Lecture #1

5-9

SCD

Software Construction

- Implementation
- Logic Building

• SDLC

1. Requirement
2. Design
3. Implementation
4. Testing
5. Deployment
6. Maintainance

Lecture#3 12-9-2023

SCD

Software Construction

① Software Engineering → Design, Documentation

② Coding

→ Nature of Problem

Functional Requirement

① Understand Problem Domain

② Requirement Gathering (What is required?)

ⓐ User requirement (User language) what?

ⓑ System requirement How? (How to implement)

Non-Functional Requirement

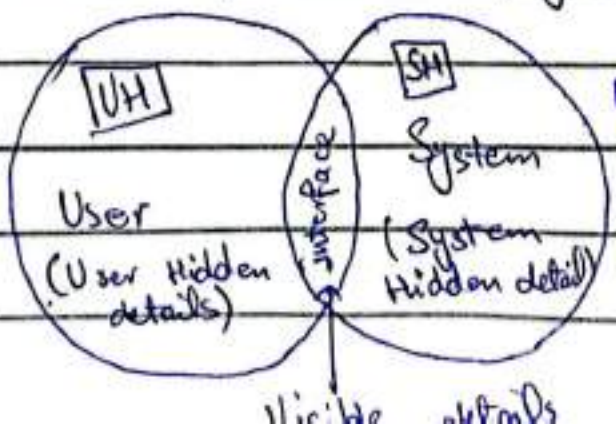
① Product Requirement (Hardware Req. Product Deployment

② Organization (organization Proof) (Security)

③ External Requirement (All factors that are imported from external system)

↳ Python old version.

↳ Ecomerce purchase payment method failed



WRSPM

~~W → World (Domain)~~

R → Requirement

S → Specification

P → Programming

M → Machine

③ Architecture

→ Decompose an enterprice system into independent sub-system that have value in the system.

→ How these sub system interact

→ Principles and guidance for the design evolution over time.

Lecture #4 13-09

SCD

→ Module depends on other

Persons or companies may own cars. The car owner ID is the ID either the person or company, that owns the car. A car may have only one owner (person or company). A car may have a loan on 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 company.

Make a list of nouns and Verbs

classes

1. Person
2. Company
3. Car
4. Loan
5. Bank

Lecture #5 20-9

SCD

Domain Modelling

→ Understand the problem

WRSPM

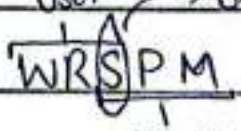
↓

World (Domain) → Understand your domain

Requirement Elicitation

↳ User requirement (what the user need?)

User → Between user & environment



Software Architecture

→ Software Components (Sub-system) | That have their own value

→ How they interact (sub-system)

→ Independant feature and how they provide facility and interact with each other

SA Model

Module → Dependant (Can be ~~many~~ many)

① Pipe & Filters (Same input, output) (Compiler)

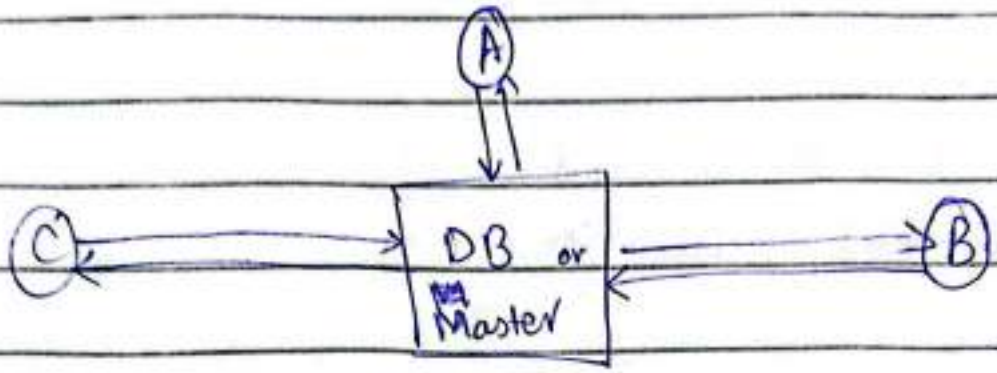
② Blackboard (Central database, Different Modules)

③ Layered (MVC, MVM) (Implement in different layers and don't interact)

④ Client Server (Request, Response or Served)

⑤ Event based (ASP .NET) Output on specific event

→ Architectural Models that can be follow together or sp seperately



Blackboard

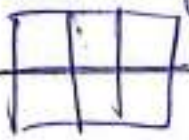
Event based => Occurs on specific event

Software Design

System Structure (Decompose into Components)
Interaction of components (sub-systems)

→ Modular Decomposition → (Software Design)

↓ Design classes



Software Design

→ Design of a specific (particular) module

Modularity

Breaking down of a component and interaction

A module must have these four

things:

① Coupled - Coupling

② Cohesion

③ Information Hiding

④ Data encupsulation

SCD Quiz#1(+ Mids)

- SDLC

1- Requirement Grathering

- Initial phase where you collect and document what the software needs to do.
- User requirement (what) => What the software do? in user language.
- Foundation of the entire project so it is most crucial part. (Includes Domain Modeling

2- Design

- Create a detailed plan how the software work - Blueprint
- Based on requirement gathered

3- Implementation

- Write the code to buil the software
- Design is the plan, Implementation is the execution of the plan

4- Testing

- Check if the software works correctly and meets the requirement.

5- Deployment

- Phase where the software is made available

to users.

6- Maintenance

- Ongoing phase where you keep the software up-to-date, fix issues and make improvements.

Requirements

- **User requirements**
- [what] end-users expect or need
- user language (user-friendly) not technical
- High-level and guide the development process
- ~~Relation to Others~~ **Product req.**
- Constraint on the software to be developed
- [what?] → Specifies what [Functionality + Hardware]
- **Process req.**
- Constraint on the development of the software
- [How?] → Specifies How? [Method to create]
- **Functional req.**
- Functions that the software is to do
- [Must] [How?]
- **Non-functional req.**
- Quality requirements (Performance, security)

- **Software req.**
- Subset of system requirement.

