**Introduction note**

Thanks, <<>>, for the introduction. Hello all Good afternoon and thanks for joining this discussion. I am very excited today to speak about an interesting topic of Adaptive code, a way to write code that is easy to maintain, extend, and understand.

Code is all about intent, as it is built from a design solving a business problem in a given context. If the code doesn’t reveal the intent, in a way it is not solving the business problem.

The problem with code not revealing intent is like a trail of missing bread crumbs, problem with that is navigation through code becomes difficult and meaningless. This leads to a lot of maintenance issues as “WHY” does this code exist is a question, purpose is missing and business need is not getting covered. The cost of this mess is also enormous.

During our discussion, we will go into details on how to make code adaptable to solve the problem of the missing trail of bread crumbs using a set of proven practices called design principles, mainly focusing on SOLID.

With the context set, let's look into the agenda of today's discussion.

**Agenda**

We will discuss the developer mindset that has to do with design and software development? We will discuss the emotional journey a developer goes through when the code is meaningless.

Change is always constant and is inevitable. I shall discuss what blocks change.

We will discuss why software rot happens? What symptoms help us to identify design rot?

We will discuss design principles, SOLID principles, and foundations of SOLID that help to solve the design rot problem.

Deep dive into two principles of SOLID - mainly SRP and OCP with code examples. Let’s also discuss violations and symptoms in detail.

**Developer’s reaction and mind set:**

This image clearly explains what's going on when a developer is reading code and trying to understand what it is doing?

If we ask a developer what they spend their time doing, they will tell us that they spend most of their time writing code.

However, if we probe them and observe, developers spend most of their time trying to understand code written by them or someone else.

I have seen myself thinking about why I wrote this code after coming back to work from a break.

If we look into the ratio of time spent reading versus writing code, it is well over 10 to 1. In a way, it indicates if we make the code easy to read, it makes the developer easier to write. So, looking at the ways to make our code more understandable seems like the right thing to do.

But does that happen? May be or may not be (Craftsmanship skills does help, no question about it).

Some of the feelings developers will go through, one start to think what is this code doing, it doesn't take care of requirements, there are a lot of gaps. What the hell is this code doing? I will delete the code and rewrite everything. These feelings of the developer start increasing day by day if they are working not just on legacy code but possibly on the new code built.

**Work Around:**

In the picture, a plumber instead of removing the stone and laying the pipe, it is extended above the stone

At a stage after going through many feelings and reactions, developers tend to do, start thinking about workarounds to solve the problem in the given context. There could be many reasons for this, could be with the pressure of delivery and lot many other factors. Now, what could be a workaround? It could be so-called interim fixes, temporary code changes, cut copy and paste from one place to another place, creating lots and lots of hacks in the code

Let's ask a developer question on why opting for a workaround instead of doing it right? One of the popular answers is risk avoidance and why risk avoidance is to avoid unwanted side effects in the application by introducing breaking changes.

It could also mean they want to avoid bugs that may come in because of some of the changes done.

Finally, they don't want any further regression testing to impact the delivery.

It's all about avoiding any issues that may occur. Is this bad? It May not be given the context. The problem with ignoring these issues leads to potential damage in the future.

Result:

What could be the damages if we ignore these issues? The code is already messy, and developers are adding further mess.

Since we already understood that majority of the time, the developer spends more time to understand code than the real development, the overall time to market further increases.

With all these, maintenance of the code further degrades, leading to an overall increase in maintenance costs.

If we continue to build applications on top of poor code, technical debt increases, which is not a good sign.

Why do you think this happened? The code is not adaptable to change.

In a book named Refactoring for software design smells, Grady Booch, in his foreword, compares software-intensive systems with a city. His view is that when cities grow and evolve, some parts will flourish, and some will die. Cities must be renewed refactored considering the increase in population from time to time.

The same analogy is applicable for software-intensive systems. Systems grow and evolve from time to time, some of them flourish, and some die naturally. Some systems flourish because they are adaptable to change with time.

"CHANGE" for software systems is inevitable

How software systems adapt to change?

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. We all understand that there are multiple levels of software design.

Architectural design is at a solution domain level.

High-level design is a view of subsystems, modules, and their interactions.

Detailed design is the implementation part. It defines the logical structure of each module and its interfaces to communicate with other modules.

Can we build a system with a set of components or modules without any dependency? I don't think so. Interactions are necessary for the application/systems to work. When components/modules interact, dependencies get created.

Two aspects that help in creating well-defined dependencies are cohesion and coupling.

Cohesion is the measure of how elements of the class/component/module are related to each other. It is good to have high cohesion, and it indicates that functionally related aspects are together.

Coupling is a measure of interdependencies between modules/components/classes. It is good to have low coupling, and it is a measure of good design.

If dependencies get managed well, the design is adaptable, or else it's the beginning of software design rot. Next would be to understand the symptoms of software design rot.

17 minutes

Symptoms:

Rigidity

Rigidity is the tendency of a system that is hard to change, as one change cause another change and it goes on and it will be very frustrating.

The software is a compound word and is flexible.

When can we call the design rigid? Let's take a case, a developer is asked to fix some bugs in the application, and an agile master comes and asks for the timelines for fixing.

We might say it will take a day to fix. You know exactly where to fix, but when we started to analyze, the impact of that fix can be profound. The bug fix can span to one or more modules and is hard to fix as ripple effect in a way.

When agile master asks for a reason, the typical answer we say is, it is complicated than I thought.

It is also possible that developers will end up applying for the workaround. It could be a series of if-else statement written to fix the bug.

Fragility

This symptom is related to rigidity. Fragility is the tendency of the system to break in surprising places that are not localized when some part of the code is changed. Dependencies are so intense that a small fix can make multiple dependencies to break.

Let's take an example, a developer is asked to fix a bug in the code, and with the analysis done he finds the place where to fix the bug and fixed it. Once the testing is done and deployed to production it started to break in some other areas.

Immobility

The tendency of the system to have desirable parts for other systems that can’t be moved. We have to put this feature in this system which is already there in some other system but you can’t take that code to new system.

Viscosity

Tendency of system to slow down the right thing to do than the wrong thing. This is seen when the system is messy as it is always easy to add further mess than fixing the mess.

These are symptoms are bad design.

SOLID Principles/design pricniples help in making the design less rigid, less fragile, less immobile

Neglect is Contagious

Broken windows theory a psychology 25 years old concept about the psychology of humans

If we walk next to a building good looking and had a lot of glass windows, do we pick a stone and throw it on? We don't

On the contrary, if we walk past a building which has only one glass good and the rest all broken? Whether you should or not is a different question, but for a second, we get tempted to break the one glass window.

This analogy of a broken window concept applies to software. If a module is developed with the right intent

https://www.tutorialspoint.com/software\_engineering/software\_design\_basics.htm

http://se-thoughtograph.blogspot.com/2013/12/how-do-you-manage-dependencies-in-agile.html