Clean Architecture

소프트웨어 구조와 설계의 원칙

12장 - 컴포넌트
13장 - 컴포넌트 응집도
14장 - 컴포넌트 결합
15장 - 아키텍처란?

Date
2023.08.23

Presenter
황유란

컴포넌트

SOLID - 벽과 방에 벽돌을 배치하는 방법

컴포넌트 - 빌딩에 방을 배치하는 방법

- 시스템이 **배포할 수 있는 가장 작은 단위** (자바의 jar파일)
- 여러 컴포넌트들을 링크하여 실행 가능한 단일 파일로 생성 가능하고 독립적 배포도 가능
- 잘 설계된 컴포넌트는 **독립적으로 배포 가능**한, **개발 가능**한 능력을 갖춰야 한다.

컴포넌트의 역사

- **프로그래머가** 메모리에서의 **프로그램 위치와 레이아웃을 직접 설정**
- **라이브러리는** 소스 코드 형태로 유지되어 **컴파일시 애플리케이션 코드에 포함시**켜서 단일 프로그램으로 컴파일
  - 프로그램이 커질수록 컴파일이 오래 걸림
- 함수 라이브러리 소스 코드를 **애플리케이션 코드와 분리**
  - 라이브러리를 개별 컴파일하고 메모리 특정 위치에 로드
  - 애플리케이션이 커지면서 **함수 라이브러리 공간을 사이에 두고 애플리케이션 코드가 두 세그먼트로 분리**되어 배치

재배치성

- **재배치가 가능한 바이너리**가 해결책
- **로더는 재배치 코드가 자리할 위치 정보**를 전달받음
    - 재배치 코드에는 로드한 데이터에서 어떤 부분을 수정해야 정해진 주소에서 로드할 수 있는지 알려주는 플래그가 삽입 (보통 메모리 시작 주소)
- **컴파일러는** 재배치 가능한 바이너리 안 **함수 이름을 메타데이터 형태로 생성**하도록 수정
    - 라이브러리 함수 이름을 외부 참조로 생성
    - 라이브러리 정의 프로그램이라면 외부 정의로 생성
    - 외부 정의 로드 위치가 정해지면 로더가 외부 참조를 외부 정의에 링크시킬 수 있음
- **링킹 로더의 탄생**

링커

- 링킹 로더 덕분에 프로그램을 개별적으로 컴파일하고 로드할 수 있는 단위로 분할할 수 있게 됨
- 프로그램이 커지면서 링킹 로더가 너무 느려져 로드와 링크가 두 단계로 분리
- 링크는 링커라는 별로 애플리에이션이 처리 → 로드가 빨라짐
- 머피의 법칙
  - 컴파일하고 링크하는 데 사용 가능한 시간을 모두 소모할 때까지 프로그램은 커진다
  - **로드는 빨라졌지만 컴파일 - 링크의 병목 구간**
- 무어의 등장
  - **프로그램이 커지는 속도보다 링크 속도가 더 빨라짐**
  - **링크-로더 동시**에 하는 게 가능해짐 → **컴포넌트 플러그인 아키텍처**가 탄생

결론

동적 링크 파일 = 소프트웨어 컴포넌트

컴포넌트 응집도

- 어떤 클래스를 어떤 컴포넌트에 포함시켜야 할까?
- 컴포넌트 응집도와 관련된 세가지 원칙
  - REP: 재사용/릴리스 등가 원칙
  - CCP: 공통 폐쇄 원칙
  - CRP: 공통 재사용 원칙

REP: 재사용/릴리스 등가 원칙 (Reuse/Release Equivalence Principle)

재사용 단위는 릴리스 단위와 같다

- **모듈 관리 도구**를 통해 소프트웨어 **재사용이 가능**해짐
- 릴리스 절차를 통해 추적 관리가 되지 않거나 **릴리스 번호가 부여되지 않는다면 컴포넌트는 재사용할 수 없다.**
  - 릴리즈 번호가 없다면 재사용 컴포넌트들이 서로 **호환되는지 보증할 방법**이 없음
  - 개발자들은 릴리즈 단위로 새로운 버전을 쓸지 기존 것을 계속 사용할 지 결정

REP: 재사용/릴리스 등가 원칙 (Reuse/Release Equivalence Principle)

- 아키텍쳐 관점
  - **단일 컴포넌트는 응집성 높은 클래스와 모듈들로** 구성되어야 함
    - 컴포넌트를 구성하는 모든 모듈은 서로 공유하는 중요한 테마나 목적이 있어야 한다.
  - 하나의 컴포넌트로 묶인 클래스와 모듈은 **함께 릴리스 할 수 있어야**
    - 버전 번호가 같아야 하고 동일한 릴리스로 **추적관리**되어야 한다.
    - 재사용을 할 수 있다.

