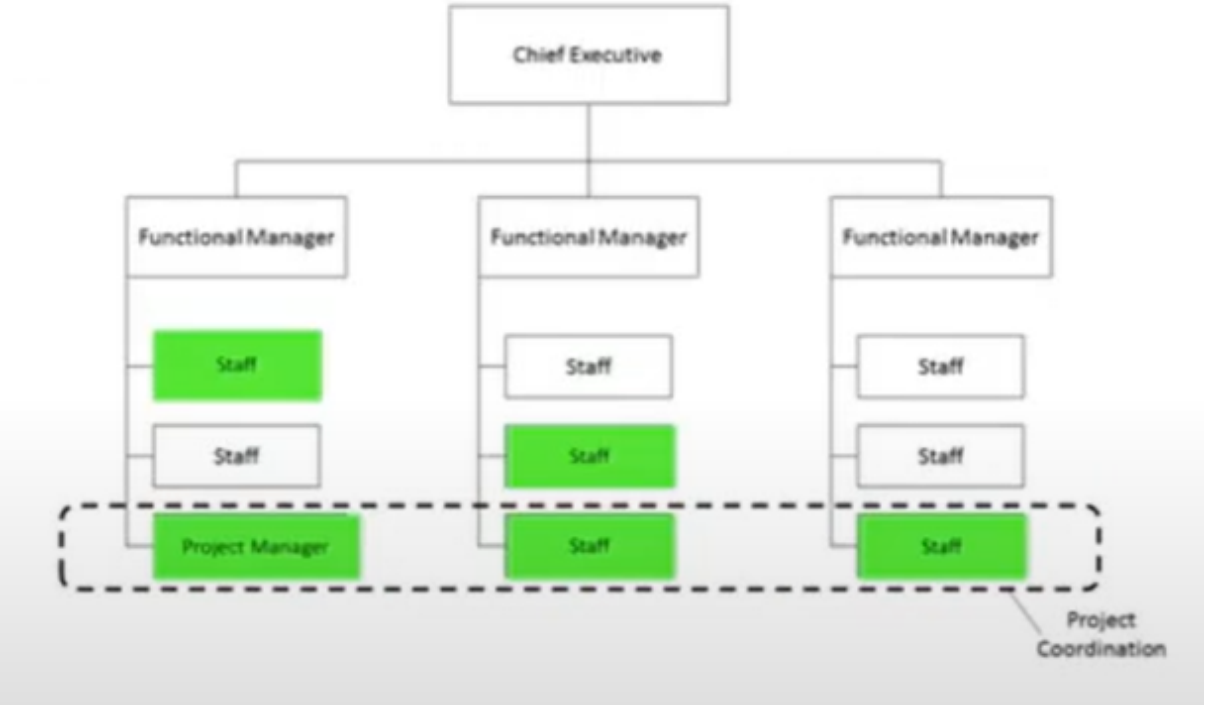
Pros

- Some Project coordination is provided
- Improved reporting
- Communication improved
- Resources optimized
- Job security for team members

Cons

- No one person is responsible
- Coordination is more complex
- No customer focal point
- Functional priorities govern over project priorities
- Project activities can easily get siloed
- Distributed budget and schedule authority

2. b) Balanced Matrix based



Pros

- Single point of responsibility identified
- Improved customer focus
- Improved communication among team members.
- Improved reporting.
- Resource flexibility
- Job security for team members

