

제6장

측정과 관리 건축적 특징

아키텍트는 소프트웨어 프로젝트의 모든 측면에서 매우 다양한 아키텍처 특성을 다뤄야 합니다. 성능, 탄력성, 확장성과 같은 운영적 측면은 모듈성, 배포 가능성과 같은 구조적 문제와 얹혀 있습니다. 아키텍트는 모호한 용어와 광범위한 정의에 매몰되기보다는 아키텍처 특성을 측정하고 관리하는 방법을 이해하는 것이 중요합니다. 이 장에서는 몇 가지 일반적인 아키텍처 특성을 구체적으로 정의하고, 이러한 특성을 관리하는 메커니즘을 구축하는 방법을 논의합니다.

건축물의 특성 측정

건축가들이 건축적 특성을 정의하는 데 어려움을 겪는 데에는 여러 가지 이유가 있습니다.

그것들은 물리학이 아닙니다. 흔히 사용되는 많은 아키텍처 특성 용어들은 모호한 의미를 지니고 있습니다. 예를 들어, 아키텍트는 어떻게 민첩성이나 배포 용이성을 고려하여 설계할까요? 극도로 빠른 성능은 또 어떻습니까? 업계 종사자들은 이러한 일반적인 용어에 대해 매우 다양한 관점을 가지고 있는데, 이는 때로는 타당한 맥락의 차이에서 비롯되기도 하고, 때로는 우연한 차이에서 비롯되기도 합니다.

정의가 너무 다양함. 같은 조직 내에서

도 부서마다 성능과 같은 핵심 특성에 대한 정의가 다를 수 있습니다. 개발자, 아키텍트, 운영 담당자 등이 공통된 정의에 합의하기 전까지는 제대로 된 대화를 나눌 수 있을까요?

너무 복합적입니

다. 바람직한 아키텍처 특성 중 상당수는 실제로 더 작은 규모의 다른 특성들의 집합체입니다. 5 장에서 다른 복합 아키텍처 특성에 대해 기억하실 수도 있을 것입니다. 예를 들어, 민첩성은 모듈성, 배포 용이성, 테스트 용이성과 같은 특성으로 나눌 수 있습니다.

복합적인 건축적 특징을 구성 요소로 분해하는 것은 건축적 특징에 대한 객관적인 정의를 확립하는 데 중요한 부분이며, 이는 앞서 언급한 세 가지 문제를 모두 해결합니다.

조직 구성원 모두가 아키텍처 특성에 대한 표준적이고 구체적인 정의를 사용하기로 합의하면, 아키텍처에 대한 보편적인 언어가 만들어집니다.

이러한 표준화를 통해 복합적인 특성을 분석하여 객관적으로 측정 가능한 특징을 밝혀낼 수 있습니다.

운영 조치

많은 아키텍처 특성은 성능이나 확장성처럼 명확하게 직접 측정할 수 있습니다. 하지만 이러한 측정값조차도 팀의 목표에 따라 다양한 해석이 가능합니다. 예를 들어, 팀에서 특정 요청에 대한 평균 응답 시간을 측정한다고 가정해 보겠습니다. 이는 운영 아키텍처 특성을 측정하는 좋은 예입니다. 하지만 평균값만 측정한다면, 어떤 예외적인 상황으로 인해 요청의 1%가 다른 요청보다 10배 더 오래 걸리는 경우는 어떻게 될까요? 사이트 트래픽이 충분하다면 이러한 이상치는 감지되지 않을 수도 있습니다. 이상치를 파악하려면 최대 응답 시간도 측정하는 것이 좋습니다.

고위급 팀은 단순히 수치적인 성능 지표를 설정하는 데 그치지 않고, 통계 분석을 기반으로 정의를 내립니다. 예를 들어, 비디오 스트리밍 서비스가 확장성을 모니터링해야 한다고 가정해 보겠습니다. 엔지니어들은 임의의 수치를 목표로 설정하는 대신, 시간 경과에 따른 확장성을 측정하고 통계 모델을 구축한 다음, 실시간 지표가 예측 모델 범위를 벗어나면 경고를 발생시킵니다. 만약 그렇다면, 이는 두 가지를 의미할 수 있습니다. 모델이 잘못되었다는 것(팀이 알고 싶어하는 정보)이거나, 무언가 잘못되었다는 것(역시 팀이 알고 싶어하는 정보)입니다.

팀에서 측정하는 특성의 종류는 도구, 목표, 장치 및 기능과 함께 빠르게 진화합니다. 예를 들어, 최근 많은 팀은 모바일 장치에서 웹 페이지 사용자의 성능 문제를 잘 보여주는 지표인 RST 콘텐츠 표시 시간(rst contentful paint) 및 RST CPU 유휴 시간(rst CPU idle)과 같은 지표에 대한 성능 예산에 집중하고 있습니다. 이러한 것들을 비롯한 수많은 변화가 계속됨에 따라 팀은 새로운 측정 지표와 방법을 찾아낼 것입니다.

