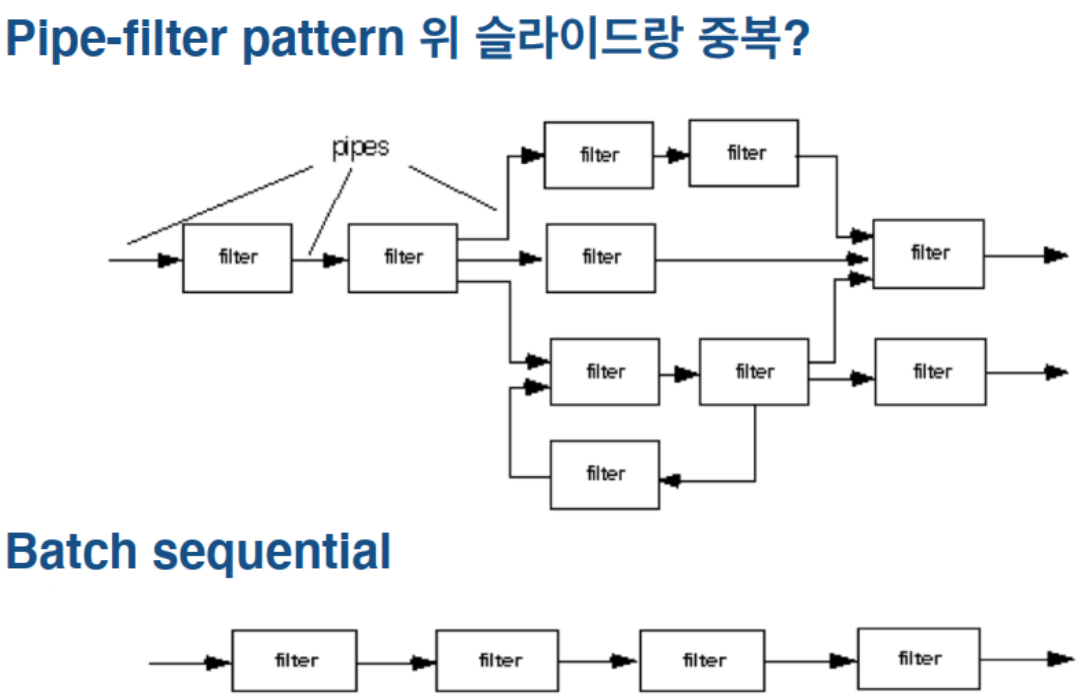
Data Flow Architecture



데이터가 데이터 연산 처리 순서를 지시하고 제어하는 데이터 처리 요소로 분리 될 수 있습니다. 각 컴포넌트는 입력으로 데이터를 받고, 출력으로 연산된 데이터를 출력합니다. 이렇게 출력된 연산 데이터는 다음 컴포넌트의 입력이 됩니다. 이 부분이 Data Flow Software Architecture 들의 가장 큰 특징입니다.

Data Flow Software Architecture의 장점은 변경 용이성과 재사용성입니다. 서브시스템은 서로간 독립적으로 구성되어 있습니다. 각 서브시스템은 서로간의 어떠한 영향없이 새로운 서브시스템으로 교체가 가능하며 중간에 새로운 서브시스템의 추가도 쉽기 때문에 아키텍처의 변경이 용이하고 각 서브시스템은 다른 아키텍처에서 재사용이 쉽습니다. 한 가지 유의점은 서브시스템 추가 시 출력되는 데이터 형태가 다음 서브시스템의 입력과 일치해야 합니다.

1. Pipe-filter

구성요소는 크게 3가지 입니다. 데이터 스트림, 필터, 그리고 파이프입니다.