Cons

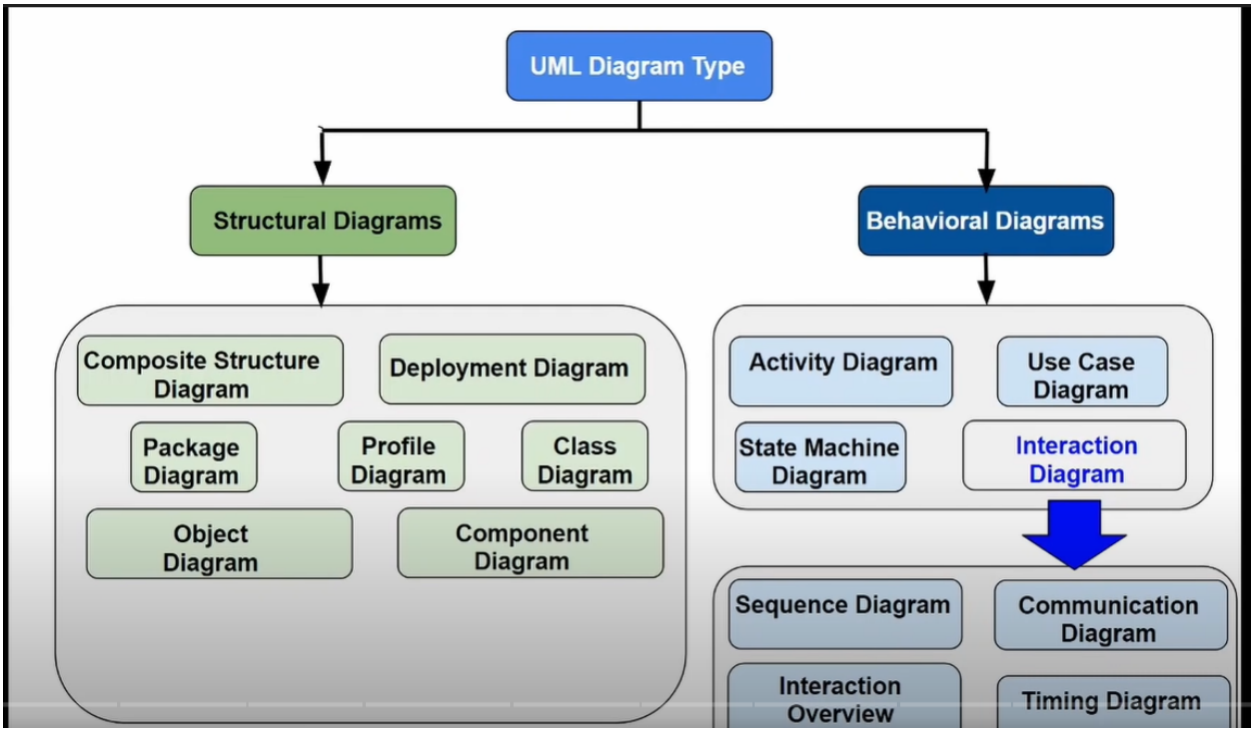
- Unclear budget and schedule authority
- Potential conflicts between Functional Managers and Project Managers
- Potential priority conflicts

