Clean Architecture

Dainish Jabeen

July 23, 2023

The goal of software architecture is to minimize the human resources required to build and maintain the system.
There are two values stakeholders care about: the behavior and structure of a system.
Behavior: Make the machine fulfill the users requirements.
Architecture: Soft ware should be easy to change, the difficulty of the change should be proportional to the scope of change and not the "shape". Shape being the type of change requested.
Eisenhower matrix: A digram showing the combination of important and urgent. The conclusion is that generally things that are urgent should not be important and things that are important should not be urgent.
Three big areas of architecture: function, separation of components and data management.

1 Object orientated

First area thought to be introduced is Encapsulation of data and functions.
Data should be kept within concepts and only accessed through this concept.
Good encapsulation is when the users of a program have or need no knowledge of the implementation of the data structure of function.

- OO does not require perfect encapsulation, modern languages have declaration and definition of the classes tied together needing knowledge of one and other to use them.

Inheritance: Able to reuse classes and modules in different scopes.
Encapsulation can exist without OO so this is not its feature.

1.1 What is polymorphism ?

At its core polymorphism is pointers to functions. Allowing the pointer to dictate based on the given parameter the functions behavior.
This can be done without OO, however it makes it much safer and more convenient.

1.2 How did OO effect the flow of dependency (Dependency inversion)

Prior to OO flow of control had to match the direction of dependency.

- All higher level functions are dependant on main and the flow of control stems down to them from main.

The dependency can be inverted using OO and interfaces; giving the architect control over the dependency tree. **Control order does not have to match dependency**
Green arrow = Control , Red arrow = dependency

- The UI can depend on the business rules and the control flow stem from the rules.

The main thing OO introduces is a safe effective method of controlling source code dependencies !!
Vital to allow modularity and plugin nature of development.
Functional Programming
Functional programs variable are not mutable, they do not vary.
All concurrent update problems derive from mutable variables

1.3 How do you utilize immutable variables

If there was infinite resources (memory and processing) using immutable variables would be an option. Instead need to split the program into mutable and immutable components.
The immutable components will feed the mutable components which then use transactional memory.
Transactional memory acts as disk database with safety measures against race conditions; locking mechanisms on reading and writing.

1.4 What is event sourcing ?

Store transactions instead of states to avoid mutable variables. The state is then calculated by summing transactions.

2 Components

Well designed components are independently deployable and therefore independently developable.
Three principles associated with component design :
Module managment tools: Maven, Leiningen and RVM.

2.1 Reuse/Release equivalence principle

Should be an overarching theme of the modules inside a component.
To allow effective developing and reuse of a component there should be scheduled releases of new versions. And therefore the classes and components should be releasable together as a unit.

2.2 Common closure principle

Gather classes/modules into a component that change change for the same reasons at the same time.
Much easier to change related classes in a single component than across many components.

2.3 Common reuse principle

Dont force users of a component to depend on a code they dont need. Places reused classes and modules together into a component. There should be a high dependency in a component between classes and modules.
If dependant on a component better to be completely dependant on the entire component.

2.4 How do you balance the three principles

It is dependant on the needs of the software system and there will be tension between all three. i.e Early on development CCP is more important than REP ! As development is more important than reuse.

3 Component Cohesion

3.1 Acyclic dependencies principle

No cycles in the component dependency graph.
Released components allow devs to work in isolated teams and decide which version to integrate with their dependant component.
If there is a dependency cycle between components multiple components will be forced to use the same version of a dependant components disrupting the isolated teams.
To break a cycle create an interface component which will inverse the dependency. Or create another component between.

3.2 Stable dependency principle

Do not make easy to change modules be depended on by hard to change modules.
A method to make software stable and therefore hard to change is to have a lot of software dependant on it.

3.2.1 How do you measure stability ?

By the number of dependencies:
Fan in: Incoming dependencies (Increasing this number is good for stability) Fan out: Outgoing dependencies
Instability : I = Fan out/(Fan in + Fan out), I=0 is maximum stability and I=1 is max unstable.

3.3 Stable abstraction principle

A component should be as abstract as it is stable.
A stable component should be abstracted so it can be **extended**, it should be hard to change however being extendable does not effect this.

3.3.1 How can you measure abstractness

Abstractness = Number of abstract classes and interfaces / number of classes in component

3.3.2 Metric to determine good design

Distance from main sequence :
D = | A + I -1 |

4 Architecture

Architecture is about deployment, maintenance and ongoing development. Not directly linked with behavior or operation however it can aid in this.
The focus is on reliability and development speed not performance.

As many options as possible should be kept open for as long as possible by good design. Until you are in the most optimum position to make a judgment on an option (database, language, hardware etc)
The concept is to ensure policy is completely decoupled from details.