
클린 아키텍처

25장 ~ 29장

23/09/15

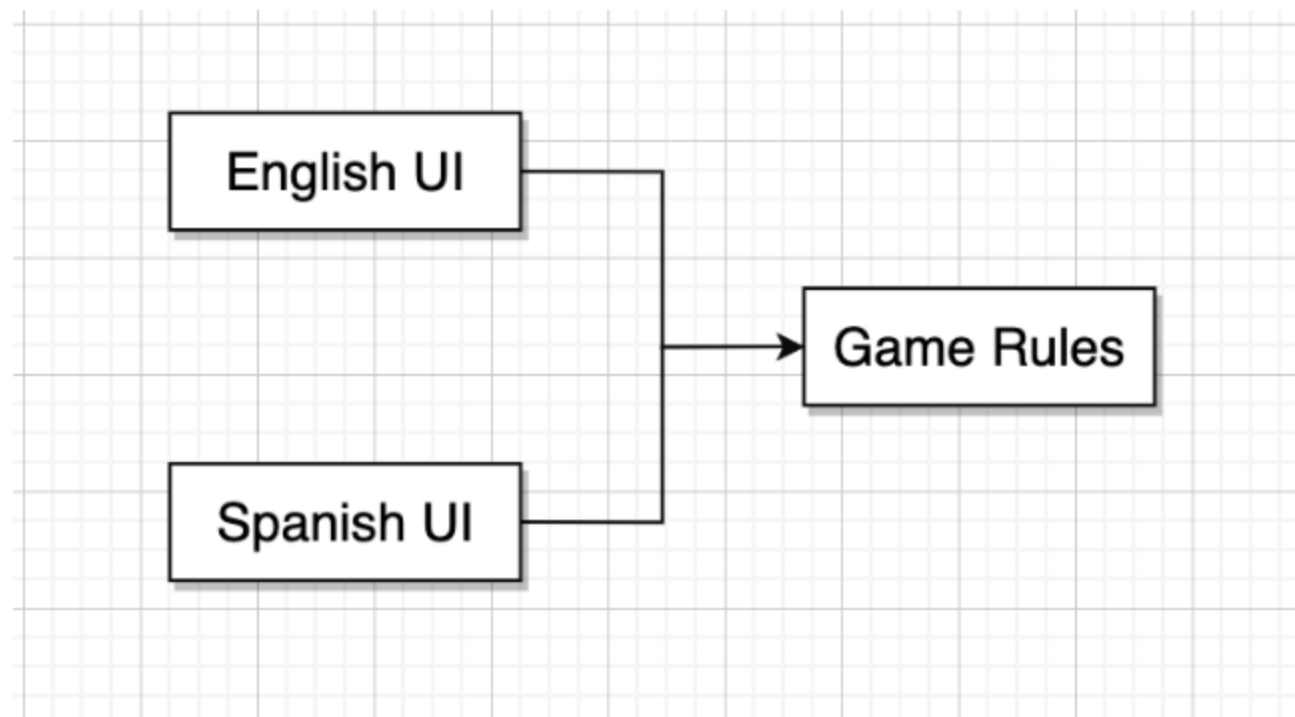
25장

계층과 경계

시스템은 세가지 컴포넌트 (UI, 업무 규칙, 데이터베이스)로 구성되어있다고
생각할 수 있지만, 실제로는 훨씬 많다.

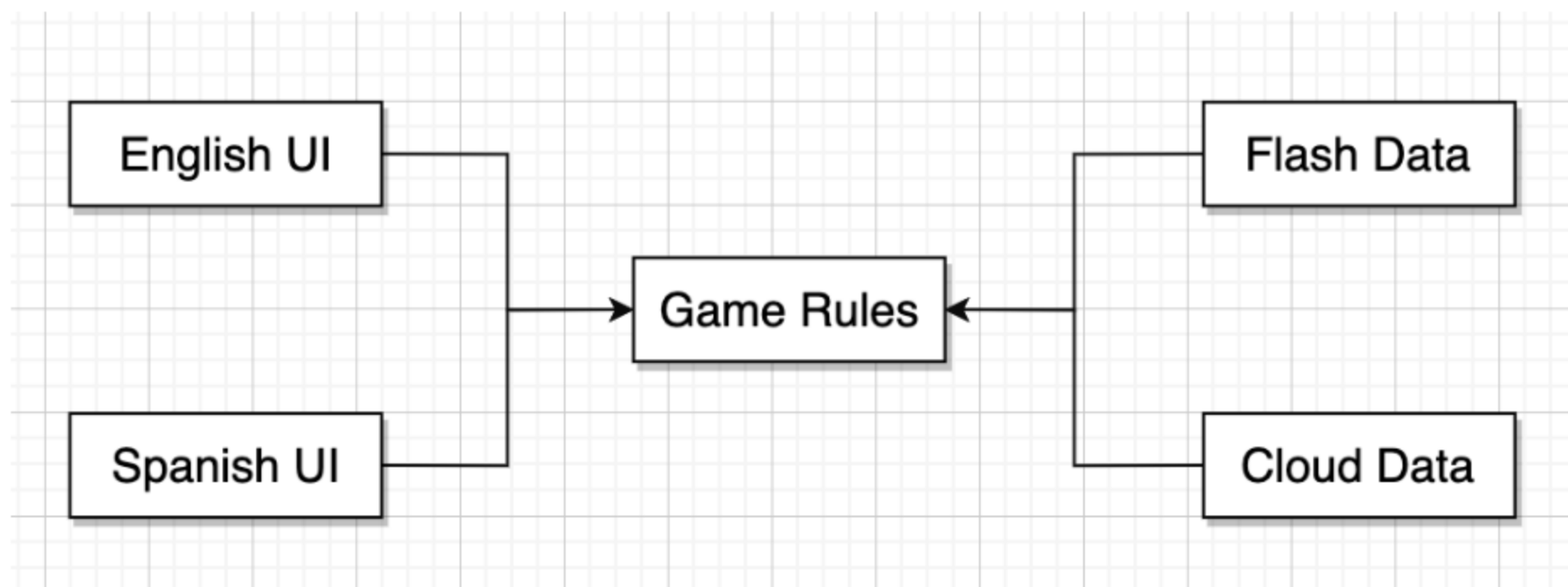
01 움퍼스 사냥 게임

게임 규칙은 게임의 상태를 영속성을 가지는 특정한 데이터 구조로 저장한다.