SE Notes

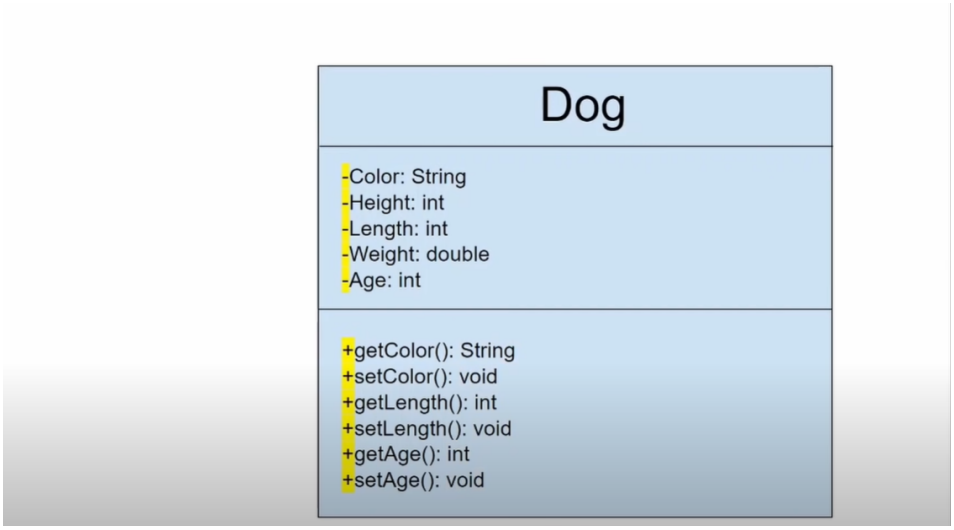
Lecture 10

UML Diagrams

The various types of UML diagrams



Class Based Diagram



➖ private

➕public

#️⃣protected

~local package

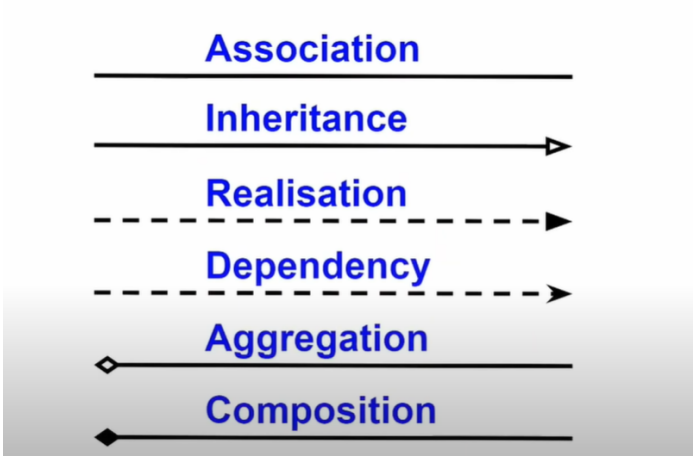
Three Levels of Abstraction

1) Conceptual Perspective: classes are real world objects: irrespective of language

2)Specification Perspective: irrespective of language

3)Implementation perspective: language specific

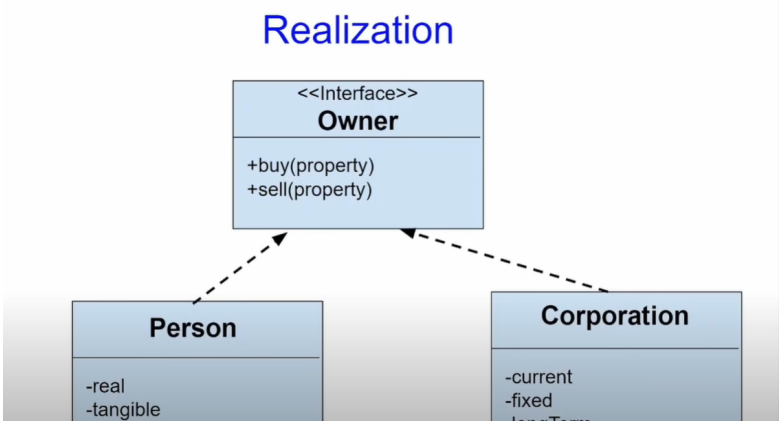
Relationships b/w classes



Association- Simple association b/w 2 classes

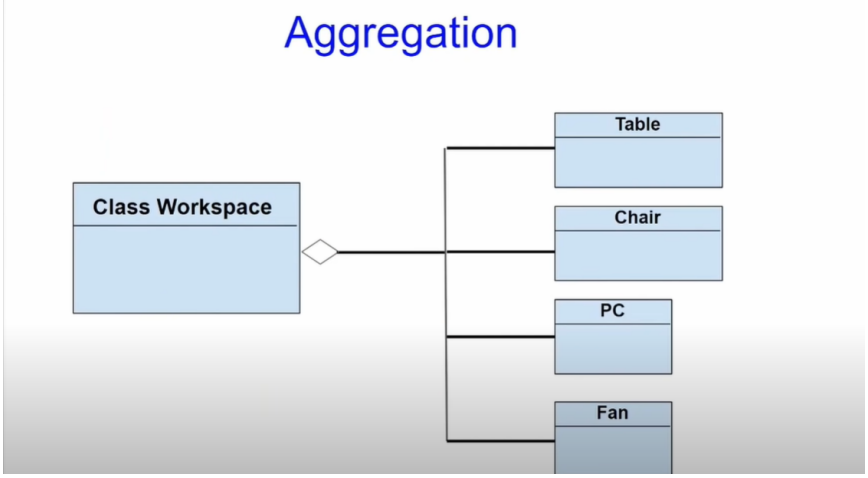
Inheritance- Eg. Circle, Square, Rectangle inherit from Shape.

Realization-



Dependency-When a class uses an object of another class without storing it in any local variable.

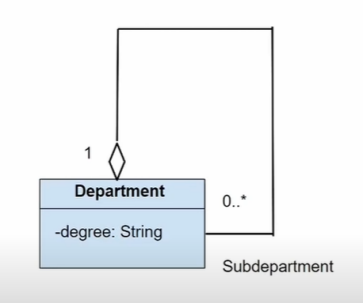
Aggregation-When a set of Classes come together and form Collective group.



