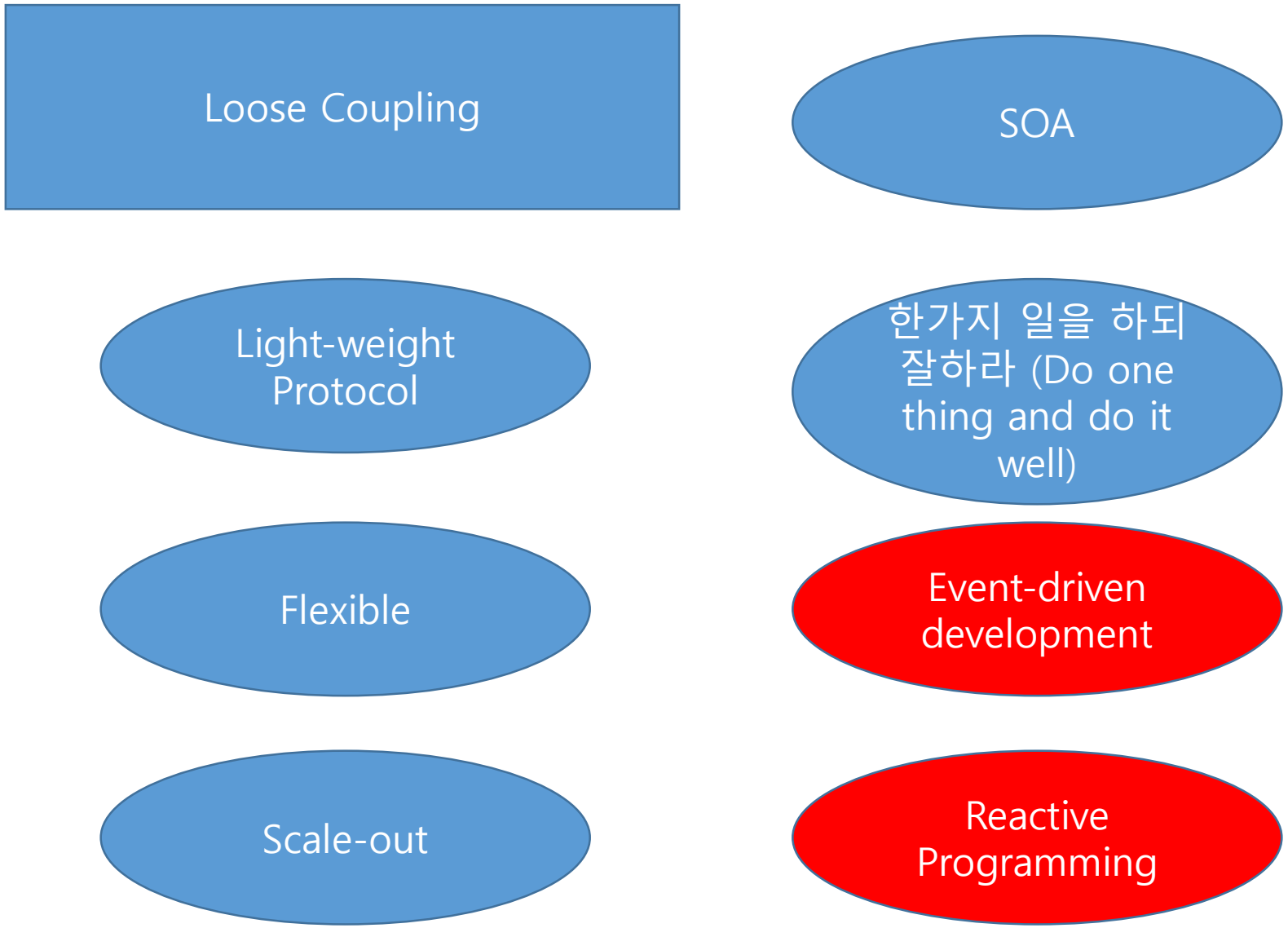
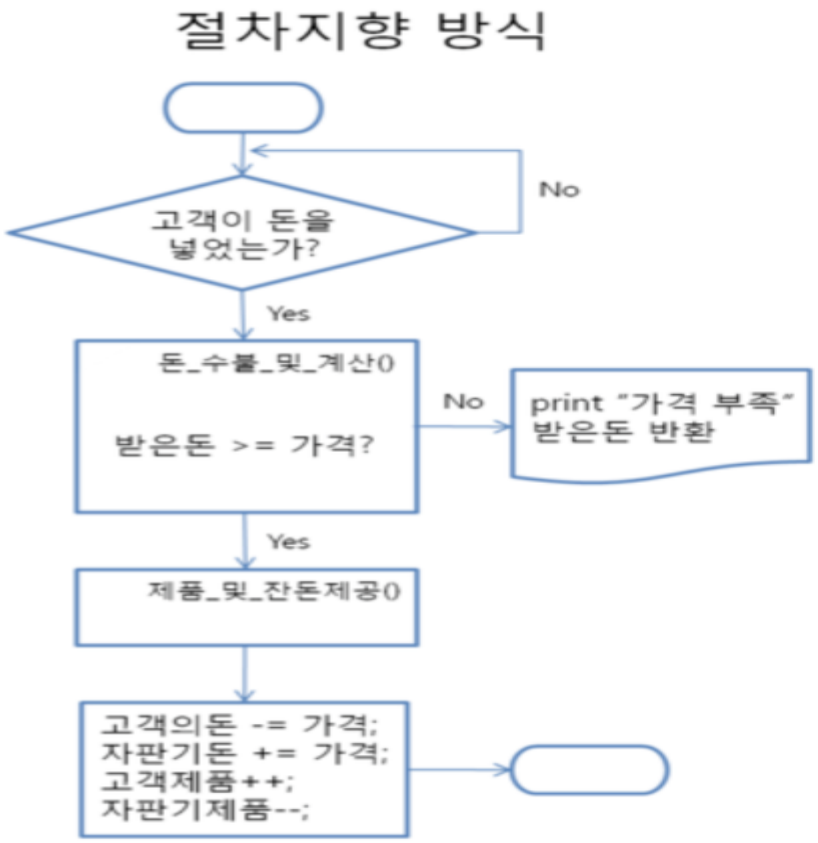