UI 컴포넌트가 어떤 언어를 사용해도 게임 규칙을 재사용 가능.

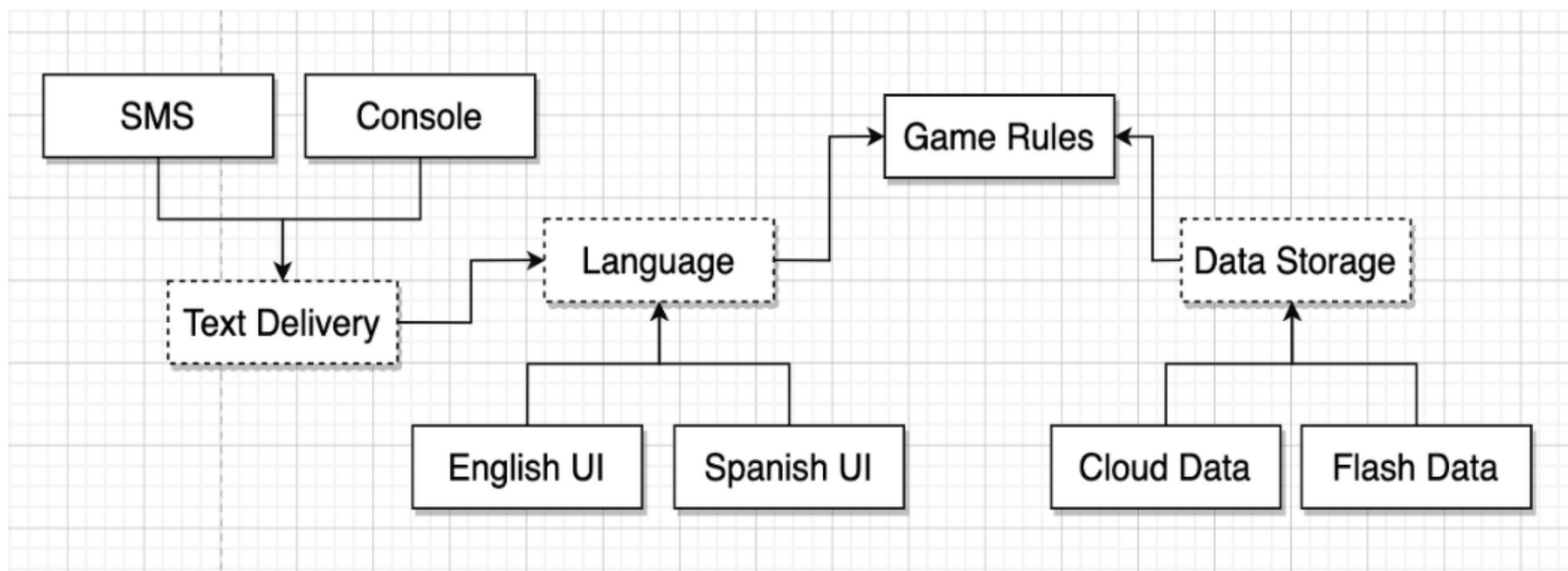
01 움퍼스 사냥 게임



데이터 저장소 또한, 고수준인 게임 규칙을 의존하는 형태이다.

02 클린 아키텍처?

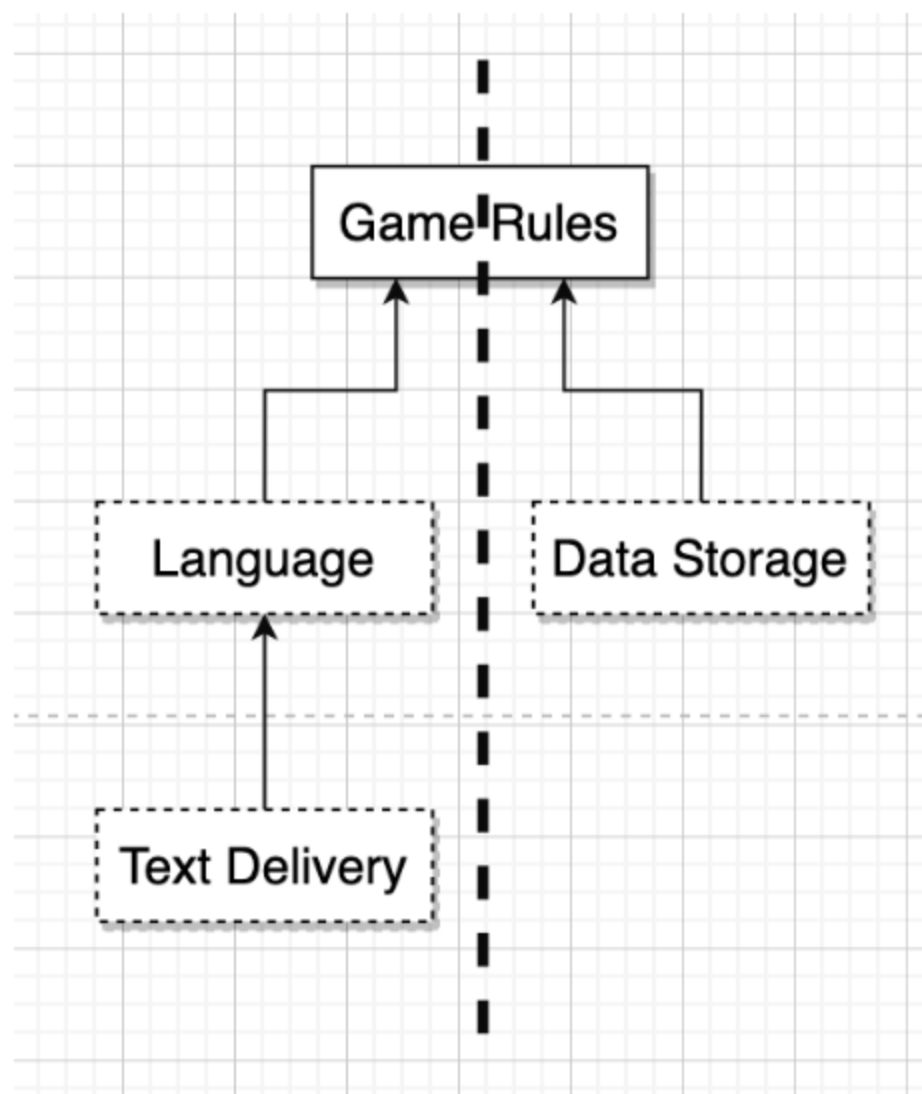
잠재된 아키텍처 경계도 존재할 수 있다. 이를 그림으로 나타내면 다음과 같다.



