

## 건축적 특징 정의

이제 소프트웨어 아키텍트의 핵심 역할 중 하나인 구조 설계에 대한 자세한 내용을 살펴보겠습니다. 구조 설계는 크게 두 가지 활동으로 구성됩니다. 하나는 이 장에서 다루는 아키텍처 특성 분석이고, 다른 하나는 8 장에서 다루는 논리적 구성 요소 설계입니다.

건축가는 이 두 가지 활동을 어떤 순서로든 (또는 동시에) 수행할 수 있지만, 중요한 접점에서 만나게 됩니다.

기업이 소프트웨어를 사용하여 특정 문제를 해결하기로 결정하면 해당 시스템에 대한 요구 사항 목록을 수집합니다 (요구 사항을 도출하는 다양한 기법이 있으며, 8 장에서 다룹니다). 이 책 전체에서 이러한 요구 사항을 문제 영역 (또는 단순히 영역)이라고 부르겠습니다. 1 장에서 아키텍처 특성이란 문제 영역과는 독립적이며 시스템의 성공에 중요한 요소라는 것을 배웠습니다. 이 장에서는 아키텍처 특성이라는 용어를 더 자세히 정의하고 구체적인 아키텍처 특성을 살펴보겠습니다.

아키텍트는 도메인 정의에 협력하는 경우가 많지만, 도메인 기능과 직접적으로 관련되지 않은 소프트웨어의 모든 기능, 즉 아키텍처적 특성을 정의하고, 발견하고, 분석해야 합니다. 아키텍처적 특성을 정의하는 아키텍트의 역할은 소프트웨어 아키텍처를 코딩 및 디자인과 구별하는 중요한 요소 중 하나입니다. 또한 그림 4-1에서 볼 수 있듯이 소프트웨어 솔루션을 설계할 때 고려해야 할 다른 많은 요소들도 있습니다.

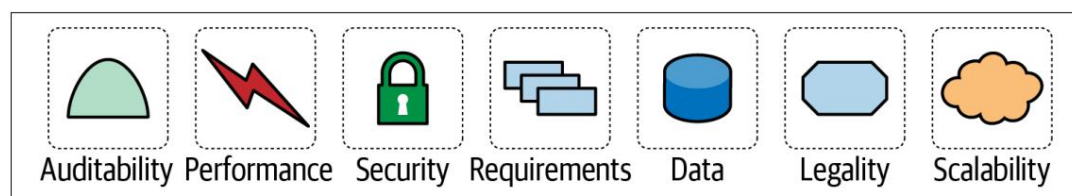


그림 4-1. 소프트웨어 솔루션은 도메인 요구사항과 아키텍처적 특성을 모두 포함한다.

## "비기능 요구사항"이라는 용어의 지속성

많은 조직에서 아키텍처적 특성을 설명하기 위해 다양한 용어를 사용하는데, 그중 하나가 '비기능 요구사항'입니다. 이 용어는 아키텍처적 특성을 기능 요구사항과 구분하기 위해 만들어졌지만, 저희는 이 용어를 좋아하지 않습니다. 자기비하적인 표현일 뿐만 아니라 언어적인 측면에서도 부정적인 의미를 내포하고 있기 때문입니다. "비기능적"인 것에 팀원들이 충분한 관심을 기울이도록 어떻게 설득할 수 있을까요? 또 다른 일반적인 용어로는 '품질 속성'이 있는데, 이 역시 설계 단계가 아닌 사후 품질 평가를 암시하는 의미로 받아들여 선호하지 않습니다.

우리는 아키텍처적 특성이라는 용어를 선호하는데, 이는 아키텍처의 성공, 나아가 시스템 전체의 성공에 매우 중요한 요소들을 설명하면서도 그 중요성을 간과하지 않기 때문입니다. [Head First Software Architecture\(O'Reilly, 2024\)](#)에서는 아키텍처적 특성을 시스템의 역량으로, 도메인을 시스템의 동작으로 정의합니다.

때때로 용어들이 "고착"되는 경우가 있는데, 특히 비기능 요구사항은 소프트웨어 아키텍트들 사이에서 고착화되기 쉬운 용어인 것 같습니다. 이는 여전히 많은 조직에서 흔히 볼 수 있는 현상입니다. 이 용어는 1970년대 후반 소프트웨어 공학 문헌에 처음 등장했는데, 이는 시스템 요구사항을 각각 작업 단위를 나타내는 "기능점"으로 분해하는 추정 기법인 기능점 분석과 거의 같은 시기입니다. 이론적으로 팀은 분석 과정이 끝난 후 모든 기능점을 합산하여 프로젝트에 대한 통찰력을 얻을 수 있었습니다. 하지만 안타깝게도 이는 겉보기에는 확실해 보였지만, 다른 많은 추정 방식처럼 주관성에 크게 의존했으며, 현재는 더 이상 사용되지 않습니다.

