제13장

마이크로커널 아키텍처 스타일

마이크로커널 아키텍처 스타일(플러그인 아키텍처라고도 함)은 수십 년 전에 개발되었으며 오늘날에도 여전히 널리 사용되고 있습니다. 이 아키텍처 스타일은 제품 기반 애플리케이션, 즉 단일 모놀리식 배포 형태로 패키징되어 다운로드 및 설치가 가능한 애플리케이션에 적합합니다. 일반적으로 이러한 애플리케이션은 타사 제품으로 고객 사이트에 설치됩니다. 하지만 제품이 아닌 맞춤형 비즈니스 애플리케이션, 특히 맞춤화가 필요한 문제 영역에도 널리 사용됩니다. 예를 들어, 각 주마다 고유한 규정을 적용해야 하는 미국 보험 회사나 다양한 법률 및 물류 규정을 준수해야 하는 국제 운송 회사는 모두 이 아키텍처 스타일의 이점을 누릴 수 있습니다.

위상수학

마이크로커널 방식은 코어 시스템과 플러그인이라는 두 가지 구성 요소로 이루어진 비교적 단순한 모놀리식 아키텍처입니다. 애플리케이션 로직은 독립적인 플러그인 구성 요소와 기본 코어 시스템으로 분리되어, 애플리케이션 기능을 격리하고 확장성, 적응성 및 사용자 정의 처리 로직을 제공합니다.
그림 13-1은 마이크로커널 아키텍처 스타일의 기본 토폴로지를 보여줍니다.

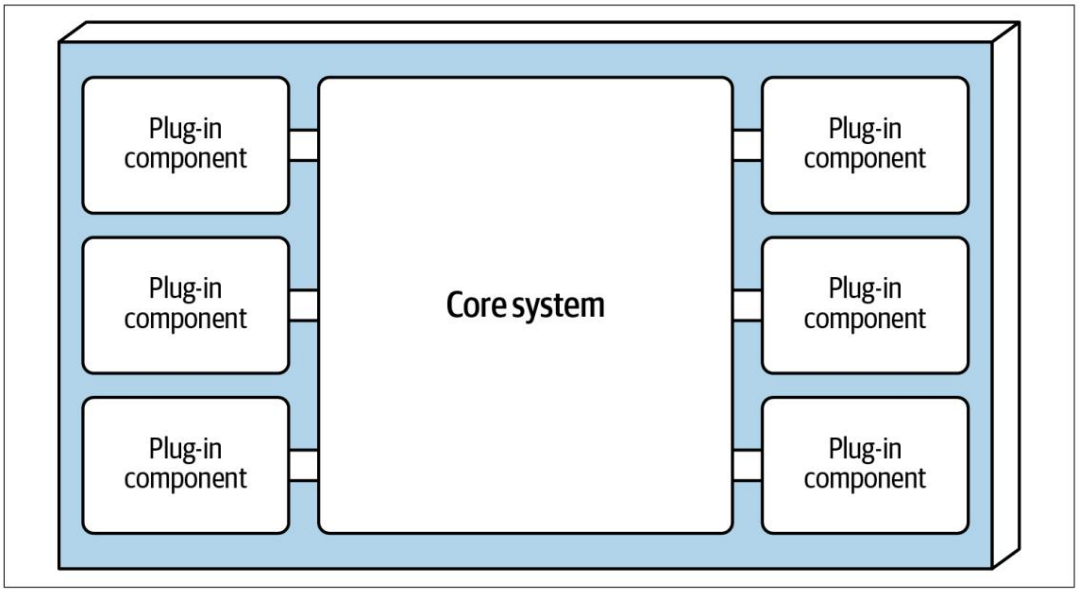


그림 13-1. 마이크로커널 아키텍처 스타일의 기본 구성 요소

스타일 사양

마이크로커널 아키텍처의 핵심은 코어 시스템과 플러그인이라는 두 가지 유형의 구성 요소로 이루어져 있습니다.

핵심 시스템 은

시스템을 실행하는 데 필요한 최소한의 기능으로 공식적으로 정의됩니다. 좋은 예로 이클립스 IDE를 살펴보겠습니다. 이클립스의 핵심 시스템은 기본적인 텍스트 편집기일 뿐입니다. 파일을 열고, 텍스트를 변경하고, 파일을 저장하는 것이 전부입니다. 플러그인을 추가해야 비로소 이클립스가 유용한 제품이 됩니다.

하지만 핵심 시스템에 대한 또 다른 정의는 정상적인 실행 경로, 즉 사용자 정의 처리가 거의 또는 전혀 필요 없는 애플리케이션의 일반적인 처리 흐름입니다. 마이크로커널 아키텍처는 애플리케이션의 순환 복잡도를 핵심 시스템에서 분리하여 별도의 플러그인 구성 요소에 배치합니다. 이를 통해 확장성과 유지 관리성이 향상될 뿐만 아니라 테스트 용이성도 높아집니다.

좀 더 자세히 살펴보기 위해 7 장 에서 소개했던 전자 기기 재활용 애플리케이션인 Going Green으로 돌아가 보겠습니다 . 여러분이 Going Green 애플리케이션을 개발하고 있다고 가정해 봅시다. 이 애플리케이션은 수거된 각 전자 기기를 특정 사용자 지정 평가 규칙에 따라 평가해야 합니다. 이러한 처리를 위한 Java 코드는 다음과 같을 수 있습니다.

```
public void assessDevice(String deviceID) { if
    (deviceID.equals("iPhone6s")) {
        assessiPhone6s();
    } 그렇지 않고 기기 ID가 "iPad1"인 경우 , iPad1을 평가
        합니다. } 그렇지 않고
    기기 ID가 "Galaxy5" 인 경우 , Galaxy5를 평가합니다. }

            ...
        ...
    }
}
```

클라이언트별 맞춤 설정(순환적 복잡성이 매우 큼)을 코어 시스템에 모두 넣는 대신, 평가 대상 전자 기기마다 별도의 플러그인 구성 요소를 만들 수 있습니다. 특정 클라이언트 플러그인 구성 요소는 독립적인 기기 로직을 나머지 처리 흐름과 분리할 뿐만 아니라 확장성도 제공합니다. 평가할 새 기기를 추가하는 것은 새 플러그인 구성 요소를 추가하고 레지스트리를 업데이트하는 것만으로 간단하게 해결할 수 있습니다. 마이크로커널 아키텍처 방식을 사용하면 전자 기기를 평가할 때 코어 시스템은 다음 수정된 소스 코드에서 볼 수 있듯이 해당 기기 플러그인을 찾아서 호출하기만 하면 됩니다.

```
public void assessDevice(String deviceID) {
        String plugin = pluginRegistry.get(deviceID); Class<?>
        theClass = Class.forName(plugin); Constructor<?>
        constructor = theClass.getConstructor(); DevicePlugin devicePlugin =
        (DevicePlugin)constructor.newInstance();

        devicePlugin.assess();
}
```

이 예시에서는 특정 전자 장치를 평가하기 위한 모든 복잡한 규칙과 지침이 핵심 시스템에서 범용적으로 실행될 수 있는 독립형 플러그인 구성 요소에 자체적으로 포함되어 있습니다.

