Creating Evolutionary Design and Architecture

What’s Architecture?

- A design is, relatively, low level (class level, method level).
- An architecture is at a much higher level, relatively (components— a collection of classes, interfaces).
    - Details like component responsibilities, their interactions, and interfaces are architectural concerns.
        - Components could be a group of classes that are cohesive and are intended to provide one certain functionality or service.
    - How major pieces are communicating with each other.

Why evolutionary architecture?

- A few decades back, projects commonly used to be released after years of development. Today, projects are developed and released in as little as a few weeks or months — every quarter or every six months, a new version could be released.
    - If there is continuous deployment, then there might be several deployments *each day* — as soon as a feature is developed, it might be released.
- People tend to create the architecture towards the beginning of the project.
- Committing to an architecture in the beginning of a project can be disastrous. It is a very high risk.
    - Eg. deciding that the application has to on the cloud — it may not need to be, or maybe it doesn’t need to be on the cloud in the beginning.
    - The architecture should be allowed to evolve.
- The initial architecture we create should be treated as a **suggestion**.
- A good architect:

    - creates a team that can create architecture.
    - is a mentor who can facilitate team to learn and advance and create the architecture.
- Job of an architect isn't to deliver an architecture, but to facilitate the creation of an architecture draft in the beginning.
- You shouldn't commit to anything until you cannot stay without committing.
- Architecture is a shared vision.

The risk of not evolving

- As time goes, as we understand the details of the project, we tend to deviate from the reality from the architecture POV, and there's a mismatch between the reality of the project and the architecture.
- Extensibility issues.
- High cost of maintenance.
- Cost prohibitive when making changes.
- The architecture and the design end up being disconnected from the realities of the application.
- **Economic impact.**

The risk of evolving

- There's always a time when an entirely unexpected feature might pop up (both, in evolutionary and non-evolutionary arch).
    - In evolutionary architecture, it's much easier to make this change affordable.

How to deal with that?

- Features should be prioritising based on the architecture as well as business value.

Evolutionary should be a deliberate effort

- Evolutionary architecture is not an accidental approach. You can’t just ignore it and say that you’ll arrive at the architecture one day.
- Come up with an architectural draft — a vision and a set of suggestions — and come up with an architecture that seems to make sense based on the information known so far.

Keep it simple

- A simple solution is one that focuses on the real problem; it solves the problem as we know and understand it.
  - fails less
  - keeps our focus
  - easier to maintain and evolve
- Only solve problems that we know are worth solving.
  - If something is not worth solving, don’t solve it now. Postpone until it is necessary to solve.
- Keep independent things separate from one another.
  - If independent things aren’t kept separate, then it becomes hard to evolve and maintain them as time goes on and the needs of different things diverge.

Refactor

- Refactor the code quite a bit along the way.

Reversibility

- The ability to back out of a decision, called reversibility, is an important consideration in evolutionary design.
- In order to be reversible, we should design for it and actively decouple.

    - Eg. design your application in such a way that you can switch out your DB with another one with minimal changes. If your code is tightly coupled with a DB, then switching it with another DB will be very expensive and time consuming.
- Some decisions are reversible and other aren't.
    - Languages cannot be thrown out for completely different languages. You can create some components in some other language, but there isn't much flexibility there.
- If you want something to be reversible, you have to decouple from it. The "surface area" should be a very small area of contact.
- **Making *everything* reversible is going to be cost-prohibitive.**
    - Don't make things that don't deserve to be reversible reversible.
        - Eg. you have to pick a language and use it. Making your own language and having it point to other languages will only increase your work load and expenses.
- You don't want to, for an example, commit to a single database vendor.
    - Don't implement niche features available only on a single version of some database; this is not reversible.
    - Implement this as reversible at the architecture and the design level.
- Just because something can be reversed does not mean that it should be. And just because something is reversible does not mean that is implement in a manner that it can be reversed.
- If something is very hard to reverse, postpone the decision to commit to it as much as possible.
    - Eg. languages — don't decide on a language until the moment beyond which you cannot go without deciding on a language.

