5부.아키텍처

15장. 아키텍처란?

소프트웨어 아키텍트 == 프로그래머

소프트웨어 아키텍터는 코드와 동떨어져서는 안됨.

프로그래밍을 계속하면서도 동시에 다른 팀원들이 생산성을 극대화할 수 있는 설계를 하도록 방향을 이끌어줘야함.

소프트웨어 시스템의 아키텍처란 시스템을 구축했던 사람들이 만들어낸 시스템의 형태다. 그 모양은 시스템을 컴포넌트로 분할하는 방법, 분할된 컴포넌트를 배치하는 방법, 컴포넌트가 서로 의사소통하는 방식에 따라 정해진다.

> 이러한 일을 용이하게 만들기 위해서는 가능한 한 많은 선택지를, 가능한 한 오래 남겨두는 전략을 따라야 한다.

시스템 아키텍처의 최우선 목표는 시스템의 정상 동작이다. 아키텍처의 주된 목적은 시스템의 생명 주기를 지원하는 것이다. 좋은 아키텍처는 시스템을 쉽게 이해하고, 쉽게 개발하며, 쉽게 유지복수하고, 또 쉽게 배포하게 해준다. 아키텍처의 궁극적인 목표는 시스템의 수명과 관련된 비용은 최소화하고, 프로그래머의 생산성은 최대화하는데 있다.

개발

개발하기 힘든 시스템이라면 수명이 길지도 않고 건강하지도 않을 것이다. 따라서 시스템 아키텍처는 개발팀이 시스템을 쉽게 개발할 수 있도록 뒷받침해야 한다.

좋은 아키텍처는 서비스가 클 때 그 효과가 발한다.

배포

소프트웨어 시스템이 사용될 수 있으려면 반드시 배포할 수 있어야 한다. 배포 비용이 높을수록 시스템의 유용성은 떨어진다. 따라서 소프트웨어 아키텍처는 시스템을 단 한 번에 쉽게 배포할 수 있도록 만드는데 그 목표를 두어야 한다.

안타깝지만 초기 개발 단계에서는 배포 전략을 거의 고려하지 않는다. 이로 인해 개발하기는 쉬울지 몰라도 배포하기는 상당히 어려운 아키텍처가 만들어진다.

운영

아키텍처가 시스템 운영에 미치는 영향은 개발, 배포, 유지보수에 미치는 영향보다는 덜 극적이다 . 운영에서 겪는 대다수의 어려움은 소프트웨어 아키텍처에서 극적인 영향을 주지 않고도 단순히 하드웨어를 더 투입해서 해결할 수 있다.

유지보수

유지보수는 모든 측면에서 봤을 때 소프트웨어 시스템에서 비용이 가장 많이 든다. 새로운 기능은 끝도 없이 행진하듯 발생하고, 뒤따라서 발생하는 결함은 피할 수 없으며, 결함을 수정하는 데도 엄청난 인적 자원이 소모된다.

선택사항 열어두기

소프트웨어를 부드럽게 유지하는 방법은 선택사항을 가능한 많이, 그리고 가능한 한 오랫동안 열어 두는 것이다. 그렇다면 열어 둬야 할 선택사항이란 무엇일까? 그것은 바로 중요치 않는 세부사항이다. 소프트웨어 시스템은 주요한 두 가지 구성요소로 분해할 수 있다. 바로 정책과 세부 사항이다. 정책 요소는 모든 업무 규칙과 업무 절차를 구체화한다. 정책이란 시스템의 진정한 가치가 살아있는 곳이다. 세부사항은 사람, 외부 시스템, 프로그래머가 정책과 소통할 때 필요한 요소지만, 정책이 가진 행위에는 조금도 영향을 미치지 않는다. 이러한 세부사항에는 입출력 장치, 데이터베이스, 웹 시스템, 서버, 프레임워크, 통신 프로토콜이 있다.

아키텍트의 목표는 시스템에서 정책을 가장 핵심적인 요소로 식별하고, 동시에 세부사항은 정책에 무관하게 만들 수 있는 형태의 시스템을 구축하는데 있다.

장치 독립성

인프라 요소의 세부사항에 대해서 열어둠.

어떤 장치를 사용할지 전혀 모른채 그리고 고려하지 않고도 프로그램을 작성할 수 있었다.

좋은 아키텍트란?

세부사항을 정책으로부터 신중하게 가려내고, 정책이 세부사항과 결합되지 않도록 엄격하게 격리한다.