점선으로 된 테두리 : API를 정의하는 추상 컴포넌트
해당 API는 위나 아래의 컴포넌트가 대신 구현한다.
api는 구현하는 쪽이 아닌, 사용하는 쪽에 정의되고 소속된다.

02 클린 아키텍처?

EX) Game Rules 내부 코드에서 사용하나 Language 내부 코드에서 구현하는
다형적 Boundary interface가 존재한다.



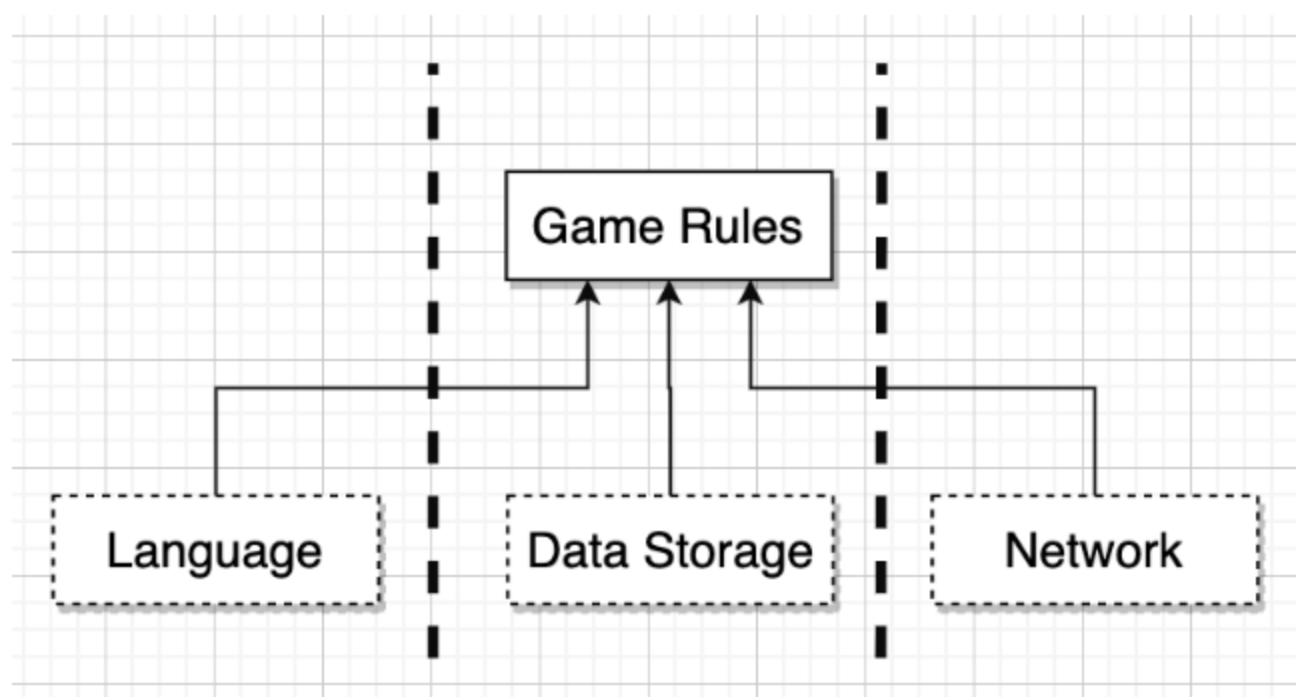
순전히 API 컴포넌트에만 집중하여
다이어그램을 단순화하면 다음과 같다.

GameRules가 최상위 수준 정책의
컴포넌트이므로 가장 위쪽에 위치하고
화살표가 위로 맞춰져있다.

03 흐름 횡단하기

하지만 항상 흐름이 두 가지인것은 아니다.

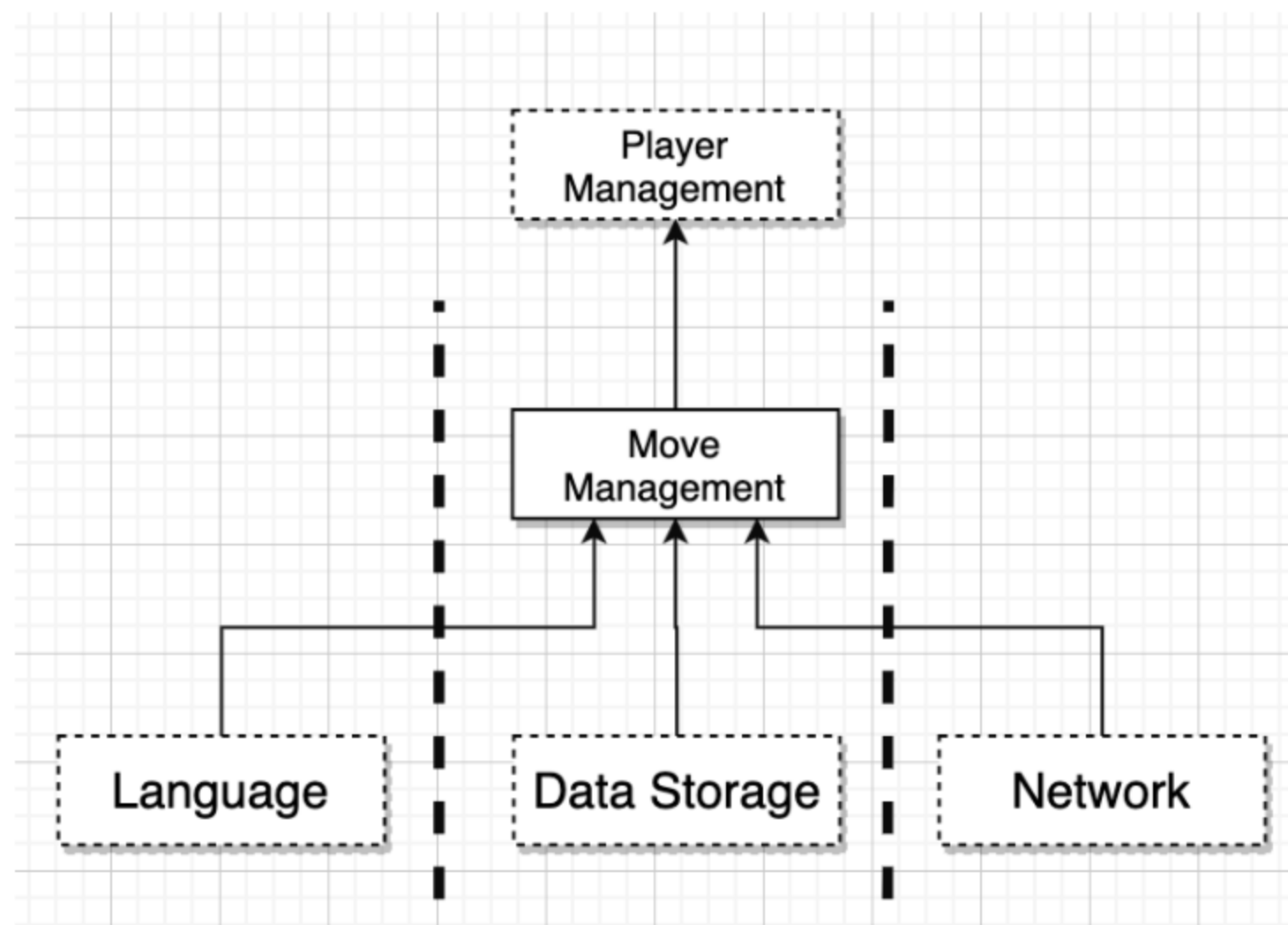
예를들어 해당 게임이 온라인 게임이 된다면,
다음과 같이 네트워크 컴포넌트 또한 추가된다.