성능의 다양한 측면 우리가 설명하는 아키텍처

특성 중 상당수는 여러 가지 정의를 가지고 있습니다. 성능이 바로 좋은 예입니다. 많은 프로젝트에서 일반적인 성능, 예를 들어 웹 애플리케이션의 요청 및 응답 주기 시간을 살펴봅니다. 하지만 많은 기업의 아키텍트와 DevOps 엔지니어들은 엄청난 노력을 통해 애플리케이션의 특정 부분에 대한 구체적인 성능 예산을 설정해 왔습니다. 예를 들어, 많은 기업들이 사용자 행동을 연구하여 첫 페이지 렌더링(브라우저나 모바일 기기에서 웹 페이지의 진행 상황을 처음으로 보여주는 표시)에 최적의 시간이 몇 분의 1초라는 것을 알아냈습니다. 대부분의 애플리케이션은 이 지표에서 두 자릿수 범위에 속합니다. 하지만 가능한 한 많은 사용자를 확보하려는 최신 웹사이트의 경우, 이 지표는 추적해야 할 중요한 요소이며, 이러한 웹사이트를 개발하는 기업들은 매우 세밀한 측정 기준을 마련해 왔습니다.

이러한 지표 중 일부는 애플리케이션 설계에 추가적인 영향을 미칩니다. 많은 선구적인 조직에서는 페이지 다운로드에 K-가중치 예산을 설정합니다. 즉, 특정 페이지에 허용되는 라이브러리 및 프레임워크의 최대 용량(바이트 수)을 제한하는 것입니다. 이러한 접근 방식의 근거는 물리적 제약에 있습니다. 특히 대역폭이 낮은 모바일 기기의 경우, 네트워크를 통해 한 번에 전송할 수 있는 바이트 수에 한계가 있기 때문입니다.

지역.

구조적 조치

성능처럼 명확하게 드러나지 않는 객관적인 측정 기준도 있습니다. 예를 들어, 잘 정의된 모듈성 같은 내부 구조적 특성은 어떨까요? 이는 암묵적인 아키텍처 특성의 좋은 예입니다. 아키텍트는 구성 요소와 상호 작용을 정의하고, 전반적으로 높은 품질을 보장하는 지속 가능한 구조를 구축하고자 합니다. 안타깝게도 아키텍처의 품질을 종합적으로 평가할 수 있는 지표는 아직 없습니다. 하지만 아키텍트가 코드 구조의 몇 가지 중요한 측면을, 비록 좋은 범위일지라도, 다룰 수 있도록 해주는 지표와 일반적인 도구들은 존재합니다.

코드의 측정 가능한 측면 중 하나는 순환 복잡도(Cyclomatic Complexity)라는 지표로 정의되는 복잡도입니다.

순환 복잡도 (Cyclomatic)

Complexity, CC) 는 1976년 토마스 맥케이브 시니어가 함수/메서드, 클래스 또는 애플리케이션 수준에서 코드의 복잡도를 객관적으로 측정하기 위해 고안한 코드 수준 메트릭입니다. CC는 그래프 이론을 코드에 적용하여 계산되며, 특히 서로 다른 실행 경로를 발생시키는 결정 지점을 이용합니다. 예를 들어, 함수에 if 문과 같은 결정문이 없으면 CC는 1입니다.

함수에 조건문이 하나만 있는 경우, 실행 경로가 두 개 가능하므로 CC는 2가 됩니다.

단일 함수 또는 메서드에 대한 CC를 계산하는 공식은 $CC = E - N + 20$ 이며, 여기서 N은 노드(코드 라인)를 나타내고 E는 에지(가능한 결정)를 나타냅니다. [예제 6-1에 나와 있는 C와 유사한 코드를 고려해 보세요.](#)

예제 6-1. 순환 복잡도 평가를 위한 샘플 코드

```
public void decision(int c1, int c2) { if (c1 < 100) return 0; else
    if (c1 + c2 > 500)
        return 1; }

    -1을 반환합니다.
}
```

예제 6-1 의 순환 복잡도는 $3(3 - 2 + 2)$ 이며, [그림 6-1에 나타나 있습니다.](#)

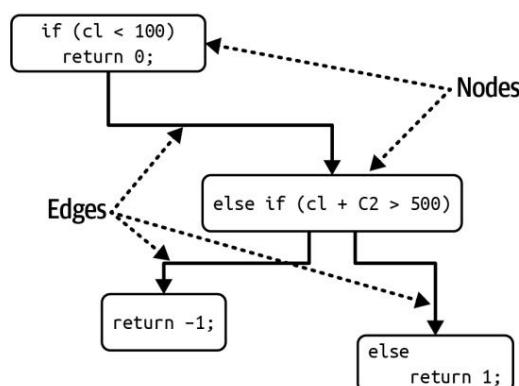


그림 6-1. 결정 함수의 순환 복잡도 그래프

순환 복잡도 공식에 나타나는 숫자 2는 단일 함수/메서드에 대한 간소화를 나타냅니다. 다른 메서드를 호출하는 팬아웃 호출(그래프 이론에서 연결 요소라고 함)의 경우, 보다 일반적인 공식은 $CC = E - N + 2P$ 이며, 여기서 P는 연결 요소의 개수를 나타냅니다.