Pipe and Filter Architecture의 장점은 Concurrency(동시성), Reusability(재사용), Modifiability(변경용이성), Simplicity(단순성), Flexibility(유연함)입니다. Concurrency는 과도한 데이터 처리에 대해 각 필터가 독립적으로 동작하여 높은 처리량을 얻을 수 있습니다. Reusability는 각 필터가 독립적으로 동작되며 다른 필터와의 종속성이 없으므로 각 필터를 다른 시스템에 재사용이 가능합니다. Modifiability는 필터 간 종속성이 낮기에 새로운 필터를 추가하거나 수정, 제거했을때에도 시스템에 다른 수정을 최소화 할 수 있습니다. 두 필터 사이의 파이프가 존재한다는 매우 단순한 구조를 가지고 있으며 각 해당 시스템의 데이터를 Sequential하게 Parallel하게 수행이 가능함으로 유연한 구조를 만들 수 있습니다.

하지만, Pipe and Filter 구조에도 여러 단점이 있습니다. 데이터 스트림 형태가 고정된 형태의 구조이기에 동적으로 데이터 포맷을 변경하는 구조에는 알맞지 않습니다. 만약 A 필터로 이미지가 입력되었는데 출력으로는 XML 포맷으로 출력하고 B 필터는 XML을 입력받아 Character Stream으로 출력한다면, 이 구조의 장점인 변경용이성, 유연함을 잃어버리게 됩니다.

2. Batch sequential

각 데이터 전송 서브시스템 또는 모듈은 이전 서브시스템의 데이터 처리가 끝나기 전에는 스스로 시작할 수 없습니다. 정리하자만 A-B 로 연결된 Batch Sequential Architecture에서 B는 A가 모든 데이터 처리를 완료한 후 결과 데이터가 출력되기 전까지 스스로 독립적으로 시작할 수 없습니다. 데이터를 분리해 중간중간 처리가 아닌 하나의 서브시스템이 데이터를 처리한 전체 결과를 출력해야만 다음 서브시스템이 시작할 수 있습니다.

Batch Sequential Architecture는 서브시스템들이 단순하게 분리되어 있고 입력 데이터와 출력 데이터에 맞춘 서브시스템의 교체도 가능합니다. 서브시스템간 연결은 오직 데이터 이므로 데이터만 맞추면 됩니다. 하지만 외부에서 서브시스템을 제어하기 위한 구현에서는 부적합하다. 또한 동시성을 지원하지 않기 때문에 낮은 성능과 높은 Latency를 가지는게 이 아키텍처의 한계입니다.

차이 :

Batch Sequential은 데이터가 처리되고 다음 데이터 처리 단계로 넘어가기 위해선 이전 데이터 처리가 모두 완료되어야 합니다. 즉 A에서 B 처리 단계로 데이터가 전달되기 위해선 모든 데이터가 A처리가 완료된 이후 B로 입력됩니다. 하지만 Pipe And Filter Architecture는 데이터 스트림 처리를 위한 구조로서 A 단계에서 모든 데이터가 처리되고 B의 입력이 되는 것이 아닌 A 단계에서 먼저 처리된 데이터는 바로 B의 입력으로 Pipe를 통해 전달됩니다. 100개의 데이터가 있을 때 Batch Sequential은 100개가 모두 처리된 이후 다음 스텝으로 입력되지만, Pipe And Filter는 100개 중 처리된 데이터는 B로 전달됩니다. 이 점이 가장 큰 차이입니다.

Data centered

Repository - Repository에 저장되는 데이터를 중심으로 Client들이 데이터를 교환하는 구조. Repository는 수동적, Client 는 능동적으로 동작한다. Repository의 확장과 데이터의 백업, 복구가 쉽다. 데이터 저장소의 데이터 구조와 Client 간에 의존성이 크고, 따라서 데이터 구조의 변경에 따라 많은 영향을 받으며, 데이터 이동에 따른 네트워크 비용이 발생한다. 일반적인 DBMS에서 사용된다.

