1. 소개

1.1. 설계와 아키텍처란?

“엉망으로 만들면 깔끔하게 유지할 때보다 항상 더 느리다”

- 전체 시스템을 재설계하는 것 또한 다시 시작하면 더 좋은 코드를 만들 수 있을거라는 자기 과신이라고 말하는데, 저자는 처음에 좋은 아키텍처를 설계하는 것과 더불어 기존 코드에 대한 리팩토링도 중요하게 생각하는 것 같다.
- 좋은 아키텍처를 이해하려면 좋은 아키텍처가 지닌 속성들을 알아야 한다.

1.2. 두 가지 가치에 대한 이야기

1) 행위: 기능 명세서나 요구사항의 구현과 구체화. 일반적으로 프로그래머는 행위를 자신의 일의 전부라고 착각한다.

2) 아키텍처: 변경하기 쉬운 제품을 개발하는 것. 변경에 드는 비용(시간)은 변경되는 범위에 비례할 뿐 형태에 따라 달라져서는 안 된다.

- 행위는 긴급하고 아키텍처는 중요하다. 중요한 일은 긴급한 일보다 우선된다.
- 개발자는 긴급하지만 중요하지 않은 일을 먼저 처리하라는 요구에 대해 소프트웨어를 보호할 책임이 있다.

2. 벽돌부터 시작하기: 프로그래밍 패러다임

2.1. 구조적 프로그래밍

- 프로그래밍을 테스트 가능한 기능 단위로 분해한 형태
- 기존에 goto로 어느 라인이나 이동할 수 있었던 프로그래밍을, 순차 실행과 제어문(순차, 분기, 반복)이라는 구조로 제한하게 되었다.

2.2. 객체 지향 프로그래밍

- 일반적으로 OO(Object-Oriented)는 캡슐화, 상속, 다형성으로 설명된다. 하지만 C에서도 헤더에서 구조체 정의를 숨김으로써 더 완벽한 캡슐화를 구현할 수 있으며, 상속도 동일한 순서의 변수를 선언한 구조체로 흉내낼 수 있다.
- OO의 핵심은 다형성으로, 이는 고수준 모듈이 저수준 모듈에 의존하지 않고 둘 다 추상화된 인터페이스에 의존하는 형태가 가능하게 한다. 이를 통해 각 모듈은 독립성을 가질 수 있다.

2.3. 함수형 프로그래밍

- 가변 변수 대신 트랜잭션을 저장함으로써 동시성과 교착상태 등의 문제로부터 데이터를 보호할 수 있다.

3. 설계 원칙 (SOLID 원칙)

3.1. SRP(Single Responsibility Principle): 단일 책임 원칙
“단일 모듈은 변경의 이유가 하나뿐이어야 한다.”
- 함수가 하나의 일만 해야 한다는 원칙과는 다르다. 오히려 서로 다르게 변경될 수 있는 부분들에 의도적으로 중복된 함수가 존재할 경우 중복 코드의 병합에 위험이 뒤따른다.
- 이 상황에 대한 해결책으로 데이터와 함수를 분리하여, 같은 데이터를 이해관계자에 따라 다른 방식으로 처리하게 만들 수 있다.
ㄴ 이 경우 각각의 클래스를 인스턴스화하고 추적해야 하는데, Facade 패턴을 도입하거나 공통적인 메서드는 공통 클래스에 유지하는 방식을 고려할 수 있다.

3.2. OCP(Open-Closed Principle): 개방-폐쇄 원칙
“확장에는 열려 있어야 하고, 변경에는 닫혀 있어야 한다.”
- 컴포넌트 관계는 단방향으로, 보호하려는 컴포넌트의 방향으로만 이루어진다. 핵심적인 규칙을 담은 컴포넌트는 외부의 어떤 변경에도 영향을 받지 않는 계층 구조가 형성된다.
ㄴ 인터페이스를 이용하여 의존성을 역전시켜 방향성을 제어하거나, 외부에서 보호하려는 컴포넌트의 내부에 대해 너무 많이 알지 못하도록 은닉할 수 있다.

3.3. LSP(Liskov Substitution Principle): 리스코프 치환 원칙
- 상위 타입의 자리를 하위 타입으로 전환해도 프로그램의 동작이 변하지 않아야 한다.

3.4. ISP(Interface Segregation Principle): 인터페이스 분리 원칙
- 오퍼레이션 단위로 인터페이스를 분리해야 한다.
- 사용하는 쪽에서 불필요한 기능이 포함되지 않도록, 필요 이상으로 많은 것을 포함하는 모듈에 의존하지 않는 것이 좋다.