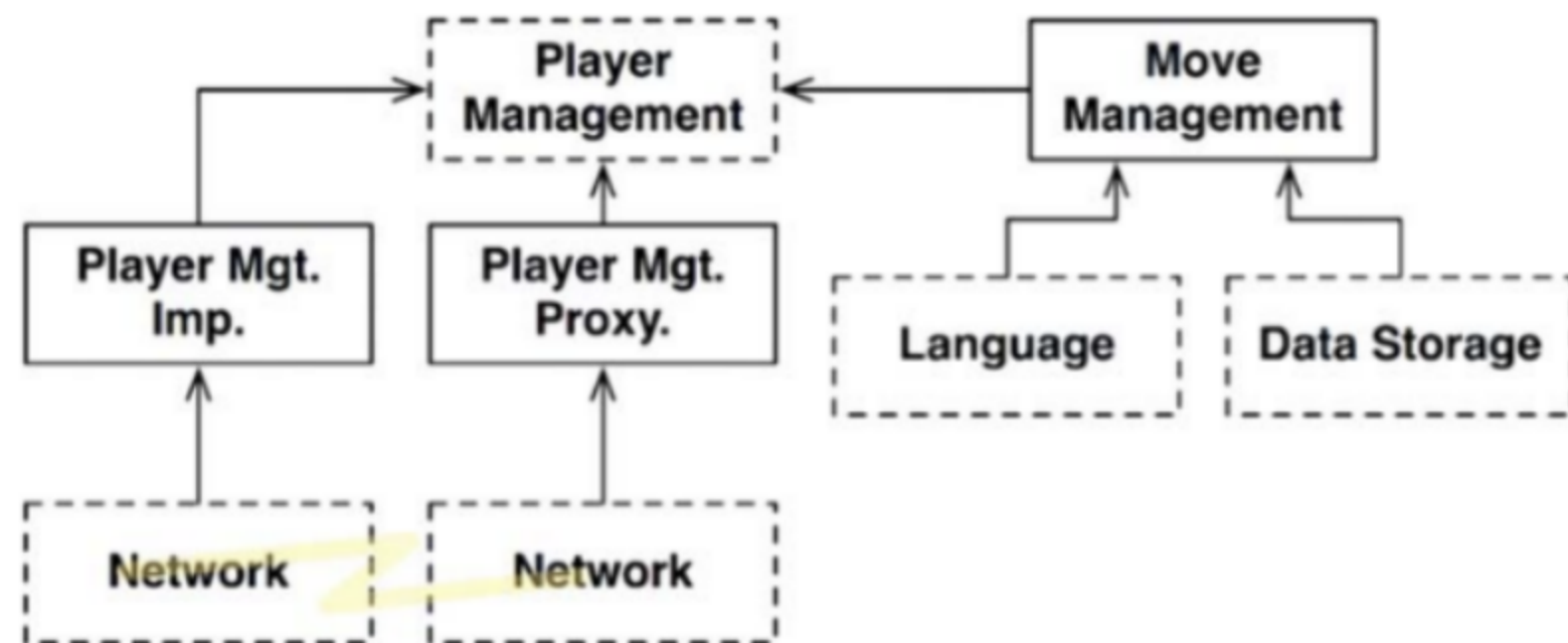
04 흐름 분리하기

게임 규칙중 일부인 지도 관련 메커니즘을 처리하는 MoveManagement를 추가한 경우 지도규칙에 의해 플레이어의 지도 관련 사건을 처리한다.

사건이 발생하면 MoveManagement은 이를 판단한 후
고수준인 PlayerManagement 정책에 알려
플레이어의 상태를 관리하도록 한다.



04 흐름 분리하기



마이크로서비스 API추가.

대규모 플레이어들이 동시에 플레이하는 경우, Player Management는 접속된 모든 Move Management 컴포넌트에 API를 제공한다.

따라서 위와 같이 아키텍처 경계를 만들어 시나리오를 묘사할 수 있다.