→ **Requirement Sources**

- Requirement have many sources
- **Goals** - high-level objectives of software
- **Domain Knowledge** - knowledge about domain
- **Stakeholders** - affected by software development
- **Business Rules** - structure of business itself

→ **Software Design**

- Design can be viewed as a form of problem solving.
- **Environment:** Requirement will be derived from the environment in which the software will execute.

- **Abstraction** is a view of an object that focuses on the information relevant to a particular purpose and ignores the remaining remainder of the information.
- **Coupling** is defined as a measure of the interdependance among modules in prog.
- **Cohesion** is defined as a measure of the strength of association of the element within a module.
- **Coupling** measures how closely one module relies on another module.

① Low coupling: It means they have minimal or no dependencies on each other. Change in one should not impact the other module significantly. This is desirable because it leads to more modular and maintaiable software.

② High coupling: It mean two module have closely intertwined, depend on each other. changes in one cause affect the other.

- **Cohesion** refers to the degree to which the element within a module or

component are related to each other How well a module's internal components work together.

① High Cohesion: It means that its internal components are closely related and contribute to a single. It lead to more understandable, reusable and maintainable modules.

② Low Cohesion: When module loosely related or unrelated components that perform various tasks. Makes a module harder to understand.

Goal:

→ Minimize coupling, low dependence

→ Maximize cohesion, high internal relatedness

- **Decomposition and modularization** means that large software is divided into a number of smaller named components having well-defined interfaces.
- **Encapsulation and information hiding** means grouping and packing packaging the internal details of an abstraction and making those details inaccessible to external entities

WRSPM

W → World Assumption (Domain model)

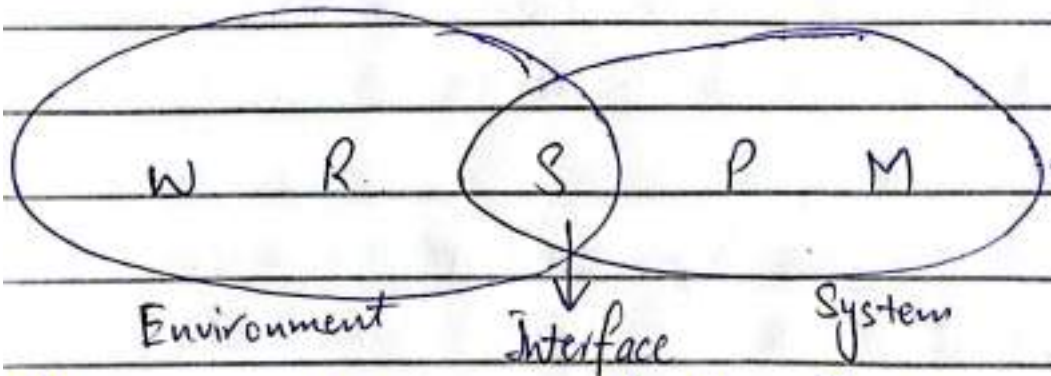
R → Requirement

S → Specification

P → Program

M → Machine

→ Understand the difference between requirements and specifications



Requirement	Specification
→ Statements of what the software do or what properties it should possess.	→ Detailed description of how the software will fulfill the requirement
→ focus on "what"	→ provide "how"
→ "The software must allow user to log-in"	→ User interface design, database structure, and algorithm to use.

Software Design

Process of creating a detailed plan or blueprint for how a software system will be built.

=> Architecture guides the design

Object-Oriented Design

- focuses on organizing data and behavior into objects, which are instances of classes

① Encapsulation is the bundling of data and methods into a single unit called/ class and the data member are not accessed directly.

② Inheritance allows a class to inherit the attributes and method of another class (parent). Promotes code reuse

③ Polymorphism allows objects of different types being used interchangeably.

④ Abstraction is like creating a simplified model of something complex. It focus on important stuff and hides the details you don't need to see

Tightly Coupling: (Not Good) Content coupling

① Module A directly accen Module B data member.

② Common coupling

② Module A and B are relied on some global data.

External Coupling

③ Module s reling on externally ~~importe~~ imposed format (protocol / interface).

→ XML and Jason

Loosely Coupling: (Highly) Data Coupling

① Module A only pass parameter for requesting functionality of Module B

② Module A sends message to Module B,

Message Coupling

Medium Coupling: (Some what accepted)

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

② Module A and B rely on some composite data structure changing data structure directly affects the other module. (Data Structure Coupling)

Cohesion:

Weak Cohesion:

. Coincidential Cohesion

① Different parts of module are together just because they are in a file.

Temporal Cohesion

② Different parts/code / functions are activated at the same time.

Procedural Cohesion

③ One part follows the other in time

Communicational Cohesion

④ Similer parts/functions are grouped. They are similer but perform different thing.

Medium

communicational

① All elements operate on same inputs and produces same output.

② One part output serves as input to other part. → Sequential

Strong

① Each operation in module can manipulate object attributes → Object Cohesion

functional

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

↓ Functional Cohesion

SCD

Testing

① Blackbox testing

→ Testing a product without knowing internaly (Blackbox → hidden).
→ without code, only check input and output (funtionality working properly)

② Whitebox testing

→ Testing the code internally
→ Internal code / logic

③ Top-Down testing

First test high-level modules, then low level

④ Bottom-Up testing

Begin with testing low level modules to high level modules.

