제10장

겹겹이 쌓인 건축 양식

계층형 아키텍처 스타일(n계층형이라고도 함)은 가장 일반적인 아키텍처 스타일 중 하나입니다. 단순성, 친숙성 및 낮은 비용 덕분에 많은 레거시 애플리케이션에서 사실상의 표준으로 자리 잡았습니다.

계층형 아키텍처 스타일은 '암시적 아키텍처' 와 ' 우연한 아키텍처' 를 포함한 여러 아키텍처 안티패턴에 해당될 수 있습니다 . 개발자나 아키텍트가 어떤 아키텍처 스타일을 사용하고 있는지 확신하지 못한 채 "코딩을 막 시작할 때"에는 계층형 아키텍처 스타일을 구현하고 있을 가능성이 높습니다.

위상수학

계층형 아키텍처 스타일 내의 구성 요소는 그림 10-1에서 보는 것처럼 논리적 수평 계층으로 구성되며, 각 계층은 애플리케이션 내에서 특정 역할(예: 프레젠테이션 로직 또는 비즈니스 로직)을 수행합니다. 계층의 수와 유형에 대한 구체적인 제한은 없지만, 대부분의 계층형 아키텍처는 프레젠테이션, 비즈니스, 영속성 및 데이터베이스의 네 가지 표준 계층으로 구성됩니다. 일부 아키텍처는 비즈니스 계층과 영속성 계층을 결합하기도 하는데, 특히 영속성 로직(예: SQL 또는 HSQL)이 비즈니스 계층 구성 요소에 포함된 경우 그렇습니다. 소규모 애플리케이션은 세 개의 계층만 가질 수 있지만, 규모가 크고 복잡한 비즈니스 애플리케이션은 다섯 개 이상의 계층을 포함할 수 있습니다.

더.

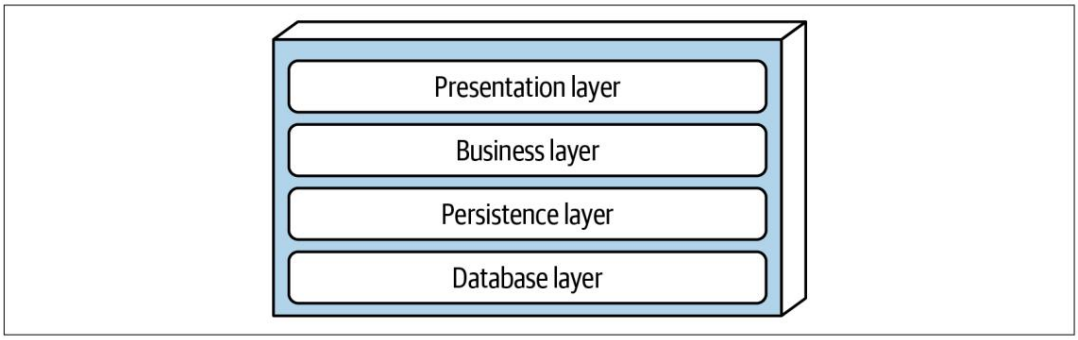

그림 10-1. 계층형 아키텍처 스타일 내의 표준 논리 계층

그림 10-2는 물리적 계층 구조(배포 방식) 관점에서 토폴로지 변형을 보여줍니다. 첫 번째 변형은 프레젠테이션, 비즈니스 및 영구 저장 계층을 단일 배포 단위로 결합하고, 데이터베이스 계층은 일반적으로 별도의 외부 물리적 데이터베이스(또는 파일 시스템)로 표현됩니다. 두 번째 변형은 프레젠테이션 계층을 별도의 배포 단위로 물리적으로 분리하고, 비즈니스 및 영구 저장 계층을 두 번째 배포 단위로 결합합니다. 이 변형에서도 데이터베이스 계층은 일반적으로 외부 데이터베이스 또는 파일 시스템을 통해 물리적으로 분리됩니다. 세 번째 변형은 데이터베이스 계층을 포함한 네 가지 표준 계층을 모두 단일 배포 단위로 결합합니다. 이 변형은 모바일 장치 애플리케이션과 같이 내부에 내장된 데이터베이스 또는 인메모리 데이터베이스를 사용하는 소규모 애플리케이션에 유용할 수 있습니다. 많은 온프레미스(온프레미스) 제품이 이 세 번째 변형을 사용하여 구축되고 고객에게 제공됩니다.