CCP: 공통 폐쇄 원칙 (Common Closure Principle)

동일한 이유로 동일한 시점에 변경되는 클래스를 같은 컴포넌트로 묶어라.

서로 다른 시점에 다른 이유로 변경되는 클래스는 다른 컴포넌트로 분리하라

- **SRP를 컴포넌트 관점**으로 작성한 것
- **유지보수성이** 재사용성보다 훨씬 중요
  - 여러 컴포넌트에서 변경이 발생하는 것보다는 **한 컴포넌트에서 다 발생하는 편이 낫다.**
- OCP의 Closure와 뜻이 같다.
  - **변경에 영향을 주는 컴포넌트가 최소한**으로 한정되도록

CRP: 공통 재사용 원칙 (Common Reuse Principle)

> **컴포넌트 사용자들은 필요하지 않는 것에 의존하게 강요하지 말라**

- **같이 재사용되는 경향이 있는 클래스와 모듈들은 같은 컴포넌트에** 포함해야 한다.
  - ex. container 클래스와 해당 클래스의 이터레이터(iterator) 클래스
- **동일한 컴포넌트로 사용하면 안되는 클래스가 무엇인지**도 말해준다.
  - 사용하는(using) 컴포넌트가 사용되는 (used) 컴포넌트의 단 하나의 클래스만 사용하더라도 의존성 발생
  - ISP의 포괄적인 버전
    - ISP - 사용하지 않는 메서드가 있는 클래스에 의존하지 말라
    - CRP - 사용하지 않는 클래스를 가진 컴포넌트에 의존하지 말라

컴포넌트 응집도에 대한 균형 다이어그램

- **REP와 CCP는 컴포넌트를 크게** 만든다
- **CRP는 컴포넌트를 작게** 만든다.
- REP, CRP에 중점 - 사소 변경이 발생했을 때 많은 컴포넌트에 영향을 미침
- REP, CCP에 중점 - 불필요한 릴리스가 빈번
  - **개발팀이 관심을 기울이는 부분을 충족시키게 균형을 맞춰야**
  - 초기에는 CCP가 중요하지만 시간이 지날수록 REP가 중요해짐

ADP: 의존성 비순환 원칙

> 컴포넌트 의존성 그래프에 순환(cycle)이 있어서는 안 된다.

- **숙취 증후군 (the morning after syndrome)**
  - 내가 의존한 무언가를 다른 사람이 수정하면서 망가진 코드를 다시 수정하는 작업을 반복하는 과정
  - 주 단위 빌드와 의존성 비순환 원칙이 해결책