⑤ Unit testing

It tests small, isolated piece of code (units) / individual components

⑥ Integration testing

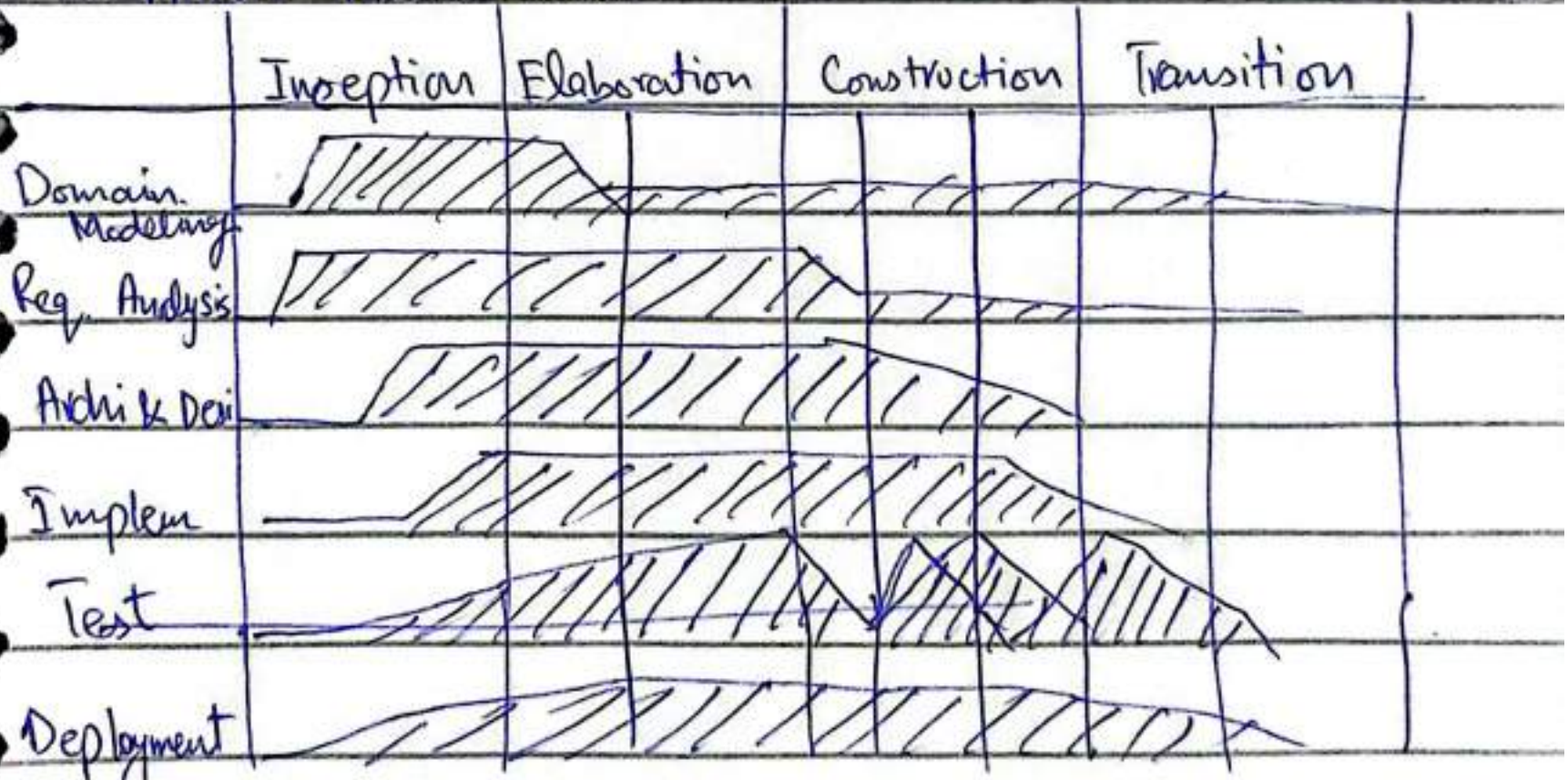
Like connecting and testing different parts to ensure they work together.

SCD

- Itrative Model

→ Unified Model (Iterative and Incremental)

→ ~~Attidr~~ Architecture

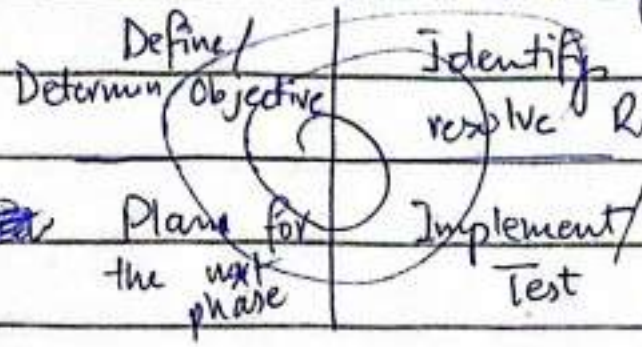


→ Understand & Deployment is Difficult

scope

↑ Provide support to build prototypes at every phase

- Spiral Model => Each activity is an "spiral"



→ Risk analysise

→ Only those activities or steps are passed through spiral

↳ GATE CHECK Phase Gate Mode

→ A gate is applied after every step → Stop / Continue

→ There are multiple way to solve a problem in software development

→ Team is stays motivated

→ Saves time and Money

- Agile

→ 17 people brainstroming → agile mindset

SCD

10-10

Agile ~~Principles~~ Values

17 People -> (Senior & Junior) { Agile Manifesto }

4 -> values 12 - Principles

(1) Individuals and Interactions over Process and Tools

(2) Working Software over comprehensive documentation

(3) Customer Collaboration over Contract negotiation

(4) Responding to change over following a plan

Agile Principles

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

(2) Welcome changing requirement ~~are~~ but even late in development. Agile process harness change for the customer is competitive advantage.

(3) Deliver working software from couple of weeks to couple of months with the preference to the shortest time scale

(4) Business people and developers must work

- DSDM
- FDD
- SCRUM ☜
- Crystal
- XP
- Custom
- Lean

together daily throughout 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 user should be able to maintain a constant pace ~~ident~~ indefinitely.

