제24장

효과적인 팀 만들기

소프트웨어 아키텍트는 기술 아키텍처를 설계하고 아키텍처 관련 결정을 내리는 것 외에도 개발팀을 이끌고 아키텍처 구현 과정을 안내하는 책임도 맡습니다. 이러한 역할을 잘 수행하는 아키텍트는 개발팀이 긴밀하게 협력하여 문제를 해결하고 성공적인 솔루션을 만들어낼 수 있도록 효과적인 팀을 구축합니다. 당연한 이야기처럼 들리겠지만, 많은 아키텍트들이 개발팀을 소홀히 하고 고립된 환경에서 혼자 아키텍처를 설계하는 경우가 너무나 많습니다. 이렇게 설계된 아키텍처를 개발팀에 넘겨주면 개발자들은 제대로 구현하는 데 어려움을 겪는 경우가 흔합니다.

팀의 생산성을 높이는 것은 성공적인 소프트웨어 아키텍트가 차별화되는 요소 중 하나입니다. 이 장에서는 개발 팀의 효율성을 향상시키는 몇 가지 기본 기법을 소개합니다.

협동

소프트웨어 업계에서는 아키텍처와 개발을 완전히 별개의 활동으로 취급하는 경우가 너무나 많습니다. 그림 24-1을 보면 전통적인 아키텍트와 개발자의 책임 범위를 비교해 볼 수 있습니다. 아키텍트는 비즈니스 요구사항을 분석하여 아키텍처 특성을 추출 및 정의하고, 문제 영역을 해결하기 위한 아키텍처 패턴과 스타일을 선택하며, 논리적 구성 요소를 생성하는 등의 활동을 담당합니다. 개발팀은 아키텍트가 이러한 활동을 통해 생성한 산출물을 활용하여 구성 요소의 클래스 다이어그램을 작성하고, UI 화면을 구축하며, 소스 코드를 작성하고 테스트합니다.

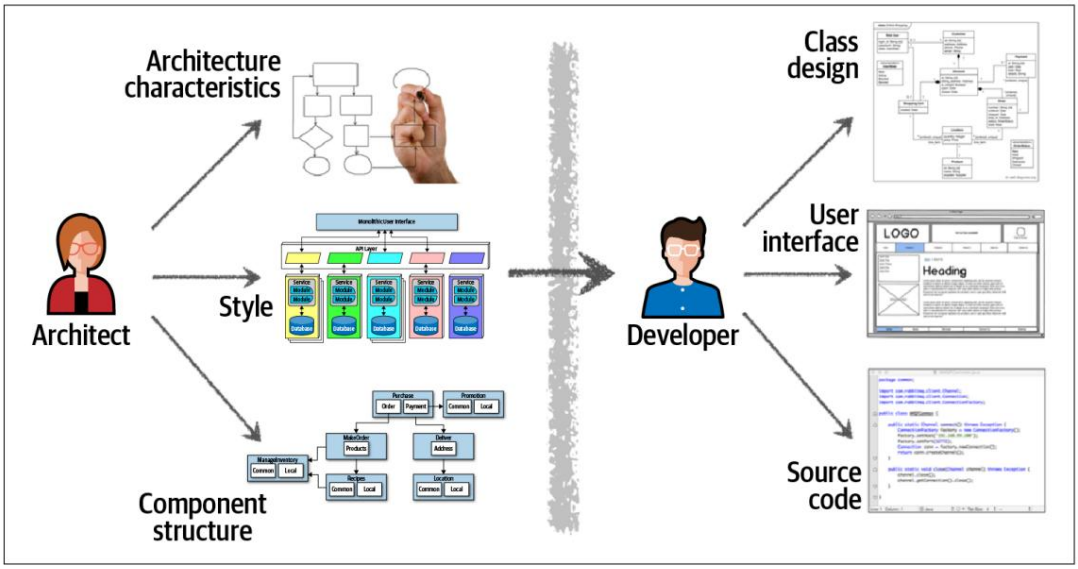

그림 24-1. 건축가와 개발자의 전통적인 역할 비교

그림 24-1 은 이러한 전통적인 아키텍처 접근 방식이 왜 제대로 작동하지 않는지 보여줍니다. 아키텍트와 개발자를 분리하는 가상 및 물리적 장벽을 관통하는 일방향 화살표를 보십시오. 바로 이것이 모든 문제의 원인입니다. 아키텍트의 결정이 개발팀에 제대로 전달되지 않는 경우가 많고, 개발팀이 아키텍처를 변경하더라도 다시 아키텍트에게 돌아오는 경우는 드뭅니다. 이 모델에서는 아키텍트와 개발팀이 너무 단절되어 있기 때문에 아키텍처가 목표를 달성하는 경우가 거의 없습니다.

건축 프로젝트를 성공적으로 이끌어가는 핵심은 건축가와 개발자 사이의 물리적, 가상적 장벽을 허물고 강력한 양방향 협력 관계를 구축하는 것입니다. 건축가와 개발팀은 그림 24-2에 나타난 협업 모델처럼 동일한 가상 팀에 속해야 합니다.

이 모델은 강력한 양방향 소통과 협업을 촉진할 뿐만 아니라, 설계자가 개발자들을 멘토링하고 코칭할 수 있도록 해줍니다.

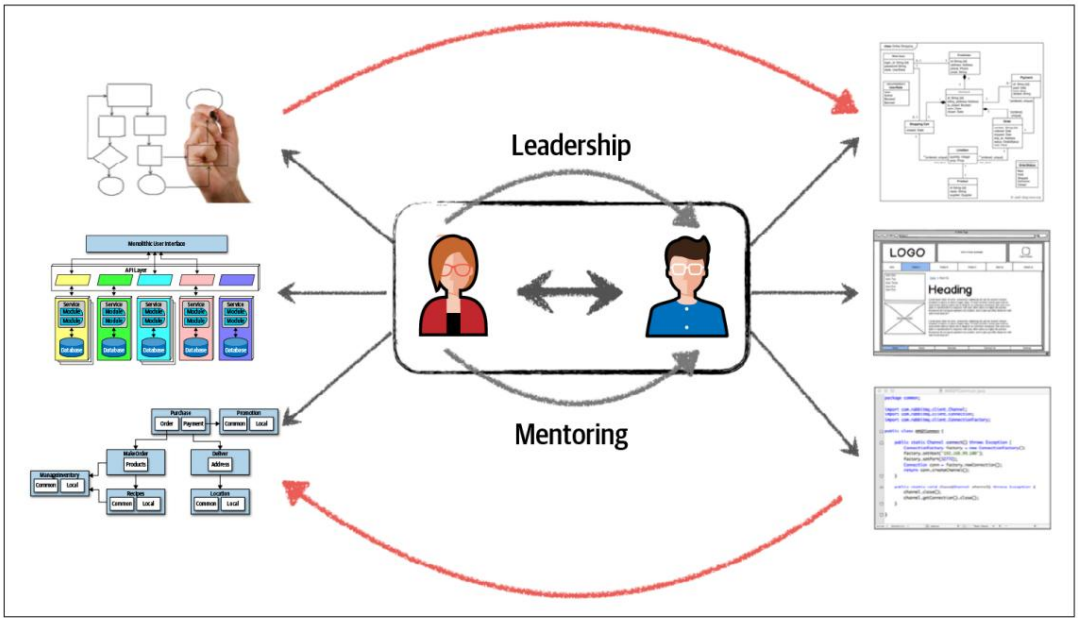

그림 24-2. 협업을 통해 건축을 기능적으로 만드는 방법

정적이고 경직된 구식 폭포수 모델과는 달리, 오늘날의 소프트웨어 아키텍처는 제품 개발의 거의 모든 반복 또는 단계마다 변화하고 발전합니다.
건축가와 개발팀 간의 긴밀한 협력은 필수적입니다.
성공.

이 장의 나머지 부분과 25장 에서는 개발 팀의 효율성을 높일 뿐만 아니라 더욱 견고하고 성공적인 아키텍처를 구축하는 데 도움이 되는 건전하고 양방향적인 협력 관계를 형성하는 기술을 보여드리겠습니다.