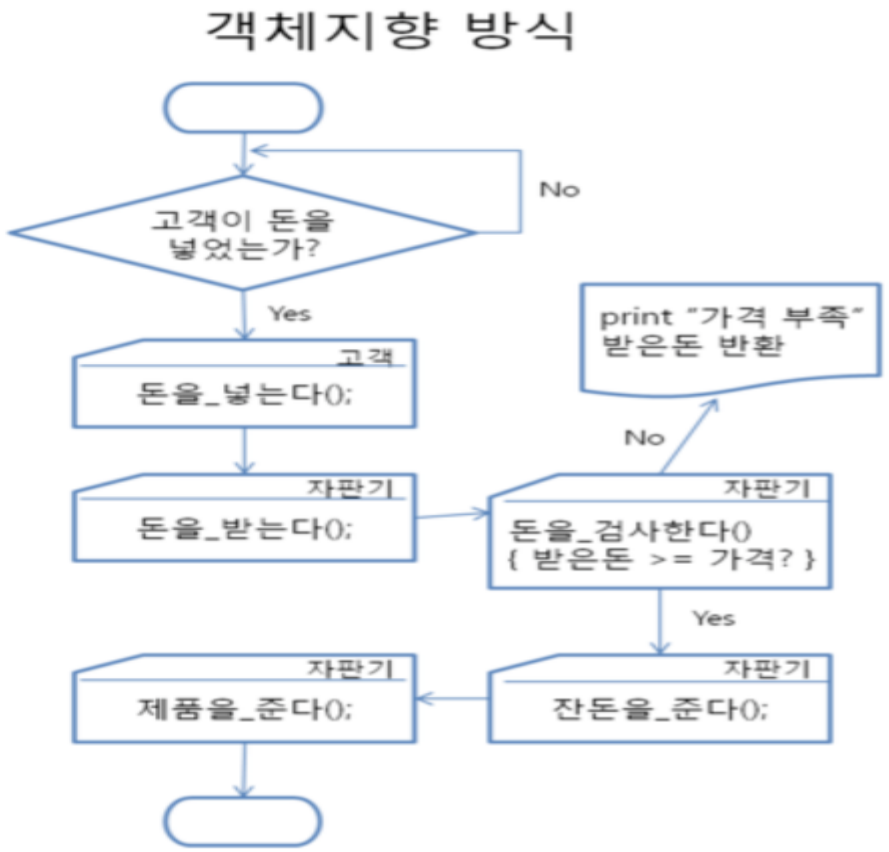

