제11장

모듈형 모놀리스 건축 양식

도메인 주도 설계 (DDD) 의 광범위한 도입 과 도메인 분할에 대한 관심 증가 덕분에 모듈형 모놀리식 아키텍처 스타일은 2020년 이 책의 초판 출간 이후 큰 인기를 얻었으며, 이에 따라 2판에는 해당 스타일을 설명하고 평가하는 장을 추가하기로 결정했습니다.

위상수학

이름에서 알 수 있듯이, 모듈형 모놀리스 아키텍처 스타일은 단일체 아키텍처입니다. 따라서 웹 아카이브(WAR) 파일, .NET의 단일 어셈블리, Java 플랫폼의 엔터프라이즈 아카이브(EAR) 파일 등과 같이 단일 소프트웨어 단위로 배포됩니다. 모듈형 모놀리스는 도메인 분할 아키텍처(기술적 역량보다는 비즈니스 도메인을 기준으로 구성됨)로 간주되므로, 그 형태는 도메인 영역별로 기능이 그룹화된 단일 배포 단위로 정의됩니다.

그림 11-1은 모듈형 모놀리스의 일반적인 토폴로지를 보여줍니다.

모듈형 모놀리스의 도메인 집중 특성을 이해하려면 기존의 계층형 아키텍처(10 장 에서 설명)를 생각해 보세요. 이 아키텍처의 구성 요소는 프레젠테이션, 비즈니스, 영속성 계층 등과 같이 기술적 기능에 따라 정의되고 구성됩니다. 예를 들어, 고객 프로필 정보를 관리하는 프레젠테이션 로직은 `com.app.presentation.customer.prole` 네임스페이스를 가진 구성 요소로 표현될 수 있습니다. 네임스페이스의 세 번째 노드는 해당 계층의 기술적 관심사(이 경우 프레젠테이션 계층)를 나타냅니다.

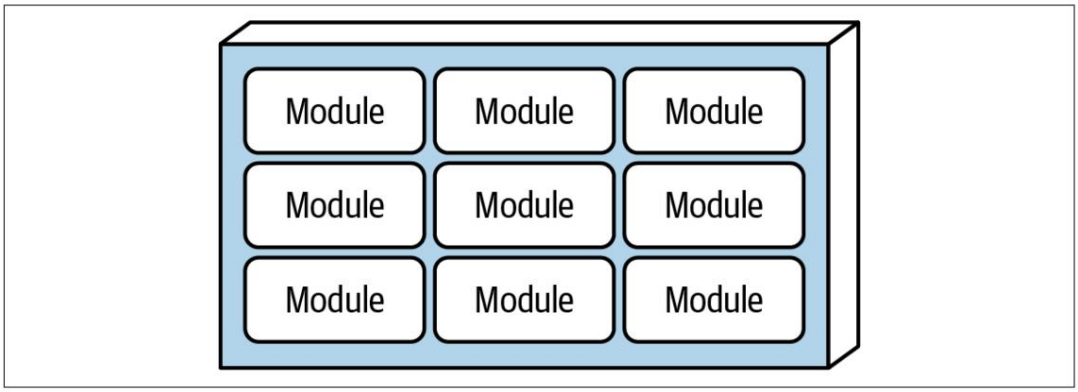


그림 11-1. 모듈형 모놀리식 아키텍처 스타일에서는 기능이 도메인 영역별로 그룹화됩니다.

반대로, 모듈형 모놀리식 구성 요소는 주로 도메인별로 구성됩니다. 따라서 모듈형 모놀리식 아키텍처에서 동일한 고객 프로필 유지 관리 구성 요소는 `com.app.customer.prole` 네임스페이스로 표현됩니다. 여기서 네임스페이스의 세 번째 노드는 기술적인 측면이 아닌 도메인 관련 측면을 나타냅니다. 구성 요소의 복잡성에 따라 네임스페이스는 도메인 관련 측면 이후에 기술적인 측면으로 세분화될 수 있으며, 예를 들어 `com.app.customer.prole.presentation` 또는 `com.app.customer.prole.business`와 같이 표현될 수 있습니다.

스타일 사양

이러한 아키텍처 스타일에서는 도메인(또는 경우에 따라 서브도메인)을 모듈이라고 부릅니다. 모듈은 두 가지 방식으로 구성할 수 있습니다. 가장 간단한 아키텍처는 모놀리식 구조로, 모든 모듈과 해당 논리적 구성 요소가 동일한 코드베이스 내에 포함되며, 네임스페이스 또는 디렉터리 구조로 구분됩니다. 조금 더 복잡한 옵션은 모듈형 구조로, 각 모듈은 독립적인 자체 포함 아티팩트(예: JAR 또는 DLL 파일)로 표현되고, 배포 시 하나의 모놀리식 소프트웨어 단위로 결합됩니다.

소프트웨어 아키텍처의 모든 것과 마찬가지로, 이 두 가지 구조적 옵션 중 하나를 선택하는 것은 여러 요소와 장단점에 따라 달라집니다. 다음 섹션에서는 두 가지 옵션을 모두 설명하고 각각의 장단점을 논의합니다.

일체형 구조

모놀리식 구조에서는 시스템을 구성하는 모든 모듈이 단일 소스 코드 저장소에 포함됩니다. 각 모듈과 관련된 모든 코드는 소프트웨어 배포 또는 릴리스 시 단일 단위로 배포됩니다. 이 구조 옵션은 그림 11-2에 나와 있습니다. 각 모듈은 해당 모듈을 구성하는 구성 요소와 하위 도메인을 포함하는 별도의 상위 디렉터리로 표현됩니다.