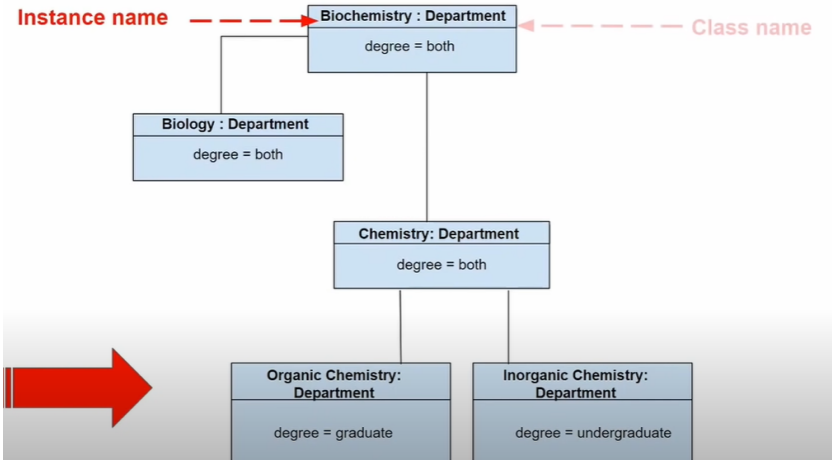
Composition-Aggregation but when the Group is destroyed the individual classes are also destroyed.

Object Based Diagrams

UML object diagrams represent an instance of a class diagram to show the general dynamic interaction of object of one class with another. It is convenient when we want to see a part of the system where a class diagram would be an over kill. A good way is to model an object diagram from a class diagram.



