

제8장

구성 요소 기반 사고

3 장에서는 모듈이라는 개념을 관련 코드들의 모음으로 소개했습니다.

이 장에서는 시스템의 구성 요소인 논리적 구성 요소 측면에서 모듈성의 아키텍처적 측면에 초점을 맞춰 이 개념을 훨씬 더 심층적으로 살펴보겠습니다.

논리적 구성 요소를 식별하고 관리하는 것은 아키텍처적 사고의 일부이며(2 장 참조), 이러한 활동을 구성 요소 기반 사고라고 부릅니다.

컴포넌트 기반 사고방식은 시스템의 구조를 특정 비즈니스 기능을 수행하기 위해 상호 작용하는 논리적 컴포넌트들의 집합으로 보는 것입니다. 아키텍트는 바로 이 수준(클래스 수준이 아닌)에서 시스템을 "관찰"합니다.

이 장에서는 소프트웨어 아키텍처 내의 논리적 구성 요소를 정의하고, 이를 식별하는 방법, 그리고 응집도(이 장 뒷부분에서 자세히 설명합니다) 분석을 통해 적절한 세분화 수준을 결정하는 방법을 다룹니다. 또한 구성 요소 간의 결합도와 느슨하게 결합된 시스템을 구축하는 방법 및 이유에 대해서도 논의합니다.

논리적 구성 요소 정의

그림 8-1에 나와 있는 전형적인 서양 주택의 평면도를 생각해 보십시오 .

평면도를 보면 주방, 침실, 욕실, 거실, 사무실 등 다양한 방들로 구성되어 있으며, 각 방은 서로 다른 용도를 가지고 있음을 알 수 있습니다.

이 방들은 집을 구성하는 기본 요소, 즉 건축 블록을 나타냅니다.

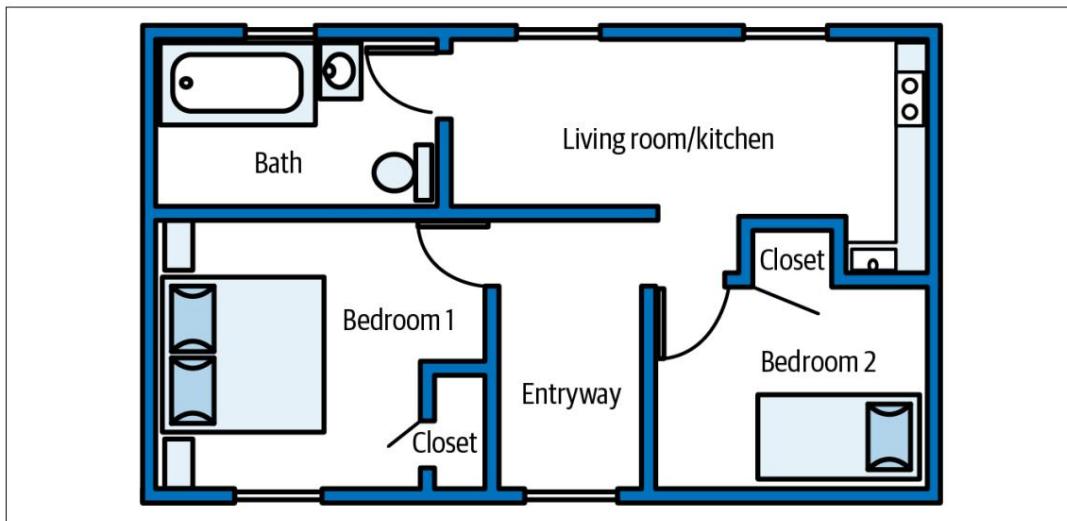


그림 8-1. 다양한 방들은 집의 구성 요소를 나타냅니다.

이와 마찬가지로, 시스템이 수행하는 주요 기능은 [그림 8-2에서 보는 바와 같이 시스템의 구성 요소를 나타냅니다](#). 집의 방처럼 각 구성 요소는 재고 관리, 주문 배송 또는 결제 처리와 같은 특정 기능을 수행합니다. 이러한 구성 요소들이 모여 시스템을 구성합니다. 각 구성 요소에는 해당 비즈니스 기능을 구현하는 소스 코드가 포함되어 있습니다.

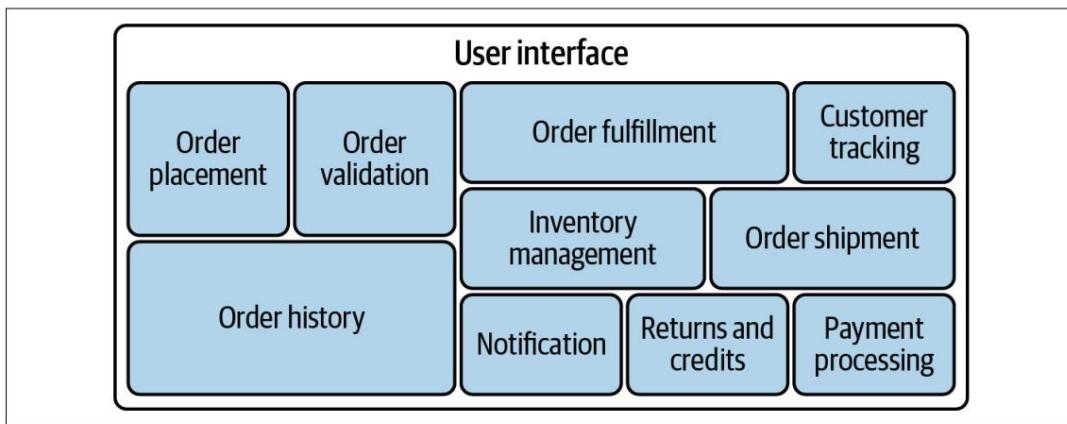


그림 8-2. 다양한 주요 기능은 시스템의 구성 요소를 나타냅니다.

소프트웨어 아키텍처의 논리적 구성 요소는 일반적으로 해당 기능을 구현하는 소스 코드가 포함된 네임스페이스 또는 디렉터리 구조를 통해 표현됩니다. 일반적으로 소스 코드가 포함된 디렉터리 구조 또는 네임스페이스의 최하위 노드는 아키텍처의 논리적 구성 요소를 나타내고, 상위 디렉터리 또는 네임스페이스 노드는 시스템의 도메인과 하위 도메인을 나타냅니다. 그림 8-3에 나타난 디렉터리 구조에서, `order_entry/ordering/payment` 경로는 결제 처리 구성 요소를 나타내고, `order_entry/processing/fullment` 경로는 주문 처리 구성 요소를 나타냅니다. 이러한 디렉터리 아래에 있는 소스 코드는 해당 논리적 구성 요소를 구현합니다.

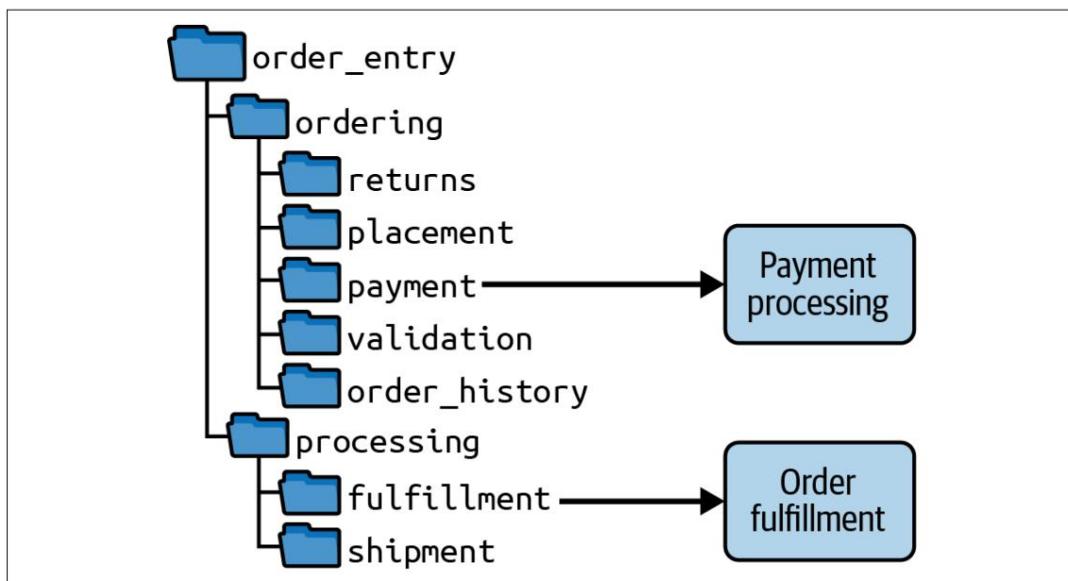


그림 8-3. 디렉터리의 리프 노드는 시스템의 구성 요소를 나타냅니다.

아키텍트는 소프트웨어 시스템의 디렉터리 구조 또는 네임스페이스를 분석하여 내부 구조, 즉 논리적 아키텍처를 이해할 수 있습니다. 다음 섹션에서는 논리적 아키텍처와 물리적 아키텍처의 차이점을 설명합니다.