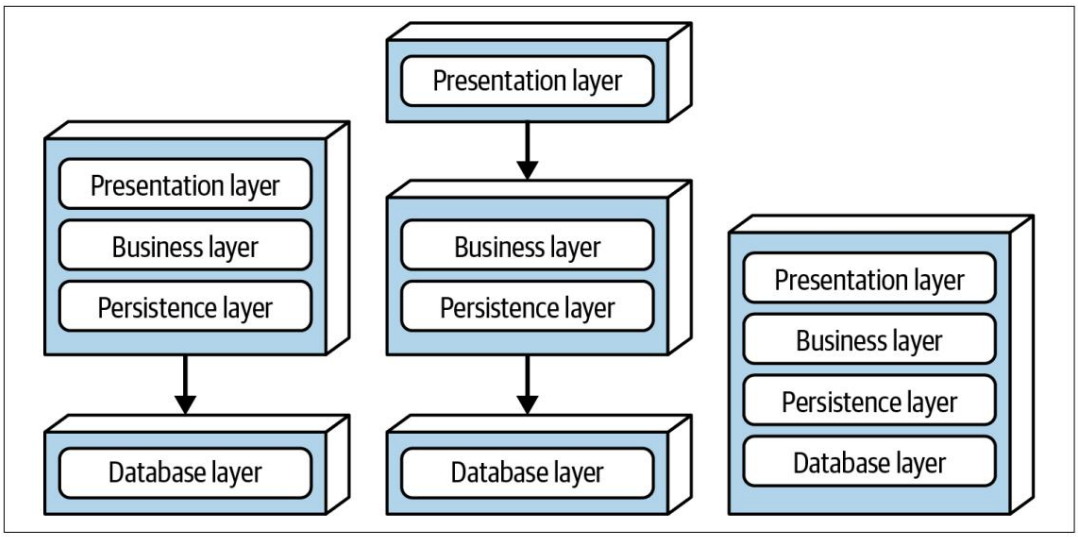

그림 10-2. 물리적 토폴로지(배포) 변형

각 계층은 특정한 역할과 책임을 가지며, 특정 비즈니스 요청을 충족하기 위해 필요한 작업에 대한 추상화를 제공합니다. 예를 들어, 프레젠테이션 계층은 모든 UI 및 브라우저 통신 로직을 처리하는 반면, 비즈니스 계층은 요청과 관련된 특정 비즈니스 규칙을 실행하는 역할을 합니다. 프레젠테이션 계층은 고객 데이터를 가져오는 방법을 알 필요도 없고, 특정 형식으로 화면에 정보를 표시하기만 하면 됩니다. 마찬가지로, 비즈니스 계층은 고객 데이터를 화면에 표시하기 위한 형식을 지정하는 방법이나 데이터의 출처에 대해 신경 쓸 필요가 없습니다. 영구 저장 계층에서 데이터를 가져와 비즈니스 로직(예: 값 계산 또는 데이터 집계)을 수행한 후, 해당 정보를 프레젠테이션 계층으로 전달하기만 하면 됩니다.

계층형 아키텍처 스타일에서처럼 관심사를 분리하면 효과적인 역할 및 책임 모델을 쉽게 구축할 수 있습니다. 특정 계층 내의 구성 요소는 해당 계층과 관련된 로직만 처리하도록 범위가 제한됩니다. 예를 들어, 프레젠테이션 계층의 구성 요소는 프레젠테이션 로직만 처리하고, 비즈니스 계층의 구성 요소는 비즈니스 로직만 처리합니다. 이를 통해 개발자는 자신의 전문 기술을 활용하여 도메인의 기술적 측면(예: 프레젠테이션 로직 또는 영속성 로직)에 집중할 수 있습니다. 하지만 이러한 장점의 단점은 시스템 전체의 민첩성(변화에 신속하게 대응하는 능력)이 부족하다는 것입니다.

계층형 아키텍처는 기술적으로 분할된 아키텍처입니다(도메인 분할 아키텍처와는 반대되는 개념). 즉, 9 장 에서 배웠 듯이 구성 요소들이 도메인(예: 고객)이 아닌 아키텍처 내에서의 기술적 역할(예: 프레젠테이션 또는 비즈니스)에 따라 분리됩니다. 결과적으로 특정 비즈니스 도메인은 아키텍처의 모든 계층에 걸쳐 분산됩니다. 예를 들어, "고객" 도메인은 프레젠테이션 계층, 비즈니스 계층, 규칙 계층, 서비스 계층 및 데이터베이스 계층에 모두 포함되어 있어 해당 도메인에 변경 사항을 적용하기가 어렵습니다. 따라서 DDD(도메인 주도 개발) 접근 방식은 계층형 아키텍처 스타일에 특히 적합하지 않습니다.

스타일 사양

이러한 아키텍처 스타일의 계층 구조는 특정 기술적 책임 영역을 포괄하지만, 계층 자체는 다른 특성을 나타낼 수도 있습니다.