하지만 그 당시의 통찰 중 하나는 시스템을 구축하는 데 드는 노력의 상당 부분이 시스템의 요구사항보다는 시스템의 기능에 관한 것이라는 점입니다. 그들은 이러한 노력을 비기능적 요소(non-function points)라고 불렀고, 이것이 계기가 되어 비기능적 요구사항(non-functional requirements)이라는 용어가 널리 사용되게 되었습니다.

## 건축적 특징 및 시스템 설계

아키텍처적 특성으로 간주하려면 요구사항은 세 가지 기준을 충족해야 합니다. 첫째, 도메인과 무관한 설계 고려 사항을 명시해야 하고, 둘째, 설계의 구조적 측면에 영향을 미쳐야 하며, 셋째, 애플리케이션의 성공에 필수적이거나 중요해야 합니다. 이러한 상호 연관된 정의 요소들은 [그림 4-2에 나타나 있으며, 그림 4-2](#)는 이 세 가지 구성 요소와 몇 가지 수정자로 이루어져 있습니다.

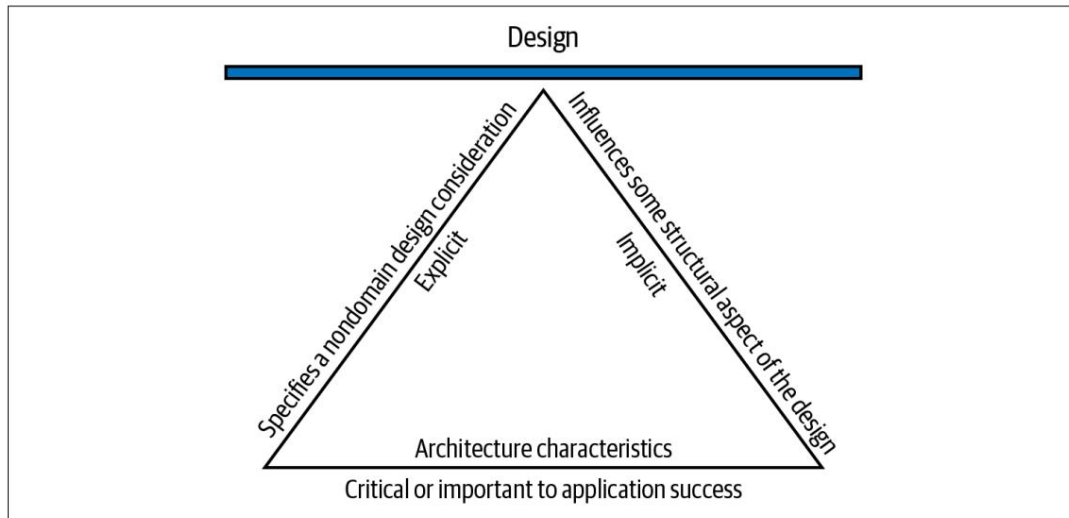


그림 4-2. 건축적 특징의 차별화 요소

이러한 구성 요소를 좀 더 자세히 살펴보겠습니다.

건축적 특성은 비영역 설계 고려 사항에 해당합니다.

소프트웨어 아키텍처에서 구조 설계는 아키텍트가 수행하는 두 가지 활동으로 구성됩니다. 첫째는 문제 영역을 이해하는 것이고, 둘째는 시스템이 성공적으로 작동하기 위해 지원해야 하는 기능들을 파악하는 것입니다. 영역 설계 고려 사항은 시스템의 동작을 다루고, 아키텍처 특성은 기능을 정의합니다. 이 두 가지 활동을 합쳐 구조 설계라고 합니다.

설계 요구사항은 애플리케이션이 수행해야 할 기능을 명시하는 반면, 아키텍처 특성은 요구사항을 구현하는 방법과 특정 선택이 이루어진 이유를 명시합니다. 즉, 프로젝트 성공을 위한 운영 및 설계 기준을 제시합니다.