제약 조건 및 경계

저희 경험상 소프트웨어 아키텍트는 개발팀의 성공과 실패에 상당한 영향을 미칠 수 있습니다. 아키텍트와 소통이 원활하지 않거나 정보 공유에서 소외감을 느끼는 팀은 시스템의 다양한 제약 조건에 대한 지식이 부족한 경우가 많습니다. 적절한 지침 없이는 아키텍처를 제대로 구현하는 데 어려움을 겪습니다.

소프트웨어 아키텍트의 역할 중 하나는 개발자가 아키텍처를 구현해야 하는 제약 조건을 만들고 전달하는 것입니다. 이러한 제약 조건은 개발팀이 아키텍처를 구현하는 "공간"을 형성한다고 생각하면 됩니다. 그림 24-3 에서 볼 수 있듯이, 제약 조건이 너무 좁거나 너무 넓으면 팀이 아키텍처를 성공적으로 구현하는 데 직접적인 지장을 초래합니다.

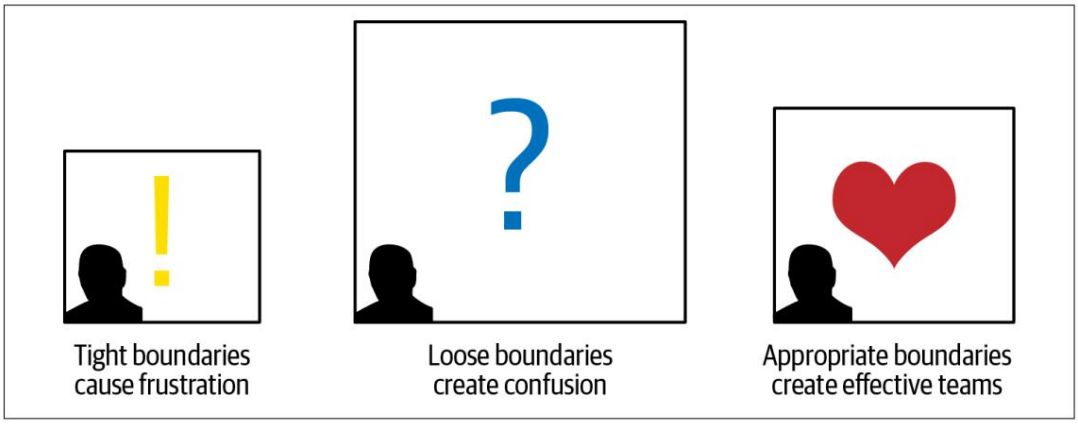

그림 24-3. 소프트웨어 아키텍트가 만든 경계는 팀의 아키텍처 구현 능력에 영향을 미칩니다.

지나친 제약 조건은 개발팀에게 공간을 너무 좁게 만들어 시스템 구현에 필요한 많은 도구, 라이브러리 및 모범 사례에 접근하지 못하게 합니다. 이는 좌절감을 유발하고, 대개 개발자들이 더 나은 환경을 찾아 프로젝트를 떠나는 결과를 낳습니다.

반대의 경우도 발생할 수 있습니다. 제약 조건이 너무 느슨하거나 아예 없는 경우, 공간이 너무 넓어집니다. 이 경우 선택지가 너무 많아 개발팀이 사실상 건축가의 역할을 떠맡아 모든 중요한 아키텍처 결정을 내려야 합니다. 적절한 지침이 없으면 개발자들은 개념 증명을 너무 많이 수행하고, 설계 결정에 어려움을 겪으며, 생산성이 저하되고 혼란스러워지며 좌절감을 느끼게 됩니다.

유능한 소프트웨어 아키텍트는 팀이 필요한 모든 것을 갖출 수 있도록 적절한 수준의 지침과 제약 조건을 제공하기 위해 노력합니다. 이 장의 나머지 부분에서는 이러한 적절한 경계를 설정하는 방법을 설명합니다.

건축가들의 개성

개념의 명확성을 위해 다소 일반화된 설명을 하자면, 건축가의 성격은 크게 세 가지 유형으로 나눌 수 있습니다. 바로 통제광형 건축가, 이론형 건축가, 그리고 실용형 건축가입니다. 통제광형 건축가는 엄격한 경계를 설정하는 경향이 있고, 이론형 건축가는 유연한 경계를 설정하는 경향이 있으며, 실용형 건축가는 적절한 경계를 설정합니다. 다음 절에서는 각 건축가 성격 유형에 대한 자세한 설명을 제공합니다.

통제광 건축가

통제광 아키텍트는 소프트웨어 개발 과정의 모든 세부 사항을 통제하려고 합니다. 그들이 내리는 모든 결정은 대개 지나치게 세밀하고 저수준적이어서 개발팀에 엄격한 제약과 과도한 부담을 안겨 줍니다.

예를 들어, 통제광 아키텍트는 개발팀이 유용하거나 심지어 필수적인 오픈 소스 또는 타사 라이브러리를 다운로드하는 것을 제한하거나, 명명 규칙, 클래스 설계, 메서드 길이 등에 엄격한 제약을 둘 수 있습니다. 심지어 개발팀이 구현해야 할 의사 코드를 작성하여 개발자로부터 프로그래밍의 기술을 사실상 빼앗아 갈 수도 있습니다.

개발자들은 이러한 점에 불만을 느끼고 종종 설계자에 대한 존경심을 잃습니다.

안타깝게도, 특히 소프트웨어 개발자에서 아키텍트로 전향하는 경우, 통제광 아키텍트가 되기 쉽습니다. 아키텍트의 역할은 애플리케이션의 구성 요소(논리적 요소)를 만들고 이러한 요소들이 어떻게 상호 작용하는지 결정하는 것입니다. 개발자의 역할은 클래스 다이어그램과 디자인 패턴을 사용하여 이러한 논리적 구성 요소를 가장 효과적으로 구현하는 방법을 결정하는 것입니다. 개발자로서 직접 클래스 다이어그램을 작성하고 디자인 패턴을 선택하는 데 익숙한 신입 아키텍트들은 이러한 유혹을 뿌리치기 어려워하는 경우가 많습니다.

예를 들어, 아키텍트가 시스템 내 참조 데이터를 관리하는 논리적 구성 요소를 만든다고 가정해 보겠습니다. 웹사이트에서 사용되는 정적 이름-값 쌍 데이터, 제품 코드, 창고 코드 등이 여기에 해당합니다. 아키텍트의 역할은 이 논리적 구성 요소(이 경우 참조 관리자)를 식별하고, 핵심 연산(예: GetData, SetData, ReloadCache, NotifyOnUpdate)을 결정하며, 참조 관리자 와 상호 작용해야 하는 다른 구성 요소를 파악하는 것입니다 . 완벽주의자 아키텍트는 특정 데이터 구조를 가진 내부 캐시를 활용하는 병렬 로더 패턴이 이 구성 요소를 구현하는 가장 좋은 방법이라고 생각할 수 있습니다. 이는 효과적인 설계일 수 있지만 유일한 설계는 아니며, 더 중요한 것은 참조 관리자의 내부 설계를 구상하는 것은 아키텍트의 역할이 아니라 개발자의 역할이라는 점입니다.

이 장에서 자세히 살펴보겠지만, 프로젝트의 복잡성과 팀의 숙련도에 따라 건축가는 때때로 통제광의 역할을 해야 할 필요가 있습니다.
하지만 대부분의 경우, 통제광 아키텍트는 개발팀의 흐름을 방해하고, 적절한 지침을 제공하지 못하며, 일을 가로막고, 전반적으로 리더로서 효과적이지 못합니다.

안락의자 건축가

안락의자 건축가는 코딩 경험이 거의 없거나 아예 없는 건축가로, 아키텍처를 설계할 때 구현 세부 사항을 고려하지 않습니다. 이들은 일반적으로 개발팀과 단절되어 있으며, 초기 아키텍처 다이어그램만 완성하고 다음 프로젝트로 넘어가는 경우가 많습니다.

일부 자칭 아키텍트들은 역량 부족으로 제대로 된 리더십이나 지침을 제공하지 못합니다. 기술이나 비즈니스 영역에 대한 이해가 너무 부족하죠. 생각해 보세요. 개발자는 뭘 할까요? 당연히 소스 코드를 작성합니다. 소스 코드 작성 능력은 흉내내기가 정말 어렵습니다. 할 수 있거나 못하거나 둘 중 하나죠. 그럼 아키텍트는 뭘 할까요? 아무도 정확히 모릅니다! 선과 궤적을 많이 그리는 걸까요? 아키텍트인 척하기는 너무 쉽죠.