Last responsible moment

> Courage is postponing the decisions of tomorrow to tomorrow.
> -Kent Beck

- **When you have to make a decision, postpone the decision to the last responsible moment.**
- At every single level, see if you can wait to take a decision later.
    - Why? We'll know more at a later time. We'll know more about the context.
- **"One of the most important decisions is what to decide and when to decide."**

YAGNI — You Aren't Gonna Need It (yet)

> There is nothing so useless as doing efficiently that which should not be done at all
> - Peter Drucker

- If cost of developing now > cost of developing later, postpone.
- If cost of developing a feature later is the same as doing it now, we should postpone. This is because if the cost is going to be the same, might as well wait until you have more context/information/knowledge and build it then.
- If cost of developing a feature later is less than doing it now, we should
    - Ask — what's the probability of you need this? If it's very high, build it now. If it's less or you don't know, it may be better to build it later.

Tracer bullet

- "Like a laser beam, you see a trajectory."
    - This trajectory gives a good feedback loop.
- When using the tracer bullet approach we will actively
    - mock out parts — provide fake implementations and use them to evolve and replace those fake implementations with real ones (eg. you could use in-memory database instead of a real database in the beginning.)
- We're essentially trying to create something, play with it and see if it fits and then lock it in.

- Analogy → when putting together some furniture, you don’t screw the screws tightly right in the beginning. You put the screws loosely and once the entire structure is complete, you go back and tighten all the screws to ‘lock’ the entire structure.

Extensible

- Look first at how the teams are organised, not at the technology.
    - If you have a very rigid team structure (eg. a DB team, a Service team, a Security team), it becomes difficult to communicate between teams when you have to build a feature, and extensibility becomes difficult.
    - If the teams are organised across the technology stack, that is a big disadvantage.
- If we can minimise the impact of change, it’d be a lot better and easier to make the code extensible.

Triangulate

- Write a class first. When you need it, write another class. If there is enough commonality, *now* you can extract out a base class for these two classes.
    - Don’t start by building a base class/interface.
- When it comes to building an hierarchy, do not start with the the hierarchy, but end with the hierarchy; this way you’ll have an hierarchy that’s very practical, otherwise you’ll have something very complex.

Reuse?

- For something to be usable, it has to be specific and direct.
- For something to be reusable, it has to be general enough.
- How to create something reusable? Start by using it.
    - Eg. Spring, POJO, Rails are frameworks that were built as applications and then extracted out as frameworks.

- Standardisation is important, but so is innovation.
    - Don’t be quick to standardise — by creating a library or a framework.
    - **Standardise after Innovation.**

Parsimony

- Parsimony is minimalisms. Do the minimum you can and postpone whatever you can.
- The more libraries and frameworks your application depends on, that provides you efficiency, but it comes with enormous costs.
    - As these libraries evolve, you have to evolve your application with it. If a library is evolving very quickly or if they break compatibility, then you have to go and fix your application to stay in tune with it. If you don’t, you’re increasing your technical debt.

Minimise libraries and frameworks

- Before picking libraries, think several times. You should use libraries very reluctantly.
    - The cost of writing code by using some library may be small now, but the cost of maintaining the code may be quite high.
- Implementing the feature yourself can help you be confident that the feature will still run, confirmed by automated tests, down the line. This isn’t the case with libraries.
    - We know that our own code won’t have any compatibility issue later on because there is not external dependencies.
    - The cost of implementing the features yourself may be higher now, but the cost of maintaining may be minimal.
- Don’t bring in libraries because you can use libraries; bring in libraries if it’ll make you more productive than implementing it yourself.
    - Bring in libraries only if the effort to implement the features is too great.
    - Eg. you shouldn’t be creating your own database management systems.

- Don’t be infatuated with technology. It is important to pick things for the right reasons.

Postel’s Law

> When you are interacting with systems, be very conservative in what you send out and be very flexible in what you receive.

- For the data format of what we receive, it's better to be general.
  - Eg. `List<>` is more general than `ArrayList<>`.

```
class MyClass {
    receive(ArrayList<Foo> values {

    }

    receive2(List<FooInterface>) {}
}
```

- When you send data, don’t send verbose amount of data. Try to be minimalistic and send only what is relevant.

Testable

- Create with testability in mind.
- When you architect for testability, your components and classes become more modular; they become smaller and more cohesive and they also end up being decoupled.

Conway’s Law

> Usually the architectural impact/structure of an application corresponds to organisational structure.

Integrate early, Integrate often

- If we integrate too infrequently, it becomes quite difficult to integrate.
- If we integrate quite often or use continuous integration, it is relatively easier to integrate.