격리 계층 각 계층은 닫

혀 있거나 열려 있을 수 있습니다. 계층이 닫혀 있는 경우, 요청이 최상위 계층에서 최하위 계층으로 이동할 때 어떤 계층도 건너뛸 수 없습니다.
다음 레이어로 이동하려면 바로 아래 레이어를 통과해야 합니다(그림 10-3 참조). 예를 들어, 모든 레이어가 닫혀 있는 아키텍처에서 요청은

프레젠테이션 계층에서 시작된 데이터는 먼저 비즈니스 계층을 거쳐 영구 저장 계층으로 이동한 후 최종적으로 데이터베이스 계층에 도달해야 합니다.

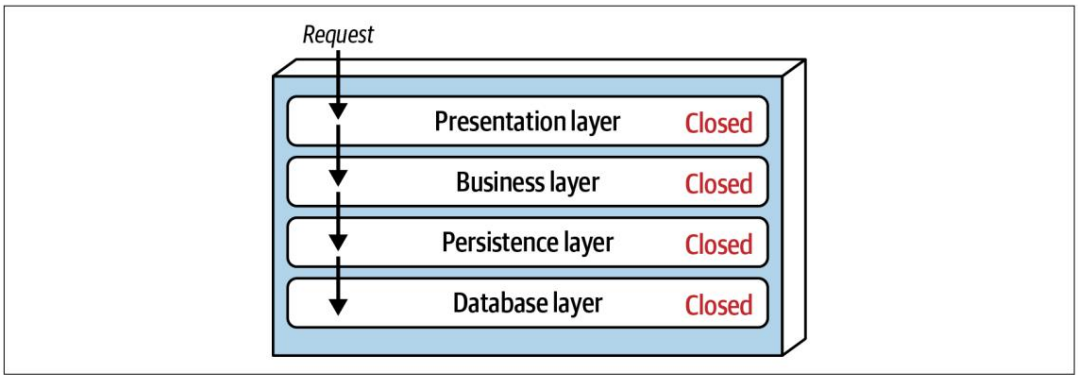

그림 10-3. 계층형 구조 내의 닫힌 층

그림 10-3 에서 볼 수 있듯이, 프레젠테이션 계층이 간단한 데이터 검색 요청에 대해 불필요한 계층을 거치지 않고 데이터베이스에 직접 접근하는 것이 훨씬 빠르고 간편합니다(2000년대 초반에 '패스트 레인 리더(Fast-Lane Reader)' 패턴으로 알려졌던 방식). 이를 위해서는 비즈니스 계층과 영속성 계층이 개방되어 있어야 하며, 요청이 다른 계층을 우회할 수 있어야 합니다. 그렇다면 개방형 계층이 좋을까요, 아니면 폐쇄형 계층이 좋을까요? 이 질문에 대한 해답은 '격리 계층(Layer of Isolation)'이라는 핵심 개념에 있습니다.

계층 격리 개념은 아키텍처의 한 계층에서 변경 사항이 발생하더라도 해당 계층 간의 계약이 변경되지 않는 한 다른 계층의 구성 요소에는 일반적으로 영향을 미치지 않는다는 것을 의미합니다. 각 계층은 다른 계층과 독립적이며, 다른 계층의 내부 작동 방식에 대해 거의 또는 전혀 알지 못합니다. 그러나 계층 격리를 제대로 구현하려면 요청의 주요 흐름과 관련된 계층은 폐쇄적이어야 합니다. 만약 프레젠테이션 계층이 퍼시스턴스 계층에 직접 접근할 수 있다면, 퍼시스턴스 계층의 변경 사항은 비즈니스 계층과 프레젠테이션 계층 모두에 영향을 미쳐 구성 요소 간의 계층적 상호 의존성이 매우 높은 결합된 애플리케이션이 됩니다. 이는 계층형 아키텍처를 매우 취약하게 만들 뿐만 아니라 변경을 어렵게 하고 비용을 증가시킵니다.

격리 계층을 사용하면 아키텍처의 어떤 계층이든 다른 계층에 영향을 주지 않고 교체할 수 있습니다(물론, 잘 정의된 계약과 비즈니스 델리게이트 패턴을 사용한다는 가정 하에).1 예를 들어, 프레젠테이션 계층 내에서 격리 계층을 활용하여 기존 UI 프레임워크를 새로운 프레임워크로 교체할 수 있습니다.

1. 비즈니스 델리게이트는 비즈니스 서비스와 사용자 인터페이스 간의 결합도를 줄이기 위해 설계된 패턴입니다. 비즈니스 델리게이트는 프레젠테이션 계층에서 비즈니스 객체를 호출하기 위한 어댑터 역할을 합니다.