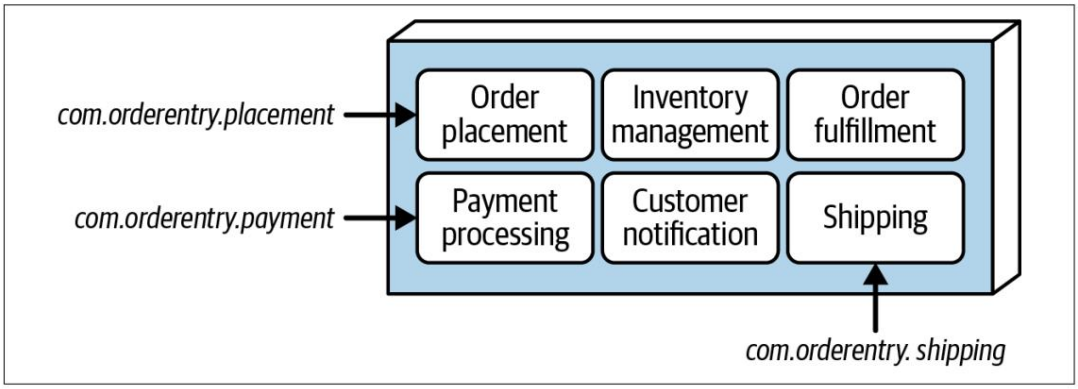


그림 11-2. 일체형 구조 옵션의 예

다음 네임스페이스는 그림 11-2 에 표시된 아키텍처에 대한 모듈형 모놀리스의 형태를 보여줍니다 .

```
com.orderentry.orderplacement
com.orderentry.inventorymanagement
com.orderentry.paymentprocessing
com.orderentry.notification
com.orderentry.fulfillment
com.orderentry.shipping
```

이는 모듈형 모놀리식 아키텍처에 있어 가장 간단한 옵션입니다. 시스템의 모든 소스 코드가 한 곳에 위치하므로 유지 관리, 테스트 및 배포가 더 쉽습니다.
하지만 각 모듈의 경계를 유지하려면 엄격한 관리 체계가 필요합니다(172페이지의 "관리 체계" 참조) . 이러한 구조적 방식은 간단하지만, 개발자들은 모듈 간에 코드를 너무 많이 재사용하거나 모듈 간 통신을 과도하게 허용하는 경향이 있습니다(168페이지의 "모듈 통신" 참조). 이러한 관행은 잘 설계된 모듈형 모놀리스를 구조화되지 않은 거대한 진흙 덩어리로 만들 수 있습니다.

모듈형 구조

모듈형 구조에서는 모듈이 자체 포함된 아티팩트(예: JAR 및 DLL 파일)로 표현된 다음 배포 시 단일 배포 단위로 결합됩니다. 그림 11-3은 Java 플랫폼에서 JAR 파일을 사용하여 이 옵션을 보여줍니다.

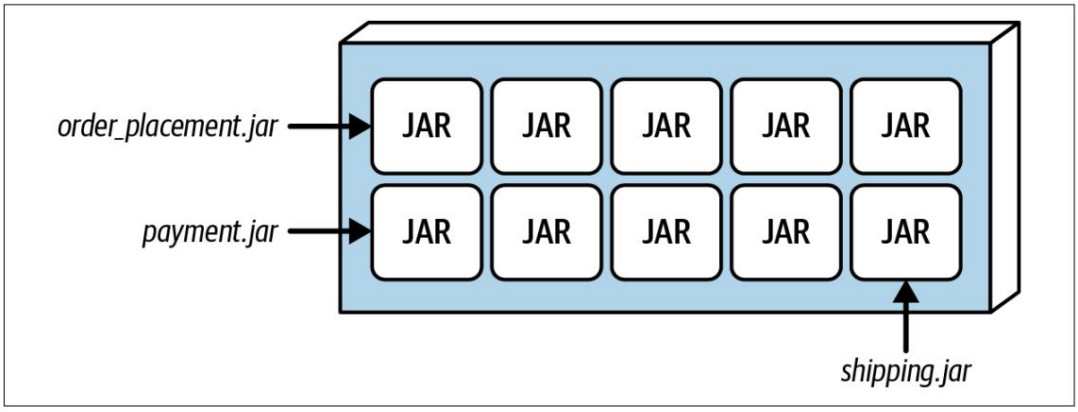


그림 11-3. JAR 파일을 사용한 모듈형 구조 옵션의 예

이러한 구조의 장점은 각 모듈이 독립적이어서 팀이 개별 모듈에서 작업할 수 있다는 점입니다(174 페이지의 "팀 구성 고려 사항" 참조). 심지어 팀 전용 소스 코드 저장소 내에서 작업하는 경우도 많습니다. 이 방식은 모듈들이 서로 상당히 독립적일 때 효과적입니다. 또한 각 모듈에 서로 다른 전문 지식이나 비즈니스 지식이 요구되는 규모가 크고 복잡한 시스템에도 적합합니다. 모듈식 구조를 사용하면 개발자는 코드 재사용을 최소화하고 모듈 간의 과도한 통신을 방지할 수 있습니다(168페이지의 "모듈 통신" 참조). 또한 모듈 간의 경계가 명확해지고 관심사 분리가 더욱 효과적으로 이루어지는 경향이 있습니다.

하지만 이러한 구조적 옵션은 서로 의존적인 모듈들이 통신해야 할 때 효과를 잃습니다. 이러한 경우에는 단일체 구조 방식이 더 효과적입니다.

모듈 통신

이러한 아키텍처 스타일에서 모듈 간 통신은 결코 바람직한 것은 아니지만, 많은 경우 불가피하다는 점을 인정합니다. 예를 들어, 그림 11-2 에 나타난 아키텍처에서 주문 처리 모듈은 재고 관리 모듈과 통신 하여 주문된 품목의 재고를 조정하고 추가 처리(예: 재고가 부족할 경우 추가 재고 주문)를 수행해야 합니다. 또한 주문에 대한 결제를 처리하기 위해 결제 처리 모듈과도 통신해야 합니다. 모듈 간 통신에는 크게 두 가지 옵션이 있으며, 다음 섹션에서 설명합니다.