예를 들어, 주식 거래 시스템을 설계하는 아마추어 건축가가 자신의 역량을 훨씬 뛰어넘는 일을 맡았다고 가정해 봅시다. 그 건축가가 그린 아키텍처 다이어그램에는 거래 시스템을 나타내는 상자 하나와 거래 시스템과 통신하는 거래 규정 준수 엔진을 나타내는 상자 하나만 있을 수 있습니다. 이 아키텍처 자체에 문제가 있는 것은 아니지만, 너무 개략적이어서 누구에게도 도움이 되지 않습니다.

자기주장이 강한 설계자들은 개발팀 주변에 느슨한 경계를 만들어 결국 설계자가 해야 할 일을 개발팀이 하게 됩니다. 그 결과 개발 속도와 생산성이 저하되고, 시스템 작동 방식에 대한 혼란이 발생합니다.

방구석 건축가가 되는 것은 통제광이 되는 것만큼이나 쉽습니다. 건축가가 아키텍처를 구현하는 개발팀에 시간을 할애하지 않거나(혹은 아예 시간을 내지 않으려 한다면), 이는 그들이 방구석 건축가 성향에 빠지고 있다는 신호일 수 있습니다.

개발팀은 건축가의 지원과 지침이 필요하며, 건축가가 질문에 답변할 수 있도록 항상 대기해야 합니다. '탁상공론만 하는 건축가'의 다른 특징은 다음과 같습니다.

- 비즈니스 영역, 비즈니스 문제 또는 기술을 제대로 이해하지 못함
  - 사용 중인 방

식이 • 소프트웨어 개발에 대한 실무 경험 부족 • 특정 아키텍처 솔루션 구현

과 관련된 영향(복잡성, 유지 관리 및 테스트 등)을 고려하지 않음

건축가들이 의도적으로 탁상공론만 하는 건축가가 되는 경우는 드뭅니다. 프로젝트나 팀에 너무 많은 시간을 쏟다 보면 기술이나 비즈니스 영역과의 연결고리를 잃어버리는 경우가 종종 발생합니다. 이를 방지하기 위해 프로젝트의 기술에 더 적극적으로 참여하고 비즈니스 문제와 영역에 대한 깊이 있는 이해를 쌓는 것이 좋습니다.

효과적인 건축가

유능한 소프트웨어 아키텍트는 적절한 제약 조건과 경계를 설정하고, 팀 구성원들이 원활하게 협력하도록 보장하며, 적절한 수준의 지침을 제공합니다. 또한 유능한 아키텍트는 팀이 필요한 모든 것을 갖추도록 합니다.

적절한 도구와 기술을 도입하고 개발팀과 목표 달성 사이에 있는 모든 장애물을 제거합니다.

언뜻 보기엔 당연하고 쉬워 보이지만, 사실은 그렇지 않습니다. 효과적인 리더가 되는 것, 그리고 효과적인 소프트웨어 아키텍트가 되는 데에는 기술이 필요합니다. 개발팀과 긴밀히 협력하고 그들의 존경을 얻는 것이 핵심입니다. 다음 섹션에서는 아키텍트가 개발팀과 어느 정도 관여해야 하는지를 판단하는 몇 가지 기법을 소개합니다.

어느 정도의 참여가 필요합니까?

효과적인 아키텍트가 되려면 개발팀에 얼마나 관여해야 하는지, 그리고 언제 그들의 업무에 간섭하지 말아야 하는지를 아는 것이 중요합니다. 로이 오셰로브(Roy Osherove) 작가이자 컨설턴트가 널리 전파한 개념이 바로 '탄력적 리더십(Elastic Leadership)' 입니다.
우리는 이 분야에서 오셰로브의 연구와는 다소 다른 방향으로 나아가 소프트웨어 아키텍처 리더십에 특정한 요소들에 집중할 것입니다.

개발팀에 어느 정도 관여해야 할지, 그리고 동시에 몇 개의 팀이나 프로젝트를 관리할 수 있을지 결정하는 것은 어려울 수 있습니다. 다음은 고려해야 할 다섯 가지 핵심 요소입니다.

팀 구성원 간의 친
: 밀도: 팀 구성원들은 서로를 얼마나 잘 알고 있나요? 이전에 함께 일해 본 경험이 있나요? 일반적으로 팀 구성원들이 서로를 잘 알수록 자율적으로 조직화할 수 있고 아키텍트의 개입이 줄어듭니다. 반대로 팀 구성원이 새로 합류했을수록 아키텍트의 역할이 더욱 중요해지고 파벌 형성을 방지하는 데 도움이 됩니다.

팀 규모 저희는
: 개발자가 12명 이상인 팀을 큰 팀으로, 5명 이하인 팀을 작은 팀으로 간주합니다. 팀 규모가 클수록 아키텍트의 역할이 더욱 중요해집니다. 이 주제에 대해서는 443페이지의 "팀 경고 신호"에서 더 자세히 다룹니다.

전반적인 경험 측면에
: 서, 팀의 숙련된 개발자와 경험이 부족한 개발자의 비율은 어떻게 되나요? 팀원들은 해당 기술과 비즈니스 도메인에 대해 얼마나 잘 알고 있나요? (비즈니스 도메인이 특히 복잡한 경우, 팀원들의 전반적인 기술 경험 수준과 비즈니스 도메인 경험을 별도로 평가하는 것을 고려해야 합니다.) 주니어 개발자가 많은 팀은 아키텍트의 더 많은 참여와 멘토링이 필요합니다. 반면, 숙련된 개발자가 많은 팀에서는 아키텍트가 멘토보다는 조력자 역할을 더 많이 수행할 수 있습니다.

프로젝트 복잡성이
: 높을수록 건축가는 문제 해결을 위해 더 많은 시간을 할애해야 하지만, 비교적 단순하고 명확한 프로젝트는 건축가의 참여가 덜 필요합니다.

프로젝트 기간은
: 짧은 프로젝트(예: 2개월), 긴 프로젝트(2년), 또는 평균 기간(약 6개월) 중 어떤 유형인가요? 프로젝트 기간이 길어질수록 건축가의 참여도 또한 높아집니다.

이러한 요소들 대부분은 당연해 보일 수 있지만, 프로젝트 기간은 때때로 혼란을 야기합니다. 앞서 언급했듯이 프로젝트 기간이 짧을수록 아키텍트의 참여는 줄어들고, 프로젝트 기간이 길수록 더 많은 참여가 필요합니다. 이는 직관과 어긋나는 것처럼 들리나요? 두 달짜리 프로젝트를 생각해 보세요. 두 달은 요구사항을 검증하고, 실험하고, 코드를 개발하고, 모든 시나리오를 테스트하고, 프로덕션 환경에 배포하기에는 충분한 시간이 아닙니다. 이 경우 아키텍트는 마치 전문가처럼 실무를 구상하는 역할을 하는 것이 좋습니다. 개발팀은 이미 프로젝트에 대한 긴박감을 느끼고 있으며, 모든 것을 통제하려는 아키텍트는 오히려 방해가 되어 프로젝트를 지연시킬 뿐입니다. 이제 2년짜리 프로젝트를 생각해 보세요. 개발자들은 훨씬 여유롭고 긴박감을 느끼지 않습니다.

그들은 휴가 계획을 세우거나 긴 점심시간을 보내는 데 시간을 보낼 가능성이 높습니다. 따라서 장기 프로젝트의 경우, 건축가는 프로젝트가 일정대로 진행되고 팀이 가장 복잡한 작업을 먼저 완료하도록 관리해야 합니다.

이러한 요소를 사용하여 적절한 참여 수준을 결정하는 방법을 설명하기 위해 그림 24-4에 표시된 것처럼 각 요소에 대해 20점의 고정 척도를 가정합니다.
척도의 음수 값은 참여도가 낮음을 나타내며, 극단적으로는 방관자형 건축가로 이어집니다. 양수 값은 참여도가 높음을 나타내며, 극단적으로는 모든 것을 통제하려는 건축가로 이어집니다.