아키텍트와 개발자들은 지나치게 복잡한 코드가 "코드 스멜"을 나타낸다는 데 만장일치로 동의합니다. 코드 스멜이란 마치 악취가 나는 것처럼 코드가 매우 나쁘다는 것을 의미합니다. 이는 모듈성, 테스트 용이성, 배포 용이성 등 코드베이스의 바람직한 특성 대부분을 저해합니다. 팀이 점진적으로 증가하는 코드 복잡성을 주의 깊게 살펴보지 않으면 결국 코드베이스 전체를 지배하게 될 것입니다.

순환 복잡도(Cyclomatic Complexity, CC)는 아키텍트가 활용할 수 있는 측정 지표의 한계를 보여주는 대표적인 예입니다. CC는 코드의 복잡도를 측정하지만, 그 복잡도가 필수적인지(복잡한 문제를 해결하기 때문인지) 아니면 의도치 않은지(잘못된 설계로 인해 발생하는지)는 판단할 수 없습니다. CC와 같은 지표는 개발자가 작성한 코드든 생성형 AI가 생성한 코드든 평가하는 데 매우 유용합니다. 생성형 AI는 종종 무차별 대입 방식으로 문제를 해결하려 하기 때문에 의도치 않은 복잡도가 발생하는 경우가 많습니다.

순환 복잡도에 대한 적절한 값은 무엇일까요?

저자들이 이 주제에 대해 이야기할 때 자주 받는 질문은 "적절한 CC 임계값은 얼마인가요?"입니다. 물론 소프트웨어 아키텍처에 관한 모든 질문과 마찬가지로 답은 "상황에 따라 다릅니다!"입니다. 구체적으로 말하면, 문제 영역의 복잡성에 따라 달라집니다. 예를 들어 알고리즘적으로 복잡한 문제라면 해결책은 복잡한 함수를 생성할 것입니다. 아키텍트가 모니터링해야 할 CC의 주요 측면은 다음과 같습니다. 함수가 복잡한 이유는 문제 영역 때문인가요, 아니면 코딩이 미흡하기 때문인가요?

또는 코드가 제대로 분할되지 않은 것은 아닐까요? 다시 말해, 큰 메서드를 더 작고 논리적인 단위로 나누어 작업(및 복잡성)을 더 잘 분리된 메서드로 분산시킬 수 있을까요?

일반적으로 코드 커버리지(CC)에 대한 업계 표준은 복잡한 도메인과 같은 다른 요소를 고려하지 않는 한 10 미만의 값을 허용 가능한 것으로 제시합니다. 하지만 저희는 이 기준이 너무 높다고 생각하며, 응집력 있고 잘 분리된 코드를 나타내는 5 미만의 CC 값을 선호합니다. 자바 개발 환경에서 사용되는 [Crap4J](#)라는 메트릭 도구는 CC와 코드 커버리지를 조합하여 코드의 질이 얼마나 나쁜지(영성한지)를 평가합니다. CC가 50을 넘으면 아무리 코드 커버리지를 높여도 영성한 코드를 개선할 수 없습니다. Neal이 접했던 가장 끔찍한 전문적인 결과물은 상용 소프트웨어 패키지의 핵심 역할을 하는 단 하나의 C 함수였는데, 그 함수의 CC가 800을 넘었던 것입니다!

그것은 4,000줄이 넘는 코드로 이루어진 단일 함수였는데, (불가능할 정도로 깊게 중첩된 루프를 피하기 위해) GOTO 문을 많이 사용했습니다.

테스트 주도 개발(TDD)과 같은 엔지니어링 방식은 주어진 문제 영역에서 평균적으로 더 작고 덜 복잡한 메서드를 생성하는 유익한 부수적 효과를 가져옵니다. TDD를 실천할 때 개발자는 간단한 테스트를 작성한 다음, 해당 테스트를 통과하는 데 필요한 최소한의 코드만 작성하려고 합니다. 이처럼 개별적인 동작과 명확한 테스트 경계에 집중함으로써, 잘 구성되고 응집력이 높은 메서드가 생성되며, 그 결과 전달되는 전달률(CC)이 낮아집니다.

프로세스 측정

아키텍처의 일부 특성은 소프트웨어 개발 프로세스와 연관됩니다. 예를 들어, 애자일성은 바람직한 특징으로 자주 언급되지만, 테스트 용이성 및 배포 용이성과 같은 여러 특성이 복합적으로 작용하는 아키텍처 특성입니다.

테스트 용이성은 거의 모든 플랫폼에서 코드 커버리지 도구를 통해 측정할 수 있으며, 이러한 도구는 테스트가 실행되는 코드의 비율을 보고합니다. 하지만 모든 소프트웨어 검사와 마찬가지로 이러한 도구는 사고력과 의도를 대체할 수는 없습니다. 예를 들어, 코드 커버리지가 100%인 코드베이스라도 실제 코드의 정확성에 대한 확신을 제공하지 못하는 부실한 어설션을 사용할 수 있습니다.