피어투피어 방식 가장 간

단한 해결책은 모듈 간의 단순한 피어투피어 통신입니다. 이 방식을 사용하면 한 모듈의 클래스 파일이 다른 모듈의 클래스를 인스턴스화하고 해당 클래스에서 필요한 메서드를 호출하여 작업을 수행합니다 (그림 11-4 참조).

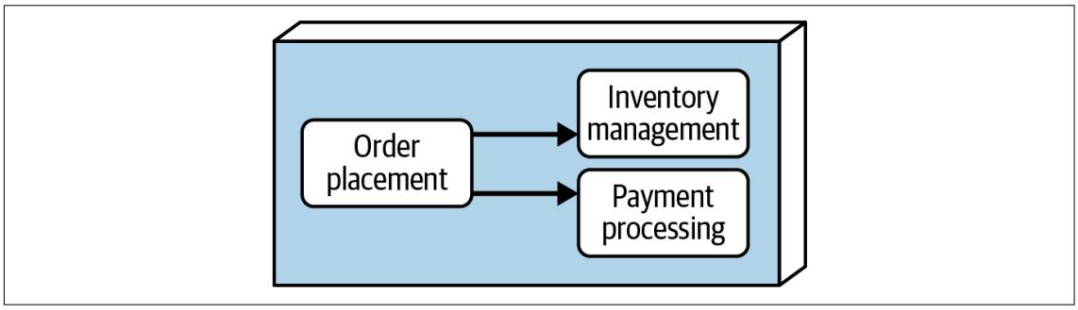


그림 11-4. 모듈 간 피어 투 피어 통신

단일체 방식의 문제점 중 하나는 개발자가 다른 모듈에 포함된 클래스를 너무 쉽게 인스턴스화할 수 있다는 점입니다. 이로 인해 잘 구조화된 아키텍처가 '빅 볼 오브 머드(Big Ball of Mud)'라는 안티패턴으로 쉽게 전락할 수 있습니다(그림 9-1 참조).

모듈식 구조를 사용하는 경우, 다른 모듈에 포함된 클래스는 소스 코드 저장소 내의 별도 디렉터리가 아닌 별도의 외부 아티팩트(JAR 또는 DLL 파일)에 위치할 수 있습니다. 다른 모듈과 통신하는 모듈은 해당 클래스 참조가 없으면 컴파일되지 않으므로 개발자는 해당 모듈 간의 컴파일 타임 종속성을 설정해야 합니다. 이 문제에 대한 일반적인 해결책은 각 모듈이 다른 모듈과 독립적으로 컴파일될 수 있도록 해당 모듈 간에 공유 인터페이스 클래스를 (별도로 공유되는 JAR 또는 DLL 파일에) 생성하는 것입니다. 어떤 방식이든 모듈식 구조 접근 방식을 사용하는 모듈 간의 과도한 통신은 DLL 지옥 (또는 Java 플랫폼에서는 JAR 지옥) 안티패턴을 초래합니다.

미디에이터 접근 방식

은 미디에이터 구성 요소를 사용하여 모듈 간에 추상화 계층을 형성함으로써 모듈을 분리합니다. 미디에이터는 오케스트레이터 역할을 하여 요청을 수락하고 적절한 모듈에 전달합니다. 그림 11-5는 이 접근 방식을 보여줍니다.

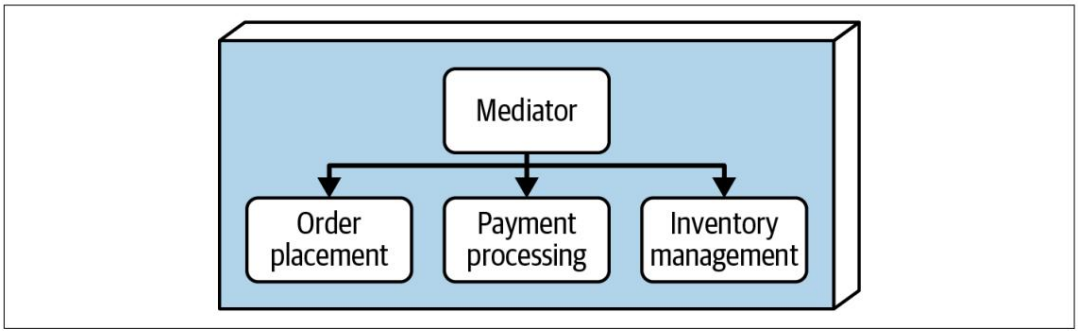


그림 11-5. 서로 e-mediator는 모듈들을 분리시켜 모듈들이 서로 통신할 필요가 없도록 합니다.

예리한 독자라면 중재자 접근 방식이 모듈 간의 결합도를 낮추지만, 각 모듈은 사실상 중재자와 여전히 연결되어 있음을 알 수 있을 것입니다. 이 접근 방식은 모든 결합과 의존성을 제거하는 것은 아니지만, 아키텍처를 단순화하고 모듈 간의 독립성을 유지합니다. 여기서 중요한 점은 다른 모듈의 기능을 호출하기 위해 API 또는 인터페이스가 필요한 것은 종속 모듈이 아니라 중재자라는 사실입니다.

데이터 토폴로지

모듈형 모놀리식 아키텍처는 일반적으로 단일 소프트웨어 단위로 배포되므로, 모놀리식 데이터베이스 토폴로지를 사용하는 것이 일반적입니다. 단일 데이터베이스를 사용하면 데이터가 공유되므로 모듈 간 통신을 줄일 수 있습니다. 그러나 모듈이 서로 독립적이고 특정 기능을 수행하는 경우, 아키텍처 자체는 모놀리식일지라도 각 모듈은 특정 컨텍스트 데이터를 포함하는 자체 데이터베이스를 가질 수 있습니다. 그림 11-6은 이러한 두 가지 데이터베이스 토폴로지 옵션을 보여줍니다.