계층 추가하기 닫

힌 계층은 계층적 격리를 제공하여 변경 사항을 격리하는 데 도움이 되지만, 특정 계층을 개방하는 것이 타당한 경우도 있습니다. 예를 들어, 계층형 아키텍처의 비즈니스 계층에 비즈니스 구성 요소에 공통 기능을 포함하는 공유 객체(예: 날짜 및 문자열 유틸리티 클래스, 감사 클래스, 로깅 클래스 등)가 있다고 가정해 보겠습니다. 프레젠테이션 계층에서 이러한 공유 비즈니스 객체를 사용하지 못하도록 아키텍처를 제한하기로 결정했습니다. 이 제약 조건은 그림 10-4 에 나와 있으며, 점선은 프레젠테이션 구성 요소에서 비즈니스 계층의 공유 비즈니스 객체로 연결됩니다. 이 시나리오는 아키텍처적으로 프레젠테이션 계층이 비즈니스 계층, 즉 해당 계층 내의 공유 객체에 접근할 수 있기 때문에 관리 및 제어가 어렵습니다.

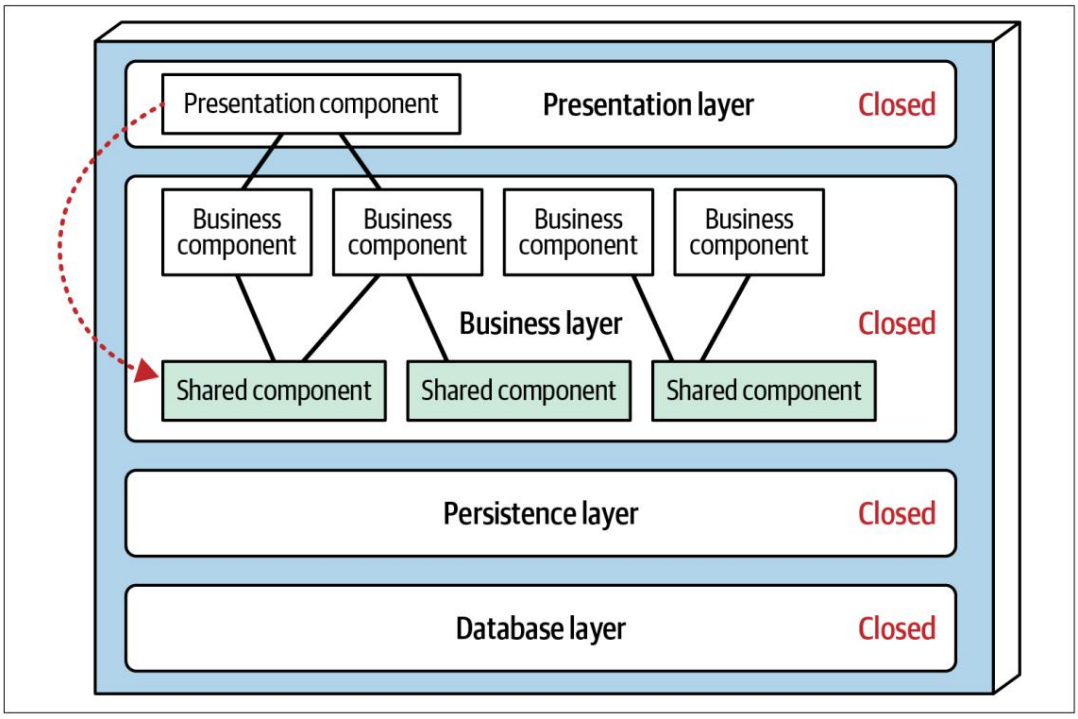

그림 10-4. 비즈니스 계층 내의 공유 객체

이러한 제한을 아키텍처적으로 강제하는 한 가지 방법은 모든 공유 비즈니스 객체를 포함하는 새로운 서비스 계층을 추가하는 것입니다(그림 10-5 참조). 이 새로운 계층을 추가하면 비즈니스 계층이 폐쇄형이므로 프레젠테이션 계층이 공유 비즈니스 객체에 접근하는 것이 아키텍처적으로 제한됩니다. 그러나 새 서비스 계층을 개방형으로 표시해야 합니다. 그렇지 않으면 비즈니스 계층은 서비스 계층을 거쳐야만 영속성 계층에 접근할 수 있습니다. 서비스 계층을 개방형으로 표시하면 비즈니스 계층은 해당 계층에 직접 접근하거나(실선 화살표 참조) 해당 계층을 건너뛰고 바로 다음 계층으로 이동할 수 있습니다(점선 화살표 참조).