(10) Continuous attention to technical excellence are good design enhance agility ~~test of exploration.~~

(11) Similarity the art of maximizing the amount of work not done - is essential. The best architectures, requirement and designs ~~emerge~~ emerge from self-~~organizationing~~

teams.

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

AI 10-10

Adversarial Search

→ Competition

→ Typical AI assumption

→ Zero-sum game (gain of one is lose for other)

Search vs Game

SCD Continue 10-10

IDEA

① Individuals should be able to plan, deliver, monitor and impose the quality and timeliness of their own walk.

② Use data to justify refute unreasonable refute demands

Say "Yes" with confidence and "No" with data & options

③ Learn from experience use data from

one piece of work to improve the next

Principles of Personal software Process

Measure stuff (size, time, efforts, defects)

② Measure Consistently

Use orelation to judge usefulness of time to predict future performance

SCD

Self Reading

Team Software Process

PSP
TSP

→ Motivated Team members

→ Scripts (2-3 page) if there is a prob. then follow this

Humphrey's Idea

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

Principles

- Measure the task for each individual/Study
- Regular meetings
- Rigourous Planning → Planning for next tasks targets (Achievable goals)

CMM — Company Management

Team — TSP — TSP — Team Improvement

PSP PSP PSP PSP PSP PSP — Personal Improvement

↓ Individuals

SCRUM

- 1 - 4 week (working software)
- Backlog
- 1 Deployment in waterfall method = 4-6 Deployment in Scrum method

Deployment = Sprint

(Scrum master manage product backlog)

(Priority) → statement

Backlog — Simple english statement

② Sprint Planning Meeting

③ Sprint Backlog ← Only for one specific sprint

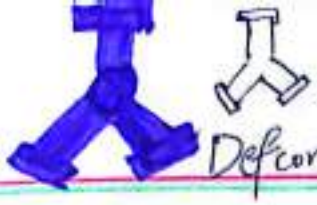
↳ Daily meeting
- what was previous task
- Problems
- Assignment
- Time Management
- Team Members

④ Finish Product (First Sprint)

⑤ Sprint Review (Problem, issues)

⑥ Burn up / Burn down charts

- Scripts is for problem solving



Defcon. 1

SCD

17-10

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

① Syntax - incorrect syntax (compiler)
② Logical - incorrect program behaviour
③ Runtime - Division by zero

Approach

- Active : Program handle at realtime
- Passive : After inputting wrong that the database cannot store type of

Error handling

- Dealing with errors
- Non-functional attributes

① Correctness - error free
② Robustness - ability to handle many inputs that might be ~~correct~~ or incorrect

Technique

- Return a null value when error occurs
- Substitute with next valid value
- Logs and time stamp error events
- Error code to identify and handle errors.
- Shutdown in critical errors

Exception Undesireable situation that can arrive in the program. There are 2 types:

① Checked - compiler can recognise
② Unchecked - Compiler can't recognise (Runtime Error) - may occur during program execution

Destrictive
Energy

try:
check set of instruction that can cause exception
code

caught:
handling routine ~~solution~~ for those exceptions if they occur
- code that specifies how to handle exception

throw: (often provide additional information)
new input mismatch exception

throws: (declare class can throw exception)
① Class signature ② Multiple

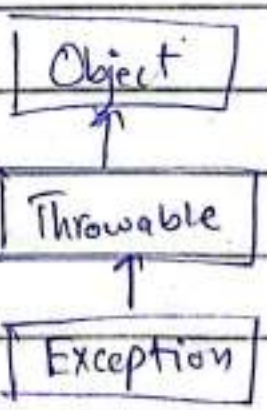
final:
- runs compulsory if the exception occurs or not

SCD 18-10

Fault Tolerance

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

1. **Backup** (creating duplicate)
2. **Retrying** (Attempts to perform a fald operation again)
3. **Auxilary code** (Time tracking of events)
4. **Voting Algorithm** (Democacy with different scenario)
5. **Replacing** (~~erroneous~~ erroneous input that phony input)
6. **Shutdown & Restart**



Rollback

Once deployed, customer want to change something get back and do again.
Rollback → Return to previous state.

Deployment Customer Stratigies:

These are plans / approaches to transition or change from old version to new.

- Cold Backup

A cold backup is a strategy where an environment is setup but remains offline until needed. It is not actively running. Suitable for system with lower tolerance.

- Warm Backup

Where environment is partially active and synchronized with primary system.

- Hot Failover

Both primary & standby system runs simultaneously.

SCD

24-10

- **Design**
  - Is a sloppy process
  - Is a wicked process
  - Is about trade off, priorities and restriction
  - ~~Is non-deterministic~~

- Is non-deterministic & it is a heuristic process] → Error and trial (Improve)
- Is emergent

Charateristic

- Minimal complexity
- Easily maintainable
- Loose coupling
- Extensibility
- Reusability
- High ~~form~~ fan-in : a class is used by many other classes
- Low to medium fan-out : a class uses low to medium number of other classes
- Portability - easily adapted, deployed and run
- Learnness
- Stratification (sorted format)
- Keep levels of decomposition stratified so that you can view the system at any single level and get a consitant view
- Standard techniques

Priciples

- **Abstraction**, hiding implementation details, abstract, interface
- **Encapsulation**: attribute and functions are contained in a single object
- access =>

->Information hiding, hiding internal data from other classes

- **Polymorphism**
  - Overloading
  - Overriding
- **Modularity**
  - Breaking down of a component and reassembling it
  - design measure
    coupling
    chesion
    information hiding
    seperation of concern

- SOLID Principles
  - Liskov substitution
  - Interface Segregation
  - Dependency Inversion
  - Open Close → close for modification
  - Single Responsibility
    One class do only one thing

Liskov substitution => Every sub class should be substitution for their parent class

Interface segregation => A client should never be forced to implement an interface that it doesnt use

Dependency Inversion: High level class should not depend upon low level classes instead both ~~dont~~use abstraction

SCD Mids

Continue

Method Overloading

→ Creating multiple methods with the same name but different parameters lists within same class

→ The method to execute is determined at compile-time based on the method signature.