논리적 아키텍처와 물리적 아키텍처의 차이점

논리적 아키텍처는 시스템의 논리적 구성 요소(빌딩 블록)와 이러한 구성 요소들 간의 상호 작용 방식을 나타냅니다. 또한 일반적으로 다양한 액터(시스템과 상호 작용하는 사용자)와의 상호 작용도 보여주며, 데이터가 어디에서 사용되거나 구성 요소 간에 전송되는지를 명확히 하기 위해 저 장소(데이터베이스가 아님)를 포함할 수도 있습니다.

그림 8-4는 논리적 아키텍처의 예를 보여줍니다.

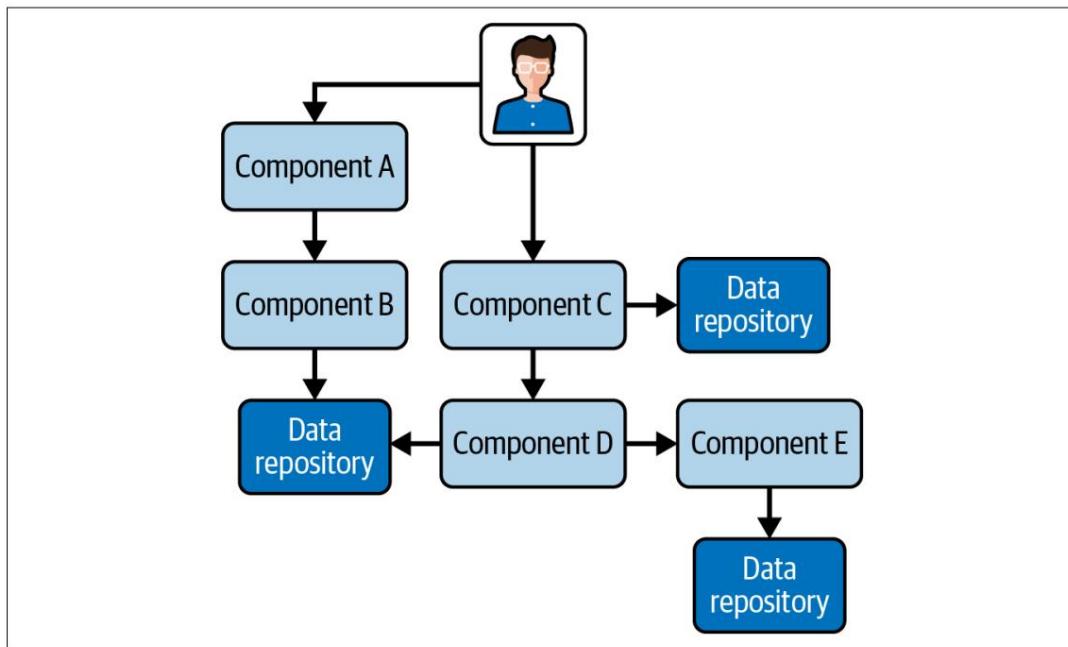


그림 8-4. 논리적 아키텍처 다이어그램

논리적 아키텍처 다이어그램은 일반적으로 사용자 인터페이스, 데이터베이스, 서비스 또는 기타 물리적 구성 요소를 보여주지 않습니다. 오히려 논리적 구성 요소와 이러한 구성 요소 간의 상호 작용 방식을 보여주며, 이는 코드를 구성하는 디렉터리 구조 및 네임스페이스와 일치해야 합니다.

반면에 물리적 아키텍처는 서비스, 사용자 인터페이스, 데이터베이스 등과 같은 물리적 구성 요소를 포함합니다. **그림 8-5**는 물리적 아키텍처 다이어그램의 예입니다.

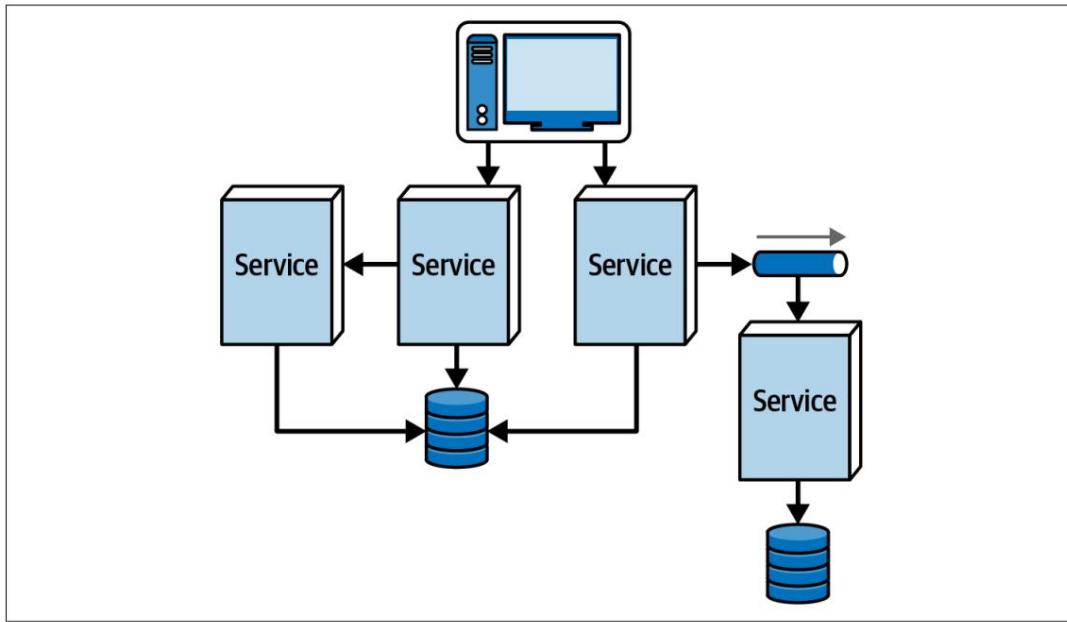


그림 8-5. 물리적 아키텍처 다이어그램

시스템의 물리적 아키텍처는 이 책 2부에서 설명하는 마이크로서비스, 계층형 아키텍처, 이벤트 기반 아키텍처 등 다양한 아키텍처 스타일 중 하나 이상을 밀접하게 반영해야 합니다.

많은 아키텍트들이 논리적 아키텍처 설계 단계를 건너뛰고 바로 물리적 아키텍처 설계를 시작하는 경우가 있습니다. 하지만 저희는 이러한 방식을 권장하지 않습니다. 물리적 아키텍처만으로는 시스템 기능의 위치와 전체적인 연결 방식을 파악하기 어렵기 때문입니다. 예를 들어, 물리적 아키텍처에서는 결제 처리 기능이 여러 서비스에 분산되어 있을 수 있어 시스템의 다른 부분과의 상호 작용 방식을 파악하기 어렵습니다. 또한, 물리적 아키텍처는 개발 팀에게 모놀리식 시스템 또는 분산 시스템을 구축하거나 코드를 구성하는 방법에 대한 지침을 제공하지 않아, 유지 관리, 테스트 및 배포가 어려운 비구조적인 아키텍처로 이어지는 경우가 많습니다.

일반적으로 시스템의 논리적 아키텍처는 물리적 아키텍처와 독립적입니다.

다시 말해, 논리적 아키텍처를 설계할 때는 시스템의 물리적 구조보다는 시스템이 무엇을 하는지, 그 기능이 어떻게 구분되는지, 그리고 시스템의 기능적 부분들이 어떻게 상호 작용하는지에 초점을 맞춥니다. 예를 들어, 그림 8-4와 같은 논리적 아키텍처를 설계하는 아키텍트는 이러한 구성 요소들을 모두 단일 배포 단위(모놀리식 아키텍처)로 구성할지, 아니면 서비스 형태로(개별 배포 단위) 배포할지 아직 결정하지 않았을 수도 있고, 어떤 아키텍처 스타일이 가장 적합한지 반드시 결정하지 않았을 수도 있습니다.

논리적 아키텍처 구축

논리적 아키텍처를 구축하려면 논리적 구성 요소를 지속적으로 식별하고 재구성해야 합니다. 구성 요소 식별은 반복적인 프로세스를 통해 가장 효과적으로 수행할 수 있습니다. [그림 8-6에서 보는 것](#)처럼 후보 구성 요소를 생성한 다음 피드백 루프를 통해 이를 개선하는 과정을 거칩니다.

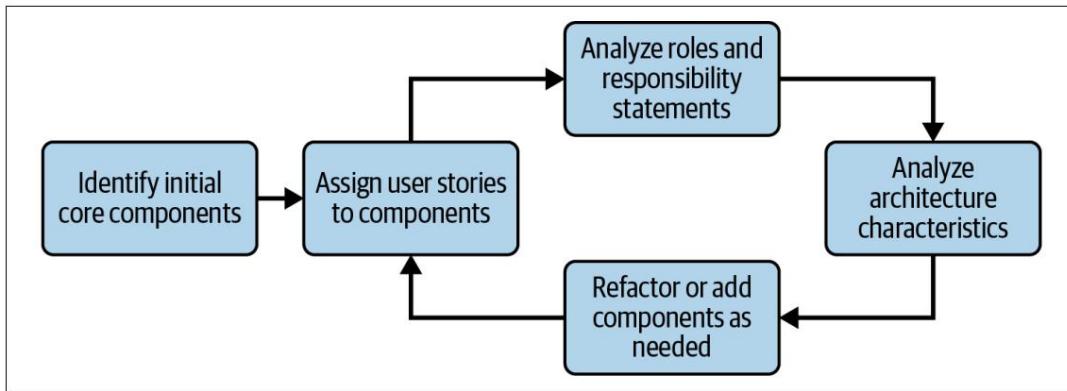


그림 8-6. 구성 요소 식별 및 리팩토링 주기

논리적 아키텍처를 개발할 때 건축가가 가장 먼저 하는 일은 초기 핵심 구성 요소를 식별하고, 그 구성 요소에 사용자 스토리 또는 요구 사항을 할당하는 것입니다.

