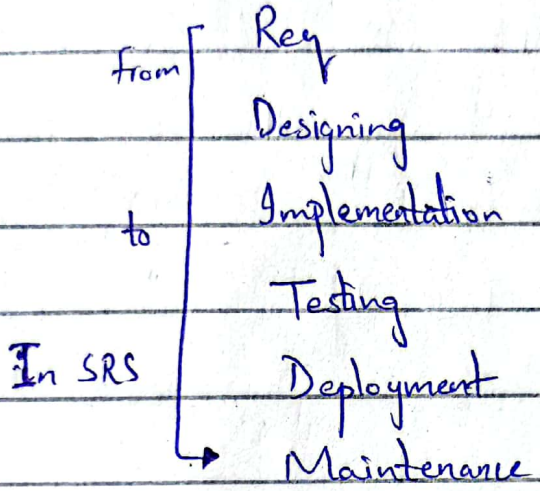
SCD-1 (07/07/23)

1) Software Engineering.

↳ 19% of softwares got rejected due to failure in req gathering.

↳ Understand domain



SCD-2 (08/07/23)

Domain:

↳ This involves req gathering:
System Req / Specification.

SDLC

↳ In this, firstly we find the domain, problem on which we are working:

Req:

User Req:

System Req:/Specification

Non-functional Req:

Waterfall Model:

↳ We follow steps, like firstly gather req then design.

+ point ↳ Each step is performed deeply.

- point ↳ Can't implement where requirements change frequently.

SCD-3 (12/09/23)

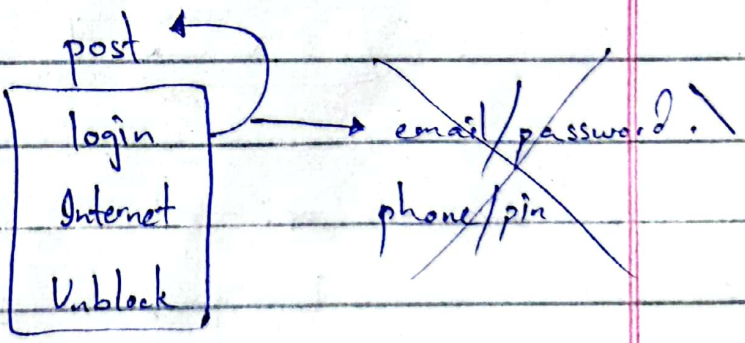
Software Engineering — Coding

1. Understand Problem Domain.
2. Req Gathering
3. Architecture.
4. Design.

2. Req Gathering

- Functional Req.
  - a) User Req. (Only what is Req). ← Post an update
  - b) System Req. (How req are implemented). ← must login to post

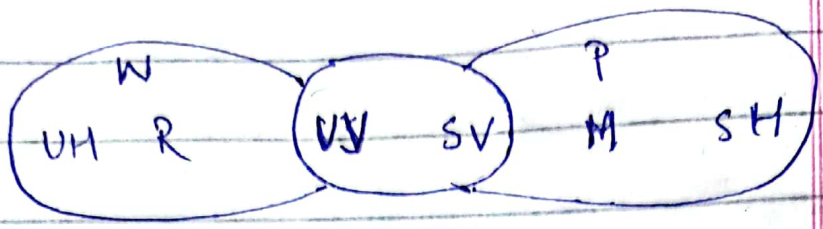
FUR
FSS



- Non-Functional Req.
  - Product Req (OS version, hardware version).
  - Organizational Req (NADRA verification)

- External Req (External companies req)

WRSPM



U → User S → System.

W → World (Domain)

R → Requirement.

S → Specification.

P → Program.

M → Hardware Machine.

S → R

P & M → S

Specification specifies req

③ Architecture.

↳ Decompose an enterprise system into independent sub-systems that have value in the system.

→ How these systems interact?

→ Principles and guidelines for the design evolution over time.

Bad Architecture ⇒ Bad Design.

4- Design.

SCD-4 (13/09/23)

- Component is build by modules.
- Module is not independent.

Persons or companies may own cars. The car owner ID is the ID either the person or the company that owns the car. A car may have only one owner (person or company). A car may have a loan or multiple loans. A bank provides a loan to a person or a company for the purchase of a car. Only the loan owner may

obtain a loan on the car. The car owner type and the loan customer type indicate whether the car owner/loan holder is a person or a company.

- **Make a list of Noun**

Class:

1) Person — name, CNIC, address, ph #
2) Company — name, address.
3) Car — Type ID, owner ID, (P, C) . 1 — purchase
4) Loan
5) Bank.

Q: Why the client needs that software?
Q: What problems he/she wants to resolve?

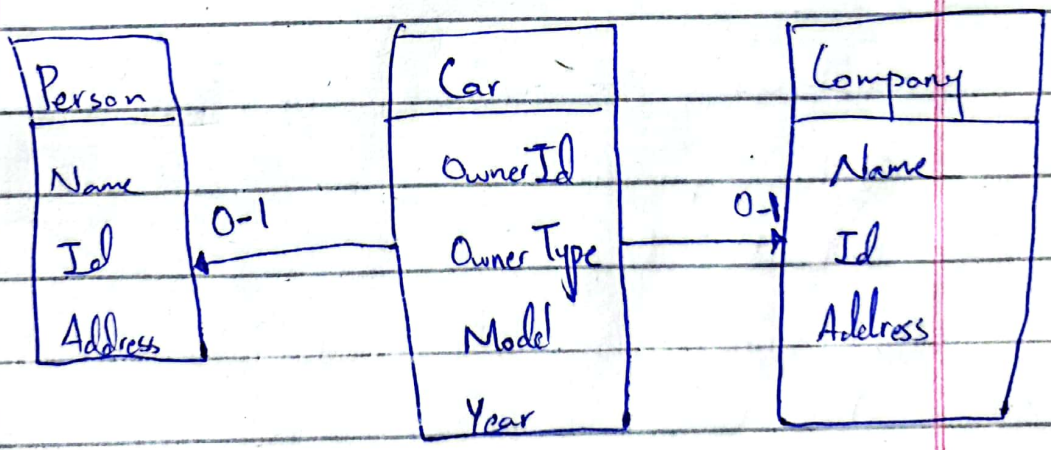
SCD-5 (19/09/23)