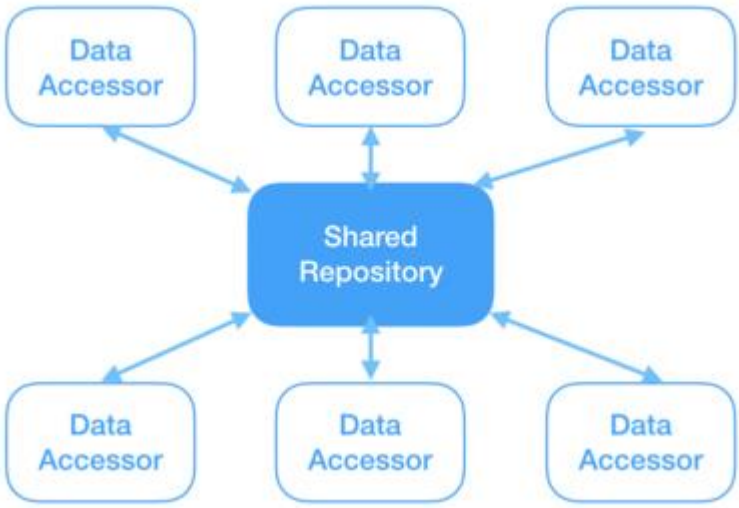
Blackboards - Repository와 비슷한데, 반대로 Data Store 는 능동적이고, Client 또는 Knowledge Sources(Listeners 또는 Subscribers)는 수동적으로 동작한다. 동적인 문제를 해결할때 효과적이며, KS를 재사용할 수 있다. 테스트 하기 어렵다. AI에 주로 사용

- Repository Architecture Style

: Data store passive; clients active | High dependency, vulnerable to failure replication

- Shared Repository Architecture Style

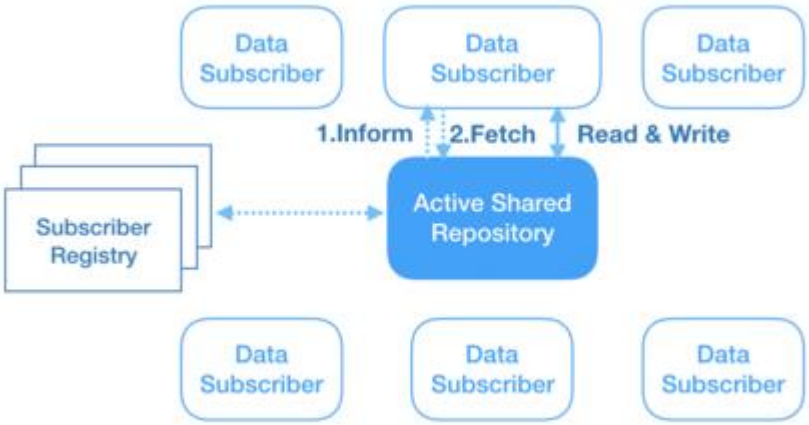
: Central repository 통한 Dataset 공유, 모든 Application에서 intermediate layer or agents없이 직접 접근이 가능, 공유에 효율 | Data schema 변경시 어려움. Performance overhhad, Security



- Active Repository Architecture Style

: Shared Repository의 slight 수정, passive sharing -> active sharing, 구독자의 레지스트리 유지,