16장. 독립

유스케이스

유스케이스란 시스템의 아키텍처가 시스템 의도를 반드시 지원 해야 하는 것을 의미한다. 예를 들어, 쇼핑 카트 앱이면 쇼핑 카트 유스케이스를 지원 해야 한다. 유스케이스는 아키텍처의 최초에 관심사이자, 최 우선 순위이다.

그러나 아키텍처는 시스템의 동작에 많은 영향을 주지는 못한다. 그러나 영향력이 전부는 아니며, 가장 중요한 일은 시스템의 의도가 아키텍처 수준에서 볼 수 있도록 동작을 명확히 하고 노출하는 것이다.

개발자는 동작이 시스템의 최상위 수준에서 볼 수 있는 일급 요소가 될 것이기 때문에 동작을 찾을 필요가 없고, 아키텍처 내에서 눈에 띄는 위치에있는 클래스 또는 기능 또는 모듈이 될 것이며 기능을 명확히 설명하는 이름을 갖게 된다.

운영

아키텍처는 핵심 비즈니스 로직이 아닌 부분에 대해서는 세부사항에 대해 열려 있어야 한다. 만약 세부사항이 열려 있지 않고 특정 방식을 우선해서 선택하고 아키텍처를 구조화하면 변경의 유연성을 잃게 된다.

예를 들어, 모놀리스, 단일 프로세스, 단일 쓰레드를 우선으로 고려하여 시스템을 구축하면 향후에 마이크로 서비스, 멀티 프로세스, 멀티 쓰레드로 전환해야하는 상황이 발생했을 때 변경이 어려워 진다.

그렇기 때문에 구성요소의 적절한 격리를 유지하며 유스케이스와 비즈니스 로직에 우선으로 집중하고 세부사항에 대해서는 나중에 선택 해야 한다.

개발

아키텍처는 개발 환경을 지원하는데 중요한 역할을 한다. 여기서 Conway의 법칙이 작용한다.

> 시스템을 설계하는 모든 조직은 그 구조가 조직의 커뮤니케이션구조의 복사본인 설계를 생성합니다.

많은 팀과 많은 관심이 있는 조직에서 개발해야 하는 시스템에는 개발 중에 팀이 서로 간섭하지 않도록 해당 팀의 독립적인 작업을 용이하게 하는 아키텍처가 있어야 합니다. 이는 시스템을 잘 격리되고 독립적으로 개발 가능한 구성 요소로 적절하게 분할하여 수행됩니다. 그런 다음 이러한 구성 요소를 서로 독립적으로 작업할 수 있는 팀에 할당할 수 있습니다.

배포

아키텍처는 또한 시스템 배포의 용이성을 결정하는데 큰 역할을 합니다. 목표는 즉각적인 배포 입니다. 좋은 아키텍처는 수십 개의 작은 구성 스크립트와 속성 파일 조정에 의존하지 않습니다. 좋은 아키텍처는 빌드 후 시스템을 즉시 배포할 수 있도록 도와줍니다.

이러한 즉각적인 배포는 각 구성요소를 적절히 분할, 격리 하면서도 통합도 되도록 함으로써 달성됩니다.

선택사항 열어두기

좋은 아키텍처는 이러한 모든 문제를 상호 만족시키는 구성 요소 아키텍처와 함께 균형을 유지 합니다. 근데 이것은 쉽지 않습니다.

문제는 대부분의 경우 모든 사용 사례가 무엇인지 모르고 운영 제약, 팀 구조, 또는 배포 요구 사항을 모른다는 것입니다. 더 나쁜 것은 우리가 그것들을 알 고 있다 하러다로 시스템 라이프 사이클을 통해 결과적으로 변경될 것이라는 점입니다. 요컨대 우리가 달성해야 하는 목표는 불분명하고 일정하지 않습니다.

이러한 상황을 방지하기 위해서라도 시스템을 잘 격리된 구성 요소로 분할하고 가능하면 많은 선택사항을 여어두어야 합니다. 선택 사항을 열어둠으로써 시스템이 변경되어야 하는 모든 방식으로 쉽게 변경될 수 있게 합니다.

레이어 분리

설계자는 시스템의 구조가 모든 필요한 유스케이스를 지원하기를 원한다. 그러나 모든 유스케이스를 알기는 어렵다. 그러나 설계자는 시스템의 기본적인 의도는 알고 있다. 그래서 설계자는 단일 책임 원칙과 공통 폐쇄 원칙을 통해서 다른 이류로 변경되는 것을 분리하고 같은 이유로 변경되는 것을 모아줘야 한다. 이는 시스템의 의도에 대한 맥락을 주기 위함이다.