이후, 아키텍트는 구성 요소의 역할과 책임을 분석하여 할당된 사용자 스토리 또는 요구 사항이 타당한지 확인합니다. 그런 다음 시스템이 지원해야 하는 아키텍처적 특성을 살펴보고, 이러한 특성을 기반으로 구성 요소를 분리하거나 통합하는 등 리팩토링이 필요한지 판단합니다. 마지막으로, 아키텍트는 이러한 분석을 바탕으로 필요에 따라 구성 요소를 개선합니다. 이 과정은 본질적으로 멈추지 않는 피드백 루프입니다.

그림 8-6의 워크플로는 신규 시스템 구축(그린필드 시스템)은 물론, 기존 시스템에 기능을 추가하거나 변경할 때에도 사용할 수 있습니다. 예를 들어, 주문 시스템에서 사용자가 상품을 집으로 배송 받는 대신 매장에서 직접 수령할 수 있도록 기능을 추가해야 한다고 가정해 보겠습니다. 이 경우 예약 코드를 추가하고 기존 주문 프로세스를 변경해야 합니다. 이러한 변경에는 새로운 구성 요소, 기존 구성 요소 수정 또는 둘 다 필요할 수 있습니다. 구성 요소가 변경됨에 따라 역할과 책임도 변경될 수 있으며, 이에 따라 코드 구조를 재구성하거나 새로운 구성 요소를 생성해야 할 수도 있습니다.

다음 섹션에서는 이러한 각 단계를 자세히 설명합니다.

핵심 구성 요소 식별 새로운 논리적 아키텍처

텍처를 시작하거나 기존 아키텍처를 크게 수정할 때 가장 어려운 점은 초기 핵심 구성 요소를 무엇으로 정해야 하는지 결정하는 것입니다. 많은 소프트웨어 아키텍트가 저지르는 실수 중 하나는 초기 논리적 구성 요소를 처음부터 완벽하게 만들려고 너무 많은 노력을 기울이는 것입니다. 더 나은 접근 방식은 시스템의 핵심 기능을 기반으로 초기 핵심 구성 요소가 어떤 모습일지 "최선의 추측"을 하고, [그림 8-6에 설명된 워크플로 단계를 통해 이를 구체화하는 것입니다.](#) 다시 말해, 시스템과 그 구체적인 요구 사항에 대해 가장 잘 모르는 상태에서 처음부터 모든 것을 완벽하게 만들려고 하기보다는 시스템에 대해 더 많이 알게 되면서 논리적 구성 요소를 반복적으로 개선하는 것이 더 효과적입니다.

대부분의 경우, 아키텍트는 논리적 구성 요소를 만들기 시작하기 위해 시스템의 모든 요구 사항과 사양을 알 필요가 없으며, 경우에 따라서는 전혀 알 필요가 없습니다. 일반적으로 초기 핵심 구성 요소는 사용자가 수행할 수 있는 주요 작업이나 시스템의 주요 처리 워크플로를 기반으로 합니다.

초기 핵심 구성 요소를 빈 양동이에 비유해 보겠습니다. 아키텍트가 사용자 스토리나 요구사항을 해당 구성 요소에 할당하여 "채우기" 전까지는 양동이는 아무런 기능도 하지 않습니다. [그림 8-7](#)에서처럼 구성 요소 이름은 제안된 역할과 책임을 나타내야 합니다. 하지만 아키텍트가 사용자 스토리를 할당하기 전까지(워크플로의 다음 단계) 구성 요소는 기본적으로 임시 자리 표시자, 즉 기능적 구성 요소에 대한 추측에 불과합니다.

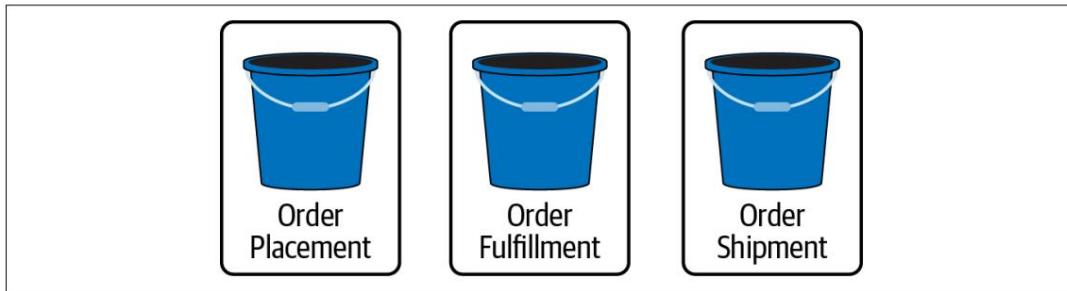


그림 8-7. 처음에는 초기 구성 요소들이 빈 양동이와 같습니다.

다음은 초기 핵심 구성 요소를 생성하는 세 가지 일반적인 접근 방식입니다. 첫 번째와 두 번째 접근 방식(워크플로우 및 액터/액션)은 유용하다고 생각하는 방식이며, 세 번째 접근 방식(엔티티 함정)은 지양해야 할 안티패턴입니다.

워크플로우 접근법

아키텍트가 논리적 아키텍처의 초기 핵심 구성 요소를 식별하는 데 일반적으로 사용하는 접근 방식은 워크플로우 접근 방식입니다. 이름에서 알 수 있듯이 이 접근 방식은 사용자가 시스템(또는 주요 요청 처리 워크플로)을 통해 거치는 주요 정상 경로(오류가 발생하지 않는 경로)를 활용합니다. 아키텍트가 전반적인 흐름을 파악하고 있다면 이러한 단계를 기반으로 구성 요소를 개발할 수 있습니다.

예를 들어, 회사에서 사용할 새로운 주문 입력 시스템을 구축한다고 가정해 보겠습니다. 아직 구체적인 요구 사항과 사양은 모르지만, 적어도 새 주문을 처리하는 일반적인 워크플로는 알고 있습니다. 따라서 워크플로의 각 단계에 구성 요소를 할당할 수 있습니다.

1. 사용자가 상품 카탈로그를 살펴봅니다(" 상품 검색 ") . 2. 사용자가 주문을 합니다(" 주문하기 "). 3. 사용자가 주문 금액을 결제합니다(" 주문 결제 ").
4. 주문 세부 정보가 포함된 이메일을 사용자에게 전송합니다. " 고객 알림 "
5. 주문 준비 " 주문 처리 " 6. 주문 발송 " 주문 배송 " 7. 고객에게 주문 배송 알림 이메일 발송 " 고객 알림 "
8. 배송 추적 " 주문 추적 "

워크플로 접근 방식에서는 주요 워크플로의 모든 단계에서 새로운 구성 요소가 생성되는 것은 아닙니다. 위의 워크플로에서 4단계와 7단계는 모두 고객 알림 구성 요소를 사용합니다. 아키텍트는 시스템에 대해 가능한 한 많은 주요 워크플로 또는 사용자 여정을 모델링하고, 해당 단계에서 대응하는 구성 요소를 식별합니다.

이것은 다시 말하지만, 논리적 아키텍처가 어떤 모습일지에 대한 "최선의 추측"일 뿐입니다. 이러한 구성 요소는 아직 책임이 없는 빈 버킷을 나타내므로 발전함에 따라 변경될 가능성이 높습니다(그림 8-6 참조). 이는 지극히 정상적인 현상이며 소프트웨어 아키텍처의 반복적인 특성 중 하나입니다. 시스템의 모든 워크플로를 모델링하려고 애쓰지 마세요. 대신 주요 워크플로에 집중하세요. 나머지는 시스템에 대해 더 많이 배우고 사용자 스토리와 요구 사항을 수집하면서 자연스럽게 발전할 것입니다.

행위자/행동 접근법

건축가들이 초기 핵심 구성 요소를 식별하는 또 다른 방법은 행위자/행동 접근법입니다. 이 접근 방식은 시스템에 여러 행위자가 있을 때 특히 유용합니다. 이 방식을 사용하면 아키텍트는 사용자가 시스템에서 수행할 수 있는 주요 작업(예: 주문하기)을 식별할 수 있습니다. 시스템 자체도 항상 행위자로서 청구 및 재고 보충과 같은 자동화된 기능을 수행합니다.

주문 입력 시스템 예시에서 아키텍트는 세 가지 행위자를 식별할 수 있습니다. 고객, 주문 포장 담당자(상자를 포장하여 배송을 위해 보내는 사람), 그리고 시스템입니다. 그런 다음 아키텍트는 이러한 각 행위자가 수행하는 주요 작업을 파악하고 해당 작업에 구성 요소를 할당합니다.