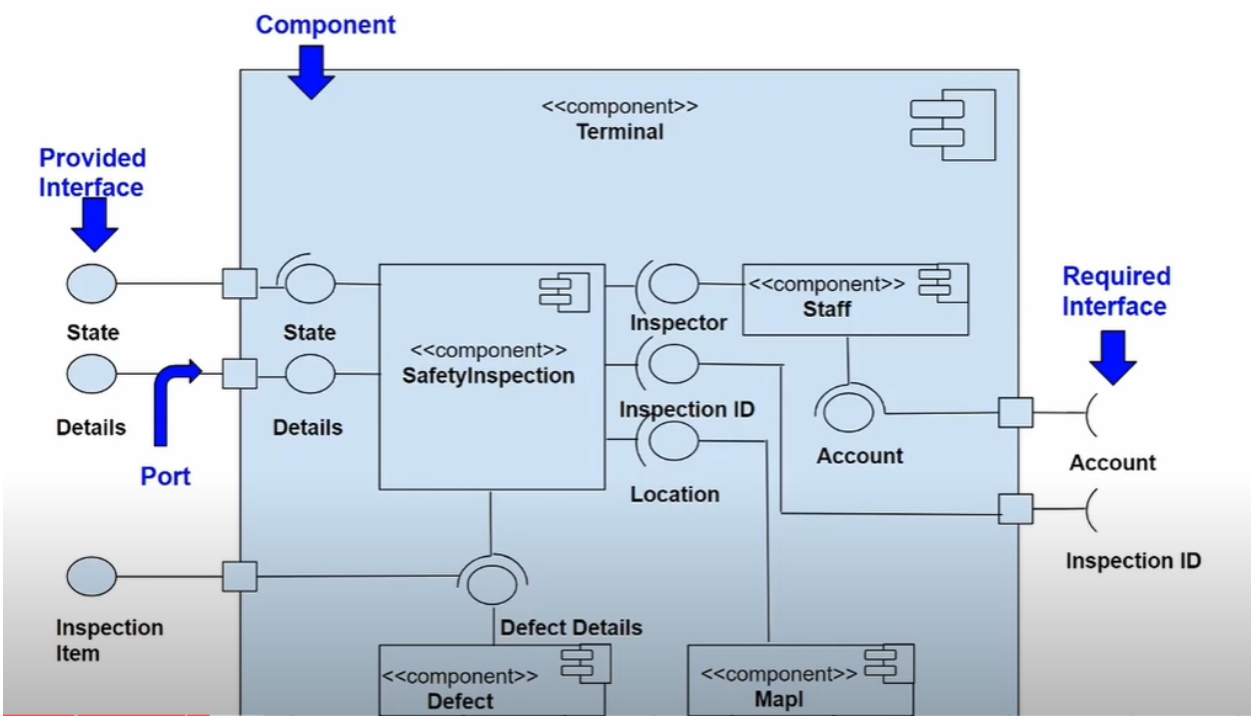
Class diagram



Corresponding object diagram

Component Diagram

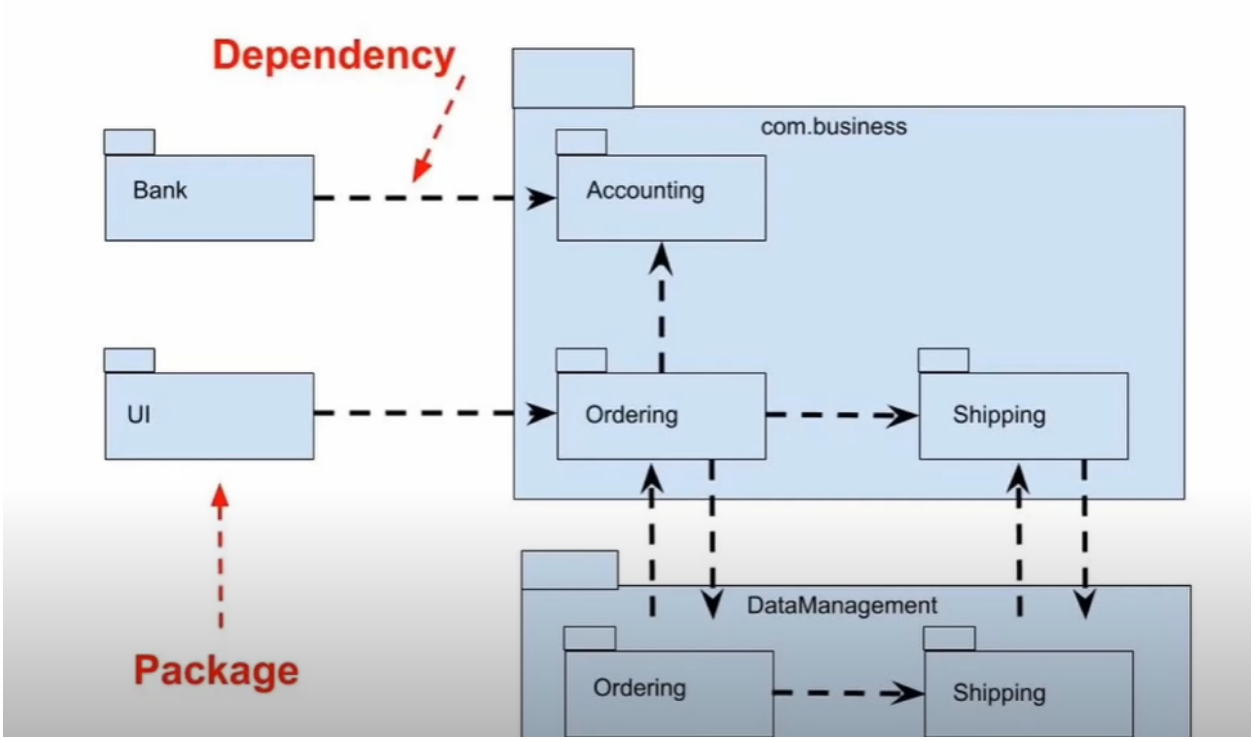
Component diagram is very much like a class diagram that talks about a particular component and it's requirements and outputs, mainly for documentation purposes.



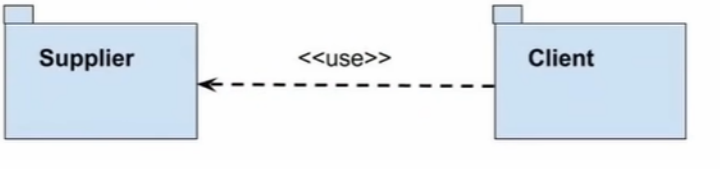
- **Required Interfaces:** The output from other components that this one needs to perform it's job.
- **Provided Interface:** The component provides some output to be used by some other component.
- **Port:** Port defines an end or starting point of a component. We can imagine them as the visible output/input pins where we can connect the plugs for interface output and input.

