Kapitel 13. Stil der Mikrokernel-Architektur

Die *Mikrokernel-Architektur* (auch als *Plug-in-Architektur* bezeichnet) wurde vor mehreren Jahrzehnten entwickelt und ist auch heute noch weit verbreitet. Diese Architektur eignet sich besonders für *produktbasierte Anwendungen*, d. h. für Anwendungen, die in einem einzigen, monolithischen Paket zum Herunterladen und Installieren bereitgestellt werden und in der Regel als Drittanbieterprodukt beim Kunden installiert werden. Es wird jedoch auch häufig für nicht produktbezogene Geschäftsanwendungen eingesetzt, insbesondere für Problembereiche, die eine Anpassung erfordern. Ein Versicherungsunternehmen in den USA, das für jeden Bundesstaat eigene Regeln hat, oder ein internationales Versandunternehmen, das verschiedene rechtliche und logistische Variationen einhalten muss, würden von diesem Stil profitieren.

Topologie

Der Mikrokernel-Stil ist eine relativ einfache monolithische Architektur, die aus zwei Komponenten besteht: einem Kernsystem und Plug-ins. Die Anwendungslogik ist zwischen unabhängigen Plug-in-Komponenten und

dem grundlegenden Kernsystem aufgeteilt, das die Anwendungsfunktionen isoliert und Erweiterbarkeit, Anpassungsfähigkeit und eigene Verarbeitungslogik bietet. Abbildung 13-1 veranschaulicht die grundlegende Topologie der Mikrokernel-Architektur.