고객 담당자 • 항목

검색 " 항목 검색 "

- 상품 상세 정보 보기 " 상품 상세 정보 "
- 주문하기 " 주문하기 "
- 주문 취소 " 주문 취소 "
- 신규 고객 등록 " 고객 등록 " • 고객 정보 업데이트 " 고객 프로필 "

주문 포장 담당자 • 상자

크기 선택 " 주문 처리 "

- 주문을 배송 준비 완료로 표시(" 주문 처리 ") • 고객에게 주문 상품 배송(" 주문 배송 ")

시스템 실행자 •

재고 조정 " 재고 관리 " • 공급업체에 추가 재고 주문 " 공급업체

주문 " • 결제 적용 " 주문 결제 "

워크플로 방식과 마찬가지로 모든 작업에 반드시 별도의 구성 요소가 필요한 것은 아닙니다. 예를 들어 주문에 사용할 상자 크기를 선택하고 배송 준비 완료로 표시하는 작업은 모두 주문 처리 구성 요소에서 수행됩니다.

일반적으로, 액터/액션 접근 방식은 워크플로 접근 방식보다 더 많은 구성 요소를 생성합니다. 이는 아키텍트가 모델링할 주요 워크플로의 수에 따라 달라집니다. 하지만 두 접근 방식 모두에서 아키텍트는 상세한 요구 사항이나 사양을 받기 전에도 초기 핵심 구성 요소와 그 구성 요소들 간의 상호 작용 방식을 파악할 수 있습니다.

엔티티 함정: 아키

텍트는 시스템과 관련된 엔티티에 초점을 맞춰 구성 요소를 식별한 다음, 해당 엔티티에서 구성 요소를 파생시키는 유혹에 빠지기 쉽습니다. 예를 들어, 일반적인 주문 입력 시스템에서 고객, 품목, 주문을 시스템의 주요 엔티티로 식별하고, 이에 따라 고객 관리자 구성 요소, 품목 관리자 구성 요소, 주문 관리자 구성 요소를 만들 수 있습니다. 하지만 다음과 같은 이유로 이러한 접근 방식을 강력히 권장하지 않습니다.

첫째, 논리적 구성 요소의 이름이 모호하고 구성 요소의 역할을 설명하지 못합니다. 예를 들어, 주문 관리자 구성 요소가 무엇을 하는지 이름만 보고 물어보면 "주문을 관리합니다"라는 무의미한 답변만 얻게 되는데, 이는 시스템 내에서 해당 구성 요소의 구체적인 역할이나 책임에 대해 아무것도 알려주지 않습니다. 이와 비교해 보면,

'주문 유효성 검사' 와 같은 컴포넌트 이름을 보면, 좋은 설명적 컴포넌트 이름의 중요성이 분명해집니다. 만약 컴포넌트 이름에 '관리자', '감독자', '컨트롤러', '핸들러', '엔진', '프로세서'와 같은 접미사가 포함되어 있다면, 아키텍트가 '엔티티 트랩' 안티패턴에 빠졌을 가능성이 높습니다.

둘째, 구성 요소는 도메인 관련 기능의 저장소가 됩니다.

그림 8-8 에 나와 있는 것처럼 '주문 관리자' 와 같은 엔티티 기반 컴포넌트 이름을 생각해 보세요 . 주문 유효성 검사, 주문 접수, 주문 내역, 주문 처리, 배송, 주문 추적 등 주문 관련 모든 기능이 이 단일 컴포넌트에 포함됩니다. 본질적으로 이는 개발자가 경력 중에 적어도 한 번쯤은 작성해 본 적 있는, 문자열 조작, 데이터 조작, 시간 계산 등 개발자가 원하는 모든 기능을 수행하는 수십 개의 메서드를 포함하는 '만능' 유ти리티 클래스와 같습니다.

셋째, 컴포넌트가 지나치게 세분화되면 너무 많은 기능을 수행하게 되어 본래의 목적을 잃게 됩니다. (주문 유효성 검사 컴포넌트 예시 처럼) 세분화되어 단일 목적을 가지는 대신 , 컴포넌트가 지나치게 커지는 경우가 있습니다. 이러한 컴포넌트는 유지 관리, 테스트 및 배포가 어려워 신뢰성이 떨어집니다.

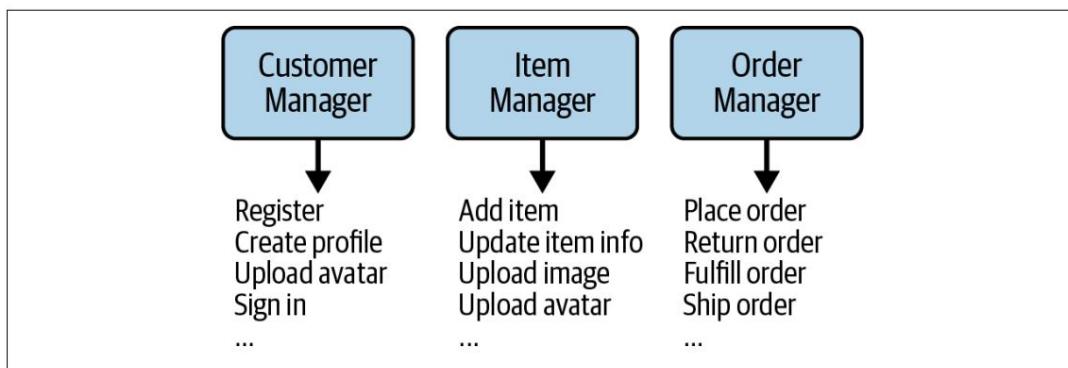


그림 8-8. 엔티티 트랩 안티패턴을 사용하면 과도한 책임을 가진 컴포넌트가 생성됩니다.

만약 아키텍트가 엔티티 기반 시스템을 구축하고, 해당 시스템이 단순히 엔티티에 대한 CRUD(생성, 읽기, 업데이트, 삭제) 작업만 수행한다면, 시스템에는 아키텍처가 필요한 것이 아니라, 개발자가 엔티티에 대한 대부분의 소스 코드를 생성할 수 있도록 해주는 CRUD 기반 프레임워크, 도구 또는 노코드/로우코드 환경이 필요합니다.

사용자 스토리를 구성 요소에 할당하기 논리

적 아키텍처를 구축하는 다음 단계는 논리적 구성 요소에 사용자 스토리 또는 요구 사항을 할당하는 것입니다. 대부분의 사용자 스토리 또는 요구 사항은 사전에 완전히 알려져 있지 않고 시스템이 발전함에 따라 진화하기 때문에 이는 반복적인 프로세스입니다. 이 단계는 그림 8-9에 나타난 것처럼 구성 요소에 구체적인 역할과 책임을 부여하여 이러한 빈 영역을 채우기 위한 것입니다.

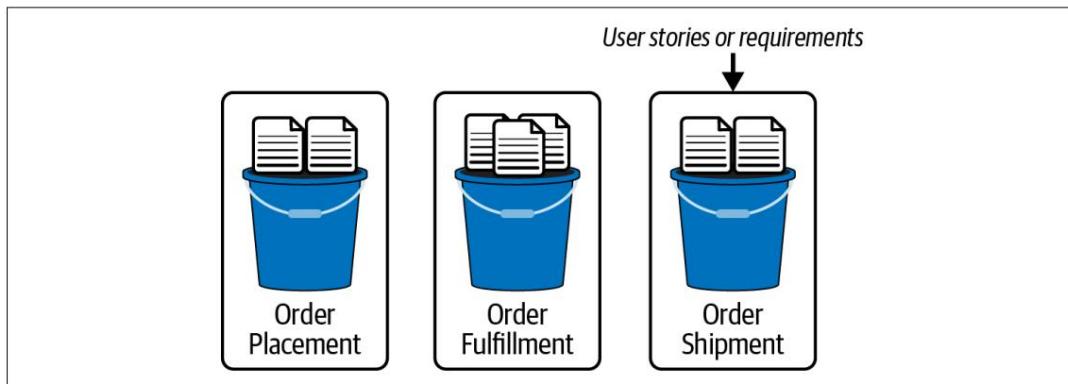


그림 8-9. 사용자 스토리 또는 요구사항으로 빈 버킷(구성 요소)을 채우기

논리적 구성 요소가 어떻게 발전하는지 알아보려면 다음 사용자 스토리를 살펴보세요.

고객 1호

고객으로서, 제가 입력한 모든 정보가 완전하고 정확한지 확인하기 위해 주문 검증을 받고 싶습니다.

주문 담당자로서, 주

문을 가장 효율적으로 포장하기 위해 어떤 크기의 상자를 사용해야 하는지 알고 싶습니다.

고객 2번

고객으로서 주문 상태가 변경될 때마다 이메일을 받고 싶습니다. 그래야 항상 주문 상태를 알 수 있으니까요.

건축가가 지금까지 다음과 같은 논리적 구성 요소를 파악했다고 가정해 보겠습니다.

- 주문하기
- 주문 처리
- 주문 배송
- 재고 관리

첫 번째 사용자 스토리를 주문 구성 요소에 할당하는 것이 타당합니다. 왜냐하면 사용자가 주문을 하기 위해 상호 작용하는 구성 요소가 바로 주문 구성 요소이기 때문입니다.

