Clean Architecture

Dainish Jabeen

August 9, 2023

Contents

1 Introduction

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.

2 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 or 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.

2.1 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.

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

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

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

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

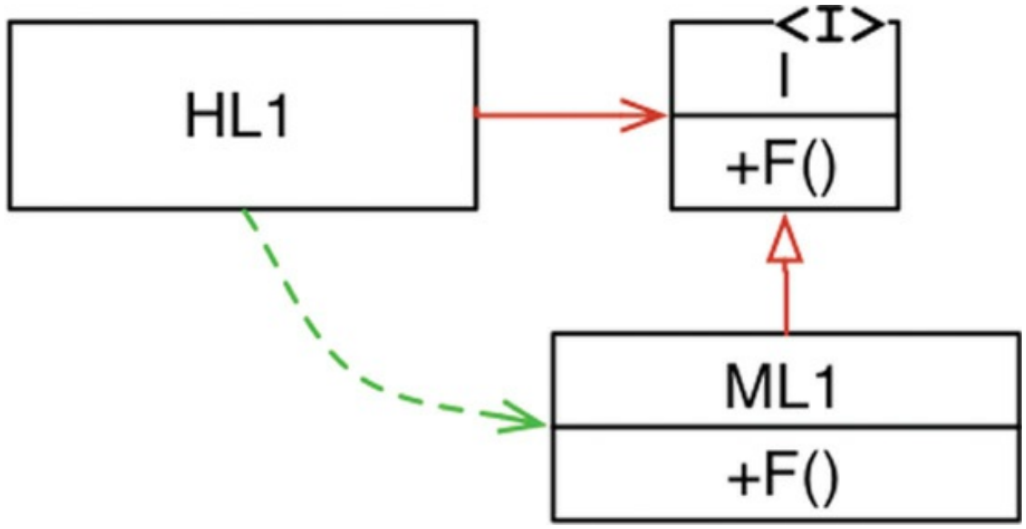

Figure 2.1: Control/Dependency Flow

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

Figure 2.2: Overview Flow

The main thing OO introduces is a safe effective method of controlling source code dependencies !!

Vital to allow modularity and plugin nature of development.

2.2.1 Functional Programming

Functional programs variable are not mutable, they do not vary.
All concurrent update problems derive from mutable variables

2.3 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.

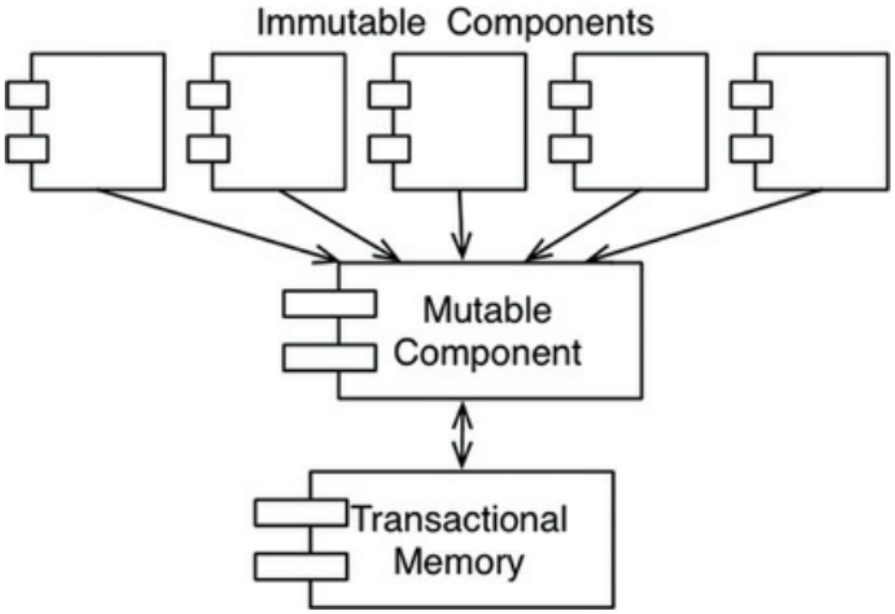

Figure 2.3: Immutable Components

2.4 What is event sourcing ?

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

3 Components

Well designed components are independently deployable and therefore independently developable.

Module managment tools: Maven, Leiningen and RVM.

Three principles associated with component design :

3.1 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.

3.2 Common closure principle

Gather classes/modules into a component that change for the same reasons at the same time.

Much easier to change related classes in a single component than across many components.

3.3 Common reuse principle

Dont force users of a component to depend on code they dont need. Place reused classes and modules together into a component. There should be a high dependency in a component's classes and modules.