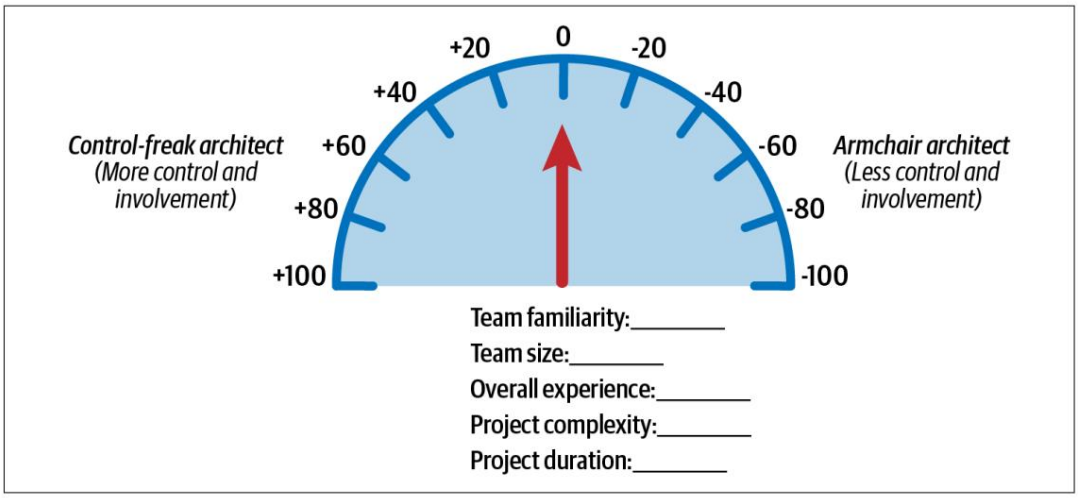

그림 24-4. 건축가의 개발 참여 정도를 측정하는 척도

팀

물론 이 척도는 정확하지는 않지만, 양을 가늠하는 데 도움이 됩니다.
건축가가 개발팀과 어떤 관계를 맺어야 하는지 예상해야 합니다. 예를 들어!
표 24-1 과 그림 24-5 에 제시된 프로젝트 시나리오, 즉 시나리오 1을 고려해 주십시오 .
표에 있는 각 요소의 값은 "바늘"을 양극단 중 쪽으로 움직입니다.
참여도가 높을수록 +20점, 참여도가 낮을수록 -20점을 부여합니다.
참여도. 시나리오 1에 대해 평가된 요인 점수의 총합은 -60으로, 이는 다음과 같은 의미를 나타냅니다.
건축가는 일상적인 상호 작용에 대한 관여를 제한하고, 원활한 진행을 돕되 직접적인 개입은 자제해야 합니다.
팀의 업무에 방해가 되지 않도록 해야 합니다. 질문에 답변하고 필요한 조치를 취해야 합니다.
팀은 계획대로 진행하고 있지만, 대부분의 경우 건축가는 크게 관여하지 않아야 합니다.
경험이 풍부한 팀이 가장 잘하는 일, 즉 소프트웨어를 빠르게 개발하는 데 집중하도록 맡기세요.

표 24-1. 참여 정도에 대한 시나리오 1 예시

요인	값	평가 성격	
팀과의 친숙도 신규 팀원 +20 통제광			
팀 규모	소규모 (4인) -20명		안락의자에 앉아 건축가 노릇을 해보세요.
전반적인 경험 모든 경험자		-20	안락의자에 앉아 건축가 노릇을 해보세요.
프로젝트 난이도는 비교적 간단합니다.		-20	안락의자에 앉아 건축가 노릇을 해보세요.
프로젝트 기간	2개월	-20	안락의자에 앉아 건축가 노릇을 해보세요.
누적 점수		-60	안락의자에 앉아 건축가 노릇을 해보세요.

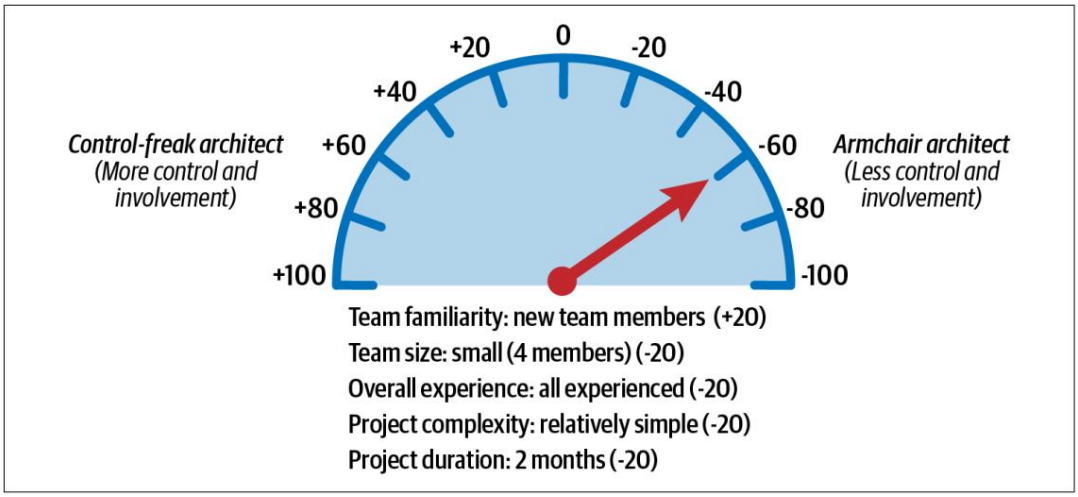

그림 24-5. 시나리오 1의 참여 정도

이제 표 24-2 에 설명되어 있고 그림 24-6 에 나타낸 시나리오 2를 고려해 보겠습니다.
팀 구성원들은 서로 잘 알고 있지만, 팀 규모가 큰 경우(12명)
팀 구성원은 소수이며, 대부분 주니어 개발자로 이루어져 있습니다. 프로젝트는 비교적 복잡합니다.
기간은 6개월입니다. 이 경우 누적 점수는 +20점이 됩니다.
이는 유능한 건축가는 멘토링 및 코칭 역할을 수행해야 함을 시사합니다.
일상적인 업무에 적절히 참여하되, 팀의 흐름을 방해할 정도로 과도하게 관여해서는 안 됩니다.

표 24-2. 참여 정도에 대한 시나리오 2 예시

요인	값	평가 성격
팀 구성원 간의 친밀도 -20		안락의자에 앉아 건축가 노릇을 해보세요.
팀 규모	대규모 (12명) +20 통제광	
전반적인 경험은 대부분 주니어급입니다.		+20 통제광
프로젝트 복잡성 높은 복잡성		+20 통제광
프로젝트 기간	6개월	-20 안락의자에 앉아 건축가 노릇을 해보세요.
누적 점수		+20 통제광

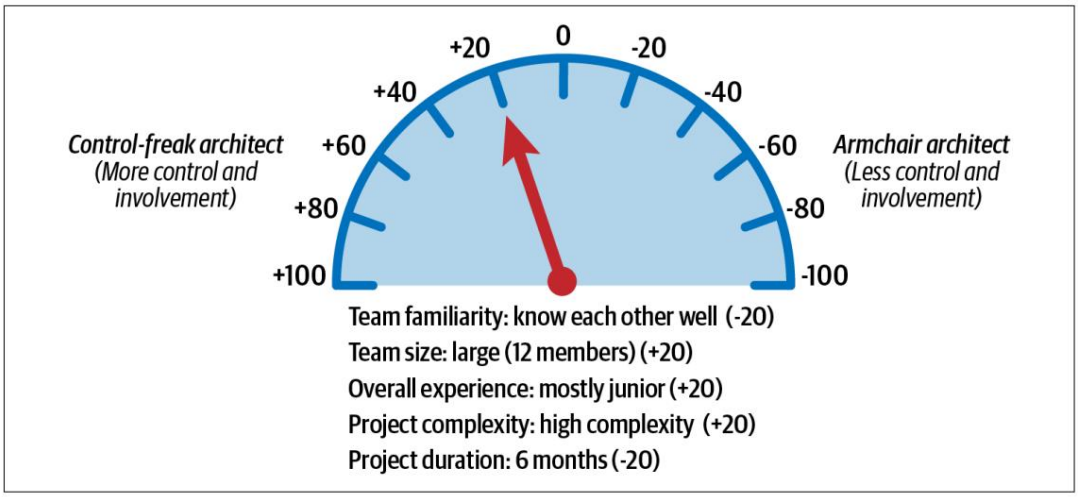

그림 24-6. 시나리오 2의 참여 정도

건축가들은 프로젝트 시작 단계에서 이러한 요소들을 활용하여 자신들의 참여 정도를 결정합니다.
처음에는 참여할 계획이지만, 프로젝트가 진행됨에 따라 참여 수준은 대개 변합니다.
따라서 프로젝트 전반에 걸쳐 이러한 요소들을 지속적으로 분석할 것을 권장합니다.
생명주기.