Scenario:

(Previous)

⇒ Nouns are used for classes.

Person	Company	Car	Loan	Bank.
Name	Name	OwnerId	CustomerType	Name
Id	Id	OwnerType	Id	Address
Address	Address	Model	Amount	BankId
		Year	Date	
			AccountId.	



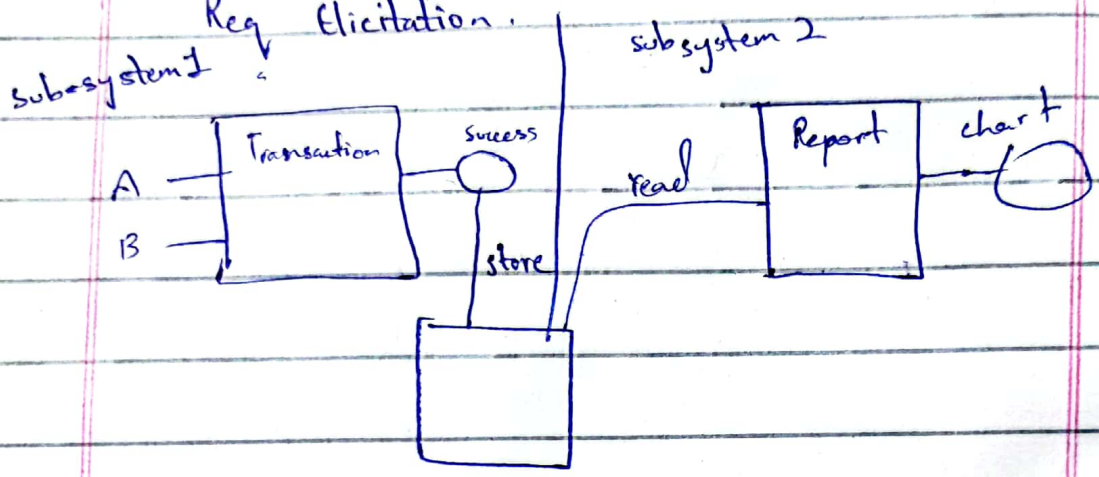
SCD-6 (20/07/23)

Domain Modeling:

Understand the problem

U & E

WRSPM



Software Architecture Models.

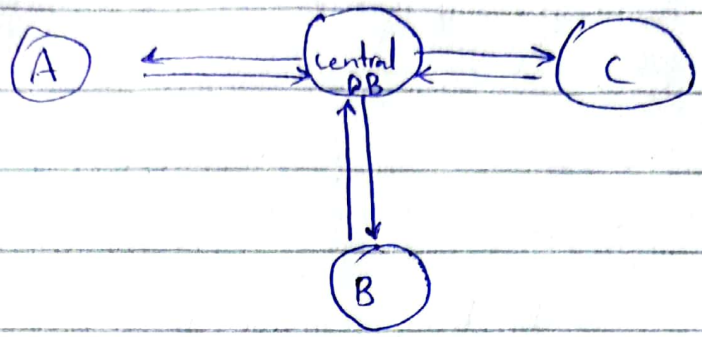
1) Pipe & filter
2) Blackboard.
3) Layered
4) Client server
5) Event Based.

1) Pipe & filter.

Same input & same output everywhere. Eg compiler.

2) Blackboard.

• Central System.



3) Layered Architecture.

⇒ view, controller.

4) Client - Server

Req → Response.

5) Event - based

Auto response on event.

⇒ similar to blackboard.

Concerns:

System Structure (Decompose into components).

Intraction of components.

Modular Decomposition: (software Design - SD)

Modularity:

Breaking down of a component and reassembling / interaction.

Modules:

1) Coupling (low coupled).
2) Cohesion (highly cohesive)
3) Information Hiding.
4) Data Encapsulation.

SCD-7 (26/07/23)

- Data Encapsulation.
- Information Hiding.
- Coupling. — If change in one class, it effects 2nd class ⇒ tightly coupling.
- Cohesion.

(Lose Coupling is recommend)

Tight Coupling:

Contents Coupling — 1) Module A can directly access Module B data members.

Common Coupling — 2) Module A and B are relied on same global data.

External Coupling — 3) Module relying on externally imposed formet / protocol / interface).

Medium Coupled: (Acceptable)

Control Coupling — 1) Module A controls the logical flow of module B by passing implementation or by using flags.

data structure coupling — 2) Module A & B rely on some ~~components~~. composite data structures. Changing data structure directly affects the other module.

Loose Coupling:

Data Coupling: 1) Module A only pass parameters for requesting functionality of Module B.

Message Coupling: 2) Module A sends messages to Module B.

3) (No Coupling)

Weak Cohesion:

Coincidential: 1) Different parts of module are together just because they are in a single file.