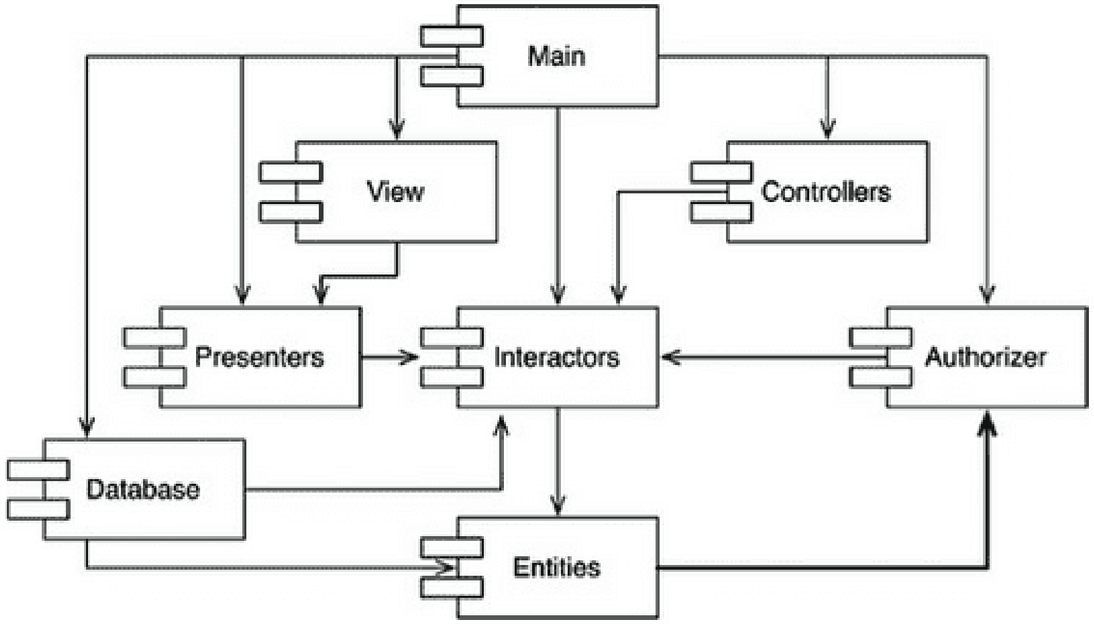
ADP: 의존성 비순환 원칙

- 주 단위 빌드 (weekly build)
  - 일주일의 첫 4일동안은 개발자들은 서로를 신경 쓰지 않는다.
  - 금요일이 되면 변경된 코드를 모두 통합하여 시스템을 빌드
  - 4일동안 개발자들이 고립된 채로 개발할 수 있다는 장점, 금요일에 모든 업보를 치뤄야 한다는 단점
  - 중간 규모 프로젝트에서는 가능하지만 **프로젝트 규모가 커지면** 금요일 하루로는 합치는 게 불가능해지고 통합하는 주기가 늘어나면서 **팀의 효율성이 낮아진다.**

ADP: 의존성 비순환 원칙

- **순환 의존성 제거하기**
  - 의존성 구조에 순환이 있어서는 안됨
  - 컴포넌트 간 의존성 구조를 그리면 **DAG(Directed Acyclic Graph)**가 나와야 한다.
  - 해당 컴포넌트 릴리스에 **영향받는 팀을 쉽게 찾을 수** 있음
    - Main → View → Presenter 순으로 의존하고 있다면 Presenter가 변경되면 View와 Main이 영향을 받는다. 반면 Main은 변경되어도 영향받는 컴포넌트가 없다.
  - **시스템 빌드 구조를 파악**할 수 있음
    - 의존하는 게 없는 Entities부터 빌드하면서 의존받는 게 없는 Main까지 컴파일, 테스트, 릴리스한다.

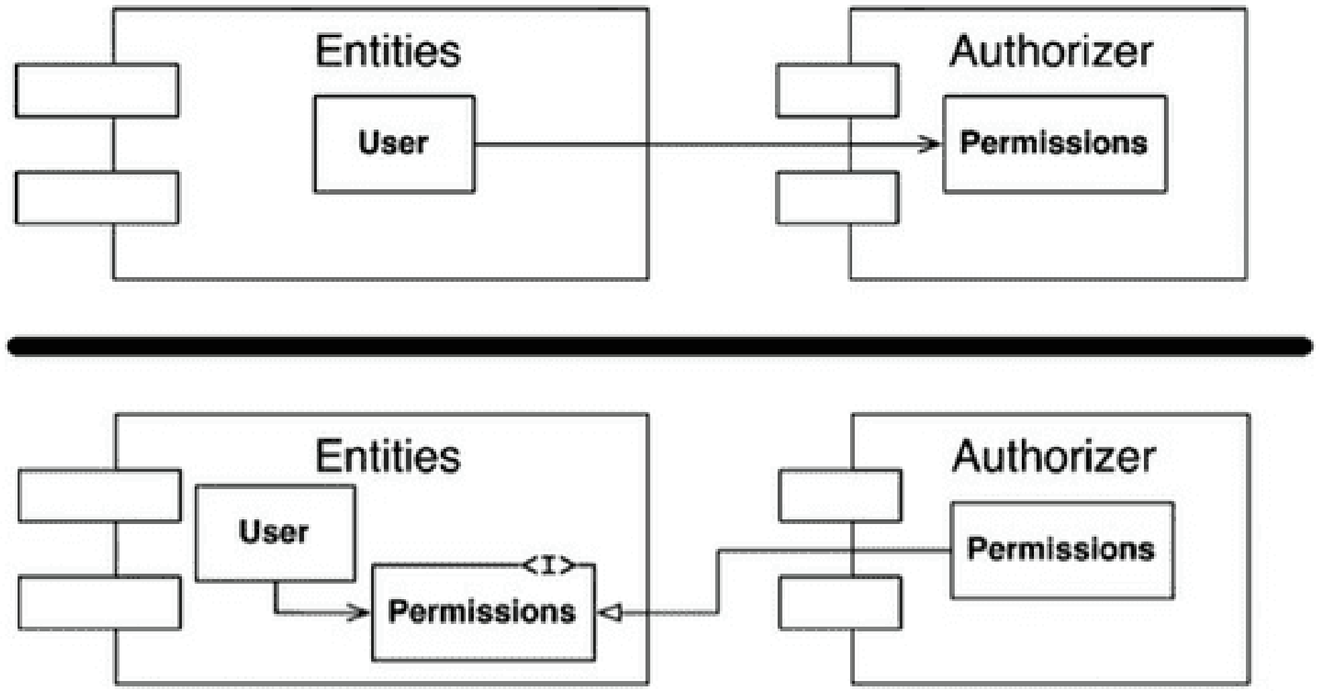
순환이 컴포넌트 의존성 그래프에 미치는 영향



Entities를 사용하는 Database 컴포넌트를 릴리스하려면 Entities와 Entities가 의존하고 있는 Authorizer, Authorizer가 의존하고 있는 Interactors까지 다 릴리스 해야 한다.