예를 들어, 특정 성능 수준은 중요한 아키텍처 특성인 경우가 많지만 요구사항 문서에는 나타나지 않는 경우가 흔합니다. 더욱 중요한 것은, 요구사항 문서에 "기술 부채를 사전에 방지해야 한다"고 명시적으로 언급하는 경우는 없지만, 이는 일반적인 설계 고려 사항이라는 점입니다. 명시적 특성과 암묵적 특성의 이러한 차이점에 대해서는 67페이지의 "[도메인 관심사에서 아키텍처 특성 추출하기](#)"에서 자세히 다룹니다.

건축적 특징은 디자인의 구조적 측면에 영향을 미칩니다.

건축가들이 프로젝트의 건축적 특징을 설명하려는 주된 이유는 중요한 설계 고려 사항을 파악하기 위해서입니다. 건축가가 설계를 통해 이를 구현할 수 있는지, 아니면 해당 건축적 특징을 성공적으로 구현하기 위해서는 특별한 구조적 고려가 필요한지 판단하기 위해서입니다.

예를 들어, 보안은 거의 모든 프로젝트에서 중요한 고려 사항이며, 모든 시스템은 설계 및 코딩 단계에서 기본적인 보안 조치를 취해야 합니다. 그러나 보안이 아키텍처의 특징으로 자리 잡는 것은 아키텍트가 이를 지원하기 위해 특별한 구조가 필요하다고 판단할 때입니다.

두 가지 일반적인 아키텍처 특징인 보안과 확장성을 고려해 보세요.

아키텍트는 암호화, 해싱, 솔팅과 같은 잘 알려진 기법을 포함한 우수한 코딩 습관을 통해 모놀리식 시스템에서도 보안을 확보할 수 있습니다. (이러한 범주에 속하는 아키텍처 적합성 함수는 6 장에서 다룹니다.) 반대로 마이크로서비스와 같은 분산 아키텍처에서는 아키텍트가 더욱 엄격한 접근 프로토콜을 적용한 강화된 서비스를 구축하는 구조적 접근 방식을 취합니다. 따라서 아키텍트는 설계 또는 구조적 접근 방식을 통해 보안을 확보할 수 있습니다. 한편, 확장성을 고려해 보면 아무리 뛰어난 설계라도 모놀리식 아키텍처는 특정 지점 이상으로 확장할 수 없습니다. 그 지점을 넘어서면 시스템은 분산 아키텍처 스타일로 전환해야 합니다.

건축가들은 기능적인 건축적 특성( 59페이지의 "**기능적인 건축적 특성**"에서 논의됨)에 세심한 주의를 기울입니다. 왜냐하면 이러한 특성들이 특별한 구조적 지원을 가장 자주 필요로 하기 때문입니다.

아키텍처의 특징은 애플리케이션 성공에 매우 중요하거나 필수적이어야 합니다.

애플리케이션은 수많은 아키텍처적 특성을 지원할 수 있지만, 그럴 필요는 없습니다. 시스템이 지원하는 각 아키텍처적 특성은 설계의 복잡성을 증가시키기 때문입니다. 따라서 아키텍트는 최대한 많은 아키텍처적 특성을 선택하기보다는 가능한 한 최소한의 특성만 선택하도록 노력해야 합니다.

우리는 아키텍처적 특성을 명시적 특성과 암묵적 특성으로 구분합니다. 암묵적 특성은 요구사항 문서에 거의 나타나지 않지만, 프로젝트 성공에 필수적입니다. 가용성, 신뢰성, 보안은 거의 모든 애플리케이션의 기반이 되지만, 설계 문서에 명시되는 경우는 드뭅니다. 아키텍트는 문제 영역에 대한 지식을 활용하여 분석 단계에서 이러한 아키텍처적 특성을 파악해야 합니다. 예를 들어, 고빈도 거래 회사의 경우 모든 시스템에 낮은 지연 시간을 명시할 필요가 없을 수 있습니다. 해당 분야의 아키텍트는 이미 낮은 지연 시간이 얼마나 중요한지 알고 있기 때문입니다. 명시적 아키텍처적 특성은 요구사항 문서나 기타 구체적인 지침에 나타납니다.

**그림 4-2**에서 삼각형을 선택한 것은 의도적인 것입니다. 각 정의 요소는 서로를 뒷받침하고, 이는 다시 시스템의 전체 설계를 뒷받침합니다. 삼각형이 만들어내는 받침점은 이러한 아키텍처적 특징들이 어떻게 상호작용하는지를 보여줍니다. 이것이 바로 건축가들이 '상충 관계(trade-o)'라는 용어를 자주 사용하는 이유입니다.