Temporal: 2) Different parts/code/functions are activated at the same time (move to one file).

Procedural: 3) One part follow the other in time.

4) Similar parts/function are grouped. They are similar but perform different things.

Medium Cohesion:

1) All elements operate on similar inputs produces same output.

2) One part output serves as input to other part.

Strong Cohesion

Object
- 1) Each operation in module can manipulate object attributes.

functional
- 2) Every part of the function is necessary for execution of single well-defined behaviour

Inheritance

- Domain
- Requirement Elicitation.
- Software Architecture.

Construction

- ⑤ Software Design.
- ⑧ Coding & Debugging.
- ⑥ Unit Testing.
- ⑦ Integration Testing.
- ⑧ Integration.
- System Testing.

Architecture Maintenance

Construction Phase → 4, 5, 6, 7, 8.

→ Building the software.

→ It takes 30-80% of the time.

Black box Testing

- Done by QA
- whole code is tested.

White Box Testing

We test it on modular approach.

V & V process

Validation & verification

Validation:

We check that in req-elicitation we check the req we gathered have completed all requirements?

Verification.

We hand over the project

to client, we ensure the client is agree on this once got agree it is verified.

Deployment:

- Physical environment.
- Hardware : also he can rollback too.
- Documentation.
- Training.
- DB related concerns.
- 3rd Party software.
- Software.

Failure/Maintenance

Gold Backup.
WARM Standby.
HOT FAILURE

Why construction is format.

→ All paper work is useless until you don't provide the code in working app, which is done on this phase.

⇒ 30 to 80%.

- Your working code is your actual documentation, So you can tackle most of the by looking at code

⇒ Does your developer know the language.
⇒ What that language provides you.
⇒ low level language ⇒ less performance.

- Key Construction Decision:
  - Choice of programming language.
  - Programming Convention.
  - Current technology wave.

  Selection of Major Construction Practice.

⇒ Your system should be such that if new things are added or changed then the rest should not be disturbed.

⇒ Your code should be reusable.
⇒ What problem is resolved by the software?
⇒ Constraints / Business Rule.

SCD-8 (27/09/23)

SCD-9 (02/10/23)

Classification while creating software:

⇒ Predictive & Adaptive:

- Predictive : Clear knowledge of what client requires and their is a clear one way path. (Impossible, never done).
- Adaptive : No clear idea what client really wants. (Requirements not clear).

Predictive = Waterfall Model.

⇒ Iterative & Incremental:

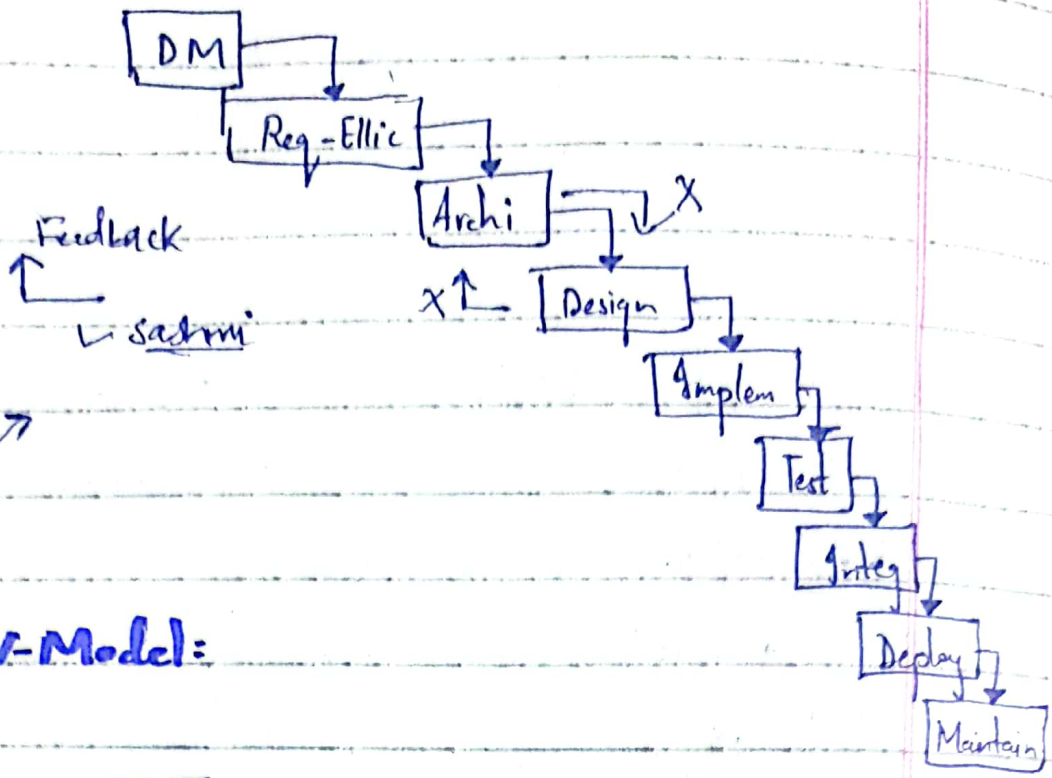
⇒ Incremental : layer-by-layer, module-by-module

if feedback then adaptive otherwise predictive.