3.5. DIP(Dependency Inversion Principle): 의존성 역전 원칙
- 변동성이 큰 구체적인 클래스는 직접적으로 참조하거나 파생하지 않는 게 좋다.
ㄴ 참조가 필요하다면 인터페이스나 추상 팩토리를 활용할 수 있다.
- DIP에 위배되는 클래스는 적은 수의 구체적인 컴포넌트로 모아서 시스템의 다른 부분과 분리할 수 있다.

4. 컴포넌트 원칙

4.1 컴포넌트
배포 단위. 시스템의 구성 요소로서 독립적으로 배포할 수 있는 가장 작은 단위.
- SOLID 원칙은 방에 벽돌을 배치하는 방법, 컴포넌트 원칙은 건물에 방을 배치하는 방법
- 초기 컴퓨터: 메모리가 작아 컴파일에 오랜 시간이 걸렸고, 라이브러리와 애플리케이션 코드를 분리했으나 함수 라이브러리를 추가하여 할당된 메모리를 초과할 경우 추가 할당이 필요해 메모리 단편화를 피할 수 없었다.
- 링킹 로더의 등장: 컴파일러가 메모리에 재배치 가능한 바이너리를 생성하고, 바이너리 안의 함수를 메타데이터로 생성하도록 수정되었다. 애플리케이션은 링킹 로더를 통해 외부 참조

되는 라이브러리를 로드, 외부 정의와 연결시켰다.
ㄴ 이 시절에 컴파일된 바이너리가 지금의 오브젝트 코드를 의미하는가?
- 링커-로더 분리: 함수 라이브러리는 느린 저장장치에 위치하고, 외부 참조를 로드하는 데 걸리는 시간은 프로그램이 커질수록 길어졌다. 링킹 로더는 링커와 로더로 분리되어, 링커가 외부 라이브러리의 링크 완료된 코드를 만들어 주었다. 컴파일과 링크 시간은 길어졌으나, 만들어진 실행 파일에서 라이브러리를 로드하는 속도는 매우 빨라졌다.
- 현재: 전체 모듈을 컴파일하는 시간은 프로그램이 커질수록 계속 길어졌으나, 이 부분은 컴퓨터 성능의 비약적인 발전으로 해소되었다. 링크 속도도 프로그램이 커지는 것 이상으로 빨라졌고, 규모가 작다면 jar, dll, 공유 라이브러리를 링크와 동시에 로드할 수 있게 되었다.

4.2. 컴포넌트 응집도
- 컴포넌트에 어떤 클래스를 포함시킬지 결정하는 원칙
- 프로젝트 초기에는 개발 가능성을 위해 CCP와 CRP를 중심으로 개발하고, 프로젝트가 발전함에 따라 REP의 중요성이 커지게 된다.
(1) 재사용/릴리스 등가 원칙 (REP, Release/Reuse-Equivalence Principle)
“재사용 단위는 릴리스 단위와 같다”
- 컴포넌트를 배포할 때는 릴리스 번호를 붙여야 하고, 하나의 컴포넌트에 속한 클래스와 모듈들은 같은 번호로 함께 릴리스할 수 있어야 한다.
- 준수하지 않을 경우 재사용이 어려운 컴포넌트가 된다.
(2) 공통 폐쇄 원칙 (CCP, Common Closure Principle)
“동일한 이유로 동일한 시점에 변경되는 클래스는 같은 컴포넌트로 묶어라”
- 단일 책임 원칙(SRP)을 컴포넌트에 적용시킨 원칙으로, 어떠한 이유로 코드가 변경되어야 할 때 가급적 변경이 단일 컴포넌트에서 일어나도록 하는 편이 유지보수에 좋다.
- 외부의 다른 변경사항에 닫혀 있어야 한다는 점에서 개방-폐쇄 원칙(OCP)과도 관련이 있다.
- 준수하지 않을 경우 컴포넌트 변경이 너무 빈번하게 발생한다.
(3) 공통 재사용 원칙 (CRP, Common Reuse Principle)
“컴포넌트 사용자들이 필요하지 않은 것에 의존하게 강요하지 말라”
- 특정 컴포넌트에 의존하는 것은 그 컴포넌트의 모든 클래스에 의존하는 것이 된다. 해당 컴포넌트의 사용하지 않는 부분이 변경되어도 함께 재배포될 위험이 있다.
- 컴포넌트 내부의 클래스들은 서로 결합하되, 사용하지 않는 클래스가 있는 다른 컴포넌트에는 의존하지 않아야 한다.
ㄴ 인터페이스 분리 원칙(ISP)을 컴포넌트 관점으로 확장시킨 것
- 준수하지 않을 경우 불필요한 릴리스가 많아진다.