다른 이유로 변경되는 것이란 무엇일까? 대표적으로 UI는 비즈니스 룰과 관련이 없는 이유로 변경됩니다. 반대로 유스케이스에는 비즈니스 룰과 관련된 요소가 모두 있습니다. 훌륭한 설계자는 유스케이스의 UI부분을 비즈니스 룰과 분리하여 좀 더 유스케이스가 직관적이고 명확하게 유지 하면서도 독립적으로 변경되기를 원할 것이다.

비즈니스 규칙은 그 종류를 구분해야 하며, 입력 필드의 유효성 검사와 같은 클라이언트 레벨에서의 비즈니스 규칙과 계정에 대한 이자 계산 같은 도메인과 연결된 비즈니스 규칙이 있으며 이 둘은 분리되어야 한다.

데이터베이스, SQL, 스키마 등도 비즈니스 룰과 관련이 없는 기술적인 세부사항으로 독립적으로 변경될 수 있어야 한다.

따라서 우리는 시스템이 분리된 레이어 아키텍처로 나누어져 있음을 알 수 있습니다.

(UI, 애플리케이션 별 비즈니스 룰, 애플리케이션 비즈니스 룰, 데이터베이스 등)

결합 분리 모드

UI와 데이터베이스 등의 세부사항이 비즈니스 룰과 분리된 경우 서로 다른 서버에서 실행할 수 있습니다. 많은 리소스를 요구하는 서버는 복제되는 것도 가능합니다. 분리된 구성요소는 네트워크를 통해 통신하는 독립적인 서비스(마이크로서비스)여야 합니다.

독립적인 개발 능력

구성 요소가 강력하게 분리되면 팀 간의 간섭이 완화됩니다. 비즈니스 룰이 UI에 의존하지 않는다면 UI에 중점을 둔 팀은 비즈니스 규칙에 중점을 둔 팀에 큰 영향을 미칠 수 없습니다.

독립적인 배포

유스케이스와 레이어의 분리는 배포 시 높은 수준의 유연성을 제공합니다. 실제로 분리가 잘 수행되면 실행 중인 시스템에서 레이어와 유스케이스를 핫스왑할 수 있어야 한다. 유스케이스의 변경은 JAR파일이나 서비스를 추가 혹은 변경 삭제하는 식과 같아야 합니다.

중복

중복은 일반적으로 소프트웨어에서 나쁜 것 입니다. 중복에는 여러 종류가 있습니다. 한 인스턴스를 복제한 인스턴스에 대해 동일한 변경을 필요로 하는 진정한 복제가 있습니다. 이와 달리 허위 복제는 서로 다른 이유로 변경되는 중복된 코드가 있을 경우 이는 나쁜 중복이 아닌 다른 것으로 보아야 합니다.

결합 분리 모드(다시)

레이어와 유스케이스를 분리하는 방법에는 여러가지가 있습니다. 소스 코드 수준, 바니어리 코드 수준, 실행 단위 수준에서 분리할 수 있습니다.

소스 레벨

소스 코드 모듈 간의 종속성을 제어하여 한 모듈에 대한 변경이 다른 모듈의 변경 또는 재컴파일을 강제하지 않도록 할 수 있습니다.

이 역결합 모드에서 구성 요소는 모두 동일한 주소 공간에서 실행되고 간단한 합수 호출을 사용하여 서로 통신합니다. 사람들은 종종 모놀리식 구조라고 합니다.

배포 수준

jar나 ddl 또는 공유 라이브러리와 같은 배포 가능한 단위 간의 종속성을제어할 수 있으므로 한 모듈의 소스 코드를 변경해도 다른 모듈을 다시 빌드하고 다시 배포하지 않습니다.

많은 구성 요소가 여전히 동일한 주소 공간에 있고 함수 호출을 통해 통신할 수 있습니다. 다른 구성 요소는 동일한 프로세서의 다른 프로세스에 있을 수 있으며 프로세스 간 통신, 소켓 또는 공유 메모리를 통해 통신할 수 있습니다.

서비스 수준

우리는 종속성을 데이터 구조 수준까지 줄이고 모든 실행 단위가 소스 및 다른 항목(예: 서비스 또는 마이크로 서비스)에 대한 변경과 완전히 독립적이도록 네트워크 패킷을 통해서만 통

신할 수 있습니다.