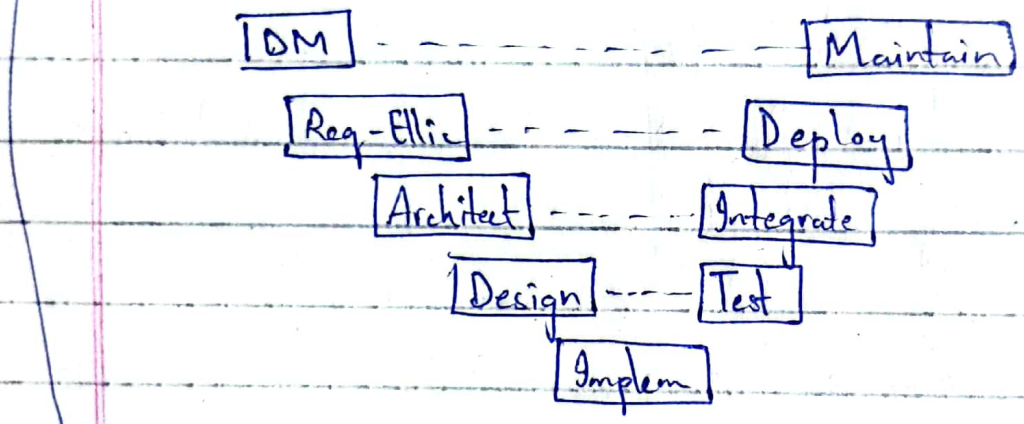
Big things divided into smaller parts.
e.g Bike → Tyre, Engine, Body

⇒ Iterative: Changes with respect to time.

Waterfall:



V-Model:



SASHMI

⇒ At a time multiple phases are running like during modelling it also makes other dept work on them.

⇒ Predictive.

⇒ Next module starts working on

(⇒ Client can't await)

the updated data from previous module as soon as it arrives.

⇒ Don't let the team to be free.

Unified Process.

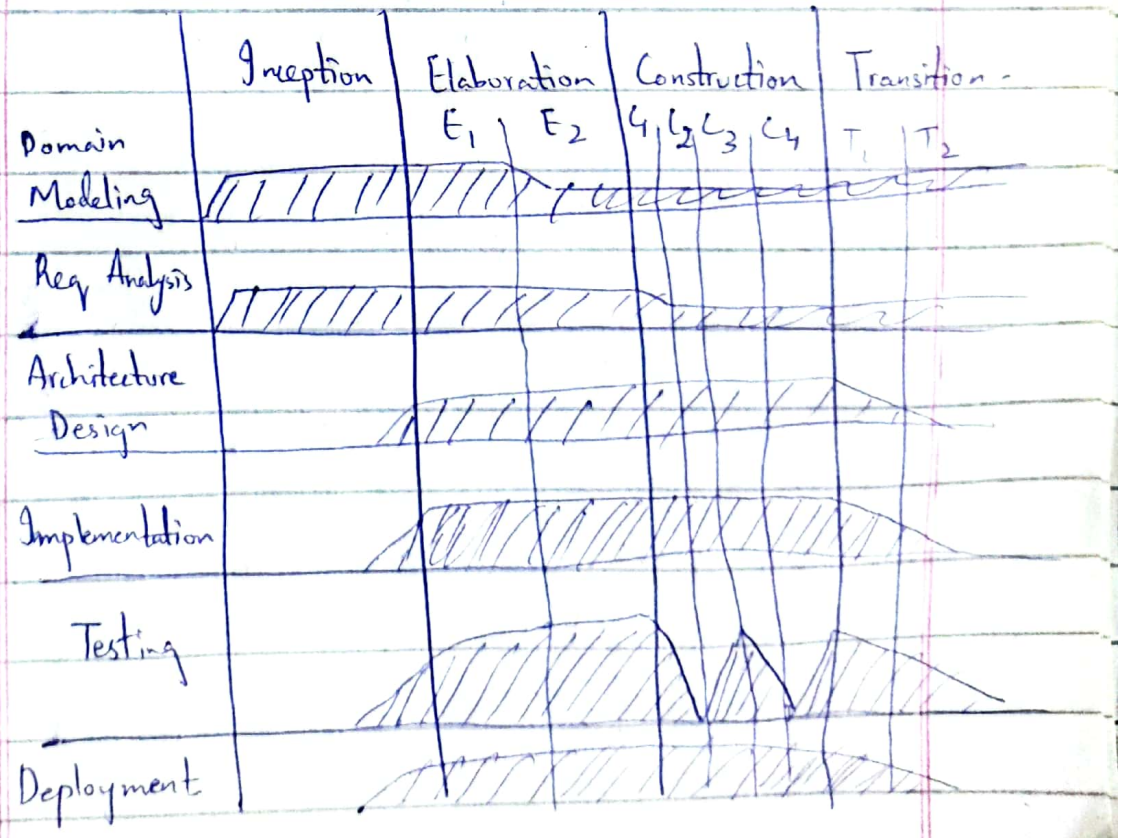
⇒ Rational UP

⇒ Enterprise UP.

SCD-10 (04/10/23)

Unified Process (Architecture Based).

↳ not completely predictive, not completely adaptive (mid).



⇒ How the phases.

⇒ framework.

Cons ⇒ Specifies Architecture.

⇒ Not a model, or iterative etc.

Cons ⇒ Implementation is difficult.

Spiral Model:

Define (define Objective)	Identify
Planning.	Implement/ Test.

⇒ for risk Analysis.

GATE:

⇒ Check after each phase/stage.

1) Should we go to next step?
2) Profitable?
3) Should we stop development?

(Is the model we are developing available in market at cheap cost?)

Cons:

Development team get demotivated.

SCD-11 (10/10/23)

Agile Methodology:

4 values

① Individuals & Interactions over Process & Tools.

② Working software over Comprehensive documentation.