4.3. 컴포넌트 결합
- 컴포넌트 사이의 관계
(1) 의존성 비순환 원칙 (ADP, Acyclic Dependencies Principle)
“컴포넌트 의존성 그래프에 순환이 있어서는 안 된다”
- 다른 부분이 수정되어 프로젝트가 제대로 동작하지 않는 현상(‘숙취 증후군’)을 피하기 위한 해결책 중 하나로, 개발 환경을 컴포넌트 단위로 분리하는 방식이다.

ㄴ 다른 방법으로 주 단위 빌드(Weekly Build)가 흔하게 사용되지만, 프로젝트가 커지면 개발보다 통합에 더 많은 시간이 소요될 수 있다.
- 개별 개발자(개발팀)는 컴포넌트에 릴리스 번호를 붙이고 다른 개발자가 사용할 수 있도록 배포한다. 자신의 컴포넌트는 로컬에서 개발하고, 다른 개발자들은 릴리스된 버전을 사용한다.
- 의존성 구조가 순환하는 부분이 있다면, 순환 중인 컴포넌트의 릴리스는 다른 컴포넌트에 서로 영향을 끼친다. 따라서 하나의 거대한 컴포넌트처럼 정확히 동일한 릴리스를 사용해야 한다.
- 순환을 끊기 위해서는 의존성 역전 원칙(DIP)을 적용하거나, 두 컴포넌트가 의존하는 새 컴포넌트를 만드는 방법이 있다.
ㄴ 두 번째 해결책에서 알 수 있듯, 요구사항의 변경에 따라 컴포넌트 구조도 변화한다.
ㄴ 따라서 컴포넌트는 하향식(top-down) 설계가 불가능하다. 컴포넌트 구조는 지속적으로 변화하므로 시스템에서 가장 먼저 설계할 수 있는 대상이 아니다.

(2) 안정된 의존성 원칙 (SDP, Stable Dependencies Principle)
“더 안정된 방향에 의존하라”
- 안정성(stability): 많은 컴포넌트가 의존하고 있어 변경이 어려운 정도
ㄴ 이 컴포넌트의 클래스에 의존하는 외부 클래스 개수(Fan-in)와 그 반대인 경우(Fan-out)에 대해서, 'Fan-out / (Fan-in + Fan-out)' 으로 측정된다. 높을수록 불안정하다.
- 필연적으로 불안정한 컴포넌트도 있으나, 이에 대한 의존성은 제거하는 것이 좋다.
ㄴ 추상 컴포넌트: 인터페이스만을 포함하는 컴포넌트로 의존성을 역전시킬 수 있다.

(3) 안정된 추상화 원칙 (SAP, Stable Abstractions Principle)
“컴포넌트는 안정된 정도만큼만 추상화되어야 한다”
- 전체 클래스 개수 대비 추상 클래스와 인터페이스 개수가 높을수록 추상화 정도가 높다.
- 안정성과 추상화 정도는 비례해야 한다. 안정성이 높은데 구체적이면 확장과 변경이 어려운 컴포넌트가 되고, 안정성이 낮은데 추상적이면 누구도 구현하지 않은 쓸모없는 클래스로 남게 된다.
ㄴ 단, 구체적이고 많은 의존성을 가진 컴포넌트라도 변동성이 없다면 해롭지 않다.
- SDP와 SAP를 결합하면 컴포넌트에 대한 의존성 역전 원칙(DIP)이 된다.

5. 아키텍처

5.1. 아키텍처란? (15장)
- 시스템의 개발, 배포, 유지보수가 용이하도록 지원하고자 만든 설계나 규칙
- 소프트웨어의 핵심적인 요소인 정책을 식별하고, 세부사항은 정책과 무관하게 분리하여 늦게 결정할 수 있도록 만드는 것이 좋은 아키텍처
ㄴ 예를 들어 추상화된 입출력 인터페이스 없이 특정 장치만 사용하도록 제한하면, 장치 변경이 일어날 때마다 소프트웨어를 전반적으로 수정해야 한다.