## 건축적 특징 (일부) 목록

아키텍처적 특성은 모듈성 같은 저수준 코드 특성부터 확장성 및 탄력성 같은 정교한 운영상의 문제에 이르기까지 광범위한 복잡성 스펙트럼에 걸쳐 존재합니다. 진정한 의미의 보편적인 표준은 없지만, 이를 정립하려는 시도는 있어왔습니다. 대신 각 조직은 이러한 용어들을 자체적으로 해석합니다. 또한 소프트웨어 생태계는 매우 빠르게 변화하기 때문에 새로운 개념, 용어, 측정 기준 및 검증 방식이 끊임없이 등장하여 아키텍처적 특성을 정의할 새로운 기회를 제공합니다.

건축적 특징의 방대한 양과 폭넓은 범위 때문에 이를 정량화하기는 어렵지만, 건축가들은 이를 범주화합니다. 다음 섹션에서는 이러한 광범위한 범주 몇 가지를 설명하고 몇 가지 예를 제시합니다.

## 운영 아키텍처 특성

운영 아키텍처 특성은 성능, 확장성, 탄력성, 가용성 및 신뢰성과 같은 기능을 포함합니다. 표 4-1은 몇 가지 운영 아키텍처 특성을 나열합니다.

표 4-1. 일반적인 운영 아키텍처 특성

용어	정의
유효성	시스템을 얼마나 오랫동안 가동해야 하는지, 만약 24시간 내내 가동해야 한다면 장애 발생 시 신속하게 시스템을 복구할 수 있는 조치가 마련되어 있어야 합니다.
연속성	시스템의 재해 복구 기능.
성능	시스템 성능은 얼마나 뛰어난가? 이를 측정하는 방법에는 스트레스 테스트, 피크 분석, 기능 사용 빈도 분석, 응답 시간 분석 등이 있습니다.
복구 가능성	비즈니스 연속성 요구사항: 재해 발생 시 시스템이 얼마나 빠르게 복구되어야 하는지. 여기에는 백업 전략 및 예비 하드웨어 요구사항이 포함됩니다.
신뢰성/안전성	시스템이 장애 발생 시에도 안전해야 하는지, 아니면 생명에 직접적인 영향을 미치는 핵심적인 임무 수행 시스템인지 여부. 시스템 장애 발생 시 회사에 막대한 금전적 손실을 초래하는지 여부. 이는 이분법적인 사고방식보다는 다양한 스펙트럼으로 나타나는 경우가 많습니다.
견고성	시스템이 실행 중에 발생하는 오류 및 경계 조건(예: 인터넷 연결 또는 전원 장애)을 처리하는 능력.
확장성	사용자 수 또는 요청 수가 증가함에 따라 시스템의 성능과 작동 능력이 향상됩니다.

운영 아키텍처 특성은 운영 및 DevOps와 상당 부분 겹칩니다.  
우려 사항.

## 구조적 건축 특성

건축가는 적절한 코드 구조를 마련할 책임이 있습니다. 많은 경우, 건축가는 다음과 같은 역할을 수행합니다. 코드의 품질(모듈성, 가독성 포함)에 대한 단독 또는 공동 책임. 성능, 구성 요소 간의 결합이 얼마나 잘 제어되는지, 가독성 있는 코드, 그리고 그 외에도 다양한 내부 품질 평가가 있습니다. 표 4-2에는 몇 가지 구조적 아키텍처 평가 항목이 나열되어 있습니다. 형질.

표 4-2. 구조적 건축 특성

용어	정의
구성 가능성, 확	최종 사용자가 인터페이스를 통해 소프트웨어 구성의 여러 측면을 얼마나 쉽게 변경할 수 있는지.
장성, 설치 용	아키텍처가 기존 기능을 확장하는 변경 사항을 얼마나 잘 수용하는지.
이성, 활용성/	필요한 모든 플랫폼에 시스템을 설치하는 것이 얼마나 쉬운지 보세요.
재사용성: 시스템의 공통 구성 요소를 여러 제품에서 활용할 수 있는 정도.	
현지화	데이터 필드의 입력/조회 화면에서 다국어 지원.
유지보수성	시스템을 변경하고 개선하는 것이 얼마나 쉬운지 보세요.
휴대성	해당 시스템은 오라클 및 SAP DB와 같은 여러 플랫폼에서 실행될 수 있습니다.
업그레이드 가능성	서버와 클라이언트를 최신 버전으로 업그레이드하는 것이 얼마나 쉽고 빠른지 알아보세요.