③ Customer Collaboration over Contract Negotiation.

④ Respond to change over following a plan.

12 Agile Principles.

SCRUM model.

↳ 70% of agile method.

Models of Agile

DSDM

FDD

SCRUM

Crystal

XP

Custom

Lean.

12 Principles of SCRUM

1) Highest priority is to satisfy customer through early and continuous delivery of valuable software.

2) Welcome changing requirements even late in development. Agile process harness change for the customer's competitive advantage.

3) Deliver working software from couple of weeks to couple of months with the preference to the shorter timescale.

4) Business people and developers must work together daily through the project.

5) Build the 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.

7) Working software is the primary measure of progress.

8) Agile processes promote sustainable development. The sponsers, developers, and should be able to maintain a constant pace indefinitely.

9) Continuous attention to technical excellence and good design enhances agility cost of exploration.

10) Similarity, the art of maximizing the amout of work not done is essential.

11) The best architectures, requirements &

designs emerge from self-organizing teams.

12| At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

Personal Software Process

A self improvement approach for software engineers.

① Bottom up approach to practice and improve engineering practice.

② Starting point is by training individuals skill and tools for work.

③ The improvement principles are not just only for software industry but neutral for all industries.

IDEA

① Individuals should be able to plan, deliver, monitor or improve the quality and timelines of their own work.

② Use data to justify refute unreasonable demands.

say "Yes" with confidence.

say "No" with data & options

③ Learn from experience: use data from one piece of work to improve the next.

Principles (Personal Software Principles).

Measure Staff

Size

Time.

Effort.

Defects (time included in finding solution and point of introduction)

Measure Consistency:

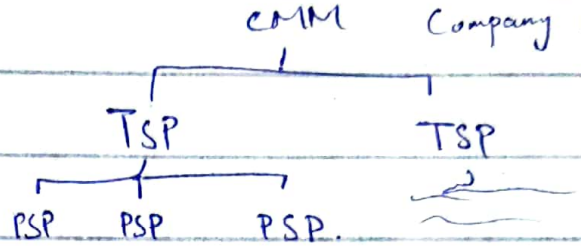
Use corelation to judge usefulness of time to predict future performance

SCD-12 (11/10/23)

Team Software Process (TSP)

Humphry's

Personal Software Process (PSP)



Humphry's

Idea

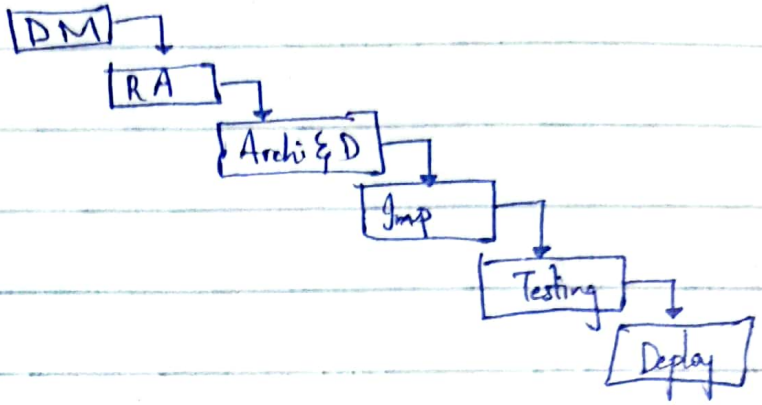
- The team should be self directed.
- Definite tasks assign to team individuals that plays role in achieving a single goal.
- Communication.
- Team members are dependent to achieve a common goal.

Principles

- Measure the tasks for each individual/analyze each member performance

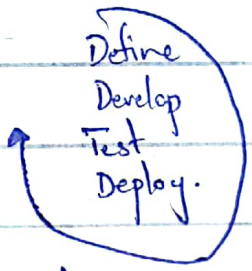
- Regular meetings.
- Rigourous planning. (Achievable goals).

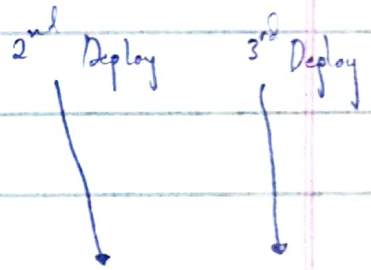
Waterfall



SCRUM

Backlog.





1-4 weeks.

Product
1) Backlog
2) Sprint Planning Meeting.
3) Sprint Backlog.

Daily Meetings
- What previous Task
- Problems.
- Assignment.
- Time - Team Member.

TSP

↳ You have a script. If problem arise then you follow that script.

4) Finish Product.

5) Sprint Review.

6) Burn up/down clients.

SCD-13 (17/10/23)

Error: Illegal operations that results in abnormal malfunctioning of a program.

→ syntax error.

⇒ logical error

⇒ runtime error/unchecked error.

Non functional attributes.

→ correctness

⇒ Robustness

Ability to handle many inputs. (correct/incorrect)

Technique Approaches

→ Descriptive / Corrective.

→ Active / Passive.

↙ frontend before service call.

↘ backend after service call.

Techniques

- Returning a neutral value
- Substitute the next valid value.
- logs & time stamps.
- Error codes / messages.
- shut down.

Exception: Undesirable situation that can arise in the program.

- **checked** — compiler can recognise
- **Unchecked.** (Incorrect input) — Compiler can't recognise.

→ try catch

→ throw

→ throws.

try → check a set of instructions that can cause exception.

catch → handling routines/set of instructions for the exceptions if they occur.

throw → (throw new missMatchException(" "))

↳ through the exception (if written in try catch, it can be handled).

throws → can't handle, itself, its caller handle.