규모와 복잡성에 따라 핵심 시스템을 계층형 아키텍처 또는 모듈형 모놀리식(그림 13-2 참조)으로 구현할 수 있습니다. 경우에 따라서는 핵심 시스템을 도메인별로 분리하여 배포하는 도메인 서비스로 분할하고, 각 도메인 서비스에는 해당 도메인에 특화된 플러그인 구성 요소를 포함할 수도 있습니다. 이 예시에서는 Going Green 시스템을 계층형 아키텍처로 구현한다고 가정합니다.

결제 처리가 핵심 시스템을 나타내는 도메인 서비스라고 가정해 보겠습니다 .
각 결제 방식(신용카드, 페이팔, 스토어 크레딧, 기프트 카드, 구매 주문서)마다 해당 결제 영역에 특화된 별도의 플러그인 구성 요소가 있습니다.

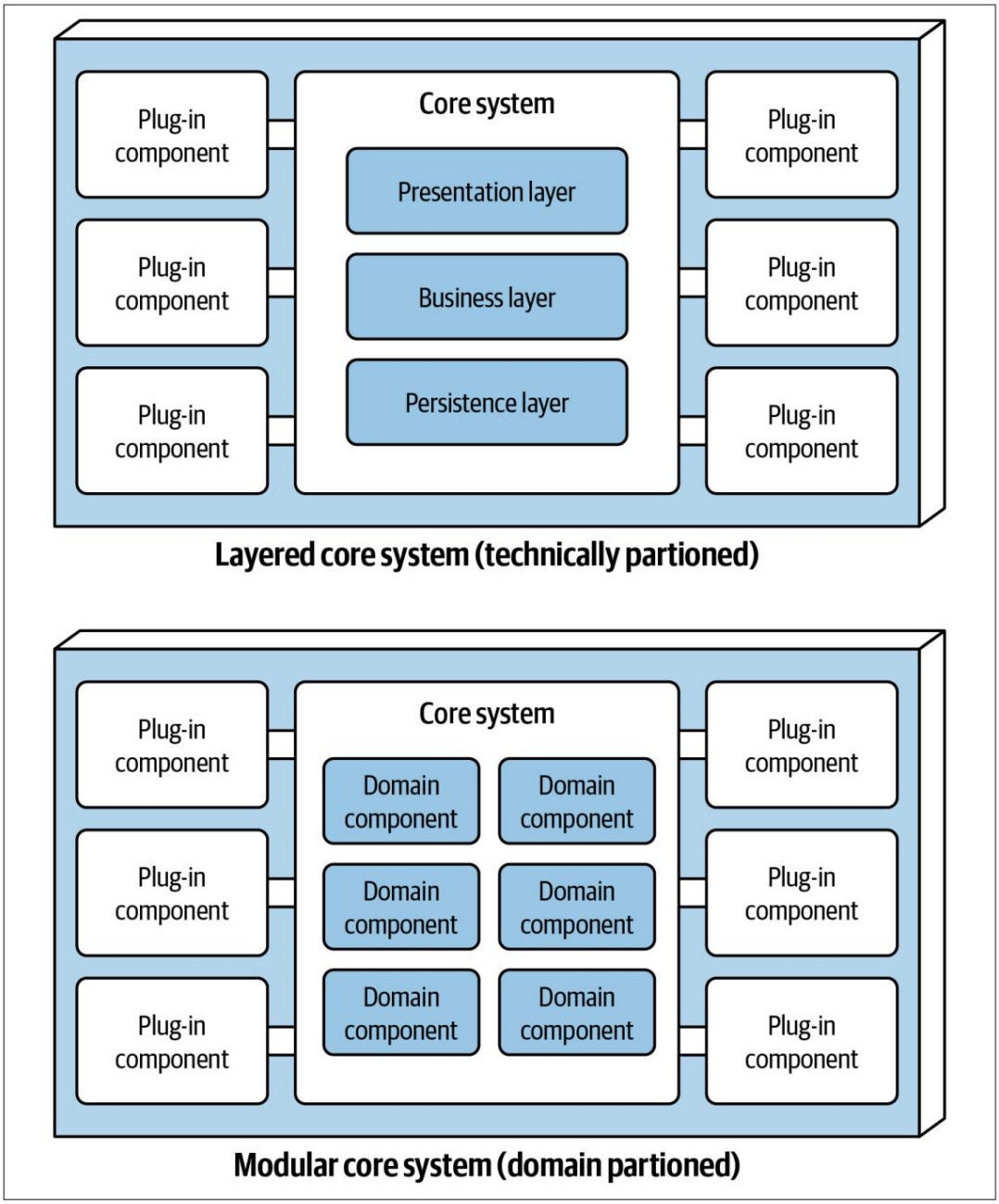


그림 13-2. 마이크로커널 아키텍처 코어 시스템의 변형

핵심 시스템의 프레젠테이션 계층은 핵심 시스템 내부에 내장되거나, 핵심 시스템이 백엔드 서비스를 제공하는 별도의 사용자 인터페이스로 구현될 수 있습니다. 실제로 마이크로 커널 아키텍처 스타일을 사용하여 별도의 UI를 구현할 수도 있습니다. 그림 13-3은 이러한 프레젠테이션 계층의 다양한 형태를 핵심 시스템과 관련하여 보여줍니다.