Package Diagram

A package diagram is used in medium to high level software systems to show the physical location as well as the arrangement of various modules.



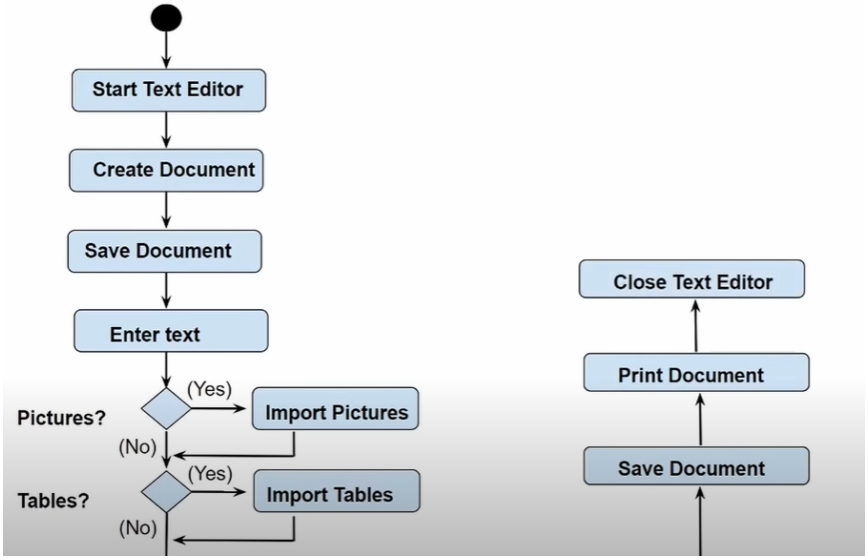
<<stereotype>>: provides some understanding of what the high level package does. Example:



UML Behavioral Diagrams

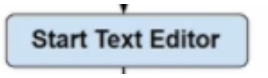
Activity Diagram

It models the transition from one activity to another. Kind of like a extended flowchart

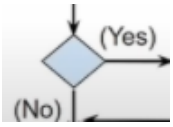


- Initial Node: The start point of logic

- Action: performs an action

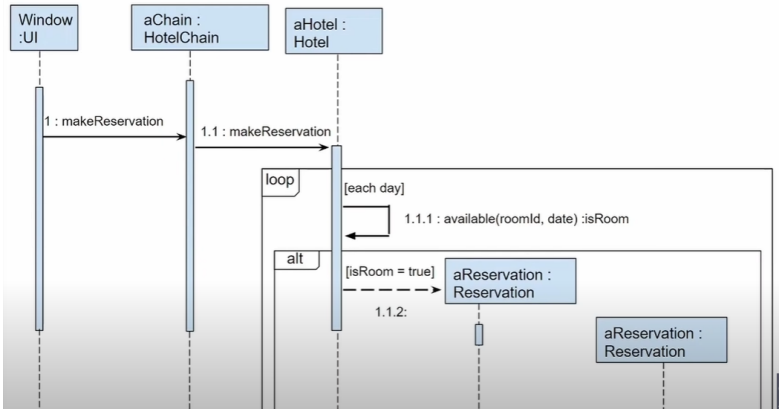


- Decision node: Decides action based on logic.



Sequence Diagram

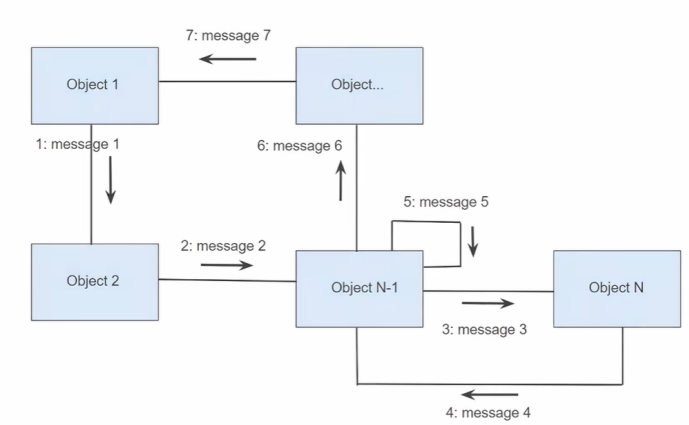
Shows how activities are organized wrt objects (x-axis) and time(y-axis).