↳ with class signature.

↳ throw multiple exception.

try
catch
final
↳ must execute.

Fault Tolerance

Collection of techniques that increase software reliability by detecting errors and then recovering from them if possible or containing their effects of recovery if possible.

1. Backup.
   - ↳ different from load balancing.
   - ↳ backup after 2, 3 days/week.
2. Retrying
   - ↳ Retry before reporting fault.
3. Auxilary Code.
   - ↳ (time tracking of code).
   - ↳ log files.
4. Voting Algorithm.
5. Replacing erroneous input with phony input. (fake)
6. Shutdown & restart.

SCD - 14 24-10

Design

- is a sloppy process.
- is a wicked problem.
- is about trade off, priorities and restrictions.
- is non-deterministic & it is a heuristic process.
- is unurgent.

Characteristics

- Minimal Complexity.
- Easily Maintainable.
- Loose Coupling.
- Extensibility.
- Reusability.
- High fan in: a class is used by many other classes.
- Low to medium fan-out: a classes uses low to midium number of other classes.
- Portability

- Leanness.
- Stratification (sorted, format).
- Keep levels of decomposition stratified. so that you can view the system at any single level and get a consistent view.
- Standard Techniques.

Principles

- Abstraction: hiding implementation details
  - ↳ Abstract class, interface.
- Encapsulation: Attributes & functions are contained in a single object.
  - ↳ access ⇒ info hiding, from other classes.
- Polymorphism:
  - ↳ overloading / overriding.
- Inheritance:
- Modularity. Breaking down of a component & reassembling it.

Design Measures

- coupling
- cohesion.
- info hiding.
- separation of concern.

SOLID

S – Singe Responsibility.

O – Open Close

L – Liskov Substitution.

Every subclass should be substitutable for their parent class.

(child class uses 100% of parent class)

I – Interface Segregation.

An client should never be forced to implement an interface that it doesn't use.

D – Dependency Inversion

High level class should not depend upon low level classes. instead both use abstraction.

SCD-15 (14/11/23)

Defensive Programming

Rule 1: You protect yourself (code) all the time.

Rule 2: Never trust users.

Strategy

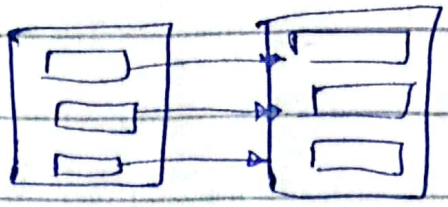
- Check the values of all data from external sources
- Check the values of all routine/function input parameters.
- Decide how to handle bad data.

Solution

- Possible input.
  - GUI
  - CLI
  - Real-time
  - External process
  - other sources.
- Barricode ⇒ Class name that validates the input.
- Internal class

  ↳ Student class - only performs its function don't validates the input.

Internal Class



Barricode GUI,

- **Assertions**

It is a logical formula inserted at same point in the program.

⇒ A type of cross-check.

① Forward reasoning. (before exec)

② Backward reasoning.. (after exec)

↳ Checking by entering wrong input.

↳ Introduce bug & test cases.

Usage:

An input/output falls within expected range, A life is open/closed as expected.

- Pointer is not null.
- Array index out of bound.
- Objects/Arrays initialized properly.

- A container is empty/full as expected verify pre-conditions/post-conditions
- Use assertion to verify the conditions that should never occur.
- Avoid putting executable code into assertion. (Don't use .exe file or very big file that takes too much time)

Code Review

- Systematic Inspection of a software.
- Phase in between implementation, before testing.

Quality Assurance

1) Reviews → Personal Reviews
   → Peer Review
   → Management Review

- White box

2) Audit → SQA testing (Ready for deploy)

- Black box
- before deploy

↳ complete SQA team test the application code.

SCD-16 (16/11/23)

Code Review

Systematic inspection of a software.

1- Peer Review

After you finish part of a program you explain your source code to another programmer.

- offline version of pair programming.
- common practice.

(Pair programming: 2 programmers work on a code simultaneously).

Advantage:

- collaboration makes a program better in quality & stability.
- catch most bugs.
- catch design flaws early.
- More than one person has seen the code.
- forces code author to articulate their decision and participate in the

discovery of flaws.

- Allow juniors to learn from seniors. experience without lowering the code quality.
- Accountability: authors and reviewers.
- Assessment of performance (Non-purpose).

⇒ Who should review?

1) Other developer.
2) Other dev from team.

Other dev or group of dev either from team or from outside.

⇒ Where should it conduct?

- In meeting
- On a decided place.

↳ The artifact should be shared before.

Fows

1) Error prone code.
2) previously discovered problem type.

3) Security.

4) Standard Checklist ⇒ Rule or style of writing the code in a company

Distinguish review types & audit types.

1) Purpose.
2) Level of independence.
3) Tools & techniques.
4) Roles.
5) Activity.

Audit Types.

1) Product assurance SQA team
2) Process assurance SQA team.

SCD-17 (21/11/23)

Management Reviews:

The main parameters of management reviews are project costs, schedule, scope and quality.

It evaluates decisions about
- corrective actions.
- changes in the allocation of resources.
- changes to the scope of project.

Personal Reviews.
- review his/her own code to enhance & ensure quality.
- ensure that your code is following standards of team/technology.
- review before peer review & management review.

SCD-18 (22/11/23)

- Banking systems were build on power builder are now being dereloped on ASP.Net Corr.