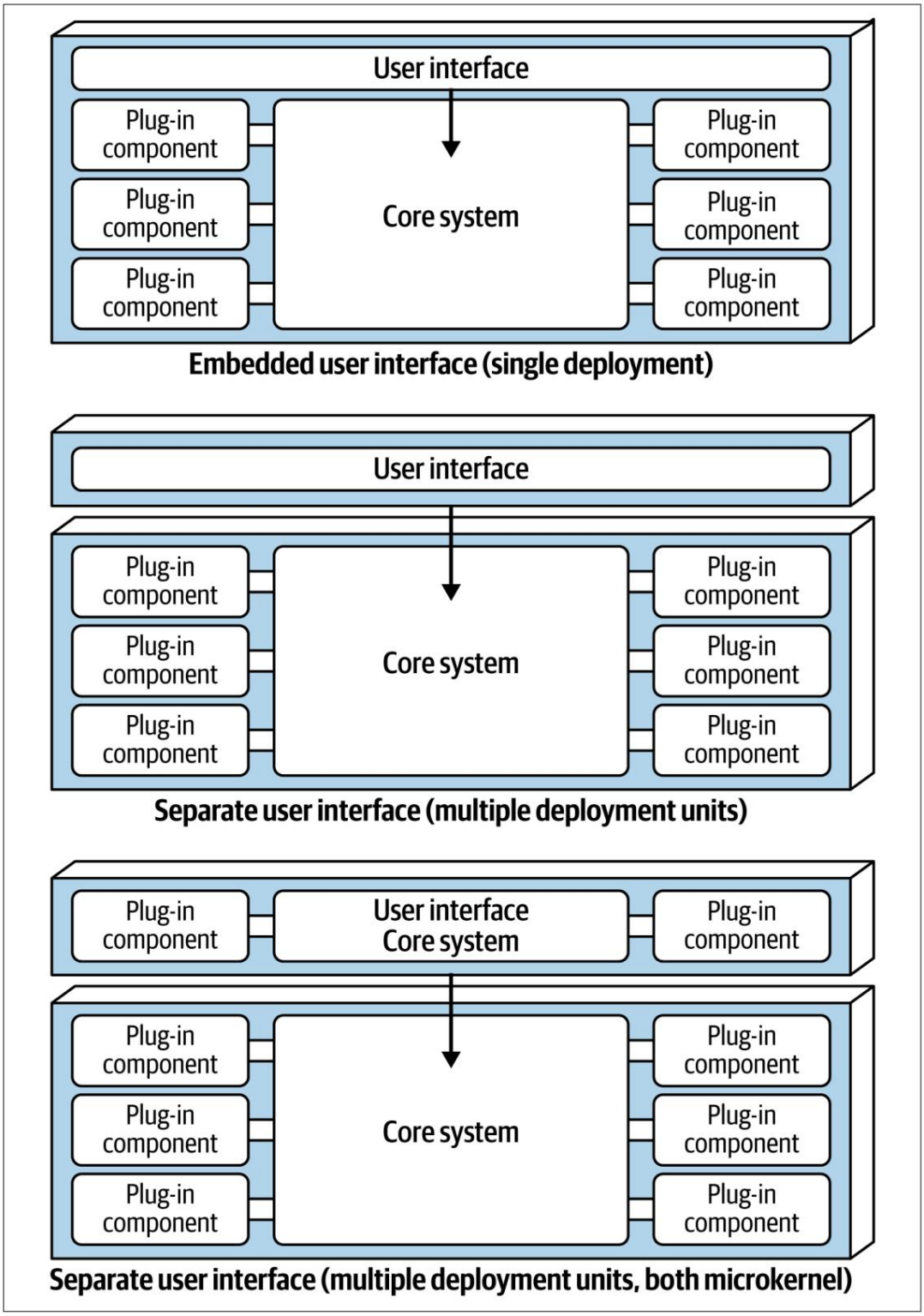


그림 13-3. 사용자 인터페이스 변형

플러그인 구성 요소는 핵심

시스템을 향상시키거나 확장하기 위해 특수 처리, 추가 기능 및 사용자 정의 코드를 포함하는 독립적인 구성 요소입니다. 또한, 변동성이 큰 코드를 분리하여 애플리케이션의 유지 관리 및 테스트 용이성을 향상시킵니다. 이상적으로 플러그인 구성 요소는 서로 종속성이 없어야 합니다.

플러그인 구성 요소와 코어 시스템 간의 통신은 일반적으로 지점 간 통신입니다. 즉, 플러그인과 코어 시스템을 연결하는 "파이프"는 일반적으로 플러그인 구성 요소의 진입점 클래스에 대한 메서드 호출 또는 함수 호출입니다. 또한 플러그인 구성 요소는 컴파일 기반 또는 런타임 기반일 수 있습니다. 런타임 플러그인 구성 요소는 코어 시스템이나 다른 플러그인을 재배포하지 않고도 런타임에 추가하거나 제거할 수 있으며, 일반적으로 OSGi(Open Service Gateway Initiative for Java), Pen!rose(Java), Jigsaw(Java) 또는 Prism(.NET)과 같은 프레임워크를 통해 관리됩니다. 컴파일 기반 플러그인 구성 요소는 관리가 훨씬 간단하지만, 수정, 제거 또는 추가하려면 전체 모놀리식 애플리케이션을 재배포해야 합니다.

지점 간 플러그인 구성 요소는 공유 라이브러리(JAR, DLL 또는 Gem 등), Java의 패키지 이름 또는 C#의 네임스페이스로 구현할 수 있습니다. Going Green 애플리케이션에서 각 전자 장치 플러그인은 JAR, DLL 또는 Ruby Gem(또는 다른 공유 라이브러리)으로 작성 및 구현할 수 있으며, 장치 이름은 그림 13-4에 표시된 것처럼 독립적인 공유 라이브러리 이름과 일치합니다.