If dependant on a component better to be completely dependant on the entire component.

3.4 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.

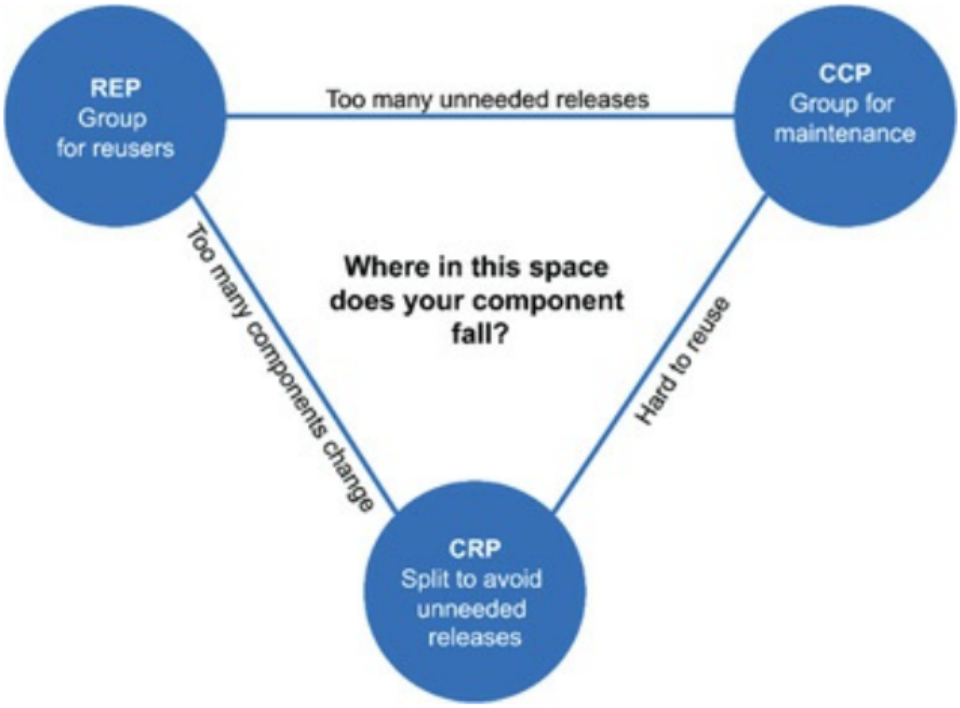

Figure 3.1: Balancing three principles

4 Component Cohesion

4.1 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.

4.2 Stable dependency principle

Do not make hard to change modules depend on easy to change modules.

A method to make software stable and therefore hard to change is to have a lot of software dependant on it.

4.2.1 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.

4.3 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.

4.3.1 How can you measure abstractness

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

4.3.2 Metric to determine good design

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

5 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. Good architecture is about making a system easy to understand, develop, deploy and maintain.

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.

Good architecture is about balancing when boundaries are needed and not needed and being flexible about this as it will be revealed as time progresses.