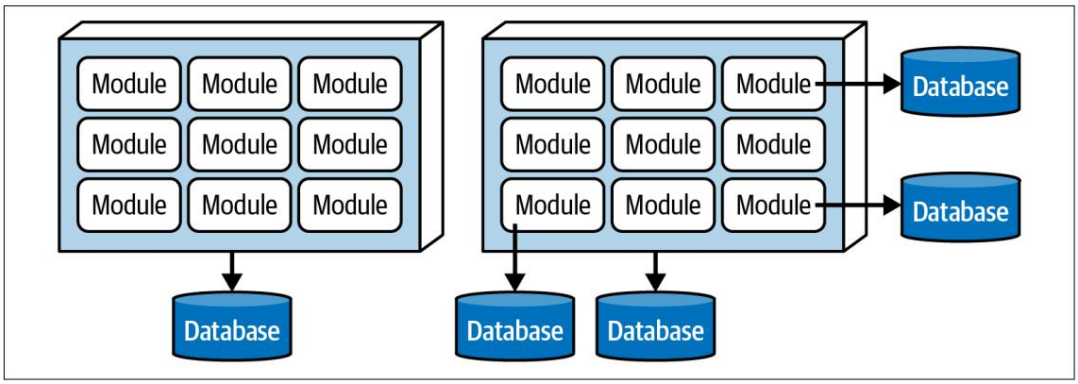


그림 11-6. 데이터는 단일체일 수도 있고, 모듈이 자체 데이터베이스를 가질 수도 있다.

클라우드 고려 사항

모듈형 모놀리식 아키텍처는 클라우드 환경에 배포할 수 있지만(특히 소규모 시스템의 경우), 일반적으로 클라우드 배포에 적합하지는 않습니다. 모놀리식 구조로 인해 클라우드 환경이 제공하는 온디맨드 프로비저닝 기능을 제대로 활용하지 못할 가능성이 높기 때문입니다. 하지만 이러한 아키텍처 방식으로 구현된 소규모 시스템이라도 파일 저장, 데이터베이스, 메시징과 같은 다양한 클라우드 서비스를 활용할 수 있습니다.

일반적인 위험

다른 모든 모놀리식 시스템과 마찬가지로 모듈형 모놀리식 아키텍처 스타일의 주요 위험은 유지 관리, 테스트 및 배포가 제대로 이루어지지 않을 정도로 규모가 커질 수 있다는 점입니다. 모놀리식 아키텍처 자체는 나쁜 것이 아니지만, 규모가 너무 커지면 문제가 발생하기 시작합니다. "너무 크다"는 기준은 시스템마다 다르지만, 시스템이 너무 커졌을 가능성을 나타내는 몇 가지 경고 신호는 다음과 같습니다.

- 변경 사항을 적용하는 데 시간이 너무 오래 걸립니다.
- 시스템의 한 부분을 변경하면 다른 부분에서 예기치 않게 오류가 발생합니다.
- 팀 구성원들이 변경 사항을 적용하는 과정에서 서로 방해를 받습니다.
- 시스템 시작 시간이 너무 오래 걸립니다.

또 다른 위험은 코드 재사용을 지나치게 하는 것입니다. 코드 재사용과 공유는 소프트웨어 개발에 필수적인 부분이지만, 이러한 아키텍처 스타일에서는 코드 재사용이 과도하면 모듈 경계가 모호해져 구조화되지 않은 모놀리식 아키텍처라는 위험한 영역으로 이어질 수 있습니다. 모놀리식 아키텍처는 코드 상호 의존성이 너무 높아 분해할 수 없는 형태입니다.

이러한 아키텍처 스타일에서 또 다른 위험은 모듈 간 과도한 통신입니다. 이상적으로 모듈은 독립적이고 자체적으로 완결되어야 합니다. 앞서 언급했듯이, 특히 복잡한 워크플로우 내에서는 일부 모듈이 다른 모듈과 통신하는 것이 일반적이며 때로는 필수적입니다. 그러나 모듈 간 통신이 지나치게 많다면, 애초에 도메인이 제대로 정의되지 않았을 가능성이 큽니다. 이러한 경우, 복잡한 워크플로우와 상호 의존성을 수용할 수 있도록 도메인을 재정의하는 데 추가적인 노력을 기울일 가치가 있습니다.

통치

모듈형 모놀리식 스타일의 핵심 구성 요소는 모듈이며, 이는 특정 도메인 또는 하위 도메인을 나타내고 일반적으로 디렉터리 구조 또는 네임스페이스(자바 플랫폼에서는 패키지 구조)를 통해 표현됩니다. 따라서 아키텍트가 적용할 수 있는 자동화된 거버넌스의 첫 번째 단계 중 하나는 아키텍처에 사용되는 모듈을 정의하고 규정 준수를 보장하는 것입니다.

자동화된 거버넌스 검사를 작성하기 위해 아키텍트는 Java 플랫폼용 ArchUnit , .NET 플랫폼용 ArchUnitNet 및 NetArchTest , Python용 PyTestArch , TypeScript 및 JavaScript용 TSArch를 비롯한 다양한 도구를 사용합니다 . 예제 11-1 의 의사 코드는 그림 11-2 에 표시된 아키텍처 예제의 모든 소스 코드가 시스템의 각 정의된 모듈을 나타내는 나열된 네임스페이스 중 하나에 속하도록 보장합니다.