## 클라우드의 특징

소프트웨어 개발 생태계는 끊임없이 변화하고 발전합니다. 가장 최근의 변화는 다음과 같습니다. 클라우드 컴퓨팅의 등장은 훌륭한 예입니다. 초판이 출간되었을 당시에는 클라우드 기반 컴퓨팅은 존재했지만 널리 보급되지는 않았습니다. 이제 대부분의 시스템에는 어느 정도 클라우드 기반 컴퓨팅 기능이 포함되어 있습니다. 최소한 어느 정도는 클라우드 기반 시스템과의 상호 작용. 몇 가지 예를 들면 다음과 같습니다! 결과는 표 4-3에 나와 있습니다 .

표 4-3. 클라우드 제공업체 아키텍처 특성

용어	정의
온디맨드 확장성: 클라우드 제공업체가 수요에 따라 리소스를 동적으로 확장할 수 있는 능력.	
온디맨드 탄력성: 리소스 수요가 급증할 때 클라우드 제공업체가 발휘하는 유연성. 확장성과 유사합니다.	
영역 기반 가용성: 클라우드 제공업체가 컴퓨팅 영역별로 리소스를 분리하여 더욱 탄력적인 환경을 구축할 수 있는 기능입니다. 시스템.	
지역 기반 개인정보 보호 그리고 보안	클라우드 서비스 제공업체가 다양한 국가 및 지역의 데이터를 저장할 수 있는 법적 권한. 여러 국가 시민들의 데이터가 어디에 저장될 수 있는지에 대한 법률을 가지고 있으며 (종종 저장 자체를 제한하기도 합니다) (그들의 지역 외부).

이 책의 두 번째 개정판에서는 각 건축 분야에 대한 섹션을 추가했습니다. 스타일 관련 장에서는 해당 스타일이 클라우드를 어떻게 수용하고 용이하게 하는지 설명합니다. 고려 사항.

다양한 건축적 특징들 많은 건축적 특징들이 쉽게 알아볼 수 있는 범주에 속하

지만, 그 범주에 속하지 않거나 분류하기 어려운 특징들도 있습니다. 하지만 이러한 특징들 중에서도 중요한 설계 제약 조건과 고 려 사항들이 있습니다. 표 4-4는 이러한 특징들 중 몇 가지를 설명합니다.

표 4-4. 공통적인 건축적 특징

용어	정의
접근성	색맹이나 청각 장애와 같은 장애가 있는 사용자를 포함하여 모든 사용자가 시스템에 얼마나 쉽게 접근할 수 있는지.
아카이빙 가능성	시스템이 특정 기간 후 데이터를 보관하거나 삭제하는 데 있어 가지는 제약 조건.
인증	사용자가 본인임을 확인하기 위한 보안 요구 사항.
권한 부여	사용자가 애플리케이션 내의 특정 기능에만 접근할 수 있도록 보장하는 보안 요구 사항(사용 사례, 하위 시스템, 웹 페이지, 비즈니스 규칙, 필드 수준 등 기준).
합법적인	시스템이 운영되는 법적 제약 조건에는 GDPR과 같은 데이터 보호법이나 미국의 사베인스-옥슬리법과 같은 금융 기록법, 또는 애플리케이션 구축 및 배포 방식과 관련된 규정 등이 포함됩니다. 여기에는 회사가 필요로 하는 예약 권한도 포함됩니다.
은둔	이 시스템은 회사 내부 직원, 심지어 데이터베이스 관리자(DBA)와 네트워크 설계자에게까지 가려 내역을 암호화하고 숨길 수 있는 기능을 갖추고 있습니다.
보안	데이터베이스 또는 내부 시스템 간 네트워크 통신에 대한 암호화 규칙 및 제약 조건, 원격 사용자 접근을 위한 인증 및 기타 보안 조치.
지원 가능성	해당 애플리케이션에 필요한 기술 지원 수준, 시스템 오류 디버깅에 필요한 로깅 및 기타 기능의 범위.
사용성/ 실현 가능성	사용자가 애플리케이션/솔루션을 통해 목표를 달성하기 위해 필요한 교육 수준.