테스트 용이성은 배포 용이성과 마찬가지로 객관적으로 측정 가능한 특성입니다. 배포 용이성 지표에는 성공적인 배포 비율, 배포 소요 시간, 배포 과정에서 발생한 문제/버그 등이 포함됩니다. 모든 팀은 조직과 팀의 우선순위 및 목표에 맞는 유용한 정성적 및 정량적 데이터를 수집할 수 있는 적절한 측정 기준을 마련해야 합니다.

애자일과 그와 관련된 요소들은 소프트웨어 개발 프로세스와 밀접한 관련이 있지만, 그 프로세스 자체가 아키텍처 구조에 영향을 미칠 수도 있습니다. 예를 들어, 배포 용이성과 테스트 용이성이 최우선 순위라면, 아키텍트는 아키텍처 수준에서 우수한 모듈성과 격리성을 강조할 것입니다. 이는 아키텍처 특성이 구조적 결정에 영향을 미치는 사례입니다. 소프트웨어 프로젝트 범위 내의 거의 모든 요소는 세 가지 기준을 충족한다면 아키텍처 특성으로 간주될 수 있으며, 아키텍트는 이를 고려하여 중요한 결정을 내려야 합니다.

관리 및 적합성 기능

아키텍트가 아키텍처 특성을 정립하고 우선순위를 정한 후, 개발자들이 일정 압박에 관계없이 이러한 우선순위를 존중하고 설계를 정확하고 안전하게 구현하도록 어떻게 보장할 수 있을까요? 많은 소프트웨어 프로젝트에서 긴급성이 가장 큰 요소이지만, 아키텍트는 여전히 아키텍처 거버넌스를 제공할 도구와 기술이 필요합니다. 모듈성은 중요하지만 시급하지 않은 아키텍처 측면의 좋은 예입니다.

아키텍처 특성 관리(Governing Architecture)

Characteristics)는 그리스어 'kubernan'(조종하다)에서 유래한 단어로, 아키텍트의 중요한 책임 중 하나입니다. 이름에서 알 수 있듯이, 아키텍처 관리의 범위는 아키텍트가 영향을 미치고자 하는 소프트웨어 개발 프로세스의 모든 측면을 포괄합니다. 예를 들어, 소프트웨어 품질 보장은 아키텍처 관리의 영역에 속합니다. 품질 관리를 소홀히 하면 심각한 품질 문제로 이어질 수 있기 때문입니다.

다행히도 아키텍트들은 이 문제에 대한 점점 더 정교한 해결책을 가지고 있는데, 이는 소프트웨어 개발 생태계 역량의 점진적 성장을 보여주는 좋은 예입니다. [이스트림 프로그래밍\(EP\)](#)에서 비롯된 자동화에 대한 열망은 지속적 통합(CI)을 탄생시켰습니다. CI는 운영 자동화로 이어졌고, 이를 오늘날 우리가 데브옵스(DevOps)라고 부릅니다. 그리고 이러한 흐름은 아키텍처 거버넌스에까지 이어집니다. 닐 포드(Neal Ford) 등이 저술한 『[진화적 아키텍처 구축 \(Building Evolutionary Architectures\)](#)』(O'Reilly, 2022)에서는 아키텍처 거버넌스의 여러 측면을 자동화하는 데 사용되는 적합성 함수(fitness function)라는 기법들을 소개합니다. 이 장의 나머지 부분에서는 적합성 함수에 대해 자세히 살펴보겠습니다.

피트니스 기능

책 제목인 "진화적 아키텍처 구축(Building Evolutionary Architectures)"에 있는 "진화적(evolutionary)"이라는 단어는 생물학보다는 진화 컴퓨팅에서 유래했습니다. 저자 중 한 명인 레베카 파슨스 박사는 유전 알고리즘과 같은 도구를 사용하는 등 진화 컴퓨팅 분야에서 상당한 시간을 보냈습니다. 개발자가 유익한 결과를 도출하도록 유전 알고리즘을 설계할 때, 종종 결과의 질을 객관적으로 측정할 수 있는 기준을 제공하여 알고리즘을 안내하고자 합니다. 이러한 안내 메커니즘을 적합성 함수라고 하는데, 이는 출력이 목표 달성을 얼마나 근접했는지 평가하는 데 사용되는 목적 함수입니다.

예를 들어, 기계 학습의 기초로 사용되는 유명한 문제인 [외판원 문제\(Traveling Salesperson Problem\)](#)를 해결해야 한다고 가정해 보겠습니다. 외판원과 방문해야 할 도시 목록, 그리고 도시 간 거리가 주어졌을 때, 거리, 시간, 비용을 최소화하는 최적의 경로는 무엇일까요? 이 문제를 해결하기 위해 유전 알고리즘을 설계한다면, 경로의 길이를 평가하는 적합도 함수 하나와 경로와 관련된 총비용을 평가하는 적합도 함수 하나를 사용할 수 있습니다. 또한 외판원이 실제로 이동하는 데 걸리는 시간을 평가하는 함수도 사용할 수 있습니다.

진화적 아키텍처의 관행은 이러한 개념을 차용하여 아키텍처 적합성 기능을 만듭니다. 이는 아키텍처 특성 또는 아키텍처 특성 조합의 객관적인 무결성 평가를 제공하는 모든 메커니즘을 의미합니다.