예제 11-1. 시스템이 정의한 모듈을 준수하는지 확인하기 위한 의사 코드

```
# 다음 네임스페이스는 시스템의 모듈을 나타냅니다. LIST module_list =

{ com.orderentry.orderplacement,
    com.orderentry.inventorymanagement,
    com.orderentry.paymentprocessing,
    com.orderentry.notification,
    com.orderentry.fulfillment,
    com.orderentry.shipping }


# 시스템에 있는 네임스페이스 목록을 가져옵니다.
LIST namespace_list = get_all_namespaces(root_directory)

# 모든 네임스페이스가 나열된 모듈 중 하나로 시작하는지 확인하세요
namespace_list의 각 네임스페이스 {
    만약 네임스페이스가 모듈 목록으로 시작 하지 않는다면 {
        send_alert(네임스페이스)
    }
}
```

개발자가 정의된 모듈 및 해당 네임스페이스(또는 디렉터리) 외부에 추가적인 상위 네임스페이스 또는 디렉터리를 생성하는 경우, 소스 코드가 아키텍처를 준수하지 않는다는 경고를 받게 됩니다.

이러한 형태의 거버넌스는 이 아키텍처 스타일의 단일 구조 옵션(167페이지의 "단일 구조" 참조) 과 잘 작동 하지만, 코드가 동일한 단일 소스 코드 저장소에 포함되어 있지 않을 수 있는 모듈형 구조 옵션(168페이지의 "모듈형 구조" 참조) 에서는 어려움이 있습니다. 모듈형 구조 옵션을 사용하는 경우, 예제 11-2에서와 같이 각 모듈을 개별적으로 테스트해야 합니다 .

예제 11-2. InventoryManagement 모듈 유효성 검사를 위한 의사 코드

```
# 시스템에 있는 네임스페이스 목록을 가져옵니다.
LIST namespace_list = get_all_namespaces(root_directory)

# 모든 네임스페이스가 com.orderentry.inventorymanagement로 시작하는지 확인합니다. FOREACH namespace
IN namespace_list { IF NOT
    namespace.starts_with("com.orderentry.inventorymanagement") {
        send_alert(네임스페이스)
    }
}
```

모듈형 모놀리식 아키텍처를 관리하는 또 다른 방법은 모듈 간 통신량을 제어하는 것입니다. "과도한" 통신량을 정의하는 것은 매우 주관적이며 시스템마다 다르지만, 대부분의 경우 아키텍트는 모듈 간 상호 의존성을 최소화하도록 노력해야 합니다. 예시 그림 11-3은 최대 총 상호 의존성이 5개의 통신(또는 결합) 지점을 초과하지 않도록 하는 의사 코드를 보여줍니다.

예제 11-3. 특정 모듈의 전체 종속성 개수를 제한하는 의사 코드

```
# 디렉토리 구조를 탐색하여 모듈과 해당 모듈에 포함된 소스 코드 파일을 수집합니다. # LIST module_list = { com.orderentry.orderplacement,
com.orderentry.inventorymanagement,
com.orderentry.paymentprocessing,
    com.orderentry.notification,
    com.orderentry.fulfillment, com.orderentry.shipping }


MAP 모듈_소스_파일_맵
module_list의 각 모듈 {
  LIST source_file_list = get_source_files(module)
  모듈, source_file_list를 module_source_file_map에 추가 }


# 각 소스 파일에 대한 참조 개수를 확인하고, 시스템의 총 종속성 개수가 5개보다 크면 경고를 보냅니다.

각 모듈, source_file_list IN module_source_file_map {
  FOREACH source_file IN source_file_list { incoming
    count = used_by_other_module(source_file, module_source_file_map) { outgoing_count =
    uses_other_module(source_file) { total_count = incoming count +
    outgoing count
  }
  total_count가 5 보다 크면
    send_alert(module, total_count) 를 보냅니다.
  }
}
```

자동화된 거버넌스의 마지막 형태는 특정 모듈이 다른 모듈과 통신하는 것을 제한함으로써 모듈 간의 독립성을 보장하는 것입니다. 예를 들어, 그림 11-2 에서 주문 처리 모듈은 배송 모듈 과 통신해서는 안 됩니다 . 예제 11-4는 이러한 종속성을 제어하는 ArchUnit의 Java 코드를 보여줍니다.

예제 11-4. 특정 모듈 간의 종속성 제한을 관리하기 위한 ArchUnit 코드

```
public void order_placement_cannot_access_shipping() {

    noClasses().that() .resideInAPackage("..com.orderentry.orderplacement..") .should().accessClassesThat() .resideInAPackage("..com.orde
}
```

팀 토폴로지 고려 사항

모듈형 모놀리식 아키텍처는 도메인 분할 아키텍처로 간주되므로, 팀 구성 또한 도메인 영역별로 정렬될 때(예: 전문성을 갖춘 교차 기능 팀) 가장 효과적입니다. 도메인 기반 요구 사항이 발생하면, 해당 도메인에 집중하는 교차 기능 팀이 프레젠테이션 로직부터 데이터베이스에 이르기까지 해당 기능을 함께 개발할 수 있습니다. 반대로, UI 팀, 백엔드 팀, 데이터베이스 팀 등과 같이 기술 범주별로 구성된 팀은 도메인 분할이라는 특성 때문에 이 아키텍처 스타일과 잘 맞지 않습니다.

도메인 기반 요구사항을 기술적으로 조직된 팀에 할당하려면 많은 의사소통과 협업이 필요하지만, 이는 종종 어려운 과제로 남습니다.

다음은 앞서 설명한 특정 팀 구성을 맞추기 위한 몇 가지 고려 사항입니다.
151페이지의 "팀 토폴로지 및 아키텍처"에서 모듈형 모놀리스 스타일을 다루고 있습니다.

흐름 중심 팀은 일반적으로 시스

템의 시작부터 끝까지 전체 흐름을 관리하며, 모듈형 모놀리스의 단일체적이고 일반적으로 자급자족적인 형태와 잘 어울립니다.