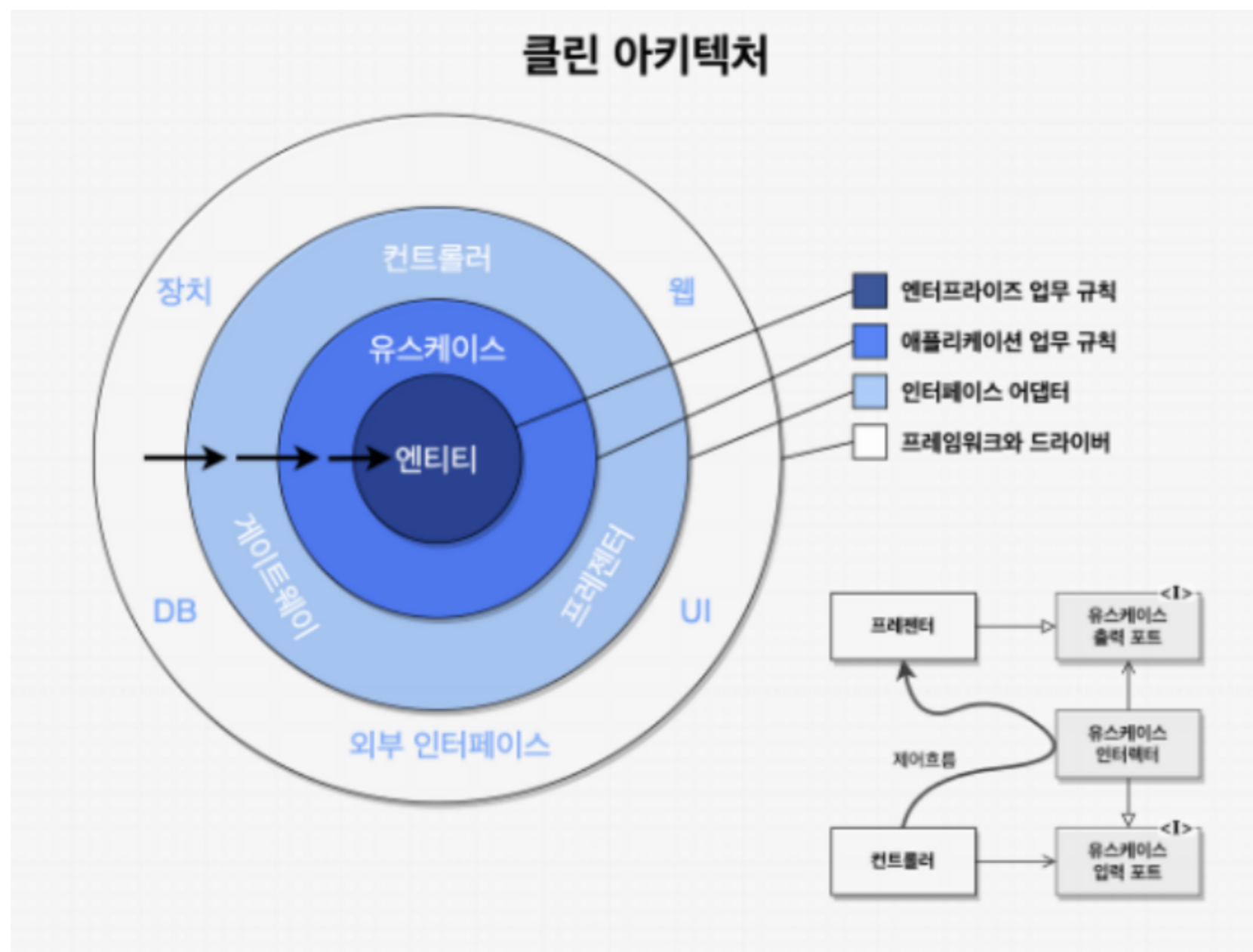
05 결론

- 아키텍처 경계는 어디에나 존재할 수 있다.
- 그러나 추상화를 미리 진행해서도 안되고, 나중에 다시 추가해서도 안된다.
- 능숙한 아키텍트라면 비용을 산정하고 어디에 아키텍처 경계를 두어야 할지 주의를 기울여 결정해야 한다.

26장

메인 컴포넌트

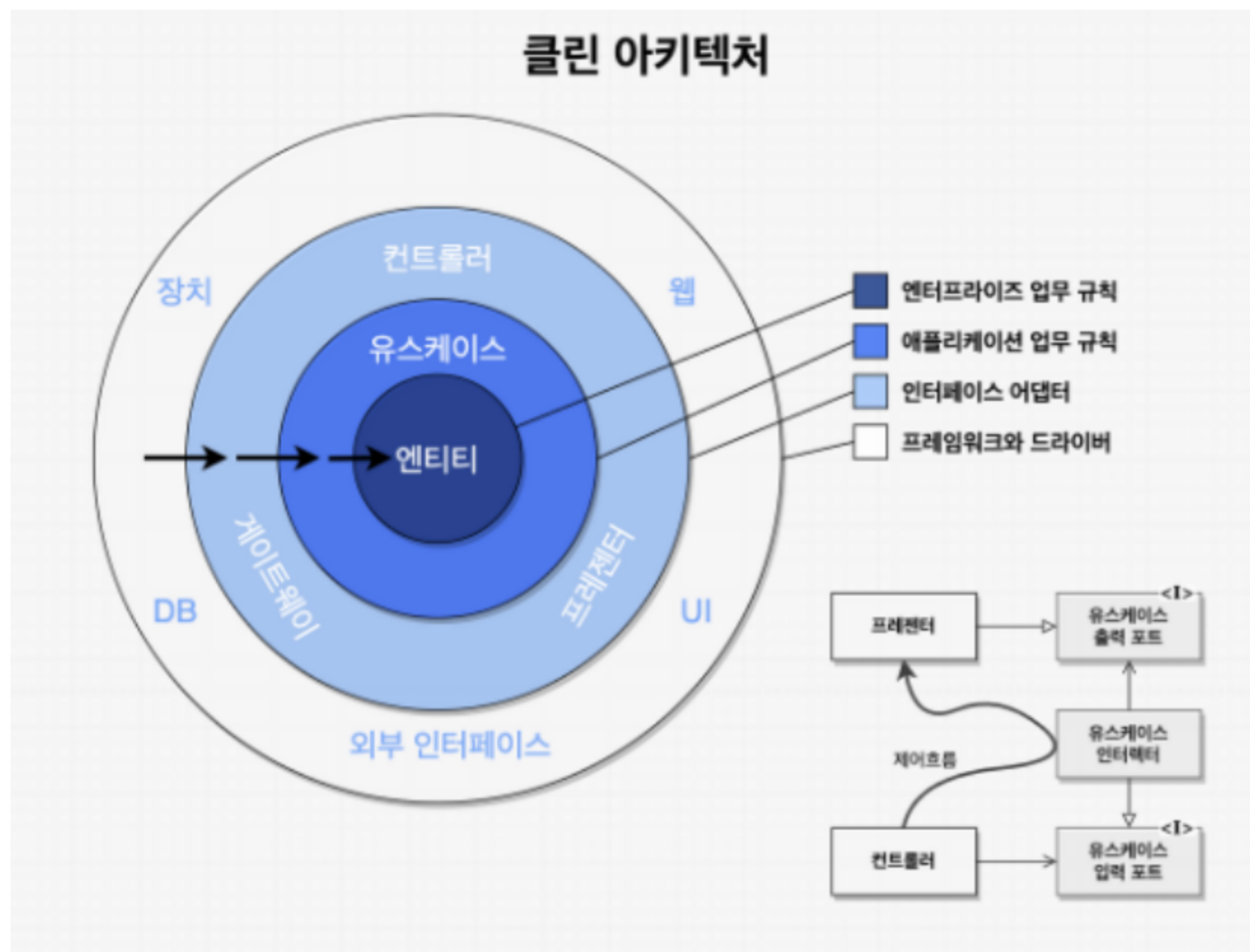
01 궁극적인 세부사항



메인 컴포넌트는 가장 바깥 원에 해당하는 가장 낮은 수준의 정책이다.