좋은 아키텍처는 시스템이 단일 파일로 배포되는 모놀리스로 태어날 수 있도록 허용하지만 독립적으로 배포 가능한 단위 세트로 성장한 다음 독립 서비스 및/또는 마이크로 서비스까지 확장할 수 있습니다. 나중에 상황이 바뀌면 진행 상황을 역전시키고 모놀리스로 완전히 다시 돌아가는 것을 허용해야 합니다.

17장. 경계: 선 그리기

소프트웨어 아키텍처는 경계라는 이름의 선을 그리스는 기술입니다. 이러한 경계는 소프트웨어 요소를 서로 분리하고 한 쪽에 있는 요소가 다른 쪽의 요소에 대해 알지 못하도록 제한합니다. 이러한 라인 중 일부는 코드가 작성되기도 전에도 프로젝트 수명 초기에 그려집니다.

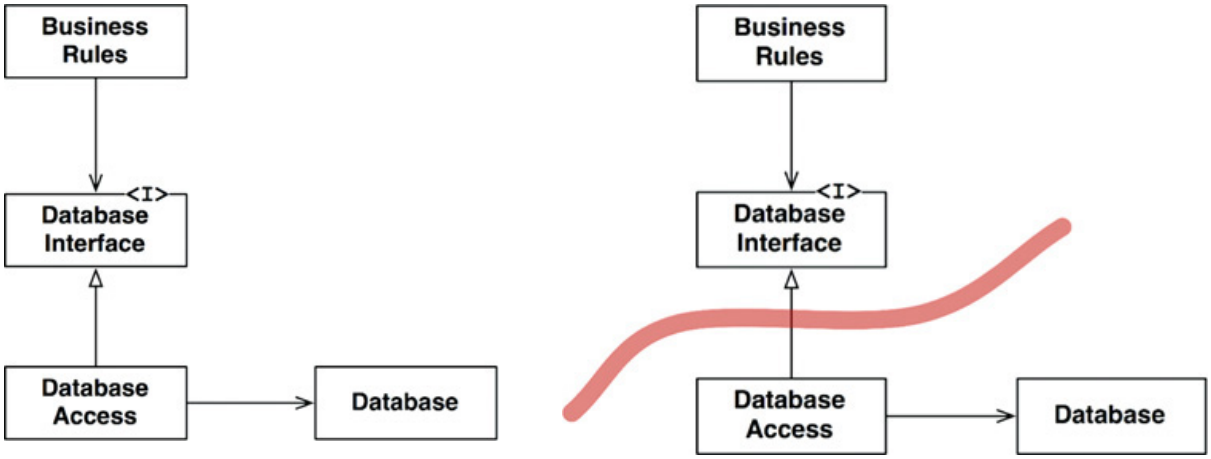
아키텍처의 목표는 필요한 시스템을 구축하고 유지하는데 필요한 인적 자원을 최소화하는 것이며 결합도는 이를 약화시킵니다.

시기상조를 주의 해야 하는 주제로 비즈니스 유스케이스와 무관한 프레임워크, 데이터베이스, 웹서버에 대한 결정이 포함됩니다. 좋은 아키텍처는 이러한 것들에 의존하지 않으며, 가장 마지막에 결정되어야 합니다.

여러 사례를 저자는 소개를 하는데 비즈니스 규칙과 UI를 분리하고, 세부사항에 대한 결정을 후순위를 두고 진행한 아키텍처 덕분에 요구사항 변화, 인프라 환경 변화에 유연하게 대응할 수 있었다는 내용입니다.

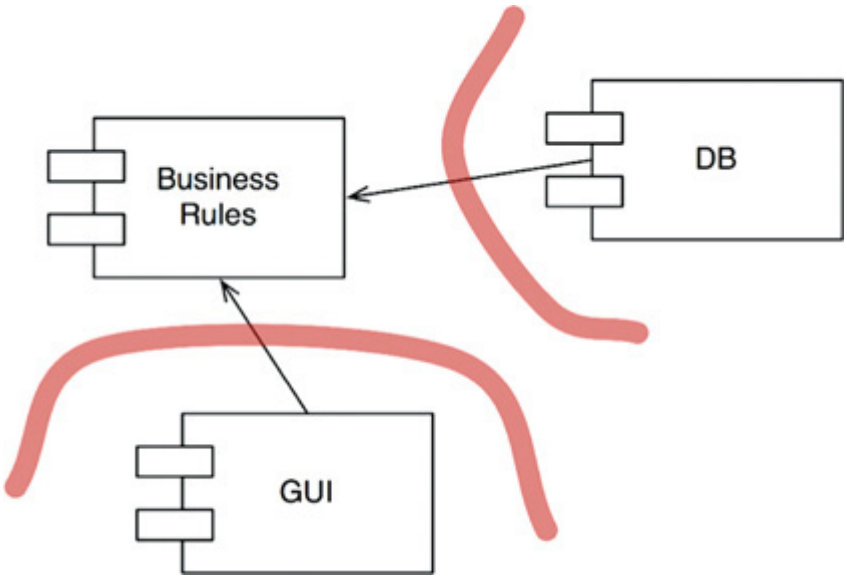
선은 어떻게 그어야 하고 언제 그릴 것인가?