아키텍처 특성 목록은 필연적으로 불완전할 수밖에 없습니다. 각 소프트웨어 프로젝트는 고유한 요소에 따라 새로운 아키텍처 특 성을 만들어낼 수 있기 때문입니다. 앞서 언급한 용어들 중 상당수는 미묘한 뉘앙스나 객관적인 정의의 부족으로 인해 모호하고 부 정확한 경우가 많습니다. 예를 들어, 상호 운용성과 호환성은 동등해 보일 수 있으며, 일부 시스템에서는 실제로 그렇습니다. 그러 나 상호 운용성은 다른 시스템과의 손쉬운 통합을 의미하며, 이는 공개되고 문서화된 API를 전제로 합니다. 반면 호환성은 업계 및 도메인 표준과 더 관련이 있습니다. 또 다른 예로 학습 용이성을 들 수 있습니다. 한 가지 정의는 "사용자가 소프트웨어 사용법 을 배우는 것이 얼마나 쉬운가"이고, 다른 정의는 "시스템이 마신 러닝 알고리즘을 사용하여 자체 구성 또는 자체 최적화를 위해 주변 환경에 대해 자동으로 학습할 수 있는 수준"입니다.

가용성과 신뢰성처럼 서로 겹치는 정의들이 많습니다. 하지만 TCP의 기반이 되는 인터넷 프로토콜인 IP를 생각해 보세요. IP는 가 용성은 높지만 신뢰성은 떨어집니다. 패킷이 순서대로 도착하지 않을 수 있고, 수신자는 누락된 패킷을 다시 요청해야 할 수도 있 습니다.

이러한 범주를 정의하는 표준의 완전한 목록은 없습니다. 국제표준화기구(ISO)는 **기능별로 정리된 목록**을 발표했는데, 이는 여기 있는 목록과 일부 겹치지만, 주로 불완전한 범주 목록을 제공합니다. 다음은 ISO 정의 중 일부를 현대적인 관심사에 맞춰 용어를 수정하고 범주를 추가한 것입니다.

#### 성능 효율성은 알려진 조건

에서 사용된 자원량 대비 성능을 측정하는 지표입니다. 여기에는 시간적 특성(응답 시간, 처리 시간 및/또는 처리량), 자원 활용도(사용된 자원의 양과 종류), 용량(설정된 최대 한계를 초과하는 정도)이 포함됩니다.

#### 호환성은 제품,

시스템 또는 구성 요소가 다른 제품, 시스템 또는 구성 요소와 정보를 교환하고 동일한 하드웨어 또는 소프트웨어 환경을 공유하면서 필요한 기능을 수행할 수 있는 정도를 나타냅니다. 여기에는 공존성(다른 제품과 공통 환경 및 리소스를 공유하면서 필요한 기능을 효율적으로 수행할 수 있는 능력)과 상호 운용성(둘 이상의 시스템이 정보를 교환하고 활용할 수 있는 정도)이 포함됩니다.

#### 사용성이

란 사용자가 시스템을 의도된 목적에 맞게 효과적이고 효율적이며 만족스럽게 사용할 수 있도록 하는 것을 의미합니다. 사용성에는 적합성, 인지성(사용자가 소프트웨어가 자신의 요구에 적합한지 인지할 수 있는 능력), 학습 용이성(사용자가 소프트웨어 사용법을 얼마나 쉽게 배울 수 있는지), 사용자 오류 방지(사용자의 오류를 방지하는 능력), 접근성(다양한 특성과 능력을 가진 사람들이 소프트웨어를 이용할 수 있도록 하는 능력)이 포함됩니다.

#### 신뢰성은 시

스템이 지정된 조건에서 지정된 기간 동안 제대로 작동하는 정도를 나타내는 특성입니다. 이 특성에는 성숙도(소프트웨어가 정상 작동 조건에서 신뢰성 요구 사항을 충족하는지 여부), 가용성(소프트웨어가 작동 가능하고 접근 가능한지 여부), 내결함성(하드웨어 또는 소프트웨어 오류에도 불구하고 소프트웨어가 의도한 대로 작동하는지 여부), 복구 가능성(소프트웨어가 영향을 받은 데이터를 복구하고 시스템의 원하는 상태를 복원하여 오류로부터 복구할 수 있는지 여부)과 같은 하위 범주가 포함됩니다.