의존성 주입 프레임워크를 이용해 의존성을 주입하는 일을 메인 컴포넌트가 한다.

01 궁극적인 세부사항



메인은 고수준의 시스템을 위한 모든것을 로드한 후 제어권을 고수준의 시스템에게 넘긴다.

➡예시인 움퍼스 게임에서도 main 함수에서 게임의 메인 루프, 입력 명령어 해석을 처리하지만 명령어의 실제 처리는 다른 고수준 컴포넌트로 위임시킨다.

02 결론

메인은 애플리케이션의 플러그인이다.
메인은 플러그인이므로 설정별로 하나씩 두어
둘 이상의 메인 컴포넌트를 만들 수도 있다.

ex) 개발용 메인 플러그인, 별도의 테스트용 메인 플러그인 등등

27장

'크고 작은 모든' 서비스들

01 '크고 작은 모든' 서비스들

서비스 지향 아키텍처와 마이크로서비스 아키텍처가
최근 큰 인기를 끄는 이유

 **NOTE** 서비스를 사용하면 상호결합이 철저하게 분리되어 있고, 개발과 배포 독립성을 지원하는 것처럼 보인다.

~~하지만 일부만 맞는 말이다.~~

02 서비스 아키텍처?

프로세스나 플랫폼 경계를 가로지르는 함수 호출에 지나지 않는 '서비스'라는 것은 아키텍처 관점에서 아키텍처적으로 중요한 서비스도 있지만, 중요하지 않은 서비스도 존재한다.

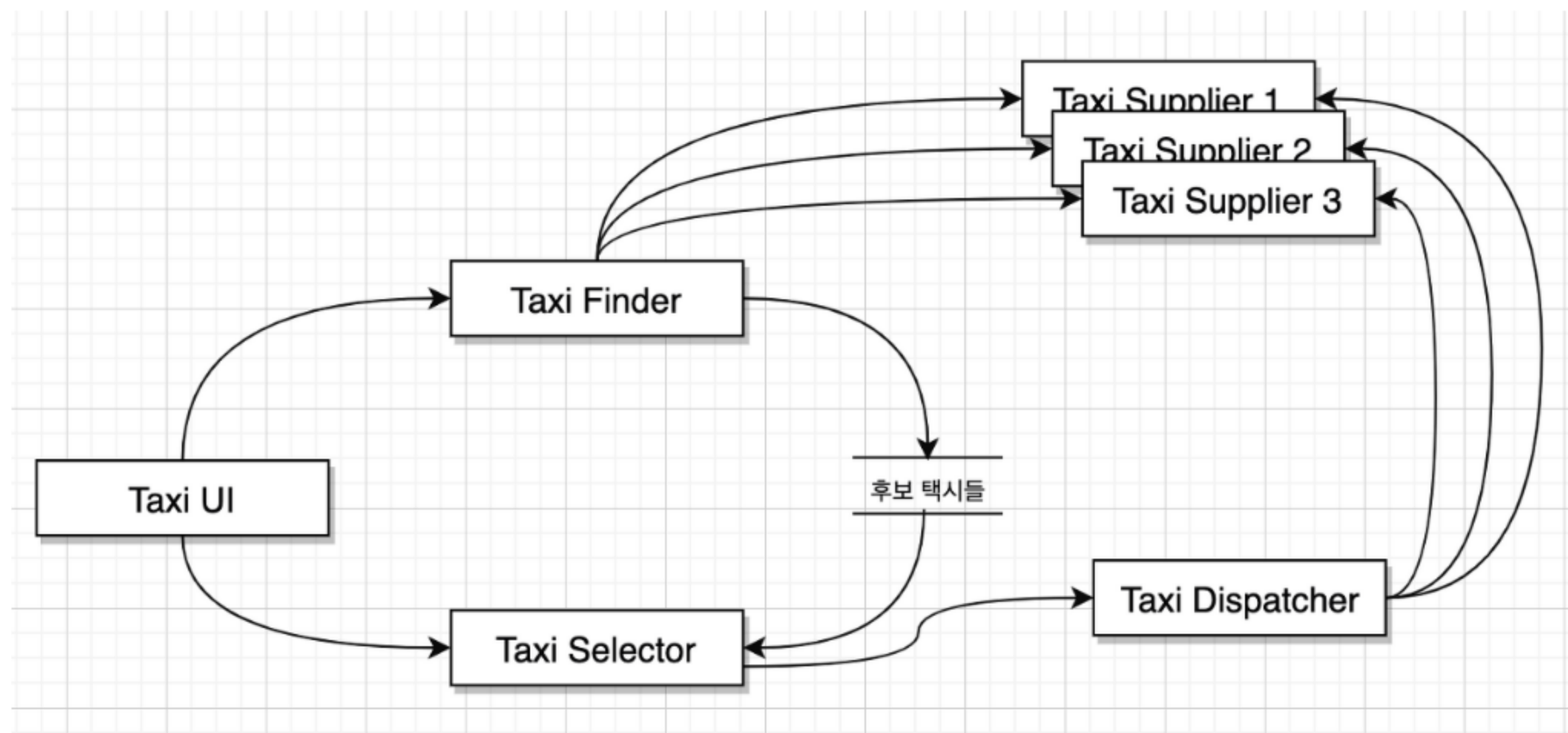
03 서비스의 이점?

- 결합 분리의 오류
 - 시스템을 서비스로부터 분리함으로서 얻는 이점? ➡ 서비스 사이의 결합이 확실히 분리된다.
 - 하지만 개별 변수 수준에서는 결합이 분리되어도, 프로세서 내의 또는 네트워크 상의 공유 데이터에 의해 더욱 강력하게 결합되어 버린다.

03 서비스의 이점?