적합도 함수는 아키텍트가 새로 다운로드해야 하는 프레임워크가 아닙니다. 오히려 기존의 여러 도구에 대한 새로운 관점을 제공합니다. 정의에서 "어떤 메커니즘"이라는 표현에 주목하십시오. 아키텍처 특성에 대한 검증 기법은 특성만큼이나 다양합니다. 적합도 함수는 사용 방식에 따라 기존의 여러 검증 메커니즘과 중복될 수 있습니다. 예를 들어 카오스 엔지니어링에서 사용되거나, 메트릭, 모니터 또는 단위 테스트 라이브러리로 사용될 수 있습니다(그림 6-2 참조).

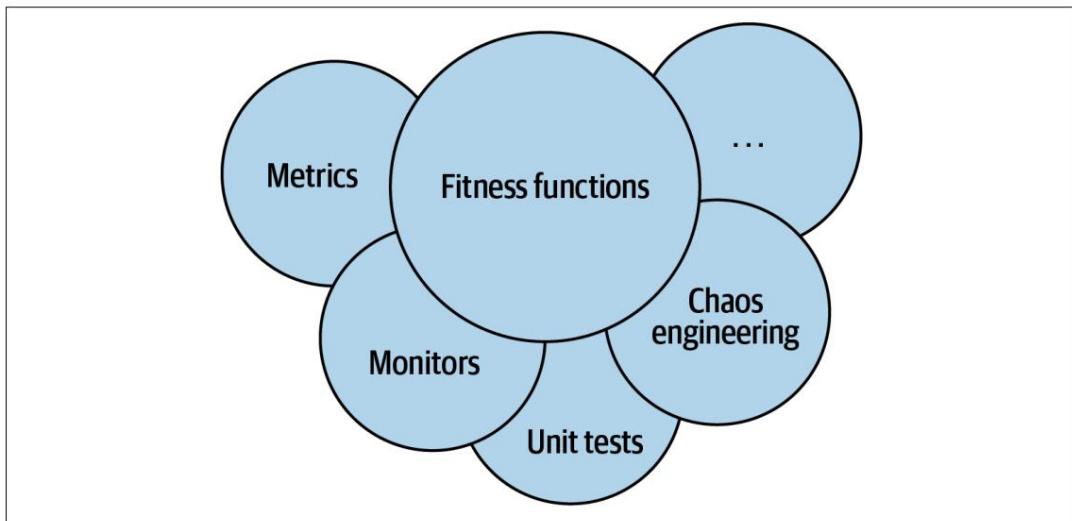


그림 6-2. 기능의 메커니즘

아키텍처 특성에 따라 적합도 함수를 구현하는 데 다양한 도구가 사용될 수 있습니다. 모듈성의 다양한 측면을 테스트하는 몇 가지 적합도 함수의 예를 살펴보겠습니다.

순환 종속성 모듈성

은 대부분의 아키텍트가 중요하게 생각하는 암묵적인 아키텍처 특성입니다.

모듈성이 제대로 유지되지 않으면 코드베이스 구조에 악영향을 미치기 때문에 아키텍트는 일반적으로 우수한 모듈성 유지를 매우 중요하게 생각합니다. 그러나 많은 플랫폼에서 이러한 좋은 의도를 방해하는 요소들이 존재합니다. 예를 들어, 널리 사용되는 Java 또는 .NET 개발 환경에서 개발자가 아직 임포트되지 않은 클래스를 참조하면 IDE에서 해당 참조를 자동으로 임포트할지 묻는 대화 상자가 표시됩니다. 이 대화 상자는 매우 자주 나타나기 때문에 대부분의 프로그래머는 무의식적으로 이를 무시합니다. 하지만 구성 요소 간에 임의로 클래스를 임포트하는 것은 모듈성에 심각한 문제를 야기할 수 있습니다. 예를 들어, [그림 6-3은](#) 아키텍트가 피하고자 하는 특히 해로운 안티패턴인 순환 종속성을 보여줍니다.

[그림 6-3](#)에서 각 구성 요소는 다른 구성 요소의 무언가를 참조합니다. 이러한 네트워크 구조는 모듈성을 저해하는데, 다른 구성 요소를 함께 가져오지 않고는 하나의 구성 요소를 재사용할 수 없기 때문입니다. 게다가 다른 구성 요소들이 또 다른 구성 요소들과 결합되어 있다면 어떻게 될까요? 아키텍처는 점점 더 '**빅 볼 오브 머드(Big Ball of Mud)**'라는 안티패턴으로 기울어지게 됩니다. 아키텍트는 개발자들을 끊임없이 감시하지 않고 어떻게 이러한 문제를 관리할 수 있을까요? 코드 리뷰는 도움이 되지만, 개발 주기 후반에 이루어지기 때문에 충분한 효과를 발휘하기 어렵습니다. 개발팀이 코드 리뷰가 있기 전까지 일주일 동안 코드베이스 전체에서 무분별하게 코드를 가져온다면, 이미 코드베이스에 심각한 손상을 입힌 것입니다.