주문 유효성 검사 (고객 #1 사용자 스토리) "주문 접수"

박스 크기 결정은 주문 처리 구성 요소에서 담당하는 것이 적절할 것입니다. 왜냐하면 해당 구성 요소는 주문을 박스에 포장하고 준비하는 데 필요한 모든 시스템 로직을 담당하기 때문입니다.

박스 크기 결정 (주문 준비 사용자 스토리) "주문 처리"

하지만 세 번째 사용자 스토리는 어떨까요? 나열된 네 가지 구성 요소 중 어떤 구성 요소가 주문 접수, 배송 준비 완료, 배송 완료 시 고객에게 이메일을 보내야 할까요? 정답은 주문 접수, 주문 처리, 주문 배송 구성 요소일 수 있습니다. 하지만 이 사용자 스토리는 소스 코드를 통해 구현되며, 해당 코드는 특정 디렉토리 또는 네임스페이스에 있어야 한다는 점을 명심해야 합니다. 세 가지 구성 요소에 코드를 중복해서 사용하는 것은 바람직하지 않으므로, 아키텍트는 이 사용자 스토리를 처리할 새로운 구성 요소를 정의해야 합니다.

고객에게 이메일 발송 (고객 #2 사용자 스토리) "고객 알림 (신규)"

주문 접수, 주문 처리 및 주문 배송 구성 요소는 새 구성 요소와 통신하여 이메일을 보내도록 알려야 합니다. 이 추가 사항을 반영하면 논리적 아키텍처는 [그림 8-10과 같습니다](#).

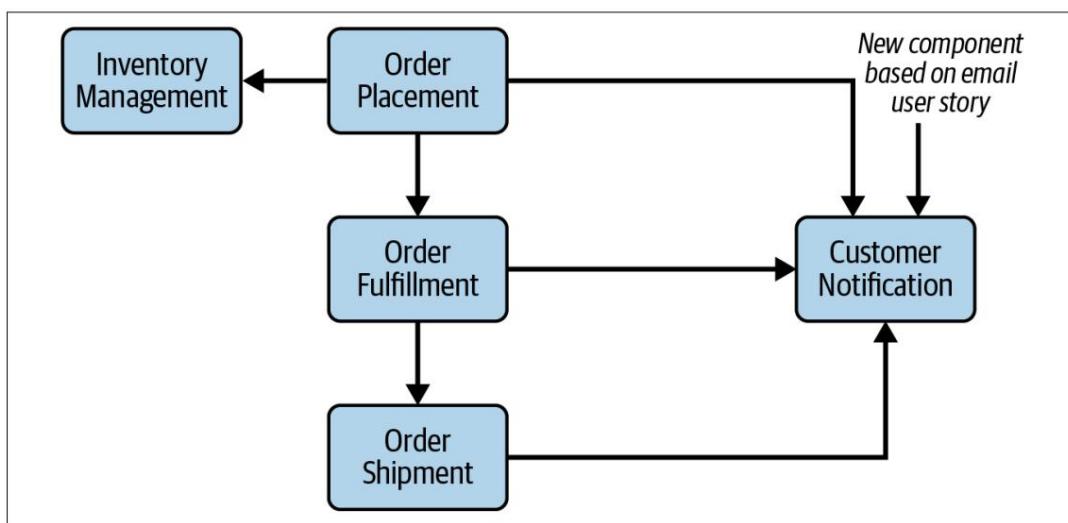


그림 8-10. 새로운 사용자 스토리에 기반한 구성 요소의 진화

역할과 책임 분석

논리적 구성 요소를 구체화하는 다음 단계는 각 구성 요소의 역할과 책임을 분석하는 것입니다. 이를 통해 아키텍트는 해당 구성 요소에 할당된 요구 사항이나 사용자 스토리가 적절한지, 그리고 구성 요소가 너무 많은 기능을 수행하지 않는지 확인합니다. 이 단계에서 아키텍트가 중점적으로 고려해야 할 사항은 응집력입니다. 즉, 구성 요소의 작업들이 어떻게, 그리고 얼마나 서로 연관되어 있는지입니다. 시간이 지남에 따라 구성 요소들의 작업들이 모두 연관되어 있더라도 구성 요소가 너무 커질 수 있습니다.

이 단계가 어떻게 진행되는지 설명하기 위해, 설계자가 주문 배치 구성 요소에 다음과 같은 요구 사항을 할당했다고 가정해 보겠습니다.

- 주문을 검증하여 모든 항목이 입력되었는지, 그리고 정확한지 확인하십시오.
- 상품 설명, 수량 및 가격이 포함된 장바구니를 표시합니다. • 정확한 배송 주소를 확인합니다. • 결제 정보를 수집합니다. • 고유 주문 ID를 생성합니다. • 주문에 대한 결제를 처리합니다. • 주문 상품의 재고 수량을 조정합니다. • 고객에게 주문 요약 이메일을 발송합니다.

만약 건축가가 이 구성 요소에 대한 역할 및 책임 명세서를 작성한다면, 그 내용은 다음과 같을 것입니다.

이 구성 요소는 주문을 검증하고 상품 사진, 설명, 수량 및 가격을 포함한 유효한 장바구니를 표시하는 역할을 합니다. 또한 주문에 대한 정확한 배송 주소를 확인하고 고객으로부터 모든 결제 정보를 수집하는 역할도 담당합니다. 더불어 결제를 적용하고 재고를 조정하며 고객에게 주문 요약 이메일을 발송하는 기능도 수행합니다.

이러한 모든 작업은 주문 처리와 관련이 있지만, 주문 처리 구성 요소는 특히 결제 적용, 재고 조정 및 고객 이메일 발송과 관련하여 지나치게 많은 책임을 맡고 있습니다. 구성 요소가 과도한 책임을 맡고 있는지 확인하는 한 가지 방법은 "그리고", "또한", "개다가", "뿐만 아니라"와 같은 접속사나 쉼표의 과도한 사용을 살펴보는 것입니다.

앞 장에서 설명했듯이 논리적 구성 요소는 코드 저장소의 네임스페이스 또는 디렉터리로 표현됩니다. 주문 처리 구성 요소의 경우, 이 구성 요소를 나타내는 모든 소스 코드는 com/app/order/placement 또는 com.app.order.placement와 같이 동일한 디렉터리 또는 네임스페이스에 있습니다. 이는 상당히 많은 기능을 포함하므로 단일 디렉터리에 저장하기에는 코드가 너무 많을 수 있습니다. 따라서 결제 처리, 재고 관리 및 이메일 통신에 필요한 클래스 파일을 각 기능을 나타내는 별도의 디렉터리로 분리하는 것이 좋습니다. 이것이 바로 논리적 구성 요소를 분리하는 핵심입니다.

아키텍트가 결제 적용, 재고 조정, 고객 이메일 발송 등의 책임을 별도의 구성 요소로 분리하면, 단일 주문 처리 구성 요소의 부담을 줄여 유지 관리, 테스트 및 배포를 더욱 용이하게 할 수 있습니다. 결과적으로 구성 요소는 다음과 같은 형태를 갖게 됩니다.

주문하기

- 주문을 검증하여 모든 항목이 입력되었는지, 그리고 정확한지 확인하십시오.
- 상품 설명, 수량 및 가격이 포함된 장바구니를 표시합니다. • 정확한 배송 주소를 확인합니다.
- 결제 정보를 수집합니다. • 고유한 주문 ID를 생성합니다.

결제 처리 • 결제를 적용합

니다.

재고 관리

- 주문한 품목의 재고 수량을 조정합니다.

고객 알림 • 고객에게 주문 요약 이

메일을 보내세요.

이제 각 구성 요소는 더욱 명확하고 뚜렷한 역할과 책임을 갖게 되었습니다.

아키텍처 특성 분석 마지막 분석 단계는 시스템에 필요

한 아키텍처 특성을 고려하는 것입니다. 확장성, 신뢰성, 가용성, 내결함성, 탄력성, 민첩성(변화에 신속하게 대응하는 능력)과 같은 일부 아키텍처 특성은 논리적 구성 요소의 크기에 영향을 미칠 수 있습니다.

예를 들어, 더 큰 구성 요소(많은 책임을 가진 구성 요소)를 더 작은 구성 요소로 분할하면 각 구성 요소의 유지 관리 및 테스트가 더 쉬워지고(애자일성), 확장성, 유연성 및 내결함성이 향상됩니다. 또 다른 좋은 예는 시스템의 두 부분이 사용자 입력을 처리하는 경우입니다. 한 부분은 수백 명의 동시 사용자를 처리하고 다른 부분은 한 번에 소수의 사용자만 지원해야 한다면, 서로 다른 아키텍처 특성이 필요합니다. 따라서 구성 요소 설계에 대한 순수 기능적 관점에서는 아키텍트가 사용자 상호 작용을 처리하는 단일 구성 요소를 할당하게 될 수 있지만, 아키텍처 특성 측면에서 구성 요소를 분석하면 세분화된 구성 요소를 사용하게 될 수 있습니다.

건축가는 논리적 아키텍처를 구축하기 전에 아키텍처적 특성을 알아야 하므로, 일반적으로 시스템에 가장 중요한 아키텍처적 특성이 무엇인지 결정한 후에 이 작업이 수행됩니다.