- 개발 및 배포 독립성의 오류
 - 데브옵스 관점에서 전담팀이 각 서비스를 작성하고 유지보수하며 운영하는 책임을 질 수 있다.
 - 대규모 엔터프라이즈 시스템은 서비스 기반 시스템 외에도, 모노리틱, 컴포넌트 기반 시스템으로도 구축 가능하다. 즉, 서비스가 유일한 선택지가 아니다.
 - 서비스라고 해서 항상 독립적으로 개발, 배포, 운영할 수 있는 것 또한 아니다.

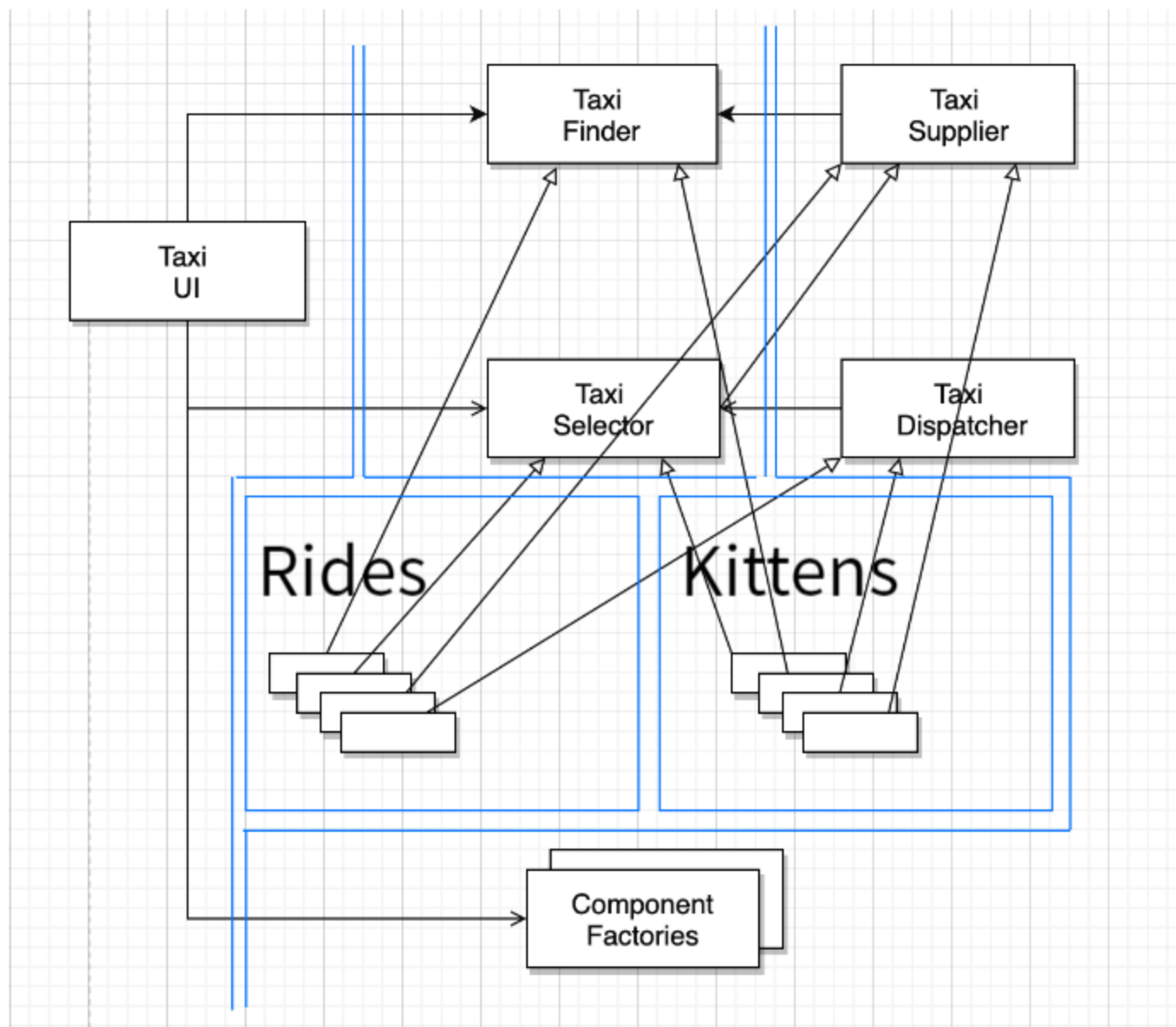
04 야옹이 문제



서비스가 모두 결합되어있어 위에서 묘사된 것과 같은 종류의 기능적 분해는 새로운 기능이 기능적 행위를 횡단하는 상황에 매우 취약하다.

➡서비스 지향이든 아니든, 모든 소프트웨어 시스템이 직면하게 될 횡단 관심사가 지닌 문제.