- Actor: Someone who interacts with the system. Might be a person or some API.
- Activation: The thin rectangle represents the lifetime of the Object.

Collaboration Diagrams

Model the messages that objects share with each other that deliver the functionality.



Lecture 11

Software design architecture

Overall shape and structure of the **software system**.

Software system is the software + where it performs + how exactly it performs.

Types of soft arch:

1. System
2. Module
3. Package

Design Principles

Design patterns follow **object oriented** Design principles.

Design objectives:

1. Maintainable
2. Accommodate changes.

- Rotten Design Tendency-
  1) Rigidity-Cannot add new changes
  2) Fragility-A fault somewhere in the system breaks the entire system

3. Reusability
   - Rotten Design Tendency-
     3) Immobility- Cannot use Software modules more than once.
4. Software architecture should be easy to use
   - Rotten Design Tendency
     4) Viscosity-The pathway to realize the software architecture is very hard and unchangeable.

Lecture 12

SOLID Principle

S-Single Responsibility Principle(SRP)

Class should perform only one job and change itself for only one reason.

Here, the paint class would change only if there is a change in paint color and not if there is some change in the toolkit.

O- Open Closed Principle

Open to extension closed to modification

Polymorphism-

- Static polymorphism
- Dynamic polymorphism

Static example: `Add(3,6)` and `Add('lol','lmao')` should call 2 different signatures of function with the same name and perform the task accordingly.

Dynamic Polymorphism: suppose we have a `printAnimal()` function that takes object of animal Class. If we pass an object of Tiger class, it should still work.

L- Liskov Substitution Principle

Subclass should be substituted as per their base class.

Suppose we have `makeZoo()` which takes Animal class. If we pass Tiger class, which is a child class of Animal class, it should still work.

I-Interface Segregation Principle

Different Interfaces for different clients.

Lecture 13

SOLID continued

D-Dependency Inversion Principle

Suppose the function Curriculum takes an object of class CSE and then if we pass an object of AI it will break. So, we should have the function Curriculum to take an object of class branch where we can pass both CSE and AI.

Packages

Principles of Package architecture

1) Package cohesion principle

2) Package coupling principle

Package Cohesion Principle

1. The release Reuse Equivalence Principle

Each reusable component should have their own release cycle.

Suppose we use `datepicker` 0.8.01 and now that version is gone and we have `datepicker` 0.8.02 where some functions change. We might have to write a long code again so it's better that it is used in small blocks

2. Common Closure Principle(CCP)

Classes that change together belong together. For example, the calendar and date field module are used in date picker module so it should be bundled up together.

Lecture 14

Cohesion Principles continued

3. Common Reuse Principle(CRP)

Classes that are not reused together should not be grouped together.

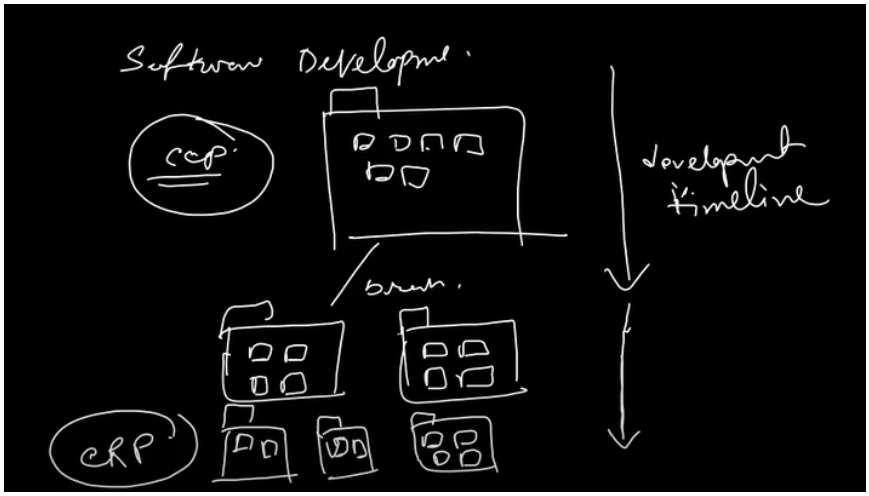
Supposes a class has 2 users Client Type 1, Type 2. Now, if there is some modification of Type 1 user, Type 2 user has to rebuild the package too.