5.2. 독립성 (16장)

- 좋은 아키텍처는 시스템의 유스케이스, 운영, 개발, 배포를 지원해야 한다.
 (1) 유스케이스: 아키텍처 수준에서 시스템의 의도를 알아볼 수 있어야 한다.
 (2) 운영: 시스템의 처리 요소 변화에 따른 형태 전환에 대응하기 용이하도록 설계해야 한다.
 (3) 개발: 각 팀이 독립적으로 행동하기 편한 아키텍처로 개발환경을 지원해야 한다.
 (4) 배포: 시스템이 빌드된 후 즉각적으로 배포될 수 있도록 지원해야 한다.

5.2.1. 결합 분리 모드
- 아키텍처에서 서로 같은 이유로 변경되는 것은 하나의 컴포넌트로 묶고, 다른 이유로 변경되는 것은 별개의 컴포넌트로 분리한다. (SRP, CCP)
 ㄴ 계층을 기준으로 UI, 업무 규칙, 데이터베이스 등으로 분리할 수 있다.
 ㄴ 계층과 별개로 유스케이스에 따라 다시 분리할 수 있다. (주문 추가, 주문 삭제 등)
- 유스케이스와 계층에 따라 컴포넌트가 분리되면 개발, 배포에서 독립성이 확보된다.
 ㄴ 개발: 서로 다른 계층이나 유스케이스를 개발하는 팀 간에 간섭의 여지가 사라짐
 ㄴ 배포: 운영 중인 시스템에 영향을 주지 않고도 간편하게 유스케이스 추가가 가능
- 두 동시에 변경해야 하는 중복 코드는 나쁜 것이지만, 서로 다른 이유로 변경되는 우발적 중복까지 통합해버리기 않도록 유의할 것
- 결합 분리는 소스 코드 수준, 바이너리 코드(배포) 수준, 실행 단위(서비스) 수준에서 가능하며, 프로젝트가 성장함에 따라 변화할 수 있도록 만드는 것이 좋다.
 ㄴ 소스 수준 분리: 다른 모듈의 재컴파일 없이도 모듈을 수정할 수 있다.
 ㄴ 배포 수준 분리: 다른 모듈의 재빌드 없이 특정 모듈을 독립적으로 배포할 수 있다.
 ㄴ 서비스 수준 분리: 소스와 바이너리 변경에 독립적으로 네트워크 패킷만으로 통신

* 유니티 프로젝트 내 모듈을 소스 수준 분리, 파이어베이스를 통해 에셋번들이나 remote config 데이터를 다운로드하는 것을 배포 수준 분리라고 볼 수 있을까?

5.3. 경계: 선 긋기 (17장)
- 경계(boundary): 소프트웨어 요소를 분리하여, 한 요소가 다른 요소를 알지 못하게 막음
 ㄴ 나중에 개발되는 요소들이 업무 규칙을 훼손하지 못하게 하여, 세부사항의 결정을 미루는 데 도움이 된다.
- 관련이 없는 것 사이엔 경계가 필요하다.
 ㄴ GUI, 업무 규칙, 데이터베이스 사이엔 경계가 필요하다.
 ㄴ 예를 들어 게임에서 내부 로직(업무 규칙)과 화면에 출력하는 것(GUI)은 관련이 없다.
- 더 중요한 방향으로 의존하게 한다.
 ㄴ GUI, DB는 핵심적인 업무 규칙에 의존한다.
 ㄴ 업무 규칙이 인터페이스를 참조하는 방식으로 의존성 역전 원칙을 적용할 수 있다.

5.4. 경계 해부학 (18장)
- 경계 횡단: 런타임에 경계 너머의 기능을 호출하는 것
- 단일체: 소스 수준 분리. 배포 관점에선 경계가 드러나지 않지만 개발과 컴파일을 독립적으로 수행할 수 있으며, 경계 횡단 비용이 매우 싸다.
 ㄴ 경계 횡단은 저수준에서 고수준으로 향해야 한다. 의존성 역전을 활용할 수 있다.