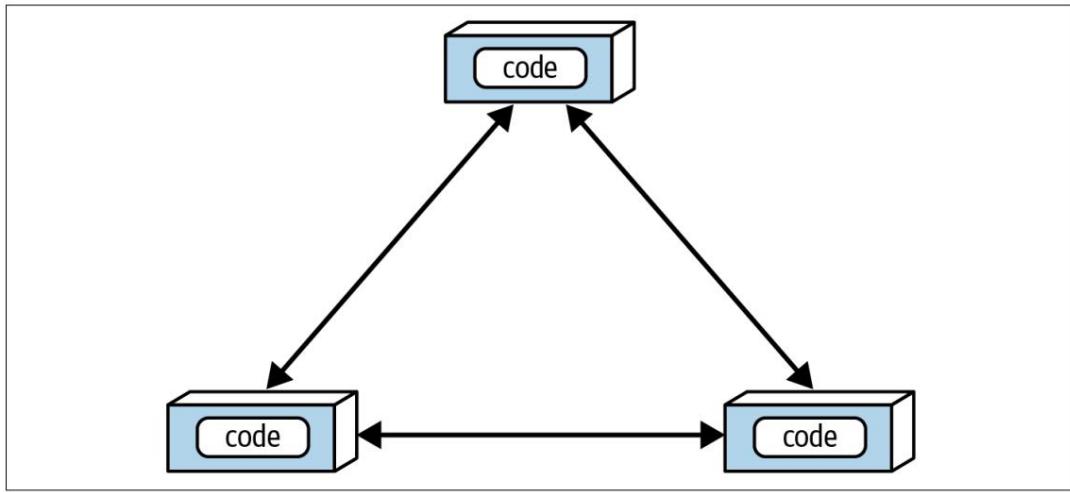


그림 6-3. 구성 요소 간의 순환적 의존성

이 문제에 대한 해결책은 [예제 6-2](#)에 나와 있는 것처럼 사이클을 고려하는 적합도 함수를 작성하는 것입니다.

예제 6-2. 구성 요소 주기를 감지하는 적합도 함수

```

public class CycleTest {
    개인 JDepend jdepend;

    @BeforeEach
    void init() { jdepend
        = new JDepend();
        jdepend.addDirectory("/path/to/project/persistence/classes"); jdepend.addDirectory("/path/
        to/project/web/classes"); jdepend.addDirectory("/path/to/project/thirdpartyjars");

    }

    @Test
    void testAllPackages() { Collection
        packages = jdepend.analyze(); assertEquals("사이클이 존재합니
        다", false, jdepend.containsCycles());
    }
}
  
```

이 코드는 **JDepend**라는 메트릭 도구를 사용하여 패키지 간의 종속성을 확인합니다. 이 도구는 Java 패키지의 구조를 이해하고 순환 참조가 발견되면 테스트를 실패 처리합니다. 아키텍트는 이 테스트를 프로젝트의 지속적인 빌드 과정에 통합하여, 개발자들이 실수로 순환 참조를 추가하는 것을 방지 할 수 있습니다. 이는 소프트웨어 개발에서 시급한 관행보다는 중요한 관행을 보호하는 적합성 함수의 훌륭한 예입니다. 즉, 아키텍트에게는 중요한 고려 사항이지만, 일상적인 코딩에는 큰 영향을 미치지 않습니다.

주계열과의 거리 적합도 함수 [44페이지의 "결합"](#)에
서 우리는 주계열과의 거리라는 다소 난해한 측정 기준을 소개했는데, 이는 [예제 6-3에서 보여주는 것](#)
처럼 적합도 함수를 사용하여 검증할 수도 있습니다.

예제 6-3. 주계열과의 거리 함수

```
@Test
void AllPackages() {
    double ideal = 0.0; double
    tolerance = 0.5; // 프로젝트 종속 Collection packages =
    jdepend.analyze(); Iterator iter = packages.iterator(); while
    (iter.hasNext()) { JavaPackage p =
    (JavaPackage)iter.next();
    assertEquals("Distance exceeded: " + p.getName(), ideal,
    p.distance(), tolerance); }

}
```

이 코드는 JDepend를 사용하여 허용 가능한 값에 대한 임계값을 설정하고, 클래스가 해당 범위를 벗어나면 테스트를 실패하도록 합니다. (다음 섹션에서 자세히 설명하는 ArchUnit 도구를 사용하면 아키텍트가 유사한 적합도 함수를 만들 수 있습니다.) 아키텍처 특성에 대한 객관적인 측정 방법의 이 예시는 적합도 함수를 설계하고 구현할 때 개발자와 아키텍트 간의 협업이 얼마나 중요한지 보여줍니다. 아키텍트 그룹이 상아탑에 올라가 개발자가 이해할 수 없는 난해한 적합도 함수를 개발하는 것이 목적이 아니라, 코드베이스의 품질을 보장하는 자동화된 관리 규칙을 구현하는 것이 목적인입니다.



건축가는 개발자에게 적합성 함수를 적용하기 전에 개발자가 그 함수의 목적을 이해하도록 해야 합니다.