구조 조정 구성 요소

소프트웨어 설계에서 피드백은 매우 중요합니다. 아키텍트는 개발자와 협력하여 구성 요소 설계를 지속적으로 반복해야 합니다. 소프트웨어 설계는 예상치 못한 다양한 어려움을 수반합니다. 따라서 구성 요소 설계에 있어 반복적인 접근 방식이 핵심입니다. 첫째, 모든 다양한 발견 사항과 문제점을 고려하는 것은 사실상 불가능합니다.

발생할 수 있는 예외적인 상황들을 파악하고, 이러한 상황들이 재설계를 유도할 수 있습니다. 둘째, 아키텍처 설계자와 개발자들이 애플리케이션 구축에 더욱 깊이 관여할수록, 애플리케이션의 동작과 역할이 어디에 있어야 하는지에 대한 더욱 세밀한 이해를 얻게 됩니다.

건축가는 시스템이나 제품의 수명 주기 전반에 걸쳐 구성 요소를 자주 재구성해야 할 것으로 예상해야 합니다. 이는 신규 시스템뿐만 아니라 잦은 유지 보수가 필요한 모든 시스템에서 마찬가지입니다.

컴포넌트 커플링

구성 요소들이 서로 통신하거나, 한 구성 요소의 변경이 다른 구성 요소에 영향을 미칠 수 있는 경우, 해당 구성 요소들은 서로 결합되어 있다고 합니다.

시스템 구성 요소 간의 결합도가 높을수록 시스템 유지 관리 및 테스트가 어려워집니다([그림 8-11 참조](#)). 따라서 결합도에 세심한 주의를 기울이는 것이 중요합니다.

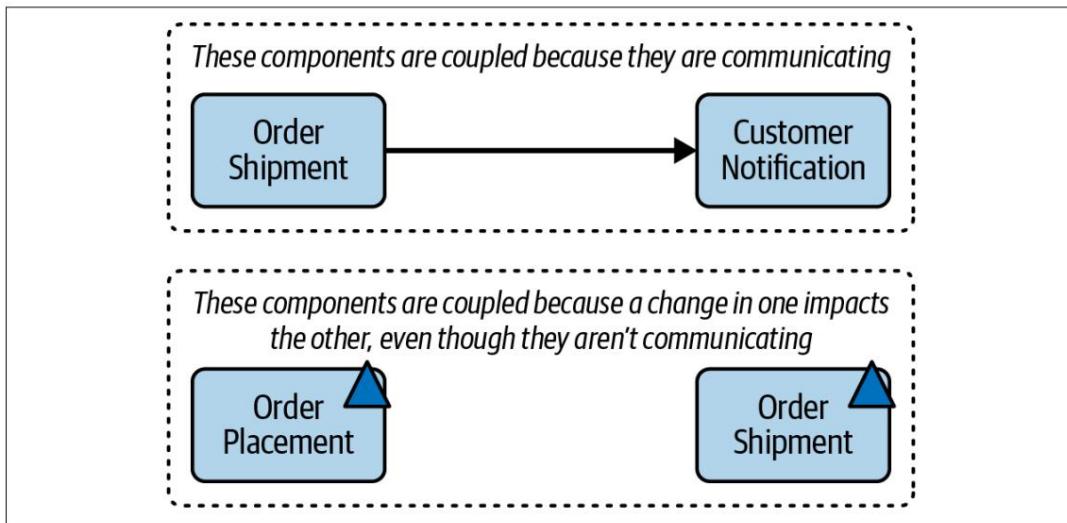


그림 8-11. 구성 요소들은 직접 통신하지 않더라도 서로 연결될 수 있다.

정적 결합이란 구성 요소

들이 서로 동기적으로 통신할 때 발생합니다. 아키텍트는 입력 결합과 출력 결합이라는 두 가지 유형의 결합을 고려해야 합니다.

간접 결합(또는 유입 결합, 팬인 결합)은 다른 구성 요소가 대상 구성 요소에 의존하는 정도를 나타냅니다. 예를 들어, [그림 8-12](#)의 주문 입력 예제에 있는 고객 알림 구성 요소를 생각해 보겠습니다. 고객에게 이메일을 보내려면 주문 접수 및 주문 배송 구성 요소 모두 고객 알림 구성 요소 와 통신해야 합니다 . 즉 , 고객 알림 구성 요소는 주문 접수 및 주문 배송 구성 요소 와 간접적으로 결합되어 있으며 , 간접 결합을 갖는다고 할 수 있습니다.

레벨 2(입력 의존성 수). 구심성 연결은 일반적으로 CA로 표시됩니다.

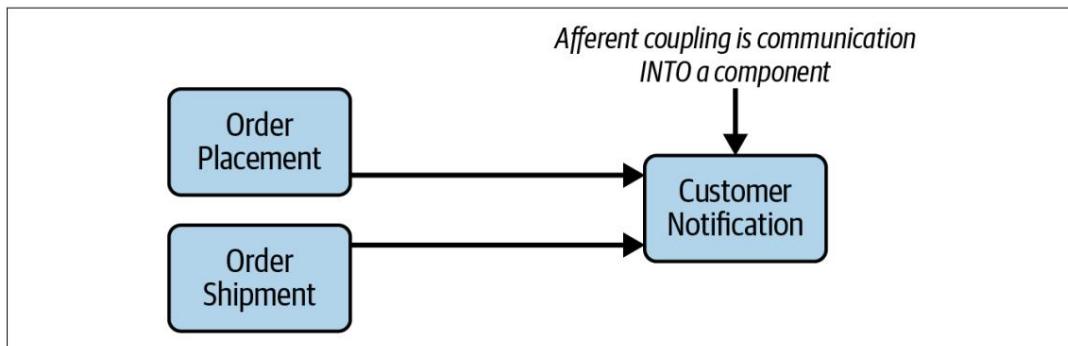


그림 8-12. 구 고객 알림 구성 요소는 다른 구성 요소와 직접적으로 연결되어 있습니다.
성 요소

외부 결합(또는 출력 결합, 팬아웃 결합)은 대상 구성 요소가 다른 구성 요소에 의존하는 정도를 나타냅니다. 예를 들어, [그림 8-13](#)에서 볼 수 있듯이 주문 배치 구성 요소는 주문 처리 구성 요소에 의존하므로 외부 결합되어 있으며, 외부 결합 수준은 1(출력 의존성 수)입니다. 외부 결합은 일반적으로 CE로 표시됩니다.

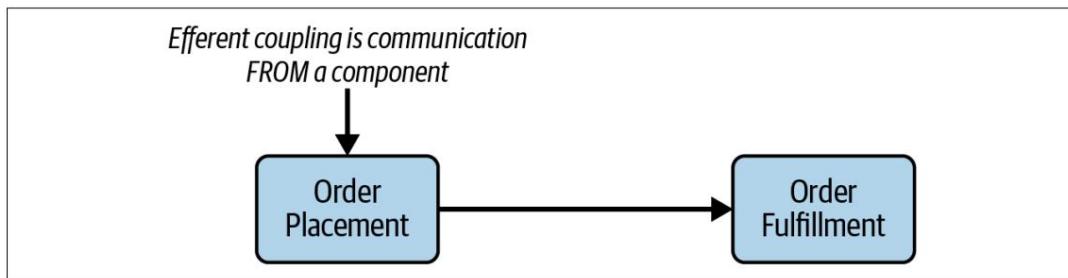


그림 8-13. 주문 배치 구성 요소는 주문 과 직접적으로 연결되어 있습니다.
이행 구성 요소

시간적 결합(Temporal)

Coupling)은 비정적 종속성, 일반적으로 시간 또는 트랜잭션(단일 작업 단위)에 기반한 종속성을 설명합니다. 예를 들어, 시스템이 주문을 처리할 때 주문 접수 구성 요소의 기능은 주문 배송 구성 요소의 기능보다 먼저 호출되어야 합니다. 따라서 이러한 구성 요소들은 시간적으로 결합되어 있다고 합니다.

시간적 결합의 문제점은 현재 시장에 나와 있는 도구로는 이를 감지하기 어렵다는 것입니다. 대부분의 경우 이러한 종류의 결합은 설계 문서나 오류 조건을 통해 파악됩니다.

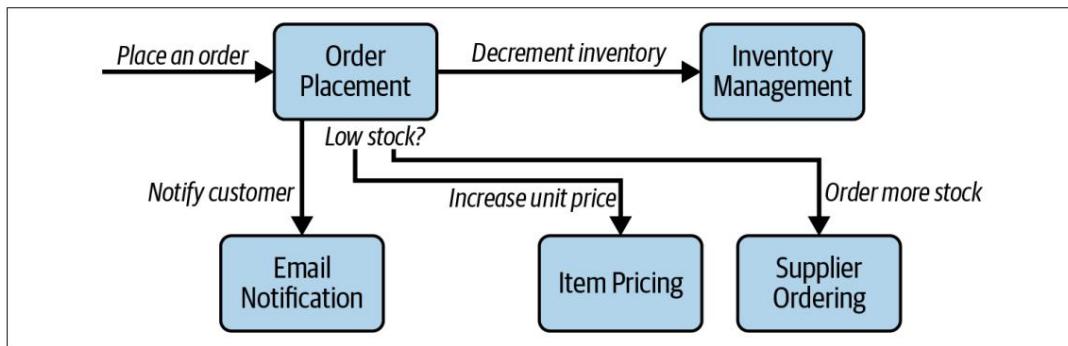
데메테르의 법칙

대부분의 아키텍트는 시스템 설계에서 결합도를 낮추도록 권장받습니다. 구성 요소나 서비스 간의 결합도가 낮을수록 시스템 유지 관리가 용이해지고 테스트가 간편해지며 배포 위험이 줄어듭니다. 또한 변경 사항이 영향을 미치는 구성 요소가 적어 오류 발생 가능성이 낮아지므로 시스템의 전반적인 신뢰성이 향상됩니다.