repository 업데이트 시 notify

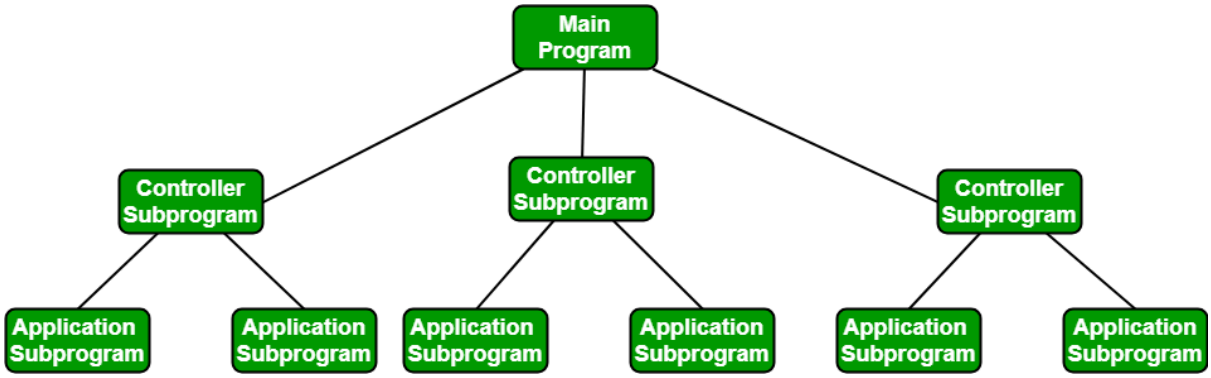


Call and return

→ 구조 설명용으로 쓰이는 듯

It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.

- **Remote procedure call architecture:** This components is used to present in a main program or sub program architecture distributed among multiple computers on a network.
- **Main program or Subprogram architectures:** The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.



Layered

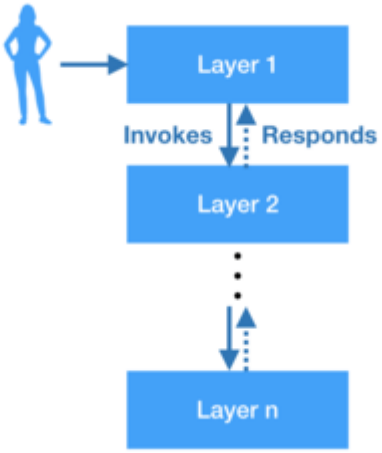
계층 하나를 서브시스템으로 생각하여 하위 계층은 서버, 상위 계층은 클라이언트 역할을 하도록 구성한다.

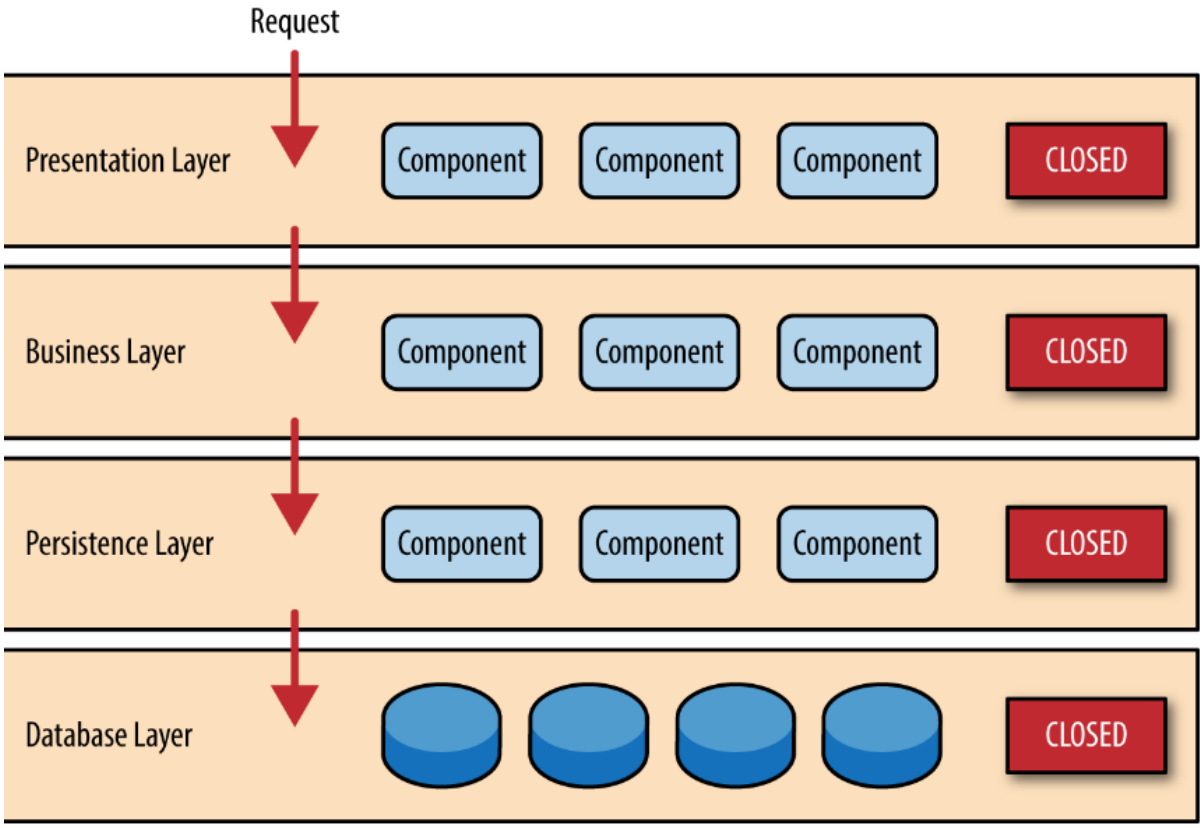
각 서브 시스템이 하나의 계층이 되어 하위층이 제공하는 서비스를 상위층의 서브시스템이 사용