순환 끊기

의존성 역전 원칙 (DIP)를 사용한다.

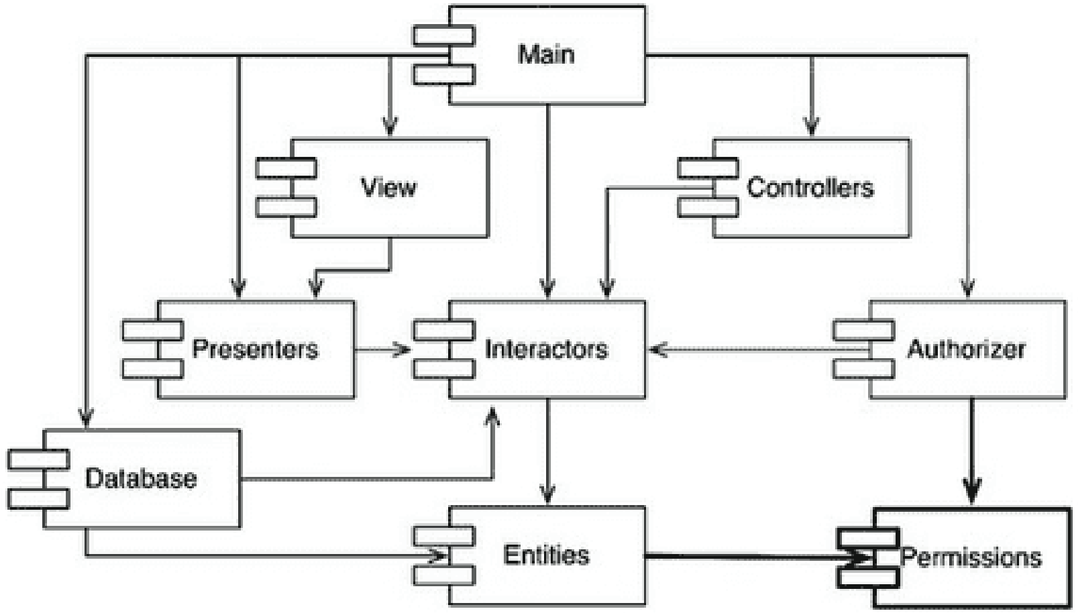


순환 끊기

Entities와 Authorizer가 모든 의존하는

새로운 컴포넌트를 만듦

- 흐트러짐 (Jitters)
  - 요구사항이 변경되면 컴포넌트의 의존성 구조는 서서히 흐트러지며 성장한다.
  - 의존성 순환이 발생하는지 항상 관찰하여야 한다.