05 객체가 구출하다

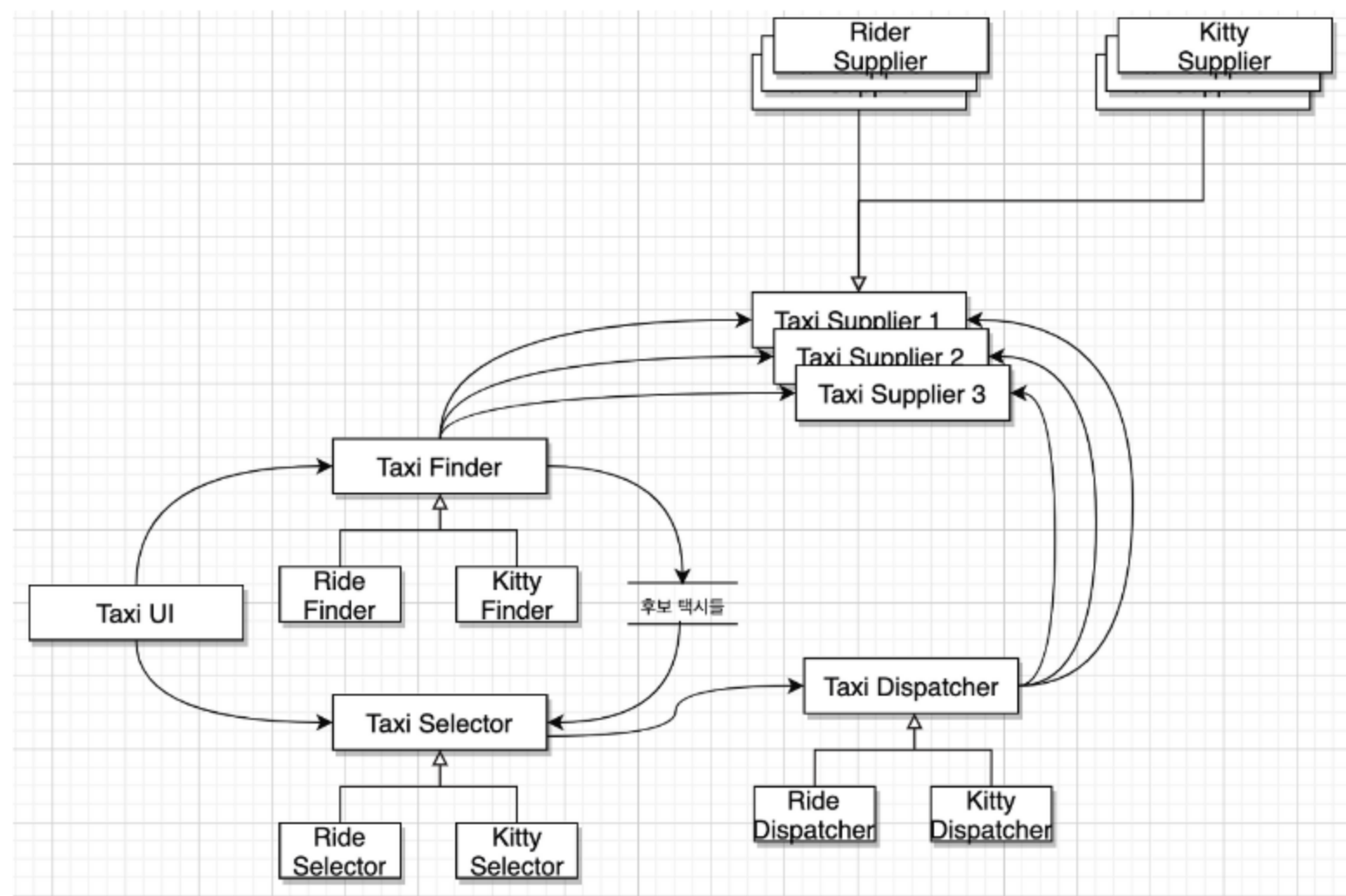


컴포넌트 기반 아키텍처에서 SOLID 설계 원칙대로 설계했다면, 위와 같이 배차에 특화된 로직은 Rides 컴포넌트로 추출되고, 야옹이 신규 기능은 Kittens 컴포넌트에 들어간다.

두 신규 컴포넌트는 의존성 규칙을 준수하고, 다른 컴포넌트들은 변경이 필요하지 않아서, 야옹이 기능은 결합이 분리되며 독립적으로 개발 배포가 가능하다.

06 컴포넌트 기반 서비스

서비스또한 SOLID 원칙대로 설계할 수 있고,
컴포넌트 구조를 갖출 수 있어
기존 컴포넌트를 변경하지 않고도
새로운 컴포넌트를 추가할 수 있다.



각 서비스의 내부가 자신만의 컴포넌트 설계로 되어 있어
파생 클래스를 만드는 방식으로 신규 기능 추가 가능하다.

06 횡단 관심사

아키텍처 경계는 서비스 사이에 있지 않고, 서비스를 컴포넌트 단위로 분할한다.
아키텍처 경계를 정의하는것은 서비스 내에 위치한 컴포넌트이다.

결론

서비스는 아키텍처적으로 그리 중요한 요소는 아니다.

28장

테스트 경계

01 테스트 경계

테스트에는 굉장히 여러 종류가 있지만, 아키텍처 관점에서는 모든 테스트가 동일하다.
테스트는 태생적으로 테스트 대상이 되는 코드를 의존한다.
테스트는 시스템 컴포넌트 중 가장 고립되어 있고, 운영에 꼭 필요치는 않다.

그럼에도, 테스트가 시스템 컴포넌트가 아니라는 뜻은 아니다.

02 테스트를 고려한 설계

테스트는 대체로 배포하지 않는다고 생각하고, 개발자는 종종 테스트가 시스템의 설계 범위 밖에 있다고 여김
➡ 굉장히 치명적인 관점이다.

테스트가 시스템의 설계와 통합되지 않으면, 테스트는 깨지기 쉬워지고 시스템은 변경하기 어려워진다.

시스템에 강하게 결합된 테스트라면, 시스템의 공통 컴포넌트가 변경되면 수백, 수천개의 테스트가 망가진다. 이를 두고 깨지기 쉬운 테스트 문제라고 한다.

02 소프트웨어 설계의 첫 번째 규칙!



Tip

변동성이 있는 것에 의존하지 말라.

예를 들어 GUI에 의존하는 테스트라면, 깨지기 쉽다.
GUI는 변동성이 매우 크기 때문이다.

따라서 시스템과 테스트를 설계할 때, 이를 잘 고려해서 설계해야 한다.

03 테스트 API


위의 목표를 달성하려면

- 값비싼 자원(DB)은 건너뛰고
- 보안 제약사항은 무시하며
- 모든 업무 규칙을 검증할 수 있는 API를 만들면된다.

테스트 API는 테스트를 애플리케이션으로부터 분리할 목적으로 사용된다.

03 테스트 API - 구조적 결합

구조적 결합은 테스트 결합 중에서
가장 강하며 상용 클래스나 메서드 중
하나라도 변경되면 다수의 테스트가 변경되어야 한다.

 **NOTE** 상용 코드란? 수정하지 않거나 최소한의 수정만을 거쳐 여러 곳에 필수적으로 사용되는 코드를 말한다.

테스트 API의 역할은 애플리케이션의 구조를 테스트로부터 숨겨
상용 코드를 리팩터링 하거나 진화시키더라도
테스트에는 전혀 영향을 주지 않는것에 있다. (반대도 마찬가지이다.)

구조적 결합을 약하게하여

- 1.시간이 지날수록 테스트는 더 구체적이고 특화된형태로 변화되어야한다.
- 2.상용 코드는 더 추상적이고 범용적인 형태로 변화해야함.

03 테스트 API - 보안

하지만 보안적인 측면에서는 테스트 API 자체와 테스트 API 중 위험한 부분의 구현부는 독립적인 컴포넌트로 분리해야 한다.

04 결론

테스트는 시스템의 일부다.

테스트를 시스템의 일부로 설계하지 않으면,
테스트는 깨지기 쉽고 유지보수하기 어려워지는 경향이 있다.
테스트가 유지보수가 어렵다면 결국 버려지는 최후를 맞게 된다.