추상화의 성질을 잘 이용한 구조

- Layered Architecture Style(*필수)

: Layer-group of related classes, 인접레이어 통해서 전달, 수직적 관계, 각각의 레이어는 Unique 역할을 가지며 immediate lower layer에서 제공되는 서비스를 invoke함. | 광범위한 적용이 어려움, 특정 상황에서는 레이어가 불편할 수 있음.





- 위의 그림에서 나오듯이 많은 계층을 통과하기 때문에 Performance에서 안좋을수 있다.
- 한번 적용되면 다른 패턴을 섞는다거나 조금 수정하는 유연함은 조금 떨어진다.
- 어플리케이션의 서비스가 커짐에 따라 유지하기 힘든 구조이다.
- 하지만, 각 계층의 역할이 명확하여 개발과 테스트가 편해진다.

Presentation layer

사용자와 가장 가까운 계층으로 API에서 엔드포인트에 해당하며 http 통신의 요청과 응답을 처리하는 계층이다. 즉 모든 시스템의 상호작용은 이 층을 통하여 이루어 진다. 모든 에러 처리는 이 계층에서 진행되며 하위 계층의 영향을 받게 된다. 웹페이지가 될 수도 있고 Restrul API가 될 수 있다.

Control layer: Controller

시스템의 모든 것을 통제하는 계층으로 생각하자. 즉 Database와의 연결을 맺고 끊는 것 또한 해당 계층에서 발생하게 된다. 하위 계층에서 Database와의 연결이 진행될 시 트랜잭션 처리가 온전히 이루어질 수 없게 된다.

Business layer: Service

모든 Business Logic이 구현되는 부분으로 Persistence 계층에 종속적이다. 즉 하위 계층으로부터 데이터를 받아서 가공하는 작업이 실행되는 계층이라 이해하자.

Objected-oriented

Mvc

모델 서브시스템: 도메인의 지식을 저장보관

뷰 서브시스템: 사용자에게 보여줌

제어 서브시스템: 사용자와의 상호 작용을 관리

분리하는 이유 : 사용자 인터페이스, 즉 뷰와 제어가 도메인 지식을 나타내는 모델보다는 더 자주 변경될 수 있기 때문

Model

애플리케이션의 상태(data)를 나타낸다.