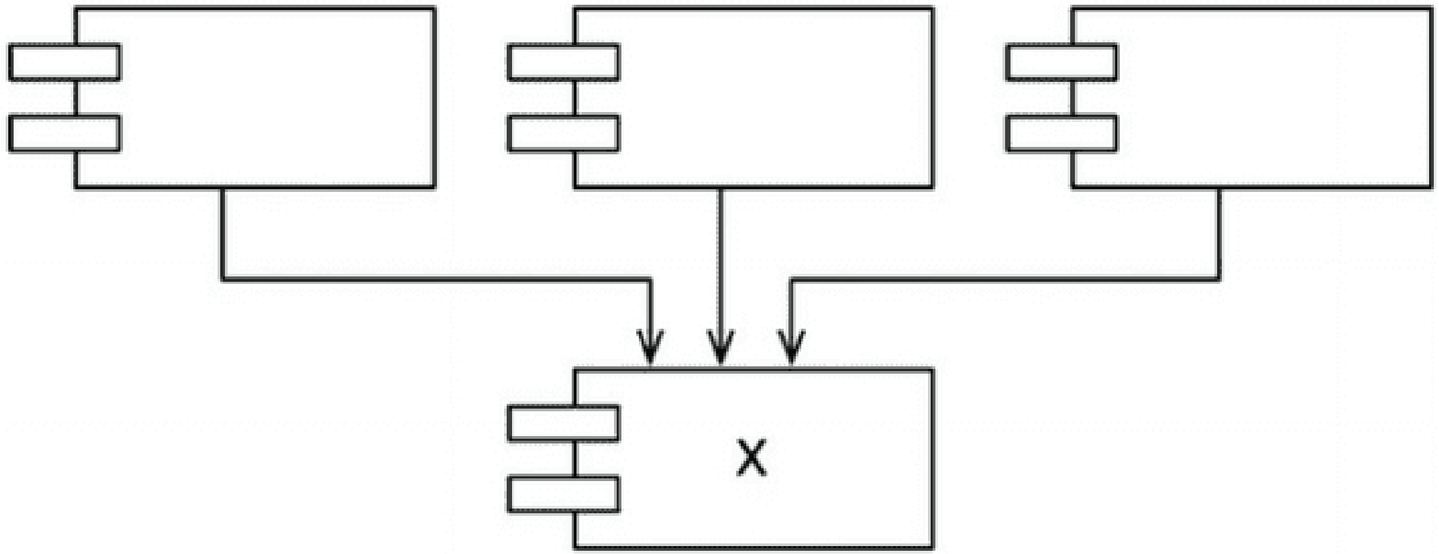


SDP: 안정된 의존성 원칙

안정성의 방향으로 (더 안정된 쪽에) 의존하라

- 설계에서 변경은 불가피하다
- 변동성을 지니도록 컴포넌트를 설계 → 다른 컴포넌트가 의존성을 가지게 되면 변경하기 어려워짐
    - SDP는 **변경하기 어려운 모듈이 변경하기 쉬운 모듈에 의존하지 않도록** 만드는 원칙

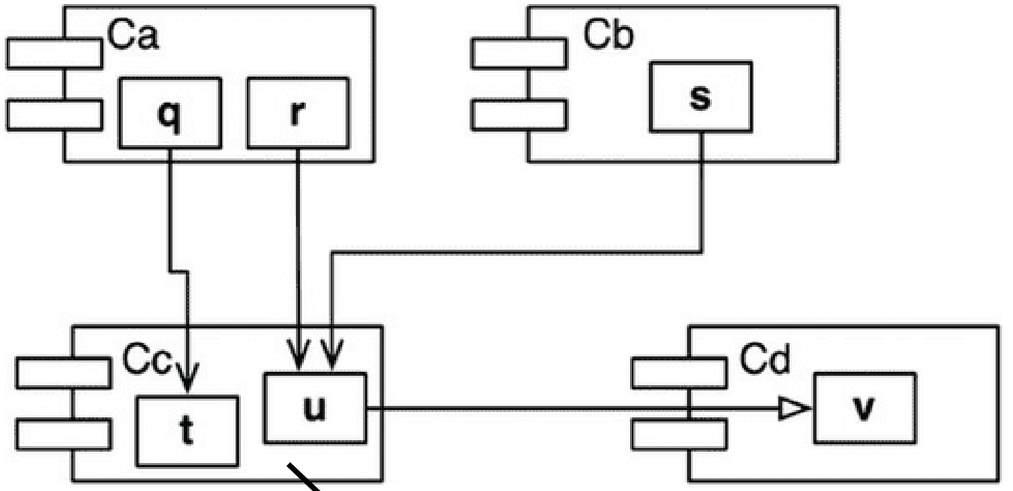
안정성



- X는 안정적이며 세 컴포넌트를 책임지는 책임성이 있고 X가 의존하는 것은 없으므로 독립적이다
- 반대라면 책임성이 없고 의존적인 것

안정성 지표

- Fan-in
  - 안으로 들어오는 의존성. 컴포넌트 **내부 클래스에 의존하는 컴포넌트 외부의 클래스 개수**
- Fan-out
  - 바깥으로 나가는 의존성. 컴포넌트 **외부 클래스에 의존하는 컴포넌트 내부 클래스의 개수**
- **I (불안정성) = Fan-out / (Fan-in + Fan-out)**
  - 0이면 안정된 컴포넌트, 1이면 불안정한 컴포넌트
  - SDP에서 I 지표는 컴포넌트가 의존하는 다른 컴포넌트들의 I보다 커야 한다.