이 스타일은 높은

수준의 모듈성과 관심사 분리 덕분에 팀 구성의 유연성을 높여줍니다. 전문가와 여러 분야에 걸친 팀 구성원들이 기존 모듈에 미치는 영향을 최소화하면서 시스템에 추가 모듈을 도입하여 제안을 하고 실험을 수행할 수 있습니다.

복잡한 하위 시스템 팀

모듈형 모놀리식 아키텍처에서 각 모듈은 일반적으로 도메인 또는 하위 도메인(예: 결제 처리) 에 따라 특정 역할을 수행합니다 . 이는 복잡한 하위 시스템 팀 구성에 적합한데, 각 팀 구성원이 다른 팀 구성원(및 모듈)과 독립적으로 복잡한 도메인 또는 하위 도메인 처리에 집중할 수 있기 때문입니다.

플랫폼 팀 개발자

는 이러한 아키텍처 스타일에서 발견되는 높은 수준의 모듈성 덕분에 공통 도구, 서비스, API 및 작업을 활용하여 플랫폼 팀 토폴로지의 이점을 누릴 수 있습니다.

스타일 특징

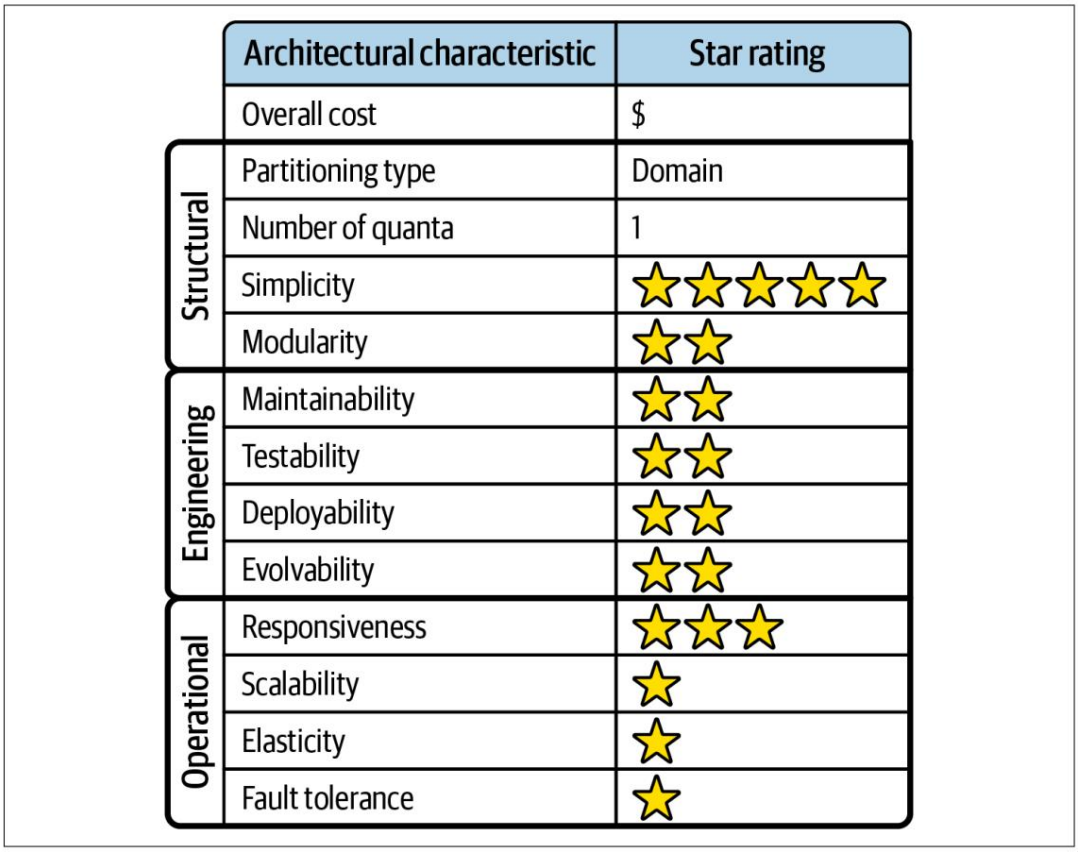
그림 11-7의 특성 평가표에서 별 1개는 특정 아키텍처 특성이 해당 아키텍처에서 제대로 지원되지 않음을 의미하고, 별 5개는 해당 아키텍처 특성이 아키텍처 스타일에서 가장 강력한 기능 중 하나임을 의미합니다. 평가표에 포함된 특성은 4 장 에서 설명하고 정의합니다 .

모듈형 모놀리식 아키텍처 스타일은 애플리케이션 로직이 모듈로 분할되기 때문에 도메인 분할 아키텍처입니다. 일반적으로 모놀리식 배포로 구현되므로 아키텍처 퀀텀은 일반적으로 1입니다.

모듈형 모놀리식 아키텍처 스타일의 주요 강점은 전반적인 비용 효율성, 단순성 및 모듈성입니다. 모놀리식 구조 특성상, 이러한 아키텍처는 분산 아키텍처 스타일에서 나타나는 복잡성을 갖지 않습니다. 더 단순하고 이해하기 쉬우며, 구축 및 유지 관리 비용 또한 상대적으로 저렴합니다. 아키텍처의 모듈성은 도메인과 서브도메인을 나타내는 다양한 모듈 간의 관심사 분리를 통해 구현됩니다.

배포 용이성과 테스트 용이성은 별 2개에 불과하지만, 모듈형 모놀리식 아키텍처는 모듈성 덕분에 계층형 아키텍처보다 약간 더 높은 점수를 받았습니다. 하지만 이 아키텍처 스타일 역시 모놀리식 아키텍처이므로, 절차, 위험, 배포 빈도, 테스트의 완성도 측면에서 부정적인 영향을 미칩니다.