팀의 전반적인 성적과 같은 일부 요소는 객관적이지 않기 때문에 이러한 요소들을 객관화하기는 어렵습니다.
경험 수준)은 다른 것보다 더 큰 의미를 가질 수 있습니다. 이러한 경우 met!
rics는 특정 상황에 맞게 쉽게 가중치를 부여하거나 수정할 수 있습니다.

여기서 핵심은 건축가의 적절한 참여 정도가 중요하다는 것입니다.
개발팀은 다음 다섯 가지 요소에 따라 달라집니다. 이러한 요소를 활용하면
건축가는 참여 수준을 측정하여 적절한 경계를 설정할 수 있습니다.
팀에 적합한 크기의 "공간"을 만드세요.

팀 경고 신호

우리는 아키텍트가 개발팀에 얼마나 관여해야 하는지를 결정하는 데 도움이 되는 다섯 가지 요소 중 하나로 팀 규모를 언급했습니다. 팀 규모가 클수록 아키텍트의 관여가 더 많이 필요하고, 팀 규모가 작을수록 관여는 덜 필요합니다.

하지만 "대규모 팀"이란 정확히 무엇을 의미할까요? 이 섹션에서는 아키텍트가 팀 규모가 너무 커서 효율성을 저해하는지 판단하는 데 도움이 되는 세 가지 요소를 살펴보겠습니다.

공정 손실

프로세스 손실(Process Loss)이라는 용어는 프레드 브룩스가 그의 저서 『미신적인 맨먼스(Mythical Man-Month)』(Addison-Wesley, 1995)에서 처음 사용했습니다. 브룩스의 법칙 이라고도 불리는 프로세스 손실의 기본 개념은 프로젝트에 투입되는 인원이 많을수록 프로젝트에 소요되는 시간이 늘어난다는 것입니다. 그림 24-7 에서 볼 수 있듯이, 팀의 잠재력은 팀 구성원 모두의 공동 노력으로 정의됩니다. 그러나 브룩스는 어떤 팀이든 실제 생산성은 항상 잠재 생산성보다 낮을 것이며, 그 차이를 팀의 프로세스 손실이라고 부른다고 주장합니다.

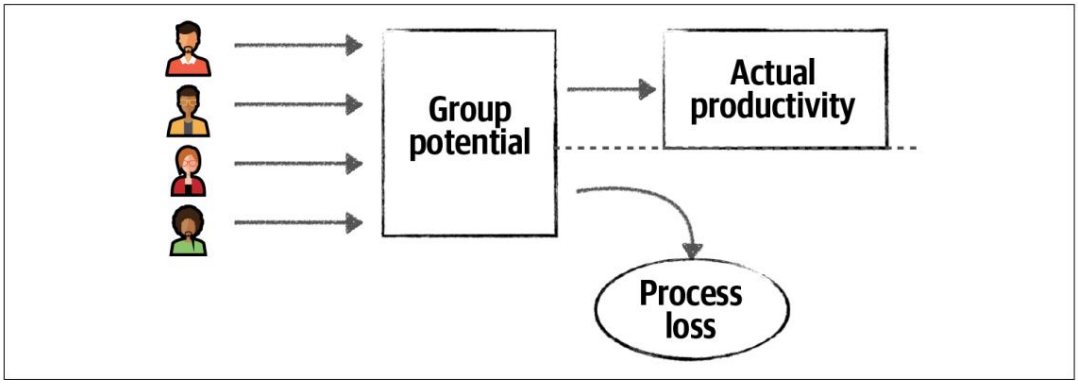

그림 24-7. 브룩스의 법칙은 팀 규모가 실제 생산성에 영향을 미친다는 것을 나타낸다.

프로세스 손실은 특정 프로젝트에 적합한 팀 규모를 결정하는 데 중요한 요소이며, 유능한 소프트웨어 아키텍트는 개발 팀을 관찰하여 프로세스 손실의 징후를 찾습니다. 예를 들어, 팀 구성원들이 저장소에 코드를 푸시할 때 병합 충돌이 자주 발생한다면, 이는 그들이 동일한 코드를 작업하고 있으며 서로 방해하고 있을 가능성을 시사합니다.

프로세스 손실을 방지하기 위해 병렬 처리가 가능한 영역을 찾고 팀 구성원들이 애플리케이션의 서로 다른 서비스 또는 영역을 담당하도록 하는 것이 좋습니다. 프로젝트 관리자가 프로젝트에 새로운 팀원을 추가하자고 제안할 때마다 유능한 아키텍트는 병렬 작업 흐름을 만들 수 있는 기회를 찾습니다. 만약 그러한 기회를 찾지 못하면, 새로운 팀원 추가가 팀에 부정적인 영향을 미칠 수 있음을 프로젝트 관리자에게 알립니다.

다원적 무지란 모든 구성원

이 내심 어떤 규범을 거부하면서도, 무언가 명백한 것을 놓치고 있다고 생각하여 공개적으로는 동의하는 현상을 말합니다. 예를 들어, 대규모 팀의 대다수 구성원이 두 원격 서비스 간에 메시징을 사용하는 것이 최선의 해결책이라고 동의한다고 가정해 보겠습니다. 한 사람은 두 서비스 사이에 방화벽이 존재하기 때문에 이것이 어리석은 생각이라고 여깁니다. 그러나 이 사람은 공개적으로는 다른 모든 사람과 함께 메시징 사용에 동의합니다. 비록 내심 거부하지만, 자신의 의견을 밝히면 자신이 뭔가 명백한 것을 놓치고 있는 것은 아닌지 두려워하기 때문입니다. 그룹의 규모가 클수록 다른 사람과 의견을 나누려는 의지가 줄어듭니다. 소규모 팀이었다면, 이 사람이 먼저 나서서 원래 해결책에 이의를 제기하고, 팀이 더 나은 해결책으로 다른 프로토콜(예: REST)을 선택하도록 유도했을 수도 있습니다.

다원적 무지라는 개념은 한스 크리스티안 안데르센의 덴마크 동화 "벌거벗은 임금님" 을 통해 유명해졌습니다 . 이 이야기에서 두 사기꾼은 재단사로 가장하여 왕에게 자신들이 "만들어 준" 새 옷은 자격이 없는 사람에게는 보이지 않는다고 속입니다. 옷을 볼 수 없는 왕은 자신이 자격이 없다는 사실을 인정하고 싶지 않아 완전히 벌거벗은 채 돌아다니며 신하들에게 새 옷이 마음에 드는지 묻습니다. 신하들은 자격이 없다고 여겨질까 두려워 왕에게 새 옷이 최고라고 칭찬합니다. 이러한 어리석음은 한 아이가 마침내 왕에게 옷을 입지 않았다고 외칠 때까지 계속됩니다.

회의 중에 유능한 소프트웨어 아키텍트는 사람들의 표정과 몸짓을 주의 깊게 살피며, 다양한 의견 차이로 인해 회의적인 생각을 숨기는 사람이 있는지를 파악합니다. 만약 아키텍트가 이러한 상황을 감지한다면, 중재자 역할을 수행해야 합니다. 예를 들어, 회의적인 사람에게 제안된 해결책에 대한 의견을 묻고, 설령 그 사람이 틀렸더라도 자신의 의견을 말할 때 지지해 주어야 합니다. 여기서 중요한 것은 아키텍트가 중재자로서 모든 사람이 자유롭게 의견을 말할 수 있는 안전한 환경이라고 느끼도록 하는 것입니다.