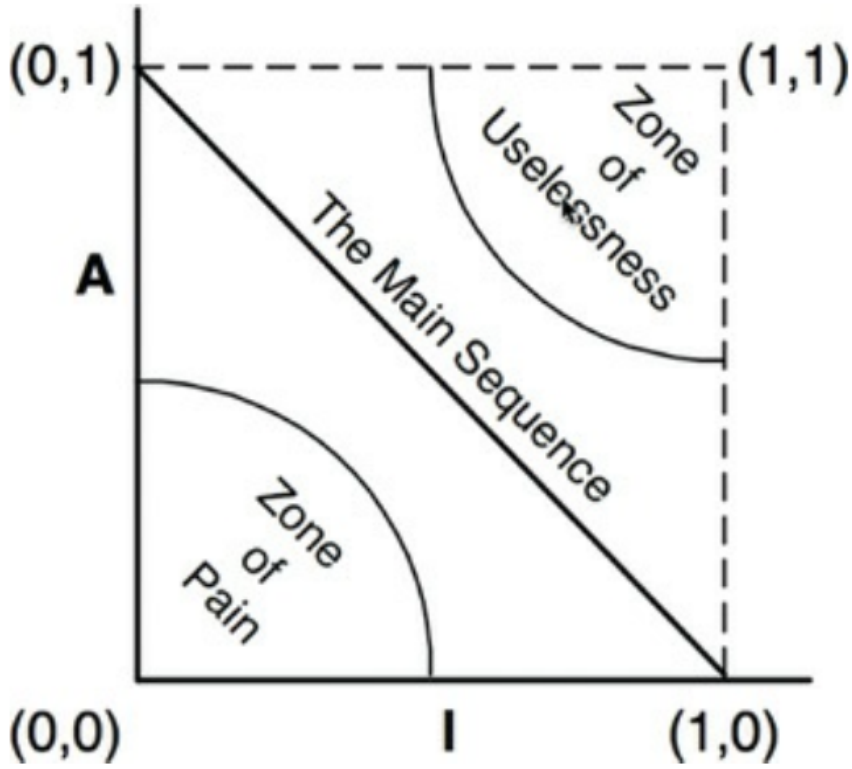

Figure 4.1: Abstractness measurement

5.1 Size

The needs for different sized groups will be different. Small groups may find many interfaces and complete modularization a hindrance. A component per team may not be the optimal method of balancing deployment, operation and maintenance even if it makes development easier.

I.e Following strict micro service method will add heavier interconnections burden and bottleneck deployment, even if it helps development via strong boundaries. Immediate deployment should always be the goal.

5.2 Maintenance

The primary maintenance cost is splunking; finding the best place in software to add a feature or repair. With the risk of changes causing unwanted bugs. Clean architecture should via screaming its use cases and strong boundaries limit this.

5.3 Use Cases

The use case should be made clear by the architecture, behavior/features are easy to find via prominent modules.

Many human resources have been wasted by decisions made on the details not relevant to the use case. These should be deffered as long as possible.

5.4 Diagrams

Good to simplfy diagrams by only showing interface points and using splits to show different data flow streams.

6 Dependency

Dependency can be thought of as import mechanisms, if a module or files requires an import or using statement then it is **dependant on the thing its is importing!**

An interface allows for dependency inversion without changing the flow of control. This is vital to a modularized methodology. The key thing to check in dependency is if changes or breakages in a module effect other modules, if they do they are dependant.

Generally an interface is called via a data object allowing for polymorphic methods and independence.

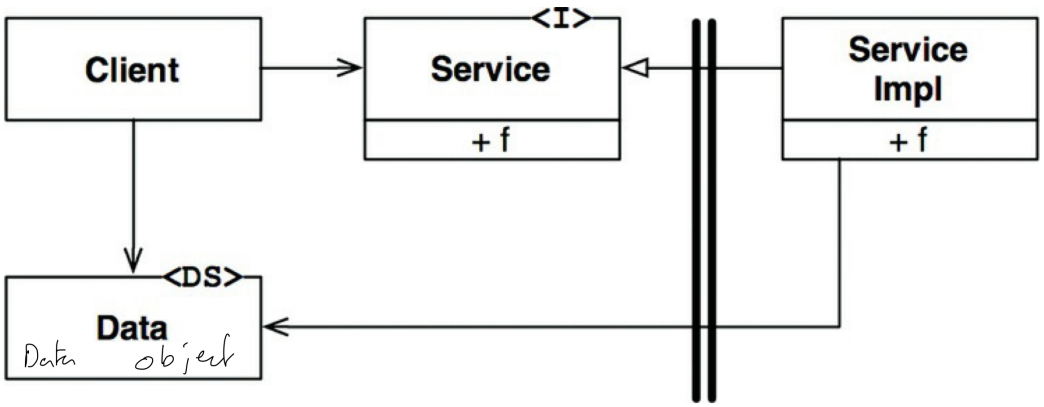

Figure 6.1: Interfaces boundaries

The lower level component should depend on the higher level ones. I.e DB access method should depend on the procedure that uses that data and not the other way. This allows for plugin design where details are dependant on procedures so they can be easily swapped.

Vital to remember to create reusable interfaces/frameworks, must create them concurrently with several plugins/reusing apps.

Error handling becomes clear when there is clear dependency as it can happen post interface.

7 Component Cohesion

A module should be determined by the axis of changes, is it expected for all elements in this module to change for the same reason and rate!

A module hard to change should be easy to extend.

8 Decoupling

Separate things that change for different reasons collect things that change for the same reason. (UI and procedures change for different reasons). All implementation details should change for different reasons than core procedures and therefore should be independent.

Technical elements are the horizontal layers, and use cases/actors are the vertical layers. A use case may need many technical elements across the layers (UI,procedure and DB).

This may cause duplication, however if this will eventually diverge (accidental duplication) it is acceptable.

The data passing between boundaries should not have structure that causes dependencies, as this will cause dependency inversion to be broken. It should be kept as simple as possible and then formatted after the boundary.

To determine coupling determine if any new features require coordination between components.

8.1 Methods

- Source code level: Monolithic structure, comms via functions calls
- Deployment level: Individually deployed units, comms via sockets or shared mem.
- Service level: Independent at source level, comms via network

Tip: Best to keep modules ready for separation as services, however not to separate unless good reason.