Package Cohesion Principles are not mutually exclusive.

One cannot have them all.

CCP- Benefits the maintainer(developer), increases package size.

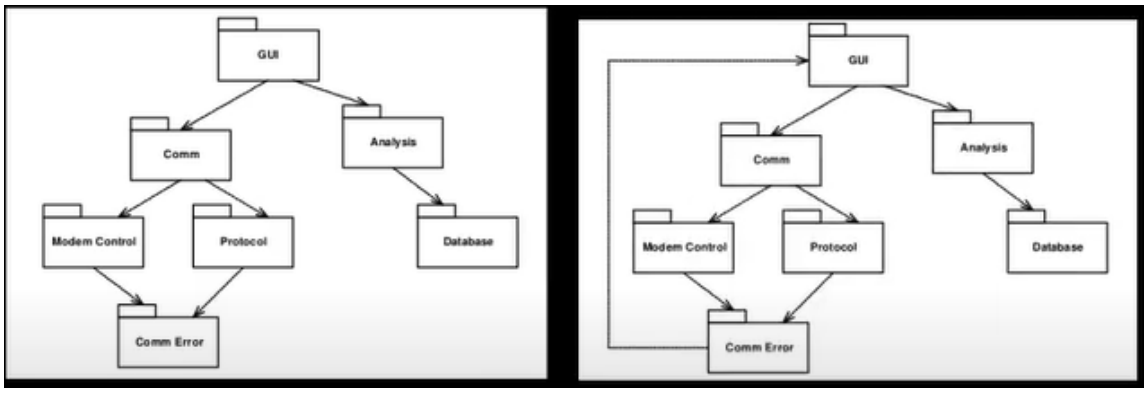
CRP & REP helps the re user, increases number of packages.



Start with CCP and move towards CRP.

Package Coupling Principles

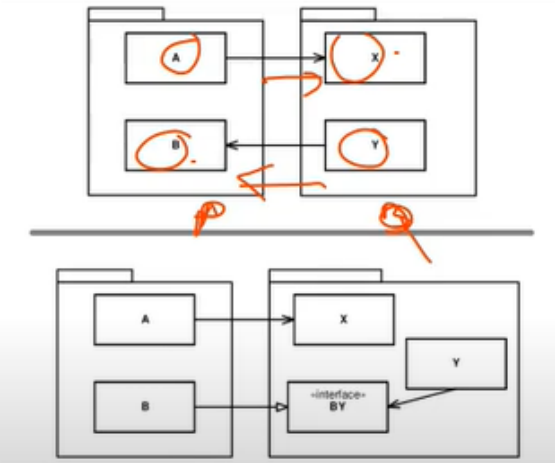
1)Acyclic Dependency Principle (ADP)



✅- no cyclic dependency ❌- cyclic dependency

How to break the cycle-

1. Use new packages
2. User DIP or ISP



Example of DIP

2)Stable Dependency Principle (SDP)

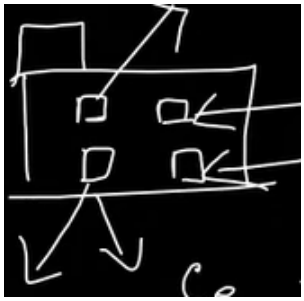
Move towards a direction of stability.

Stability metric:

Ca = no. of outgoing dependencies from a class. That is number of packages that have a dependency on the class we are talking about.

Ce = no. of classes that our class depends upon.

💡 Instability(I) = $C_e / (C_a + C_e)$



here, **Ca** = 2, **Ce** = 3

3)Stable Abstraction Principle

Stable packages are abstract packages. Unstable ones are concrete. Stable ones are easy to extend, but not change.

Abstraction metric:

Ne = no. of classes

Na = no. of abstract classes.

> 💡 $Abstraction(a) = N_a / N_e$