적절한 팀 규모를 판단하는 세 번째 요소는 책임 분산입니다. 팀 규모가 커질수록 의사소통에 부정적인 영향을 미칩니다. 팀원들이 누가 무엇을 담당해야 하는지 혼란스러워하고 중요한 업무가 누락된다면, 팀 규모가 너무 크다는 신호입니다.

그림 24-8은 시골길 갓길에 고장 난 차 옆에 서 있는 사람을 보여줍니다. 이런 상황에서 얼마나 많은 사람들이 멈춰 서서 운전자에게 괜찮은지 물어볼까요? 길이 좁고 마을도 작을 테니 지나가는 사람들이 모두 멈춰 설 수도 있겠죠. 하지만 같은 운전자가 대도시의 번잡한 고속도로 갓길에 갇혔다면 어떨까요? 수천 대의 차가 그냥 지나가 버릴 테고, 아무도 멈춰 서서 괜찮은지 물어보지 않을 겁니다. 이는 책임 분산의 좋은 예입니다. 도시가 점점 더 번잡해지고 혼잡해질수록 사람들은 운전자가 이미 도움을 요청했거나, 아니면 주변 사람들 중 누군가가 도움을 줄 거라고 생각하기 쉽습니다.

현장을 목격하는 것이 도움이 될 수 있습니다. 하지만 대부분의 경우 구조대가 도착하지 않아 운전자는 배터리가 방전되었거나 휴대전화를 잊어버린 채 오도 가도 못하는 상황에 처하게 됩니다.

그림 24-8. 책임 분산

유능한 아키텍트는 개발팀이 아키텍처를 구현하는 과정을 안내할 뿐만 아니라, 팀원들이 건강하고 행복하게 협력하여 공동의 목표를 달성할 수 있도록 지원합니다. 다음 세 가지 경고 신호를 파악하고 그로 인해 발생하는 문제를 해결하도록 돕는 것은 효과적인 개발팀을 만드는 좋은 방법입니다.

체크리스트 활용하기

항공기 조종사들은 모든 비행에서 체크리스트를 사용합니다. 아무리 경험이 많고 노련한 베테랑 조종사라도 이륙, 착륙, 그리고 흔한 상황부터 드문 예외 상황까지 수천 가지 상황에 대한 체크리스트를 가지고 있습니다. 조종사들이 체크리스트를 사용하는 이유는 항공기 설정이나 절차 중 하나라도 놓치는 것(예를 들어 이륙 전에 플랩을 10도로 설정하는 것을 잊는 것)이 안전한 비행과 참사의 차이를 만들 수 있기 때문입니다.

아툴 가완데 박사의 훌륭한 저서 『체크리스트 선언(e Checklist Manifesto)』 (피카도르, 2011)에서 그는 수술 절차를 더욱 안전하게 만드는 체크리스트의 힘에 대해 설명합니다. 병원에서 포도상구균 감염률이 높은 것에 경각심을 느낀 가완데 박사는 수술 체크리스트를 개발했습니다.
체크리스트를 사용한 병원의 감염률은 거의 0에 가까워진 반면, 체크리스트를 사용하지 않은 대조 병원의 감염률은 계속 증가했습니다.

체크리스트는 효과적입니다. 모든 작업이 빠짐없이 처리되었는지 확인하는 데 매우 유용한 도구입니다. 그렇다면 소프트웨어 개발 업계에서는 왜 체크리스트를 활용하지 않을까요?
저희는 이 업계에서 오랜 기간 일해오면서 체크리스트가 개발팀의 효율성에 큰 영향을 미친다고 확신합니다. 물론 대부분의 소프트웨어 개발자는 여객기 조종이나 공연과 같은 생사를 건 문제를 다루는 것은 아닙니다.

심장 수술처럼 모든 것에 체크리스트가 필요한 것은 아닙니다. 다시 말해, 소프트웨어 개발자는 모든 것에 체크리스트가 필요한 것이 아닙니다. 중요한 것은 언제 체크리스트를 활용하고 언제 활용하지 말아야 하는지 아는 것입니다.

그림 24-9는 체크리스트가 아니라 새 데이터베이스 테이블을 생성하기 위한 절차적 단계들의 집합이므로 체크리스트 형태로 제시해서는 안 됩니다. 일부 작업은 상호 의존적입니다. 예를 들어, 양식이 제출되지 않으면 데이터베이스 테이블을 검증할 수 없습니다. 절차적 흐름에 따라 작업이 종속되는 모든 프로세스는 체크리스트에 포함되어서는 안 됩니다. 또한, 자주 실행되는 간단하고 익숙한 프로세스도 체크리스트에 포함해서는 안 됩니다.

오류.

Done	Task description
☐	Determine database column field names and types
☐	Fill out database table request form
☐	Obtain permission for new database table
☐	Submit request form to database group
☐	Verify table once created

그림 24-9. 잘못된 체크리스트의 예

체크리스트를 활용하기에 좋은 프로세스는 정해진 절차 순서나 종속적인 작업이 없는 프로세스, 그리고 사람들이 단계를 자주 건너뛰거나 오류를 범하는 프로세스입니다. 하지만 모든 것을 체크리스트로 만들려고 과도하게 노력해서는 안 됩니다. 아키텍트는 체크리스트가 개발팀의 효율성을 실제로 향상시킨다는 것을 알게 되면 종종 이렇게 하는데, 이는 ' 수확 체감의 법칙'을 위반하는 행위입니다. 아키텍트가 체크리스트를 많이 만들수록 개발자들이 이를 활용할 가능성은 줄어듭니다. 또한 필요한 모든 단계를 포함하면서도 체크리스트를 최대한 간결하게 만드는 것이 좋습니다. 개발자들은 일반적으로 지나치게 긴 체크리스트를 따르지 않습니다. 나열된 작업 중 자동화할 수 있는 작업은 자동화하고 체크리스트에서 삭제하세요.

체크리스트에 당연한 내용을 굳이 적으려고 애쓰지 마세요. 당연한 것들이야말로 사람들이 놓치는 경우가 대부분입니다.

저희가 가장 유용하다고 생각하는 체크리스트 세 가지는 개발자 코드 완성도, 단위 및 기능 테스트, 그리고 소프트웨어 릴리스에 관한 것입니다. 각 체크리스트는 다음 섹션에서 자세히 설명합니다.

호손 효과

개발팀에 체크리스트를 도입할 때 가장 어려운 점은 개발자들이 실제로 체크리스트를 사용하도록 하는 것입니다. 시간이 부족하다는 이유로 체크리스트의 항목만 확인하고 실제 작업은 수행하지 않는 경우가 너무나 흔합니다.

이 문제를 해결하는 한 가지 방법은 체크리스트 사용이 가져올 수 있는 변화에 대해 팀원들과 이야기를 나누고, 아툴 가완데의 '체크리스트 선언문'을 읽어보도록 하는 것입니다.
각 팀원이 체크리스트의 근거를 확실히 이해하고 있는지 확인하십시오.
체크리스트에 어떤 절차를 포함하고 어떤 절차를 제외할지 직원들이 협력하여 결정하도록 하는 것을 고려해 보세요. 주인의식을 심어주는 것도 도움이 됩니다.

다른 모든 방법이 실패했을 때는 호손 효과가 있습니다 . 사람들이 자신이 관찰되거나 감시당하고 있다는 사실을 알게 되면 행동을 바꾸는 경향이 있는데, 대개는 옳은 일을 하는 것입니다. 이 효과는 실제 감시보다는 인식만으로도 충분합니다. 예를 들어, 많은 고용주들이 눈에 잘 띄는 곳에 작동하지 않는 카메라를 설치하거나, 거의 확인하지 않는 웹사이트 모니터링 소프트웨어를 설치합니다. (이러한 보고서 중 실제로 관리자가 확인하는 것은 몇 건이나 될까요?)

호손 효과를 활용하여 체크리스트를 관리하려면, 체크리스트 사용이 팀 생산성에 매우 중요하기 때문에 모든 체크리스트의 작업이 실제로 수행되었는지 확인하기 위해 검증할 것이라고 팀원들에게 알리세요. 실제로는 간헐적인 무작위 점검만으로도 충분하며, 개발자들이 항목을 건너뛰거나 완료했다고 잘못 표시할 가능성이 훨씬 줄어들 것입니다.