지난 몇 년 동안, 특히 특정 목적에 특화된 도구들이 등장하면서 적합도 함수 도구의 정교함이 크게 향상되었습니다. 그중 하나가 [ArchUnit으로, JUnit](#) 생태계의 여러 요소를 기반으로 개발되고 활용하는 Java 테스트 프레임워크입니다.

ArchUnit은 단위 테스트로 코드화된 다양한 사전 정의된 거버넌스 규칙을 제공하며, 아키텍트가 모듈성을 다루는 특정 테스트를 작성할 수 있도록 합니다. [그림 6-4에 나타난 계층형 아키텍처를 참조하십시오.](#)

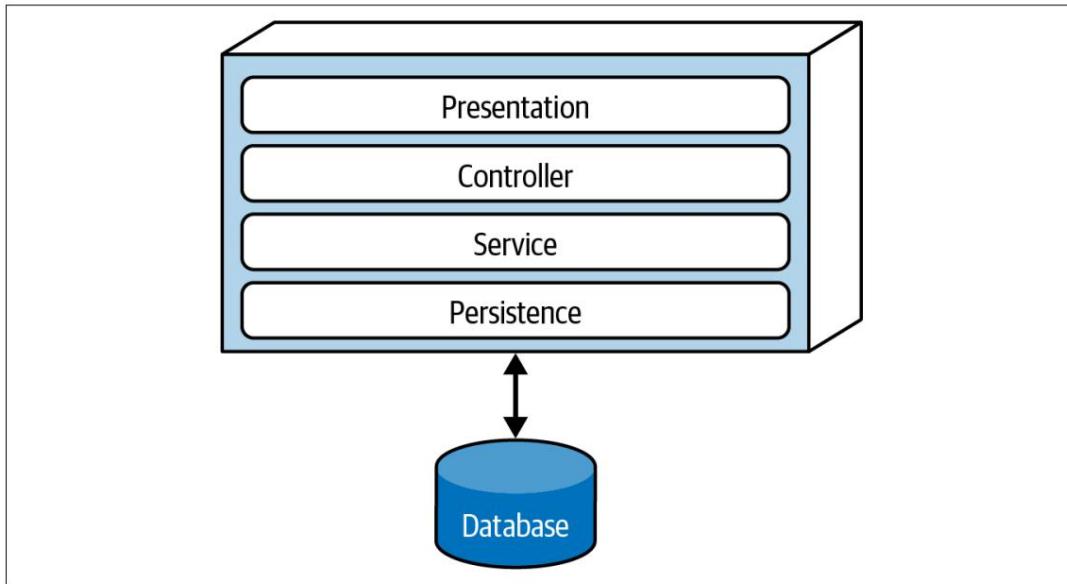


그림 6-4. 계층형 아키텍처

그림 6-4 와 같은 계층형 모듈리를 설계할 때 , 아키텍트는 타당한 이유로 계층을 정의합니다(이러한 동기, 절충 점 및 기타 측면은 10 장 에서 설명합니다). 그러나 일부 개발자는 이러한 패턴의 중요성을 이해하지 못하거나, 성능과 같은 최우선 고려 사항 때문에 "허락을 구하기보다는 용서를 구하는 편이 낫다"는 태도를 취할 수도 있습니다.

하지만 그들이 건축물의 존재 이유를 훼손하도록 내버려 두는 것은 건축물의 장기적인 건전성을 해칠 것입니다.

ArchUnit은 예제 6-4에 나와 있는 적합도 함수를 통해 설계자가 이 문제를 해결할 수 있도록 합니다 .

예제 6-4. 레이어를 제어하는 ArchUnit의 적합성 함수

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

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

예제 6-4 에서 설계자는 계층 간의 바람직한 관계를 정의하고 이를 제어하기 위한 검증 적합성 함수를 작성합니다.

.NET 환경에도 NetArchTest라는 유사한 도구가 있습니다 . 예제 6-5는 C#에서 레이어 검증을 보여줍니다.

예제 6-5. 레이어 종속성 검사를 위한 NetArchTest

```
// 프레젠테이션의 클래스는 리포지토리를 직접 참조해서는 안 됩니다. var result = Types.InCurrentDomain()

    .저것()
    .ResideInNamespace("NetArchTest.SampleLibrary.Presentation")
    .ShouldNot()
    .HaveDependencyOn("NetArchTest.SampleLibrary.Data")
    .GetResult()
    .성공적입니다;
```

테스트 용이성에 대한 논의는 모든 측정 지표의 문제점, 즉 개발자들이 시스템을 악용하려 들 가능성 을 부각합니다. 개발자들은 아키텍트가 규정 준수를 측정하는 방식을 알게 되면, 올바른 것을 만드 는 대신 측정 지표에 맞춰 코드를 작성할 수 있습니다. 예를 들어, 개발자들이 지름길을 택하기 위해 어설션 없이 유닛 테스트를 작성하는 경우가 흔한데, 이는 코드 커버리지 지표를 속이는 행위입니다. 코드를 "만지기만" 하고 제대로 작동하는지 검증하지 않는 것은 코드 커버리지 지표를 속이는 것과 마찬가지입니다. ArchUnit과 같은 도구를 사용하여 거버넌스 코드를 작성하면 모든 유닛 테스트에 최소한 하나의 어설션이 포함되도록 함으로써 이러한 행위를 방지할 수 있습니다. 물론, 악의적인 규 칙 위반자는 항상 방법을 찾아내겠지만, 이와 같은 적합성 함수는 의도치 않은 오류를 막아줍니다.