#### 보안 수준

은 소프트웨어가 정보와 데이터를 보호하는 정도를 나타내며, 이를 통해 사람, 다른 제품 또는 시스템은 각자의 유형과 권한 수준에 맞는 데이터 접근 권한을 갖게 됩니다. 이러한 특성에는 기밀성(데이터는 접근 권한이 있는 사람만 접근할 수 있음), 무결성(소프트웨어가 무단 접근이나 수정을 방지함) 등이 포함됩니다.

부인 방지(행동이나 사건이 발생했음을 증명할 수 있음), 책임성(사용자의 행동을 추적할 수 있음), 그리고 진위성(사용자의 신원을 증명할 수 있음)이 있습니다.

#### 유지보수성은 개발자

가 소프트웨어를 개선, 수정하거나 환경 및/또는 요구 사항의 변화에 적응시키기 위해 소프트웨어를 얼마나 효과적이고 효율적으로 수정할 수 있는지를 나타내는 정도입니다. 이 특성에는 모듈성(소프트웨어가 얼마나 독립적인 구성 요소로 이루어져 있는지), 재사용성(개발자가 하나의 자산을 여러 시스템에서 사용하거나 다른 자산을 구축하는 데 사용할 수 있는 정도), 분석성(개발자가 소프트웨어에 대한 구체적인 지표를 얼마나 쉽게 수집할 수 있는지), 수정성(개발자가 결함을 발생시키거나 기존 제품 품질을 저하시키지 않고 소프트웨어를 수정할 수 있는 정도), 테스트 용이성(개발자 및 다른 사람이 소프트웨어를 얼마나 쉽게 테스트할 수 있는지)이 포함됩니다.

#### 이식성이란 개

발자가 시스템, 제품 또는 구성 요소를 하나의 하드웨어, 소프트웨어 또는 기타 운영/사용 환경에서 다른 환경으로 이전할 수 있는 정도를 나타냅니다.

이 특성에는 적응성(개발자가 다양하거나 진화하는 하드웨어, 소프트웨어 또는 기타 운영/사용 환경에 소프트웨어를 효과적이고 효율적으로 적용할 수 있는지 여부), 설치 용이성(지정된 환경에 소프트웨어를 설치 및/또는 제거할 수 있는지 여부) 및 교체 용이성(개발자가 다른 소프트웨어로 기능을 얼마나 쉽게 교체할 수 있는지)과 같은 하위 특성이 포함됩니다.

ISO 목록의 마지막 항목은 소프트웨어의 기능적 측면을 다룹니다.

#### 기능적 적합성 이 특성은 제

품 또는 시스템이 특정 조건에서 사용될 때 명시적 및 묵시적 요구 사항을 충족하는 기능을 제공하는 정도를 나타냅니다. 이 특성은 다음과 같은 하위 특성으로 구성됩니다.

##### 기능적 완전성: 기능 집합이 명

시된 모든 작업과 사용자 목표를 포괄하는 정도.

##### 기능적 정확성은 제품이나 시

스템이 필요한 정밀도로 정확한 결과를 제공하는 정도를 나타냅니다.

##### 기능적 적합성

해당 기능들이 명시된 과제와 목표 달성을 얼마나 용이하게 하는지.

하지만 기능적 적합성은 이 목록에 포함되어서는 안 된다고 생각합니다. 기능적 적합성은 아키텍처적 특성을 설명하는 것이 아니라 소프트웨어 구축을 위한 동기 부여 요건을 나타내기 때문입니다. 이는 아키텍처적 특성과 문제 영역 간의 관계에 대한 사고방식이 어떻게 진화해 왔는지를 보여줍니다. 이러한 진화 과정은 7 장에서 자세히 다룹니다.

## 소프트웨어 아키텍처의 수많은 모호성

아키텍트들이 지속적으로 불만을 느끼는 부분 중 하나는 소프트웨어 아키텍처 활동 자체를 포함하여 수많은 핵심 요소에 대한 명확한 정의가 부족하다는 점입니다! 표준이 없기 때문에 기업들은 공통적인 개념에 대해 자체적인 용어를 정의하게 됩니다.

이로 인해 업계 전반에 걸쳐 혼란이 발생하는 경우가 많은데, 건축가들이 모호한 용어를 사용하거나, 더 심각하게는 완전히 다른 의미로 같은 용어를 사용하기 때문입니다.

소프트웨어 개발 세계에 표준 용어를 강요할 수는 없지만, 용어로 인한 오해를 방지하기 위해 도메인 주도 설계(DDD)에서는 조직 구성원 간에 공통 언어를 정립하고 사용하도록 권장합니다. 저희 또한 이러한 권장 사항을 따르고 있습니다.