개발자 코드 자동 완성 체크리스트

개발자 코드 완성 체크리스트는 특히 개발자가 코드 작업을 "완료"했다고 말할 때 유용한 도구입니다. 또한 "완료의 정의"를 내리는 데에도 유용합니다. 체크리스트의 모든 항목이 완료되면 개발자는 자신이 작업하던 코드 작업을 실제로 완료했다고 말할 수 있습니다.

개발자 코드 자동 완성 체크리스트에 포함해야 할 몇 가지 사항은 다음과 같습니다.

- 자동화 도구에 포함되지 않은 코딩 및 서식 표준
- 자주 간과되는 항목(예: 흡수된 예외 사항)
- 프로젝트별 표준
- 특별 팀 지침 또는 절차

그림 24-10은 개발자 코드 완성 체크리스트의 예시를 보여줍니다. 여기에는 "코드 정리 및 코드 서식 지정 실행"이나 "흡수된 예외가 없는지 확인"과 같이 명백한 작업들이 있습니다. 바쁜 개발자들이 IDE에서 코드 정리 및 서식 지정을 실행하는 것을 얼마나 자주 잊어버릴까요? 아주 자주 있습니다. 가완데는 그의 저서 『체크리스트 선언』에서 수술 절차에 대해서도 같은 현상을 발견했는데, 명백해 보이는 작업들이 종종 누락된다는 것입니다.

Done	Task description
☐	Run code cleanup and code formatting
☐	Execute custom source validation tool
☐	Verify the audit log is written for all updates
☐	Make sure there are no absorbed exceptions
☐	Check for hardcoded values and convert to constants
☐	Verify that only public methods are calling setFailure()
☐	Include @ServiceEntrypoint on service API class

그림 24-10. 개발자 코드 완성 체크리스트 예시

아키텍트는 체크리스트를 검토하여 자동화하거나 코드 유효성 검사 도구의 플러그인으로 작성할 수 있는 항목이 있는지 항상 확인해야 합니다. 체크리스트에 프로젝트별 작업(예: 사용자 지정 유효성 검사기 실행, 감사 로그 기록 확인, setFailure () 메서드 호출, @ServiceEntrypoint 어노테이션 포함)을 포함하는 것은 좋지만, 이 중 일부는 자동화할 수 있습니다. 예를 들어, "서비스 API 클래스에 @ServiceEntry포인트 포함"과 같은 작업은 자동화가 어려울 수 있지만, "공개 메서드만 setFailure()를 호출하는지 확인"과 같은 작업은 코드 크롤링 도구를 사용하여 간단하게 자동화할 수 있습니다. 자동화 가능한 부분을 확인하면 체크리스트의 크기를 줄이고 유용한 정보를 더 많이 얻을 수 있습니다.

단위 및 기능 테스트 체크리스트는 아마도 가장 유용한 체

크리스트 중 하나일 것입니다. 이 체크리스트에는 소프트웨어 개발자가 테스트하는 것을 종종 잊어버리는 특이하고 예외적인 상황들이 포함되어 있습니다. QA 담당자가 특정 테스트 케이스를 기반으로 코드 문제를 발견할 때마다 해당 테스트 케이스를 이 체크리스트에 추가하세요.

이 체크리스트는 코드에 대해 실행할 수 있는 모든 유형의 테스트를 포함하기 때문에 일반적으로 가장 긴 체크리스트 중 하나입니다. 그 목적은 가능한 한 완벽한 테스트를 보장하여 개발자가 체크리스트를 완료했을 때 코드가 사실상 프로덕션 환경에 배포할 준비가 되도록 하는 것입니다.

다음은 일반적인 단위 및 기능 테스트 체크리스트에서 찾아볼 수 있는 몇 가지 항목입니다.

- 텍스트 및 숫자 필드의 특수 문자 포함
- 최소값 및 최대값 범위
- 특이하고 극단적인 테스트 사례
- 누락된 필드

개발자 코드 완성 체크리스트와 마찬가지로, 자동화 테스트로 작성할 수 있거나 이미 자동화 테스트 스위트에 포함된 항목은 체크리스트에서 제거하고 자동화해야 합니다.

개발자들은 유닛 테스트를 작성할 때 어디서부터 시작해야 할지, 또는 몇 개를 작성해야 할지 모르는 경우가 있습니다. 이 체크리스트는 일반적인 테스트 시나리오와 구체적인 테스트 시나리오가 개발 프로세스에 포함되도록 돕는 방법을 제공합니다. 테스트와 개발이 분리된 팀에서 운영되는 조직의 경우, 이 체크리스트는 두 팀 간의 격차를 해소하는 데 도움이 됩니다. 개발팀이 모든 테스트를 완벽하게 수행할수록 테스트팀의 업무 부담이 줄어들고, 테스트팀은 체크리스트에 포함되지 않은 비즈니스 시나리오에 집중할 수 있게 됩니다.

소프트웨어 릴리스 체크리스트

소프트웨어를 실제 운영 환경에 배포하는 것은 소프트웨어 개발 수명주기에서 오류 발생 가능성이 가장 높은 단계 중 하나이므로, 이를 위한 체크리스트가 매우 유용합니다. 이 체크리스트는 빌드 및 배포 실패를 방지하고 소프트웨어 배포와 관련된 위험을 크게 줄여줍니다.

소프트웨어 릴리스 체크리스트는 배포가 실패하거나 문제가 발생할 때마다 새로운 오류와 변화하는 상황을 해결하기 위해 변경되기 때문에 일반적으로 여기에 제시된 체크리스트 중 가장 변동성이 큰 목록입니다.

소프트웨어 릴리스 체크리스트에는 일반적으로 다음이 포함됩니다.

- 서버 또는 외부 구성 서버의 구성 변경
- 프로젝트에 추가된 타사 라이브러리(JAR, DLL 등)
- 데이터베이스 업데이트 및 해당 데이터베이스 마이그레이션 스크립트

빌드 또는 배포가 실패할 때마다 아키텍트는 실패의 근본 원인을 분석하고 소프트웨어 릴리스 체크리스트에 해당 항목을 추가해야 합니다. 이렇게 하면 다음 빌드 또는 배포 시 해당 항목이 검증되어 문제가 다시 발생하는 것을 방지할 수 있습니다.

안내 제공

소프트웨어 아키텍트가 팀의 효율성을 높이는 또 다른 방법은 설계 원칙을 활용하여 지침을 제공하는 것입니다. 이는 개발자들이 아키텍처를 구현하는 데 필요한 제약 조건을 설정하는 데에도 도움이 됩니다. 설계 원칙을 효과적으로 전달하는 것은 성공적인 팀을 만드는 핵심 요소 중 하나입니다.

이 점을 설명하기 위해, 애플리케이션을 구성하는 타사 라이브러리 모음인 계층형 스택을 사용하는 개발팀을 안내한다고 가정해 보겠습니다. 개발팀은 일반적으로 어떤 라이브러리가 괜찮은지, 어떤 라이브러리는 안 되는지, 그리고 라이브러리 사용에 대한 자체적인 결정을 언제 내릴 수 있는지 등 계층형 스택에 대해 많은 질문을 가지고 있습니다.

먼저 개발자들에게 사용하려는 라이브러리에 대해 다음과 같은 질문에 답해달라고 요청할 수 있습니다.

- 제안된 라이브러리와 시스템의 기존 라이브러리 사이에 중복되는 부분이 있습니까?
  기능은 무엇인가요?
- 제안된 라이브러리를 사용하는 것에 대한 정당성은 무엇입니까?

첫 번째 질문은 개발자가 새 라이브러리가 제공하는 기능이 기존 라이브러리나 기존 기능으로 구현 가능한지 확인하도록 안내합니다. 개발자가 이 단계를 무시할 경우(가끔 발생하기도 합니다), 특히 대규모 프로젝트나 팀 환경에서 중복 기능을 많이 만들어낼 수 있습니다.

두 번째 질문은 개발자에게 새로운 라이브러리나 기능이 정말 필요한지 묻도록 유도합니다. 기술적 타당성과 비즈니스적 타당성 모두를 요구하는 것이 좋습니다. 이 기법을 통해 개발자는 비즈니스적 타당성을 제시해야 할 필요성을 인식하게 되기 때문입니다.