- 배포형 컴포넌트: 배포 수준 분리. DLL과 같이 컴파일하지 않고 바로 사용가능한 배포 단위로, 배포 과정을 제외하면 단일체와 유사하다.
 ㄴ 모든 함수가 동일 프로세서와 주소 공간에 놓이고, 컴포넌트 간 의존성 관리도 단일체와 동일하다. 최초 호출 시 로딩시간이 걸릴 수 있지만 그 외에는 호출 비용도 저렴하다.
 ㄴ 단일체와 배포형 컴포넌트는 스레드를 활용하여 실행 계획과 순서를 체계화할 수 있다.
- 로컬 프로세서: 독립된 주소공간에서 실행되는 물리적인 경계. 터미널이나 시스템 콜을 통해 생성된다.
 ㄴ 소켓이나 메시지 큐 등 운영체제가 제공하는 통신 기능으로 통신하므로, 운영체제 호출, 데이터 변환, 문맥 교환 등의 비용이 발생한다.
 ㄴ 역시 저수준 프로세스가 고수준을 향하도록 의존성을 설계해야 한다.
- 서비스: 물리적인 형태를 띠는 가장 강력한 경계. 로컬 프로세서와 마찬가지로 생성되지만 자신의 물리적 위치에 구애받지 않는다.
 ㄴ 서로 다른 프로세서나 멀티코어에서 동작할 수 있다.
 ㄴ 모든 통신은 네트워크를 통해 이루어진다. 통신비용이 비싸므로 빈번한 통신을 피해야 하며 지연(latency) 문제를 고수준에서 처리할 수 있어야 한다.

5.5. 정책과 수준 (19장)
- 프로그램은 입력을 출력으로 변환하는 정책을 상세하게 기술한 설명서
 ㄴ 한 정책은 여러 개의 정책으로 분리될 수 있다.
 ㄴ 다른 이유로 혹은 다른 시점에 변경되는 정책은 다른 수준의 정책으로 분리해야 한다.
 ㄴ 소프트웨어 아키텍처 개발은 정책을 재편성하고, 비순환 방향 그래프로 구성하는 일
- 수준(level): 입력과 출력까지의 거리. 입출력에서 멀리 떨어질수록 고수준이 된다.
 ㄴ 데이터 흐름과 의존성 방향은 일치하지 않을 수 있다. 데이터 흐름은 입력(저수준) -> 고수준 -> 출력으로 향하더라도, 소스 코드 의존성은 저수준에서 고수준을 향하게 해야 한다.
 ㄴ 고수준 정책은 저수준에 비해 더 추상화되어 있으며 변경될 가능성이 낮다.

5.6. 업무 규칙 (20장)
- 사업적으로 수익을 얻거나 비용을 줄일 수 있는 규칙 또는 절차. 프로그램의 핵심 기능.
 ㄴ ex) 대출에 이자를 부과하는 것
 ㄴ 누가 어떻게 실행하는지와 관계가 없다
- 엔티티: 핵심 업무 데이터를 기반으로 동작하는 핵심 업무 규칙을 구체화한 것
 ㄴ 핵심 업무 데이터와 규칙을 하나로 묶어 별도의 모듈로 만들어야 하며, 규칙은 인터페이스로 제공한다. 클래스 형태 또는 객체지향 언어를 반드시 사용할 필요는 없다.
 ㄴ 독립적으로 존재하며 어디에서든 업무 수행이 가능해야 한다. DB, UI 등에 오염되면 안 된다.
- 유스케이스: 특정 어플리케이션에 특화된 업무 규칙. 엔티티의 업무를 언제, 어떻게 호출할지 명시한다.
 ㄴ 입력과 출력, 처리 단계를 포함한다. UI가 어떤 형태인지는 기술하지 않는다.
 ㄴ 엔티티보다 저수준 개념이며 엔티티에 의존한다.
- 요청 및 응답 모델: 유스케이스의 입력과 출력을 사용자나 다른 컴포넌트와 주고받는 방식
 ㄴ 유스케이스는 단순한 데이터 구조만을 입력받고 출력하며, 요청 및 응답 모델에 대해서는

모르도록 의존성을 제거해야 한다.

5.7. 소리치는 아키텍처 (21장)
- 아키텍처를 보고 애플리케이션의 유스케이스를 떠올릴 수 있어야 한다.
 ㄴ 아키텍처는 애플리케이션의 유스케이스를 지원하기 때문
- 아키텍처는 프레임워크에 의존해서는 안 된다. 프레임워크의 결정을 미룰 수 있어야 하며, 프레임워크 없이도 필요한 유스케이스에 대한 단위 테스트가 가능해야 한다.