Loose Coupling
SOA
Light-weight Protocol
한가지 일을 하되 잘하라 (Do one thing and do it well)
Flexible
Event-driven development
Scale-out
Reactive Programming

1. 절차지향 vs 객체지향





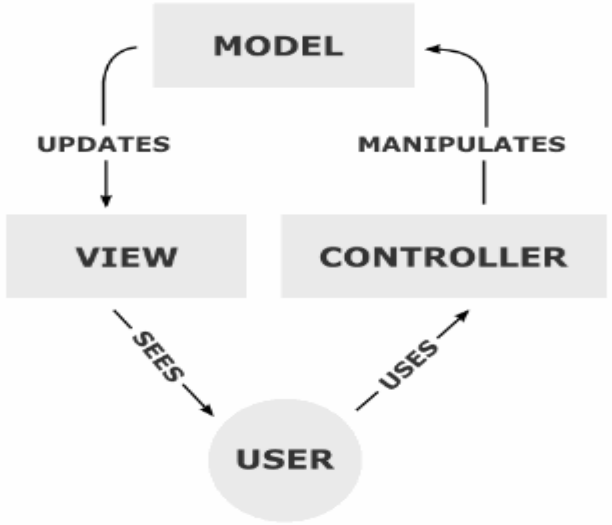
절차지향: 서비스의 생명주기(life-cycle)동안 **데이터**의 상태가 Syncronize하게 유지되어야 한다. (stateful)

객체지향: 서비스의 생명주기동안 **객체**의 상태가 Syncronize하게 유지되어야 한다. (stateful)

데이터 중심에서 객체 중심으로 변화된 Paradigm은 시스템이 보다 유연해질 수 있는 움직임이다.

2. MVC 패턴

MVC 는 Model, View, Controller의 약자 입니다. 하나의 애플리케이션, 프로젝트를 구성할 때 그 구성 요소를 세가지의 역할로 구분한 패턴입니다.



출처: 오픈듀토리얼스

객체지향 프로그래밍을 구현하는 디자인 패턴의 종류이며 1990년대 초에 표준처럼 등장하였고 가장 널리 쓰이면서, 현재도 울며 겨자먹기로 많은 국내 기업들이 사용하고 있는 디자인 패턴이다. Monolithic의 대표적인 애플리케이션 모델이며, 2000년대 초반까지도 "잘 만든 애플리케이션 1개는 10개의 애플리케이션이 부럽지 않다" 를 주장하는 사람들의 근거가 되어주는 사상이기도 하다.

Model: Business를 구성하는 동작단위. 서비스 중심으로 클라이언트의 needs를 구현하는 핵심 요소가 된다. (a.k.a. Service Layer, Data Layer)

Controller: 사용자가 서비스를 이용하기 위해 접근하는 매개체. 컨트롤러와 모델이 역할이 구분됨으로써 View와 Model의 구현 방식에 의존성을 최소화한다. (Controller가 호환할 수 있는 Language 또는 Framework가 사용되어야 한다.) (Protocol의 개념과 흡사하다)

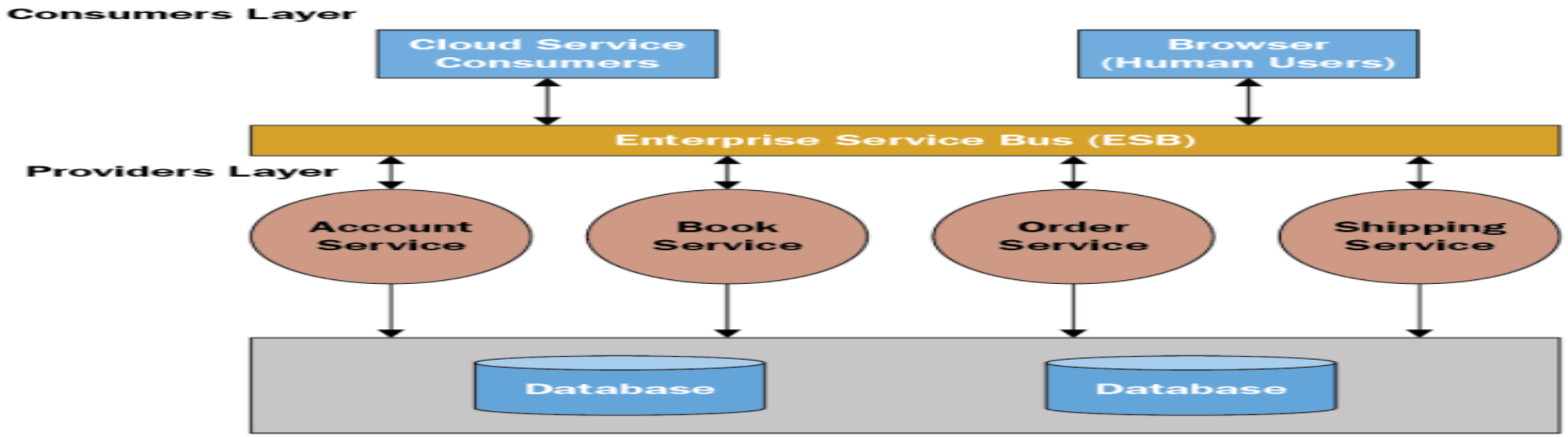
View: 사용자 편의성에 집중된 UI. Model로부터 전달받은 응답 데이터(Response)를 사용자 편의에 맞춰 다양한 형태로 Rendering할 수 있어야 한다.

