**CORE DESIGN PRINCIPLES FOR SOFTWARE DEVELOPERS**

**Q.) What’s a good design?**

->Design is good if cost of changing it is minimum.

->A design which is easier to change.

->It is almost impossible to get it right the first time. So, as a developer we should give time to really create better quality softwares and we have to really on what we do rather than creating once and be done with it. Software is never written, it is re-written.

**Q.) How to create a good design and evaluate it?**

->To create good design, first step is to let go of the ego. You should have the right amount of ego to start with a design to make it good.

-> To create good design, be unemotional. When we are unemotional we are not attached to the solution and we are really into solving the problems we have on hand.

->We should challenge people for that design so that we can create it more better as we will face different ideas from different individuals.

->Take time to review design and code because when we review somebody else’s code we help make them better in design and we learn from their design and code as well.

**----KIS Principle (Keep it simple) ----**

->Keep it simple and stupid.

->We have care to about our urge to create complexity as we can’t believe how much we ended up creating complexity. More the complex code, it is more difficult to modify it.

->Simple keeps you focused. Simple means use of declarative approach rather than creating using imperative approach. (Declarative tells what, Imperative tells how)

->Simple solves only real problems we know about. So before writing code or designing we should first think how to solve the problem rather than simply writing the code.

->Simple fails less.

->Simple is easier to understand.

**Note: -** Simple is not necessarily familiar.

**----COMPLEXITY----**

Two types:-

->**Inherent:**

* It is the complexity from the problem domain. There’s nothing we can do about it. It is the application nature of domain and we have to deal with it.

->**Accidental:**

* It is the complexity that we deal with most of the time.
* It comes from the solution that we use to solve a particular problem. The solution we are using brings certain complexities into the table and to solve those complexities we deal with new solution which brings new complexities and we get dragged into vicious cycle easily and it gets really hard.
* Example of accidental complexity can be concurrency

----Ask ‘whether something is simple or complex?’

----Then ask ‘is it inherent complexity or accidental complexity?’

----Then we can argue about good design

1. **A good design is the one that hides inherent complexity and eliminates the accidental complexity.**

**----Think YAGNI----**

->It is one such principle which can help to remove complexity.

->It stands for ‘You Aren’t Gonna Need It’. Attach a ‘y’ in the end of it, then it becomes ‘you aren’t gonna need it yet’.

->It is really important to postpone things until they are really needed.

----We should ask ‘When should I implement something?’

---- ‘How much do I know about that implementation or something?’

---- ‘Cost of implementing particular code or feature’

----Now Later

$N > $L -postpone

$N == $L -postpone

$N < $L -High probable or low probable? -do it now (high) else postpone (low)

->We should not misunderstand postponing to procrastination. Procrastination means not doing something that should have been done yesterday.

->Courage is to postpone things tomorrow to tomorrow, it needs courage to postpone things and say if we wait longer, we can get more visibility and can do better work than today.

->Courage is don’t rush through than to really giving time.

->Not do stuff that we don’t really need to do.

----‘Why should we postpone?’

----Manual Testing

-in the end if someone says let’s change-are you crazy

----Automated Testing

-in the end if someone says let’s change-give it a try

🡪 **So it means YAGNI principle is not very great if we have a poor feedback loop**.

🡪 **So good design is a design that is automatically verifiable as well**.

----If we want to postpone we need to have a good automated testing, without it things can be very difficult.

**----COHESION----**

->It is where a piece of code is narrow, focused and does one thing & one thing well which is cohesive code.

->Cohesive code is the code that is not taking on several responsibilities, it is only focused on doing something.

->For example, if we want a software to change, but not too expensive. It means if a code is cohesive, it has to change less frequently.

->Cohesion lowers the cost of developing software.

**----COUPLING----**

->Degree of connectivity among things.

->Worst form of coupling----Inheritance.

->Try to see if you can remove coupling.

->Dependencies should be handled if we struck into a design. So we should ‘knock out before we mock out’.

->We can’t remove all the dependencies as system will become very stable and it will not run.