중요한 것과 중요하지 않은 것 사이에 선 긋습니다. 중요한 것에는 비즈니스 규칙이 있고 중요하지 않은 것에는 UI, Database가 있습니다.



데이터베이스는 비즈니스 규칙이 간접적으로 사용할 수 있는 도구 일 뿐이며, 필요시 변경될 수 있어야하고 이러한 변경은 중요한 비즈니스 규칙에 영향을 주어서는 안됩니다. 그래서 비즈니스 규칙이 데이터베이스에 대해 직접적으로 의존하지 않도록 중간에 인터페이스를 두어 의존하도록 함으로써 데이터베이스의 기능을 사용하면서도 의존 방향을 역전시킬 수 있습니다.

결과적으로 의존성의 방향은 아래와 같이 됩니다.



플러그인 아키텍처

종합하면 데이터베이스와 GUI에 대한 이 두 가지 결정은 다른 구성 요소를 추가하기 위한 일종의 패턴을 만듭니다. 이 패턴은 타사 플러그인을 허용하는 시스템에서 사용하는 것과 동일한 패턴입니다.

실제로 소프트웨어 개발 기술의 역사는 확장 가능하고 유지 관리 가능한 시스템 아키텍처를 구축하기 위해 플러그인을 편리하게 만드는 방법에 대한 이야기입니다. 핵심 비즈니스 규칙은 선택 사항이거나 다양한 형태로 구현될 수 있는 구성 요소와 분리되어 독립적으로 유지됩니다.

GUI는 웹 기반, C/S 기반, 콘솔 기반으로 대체 가능하며, DB는 RDB, NoSQL 로 대체 될 수 있습니다.

결론

소프트웨어 아키텍처에서 경계선을 그리려면 먼저 시스템을 구성 요소로 분할합니다. 이러한 구성 요소 중 일부는 핵심 비즈니스 규칙입니다. 나머지는 핵심 비즈니스와 직접적인 관련이 없는 필수 기능을 포함하는 플러그인입니다. 그런 다음 구성 요소 사이의 화살표가 핵심 비즈니스를 가리키는 한 방향을 가리키도록 해당 구성 요소의 코드를 정렬합니다.

이것은 종속성 반전 원칙과 안정적인 추상화 원칙의 적용으로 인식해야 합니다. 종속성 화살표는 하위 수준 세부 정보에서 상위 수준 추상화를 가리키도록 정렬됩니다.

18장. 경계 해부학

시스템의 아키텍처는 소프트웨어 구성 요소 집합과 이를 구분하는 경계에 의해 정의됩니다. 이러한 경계는 다양한 형태로 나타납니다. 이 장에서는 가장 일반적인 몇 가지를 살펴보겠습니다.

경계 횡단하기

런타임 시 경계 교차는 경계의 한족에서 함수를 호출하고 다른 쪽에서 일부 데이터를 전달하는 함수일 뿐입니다. 적절한 경계 교차를 만드는 비결은 소스 코드 종속성을 관리하는 것 입니다.

왜 소스 코드인가? 하나의 소스 코드 모듈이 변경되면 다른 소스 코드 모듈을 변경하거나 다시 컴파일한 다음 다시 배포해야 할 수 있기 때문입니다.

두려운 모놀리스

가장 단순하고 흔한 경계는 물리적으로 엄격하게 구분되지 않는 형태이다. 단순히 단일 프로세서와 단일 주소 공간 내에서 기능과 데이터의 규칙적인 분리입니다.

배포의 관점에서 이것은 단일 실행 파일, 즉 모놀리스에 불과합니다.