느슨하게 결합된 시스템을 만드는 한 가지 기법은 데메테르의 법칙 또는 최소 지식의 원칙이라고 불립니다. 그리스 신화에서 여신 데메테르는 온 세상의 곡식을 생산했지만, 사람들이 그 곡식을 어떻게 사용하는지(가축에게 먹이를 주거나 빵을 만드는 등)는 알지 못했습니다. 데메테르는 사실상 세상과 단절된 존재였습니다.

데메테르의 법칙은 구성 요소 또는 서비스가 다른 구성 요소 또는 서비스에 대해 제한적인 지식만 가져야 한다고 명시합니다. 이는 단순하고 당연하게 들릴 수 있지만, 실제로는 그렇지 않습니다. 저희가 무슨 의미인지 보여드리겠습니다.

그림 8-14에 나타난 구성 요소와 그에 상응하는 통신 과정을 살펴보겠습니다. 고객 주문을 접수하면 주문 처리 구성 요소는 재고 관리 구성 요소에 재고를 차감하도록 지시해야 합니다. 재고가 너무 낮아지면 주문 처리 구성 요소는 두 가지 작업을 수행해야 합니다. 첫째, 공급업체 주문 구성 요소에 공급업체로부터 추가 재고를 주문하도록 지시하고, 둘째, 품목 가격 책정 구성 요소에 제한된 공급량을 고려하여 가격을 조정하도록 지시합니다. 마지막으로, 이 모든 과정이 완료되면 주문 처리 구성 요소는 이메일 알림 구성 요소에 고객에게 주문 세부 정보를 이메일로 발송하도록 지시합니다.



당 구성 요소가 주문 처리 구성 요소는 시스템의 나머지 부분과 매우 밀접하게 연결되어 있습니다(그림 8-14). 이는 해 너무 많은 정보를 가지고 있기 때문입니다.

이 아키텍처에서 주문 처리 구성 요소는 다른 구성 요소들과 매우 밀접하게 연결되어 있음을 알 수 있습니다. 주문 처리 구성 요소는 이러한 작업을 직접 수행할 책임은 없지만, 이러한 작업이 필요하다는 것을 알고 있으며, 더 많은 정보를 알수록 연결성이 높아집니다.

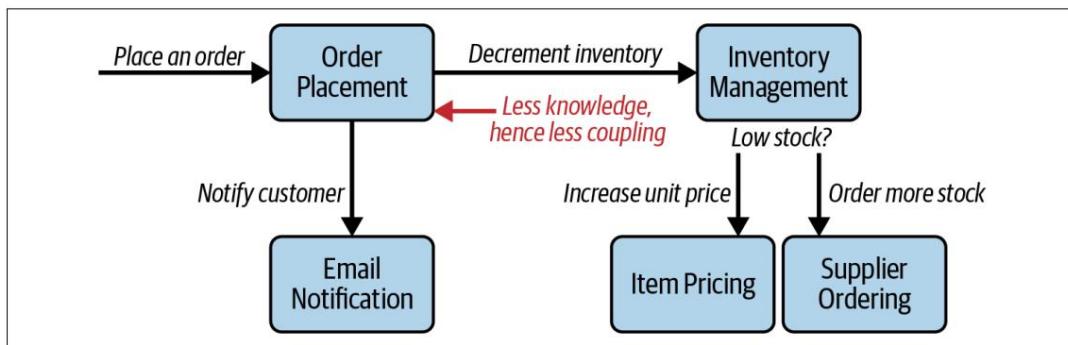
데메테르 법칙의 핵심 아이디어는 시스템의 각 구성 요소가 나머지 시스템에 대해 갖는 정보를 제한하는 것입니다. [그림 8-14](#)에서 주문 처리 구성 요소는 재고를 줄여야 하고, 추가 재고를 주문해야 할 수도 있으며, 품목 가격을 조정해야 할 수도 있고, 고객에게 이메일을 보내야 한다는 것을 알고 있습니다. (정말 많은 정보를 가지고 있죠!) 하지만 이러한 정보를 다른 구성 요소에 분산시킬 수 있다면 어떨까요?

그렇다면 설계자는 해당 구성 요소를 시스템의 나머지 부분에서 효과적으로 분리할 수 있습니다.

데메테르 법칙을 적용하여 구성 요소 간 결합도를 줄이는 방법을 보려면 [그림 8-14의 시스템을 참조하십시오](#). 만약 설계자가 주문 처리 와 재고 관리 사이에 재고 감소 정보를 지연시키는 구성 요소를 추가한다면, 주문 처리 구성 요소의 출력(출력) 결합도는 여전히 동일하게 유지될 것입니다. 따라서 해당 결합 지점은 그대로 유지되어야 합니다.

있는 그대로.

하지만 공급량이 너무 줄어들면 시스템이 재고를 더 주문하고 품목 가격을 조정해야 한다는 사실은 어떻게 처리할까요? 이러한 두 가지 정보는 모두 재고 관리 구성 요소로 이관할 수 있으므로 주문 배치 구성 요소의 결합도를 낮출 수 있습니다. [그림 8-15는](#) 데메테르 법칙을 적용한 후의 아키텍처를 보여줍니다. 특정 기능이 실행되어야 한다는 정보를 제거함으로써 주문 배치 구성 요소는 시스템의 나머지 부분과 덜 결합되게 됩니다.



참조). 주문 배치 구성 요소는 지식이 적을수록 시스템과의 연결성이 낮아집니다(그림 8-15

예리한 독자라면 데메테르 법칙을 적용하면 주문 처리 구성 요소의 결합 수준은 낮아지지만 재고 관리 구성 요소의 결합 수준은 높아진다는 것을 알 수 있을 것입니다. 데메테르 법칙을 적용한다고 해서 시스템 전체의 결합 수준이 반드시 낮아지는 것은 아니며, 오히려 시스템의 다른 부분으로 결합 수준이 재분배되는 경우가 많습니다.

사례 연구: 떠나고, 떠나고, 사라지다—발견하다 구성 요소

팀에 특별한 제약 조건이 없고 범용적인 구성 요소 분해 방식을 찾고 있다면, 액터/액션 접근 방식이 일반적인 해결책으로 효과적입니다.

건축가가 Going, Going, Gone(GGG)에 액터/액션 접근 방식을 적용하면 이 시스템에는 입찰자, 경매 진행자, 그리고 시스템 자체(내부 동작 모델링 기법에서 자주 등장하는 요소)라는 세 가지 주요 역할이 있음을 알게 될 것입니다. 이 역할들은 다음과 같은 주요 액션을 사용하여 시스템과 상호 작용합니다.

입찰자 •

- 실시간 영상 스트림을 시청하세요.
- 실시간 입찰 스트림을 시청하세요.
- 입찰하세요.

경매인

- 시스템에 실시간 입찰을 입력합니다. •
- 온라인 입찰을 접수합니다.
- 판매 완료로 표시하세요.

시스템 •

- 경매 시작.
- 결제하기. • 입찰자 활동 추적하기

이러한 조치를 통해 설계자는 GGG를 위한 일련의 초기 구성 요소를 구축한 다음 이를 반복적으로 개선할 수 있습니다. 그러한 솔루션의 한 예가 [그림 8-16에 나와 있습니다](#).