Method Overriding

→ Occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

→ The method to execute is determined at the runtime based on objects type.

SOLID Design Principles

SOLID is a set of five design principles in object-oriented programming that aim to make software more

① Understandable ② Flexible ③ Maintainable

→ Writing clean modular code

① S: Single responsibility

- A class should have a single responsibility or job.
- It should do one thing and do it well.

② O: Open-Closed

- Software classes, modules, functions etc. should be open for extension but closed for modifications.
- Should be able to add new functionality without changing existing code

Example:

If you want to add new payment method to a processing system, you should be able to do so without altering existing payment method.

③ L: Liskov Substitution

- If you have a base class, you should be able to use any of its subclasses without causing issues

④ I: Interface Segregation

- Classes or module should not be forced to depend on interface they don't

use.

- Interfaces should be specific to the need of client (classes or modules)

⑤D: Dependency Inversion

- High-level modules (abstractions) should not depend on low-level modules (details)
- Rely on abstract interfaces or classes, not concrete implementation

Design Process

(noun = object ; verb = method ; adjective = attribute)

- Inheritence and polymorphism play a key role

Development Models

- Method used in SE to plan, design develop, and deliver software

1. Iterative Development

- Software is built incrometally through a series of repeating cycles.
- Each cycle refines and enhances the software based on feedback and changing requirements

Incremental Development

- Incremental models divide the project into smaller, manageable parts (increments) that are developed and delivered seperately.
- Each increment adds new functionality.

2. Predictive (Traditional) Models

- Predictive Models follow a well-defined plan from start to finish.
- They work best when requirements are well-understood and unlikely to change.

Adaptive (Agile) Models

- Adaptive models are more flexiable and adapt to changes throughout the project.
- They emphasize collaboration, customer feedback and incremental development.

Development Models:

1. Waterfall

Type : Predictive

- Sequential approach with distinct phases,

requirement → design → implementation → testing → deployment → maintenance

Adv: Simple, easy, good for well-define projets

Disadv: Inflexiable to changes, no feedback

2. V (Validation and Verification) Model

Type: Predictive

- Extands waterfall model by adding a validation and verification phase for each development phase.

Adv : Emphasise V's, enhances quality

Disadv: Still inflexiable.

3. Sashimi

Type : Predictive

- Variatim of waterfall that allows for overlapping phases

Adv: Better for accomadating changes, faster than waterfall

Dis: Still sequential, Not good for large projects

4. Unified

Type : Adaptive

- An iterative and incremental approach that combines elements of waterfall and

iterative development

Adv: Highly adaptable,

Dis: Complex for small projects

5. Spiral

Type: Adaptive

- An iterative approach that emphasize risk analysis and management.
- Each iteration is a "spiral"

Adv: Excellent risk management, accomodates changes as well

Dis: Complex, longer development time

6. Phase Gate

Type: Hybrid

- Combine phases with iterative and adaptive elements.
- It includes gates to evaluate progress before moving to next phase.

Adv: Allows for controled flexiabi'lity, risk assessment

Dis: Complex to manage

Agile

- Flexible and iterative approach that prioritize customer collaboration, responsiveness to change
- It divides the project into short development cycles called sprints (e.g. two weeks) and emphasize delivering functional software at the end of each sprint.
- Agil approach includes practices like Scrum, Kanban and XP (Extreme Programing)

Agile Methodology

- Set of principles that emphasizes

① Flexibility ② Collaboration ~~③ Iterative~~

③ Customer feedback ④ Iterative ~~feedback~~ progress.

Values

1. Individuals and Interactions over process and tools
2. Working software over comprehensive document
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan.

Scrum

- Agile framework
- Scrum Backlog: A prioritized list of work items that need to be addressed during a Scrum project.

① Product Backlog

List of all features, requirements and fixes that are required for the product

② Sprint Backlog

This is subset of Product Backlog and contains items that the development team have committed to completing during current sprint.

⇒ Sprints

- Time-boxed development cycles
- last (2-4 weeks)
- Scrum mastery's responsibility is to ensure that the scrum is understood and followed. He serve as coach and servent-leader for the Scrum team.
- Meetings

IDEA

① **Daily Scrum** - daily plan and progress
② **Sprint Review** - at the end of sprint
③ **Burn-up and down Charts**
- Visual tools used to track the progress of work during a sprint.
Burn-up shows the completed work
Burn-down shows the remaining work

Agile Values
① Individuals and ~~Interconnection~~ Interactions over processes and tools
② Working software over comprehensive documentations.
③ Customer Collaboration over Contract negotiation
④ Responding change to following plan.

Agile Principles
① Highes priority is ~~to do~~ customer satisfaction by early and continous delivery of

working software
② Welcome Changing requirements, even in the middle of development. Agile processes harness for the customer's competitive advantage.
③ Working software should be deliver frequently, ~~focus~~ on preference on shortest time scale
④ Business people and developers must work together daily throughout the project.
⑤ The working software is the measure of progress.
⑥ Develop project around motivated individuals. Provide them support and environment and trust them to get the job done.
⑦ Agile promots the sustainable development.
⑧ The sponsors, developer and user should work in constant pace indefinately.
⑨

① Our highest priority is to satisfy customers through cool early and continuous delivery of valuable software.

SCD

14-11

- Defesevive Programming → code

Rule 1: You protcit yourself all the time

Rule 2: Never trust users

Strategy

- Check the values of all data from external. Sources.
- Check the values of all rantime input parameters.
- Decide how to handle bad data/inputs.