-> Dealing with coupling:-

🡪Make it loose coupling instead of tight.

🡪Get rid of it.

->Dependencies on a class is tight coupling whereas on interface it is loose coupling.

->Use caution rather than throwing interfaces.

🡪**A good design has high cohesion and low coupling.**

**----KEEP IT DRY----**

->DRY stands for ‘Don’t Repeat Yourself ’.

-> Don’ duplicate -code and efforts.

->Every piece of knowledge in a system should have a single unambiguous authorative representation.

->It reduces the cost of development.

->We can use tools to remove duplicity. Example – CPD (Copy paste detector).

->Tend not to create abstractions or functions, tend to refactor the code on second time when we need it. (Postpone until u don’t get the sign)

---- ‘Why should we care?’

🡪The future you will thank you.

**----FOCUS ON SINGLE RESPONSIBILITY----**

->When we write a class, we start writing methods in the class. We should first focus on what is the primary responsibility of this class or this method.

->We should first define the primary responsibility of a class and the focus on that single responsibility.

->**Long Methods** are the general case of facing most of the problems. It is hard to test, hard to read, difficult to remember, obscured business rules, hard to reuse, leads to duplication, many reasons to change, can’t be optimised by anything, lot of variables used, low cohesion and high coupling, not developer friendly, hard to debug, etc.

->**SLAP (Single Level of Abstraction Principle)** (Where did u go to despite of what are the details of the place you went). Focus on single level of abstraction.

----Commenting is used in bad codes.

----Don’t comment what, instead comment why. (Make the code readable)

->Then we can achieve **Composed Method Pattern** (Easier to understand).

**----Don’t violate the Open-Closed Principle----**

->Software Module (class, method, etc.) should be open for extension but closed for modification.

->We should better know first what kind of change it actually needs.

->Abstraction and polymorphism is the key to make this happen.

->Two options to change a code for enhancement:

1. Change the existing code

2. Add a small new module of code

---- (We choose the second option because changing an existing code is really difficult rather than adding a small new module of code)

->Try to make code extensible. So to make the code extensible, we need to know the software and domain.

->But don’t assume that every software is extended is extensible or should be extensible because that can be very expensive.

**----Liskov’s Substitution Principle ----**

->Keep in mind about Liskov’s Substitution principle while designing softwares.

->Inheritance has been overused.

->Inheritance should be used only for substitutability.

->If an object of B should be used anywhere, an object of A is used, then, use inheritance.

->If an object of B should use an object of A, then use composition/ delegation.

->Inheritance demands more from a developer than composition or delegation does.

->Services of the required class should require no more and promise no less than the corresponding service of the base class. Because the user of a base class should be able to use the instance of the derived class without knowing the difference.

->We can be more generous in the derived class but we should not be more restricted.

->Derived function cannot throw any new checked exception not thrown by its base unless the new exception extends the old one because then we are violating the rules or contract through inheritance.

->Collection of derived does not extends from collection of base class.

->So use composition or delegation instead of inheritance unless you want substitutability.

->So you can use inheritance only when you can guarantee that the user of your base class can use the object of derived class without knowing the difference.

**----Decouple using the Dependency Inversion Principle----**

->A class should not depend on another class, they both have to be depend on an abstraction (Interface).

->Use caution.

->Person should not depend on a concrete class (clock), instead both the person and concrete class should depend on the Alarm.

**----Keep Interfaces Cohesive with Interface Segregation Principle----**

->It is a cohesion-single responsibility principle at the interface level.

-> For example, a clock class implements Timepiece, Alarm and Radio, so the user need not to know about all the implementation through clock, user is focused on its use, like user 1 wants timepiece (so it is focused only on time not on alarm and radio).

**Q.) When and how to apply these principles?**

->Principles learned so far:

1. DRY
2. YAGNI
3. SOLID(Single Responsibility Principle, Open-Closed Principle, Liskov’s Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle)

->Almost every time we code or design softwares we need to think about or apply all these principles.

->Use these principles in this way:

1. Through communicate between colleagues.
2. Through Test-Driven Development (TDD).