- **Refactoring:**
Improving a piece of software's internal structure without altering its external behaviour.

Each part of your code has 3 purposes.

1) Execute functionality.
2) Allow change & easy to maintain.
3) Communicate well to developers who read it.

2 Types of Refactoring:

1) Low level Refactoring.
2)

Low level Refactoring.

1) Naming:

Use descriptive names for variables & functions rather than dummy variables like a, b, a1, a2, etc.

- Avoid using 'Magic' constants.
  ↓
  anti-patter of using numbers for constant name.
- const double PI = 3.1514.

2) Procedure:

- Extract code into method.
- extract common function into method.
- Inlining an operation/procedure.
  - This technique expose significant optimization opportunities.

* Inlining:

compiler copies the code from function definition directly into the code of calling function rather than creating a separate set of instruction in a memory.

- - changing operation signatures (overloading).

3) Re-ordering:

- split one operation into several methods to improve cohesion & readability.
- put the semantically related statements near each other physically within your program.

SCD-19 (28/11/23)

Code Refactoring.

SCD-20 (29/11/23)

High level Refactoring

Significance:

- More important than low level refactoring.
- Improves overall structure of your problem.

Principles:

1- Exchange obscure language idioms with safer alternatives.

Example:

If you can write an "if" statement in one single line; Some other developers may not be familiar. Use coding style that has wide familarity & is good in term of readability.

Use switch, break, continue, return, instead of loop control variables.

Avoid loop control variables as much as possible.

2) Clarify statements that has evolved over time using comments.

3) Performance Optimization.

Process of modifying a software system to make it work more efficiently and execute more rapidly.

- Design level.
- Algorithm.
- Data Structure.
- Source Code.
- Build/Deployment.

God Class Features

- Difficult to read.
- Difficult to maintain.
- Encapsulate collection.

4) Refactor to design pattern.

5) Use polymorphism to replace condition.

6) Introduce enumeration.

7) Convert premitive type to a class.

Note

Compared to low level refactoring, high level refactoring is not well supported by tools.

God Class

A class that try to do everything in the system.

SCD-21 (05/12/23)

Q: How to refactor a God Class?

Step 1:

Identify/categorize related attributes & operations from class diagram.

- put together related items.
- find natural home of operations in related class.

Step 2:

Remove all transient association (transitive association from DBMS).

"property of any element in the system that is temporary."

- Associated class should be accessed through proper class rather than direct relation.

e.g: Library class associated with, catelogue associated with Item, Library should not be associated with item.

Question: Add a new feature to code that is not well designed.

- Assume you have plenty of time (Although it is not true most of time).
- Write unit tests that verify external code's behaviour, correctness.
- Low level refactoring.
- High level refactoring.

3 Steps to add new feature

↳ After all of these only to add a new feature.

Benifits.

1) Costs 500% ROI when refactoring (Even if dev or management do not want to do it).
- Time

2) Conductive to rapid dev.

3) Programming morale : well structured.

4) Programmers prefer to work in "clean-house"

→ Refering to code i.e., Programmers will prefer to work with well structured, well written code.

When to refactor?

Best Practice: Continuously as part of development process.

It is hard to refactor software late in the project.

Why? ⇒ Later in the project, a lot of features are added and changes affect layout huge part/features of software.

Reasons to Refactor?

1) Duplicated Code.
2) long routine (improve system by introducing modularity).
3) long & deep nested loops.
4) Poor cohesion / class has >1 responsibility.
5) Inconsistent level of abstraction.
6) Too many parameters.
7) Tight Coupling.
8) Related items are not organized.
9) A routine uses more features / attributes of the classes than its own attributes/features.

10) Inheritance hierarchies are not modified in parallel ⇒ One child doing more work.

11) Primitive data type is overloaded ⇒ e.g. currency (create class rather than int).

12) Global variables.

13) Improper / No comments.

14) Sub classes do not fully use parent class.

15) Public data structures.

16) Poor names.

17) Middle class / middle man isn't doing anything [Tramp data] e.g. only passing parameters / calling functions of other classes.

18) Passing data to other routines without modification / usage.

⇒ tramp class passing tramp data.

Refactoring:

(principles are separate (high/low))

1) Data level.
2) Statement level
3) Routine / function level.
4) Class implementation level.
5) Class interface level.
6) System.

Deployment:

- This stage occurs at the end of the active development of any piece of software.
- More of an event than a stage.
- current tech wave : Automated deployment

1) Azure 2) Amazon
↓ ↙
Cloud Technologies.

→ Must have backup/recovery plan.
→ Deployment includes planned steps, problem areas, and plans to recover
→ Deployment should not be initiated without rollback plans.

Deployment Plan Concerns:

- Physical Environment.
- Hardware.
- Documentation.
- Training.
- DB related activities.
- 3rd party software.
- Software executable.

Deployment flows:

- Deliver software
- Revert on failure.

Rollback:

Reversal of actions completed during a deployment, with the intent to revert a system back to its previous working state.

Reasons for Rollback

=> (Determine the point of no return before deployment)

- Installation did not go as expected.
- Problems could take longer to fix than installation window.
- Keep production system alive.

SCD-22 (06/12/23)

Software Evolution:

Various experts have asserted that most of the cost of S/W ownership arise after delivering software i.e., at maintenance.

Types of maintenance:

1) Corrective: encompuses fixing bugs/features.
2) Adaptive: Software adaptation to changing needs.
3) **Perfective:** This caters improved software in terms of performance.
4) Preventive: Deals with improved software by fixing bugs before they activate.