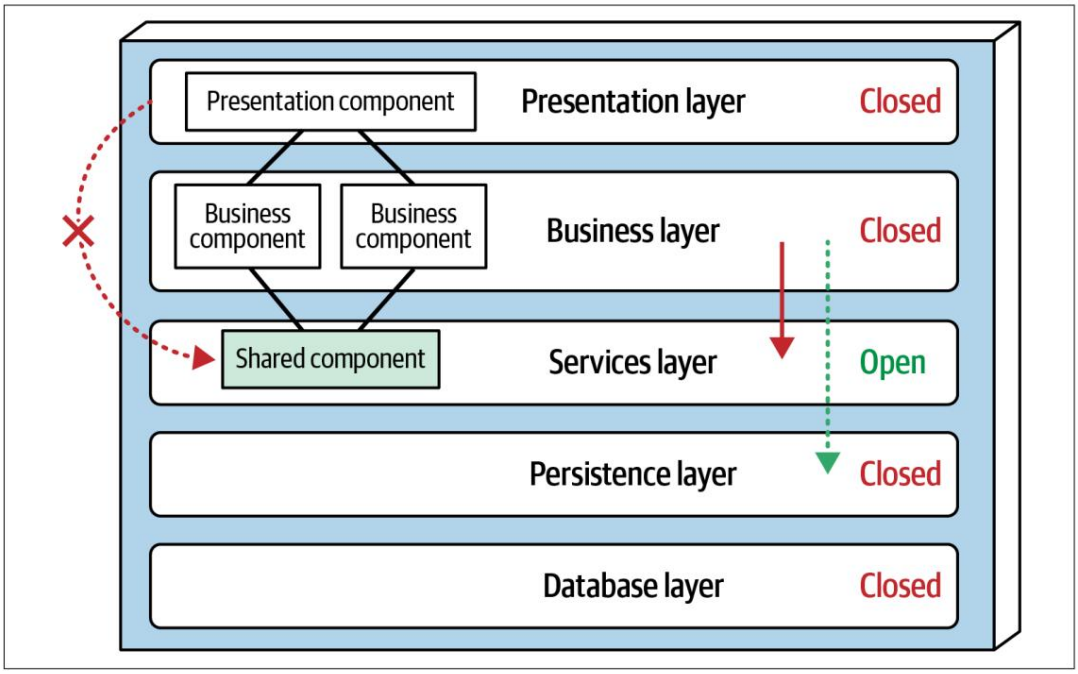

그림 10-5. 아키텍처에 새로운 서비스 계층 추가

개방형 및 폐쇄형 레이어 개념을 활용하면 아키텍처 레이어와 요청 흐름 간의 관계를 명확히 정의할 수 있습니다. 또한 개발자에게 아키텍처 내 레이어 접근 제한 사항을 이해하는 데 필요한 정보와 지침을 제공합니다. 아키텍처에서 어떤 레이어가 개방형이고 어떤 레이어가 폐쇄형인지(그리고 그 이유는 무엇인지) 문서화하거나 제대로 전달하지 않으면, 테스트, 유지 관리 및 배포가 매우 어려운, 레이어 간 결합도가 높고 취약한 아키텍처가 만들어지는 경우가 많습니다.

모든 계층형 아키텍처에는 아키텍처 싱크홀 안티패턴에 해당하는 시나리오가 적어도 몇 가지는 존재합니다. 이 안티패턴은 요청이 비즈니스 로직 없이 단순히 계층에서 계층으로 전달될 때 발생합니다. 예를 들어, 프레젠테이션 계층이 사용자의 간단한 고객 데이터(이름, 주소 등) 검색 요청에 응답한다고 가정해 보겠습니다. 프레젠테이션 계층은 요청을 비즈니스 계층으로 전달하고, 비즈니스 계층은 아무런 처리도 하지 않고 요청을 규칙 계층으로, 규칙 계층은 다시 아무런 처리도 하지 않고 요청을 영속성 계층으로 전달합니다. 영속성 계층은 데이터베이스 계층에 간단한 SQL 쿼리를 실행하여 고객 데이터를 검색합니다. 데이터는 집계, 계산, 규칙 적용, 변환 등의 추가 처리나 로직 없이 스택의 맨 위로 그대로 전달됩니다. 이로 인해 불필요한 객체 생성과 처리가 발생하여 메모리 사용량과 성능이 저하됩니다.

이러한 안티패턴이 작용하는지 여부를 판단하는 핵심은 해당 범주에 속하는 요청의 비율을 분석하는 것입니다. 일반적으로 80-20 법칙을 따르는 것이 좋습니다. 예를 들어, 요청의 20%만 싱크홀인 경우 허용 가능합니다.

하지만 80% 정도라면, 계층형 아키텍처가 해당 문제 영역에 적합한 아키텍처 스타일이 아니라는 좋은 지표가 될 수 있습니다. 아키텍처 싱크홀 안티패턴을 해결하는 또 다른 방법은 아키텍처의 모든 계층을 개방형으로 만드는 것입니다. 물론 이 경우 변경 관리가 더 어려워진다는 단점이 있습니다.