모듈형 모놀리식 아키텍처는 탄력성과 확장성이 매우 낮습니다(별 1개). 이는 주로 모놀리식 배포 방식 때문입니다. 모놀리식 아키텍처 내에서 특정 기능을 다른 기능보다 더 효율적으로 확장하는 것이 가능하지만, 이를 위해서는 멀티스레딩, 내부 메시징, 기타 병렬 처리 방식과 같은 매우 복잡한 설계 기법이 필요하며, 이러한 아키텍처는 이러한 기법에 적합하지 않습니다.



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

그림 11-7. 모듈형 모놀리스의 건축적 특성에 대한 별점 평가

모듈형 모놀리식 아키텍처는 단일 구조로 배포되기 때문에 내결함성을 지원하지 않습니다. 아키텍처의 작은 부분에서 메모리 부족 오류가 발생하면 전체 애플리케이션이 다운됩니다. 또한 대부분의 모놀리식 애플리케이션과 마찬가지로 평균 복구 시간(MTTR)이 길어 전반적인 가용성에 영향을 미치며, 시작 시간은 일반적으로 몇 분 단위로 측정됩니다.

사용 시점

모듈형 모놀리식 아키텍처는 단순하고 비용이 저렴하기 때문에 예산과 시간이 빠듯한 상황에서 좋은 선택입니다. 또한 새로운 시스템을 처음 구축할 때도 적합합니다. 시스템의 아키텍처 방향이 아직 불분명한 경우, 분산 아키텍처로 바로 넘어가는 것보다 모듈형 모놀리식으로 시작한 후 서비스 기반(14장 참조) 이나 마이크로서비스(18 장 참조)와 같이 더 복잡하고 비용이 많이 드는 분산 아키텍처로 전환하는 것이 더 효과적인 경우가 많습니다.

모듈형 모놀리식 아키텍처는 특정 도메인에 집중하는 팀, 예를 들어 전문성을 갖춘 여러 부서로 구성된 팀에게 적합합니다. 이를 통해 각 팀은 아키텍처 내의 특정 모듈에 처음부터 끝까지 집중할 수 있으며, 다른 도메인 팀과의 협업은 최소화할 수 있습니다. 또한, 이러한 아키텍처 스타일은 시스템 변경 사항의 대부분이 도메인 기반인 경우(예: 고객 위시리스트 항목에 유효기간 추가)에도 매우 적합합니다.

마지막으로, 모듈형 모놀리스는 도메인 분할 아키텍처이기 때문에 DDD(도메인 주도 개발)를 수행하는 팀에 매우 적합합니다.

사용하지 말아야 할 경우

이러한 아키텍처 스타일을 사용하지 않는 주된 이유는 시스템이나 제품이 확장성, 탄력성, 가용성, 내결함성, 응답성 및 성능과 같은 특정 운영 특성에 대해 높은 수준을 요구할 때입니다. 대부분의 모놀리식 아키텍처와 마찬가지로 모듈형 모놀리식 아키텍처는 이러한 아키텍처적 요구 사항에 적합하지 않습니다.

사용자 인터페이스나 데이터베이스 기술을 지속적으로 교체하는 등 변경 사항의 대부분이 기술적인 측면인 경우에는 모듈형 모놀리식 아키텍처 사용을 피해야 합니다.
이 아키텍처는 도메인별로 분할되어 있기 때문에 이러한 변경 사항은 모든 모듈에 영향을 미치며 일반적으로 도메인 팀 간의 상당한 의사소통과 조정이 필요합니다. 이러한 상황에서는 계층형 아키텍처 스타일(10장 참조) 이 훨씬 더 나은 선택입니다.

예시 및 사용 사례

EasyMeals는 바쁜 하루를 마치고 집에 돌아와 직접 요리할 시간이 부족한 직장인들을 위해 새롭게 등장한 배달 기반 동네 레스토랑입니다. 배고픈 고객들은 맛있는 저녁 식사를 온라인으로 주문하고 한 시간 안에 집 앞까지 배달받을 수 있습니다.

규모가 작은 지역 식당이기 때문에 높은 확장성이나 신속한 대응이 필요하지 않습니다.
예산이 제한적이기 때문에 정교한 소프트웨어 시스템에 많은 돈을 투자하고 싶지 않습니다. 이러한 비즈니스 상황을 고려할 때, 모듈형 모놀리스 아키텍처는 EasyMeals에 적합한 선택입니다.

그림 11-8은 모듈형 모놀리스 아키텍처 스타일을 사용하여 EasyMeals의 간단한 레스토랑 관리 시스템이 어떤 모습일지 보여줍니다.