- Fan-in =3
- Fan-out =1
- I = 1 / (3+1) = 1 / 4

추상 컴포넌트

- 인터페이스만 포함하는 컴포넌트
- 정적 타입 언어를 사용할 때 자주 사용하고 꼭 필요한 전략
- 추상 컴포넌트는 상당히 안정적

SAP: 안정된 추상화 원칙

컴포넌트는 안정된 정도만큼만 추상화되어야 한다

- **고수준 아키텍처나 업무 로직**에 관련된 컴포넌트는 **변동이 없기를** 바란다.
- 하지만 고수준 정책을 포함하는 소스 코드는 수정하기가 어려워지면서 **시스템이 유연함을 잃는다.**
- 안정되면서도 변동에 유연하게 만들 수는 없을까? → OCP에서 해답을 얻음
- 안정성과 추상화 정도 사이의 관계를 정의
    - 안정된 컴포넌트 = 추상 컴포넌트, 불안정한 컴포넌트 = 구체 컴포넌트

추상화 정도 측정하기

- 컴포넌트 클래스 총 수(Nc) 대비 인터페이스와 추상 클래스의 개수(Na)
- A의 추상화 정도 = Na / Nc

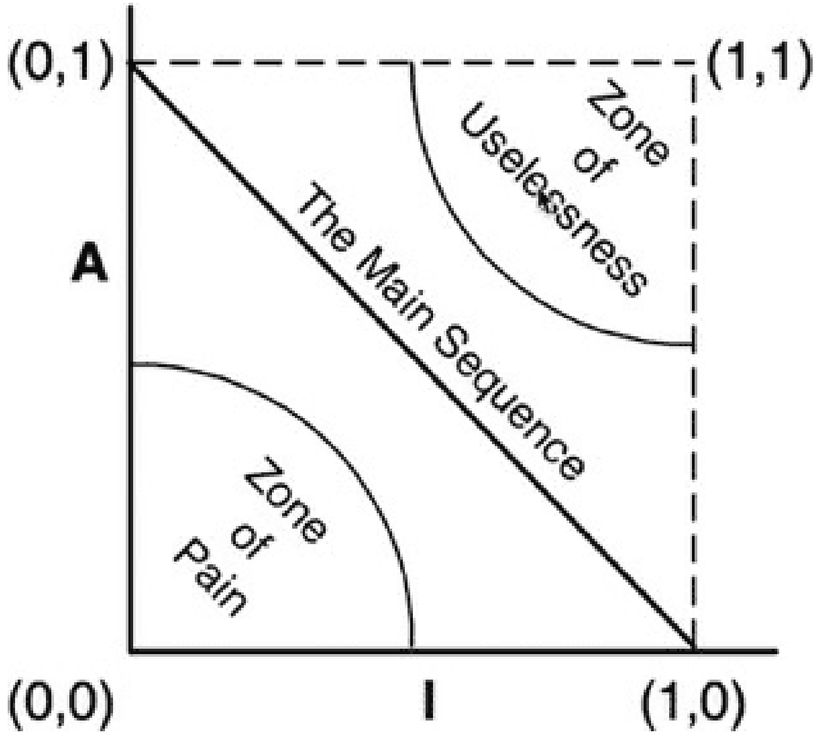
주계열

- 안정성과 추상화 정도 사이의 관계
- 최고로 안정적이고 추상화된 컴포넌트

  = (0,1)
- 최고로 불안정하고구체화된 컴포넌트

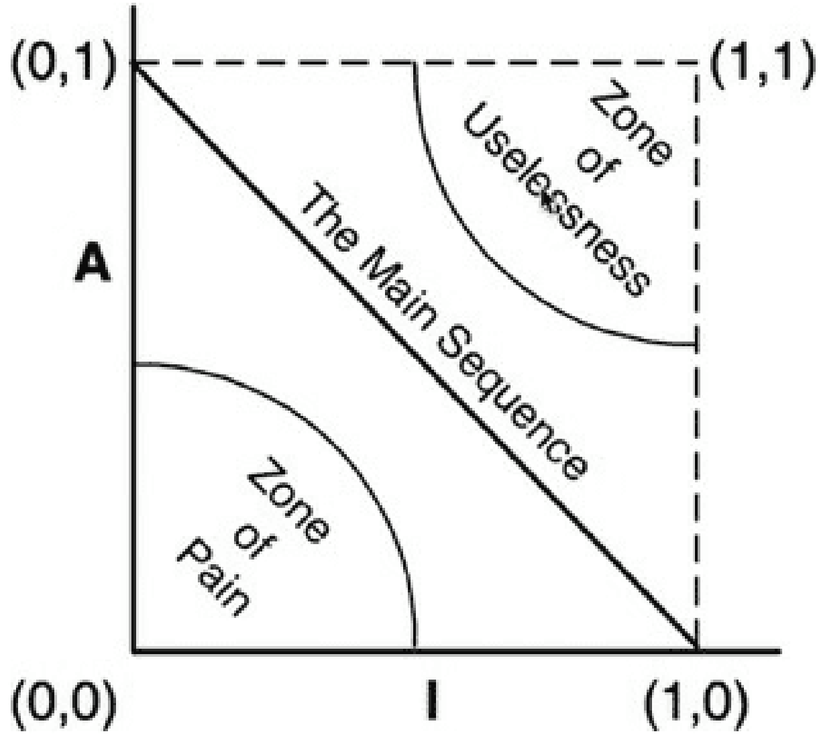
  = (1,0)
- 추상 클래스로부터 파생된 추상 클래스

  = 추상적이면서 의존성을 가짐



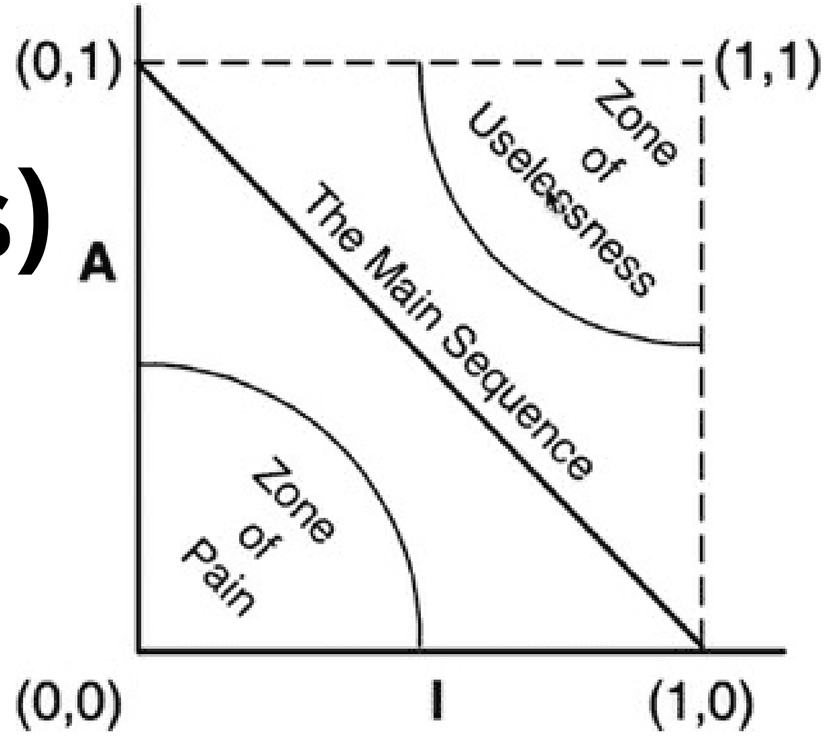
고통의 구역 (zone of pain)

- 이 구역의 컴포넌트들은 **매우 안정적이며 구체적**
- 추상적이지 않으므로 확장할 수 없고 안정적이므로 변경하기도 어렵다
  - 배제해야 할 구역
  - ex. 데이터베이스 스키마, 유틸리티 라이브러리 (String class)



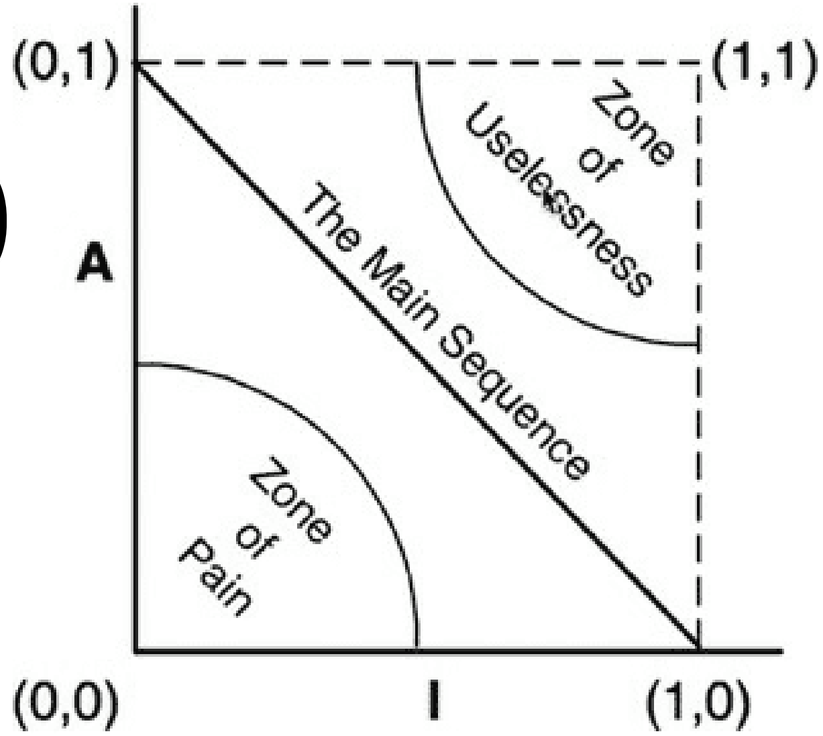
쓸모없는 구역 (zone of uselessness)