모놀리스를 배포하는 동안 경계까 표시되지 않는다는 사실이 경계가 존재하지 않고 의미가 없다는 의미는 아닙니다. 단일 실행 파일로 정적으로 링크된 경우에도 최종 조립을 위해 다양한 구성 요소로 독립적으로 개발하고 마샬링하는 기능은 매우 중요합니다.

> 호출에 필요한 정보를 모아서 정리하는 것

가장 단순한 형태의 경계 횡단은 저수준 클라이언트에서 고수준 서비스로 향하는 함수 호출입니다.런타임 의존성과 컴파일타임 의존성은 모두 같은방향이 되는데, 저수준 컴포넌트에서 고수준 컴포넌트로 향하게 되는 샘이다. 만약 고수준 클라이언트에서 저수준 서비스를 호출해야 한다면 의존성 역전을 이용하면 된다.

배포형 컴포넌트

멀티 모듈을 설명합니다.

아키텍처 경계선의 가장 단순한 물리적 표현은 동적 링크 라이브러리이다(예. DLL, jar, UNIX 공유 라이브러리 등). 배포는 컴파일과 관련이 없지만 구성 요소가 바이너리 또는 이와 동등한 배포 가능한 형태로 제공됩니다.이것이 배포 레벨 역결합 모드 이다. 배포 작업은 단순히 이러한 배포 가능한 유닛을 WAR 파일 또는 디렉토리와 같은 편리한 형식으로 함께 모으는 것이 된다.

한 가지 예외를 제외하고 배포수준 구성 요소는 모놀리스와 동일한데, 기능은 일반적으로 동일한 메모리와 프로세서에 존재한다. 배포 구성 요소 경계를 넘는 통신은 함수 호출에 불과하므로 매우 저렴하다.

스레드

모놀리스와 배포형 컴포넌트는 모두 스레드를 활용할 수 있다. 모든 스레드가 단 하나의 컴포넌트에 포함될 수 도 있고, 많은 컴포넌트에 걸쳐 분산될 수도 있다.

로컬 프로세스

훨씬 더 강력한 물리적 아키텍처 경계는 로컬 프로세스입니다. 로컬 프로세스는 일반적으로 ㅁ여령줄 또는 이에 상응하는 시스템 호출에서 생성됩니다. 로컬 프로세스는 동일한 프로세서 또는 멀티 코어 내의 동일한 프로세서 세트에서 실행되지만 별도의 주소 공간에서 실행됩니다. 공유 메모리 파티션을 사용되지만 메모리 보호는 일반적으로 이러한 프로세스가 메모리를 공유하는 것을 방지합니다.

프로세스간 통신은 소켓이나 메시지 큐 등을 통해 이뤄집니다.

서비스

가장 강력한 경계는 서비스 입니다. 서비스는 일반적으로 명령행 또는 동등한 시스템 호출을 통해 시작되는 프로세스 입니다. 서비스는 물리적인 위치에 의존하지 않습니다, 두 개의 통신 서비스는 동일한 물리적 프로세서 또는 멀티코어에서 동작하거나 동작하지 않을 수 있습니다.

서비스는 모든 통신이 네트워크를 통해 발생하며 모놀리스와 같이 함수 호출로 이뤄지는 통신과 비교 했을 때 느린 편 입니다. 그렇기 때문에 빈번히 통신하는 일이 없어야 하며, 발생할 수 있는 지연은 고수준에서 처리할 수 있어야 합니다.

저수준 서비스는 반드시 고수준 서비스의 플러그인이 되어야 한다.

결론

단일체를 제외한 대다수의 시스템은 한 가지 이상의 경계 전략을 사용한다. 서비스 경계를 활용하는 시스템이라면 로컬 프로세스 경계도 일부 포함하고 있을 수도 있다.

19장. 정책과 수준

소프트웨어 시스템이란 정책을 기술한 것이다.

컴퓨터 프로그램은 각 입력을 출력으로 변환하는 정책을 상세하게 기술한 설명서다.

대다수의 주요 시스템에서 하나의 정책은 이 정책을 서술하는 여러 개의 조그만 정책들로 쪼갤 수 있다.

의존성은 소스 코드, 컴파일 타임의 의존성이다.

- 자바 → `import` 구문
- C# → `using` 구문
- 루비 → `require` 구문

좋은 아키텍처라면 각 컴포넌트를 연결할 때 의존성의 방향이 컴포넌트의 수준을 기반으로 연결되도록 만들어야 한다. 즉, 저수준 컴포넌트가 고수준 컴포넌트에 의존하도록 설계되어야 한다.