9 Policies

Policies lead to procedures and should be separated and grouped together based on axis of change. The grouped policies are components on acyclic graph.

Listing 1: Bad design

```
1
2         func encrypt() {
3     while(true)
4       writeChar(translate(readChar()))
5   }
```

This is bad design as the procedure encrypt is dependant on the details.

A better design would be: Figure 9.1.

9.1 Critical

The highest level procedures should be those that could be done manually, at the core of product. These can be considered critical procedures.

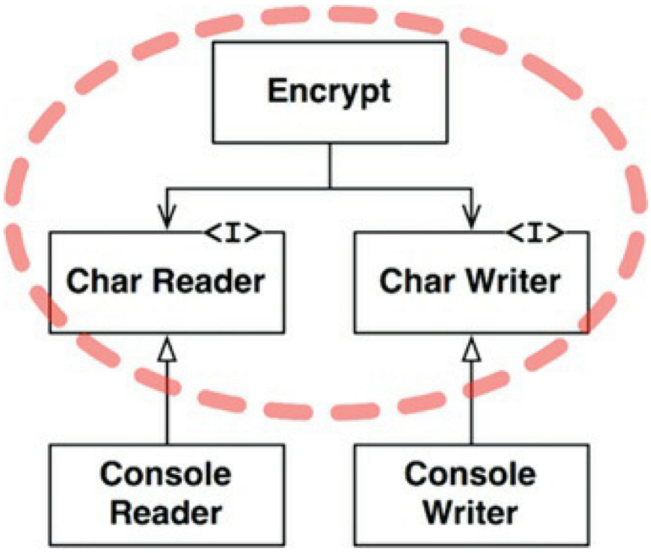

Figure 9.1: Improved Architecture

10 Partial Boundaries

If a full boundary might be needed in the future.
Facade classes are stand in place methods that don't have dependency inversion to access services.

11 Testing

Humble object pattern: Separate easy to test presenter and hard to test view.

The view is the detail hardware and the presenter is the methods used to get the data to it. The view module should be kept very simple to allow the rest of the code to be easily tested.

Interfaces allow views/controllers to be replaced with stubs or test doubles easily.

Fragile tests: Strong coupling between tests and code mean changes in the code cause all tests to break.

Create a test API to access the procedures, decoupling the structure of the tests and procedures.

12 Main component

This should oversee and coordinate everything, it is the lowest level policy. Nothing should be dependant on it, other than the operating system.

Main should setup initial conditions and configurations and hand this over to higher level policies. If designed as a plugin allows many different configurations.

13 Services

Services that don't follow the dependency inversion rule (using interfaces) are not architecturally significant, despite being computationally separate they maybe coupled via the data shared. If a change to the shared data causes many services to need change.

Each service should have its own boundaries dividing it into components, the real boundaries are not between the services they are interfaces communicating with each other. Unless the service is a single component surrounded by boundaries.

The benefit of service design is outside the goal of good architecture.

14 Embedded software

Issue of deep mingling between hardware elements and procedures.

14.1 Hardware abstraction layers (HAL)

Line between firmware and software is the HAL, the HAL handles the softwares need and access to the the hardware. The software should not know anything about the hardware instead use concepts. Ie BatteryLow NOT blinkLED.
The HAL also provides substitution points for the hardware to be used for testing to avoid the target hardware bottleneck.
Target Hardware Bottleneck: Code can be only tested on the hardware.

14.2 Processor abstraction layer (PAL)

Separates the firmware into separate pieces to allow off hardware testing. Making the processor a plugin element. Can do a similar thing with an OS.

15 Example

1. Determine actors and use cases
2. Create component architecture
3. Ensure dependency is appropriate for policy levels
4. Determine deployment method of components

15.1 Use Cases

Actors should be connected to use cases, abstract use cases can be used that connect other use cases which add details to the abstract.

15.2 Components

Example pattern: View, presenter, interactor and controller are separated

- Controller: Handle input
- Interactor: Process input into usable data
- Presenter: Format results
- Viewer: Display result

The vertical splits will be different actors/use cases.

15.3 Types

15.3.1 Layered Architecture

Boundaries drawn by technical difference, ie web, procedure and storage.

Doesn't have any link to the business or use case.

15.3.2 Feature Architecture

The boundaries are around features, but doesn't effectively separate technologies.

15.3.3 Component Architecture

Use boundaries and deployment method to separate technologies and use cases. A hybrid of the previous.

15.4 Encapsulation

If components have free access to each other, the architecture can be easily ignored. Utilizing encapsulation allows the complier to ensure the architecture.