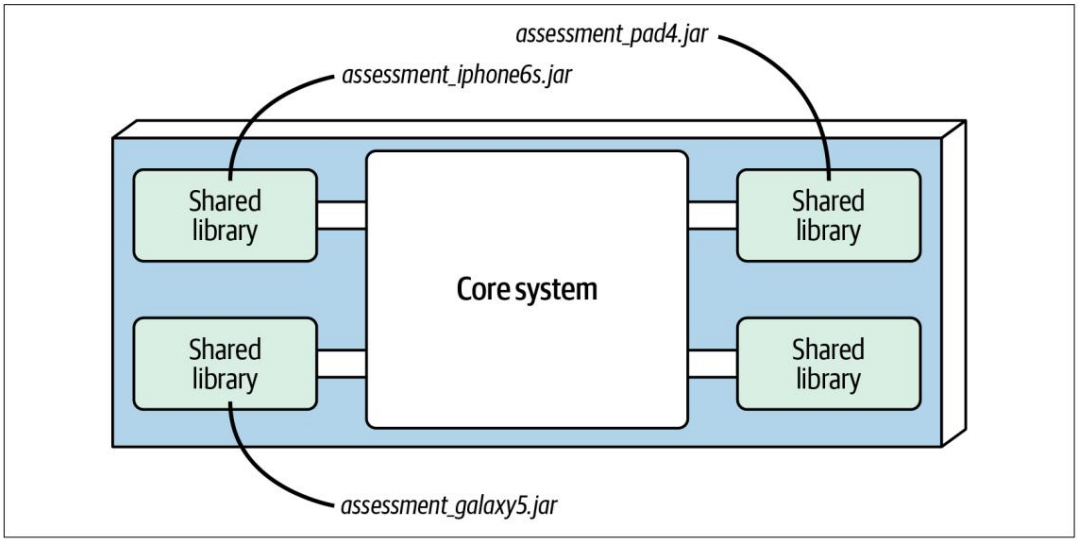


그림 13-4. 공유 라이브러리 플러그인 구현

그림 13-5 에 나와 있는 더 쉬운 접근 방식은 각 플러그인 구성 요소를 동일한 코드베이스 또는 IDE 프로젝트 내에서 별도의 네임스페이스 또는 패키지 이름으로 구현하는 것입니다.
네임스페이스를 생성할 때는 `app.plug-in.<domain>.<context>` 형식을 권장합니다. 예를 들어, `app.plug-in.assessment.iphone6s` 네임스페이스를 생각해 보세요. 두 번째 노드 (`plug-in`) 는 이 컴포넌트가 플러그인임을 명확히 하고, 따라서 기본 규칙(자체적으로 완결되어야 하며 다른 플러그인과 분리되어야 함)을 엄격히 준수해야 함을 나타냅니다. 세 번째 노드는 도메인(이 경우 ` assessment`) 을 설명하여 플러그인 컴포넌트를 공통 목적별로 구성하고 그룹화할 수 있도록 합니다. 네 번째 노드 (`iphone6s`) 는 플러그인의 특정 컨텍스트를 설명하므로 수정 또는 테스트를 위해 해당 장치 플러그인을 쉽게 찾을 수 있습니다.

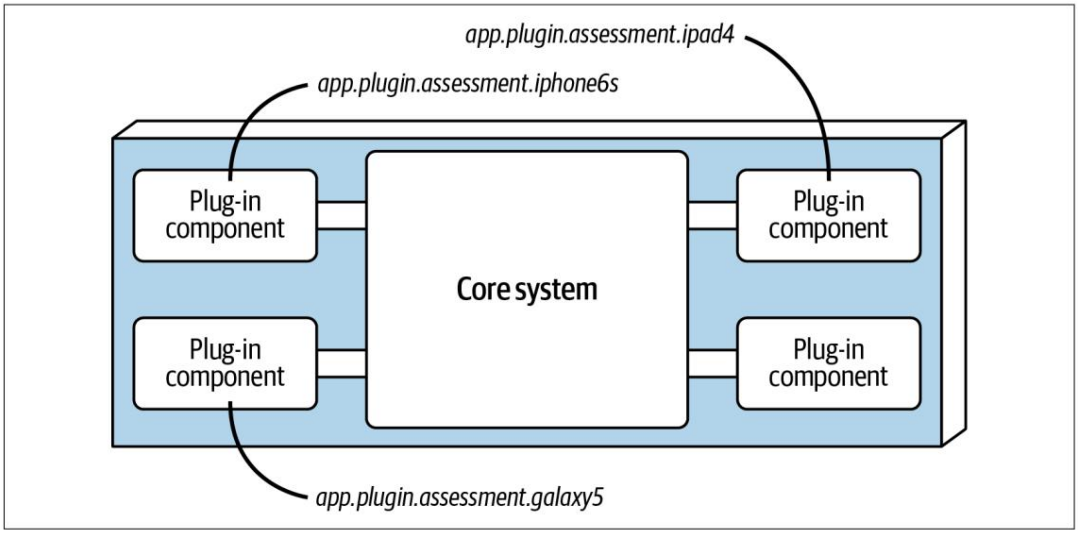


그림 13-5. 패키지 또는 네임스페이스 플러그인 구현

플러그인 구성 요소는 코어 시스템과 항상 직접적인 지점 간 통신을 사용할 필요는 없습니다. REST 또는 메시징을 사용하여 플러그인 기능을 호출하는 방식이 있으며, 각 플러그인은 독립적인 서비스(또는 컨테이너를 사용하여 구현된 마이크로 서비스)로 작동할 수 있습니다. 이러한 방식이 전반적인 확장성을 향상시키는 것처럼 보일 수 있지만, 그림 13-6에 나타낸 이 토폴로지는 단일 아키텍처 양자(quat) 에 불과하다는 점에 유의해야 합니다. 모든 요청은 플러그인 서비스에 도달하기 전에 먼저 코어 시스템을 거쳐야 합니다.

개별 서비스로 구현된 플러그인 구성 요소에 대한 원격 액세스 방식의 장점은 구성 요소 간의 결합도를 낮추고 확장성과 처리량을 향상시키며 OSGi, Jigsaw, Prism과 같은 특별한 프레임워크 없이도 런타임 변경이 가능하다는 점입니다. 또한 플러그인과의 비동기 통신이 가능하여 시나리오에 따라 전반적인 사용자 응답성을 크게 향상시킬 수 있습니다. 예를 들어, '친환경 정책(Going Green)'을 살펴보면, 전자 기기 평가가 완료될 때까지 기다리는 대신 핵심 시스템에서 특정 기기에 대한 평가를 시작하도록 비동기 요청을 보낼 수 있습니다. 평가가 완료되면 사용자는 해당 기기에 대한 평가를 시작할 수 있습니다.

완료되면 플러그인은 다른 비동기 메시징 채널을 통해 핵심 시스템에 알릴 수 있으며, 핵심 시스템은 사용자에게 평가가 완료되었음을 알립니다.

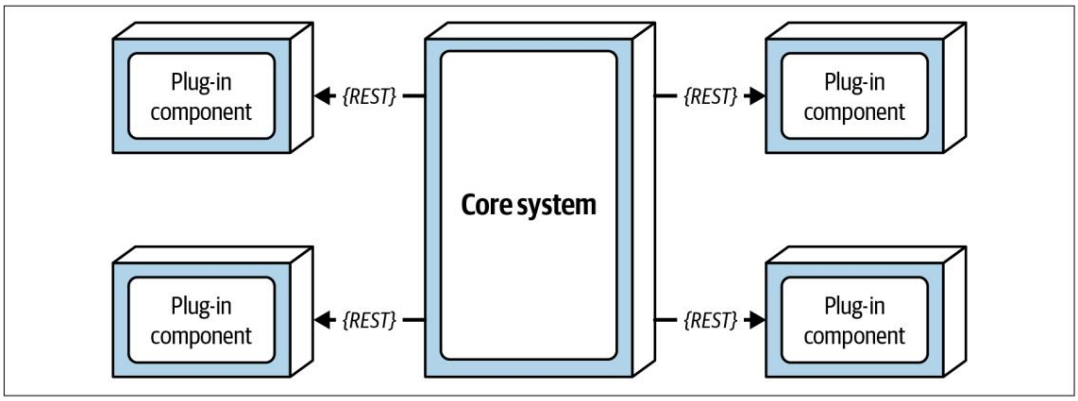


그림 13-6. REST를 이용한 원격 플러그인 액세스

이러한 이점에는 단점이 따릅니다. 원격 플러그인 액세스는 마이크로커널 아키텍처를 단일 구조가 아닌 분산 구조로 바꾸어 놓기 때문에 대부분의 타사 온프레미스 제품을 구현하고 배포하기가 어렵습니다. 또한, 전반적인 복잡성과 비용을 증가시키고 전체 배포 토폴로지를 복잡하게 만듭니다. 특히 시스템이 REST API를 사용하는 경우, 플러그인이 응답하지 않거나 실행이 중단되면 요청을 완료할 수 없습니다. 이는 단일 구조 배포에서는 발생하지 않는 문제입니다. 지점 간 통신과 원격 통신 중 어떤 방식을 선택할지는 프로젝트의 특정 요구 사항과 신중한 장단점 분석을 바탕으로 결정해야 합니다.