데이터 토폴로지

전통적으로 계층형 아키텍처는 단일 모놀리식 데이터베이스와 함께 모놀리식 시스템을 구성합니다. 공통적인 영속성 계층은 객체 지향 언어에서 선호하는 객체 계층 구조와 관계형 데이터베이스의 집합 기반 영역 간의 매핑에 자주 사용됩니다.

클라우드 고려 사항

계층형 아키텍처는 일반적으로 단일 구조이며 계층으로 분할되어 있기 때문에 클라우드 옵션은 클라우드 제공업체를 통해 하나 이상의 계층을 배포하는 것으로 제한됩니다.
이 아키텍처에 내재된 기술적 분할은 클라우드를 통한 분산 배포에 적합합니다. 그러나 워크플로가 일반적으로 이 아키텍처의 대부분 계층을 거치기 때문에 온프레미스 서버와 클라우드 간의 통신 지연으로 인해 문제가 발생할 수 있습니다.

일반적인 위험

계층형 아키텍처는 단일체 배포 방식과 아키텍처 모듈성 부족으로 인해 내결함성을 지원하지 않습니다. 계층형 아키텍처의 작은 부분에서 메모리 부족 오류가 발생하면 전체 애플리케이션이 다운됩니다.

대부분의 모놀리식 애플리케이션은 평균 복구 시간(MTTR)이 길어 전반적인 가용성에 영향을 미칩니다. 시작 시간은 소규모 애플리케이션의 경우 2분에서 대부분의 대규모 애플리케이션의 경우 15분 이상까지 다양합니다.

통치

이 아키텍처 스타일의 거버넌스 측면에서 이 소식은 매우 좋습니다. 왜냐하면 이 아키텍처가 매우 일반적이기 때문에 초기 구조 테스트 도구를 개발한 설계자들은 이 아키텍처를 염두에 두고 설계했기 때문입니다. 실제로 그림 6-4의 예제 적합도 함수는 계층형 아키텍처를 위해 만들어졌습니다(예제 10-1 참조).

예제 10-1. 레이어를 제어하는 ArchUnit의 적합성 함수

```
layeredArchitecture() .layer("Controller").definedBy("..controller..") .layer("Service").definedBy("..service..") .layer("Persistence").definedB
```

```
.whereLayer("Controller").mayNotBeAccessedByAnyLayer() .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller") .whe
```

예제 10-1 에서 아키텍트는 아키텍처의 계층을 정의하고 패키지 이름으로 표현되는 구성 요소에 Controller 와 같은 편리한 이름을 지정합니다. (ArchUnit 구문에서 패키지 이름 양쪽에 있는 두 개의 마침표 (..)는 소유권을 나타냅니다.) 그런 다음 아키텍트는 계층 간에 허용되는 통신을 정의하고 개방형 및 폐쇄형 계층을 관리합니다.

피트니스 함수 라이브러리는 계층형 아키텍처 스타일을 매우 잘 지원하며, 설계자가 구현 과정에서 계층 간에 설계한 관계에 대한 관리를 자동화할 수 있도록 해줍니다.

팀 토폴로지 고려 사항

이 책에서 설명하는 다른 아키텍처 스타일과는 달리, 계층형 아키텍처 스타일은 일반적으로 팀 구성과 무관하며 어떤 팀 구성에서도 작동합니다.

스트림 중심 팀 구성은 계층

: 형 아키텍처가 일반적으로 규모가 작고 독립적이며 시스템을 통한 단일 여정 또는 흐름을 나타내기 때문에 효과적입니다. 이러한 팀 구성에서 팀은 일반적으로 각 계층을 통해 시스템의 흐름을 처음부터 끝까지 책임지고 솔루션의 일부로 워크플로를 생성합니다.

계층형 아키텍처는

: 모듈화 수준이 높고 기술적 관심사에 따라 분리되어 있어 팀 구성에 매우 적합합니다. 전문가와 여러 분야에 걸친 팀 구성원은 하나 이상의 계층과 상호 작용하여 제안을 하거나 실험을 수행할 수 있으며, 나머지 흐름에는 영향을 미치지 않습니다. 예를 들어, 팀은 다른 계층을 변경 사항으로부터 격리한 상태에서 프레젠테이션 계층에 새로운 동작을 추가하여 새로운 UI 라이브러리를 실험할 수 있습니다.

복잡한 하위 시스템 팀

: 각 계층이 매우 특정한 작업을 수행하기 때문에 이러한 방식은 복잡한 하위 시스템 팀 토폴로지에 적합합니다. 예를 들어, 영구 저장 계층은 분석 목적으로 운영 데이터에 접근해야 하는 팀에게 완벽한 연결 고리를 제공합니다. 영구 저장 계층에 대한 접근 권한이 부여되면, 복잡한 하위 시스템 팀은 스트림 중심 팀이 여전히 소유하고 있는 다른 계층에 영향을 주지 않고 작업을 수행할 수 있습니다.