Solution

- Possible input
  - GUI
  - CLI
  - Real time
  - External processes
  - Other Sources

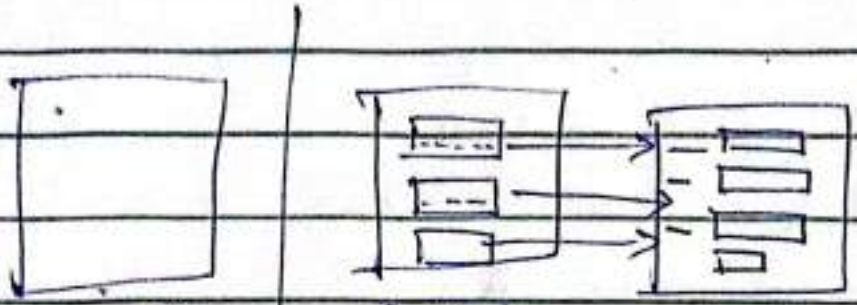
Front-end

- Barricode => Class name that validates the input

Validate class

- Internal class

↳ Student class - Only performs its fun() don't validates the input



- Validate the input that user enters at front-end

- **Assertions**

It is a logical formula inserted at some point int the program.

→ Type of cross-check

① Forward Reasoning - Before execution - Flow - Green Signal

② Backward Reasoning - After execution - Too much condition

↳ Checking by entering wrong input

↳ Introduce bug and test cases

Usage / Where to use Assertion:

- An input/output falls within expected range.
- A file 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 assertions 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
(Whit box testing) → Peer Review
→ Mangment Review

2 → Audit → SQA testing (Black box testing) (Ready for deploy)
↳ Complete SQA team test the application code

=> Assignment 10% ~~weightage~~

=> 23 design pattern
① Scenario ② UML Diagram ③ 2 Examples

SCD

15-11

Quality Assurance

- Reviews — Peer Reviews
  Managment Reviews
  Personal Reviews
  (Whit box testing)
- Audits — SQA testing team (Black box testing)

Pear Programming: Two programer/partner work together in the system

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 ~~Peer~~ Pair ✓ programming
- ~~Peer~~ Pair ✓ programming: Two programmer works on a code simultaneously.
- Common practice

Advantages

- Collaboration makes a program better in quality and stability.
- Catch most bugs.
- Catch design flaws early.
- More than one person has seen the code
- Forces code authors to articulate their decision and participate in the discovery of flaws.

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

=> Who should review?

① Other developer

② Other developer from team

③ Other developer or group of developers either from team or from outside

↳ Where should ~~be~~ it conduct?

-> In meeting

-> On a decided place

↳ The artifact should be shared before.

Focus

① Error prone code

② Previsouly discovered problem type

③ Security

④ Standard Checklist → Rule or style of writting the code in a company.

Distinguish Reviews types and audit type

① Purpose

② Level of independance

③ Tools and technique

④ Roles ⑤ Activity

Audit types

① Product assurance	SQA team
② Process assurance	SQA team

Management Review

The main parameter of management reviews are ① project cost, ② schedule, ③ scope and ④ quality.

It evaluates decisions about:

- corrective actions

③ scope → which feature needs to provide

- changes in the allocation of resources.
- changes to the scope of the project.

⇒ Manager only checks the quality of the code

Personal Reviews

- Review his her code (previous & refactor) to enhance and ensure quality.
- Ensure that your code is following standards of team/technology.
- Review before Peer Review, Management Review or Audit

• Code Refactor Overview

→ GOD class → class that have a lot of functions (code)

→ Improve the code without changing the features, altering its external behaviour

- **Decision Tree**
  - No hard and fast rule.
  - High time and space complexity
  - Not sure about efficiency.

- **Rule Based**
- **SVM** – Support vector machine
- **k- Nearest Neighbour**

SCD 22-11

Code Refactor

Why?

- Readability
- Performance and Optimization
- Maintainability
- Reusability

Refactoring

Improving a piece of software's internal structure without altering its external behaviour.

Each part of your code has 3 purpose

1- Execute functionality

2- Allow change => No tight coupling but can further grow

3- Communicate well to developers who read it.

↓ well commented code

2 ... Easy to maintain

Principles

Low level Refactoring

Naming:

- Use descriptive names for variables and functions rather than dummy variables like abc, a1, etc

- Avoid using "magic" constants

magic => anti-pattern of using numbers for constant names. For example:

const double variable = 3.1415 ✗

Pi ✓

Procedure:

1. Extra code into method
2. Extra common function into method
3. Inlining an operation / procedure
   - ↳ Implement the code that is imported in the same file, rather than generating

another file. → No new file strategy
→ Necessary in threading and deep hardware programming (eg Low level languages) C-language)
→ This technique expose significant optimization opportunities.

Inlining:

Compiler ~~code~~ copies the code from function definition directly into code of calling function. rather than creating a ~~rep~~ seperate set of instructions in a memory.
4 changing operation signatures (overloading).

Reordering

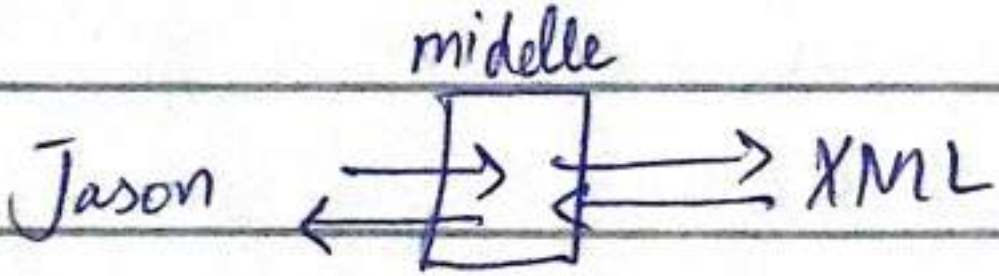
- Split one operation into several methods to improve cohesion and readability.
- Put the semantically related statements near each other physically within your program.