"마이크로커넌성"의 스펙트럼

플러그인을 지원하는 모든 시스템이 마이크로커널은 아니지만, 모든 마이크로커널은 플러그인을 지원합니다. 우리가 "마이크로커널성"이라고 부르는 시스템의 정도는 핵심 시스템에 얼마나 많은 독립적인 기능이 존재하는지에 따라 달라집니다. 이는 그림 13-7에 나타난 스펙트럼에 반영되어 있습니다.

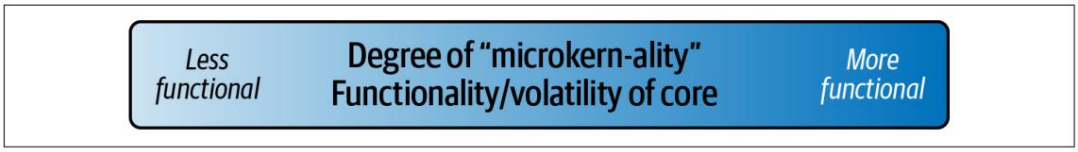


그림 13-7. "마이크로커널리티"의 스펙트럼

그림 13-7 에서 "순수" 마이크로커널 아키텍처(앞서 언급한 Eclipse IDE나 린터 도구와 같은)는 핵심 기능이 매우 제한적입니다. 예를 들어, 린터는 소스 코드를 분석하여 추상 구문 트리를 제공하므로 개발자는 언어 사용에 대한 규칙을 작성할 수 있습니다. 코어는 코드를 분석하지만, 누군가가 이를 활용하는 플러그인을 개발하기 전까지는 거의 쓸모가 없습니다. 웹 브라우저와 대조해 보면, 웹 브라우저는 다음과 같은 기능을 지원합니다.

플러그인이 필요 없지만, 플러그인 없이도 완벽하게 작동합니다. 브라우저는 스펙트럼의 오른쪽 끝에 속합니다.

코어의 변동성을 파악하는 것은 아키텍트가 플러그인만 지원하는 시스템과 보다 "순수한" 마이크로커널 시스템 중에서 선택하는 데 큰 도움이 됩니다.

레지스트리

핵심 시스템은 사용 가능한 플러그인 모듈과 해당 모듈에 접근하는 방법을 알아야 합니다. 이를 구현하는 일반적인 방법 중 하나는 플러그인 레지스트리를 이용하는 것입니다.
이 레지스트리에는 각 플러그인 모듈에 대한 정보가 포함되어 있으며, 여기에는 이름, 데이터 계약, 원격 액세스 프로토콜 세부 정보(플러그인이 코어 시스템에 연결되는 방식에 따라 다름) 등이 있습니다. 예를 들어, 세무 감사 위험도가 높은 항목을 표시하는 세무 소프트웨어용 플러그인은 서비스 이름(AuditChecker), 데이터 계약(입력 데이터 및 출력 데이터), 계약 형식(XML)을 포함하는 레지스트리 항목을 가질 수 있습니다.

레지스트리는 핵심 시스템이 소유하는 내부 맵 구조처럼 간단할 수도 있고, 키와 플러그인 구성 요소 참조를 포함할 수도 있으며, 핵심 시스템에 내장되거나 외부에 배포된 레지스트리 및 검색 도구(예: Apache ZooKeeper 또는 Consul)처럼 복잡할 수도 있습니다. 전자제품 재활용 사례를 사용하여 다음 Java 코드는 핵심 시스템 내에 간단한 레지스트리를 구현하며, iPhone 6S 장치를 평가하기 위한 지점 간 항목, 메시징 항목 및 RESTful 항목의 예를 보여줍니다.

```
Map<String, String> registry = new HashMap<String, String>(); static { // 지점 간 접근
예시
    registry.put("iPhone6s", "Iphone6sPlugin"); }


    // 메시지 전송 예시
     registry.put("iPhone6s", "iphone6s.queue");

    //RESTful 예제
     registry.put("iPhone6s", "https://atlas:443/assess/iphone6s");
}
```

계약

플러그인 구성 요소와 코어 시스템 간의 계약은 일반적으로 플러그인 구성 요소 도메인 전체에 걸쳐 표준화되어 있으며, 플러그인 구성 요소의 동작, 입력 데이터 및 반환되는 출력 데이터를 포함합니다. 사용자 정의 계약은 일반적으로 플러그인 구성 요소가 제3자에 의해 개발되어 아키텍트가 플러그인이 사용하는 계약을 제어할 수 없는 상황에서 사용됩니다. 이러한 경우, 코어 시스템이 각 플러그인에 대해 특수 코드를 작성할 필요가 없도록 플러그인 계약과 표준 계약 사이에 어댑터를 생성하는 것이 일반적입니다.

플러그인 계약은 XML, JSON 또는 플러그인과 코어 시스템 간에 주고받는 객체로 구현할 수 있습니다. 전자제품 재활용 애플리케이션에서 다음 계약(AssessmentPlugin이라는 표준 Java 인터페이스로 구현됨) 은 전체적인 동작 (assess(), register(), deregister ()) 과 플러그인 구성 요소에서 기대되는 해당 출력 데이터 (AssessmentOutput)를 정의합니다.

```
public interface AssessmentPlugin { public
        AssessmentOutput assess(); public String
        register(); public String
        deregister();
}

public class AssessmentOutput { public
        String assessmentReport; public Boolean
        resell; public Double value;
        public Double resellPrice;

}
```

이 계약 예시에서 장치 평가 플러그인은 다음과 같은 내용을 포함하는 평가 보고서를 반환해야 합니다.

- 형식화된 문자열 • 이

  장치를 제3자 시장에서 재판매하거나 안전하게 폐기할 수 있는지 여부를 나타내는 재판매 플래그 (참 또는 거짓)
- 해당 물품을 재판매할 수 있는 경우, 계산된 가치와 권장 재판매 가격

이 예시에서 핵심 시스템과 플러그인 구성 요소 간의 역할 및 책임 모델, 특히 assessmentReport 필드에 주목하십시오. 핵심 시스템은 평가 보고서의 형식을 지정하고 세부 정보를 이해하는 책임을 지지 않으며, 단지 보고서를 인쇄하거나 사용자에게 표시하는 역할만 담당합니다.