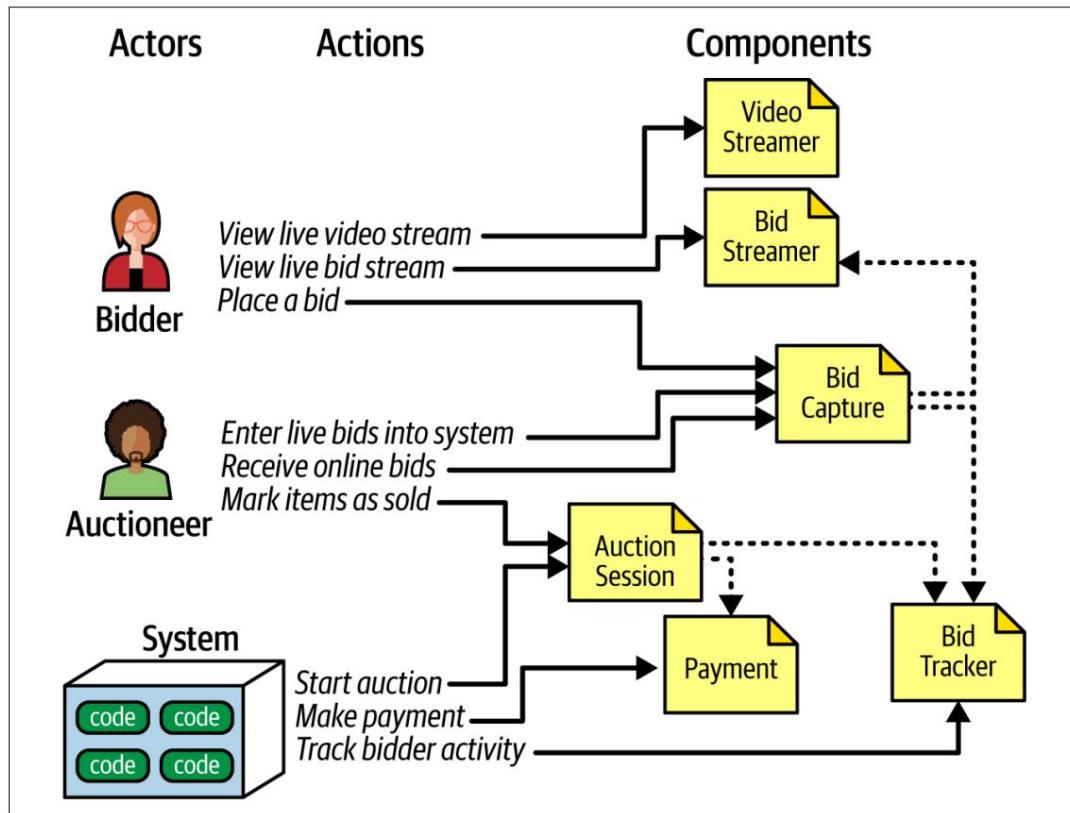


그림 8-16. Going, Going, Gone의 초기 구성 요소 세트

그림 8-16에서 각 역할과 동작은 구성 요소에 매핑됩니다. 구성 요소들은 정보를 공유하기 위해 협력해야 할 수 있습니다. 이 솔루션에 대해 식별한 구성 요소는 다음과 같습니다.

비디오 스트리머가 사용

자들에게 실시간 경매를 스트리밍합니다.

비드 스트리머

Streams는 입찰이 진행되는 동안 사용자에게 입찰 내역을 실시간으로 보여줍니다. Video Streamer 와 Bid Streamer 모두 입찰자에게 경매 내용을 읽기 전용으로 제공합니다.

입찰 캡처 이 구

성 요소는 경매 진행자와 입찰자로부터 입찰을 캡처합니다.

입찰 추적기

입찰 내역을 추적하고 기록 시스템 역할을 합니다.

경매 세션은 경매를 시작합니다

다. 입찰자가 낙찰받고 해당 품목에 대한 경매가 종료되면, 이 구성 요소는 결제 및 정산 단계를 실행하며, 다음 경매 품목에 대한 알림을 입찰자에게 보냅니다.

결제: 신용카

드 결제를 위한 제3자 결제 처리업체.

초기 구성 요소 식별 단계([그림 8-6 참조](#))가 완료되면 아키텍트는 이전에 식별된 아키텍처 특성을 분석하여 구성 요소 설계를 변경할 수 있는 요소가 있는지 확인합니다. 예를 들어, 현재 설계에는 입찰자와 경매 진행자 모두로부터 입찰을 수집하는 입찰 캡처 구성 요소가 포함되어 있습니다.

이는 기능적으로 타당합니다. 누구의 입찰이든 동일한 방식으로 접수하고 처리할 수 있기 때문입니다. 하지만 입찰 접수에는 어떤 기존 아키텍처 특성이 필요할까요? 경매 진행자는 수천 명에 달할 수 있는 입찰자만큼의 확장성이나 유연성이 필요하지 않습니다.

마찬가지로, 경매 진행자는 시스템의 다른 부분보다 신뢰성(연결 끊김 방지) 및 가용성(시스템 가동 유지)과 같은 특정 아키텍처 특성을 더 필요로 할 수 있습니다. 예를 들어, 입찰자가 사이트에 로그인할 수 없거나 연결이 끊기는 것은 사업에 악영향을 미치지만, 경매 진행자에게 이러한 문제가 발생한다면 훨씬 더 심각한 결과를 초래할 것입니다.

입찰자와 경매인이 동일한 건축적 특성에 대해 서로 다른 수준의 요구 사항을 가지고 있음을 고려하여, 건축가는 입찰 포착(Bid Capture) 구성 요소를 입찰 포착(Bid Capture)과 경매인 포착(Auctioneer Capture)의 두 가지 구성 요소로 분리하기로 결정합니다. 업데이트된 디자인은 [그림 8-17에 나와 있습니다.](#)

아키텍트는 경매 진행자용 정보 캡처(Auctioneer Capture)를 위한 새로운 구성 요소를 개발했습니다. 또한 경매 진행자 용 정보 캡처에서 입찰 스트리머(Bid Streamer, 온라인 입찰자에게 실시간 입찰 내역을 보여주는 기능)와 입찰 추적기 (Bid Tracker, 입찰 내역을 관리하는 기능)로 연결되는 정보 링크를 업데이트했습니다. 이제 입찰 추적기는 경매 진행자의 단일 정보 흐름과 입찰자와 다양한 정보 흐름이라는 두 가지 매우 다른 정보 흐름을 통합하는 구성 요소가 됩니다.

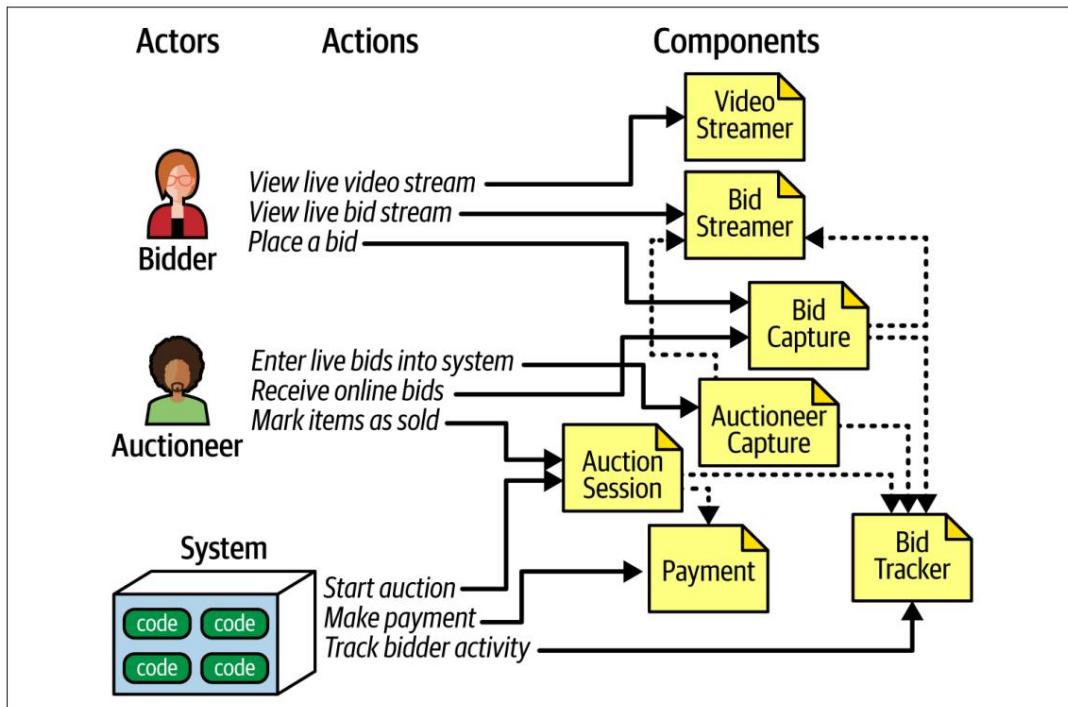


그림 8-17. GGG 사례 연구의 수정된 구성 요소

그림 8-17에 나타난 디자인은 최종 디자인이 아닐 가능성이 높습니다. 새로운 계정 등록 방법, 결제 기능 관리 방법 등 더 많은 요구 사항을 파악해야 합니다. 하지만 이 디자인은 향후 개선을 위한 좋은 출발점을 제공합니다.

이는 GGG 문제를 해결하기 위한 가능한 구성 요소 중 하나일 뿐이며, 반드시 최선의 방법이나 유일한 방법은 아닙니다. 소프트웨어 시스템을 구현하는 방법이 단 하나뿐인 경우는 거의 없습니다. 모든 설계에는 다양한 장단점이 존재합니다. 아키텍트로서 "유일한 정답"을 찾는 데 집착하지 마십시오. 충분한 성능을 발휘하는 설계는 얼마든지 있습니다. 다양한 설계 결정 간의 장단점을 최대한 객관적으로 평가하고, "최악이 아닌" 장단점을 가진 설계를 선택하십시오.

제2부

건축 양식

아키텍처 스타일과 아키텍처 패턴의 차이는 혼동될 수 있습니다. 다음 9 장에서 바로 정의해 드리겠습니다.

건축 양식을 이해하는 것은 신진 건축가들에게 많은 시간과 노력을 요구하는 일입니다. 건축 양식은 중요하고 종류도 매우 다양하기 때문입니다. 건축가는 효과적인 결정을 내리기 위해 다양한 양식과 각 양식에 내재된 장단점을 이해해야 합니다. 각 건축 양식은 잘 알려진 일련의 장단점을 가지고 있으며, 이는 건축가가 특정 비즈니스 문제에 적합한 선택을 하는 데 도움이 됩니다.