## Trade-Offs 및 최소최악 아키텍처

앞서 언급했듯이 아키텍트는 시스템의 성공에 필수적이거나 중요한 아키텍처적 특징만 지원해야 합니다. 시스템은 여러 가지 이유로 앞서 나열한 아키텍처적 특징 중 일부만 지원할 수 있습니다. 첫째, 아키텍처적 특징을 지원하는 데에는 비용이 많이 듭니다. 지원하는 각 특징에는 아키텍트의 설계 노력, 개발자의 구현 및 유지 관리 노력, 그리고 구조적 지원이 필요할 수 있습니다.

둘째, 아키텍처적 특성은 서로 그리고 문제 영역과 시너지 효과를 냅니다. 우리가 바라는 바와는 달리, 각 설계 요소는 다른 모든 요소와 상호 작용합니다. 예를 들어, 보안을 강화하기 위한 조치를 취하면 성능에 부정적인 영향을 미칠 가능성이 매우 높습니다. 애플리케이션은 더 많은 즉석 암호화, (비밀 정보를 숨기기 위한) 간접 참조 및 기타 성능 저하를 유발할 수 있는 작업을 수행해야 하기 때문입니다.

비행기 조종사들은 양손과 양발에 각각 조종 장치가 있는 헬리콥터 조종을 배우는 데 어려움을 겪는 경우가 많습니다. 하나의 조종 장치를 바꾸면 다른 모든 조종 장치에도 영향을 미치는데, 이는 모든 장치가 서로 연동되어 있기 때문입니다. 헬리콥터 조종은 균형을 잡는 작업이며, 이는 건축적 특징을 선택할 때 발생하는 절충 과정을 잘 설명해 줍니다. 건축적 특징들은 다른 특징들, 그리고 전체적인 설계와도 밀접하게 연관되어 있어 하나를 바꾸면 다른 하나도 바뀌어야 하는 경우가 많습니다. 마치 고된 비행을 하는 헬리콥터 조종사처럼, 건축가들도 서로 맞물린 요소들을 능숙하게 조율하는 법을 배워야 합니다.

셋째, 앞서 논의했듯이 아키텍처 특성에 대한 표준 정의가 부족하여 조직들이 모호함에 어려움을 겪고 있습니다. 업계에서는 이러한 문제가 더욱 두드러집니다.

아키텍처 특성에 대한 불변의 목록을 만드는 것은 결코 불가능하지만(새로운 특성이 끊임없이 나타나기 때문에), 각 조직은 객관적인 정의를 담은 자체 목록(또는 보편적인 언어)을 만들 수 있습니다.

마지막으로, 아키텍처 특성의 수가 지속적으로 증가할 뿐만 아니라, 지난 10년 동안 범주의 수도 증가했습니다. 예를 들어, 몇십 년 전만 해도 아키텍트는 운영 문제를 거의 고려하지 않았고, 이를 별개의 "블랙박스"로 여겼습니다. 그러나 마이크로서비스와 같은 아키텍처가 인기를 얻으면서 아키텍트와 운영팀은 더욱 긴밀하고 빈번하게 협력해야 합니다. 소프트웨어 아키텍처가 복잡해질수록 조직의 다른 부분과도 밀접하게 연관되는 경향이 있습니다.

따라서 건축가가 시스템을 설계하면서 모든 건축적 특성을 극대화하는 경우는 매우 드뭅니다. 대부분의 경우, 여러 상충되는 고려 사항 사이에서 절충점을 찾아야 합니다.



최고의 건축물을 추구하지 말고, 최악의 건축물을 피하는 것을 목표로 삼아라.

너무 많은 아키텍처적 특성을 지원하려고 하면 모든 비즈니스 문제를 해결하려는 일반적인 솔루션으로 이어집니다. 이러한 아키텍처는 금방 다루기 어려워지기 때문에 제대로 작동하는 경우가 드뭅니다.

가능한 한 반복적인 아키텍처를 설계하도록 노력하십시오. 아키텍처를 변경하기 쉬울수록 모든 사람이 첫 시도에 완벽한 것을 찾아야 한다는 부담감을 덜 느끼게 됩니다. 애자일 소프트웨어 개발의 가장 중요한 교훈 중 하나는 반복의 가치이며, 이는 아키텍처를 포함한 소프트웨어 개발의 모든 단계에 적용됩니다.