데이터 토폴로지

일반적으로 팀은 단일 (일반적으로 관계형) 데이터베이스를 사용하는 모놀리식 아키텍처로 마이크로커널 아키텍처를 구현합니다.

플러그인 구성 요소가 중앙에서 공유되는 데이터베이스에 직접 연결하는 것은 흔하지 않습니다. 대신 핵심 시스템이 이 역할을 맡아 필요한 모든 데이터를 각 플러그인에 전달합니다. 이러한 방식의 주된 이유는 분리입니다. 데이터베이스를 변경해도 핵심 시스템에만 영향을 미치고 플러그인 구성 요소에는 영향을 미치지 않아야 합니다. 하지만 플러그인은 해당 플러그인만 접근할 수 있는 별도의 데이터 저장소를 가질 수 있습니다. 예를 들어, Going Green 시스템의 각 장치 평가 플러그인은 해당 제품에 대한 모든 특정 평가 규칙을 포함하는 자체적인 간단한 데이터베이스 또는 규칙 엔진을 가질 수 있습니다. 플러그인이 소유하는 데이터 저장소는 다음과 같습니다.

구성 요소는 외부에 있을 수도 있고(그림 13-8 참조), 플러그인 구성 요소 또는 모놀리식 배포의 일부로 내장될 수도 있습니다(인메모리 또는 내장 데이터베이스의 경우처럼).

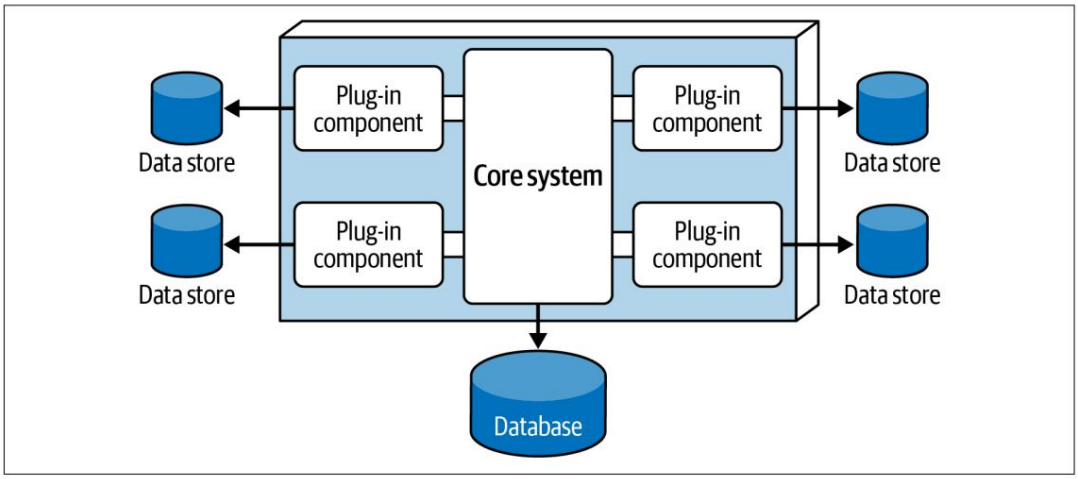


그림 13-8. 플러그인 구성 요소는 자체 데이터 저장소를 가질 수 있습니다.

클라우드 고려 사항

마이크로커널 아키텍처는 일반적으로 모놀리식 구조를 가지므로, 클라우드 환경에 적응하는 데에는 몇 가지 큰 틀의 옵션이 있습니다. 첫 번째는 클라우드 서비스나 컨테이너를 이용하여 전체 애플리케이션을 클라우드에 배포하는 것입니다. 두 번째는 데이터만 클라우드에 저장하고 마이크로커널은 온프레미스 시스템으로 구현하는 것입니다. 세 번째는 핵심 시스템은 온프레미스에 두고 플러그인만 클라우드에 배포하는 방식입니다.

모듈성 측면에서 좋은 선택처럼 보일 수 있지만, 응답성 측면에서 몇 가지 어려운 문제가 발생할 수 있습니다. 일반적으로 마이크로커널 아키텍처에서는 플러그인 호출이 빈번하게 발생하고, 팀에서 핵심 워크플로우를 플러그인을 사용하여 구현하기 때문에 각 호출마다 상당한 양의 정보가 전달됩니다. 코어와 플러그인을 분리함으로써 발생하는 지연 시간은 바람직하지 않은 오버헤드를 초래할 수 있습니다.

일반적인 위험

이러한 아키텍처와 관련된 일반적인 위험은 주로 이를 잘못 적용하는 데서 발생합니다.

변동성 핵심

마이크로커널 아키텍처에서 코어는 초기 개발 이후 최대한 안정적으로 유지되어야 합니다. 이러한 설계 방식의 장점 중 하나는 변경 사항을 플러그인에 국한시킬 수 있다는 것입니다. 끊임없이 변화하는 코어를 구축하는 것은 이러한 아키텍처의 철학을 훼손하지만, 이는 흔히 발생하는 실수입니다. 이러한 실수는 종종 아키텍트가 코어의 변동성을 잘못 판단한 결과이며, 결국 리팩토링을 통해 그 변동성을 해결해야 합니다.

플러그인 종속성 마이크로커널은 플러그

인이 서로 통신하지 않고 코어 시스템과만 통신할 때 가장 잘 작동합니다. 마이크로커널이 아닌 대부분의 플러그인 시스템은 코어 시스템 외에는 종속성이 없는 종속성 없는 플러그인을 사용합니다. 즉, 플러그인들은 서로 통신하지 않으므로 코어 시스템에서 해결해야 할 공유 종속성이 없습니다. 그러나 Eclipse IDE와 같은 복잡한 마이크로커널 아키텍처 애플리케이션에서는 구성 요소 간에 종속성이 발생할 수 있으며, 이로 인해 코어 시스템이 구성 요소 간의 전이적 종속성 충돌을 해결해야 할 수도 있습니다.

플러그인 간의 의존성은 가능하지만, 전이적 의존성 관리 측면에서 수많은 복잡성을 야기합니다. 두 플러그인이 동일한 코어 라이브러리의 서로 다른 버전에 의존하는 경우 어떻게 될까요? 코어는 이러한 의존성을 해결하고 서로 다른 플러그인 버전 간의 통신을 원활하게 해야 합니다. 전이적 의존성이 존재하는 환경에서 플러그인을 추가하는 데 어려움을 겪어본 사람이라면 누구나 버전 충돌을 해결하는 데 따르는 골칫거리를 잘 알 것입니다. 가능한 한 플러그인 간의 의존성을 피하는 것이 가장 좋습니다.