수준 Level

수준은 엄밀하게 정의하자면 입력과 출력까지의 거리이다. 시스템의 입력과 출력 모두로부터 멀리 위치할 수록 정책의 수준은 높아진다. 입력과 출력을 다루는 정책이라면 시스템에서 최하위 수준에 위치한다.

소스 코드 의존성은 그 수준에 따라 결합되어야 하며, 데이터 흐름을 기준으로 결합되서는 안 된다. 데이터 흐름과 소스 코드 의존성이 항상 같은 방향을 가리키진 않는다. 아래는 잘못된 프로그램의 예시이다.

```
function encrypt() {
  while(true)
    writeChar(translate(readChar()));
}
```

고수준에 해당하는 번역 함수가 데이터를 읽어오는 함수와 쓰는 함수에 직접적으로 의존하고 있으므로 잘못된 아키텍처 이다.

결론

정책에 대한 논의는 단일 책임 원칙, 개방 폐쇄 원칙, 공통 폐쇄 원칙, 의존성 역전 원칙, 안정된 의존성 원칙, 안정된 추상화 원칙을 모두 포함한다.

이 원칙들의 설명을 다시 읽어 보며 각 원칙이 어디에서 무슨 이유로 사용되었는지를 잘 숙지하자.

20장. 업무 규칙 Business Rule

업무 규칙은 사업적으로 수익을 얻거나 비용을 줄일 수 있는 규칙 또는 절차다. 업무 규칙은 통상적으로 데이터를 요구하는데 이를 핵심 업무 데이터라고 부르겠다.

핵심 규칙과 핵심 데이터는 본질적으로 결합되어 있기 때문에 본질적으로 객체로 만들 좋은 후보가 된다. 우리는 이러한 유형의 객체를 엔티티라고 한다.

엔티티

엔티티는 컴퓨터 시스템 내부의 객체로서, 핵심 업무 데이터를 기반으로 동작하는 일련의 조그만 핵심 업무 규칙 구체화한다. 엔티티 객체는 핵심 업무 데이터를 직접 포함하거나 핵심 업무 데이터에 매우 쉽게 접근할 수 있다. 엔티티의 인터페이스는 핵심 업무 데이터를 기반으로 동작하는 핵심 업무 규칙을 구현한 함수로 구성된다.

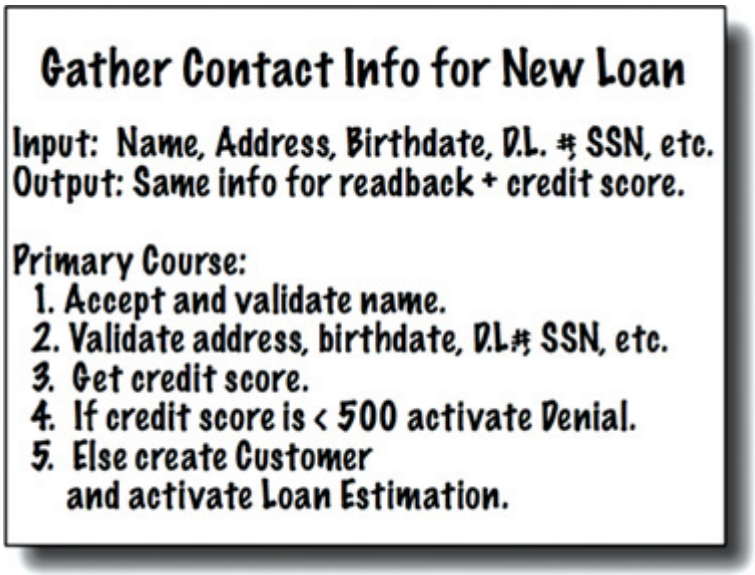
이 클래스는 데이터베이스, 사용자 인터페이스, 서드 파티 프레임워크에 대한 고려사항들로 인해 오염되어서는 절대 안 된다. 엔티티는 순전히 업무에 대한 것이며, 이외의 것은 없다.

유스케이스

유스케이스는 자동화된 시스템이 사용되는 방법을 설명한다. 유스케이스는 사용자가 제공해야할 입력, 사용자에게 보여줄 출력, 그리고 해당 출력을 생성하기 위한 처리 단계를 기술한

다. 엔티티 내의 핵심 업무 규칙과는 반대로, 유스케이스는 애플리케이션에 특화된 업무 규칙을 설명한다.