3. SOA

IT 산업이 발전되면서 단일 시스템으로의 서비스는 가치가 줄어들었고, 기존에 구축된 서비스와의 연계를 위해 Web-service의 사용률이 증가하였다. 당연히 기업(Service Provider)에서는 기존에 구축된 서비스를 "신규 구축" 하기보다 "이용" 할 수 있는 방식을 선호하였고, 2004년부터 이 개념을 이용하여 Architecture Re-Building을 시도하는 기업이 늘어났다.

Service-oriented architecture (SOA)

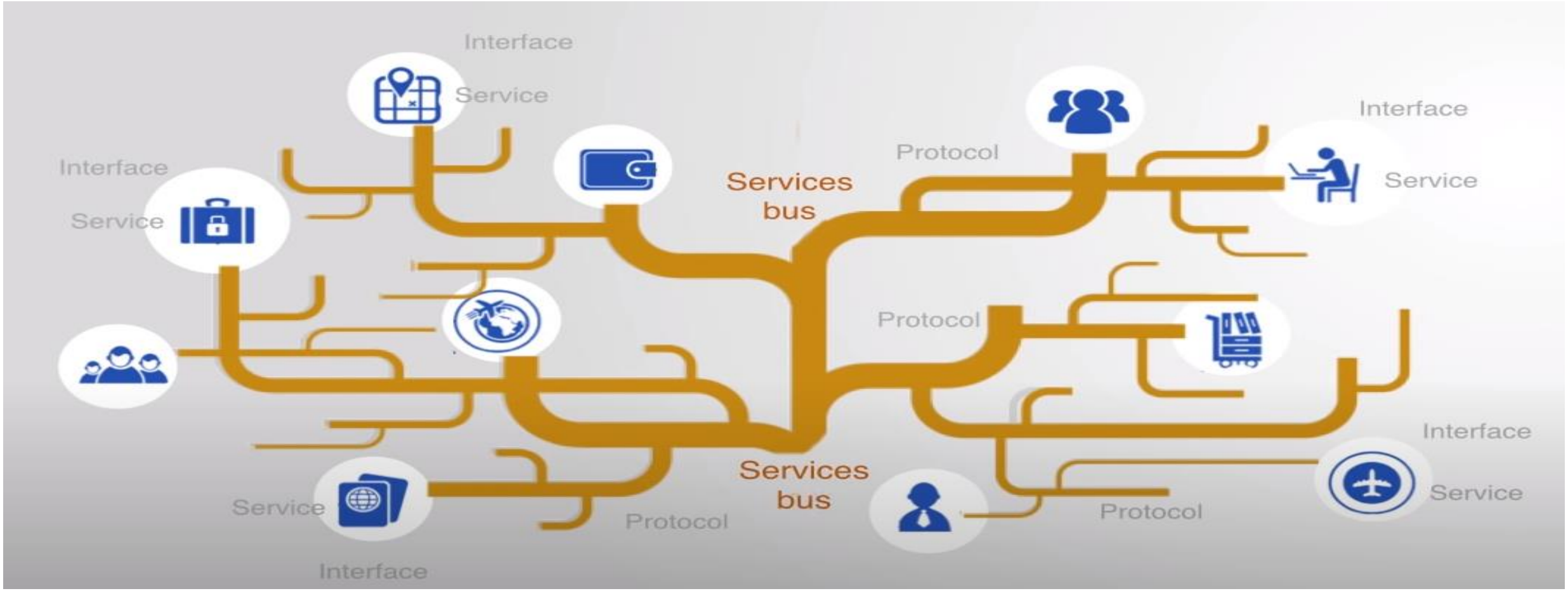
SOA is another application architectural style. In SOA, architecture services are provided to other services and to vendor components using a communication protocol over a network. These services are discrete units of functionalities that can be accessed remotely. The following diagram shows an SOA in action:



3. SOA

MSA 개발에 앞서 SOA에 대한 개념을 알아두면 도움이 될 듯 하다. 국내 서적이나 레퍼런스보다는 아래 링크의 영상이 SOA 등장배경부터 아키텍쳐 개념에 대한 요약설명이 잘 되어 있어 첨부한다. (자막이 없는 것이 단점!)

https://youtu.be/_dFJOSR-aFs



4. MSA (Microservice Architecture)

비즈니스 요구사항이 다양해지면서(특히 B2C 환경에서는 더욱 더) 시스템에 속해 있는 각각의 기능들은 요구사항에 유연하게 대응할 수 있어야 한다. 기존 Monolithic 시스템들은 side effect의 영향으로 이러한 요구사항들을 유연하게 대처할 수 없고, SOA가 대안이 될 수 있지만 SOA도 서비스간의 "계약"(contraction)이 존재하므로 coupling이 잔재한다. - *SOA(Service Oriented Architecture) 의 통신 프로토콜은 SOAP (Simple Object Access Protocol) 기반이다. 이를 보완하기 위해 REST API가 등장하였고, 효과는 굉장했다!* - 이 외에 서비스의 확장성도 중요한 이슈가 되었는데, MSA는 이 부분도 Monolithic보다 효율적으로 구성할 수 있다.

MSA의 예시를 표현한 구조도

4. MSA (Microservice Architecture)

유연성(Flexible), 느슨한 결합(Loose Coupling), 확장성(Scale-out) 등의 압도적인 장점이 있지만, MSA를 도입하였을 때의 단점도 존재한다. 아래 왼쪽 표는 장단점 중 일부 정리가 된 부분에 대한 내용이며, 현재의 MSA 모델에서는 이 단점도 cover할 수 있는 수단이 나타났는데 항목별로 기술하였다. 오른쪽 아래의 그림은 초기 MSA 모델에서 Message Queue가 추가된 모델을 간략히 표현한 그림이다.

구분		장점	단점
Monolithic		개발 환경이 동일해서 복잡하지 않다.	분산처리가 어렵다.
Monolithic		소수의 인원에서 빠르게 개발하기에 적합하다.	프로젝트가 커질수록 코드가 길어지고 배포시간이 길어진다.
Monolithic		테스트 환경구성이 쉽고 용이하다.	기능별로 효율적인 데이터베이스 선택및 개발언어 선택이 불가능하다.
MSA	Big data Platform이 필요한 이유(e.g. ELK Stack)	각 기능별로 모듈이 독립적이기 때문에 해당 모듈에 가장 효율적인 프로그래밍 언어 선택이 가능하다.	작은 여러 서비스로 분리되어 있기때문에 관리및 모니터링이 어렵다.
MSA	Reactive, MOM이 필요한 이유	각 부서별로 기능별로 해야 할일을 나눠서 효율적으로 처리할 수 있다.	서로간의 통신에러가 잦을수 있다.
MSA	가상화 배포가 필요한 이유 (Docker, Kubernetes)	중앙 집중적인 데이터 베이스가 없고 각 서비스에 알맞은 데이터 베이스를 선택할수 있다.	테스트가 불편하다. End to End 테스트를 위해서 UI, API Gateway등등 여러개의 서비스를 구동 시켜야 한다.
MSA	Circuit Breaker가 필요한 이유	새로 추가되거나 수정사항이 있는 마이크로서비스만 빠르게 빌드, 배포가 가능하다.	서비스간의 장애가 발생하는 경우 장애 추적이 어렵다.
MSA	IMDG, MOM이 필요한 이유	일부분의 오류가 있으면 해당 기능에만 오류가 발생하고 그 부분만 빠르게 고쳐서 정상화가 가능하다.	각 마이크로서비스들은 독립된 데이터 저장소를 가지기 때문에 마이크로서비스들끼리 자원을 공유하고 그 공유 자원에 접근하는 것을 구현하기 어렵고, 제약 조건도 있다.