통치

마이크로커널 아키텍처에서의 거버넌스는 설계자들이 그 철학을 얼마나 잘 준수하고 있는지 점검하는 데 중점을 둡니다.

일반적인 거버넌스 점검 사항은 다음과 같습니다.

- 코어에 대한 변동성 검사 - 이는 검사에 연결된 적합성 함수입니다.
  특정 코드 검사보다는 버전 관리의 변화를 살펴보세요.
- 핵심 기능의 변경 속도 • 계약

테스트 (특히 일부 플러그인이 다른 플러그인과 다른 버전을 지원하는 경우)
(점진적 진화로 인해) • 토폴로지

에 대한 기타 구조적 검증

팀 토폴로지 고려 사항

이 아키텍처에서 팀의 명확한 구분은 핵심 부분과 플러그인 부분으로 나뉘며, 이는 토폴로지를 반영합니다.

스트림 중심 팀은 시스

템의 핵심 기능을 구축하는 핵심 개발 영역에 속합니다. 애플리케이션 유형에 따라 플러그인 개발 또한 이 팀에서 담당할 수 있습니다.

팀 역량 강화: 마이
: 크로커널 아키텍처는 일부 동작을 플러그인으로 분리하여 A/B 테스트 및 기타 실험을 가능하게 하므로 팀 역량 강화에 매우 적합합니다.

복잡한 하위 시스템 팀
: 마이크로커널은 특수 동작을 플러그인으로 분리할 수 있기 때문에 복잡한 서브시스템 팀에 매우 적합합니다. 예를 들어 분석과 같은 특수 처리는 플러그인으로 분리할 수 있으므로 스트림 지원 팀은 핵심 동작에 집중하고 특수 동작이 필요할 때 정교한 플러그인을 호출할 수 있습니다.

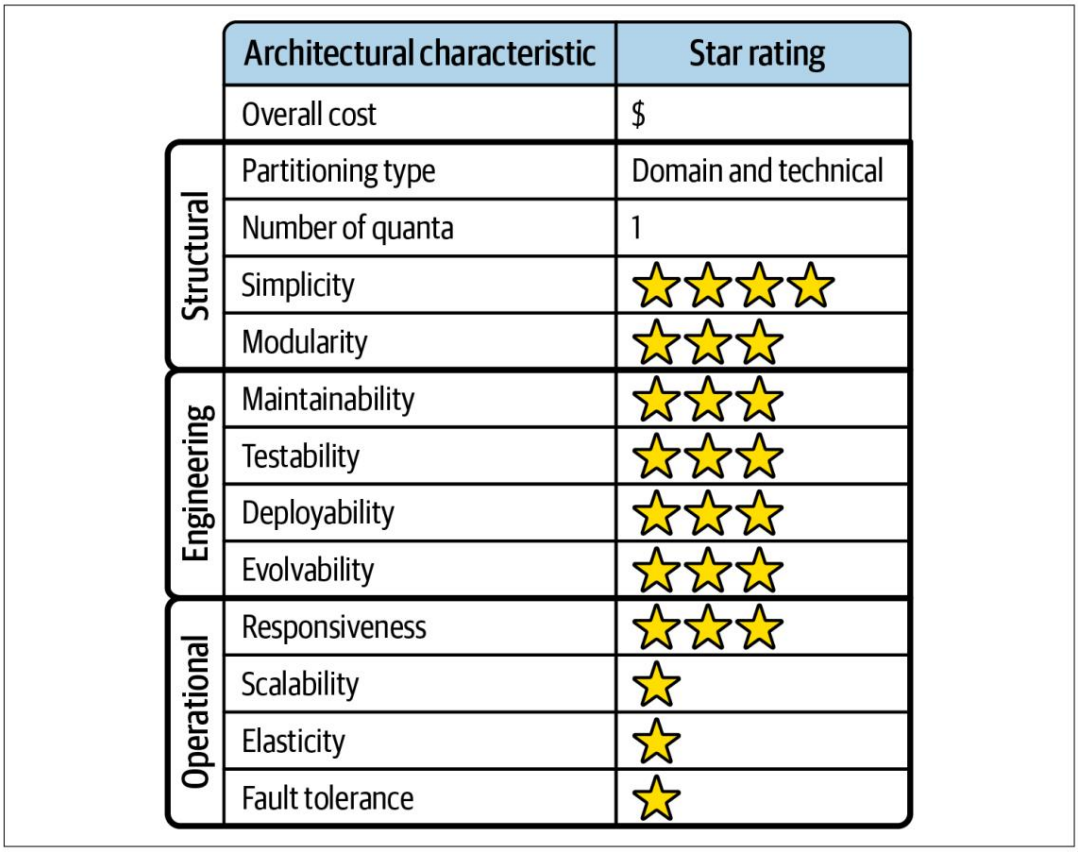
플랫폼 팀은 다른
: 모놀리식 아키텍처와 마찬가지로 이 아키텍처의 운영 세부 사항에 주로 관여합니다.

건축적 특징 평가

그림 13-9 의 특성 평가에서 별 1개는 해당 아키텍처 특성이 아키텍처에서 제대로 지원되지 않음을 의미하고, 별 5개는 해당 아키텍처 특성이 아키텍처 스타일에서 가장 강력한 기능 중 하나임을 의미합니다. 평가표에 제시된 각 특성에 대한 정의는 4 장 에서 확인할 수 있습니다 .

마이크로커널 아키텍처 스타일은 계층형 아키텍처와 유사하게 단순성과 전반적인 비용이 주요 강점이며, 확장성, 내결함성 및 탄력성이 주요 약점입니다. 이러한 약점은 마이크로커널 아키텍처에서 흔히 볼 수 있는 모놀리식 배포 방식 때문입니다. 또한 계층형 아키텍처 스타일과 마찬가지로 모든 요청이 독립적인 플러그인 구성 요소에 도달하기 위해 코어 시스템을 거쳐야 하므로 아키텍처 양자는 항상 단일합니다(1). 유사점은 여기까지입니다.

마이크로커널은 도메인 분할과 기술적 분할을 모두 가능하게 하는 유일한 아키텍처 스타일이라는 점에서 독특합니다. 대부분의 마이크로커널 아키텍처는 기술적 분할을 하지만, 도메인 분할은 주로 도메인과 아키텍처 간의 강력한 동형성을 통해 이루어집니다. 예를 들어, 각 위치 또는 클라이언트에 따라 서로 다른 구성이 필요한 문제는 이 아키텍처 스타일에 매우 적합합니다. 또한 사용자 정의 및 기능 확장성을 강조하는 제품이나 애플리케이션(예: Jira 또는 Eclipse와 같은 IDE)에도 적합합니다.