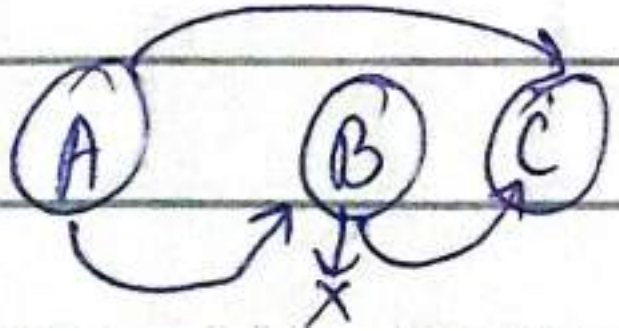
SCD

28-11

=> Refactoring = CODE COMPLETE CH#24

-> Cohesion





How to refactor a GOD Class?

Step 1

- Identify/categorize related attributes and operations
  - from class diagrams
  - put together the related items
  - find natural home of operation in related class

Step 2

- Remove all transient association
- Association that should not be directly accessed → transient association
  → Must accessed indirect

Transient: A property of any element in the system that is temporary.

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

Example: Library class should not directly

Refactoring:
500% Return over investment (ROI)

access instead it should access it through catalog class.

⑤ → Add a new feature to code that is not well designed

Assumption: You have a plenty of time (not always true).
↳ Usually takes time

① write unit test that verify external code's behaviour correctness.
② Low level refactoring
③ High level refactoring
④ Add new feature (4th step)

Cost

- Usually developers don't want to refactor.
- Managment don't want it (Not new thing made)
- Time

Benifits

① 500% ROI
~~② Conductive code is conductive~~
② Well structured and well written code is conductive to rapid development

③ Programming morale.

④ ~~Prof~~ Programmer preffer~~ed~~ to work in "clean house" (well structured and written)

clean house → Code environment

SCD

When to refactor?

- Best practice. Continuously as a part of the development process

It is hard to refactor your software in the project. Reason later in the project a lot of features are added and changes affect impact huge part / features of software

Reasons to Refactor

1- Duplicated code
2- A long routine (improve system by introducing modularity)
3- Long & deep nested loops
4- Poor cohesion, A class has more than one 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

other classes then its own attributes/feature.

10- Inheritance hierarchies are not modified in parallel.

11- Primitive data type is overloaded

12- Global variables

13- Improper / No comments.

14- Subclasses does not fully uses parent's classes.

15- Public data members.

16- Poor names.

17- Middle class/middle man is'nt doing anything. (Direct communication)

Tramp data: Data that is passed from one routine to another

18- Passing data to other routines without any usage/modification is called tramp data.

Levels of Refactoring

1- Data level refactoring
2- Statement level
3- Routine / Function
4- Class implementation
5- class interface
6- System

SCD 6-12

Software Evolution

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

Types of maintenance

1- Corrective maintenance

This encompasses fixing bugs / features.

2- Adaptive maintenance

This include software adaption to changing needs.

3- Perfective maintenance

This caters ~~import~~ improved software in terms of performance & maintainability.

4- Preventive maintenance

This type of maintenance deals with the improved software by fixing bugs before they activate.

Manny Lehman → Father of software evolution

S-type → Static

E-type → Evolutionary (Real world systems)

Software Evolution Laws

1- Law of continuing change

2- Law of Increasing complexity

3- Law of Self regulation

E-type system evolutionary process is self regulatory. with distribution of product and process

-> Size of software, time between releases, number of reported errors.

↳ If same development team makes the releases then all of the releases will have the same time and the ~~bugs will be reported by the same~~ team is reporting the bugs.

5- Law of Conservation of ~~Organizational~~ Familarity Stability

The average incremental growth rate of e-type systems tends to remain constant over time or decline over time.

-> Mastery of the system decreases

5- Law of Conservation of Familarity

6- Law of Continuing Growth

7- Law of declining quality

8- Law of feedback system

4- Law of Conservation of Organizational Stability

Average activity rate in an e-type process tends to remain constant over system lifetime or segment of that lifetime

Legacy

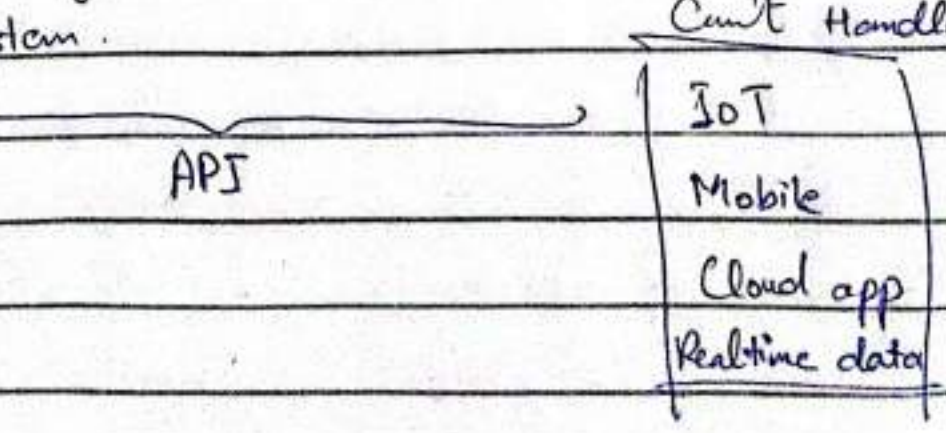
Outdated computing software and/or hardware that is still in use.

Challenges

- Mission Critical
- Not equipped to deliver new services
- Do not upgrade at the speed and scale of user expectations.

Worst case Scenario

- Legacy system can't connect to the new system.



5-12 Continue...

SED

Deployment

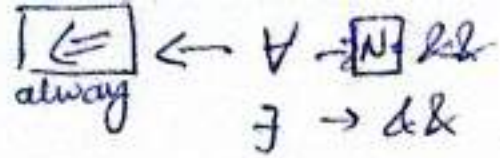
- This stage occurs at the end of active development of any piece of software
- It is more of an event than stage current technology wave automated deployment (cloud technology)
  ① Azure ② Amazon
- Deployment should not occur without rollback plans
- Must have a plan for backup/recovery.
- Deployment includes planned steps, problem areas & plan to recover.