유스케이스는 엔티티 내부의 핵심 업무 규칙을 어떻게, 그리고 언제 호출할지를 명시하는 규칙을 담는다. 엔티티가 어떻게 춤을 출지는 유스케이스가 제어하는 것이다. 유스케이스만 봐서는 이 애플리케이션이 웹을 통해 전달되는지, 리치 클라이언트인지, 콘솔기반인지. 아니면 순수한 서비스인지를 구분하기란 불가능하다



유스케이스는 시스템이 사용자에게 어떻게 보이는지를 설명하지는 않는다. 이보다는 애플리케이션에 특화된 규칙을 설명하며, 이를 통해 사용자와 엔티티 사이의 상호작용을 규정한다.

엔티티는 자신을 제어하는 유스케이스에 대해 아무것도 알지 못한다. 엔티티와 같은 고수준 개념은 유스케이스와 같은 저수준 개념에 대해 아무것도 알지 못한다, 반대로 저수준인 유스케이스는 고수준인 엔티이에 대해 알고 있다. 왜냐하면 유스케이스는 단일 애플리케이션에 특화되어 있으며, 따라서 해당 시스템의 입력과 출력에 보다 가깝게 위치하기 때문이다.

요청과 응답 모델

우리는 유스케이스 클래스의 코드가 HTML이나 SQL에 대해 알게 되는 일을 절대로 원치 않는다.

엔티티 객체를 가리키는 참조를 요청과 응답 데이터 구조에 포함하려는 유혹을 받을 수도 있다. 엔티티와 요청/응답 모델은 상당히 많은 데이터를 공유하므로 이러한 방식이 접합해 보일 수도 있다. 하지만 이 유혹은 떨쳐내라! 이들 두 객체의 모적은 완전히 다르다.

결론

업무 규칙은 사용자 인터페이스나 데이터베이스와 같은 저수준의 관심사로 인해 오염되어서는 안 되며, 원래 그대로의 모습으로 남아 있어야 한다.

업무 규칙은 시스템에서 가장 독립적이며 가장 많이 재사용할 수 있는 코드여야 한다.

21장. 소리치는 아키텍처

아키텍처의 테마

이바 야콥슨의 저서 객체 지향 소프트웨어 엔지니어링에서 소프트웨어 아키텍처는 시스템의 유스케이스를 지원하는 구조라고 했다.

아키텍처는 프레임워크에 대한 것이 아니다(그리고 절대로 그래서도 안된다). 아키텍처를 프레임워크로부터 제공받아서는 절대 안 된다. 프레임워크는 사용하는 도구일 뿐, 아키텍처가 준수해야 할 대상이 아니다. 아키텍처를 프레임워크 중심으로 만들어버리면 유스케이스가 중심이 되는 아키텍처는 절대 나올 수 없다.

아키텍처의 목적

좋은 아키텍처는 유스케이스를 그 중심에 두기 때문에 프레임워크나 도구, 환경에 전혀 구애받지 않고 유스케이스를 지원하는 구조를 아무런 문제 없이 기술할 수 있다.

하지만 웹은?

웹은 아키텍처가 아니다. 과도한 문제를 일으키거나 근본적인 아키텍처를 뜯어고치지 않더라도 시스템은 콘솔 앱, 웹, 웹 앱, 리치 클라이언트 앱, 심지어 웹서비스 앱으로도 전달할 수 있어야 한다.

프레임워크는 도구일 뿐, 삶의 방식은 아니다.

어떻게 하면 아키텍처를 유스케이스에 중점을 둔 채 그대로 보존할 수 있을지를 생각하라. 프레임워크가 아키텍처의 중심을 차지하는 일을 막을 수 있는 전략을 개발하라.

테스트하기 쉬운 아키텍쳐

아키텍처가 유스케이스를 최우선으로 한다면, 그리고 프레임워크와는 적당한 거리를 둔다면 프레임워크를 전혀 준비하지 않더라도 필요한 유스케이스 전부에 대해 단위 테스트를 할 수 있어야 한다. 테스트를 돌리는 웹 서버가 반드시 필요한 상황이 되어서는 안된다.

엔티티 객체는 반드시 POJO여야 하며, 프레임워크나 데이터베이스, 또는 여타 복잡하 것들에 의존해서는 안된다.

결론

새로 합류한 프로그래머가 시스템이 어떻게 전달될지 알지 못한 상태에서도 시스템의 모든 유스케이스를 이해할 수 있어야 한다.