5.8. 클린 아키텍처 (22장)
- 모든 아키텍처는 관심사의 분리를 목표로, 시스템이 몇 가지 특징을 지니게 만든다.
 ㄴ 프레임워크 독립성: 아키텍처는 프레임워크 존재 여부에 의존하지 않는다.
 ㄴ 테스트 용이성: 업무 규칙은 UI, DB, 웹 서버 등이 없어도 테스트할 수 있다.
 ㄴ UI 독립성: 시스템의 나머지 부분을 변경하지 않도고 쉽게 UI를 변경할 수 있다.
 ㄴ DB 독립성: 업무 규칙이 DB에 결합되지 않아 DB 교체가 가능하다.
 ㄴ 외부 에이전시에 대한 독립성: 업무 규칙은 외부 세계와의 인터페이스에 대해 모른다.

5.8.1. 의존성 규칙
- 소스 코드 의존성은 반드시 고수준을 향해야 한다.
 ㄴ 소프트웨어에서 저수준의 무언가가 변화하더라도 고수준이 영향을 받아서는 안 된다.
- 다음은 고수준부터 저수준까지의 정책을 계층별로 나열한 것이다.
 ㄴ 계층을 꼭 4개로 구분할 필요는 없지만, 의존성 규칙은 반드시 적용되어야 한다.
(1) 엔티티
 - 엔터프라이즈 업무 규칙에 해당하며, 가장 일반적이고 고수준인 핵심 업무를 캡슐화한다.
(2) 유스케이스
 - 애플리케이션에 특화된 업무 규칙을 포함한다.
 - 엔티티로 들어오고 나가는 데이터 흐름을 조정한다.
 - 엔티티가 유스케이스에 의존하면 안 되고, 인터페이스나 데이터베이스 등 저수준의 변경이 유스케이스에 영향을 줘도 안 된다.
 ㄴ 하지만 운영 관점에서 애플리케이션이 변경되면 유스케이스도 영향을 받을 것이다.
 (3) 인터페이스 어댑터
 - 어댑터들로 구성된다. 컨트롤러, 게이트웨이, 프레젠터 등이 포함된다.
 - 데이터를 유스케이스, 엔티티에게 편리한 형식에서 외부 에이전시에게 편리한 형식으로 변환한다.
 (4) 프레임워크와 드라이버
 - 외부 에이전시로서 웹, UI, 외부 인터페이스, DB 등 세부사항으로 구성된다.

5.8.2. 경계 횡단하기
 * 경계 횡단의 개념에 대해서는 5.4 참고
- 외부 계층의 호출이 필요한 경우 해당 계층으로의 입출력 인터페이스를 만들어 직접적인 외부 데이터 호출을 피할 수 있다.
- 경계 횡단 시 데이터는 항상 내부(고수준)에서 사용하기 편리한 형태를 가져야 한다.

ㄴ 외부 원에서 사용하기 적합한 경우, 데이터 해석에 외부의 무언가를 알아야 해서 의존성 규칙이 깨질 수 있기 때문

5.9. 프레젠터와 험블 객체 (23장)
- 험블 객체 패턴: 테스트하기 쉬운 행위와 어려운 행위(=험블)를 분리하는 디자인 패턴
 ㄴ 프레젠터와 뷰: 뷰는 화면을 보면서 검사해야 하므로 테스트가 어려운 험블 객체이다. 따라서 데이터를 화면으로 전달하는 간단한 일만 처리하게 한다. 프레젠터는 애플리케이션의 데이터를 화면에 표시하는 포맷으로 변환하므로 테스트가 쉽다.
- 험블 객체 패턴을 적용하여 행위를 두 부분으로 나누면 아키텍처의 경계가 정의된다.
 ㄴ 데이터베이스 계층과 유스케이스 인터렉터 사이에는 게이트웨이가 존재하여, 테스트가 어려운 인터페이스 구현체나 데이터 매퍼는 데이터베이스 계층으로 분리한다. 이렇게 하면 게이트웨이 자체는 더미 데이터로 대체하여 쉽게 테스트할 수 있다.
 ㄴ 다른 서비스와 통신할 경우에도 서비스 리스너를 통해 외부 서비스와 내부 모듈을 분리하는 식으로 험블 객체 패턴을 사용하고 있다.