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

그림 13-9. 마이크로커널 아키텍처 특성 평가

테스트 용이성, 배포 용이성 및 안정성은 평균보다 약간 높은 수준(별 3개)입니다. 이는 주로 기능들을 독립적인 플러그인 구성 요소로 분리할 수 있기 때문입니다. 제대로 구현한다면, 이는 변경 사항에 대한 전반적인 테스트 범위를 줄이고 배포 위험을 낮추는 데 도움이 됩니다. 특히 플러그인 구성 요소 배포가 런타임 기반인 경우 더욱 효과적입니다.

모듈성과 진화 가능성 또한 평균보다 약간 높은 수준(별 3개)입니다. 마이크로커널 아키텍처 스타일 덕분에 추가 기능은 독립적이고 자체 포함된 플러그인 구성 요소를 통해 추가, 제거 및 변경할 수 있으므로 애플리케이션을 확장하고 개선하는 것이 비교적 쉽고 팀이 변화에 훨씬 빠르게 대응할 수 있습니다. 이전 섹션의 세금 신고 소프트웨어 예시를 생각해 보세요. 미국 세법이 변경되어(실제로 항상 변경됩니다) 새로운 세금 양식이 필요한 경우, 해당 새 세금 양식을 플러그인 구성 요소로 만들어 애플리케이션에 손쉽게 추가할 수 있습니다. 마찬가지로, 세금 양식이나 워크시트가 더 이상 필요하지 않은 경우 해당 플러그인을 애플리케이션에서 간단히 제거할 수 있습니다.

마이크로커널 아키텍처 스타일에서 응답성은 항상 평가하기 흥미로운 요소입니다. 저희는 응답성에 별 3개(평균보다 약간 높음)를 주었는데, 이는 마이크로커널 애플리케이션이 일반적으로 규모가 작고 대부분의 계층형 아키텍처처럼 크게 확장되지 않기 때문입니다. 또한, 아키텍처 싱크홀(Architecture Sinkhole) 현상의 영향을 덜 받습니다.

10장 에서 논의했던 안티패턴입니다. 마지막으로, 마이크로커널 아키텍처는 불필요한 기능을 제거함으로써 간소화할 수 있으며, 이를 통해 애플리케이션 실행 속도를 향상시킬 수 있습니다.
좋은 예로 WildFly (이전의 JBoss Application Server)를 들 수 있습니다. 클러스터링, 캐싱, 메시징과 같은 불필요한 기능을 제거하면 이러한 기능이 있을 때보다 애플리케이션 서버 성능이 훨씬 빨라집니다.

예시 및 사용 사례

대부분의 소프트웨어 개발 및 배포 도구는 마이크로커널 아키텍처를 사용하여 구현됩니다. 예를 들어 Eclipse IDE, PMD, Jira, Jenkins 등 이 있습니다 . Chrome과 Firefox 같은 인터넷 웹 브라우저도 일반적으로 마이크로커널 아키텍처를 사용하며, 뷰어 및 기타 플러그인은 기본 브라우저(핵심 시스템)에는 없는 추가 기능을 제공합니다. 제품 기반 소프트웨어의 예는 무수히 많지만, 대규모 비즈니스 애플리케이션은 어떨까요? 마이크로커널 아키텍처는 이러한 경우에도 적용됩니다.

이 점을 설명하기 위해 앞서 언급한 세금 신고 소프트웨어의 예를 다시 살펴보겠습니다. 미국 국세청(IRS)은 개인의 세금 납부액을 계산하는 데 필요한 모든 정보를 요약한 2페이지짜리 기본 세금 양식인 1040 양식을 사용합니다. 1040 양식의 각 항목에는 총소득과 같은 숫자가 하나씩 있으며, 이러한 숫자를 계산하려면 여러 다른 양식과 계산표가 필요합니다. 각 추가 양식과 계산표는 플러그인 구성 요소로 구현할 수 있으며, 1040 요약 세금 양식이 핵심 시스템(드라이버) 역할을 합니다.

이렇게 하면 세법 변경 사항을 독립적인 플러그인 구성 요소로 분리할 수 있으므로 변경이 더 쉽고 위험 부담이 줄어듭니다.

마이크로커널 아키텍처를 활용할 수 있는 대규모의 복잡한 비즈니스 애플리케이션의 또 다른 예는 보험금 청구 처리입니다. 보험금 청구 처리는 매우 복잡한 과정입니다. 각 관할 구역마다 보험금 청구에서 허용되는 사항과 허용되지 않는 사항에 대한 규칙과 규정이 다릅니다. 예를 들어, 일부 관할 구역(예: 미국 주)에서는 돌멩이에 맞아 앞유리가 손상된 경우 보험사가 무료로 앞유리를 교체해 줄 수 있도록 허용하지만, 다른 관할 구역에서는 허용하지 않습니다. 이로 인해 표준 보험금 청구 처리 과정에서 거의 무한대에 가까운 다양한 상황이 발생할 수 있습니다.

대부분의 보험금 청구 애플리케이션은 이러한 복잡성을 처리하기 위해 크고 복잡한 규칙 엔진을 활용합니다. 규칙 엔진은 개발자(또는 최종 사용자)가 시각적 도구나 도메인 특화 언어를 사용하여 워크플로를 선언적으로 정의하는 일련의 규칙이나 단계를 정의할 수 있도록 하는 프레임워크 또는 라이브러리입니다. 그러나 이러한 규칙 엔진은 '빅 볼 오브 머드(Big Ball of Mud)'라는 안티패턴으로 발전할 수 있는데, 간단한 규칙 변경에도 수많은 분석가, 개발자, 테스터가 투입되어 문제가 발생하지 않도록 해야 하는 상황이 발생할 수 있습니다. 마이크로커널 아키텍처 패턴을 사용하면 이러한 문제들을 상당 부분 해결할 수 있습니다.

각 관할 구역에 대한 청구 규칙은 소스 코드로 구현되거나 특정 규칙 엔진 인스턴스로 구현된 별도의 독립형 플러그인 구성 요소에 포함될 수 있습니다.

플러그인 구성 요소를 통해 접근할 수 있습니다. 이러한 방식으로 시스템의 다른 부분에 영향을 주지 않고 특정 관할 구역에 대한 규칙을 추가, 제거 또는 변경할 수 있습니다.
또한, 새로운 관할 구역을 추가하거나 제거하더라도 시스템의 다른 부분에는 영향을 미치지 않습니다. 이 예에서 핵심 시스템은 청구 접수 및 처리 표준 절차이며, 이는 자주 변경되지 않습니다.

마이크로커널 아키텍처 스타일은 매우 흔합니다. 한 번 접하고 나면 어디에서나 눈에 띄기 시작합니다. 이는 코어와 플러그인으로 구성된 아키텍처 구조가 사용자 정의라는 일반적인 문제에 잘 들어맞는 사례이며, 소프트웨어 개발에서 사용자 정의는 매우 자주 발생합니다.