비즈니스 타당성의 영향 이 글의 저자 중 한 명은 대규모 개발팀이 참여한 매우 복잡한 자바 기반 프로젝트의 수석 아키텍트였습니다. 한 팀원은 스칼라 프로그래밍 언어에 집착하여 프로젝트에 꼭 사용하고 싶어 했습니다. 스칼라 사용에 대한 그의 열망은 결국 프로젝트에 심각한 문제를 야기했고, 두 명의 핵심 팀원이 "덜 부정적인" 환경으로 이직하겠다는 의사를 밝히는 사태까지 벌어졌습니다.

필자는 그들을 설득하여 보류하게 했습니다. 그런 다음 스칼라 애호가에게 교육 및 재작성에 드는 비용에 대한 사업적 타당성을 제시한다면 프로젝트에서 스칼라를 사용하는 것을 지원하겠다고 말했습니다. 스칼라 애호가는 매우 기뻐하며 바로 시작하겠다고 했습니다. 그는 회의를 마치고 "감사합니다! 최고예요!"라고 외치며 나갔습니다.

다음 날, 스칼라 애호가는 완전히 달라진 모습으로 사무실에 찾아와 저자와 이야기를 나누고 싶다고 했습니다. 그는 곧바로 (겸손하게) 이렇게 말했습니다.

"감사합니다." 스칼라 애호가는 스칼라를 사용해야 할 기술적인 이유를 모두 생각해냈지만, 그 어떤 기술적 장점도 비용, 예산, 일정 측면에서 비즈니스 가치가 없다고 설명했습니다. 사실, 그는 두 가지를 깨달았습니다. 첫째, 비용, 예산, 일정 증가는 아무런 이득도 가져다주지 않는다는 것, 둘째, 자신이 팀에 방해가 되고 있다는 것이었습니다. 얼마 지나지 않아 그는 팀에서 가장 훌륭하고 도움이 되는 구성원 중 한 명으로 변모했습니다. 자신이 원하는 것에 대한 비즈니스 타당성을 제시하라는 요구를 받으면서 비즈니스 요구 사항에 대한 인식이 높아졌고, 이는 그를 더 나은 소프트웨어 개발자로 만들었으며, 팀을 더욱 강하고 건강하게 만들었습니다. 떠나려고 했던 두 명의 핵심 개발자도 팀에 남았습니다.

디자인 원칙을 전달하는 또 다른 좋은 방법은 개발팀이 어떤 결정을 내릴 수 있고 어떤 결정을 내릴 수 없는지를 그래픽으로 설명하는 것입니다.
그림 24-11 의 그림은 이것이 계층형 스택을 제어하는 데 어떤 모습일 수 있는지를 보여줍니다.

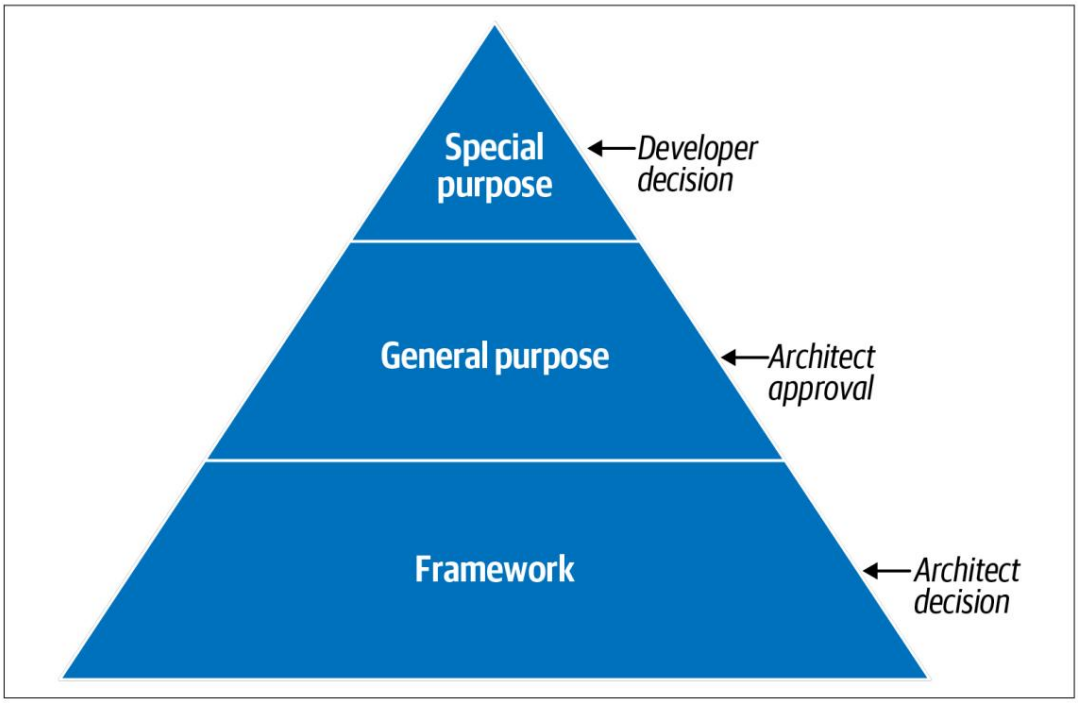

그림 24-11. 적층 구조에 대한 지침 제공

이러한 범주는 단지 예시일 뿐이며, 훨씬 더 많은 범주를 정의할 수 있습니다. 아키텍트는 그림 24-11을 팀에 제시할 때, 사용을 위해 제안된 타사 라이브러리에 각 범주를 적용해야 합니다.

특수 목적 라이브러
: 리는 PDF 렌더링, 바코드 스캔 등 사용자 정의 소프트웨어 작성이 필요하지 않은 특정 상황에 사용되는 라이브러리입니다.

범용 라이브러리는 언
: 어 API 위에 구축된 래퍼이며, Java용 Apache Commons 및 Guava 등이 포함됩니다.

뼈대
: 이러한 라이브러리는 영구 저장소(예: Hibernate) 및 제어의 역전(예: Spring)과 같은 기능에 사용됩니다. 다시 말해, 이러한 라이브러리는 애플리케이션의 전체 계층 또는 구조를 구성하며, 애플리케이션 코드에 매우 깊숙이 관여합니다.

다음으로, 건축가는 이러한 설계 원칙을 중심으로 "공간"을 만듭니다. 그림 24-11에서 볼 수 있듯이, 건축가는 특수 목적 도서관에 대해서는 개발자가 건축가와 상의 없이 결정을 내릴 수 있도록 권한을 부여했습니다. 그러나 일반 목적 도서관의 경우, 개발자는 중복 분석을 수행하고, 타당성을 제시하며, 권장 사항을 제시할 수 있지만, 이러한 유형의 도서관은 건축가의 승인을 받아야 합니다.

마지막으로, 프레임워크 라이브러리는 전적으로 아키텍트의 책임이며, 개발팀은 이러한 유형의 라이브러리에 대한 분석조차 수행해서는 안 됩니다.

요약

개발팀을 효율적으로 만드는 것은 쉽지 않은 일입니다. 풍부한 경험과 연습은 물론, 뛰어난 대인관계 능력(이에 대해서는 이후 장에서 자세히 다루겠습니다)이 필요합니다.
하지만 이 장에서 소개하는 유연한 리더십, 체크리스트 활용, 설계 원칙 전달을 통한 지침 제공과 같은 간단한 기법들은 실제로 효과가 있습니다. 개발팀이 더욱 효율적으로 일하도록 만드는 데 이러한 기법들이 얼마나 유용한지 이미 확인했습니다.

일부에서는 이러한 활동에서 건축가의 역할에 의문을 제기하며, 이러한 업무는 개발 관리자나 프로젝트 관리자에게 맡겨야 한다고 주장합니다. 우리는 이러한 주장에 강력히 반대합니다.
소프트웨어 아키텍트는 기술적인 문제에 대해 팀을 안내하고 아키텍처 구현 과정을 이끌어갑니다. 개발팀과 긴밀한 협력 관계를 구축함으로써 팀의 역동성을 파악하고 생산성 향상을 위한 변화를 주도할 수 있습니다.