5.10. 부분적 경계 (24장)
- 경계를 완벽하게 만드는 일은 많은 노력과 비용이 든다.
 ㄴ YAGNI(You Aren't Going to Need It): 나중을 위한다고 지금 필요 없는 작업을 하지 마라
- 아키텍처 경계가 나중에 필요할지도 모른다고 생각된다면, 부분적 경계를 구현할 수 있다.
 ㄴ 1) 마지막 단계 건너뛰기: 독립적인 컴파일과 배포가 가능한 형태로 제작하되, 단일 컴포넌트로 관리하는 것
 ㄴ 2) 일차원 경계: 쌍방향 인터페이스 대신 특정 기능에 의존성 역전을 적용해 추후 분리할 수 있는 인터페이스의 형태로 사용할 수 있다. (ex: 전략 패턴)
 ㄴ 3) 퍼사드(Facade): 클라이언트가 모든 서비스를 퍼사드 클래스를 통해서만 사용하는 형태. 의존성 역전이 적용되지 않아 서비스 클래스 하나가 변경되면 모두 재컴파일해야 한다.

5.11. 계층과 경계 (25장)
- 간단한 게임에서도 잠재적으로 분리해야 하는 아키텍쳐 경계가 존재한다.
 ㄴ 언어, 입출력 방식, 데이터 종류 등
- 규칙, UI, DB 외에도 네트워크와 같은 컴포넌트의 추가되며 데이터 흐름이 늘어날 수 있으며, 규칙 내에서도 여러 정책들이 분리될 수 있다.
- 아키텍트는 경계, 부분적 경계, 무시할 경계 등을 결정해야 하며, 한 번의 결정으로 끝나지 않고 지속적으로 프로젝트를 관찰하며 경계의 필요성과 경계 구현 비용을 생각해야 한다.

5.12. 메인 컴포넌트 (26장)
- 메인(Main) 컴포넌트: 시스템의 진입점. 나머지 컴포넌트를 생성, 조정, 관리하는 컴포넌트.
 ㄴ 필요한 모든 요소를 생성한 후 고수준에 제어권을 넘기는 역할을 한다.
 ㄴ 가장 지저분하며 저수준의 컴포넌트로 운영체제 외엔 무엇도 여기에 의존하지 않는다.

5.13. 크고 작은 모든 서비스들 (27장)

- 서비스: 사용자와의 상호작용 없이 백그라운드에서 특정 작업을 수행하는 프로그램
- 서비스 아키텍쳐는 서비스 사이의 독립성을 보장하지 않는다
 ㄴ 결합 분리의 오류: 프로세서나 네트워크의 공유 자원을 통해 강하게 결합될 수 있음
 ㄴ 개발/배포 독립성의 오류: 결합된 정도에 맞게 개발/배포해야 하며, 모노리틱 등의 다른 시스템에서도 독립적인 개발/배포 시스템의 구축은 가능하다.
- 각 서비스가 분리되어 있어도 강하게 결합되어 있다면, 새로운 기능을 구현할 때 시스템 전체를 변경해야 할 수도 있다. (야옹이 문제)
 ㄴ 서비스는 하나 또는 다수의 컴포넌트로 만들 수 있다. 즉 서비스는 SOLID 원칙에 따라 설계가 가능하다.

5.14. 테스트 경계 (28장)
- 테스트는 시스템의 일부이며, 테스트 대상에게 의존하는 구체적인 컴포넌트이다.
 ㄴ 다른 컴포넌트처럼 설계 규칙을 따르는 것이 좋다. 예를 들어 변동성이 있는 GUI 등에 의존하지 않고도 테스트가 가능해야 한다.
- 테스트 API: 시스템의 제약을 무시하고 테스트 가능한 상태를 강제하는 API.
 ㄴ 애플리케이션에서 테스트 구조를 분리하여 별개로 수정 가능한 형태를 목표로 제작된다.
 ㄴ 테스트 API가 일반 사용자에 노출되는 것을 피하고 싶다면 테스트 API의 일부 위험한 구현부를 별도의 컴포넌트로 분리할 수 있다.

6. 세부사항

6.1. 데이터베이스는 세부사항이다
- 데이터베이스는 데이터에 접근할 방법을 제공하는 유틸리티로, 데이터 모델 자체가 아님
- 디스크든 테이블이든 저장된 데이터를 메모리에 올려놓은 후에는 포인터나 참조를 사용하므로, 아키텍처의 관점에서 저장된 데이터의 형태는 세부사항으로만 다루어져야 한다.