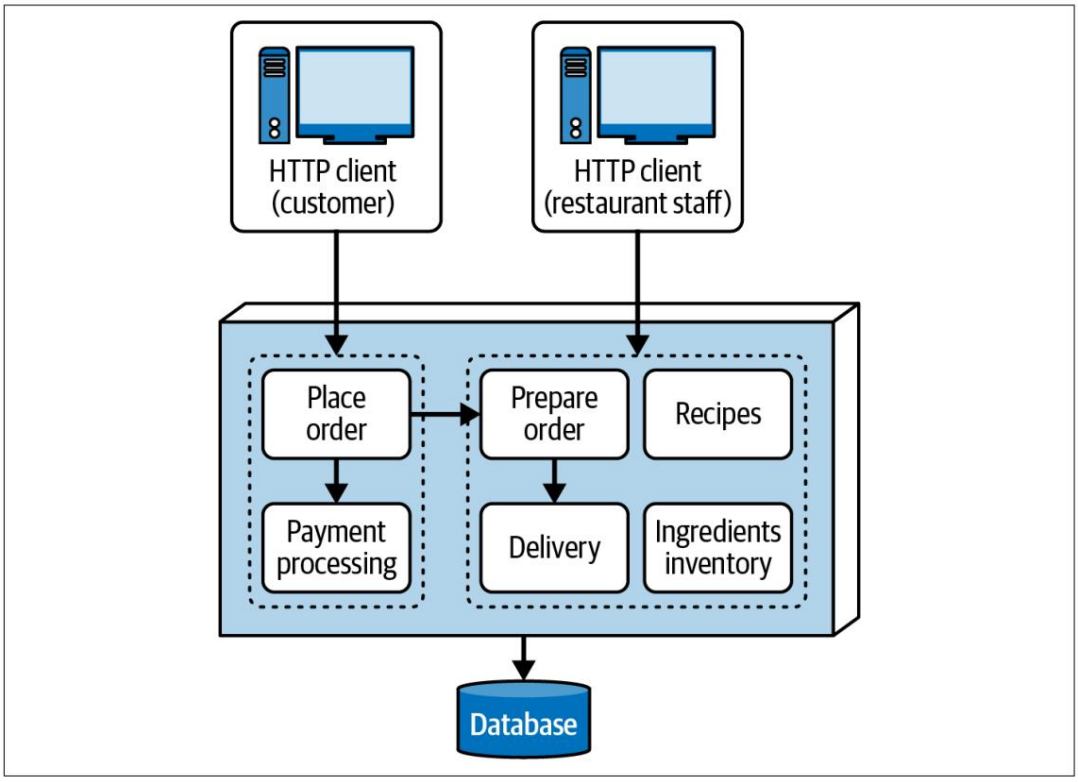


그림 11-8. 모듈형 모놀리스 스타일을 사용하는 소규모 레스토랑 주문 및 관리 시스템

고객은 별도의 사용자 인터페이스를 통해 주문 및 결제 처리 모듈에 접근합니다. 다음 네임스페이스는 이 시스템의 각 모듈을 나타냅니다.

```
com.easymeals.placeorder
com.easymeals.payment
com.easymeals.prepareorder
com.easymeals.delivery
com.easymeals.recipes
com.easymeals.inventory
```

주문 모듈 을 통해 각 고객은 메뉴를 보고, 항목을 선택하고, 이름, 주소 및 결제 정보를 입력한 후 주문을 제출할 수 있습니다. 이 모듈의 구성 요소는 다음과 같은 네임스페이스로 표현될 수 있으며, 각 주요 기능을 구현하는 소스 코드가 포함됩니다.

```
com.easymeals.placeorder.menu
com.easymeals.placeorder.shoppingcart
com.easymeals.placeorder.customerdata
com.easymeals.placeorder.paiddata
com.easymeals.placeorder.checkout
```

이 예시는 모듈형 모놀리스의 모듈이 하나 이상의 구성 요소로 이루어져 있음을 보여줍니다(8 장 참조).

PaymentProcessing 모듈 은 결제 처리를 담당합니다. EasyMeals는 신용카드, 직불카드, PayPal을 결제 수단으로 제공하며, 모듈식 아키텍처 덕분에 포인트 결제와 같은 추가 결제 방식을 쉽게 추가할 수 있습니다. 고객은 PlaceOrder 모듈에서 결제 정보를 입력하고 , PlaceOrder 모듈은 해당 정보를 PaymentProcessing 모듈로 전달합니다. 이 모듈의 구성 요소는 다음과 같은 네임스페이스로 표현될 수 있습니다.

com.easymeals.payment.creditcard
com.easymeals.payment.debitcard
com.easymeals.payment.paypal

주문 결제가 완료되면 주문 접수(PlaceOrder) 모듈은 주문 준비(Prepare Order) 모듈 과 통신하여 주방 직원에게 전체 주문 내역을 표시합니다. 조리가 완료되면 주방 직원은 주문을 배송 준비 완료로 표시하고, 주문은 배송(Delivery) 모듈로 전송됩니다. 다음 네임스페이스는 주문 준비 모듈 내의 구성 요소를 나타냅니다.

com.easymeals.prepareorder.displayorder
com.easymeals.prepareorder.ready

배송 모듈 은 주문에 배송 담당자를 배정하고 배송 주소를 지정합니다. 배송 담당자는 이 모듈을 통해 주문을 배송 완료로 표시하여 해당 주문의 처리 과정을 종료하고, 발생한 문제(예: 대문 앞의 공격적인 개 또는 부재중인 고객)를 기록할 수 있습니다. 다음 네임스페이스는 배송 모듈 내의 구성 요소를 나타냅니다.

com.easymeals.delivery.assign
com.easymeals.delivery.issues
com.easymeals.delivery.complete

레시피 모듈 을 통해 요리사와 관리 직원은 메뉴에 항목을 추가하고 각 메뉴 항목에 대한 재료 목록과 계량 정보를 관리할 수 있습니다.
다음 네임스페이스는 레시피 모듈 내의 구성 요소를 나타냅니다.

com.easymeals.recipes.view
com.easymeals.recipes.maintenance

마지막으로, 재료 재고 관리 모듈은 메뉴에 있는 레시피에 필요한 재료가 충분히 확보되어 있는지 확인합니다. 이 모듈은 다른 모듈보다 조금 더 복잡한데, 정교한 AI 구성 요소를 통해 판매량을 예측하여 주간 재료 조달 프로세스를 자동화합니다.

다음 네임스페이스는 IngredientsInventory 모듈 내의 구성 요소를 나타냅니다.

com.easymeals.inventory.maintenance
com.easymeals.inventory.forecasting
com.easymeals.inventory.ordering
com.easymeals.inventory.suppliers
com.easymeals.inventory.invoices

이게 전부입니다! 모듈형 모놀리스의 단순성과 높은 수준의 모듈성은 버그를 수정하거나 새로운 기능을 추가하기 위해 코드를 찾고 유지 관리하는 것을 상대적으로 쉽게 만들어줍니다. 이는 이러한 단순하고 직관적인 아키텍처 스타일의 강력한 장점을 보여줍니다.