적합성 함수의 또 다른 예로는 넷플릭스의 "카오스 몽키"와 그에 따른 **유인원 군대가 있습니다**. 넷플 릭스가 아마존 클라우드로 운영을 이전하기로 결정했을 때, 아키텍트들은 더 이상 운영을 직접 제어 할 수 없게 되었고, 이는 운영상의 결함이 발생할 경우 어떻게 될지 우려하게 만들었습니다. 이 문제 를 해결하기 위해 그들은 카오스 엔지니어링이라는 분야를 창안했습니다. 카오스 몽키는 본질적으로 실제 운영 환경을 시뮬레이션하는 적합성 함수 역할을 하며, 시스템이 이러한 환경에 얼마나 잘 견딜 수 있는지를 테스트합니다. 일부 AWS 인스턴스에서 지역 시간이 문제가 되었기 때문에 카오스 몽키 는 높은 지역 시간을 시뮬레이션했습니다. (실제로 이는 심각한 문제였고, 결국에는 지역 시간 문제를 해결하기 위한 특수 도구인 "레이턴시 몽키"를 개발하기도 했습니다.) 이러한 경험을 바탕으로 그들 은 추가적인 도구를 개발했습니다. 예를 들어, 아마존 데이터센터 전체의 장애를 시뮬레이션하는 "카 오스 콩"은 실제 장애 발생 시 넷플릭스가 이를 방지하는 데 도움을 주었습니다.

특히, 적합성, 보안 및 관리 "원숭이"(적합성 기능)는 자동화된 거버넌스 접근 방식을 잘 보여줍니다. 적합성 원숭이를 통해 Netflix 아키텍트는 프로덕션 환경에서 원숭이가 시행하는 거버넌스 규칙을 정 의할 수 있습니다.

예를 들어, 각 서비스가 모든 요청에 대해 오류 없이 응답해야 한다고 결정하면, 해당 검사를 적합성 검사 도구(Conformity Monkey)에 추가합니다. 보안 검사 도구(Security Monkey)는 활성화되 어서는 안 되는 포트나 구성 오류와 같은 잘 알려진 보안 결함을 각 서비스에서 검사합니다.

마지막으로, Janitor Monkey는 다른 서비스에서 더 이상 라우팅하지 않는 인스턴스를 찾습니다. Netflix는 진화하는 아키텍처를 가지고 있기 때문에 개발자들은 정기적으로 새로운 서비스로 마이그 레이션하고, 그 결과 기존 서비스는 협력 서비스 없이 계속 실행됩니다.

클라우드 서비스는 비용을 소모하므로, 청소부 원숭이는 사용되지 않는 서비스를 찾아 프로덕션 환경에서 제거합니다.

카오스 엔지니어링은 아키텍처에 대한 흥미로운 새로운 관점을 제시합니다. 즉, 무언가가 결국 고장 날지 여부가 아니라 언제 고장 날 것인가의 문제라는 것입니다. 고장 발생 시점을 예측하고 이를 방지하기 위한 테스트를 수행하면 시스템의 견고성을 크게 향상시킬 수 있습니다. 넷플릭스의 혁신가인 케이시 로젠탈과 노라 존스가 쓴 책 [『카오스 엔지니어링』\(오라일리, 2020\)](#)은 이러한 접근 방식을 잘 보여줍니다.

아툴 가완데의 영향력 있는 저서 [『체크리스트 선언\(Checklist Manifesto\)』](#) (메트로폴리탄, 2009)은 항공기 조종사나 외과의사와 같은 전문가들이 체크리스트를 어떻게 활용하는지 설명합니다. (어떤 경우에는 법적으로 의무화되어 있기도 합니다.) 전문가들이 이 업무를 모르거나 건망증이 있어서 체크리스트를 사용하는 것이 아닙니다. 오히려 전문가들이 매우 세밀한 작업을 반복적으로 수행하다 보면 세부 사항을 놓치기 쉬운데, 간결한 체크리스트가 효과적인 알림 역할을 하는 것입니다.

적합도 함수에 대한 올바른 관점은 이것입니다. 적합도 함수는 강력한 거버넌스 메커니즘이 아니라, 아키텍트가 중요한 아키텍처 원칙을 표현하고 자동으로 검증할 수 있도록 돋는 메커니즘입니다. 개발자는 안전하지 않은 코드를 배포해서는 안 된다는 것을 알고 있지만, 그 외에도 수십, 수백 가지의 우선순위가 있습니다. Security Monkey와 같은 도구, 그리고 적합도 함수는 아키텍트가 아키텍처의 기반에 중요한 거버넌스 검사를 구축할 수 있도록 해줍니다.