계층형 아키텍처
에서 작업하는 플랫폼 팀은 해당 아키텍처에 사용 가능한 다양한 도구를 활용하여 높은 수준의 모듈성을 최대한 활용할 수 있습니다.

대부분의 플랫폼 팀이 계층형 아키텍처를 사용할 때 직면하는 가장 큰 어려움은 일반적인 모놀리식 아키텍처의 문제점과 유사합니다. 즉, 모놀리식 아키텍처는 규모가 커질수록 다루기 어려워진다는 것입니다. 아무리 잘 분할되고 관리되더라도 팀이 모놀리식 아키텍처에 기능을 계속 추가하면 결국 데이터베이스 연결, 메모리, 성능, 동시 사용자 수 등 여러 제약 조건에 도달하게 됩니다. 시스템을 안정적으로 운영하려면 플랫폼 팀은 점점 더 어려운 작업을 수행해야 합니다.

스타일 특징

특성 평가표(그림 10-6 참조) 에서 별 1개 는 해당 아키텍처 특성이 아키텍처에서 제대로 지원되지 않음을 의미하고, 별 5개는 해당 아키텍처 특성이 아키텍처 스타일에서 가장 강력한 기능 중 하나임을 의미합니다. 평가표에 제시된 각 특성에 대한 정의는 4 장 에서 확인할 수 있습니다 .

	Architectural characteristic	Star rating
	Overall cost	$
Structural	Partitioning type	Technical
Structural	Number of quanta	1
Structural	Simplicity	★★★★★
Structural	Modularity	★
Engineering	Maintainability	★
Engineering	Testability	★★
Engineering	Deployability	★
Engineering	Evolvability	★
Operational	Responsiveness	★★★
Operational	Scalability	★
Operational	Elasticity	★
Operational	Fault tolerance	★

그림 10-6. 계층형 아키텍처 특성 평가

전반적인 비용 효율성과 단순성이 계층형 건축 방식의 주요 강점입니다.
단일 구조의 계층형 아키텍처는 분산 아키텍처 방식만큼 복잡하지 않습니다. 더 단순하고 이해하기 쉬우며 구축 및 유지 관리 비용도 상대적으로 저렴합니다. 하지만 단일 구조의 계층형 아키텍처가 커지고 복잡해질수록 이러한 장점은 빠르게 감소하므로 주의해야 합니다.

이 아키텍처 스타일은 배포 용이성과 테스트 용이성 모두에서 낮은 평가를 받습니다. 배포 용이성이 낮은 이유는 배포 자체가 위험 부담이 크고, 빈도가 낮으며, 많은 절차와 노력이 필요하기 때문입니다. 예를 들어, 클래스 파일에서 간단한 세 줄만 수정하더라도 전체 배포 단위를 재배포해야 하는데, 이 과정에서 데이터베이스, 구성 또는 코드의 다른 부분에도 변경 사항이 함께 반영될 가능성이 있습니다. 더욱이, 이러한 간단한 세 줄 수정은 대개 수십 개의 다른 수정 사항과 함께 이루어지며, 각각의 수정 사항은 배포의 위험성과 빈도를 더욱 증가시킵니다.

낮은 테스트 용이성 평가는 이러한 상황을 반영합니다. 간단한 세 줄 변경만으로도 대부분의 개발자는 전체 회귀 테스트 스위트를 실행하는 데 몇 시간을 소비하지 않을 것입니다(설령 그런 테스트 스위트가 있다고 하더라도). 이 스타일은 구성 요소 또는 전체 레이어를 모킹하거나 스텁 처리할 수 있어 전반적인 테스트 노력을 줄여주기 때문에 테스트 용이성에 별점 1개가 아닌 2개를 부여했습니다.

계층형 아키텍처 스타일의 엔지니어링 특성은 위에서 언급한 역동성을 반영합니다. 즉, 처음에는 모두 잘 작동하지만 코드베이스의 규모가 커짐에 따라 성능이 저하됩니다.

계층형 아키텍처의 탄력성과 확장성은 매우 낮습니다(별 1개). 이는 주로 단일체 배포 방식과 아키텍처 모듈성 부족 때문입니다. 단일체 내의 특정 기능은 다른 기능보다 더 크게 확장할 수 있지만, 이를 위해서는 멀티스레딩, 내부 메시징, 기타 병렬 처리 방식과 같은 매우 복잡한 설계 기법이 필요하며, 이러한 기법은 계층형 아키텍처에 적합하지 않습니다. 또한, 계층형 시스템의 아키텍처 양자역학은 항상 1(단일체 UI 및 데이터베이스, 백엔드 처리 방식 때문)이므로 애플리케이션은 특정 지점까지만 확장할 수 있습니다.