Manny Lehman "Father of S/W evolution".
S-type → Static E-type ⇒ Evolutionary.
[Real-world systems]

S/W evolution laws ⇒ On slides.
(All for e-type).

1) of continuous change ⇒ must change system with needs.
2) of increasing complexity ⇒ (more complex) = less efficient (must be extensible)
3) of self regulation ⇒ Distribution of products and processes (time for each release is same).
(also size, no of reported errors)

4) of conservation of organizational stability ⇒ the average activity rate of e-type process tends to remain constant over lifetime/ segment of the lifetime

⇒ The avg increment growth rate of e-type systems tends to remain constant over time or decline over time.

⇒ Mastery of sys decrease.

5) Conservation of familarity ⇒ All dev (also users) must grow if system grows.

6) Continued growth ⇒ must also maintain user satisfaction with s/w growth.

7) Declining quality ⇒ quality declines if appropriate amount of work not done.

8) Feedback Systems. ⇒ Many sources, levels of feedback & they must be focused on.

Legacy:

Outdated computing software and/or hardware that is still in use.

Challenges:

1) Mission Critical.

2) Not equipped to deliver new services.

3) Do not upgrade at the speed and scale of user expectations.

IoT, mobile, cloud app.

Legacy systems cannot connect to current/new systems.

- Legacy system do not handle real-time data.
- Well designed API to communicate with legacy systems.

SCD-23 (19/12/23)

Layout & Style

"Any tool can write code that a computer can understand, good programmers write code that humans can understand."

(Martin fowler)

Layout:

1) It doesn't affect execution speed & & memory consumption.
2) It affect how easy is it to understand the code, review & revision after months.
3) It also affect other developer's readability, understanding & modification in your absence.

Fundamentals:

chapter 31

1) Logically organized.

proper use of white spaces, new lines, extra spaces.

2) Consistent.

Not use multiple ways to write one thing at different locations in a single code.

⇒ Formatting result in:

1) Maintainable Code.
2) Improve Readability.

Fundamental Theorem of formatting:

⇒ Good virtual layout shows the logical structure of a program.

Note: Techniques make good code, look good & bad code look bad.

Techniques:

⇒ Proper use of whitespace

a) Grouping.

b) Blank Lines.

c) Indentation.

⇒ Proper use of parenthesis().

Style:

A block of related/similar code use "begin" & "end".

It is clear by looking at the code that particular block of code start & end.

Control Structure Layout:

a) Avoid unintend begin end pairs.

e.g. for (initial cond, final cond, each step){

// statements

}

b) Avoid double indentation with begin & end.

```
for (_____)
  ---{
      ==
  ....}
```

c) Use blank lines b/w paragraph(block of related code).

d) format single statement blocks consistently.

methods.

1) `if(exp)`
   `//statement`

2) `if (exp) // statement.`

3) `if (exp){`
   `// statement`
   `}`

4) `if(exp)`
   `{`
   `//statement`
   `}`

e) For complicated expression put separate expression on separate line.

```
if (expA &&
    expB) {
        //statement;
}
```

f) Avoid GOTO (It makes program hard to format).

⇒ No endline for case statement (exceptional) switch.

```
switch (exp) {
    case A: statement;
            break;
}
```

⇒

```
case A:
    statement;
    break;
```

⇒ for switch statement linespace is not recommended after a break.

Individual Statement Layout:

⇒ Statement length:

outdated rule: 80 character max.

now a days: 90 character usually.

⇒ use space for clarity & readability.

⇒ spaces in logical expression.

⇒ spaces in array references.

⇒ spaces in parameters.

⇒ Formatting continuation lines.

1) ⇒ make incomplete statements obvious.

while((expA) && (expB)) { ≡ } (not recommended)	while ((expA) && (expB)) { ≡ } (recommended).

2) Keep closely related elements close.

3) Indent routine call continuation lines the standard amount.

4) Make it easy to find the end to continuation assignment statement.

5) Indent control-statement/assignment statement/continuation lines the standard amount.

6) Don't align right side of assignment statement ≡

d) Use one statement per line.

e) Data Declaration.

⇒ only one data declaration per line.

⇒ Declare variables close to where they are first use.

⇒ Order declaration sensibly.

⇒ In C++, put asterible of pointers with variable names.

Comment layout:

⇒ Indent a comment with its corresponding code.

⇒ set off each comment with at least a line.

Routine layout:

⇒ Use blanks to separate parts of routine.

⇒ use standard indentation for routine argument.

SCD-24 (20/12/23)

SWEBOK ch 6 section 2.

Software:

↳ setup + Documentation (use, deal, handle feature)

configure:

Project Management → Software Qualification(next)

Configuration of System:

Functional & physical characteristics of hardware & software mentioned in technical doc or achieved in a product.

Process

Constaints & guidance.

Planning.

- ↳ Software Process (waterfall, V).
- ↳ Management Roles
- ↳ Resources.
- ↳ Implementation.
  - ↳ Legacy
  - ↳ Support.
  - ↳ Maintance.
  - ↳ Scope
  - ↳ Authorities.
  - ↳ vendor.
  - ↳ migration.
  - ↳ feature Data.
  - ↳ Owner.

⇒ **Software Release Management**

Identification, package & delivery of elements of a product.

Release:

1) Executable Program.
2) Documentation
3) Release Notes.
4) Configuration Data.

Concerns:

⇒ when to issue a release.
⇒ product delivery items.
⇒ version release notes.
⇒ track distribution of product to customers.
⇒ Digital Verification.