6.2. 웹은 세부사항이다
- 웹은 상호작용을 위한 GUI를 제공하는 입출력 장치이므로, 업무 규칙과는 분리되어야 한다.
 ㄴ 웹은 연산을 서버 또는 브라우저에 두는 방식 사이를 지속적으로 오가고 있다.
 ㄴ DB와 마찬가지로 외부 사정에 의해 변경될 수 있다.

6.3. 프레임워크는 세부사항이다
- 프레임워크는 의존성 규칙을 위반하며, 장기적으로 소프트웨어에 제약이 될 수 있다.
 ㄴ 프레임워크는 모든 사용자의 문제를 해결하기 위해 만들어지지 않았음에도, 핵심적인 업무 규칙이 프레임워크를 상속받기를 요구한다.
 ㄴ 프레임워크가 개발 초기엔 유용했더라도, 내 소프트웨어에 맞지 않는 방향으로 진화하거나 더 나은 프레임워크가 등장하여 갈아타고 싶을 수도 있다.
- 프레임워크는 핵심 코드에 두는 대신, 프록시 형태로 만들어 업무 규칙에 플러그인할 수 있는 컴포넌트에 위치시켜야 한다.
 ㄴ C++과 STL처럼 결합해야만 하는 프레임워크도 있다. 다만 섣불리 결합하지는 말 것.

6.4. 사례 연구
- 사례를 통해 구현 세부사항에서 생각해야 할 점을 알아본다. Java로 온라인 서점을 구축하여 고객이 주문 상태를 조회할 수 있어야 하는 유스케이스를 구현해야 한다고 가정한다.
- 최적의 설계를 했더라도 구현 전략을 고려하지 않으면 설계가 망가질 수 있다.

6.4.1. 설계방식
1) 계층 기반 패키지
- 전통적인 수평 계층형 아키텍처
- 의존성은 모두 아래 계층으로 향한다.
ㄴ 간편하게 비순환 그래프를 만들 수 있으나, 일부 계층을 건너뛰어 의도치 않은 형태로 구현될 수 있다.
- 프로그램이 작은 경우, 작동 가능한 소프트웨어를 빠르게 구축할 수 있다.
- 계층형 아키텍처들은 구조가 유사하여 아키텍처만 보고 업무 도메인을 유추할 수 없다.
2) 기능 기반 패키지
- 서로 연관된 기능을 기준으로 코드를 나누는 방식
- 구조에서 업무 도메인이 드러나며, 유스케이스 변경 시 수정 작업이 좀 더 쉬워진다.
ㄴ 변경할 코드가 한 패키지에 있어서 코드를 찾거나 수정하기 쉽기 때문
- 개발팀이 계층 기반에서 기능 기반으로 변경하는 일은 흔하지만, 어느 것이 낫다기보다 둘 다 차선책이다.
3) 포트와 어댑터
- 도메인(핵심 업무)을 내부에, 인프라(세부사항)을 외부로 분리한 형태
- UI, 프레임워크, DB 등을 도메임에서 분리한다. 반드시 외부가 내부에 의존해야 한다.
4) 컴포넌트 기반 패키지
- 컴포넌트: 인터페이스로 감싸진 연관된 기능들의 묶음
- 1번 방식에서 생기는 계층 건너뛰기 문제를 구조적으로 해결하기 위해, 건너뛰면 안 되는 연관 기능들을 컴포넌트로 묶고 인터페이스를 통해 사용하도록 만들 수 있다.

6.4.2. 조직화 vs 캡슐화
- 패키지 내의 모든 요소를 public으로 선언한다면, 패키지는 조직도를 그리기 위한 수단일 뿐 결과물에 어떤 영향도 주지 못한다.
ㄴ 위 설계방식의 어떤 것을 채택하든 동일한 접근법을 적용할 수 있게 되기 때문
- 패키지가 주는 이점을 얻으려면, 패키지에 접근할 수 있는 통로 외에는 제한적인 접근 지시자를 사용해야 한다.
- 접근 지시자 외에도 특정 모듈 시스템을 이용해 외부에 공표할 타입을 분리하거나, 외부에서 사용할 코드(인프라)와 그렇지 않은 코드(도메인)를 두 개의 소스 코드 트리로 구분하는 방법 등이 있다.