- 이 구역 컴포넌트들은 **추상적이면서 의존하지 않음**
- ex. 누구도 구현하지 않은 추상 클래스



주계열
(The main sequence)

- 변동성이 큰 컴포넌트들은 두 배제구역에서 벗어나야 함
- (1,0) ~ (0,1)을 잇는 선문위에 있으면 좋음
  → 이 선분을 **주계열(main sequence)**라 부른다.



주계열 거리

- **주계열로부터 컴포넌트가 얼마나 떨어져 있는가**
- **D 거리 = |A + I -1|**
- D가 0이면 컴포넌트가 주계열 위에 있다. 1이면 가장 멀리 떨어져있다.
- 컴포넌트들의 D 거리를 구하여 **0에 가깝지 않거나 혼자 멀리 떨어져 있는 이상한 경우, 재검토 후 재구성**할 수 있다.
- D 지표를 **시간에 따라** 그려볼 수도 있다.
- 해당 컴포넌트가 주계열에서 **벗어난 원인과 지점을 알 수** 있다.

결론

- **의존성 관리 지표**는 설계의 의존성과 추상화 정도보다 **내가 생각한 '훌륭한' 패턴 수준에 얼마나 잘 부합하는지**를 측정
- 하지만 지표도 **임의로 결정된 표준**을 기초로 한 측정값 → **불완전하다.**

아키텍처의 목적

- **시스템의 생명주기를 지원하는** 것
  - 시스템을 쉽게 이해하고, 쉽게 개발하며, 쉽게 유지보수하고, 쉽게 배포하게 해주는 것
- 시스템의 **수명과 관련된 비용은 최소화**하고, **프로그래머의 생산성은 최대화**하는 것

아키텍처의 목적 - 개발

- 팀마다 아키텍처 관련 결정이 다름
- 규모가 작은 팀 - 아키텍처가 있는 게 오히려 진전이 느릴 것. 없이 개발되는 경우가 많음
- 규모가 큰 팀 - 팀별로 컴포넌트를 구분한 아키텍처로 개발할 가능성이 높음

아키텍처의 목적 - 배포

- 시스템 아키텍처는 시스템을 단 한 번에 쉽게 배포할 수 있도록 만드는데 목표
- 개발 초기에 고려하는 것이 좋음

아키텍처의 목적 - 운영

- 대다수 운영에서 온 어려움은 **하드웨어 추가 투입**으로 해결할 수 있다.
  - 그래서 개발,배포,유지보수보다는 미치는 **영향이 덜 극적임**
- 아키텍처는 **시스템을 운영하는 데 필요한 요구**도 알려줌
  - 개발자에게 시스템의 운영 방식을 잘 드러내 준다. → **시스템을 이해하기 쉬워지고 개발과 유지보수에 큰 도움을 줌**

아키텍처의 목적 - 유지보수

- 유지보수의 **가장 큰 비용은 탐사 (spelunking)과 이로 인한 위험부담**에 있다.
  - 탐사
    - 새로운 기능을 추가하거나 결합을 수정할 때 어디를 수정하는 게, 어떤 전략을 쓰는게 가장 좋을 지 결정하는데 쓰는 비용
    - 그리고 이렇게 수정했을 때 의도치 않게 결합이 생길 위험이 존재
- 아키텍처를 신중하게 만들면 이 비용을 크게 줄일 수 있음

선택사항 열어두기

- 소프트웨어는 **선택사항을 가능한 많이, 오랫동안 열어 두어야 부드럽게 유지가 됨**
  - **선택사항이란? - 중요치 않은 세부사항**
    - 입출력 장치, 데이터베이스, 웹 시스템, 서버, 프레임워크, 통신 프로토콜 등등이 있다.
- 아키텍처는 **세부사항이 정책(업무 규칙)에 무관하도록** 구축하는게 목표
  - ex. 고수준의 정책은 어떤 데이터베이스 시스템을 써도 관련이 없도록 만들어야 한다.
  - 마찬가지로 어떤 웹 서버를 쓸 건지, 어떤 인터페이스로 통신할 건지, 어떤 의존성 주입 프레임워크를 쓸 건지 등등에 관련이 없어야 한다
- 이렇게 **무관하게 구현해야 세부사항을 더 오랫동안 열어둘 수**가 있다.