Naming conventions ( clean code )

Proper Folder Structure for solid.

1st june clean code :

Things to cover : Clean code , Basic Folder Structure for SOLID

**Clean code**

1. Clean code Formatting : Separate concepts vertically , related code vertically dense,

Related , dependent functions closed , white spaces ,

Indentation

1. Clean code cheat codes
2. Naming Conventions

Class - Should start with Uppercase letter , noun

Ex : Button , System etc..

Interface - Should start with Uppercase letter , adjective

Ex : Runnable , Printable etc..

Method - Should start with lowercase letter , verb

If multiple names are there it should start with small next Letter

Upper case

Ex : draw() , actionPerformed()

Variable - Should start with lowercase letter , no - &,$,\_

avoid single letter variables like x , y.

Ex : id , name

Package - Should start with lowercase , separated by dots.

Ex: com.javatpoint

Constant - Uppercase letter such as RED,YELLOW

Separated by \_

Ex: MIN\_AGE

1. Best Practices

General Rules - Keep it simple always leave code cleaner than you started with

Easier to understand - Be consistent , Do all things in a similar way

Use variables which are self explanatory

Don’t write methods which depend on other methods to

work properly

Use whitespaces and Indentations

Names - Choose meaningful , pronounceable and searchable names

Replace magic numbers with Constants which describe

Avoid Intprogram etc..

Comments - Explain why, not how , in your code

Don’t comment out code remove it

Clarification , warning

Objects and dataStructures - Do one thing , Hide internal Structure

Code smells :

Rigidity - should not be there , no domino effect ( should not be rigid

easier to change )

Fragility - software should not break due to single change

Immobility - code should be replaceable elsewhere

Opacity - hard to understand.

**SOLID design principles**

What ?

Subset of principles promoted by **robert c martin** . The solid acronym was first i

introduced by **michael feathers.** readability , extensibility , reusability.

Why ?

Design principles that enables to manage most of software design problems

**Understandable , flexible , maintainable.**

S - Single responsibility principle

0 - open / close principle

L - liskov substitution principle

I - Interface segregation principle

D - Dependency inversion principle

**S - Single responsibility principle**

**“ A class should have only one reason to change “ - Robert c martin**

Every module or class should have responsibility over a single part of functionality.

**O - open/close principle**

**“ Software entities should be open for extension , but closed for modification“**

The design and writing of the code should be in such a way that new functionality should be added with minimum number of changes

**L - liskov substitution principles**

Barbara Liskov - **“ Objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program”**

If there is a Base class , then reference to the Base class can be replaced with derived class without affecting the functionality of the program module.

**I - Interface segregation principle**

**“Many client specific interfaces are better than one general-purpose interface “**

Don’t enforce clients to implement interfaces that they do not use , one big interface can be broken into smaller interfaces

**D - Dependency inversion principle**

**“ depend on abstractions , not concretions “**

Abstractions should not depend on the details where as the details should depend on

Abstractions

High level modules should not depend on low level modules.

If we Don’t use :

* End up with tight coupling
* Unknown issues
* Code which is not testable
* Duplication of code
* New bug fixes raise while fixing another bug.

If we use :

* Reduction in complexity of the code
* Increase readability , extensibility and maintenance
* Reduce error and implement reusability
* Better testability
* Reduce tight coupling.

Normalization

To eliminate unwanted redundancy

No modification anomalies ( unexpected side effect from a row operation )

Business processing - single table design

Transaction processing - five table design

**Functional dependencies :**

Value neutral constraint similar to primary key and foreign key constraint

Any value as long it does not match the primary key in the existing row.

Value of column in one table matches the value primary key of column in another table.

Specification of functional dependencies cannot be automated because understanding of business rules and requirements is necessary.

X -> Y

Almost one y value for every x.

X should be unique.

**SQL Constraints :**

1. **Not Null -** Makes sure that the column does not have null values

Specifying not null key word

1. **Unique** - Unique key ensures that all the values in the column are different

Specifying Unique keyword and column name

1. **Primary key** - combination of not null and unique key , uniquely identifies each row in a table.

Specifying as primary key in the table

1. **Foreign key** - Prevents action that would destroy links between tables

Specifying constraint and mention the foreign key and the table it

References

1. **Index** - used to create and retrieve data from database very quickly.

Specifying Index keyword and mentioned the column to be used as index.

**Query Optimization :**

**−**

* **Use Index** − indexes should be created during the logical stage of the program they make the processing faster by giving location easily
* **Optimizer functionality** - Driver index and list merge
  + - Driver Index - Does not read everything goes to second condition on AND based on first
    - List Merge - Costlier than driver index , reads the table twice , for before and after the AND clause
* **Use select** − using select statements correctly can be effective in optimizing the code
  + Complex - indexes
  + Simple select - one table , from clause simple
    - Sequential search
    - Binary search
    - Indexes
* **Join Implementation**
* Set operation
  + Union
  + Intersect
  + Difference
* **Aggregate Functions**
* Aggregate Table − It can be used to pre-populating tables at higher levels so less amount of information is required to be parsed.
* Vertical Partitioning − It can be used to partition the table by columns. This method reduces the amount of information a SQL query required to process.
* Horizontal Partitioning − It can be used to partition the table by data value, most often time. This method reduces the amount of information a SQL query required to process.
* De-normalization − The process of de-normalization combines multiple tables into a single table. This speeds up query implementation because fewer table joins are required.
* Server Tuning − Each server has its parameters and provides tuning server parameters so that it can completely take benefit of the hardware resources that can significantly speed up query implementation.