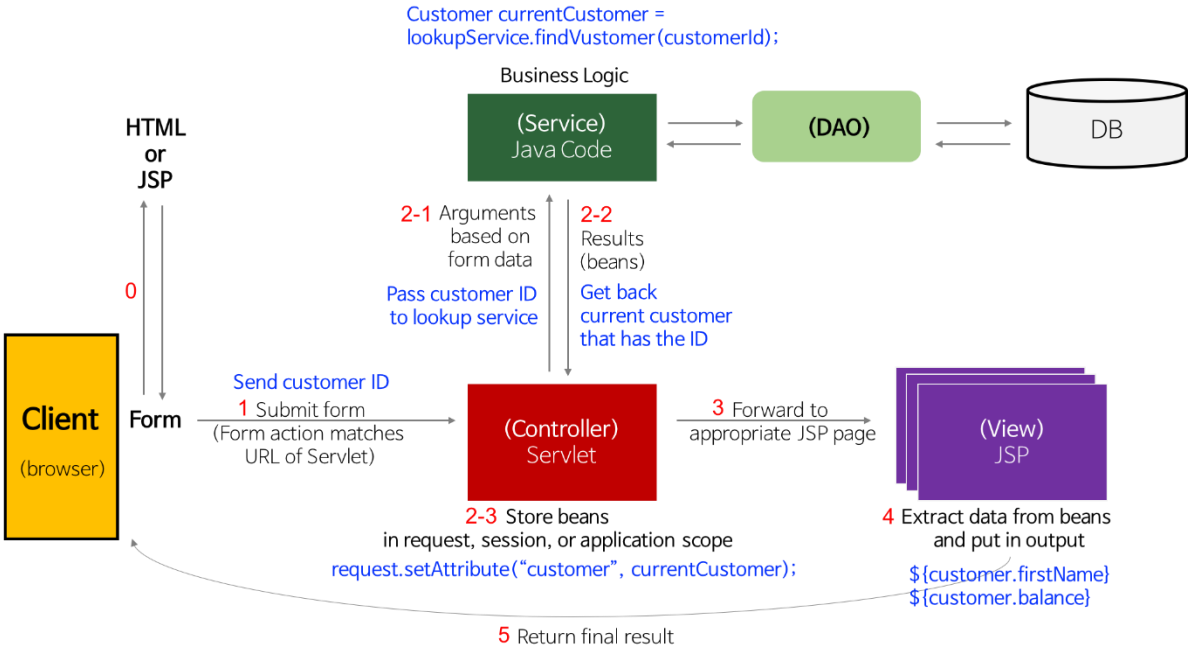
View

디스플레이 데이터 또는 프리젠테이션

Controller

View와 Model 사이의 인터페이스 역할

Model/View에 대한 사용자 입력 및 명령을 수신하여 그에 따라 적절하게 변경



Distributed

서버는 클라이언트라 불리는 서브시스템에게 서비스를 제공

클라이언트: 사용자로부터 입력을 받아 범위를 체크하고 데이터베이스 트랜잭션을 구동하여 필요한 모든 데이터를 수집

서버: 트랜잭션을 수행하고 데이터의 일관성을 보장한다

Client-Server - 다수의 클라이언트가 서버에 연결되는 구조. 사용자 인터페이스와 비즈니스 로직의 분리. 서버의 재사용이 가능. 서버 안정성에 따라 전체적 시스템 불안이 야기될 수 있다.

Multi-tiers - SoC에 따라서 여러개의 Tier 로 구성하는 구조. 비즈니스 로직, 인터페이스, 데이터베이스 등의 분리. 확장, 재사용성이 뛰어나지만 테스트가 어렵고, 네트워크 사용에 따른 비용 발생한다. **Proxy** - proxy 서버를 두어서 원격의 인스턴스를 대신하거나 보안을 강화하고, 또는 가상의 서비스나 인스턴스 역할을 하도록 할때 사용하는 구조.

Broker - broker 서버를 중간에 위치시켜서 이기종 시스템을 연결하는 구조. 변경가능성, 확장성, 재사용성이 높아진다. 오버헤드에 따른 비용, fault-tolerance 가 낮고 테스트가 어렵다.

참조 : https://blog.naver.com/wowzzin/221445510610

https://cjmyun.tripod.com/Knowledgebase/DFD.htm - DFD 그림

https://blog.msalt.net/241

https://www.tutorialspoint.com/software_architecture_design/data_flow_architecture.htm - data flow

https://www.geeksforgeeks.org/software-engineering-architectural-design/ - 전체

https://gmlwjd9405.github.io/2018/11/05/mvc-architecture.html - mvc