Deployment Plan concerns

- Physical environment
- Hardware
- Documentation
- Training
- DB related activities
- 3rd party softwar
- Software executable

Deployment Focus

- Deliver software
- Revert on failure



Rollback

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

Reasons [Rollback]

- Installation does not go as expected
- Problems would take longer to fix then installation window.
- Keep production system alive
- Determine your point of no return before deployment.

SCD

High Level Refactoring

Significance

More important then low level refactoring
Improves overall structure of your problem

Principles

1- Exchange obscure language idioms with safer attenatives.

Example

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

2- 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
- Sorce code
- Build /Deplogment

Good class features

- Difficult to read
- Difficult to maintain
- Encapsulate collection

4- Refactor to design pattern
5- Use polymorphism to replace conditional
6- Introduce enumeration
7- Convert primitive 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 13-12

Source Code Layout & Style

"Any tool can write code that a computer can understand. Good programmers write code that human can understand."

— Martin Fowles.

Layout

- It does not affect execution speed and memory consumption.
- It affects how easy is it to understand the code, review & revision after months.
- It also affects other developers readability.

CH# 31

understanding and modification in your absence.

Fundamentals

① Logically Organized

- Proper use of wide spaces (indentation, new lines,

② Consistent

Rollback

Once deployed, customer want to change something get back and do again.
Rollback → Return to previous state.

Deployment Customer Stratigies:

These are plans / approaches to transition or change from old version to new.

- Cold Backup

A cold backup is a strategy where an environment is setup but remains offline until needed. It is not actively running. Suitable for system with lower tolerance.

- Warm Backup

where environment is partially active and synchronized with primary system.

- Hot Failover

Both primary & standby system runs simultaneously.

Fundamental theorm of Formatting

- Good visual layout shows the logical structure of a program.

Note: Technique make good code look good and bad code look bad.

Techniques:

- Proper use of ~~white~~ whitespaces
  (a) Grouping
  (b) Blank lines
  (c) Indentation
- Proper use of paranthesis

Style

A block of related/similer code use "~~begin~~" begin" and "end". It is clear by looking at the code that particular block of code starts or end.

Control Structure Layout

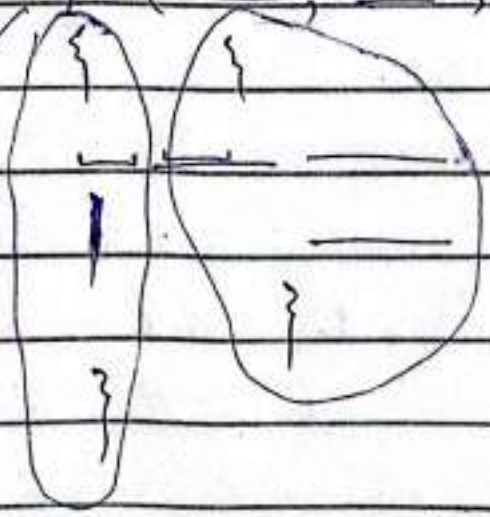
(a) Avoid unintend begin-end pairs

eg

```
for(int i=0; i<10; i++)
    Statement A
    Statement B
}
```

(b) Avoid double intendation with begin and end

eg




```
if(exp)
  stat_A;
if (exp) {
  stat.A;
}
if(exp)
{
  statementA;
}
```

(c) Use blank lines b/w paragraphs

(d) Format single statement block consistently

```
if (exp) statement A;
```

(e) For complicated expressions put separate expressions on seperate lines.

```
if (expA &&
    expB ||
    expC ){
        statement A;
    }
```

(f) Avoid GoTOs (It makes program hard to format)

(g) No endline for case statement (exceptional)

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

Not recommended

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

Recommended

Individual Statement layout

(a) Statement length

outdated rule : 80 characters max

now-a-day : 90 characters usually

(b) Use spaces for clarity & readability
- spaces in logical expression
- spaces in array refrences
- space in parameters

(c) Formatting continuous lines

(i) → Make incomplete statements obvious

```
while (exp A) &&
(exp B) && (exp C) {
    ——
    ——
}
```

Not recommended

```
while ((exp A) &&
       (exp B) &&
       (exp C)) {
    ——
    ——
}
```

Recommended

(ii) → Keep closely related elements close

(iii) → Indent routine call continuation lines the standard amount.

(iv) → Make it easy to find the end of a continuation assignment statements.

(v) → Indent control-statement / assignment statement / continuation lines the standared amount.

(vi) Don't align right side of assignment

statements.

(d) Use one statement per line.

(e) Data Declaration

- Only one data decloration per line
- Declore variables close to where they are first use
- Order declaration sensibly.
- In c++, put astertic ~~artiter~~ of pointier with variable name.

Comments layout

-> indent a comment with its corresponding code

->set off each comment with atleast a line

Routine layout

->Use blanks to seperate parts of routine

->Use standard indentation for routine arguments

Class layout

-> If a file has more than one class, ~~defi~~ identify each class clearly.

-> Put one class in one file

-> File name should be related to class.

-> Seperate routines within a file clearly.

-> Sequence routine alphabeticaly

SCD

CH#6 SWEBOK → Section 1

Configuration

↳ set up

Software Configuration Managment

→ Closest to SQA

→ Knows all the phases of software development life cycle (SDLC)

Configuration

Functional and physical charateristic of hardware and software mentioned in techical documentation or achieved in a product.

Planning for SCM

Software release Management

→ Identification Package and deliever a elegant product.

Release:

→ Executable program → Release Notes

→ Documentation → Configuration Data

Concerns:

→ When to issue a release

→ Product delivery items