계층형 아키텍처에서 신중한 설계를 통해 높은 응답성을 확보할 수 있으며, 캐싱 및 멀티스레딩과 같은 기술을 통해 이를 더욱 향상시킬 수 있습니다. 하지만 이 스타일은 내재적인 병렬 처리 기능의 부족, 폐쇄적인 계층 구조, 그리고 아키텍처 싱크홀이라는 안티패턴으로 인해 한계가 있기 때문에 전반적으로 별 3개를 부여합니다.

사용 시점

계층형 아키텍처 스타일은 작고 간단한 애플리케이션이나 웹사이트에 적합합니다. 또한 예산과 시간이 매우 빠듯한 상황에서 좋은 출발점이 될 수 있습니다. 단순하고 개발자와 아키텍트에게 친숙하기 때문에 개발 비용이 가장 저렴한 스타일 중 하나이며, 소규모 애플리케이션 개발을 용이하게 합니다. 계층형 아키텍처 스타일은 다음과 같은 경우에도 좋은 선택입니다.

건축가들은 더 복잡한 구조가 더 적합할지 여부를 아직 결정하지 못했지만, 개발을 시작해야 합니다.

이 기법을 사용할 때는 코드 재사용을 최소화하고 객체 계층 구조(상속 트리의 깊이)를 얕게 유지하여 모듈성을 높게 유지해야 합니다. 이렇게 하면 나중에 다른 아키텍처 스타일로 전환하기가 더 쉬워집니다.

사용하지 말아야 할 경우

앞서 살펴본 바와 같이, 계층형 아키텍처 스타일을 사용하는 애플리케이션은 규모가 커질수록 유지보수성, 민첩성, 테스트 용이성, 배포 용이성과 같은 특성이 부정적인 영향을 받습니다.
이러한 이유로 대규모 애플리케이션 및 시스템에는 다른 모듈형 아키텍처 스타일을 사용하는 것이 더 나을 수 있습니다.

예시 및 사용 사례

층층이 쌓는 건축 양식은 가장 흔한 건축 양식 중 하나이며, 다음과 같은 곳에서 나타납니다.
다양한 맥락에서.

리눅스나 윈도우 같은 운영체제 설계자들은 애플리케이션 설계자들과 마찬가지로 관심사 분리라는 이유로 계층 구조를 사용하는 경향이 있습니다. 운영체제의 일반적인 계층 구조는 다음과 같습니다.

하드웨어 계층
: CPU, 메모리, I/O 장치와 같은 물리적 하드웨어가 포함됩니다.

커널 레이어
: 하드웨어 추상화, 메모리 관리 및 프로세스 스케줄링을 제공합니다.

시스템 호출 인터페이스 계층
: 커널과 상호 작용하여 시스템 서비스를 제공합니다.

사용자 레이어
: 사용자가 상호 작용하는 응용 프로그램 및 유틸리티가 포함됩니다.

개방형 시스템 상호 연결(OSI) 네트워킹 모델은 네트워크가 어떻게 책임을 구분해야 하는지에 대한 개념을 제시합니다. 예를 들어, 인터넷의 기본 프로토콜인 TCP/IP는 다음과 같은 계층으로 구성됩니다.

물리 계층
: 물리적으로 데이터를 전송합니다

데이터 링크 계층
: 오류 감지 및 프레임 동기화를 활용합니다.

네트워크 계층
: 라우팅(IP 등)을 처리합니다.

전송 계층
: 안정적인 데이터 전송(예: TCP)을 보장합니다.

애플리케이션 계층
: 이메일(SMTP), 파일 전송(FTP), 웹 브라우징과 같은 서비스를 제공합니다.
(HTTP)

계층형 아키텍처는 관심사 분리를 촉진하고 유지보수성을 향상시키며 각 계층의 독립적인 개발을 가능하게 하므로 이러한 특징을 중시하는 모든 아키텍처에 유용합니다.

계층형 아키텍처는 실현 가능성이라는 아키텍처적 특성을 달성하려는 팀에게 매우 일반적입니다. 즉, 주어진 시간과 가용 자원으로 명시된 범위를 실제로 구현할 수 있는지 여부를 확인하는 데 유용합니다. 예를 들어, 투자 유치로 자금을 조달받아 최대한 빠르게 결과물을 내놓아야 하는 조직이라면, 계층형 아키텍처의 단순성 덕분에 나중에 다른 기능을 구현하기 위해 일부를 재작성해야 하더라도 좋은 선택이 될 수 있습니다.