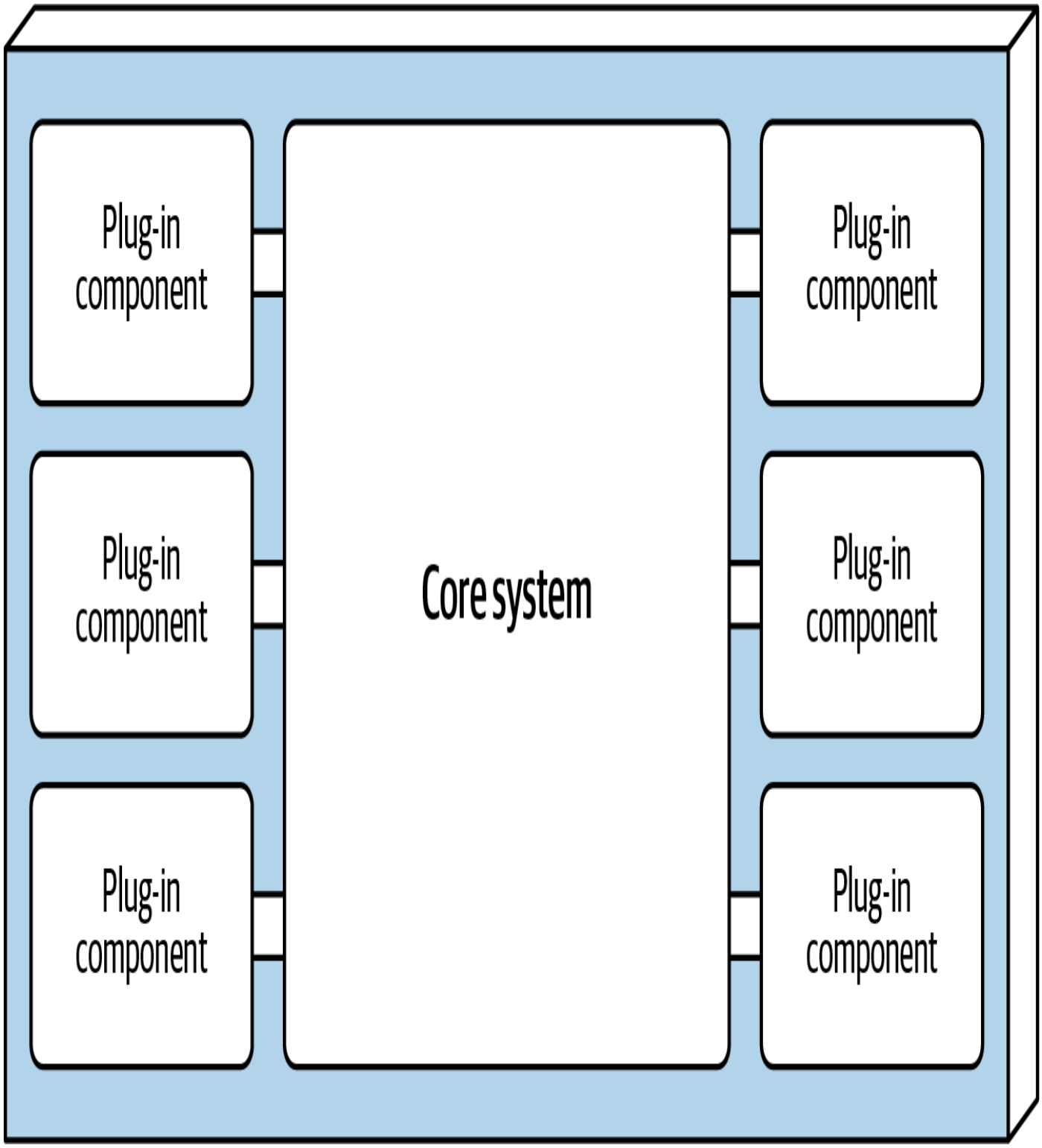


Abbildung 13-1. Grundlegende Komponenten des Mikrokernel-Architekturstils

Stil Besonderheiten

Die Mikrokernel-Architektur besteht im Wesentlichen aus zwei Arten von Komponenten: dem *Kernsystem* und den *Plug-ins*.

Kernsystem

Das *Kernsystem* ist formal definiert als die minimale Funktionalität, die zum Betrieb des Systems erforderlich ist. Ein gutes Beispiel dafür ist die Eclipse IDE. Das Kernsystem von Eclipse ist nur ein einfacher Texteditor: Öffne eine Datei, ändere den Text und speichere die Datei. Erst wenn du Plug-ins hinzufügst, wird Eclipse zu einem brauchbaren Produkt.

Eine andere Definition des Kernsystems ist jedoch der *"Happy Path"*: ein allgemeiner Verarbeitungsfluss durch die Anwendung, der wenig oder keine benutzerdefinierte Verarbeitung beinhaltet. Bei einer Mikrokernel-Architektur wird die zyklische Komplexität der Anwendung aus dem Kernsystem herausgelöst und in separaten Plug-in-Komponenten untergebracht. Dies ermöglicht eine bessere Erweiterbarkeit und Wartbarkeit sowie eine bessere Testbarkeit.

Um tiefer einzutauchen, kehren wir zu Going Green zurück, der Recycling-Anwendung für elektronische Geräte, die wir in Kapitel 7 vorgestellt haben. Nehmen wir an, du arbeitest an der Anwendung von Going Green, die jedes elektronische Gerät, das sie erhält, nach bestimmten, benutzerdefinierten Bewertungsregeln bewerten muss. Der Java-Code für diese Art der Verarbeitung könnte wie folgt aussehen:

```
public void assessDevice(String deviceID) {
   if (deviceID.equals("iPhone6s")) {
```

```
        assessiPhone6s();
    } else if (deviceID.equals("iPad1"))
        assessiPad1();
    } else if (deviceID.equals("Galaxy5"))
        assessGalaxy5();
    } else ...
        ...
    }
}
```

Anstatt all diese kundenspezifischen Anpassungen - die eine hohe zyklomatische Komplexität aufweisen - in das Kernsystem zu integrieren, könntest du für jedes zu bewertende elektronische Gerät eine eigene Plug-in-Komponente erstellen. Durch spezifische Client-Plug-in-Komponenten wird nicht nur die unabhängige Gerätelogik vom restlichen Verarbeitungsprozess isoliert, sondern sie ermöglichen auch eine Erweiterung: Wenn du ein neues Gerät zur Bewertung hinzufügst, musst du nur eine neue Plug-in-Komponente hinzufügen und die Registrierung aktualisieren. Bei der Mikrokernel-Architektur muss das Kernsystem für die Bewertung eines elektronischen Geräts nur die entsprechenden Geräte-Plug-ins finden und aufrufen, wie in diesem überarbeiteten Quellcode dargestellt:

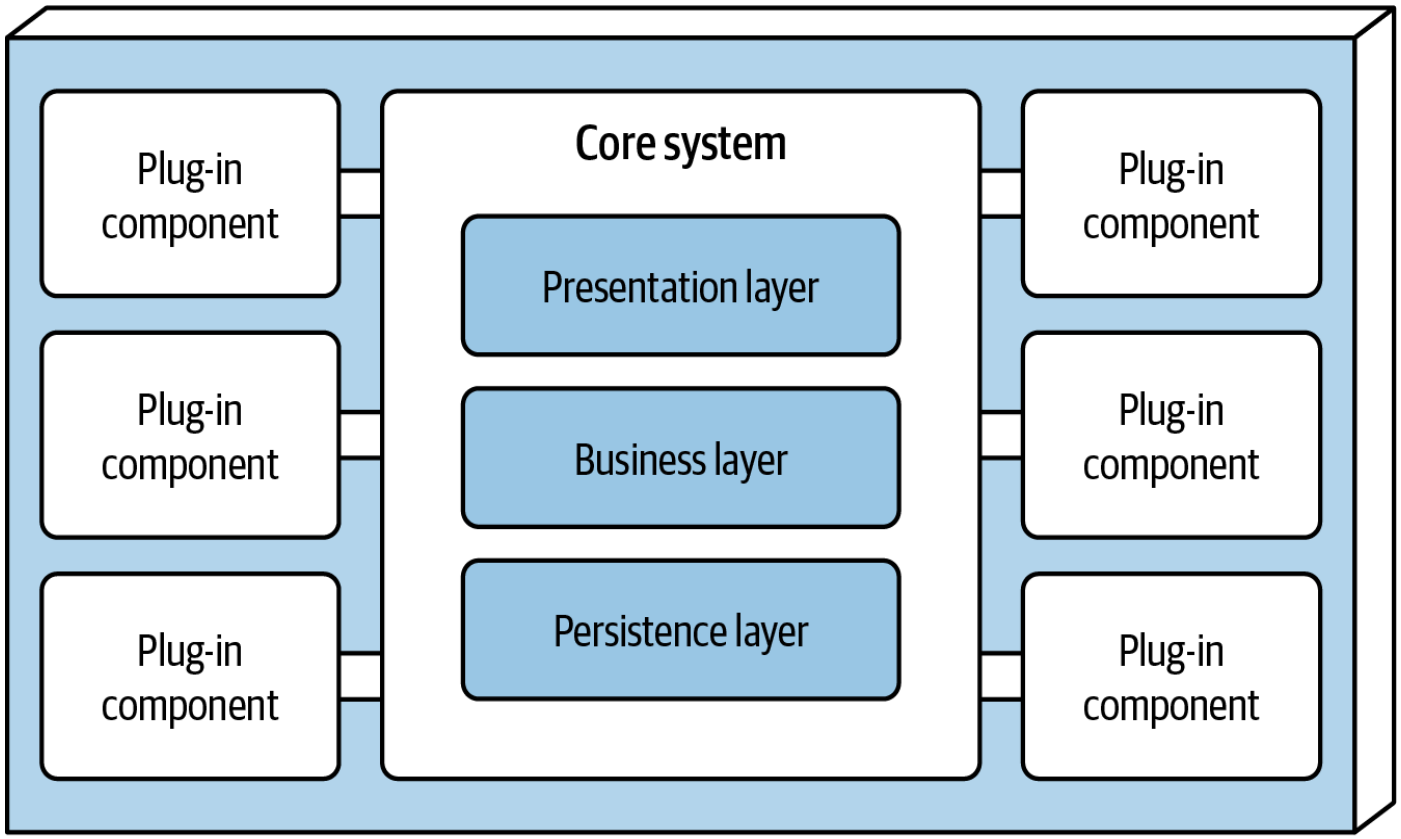
```
public void assessDevice(String deviceID) {
        String plugin = pluginRegistry.get(deviceID);
        Class<?> theClass = Class.forName(plugin);
        Constructor<?> constructor = theClass.getConstructor(
        DevicePlugin devicePlugin =
                (DevicePlugin)constructor.newInstance();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

```
            devicePlugin.assess();
    }
```

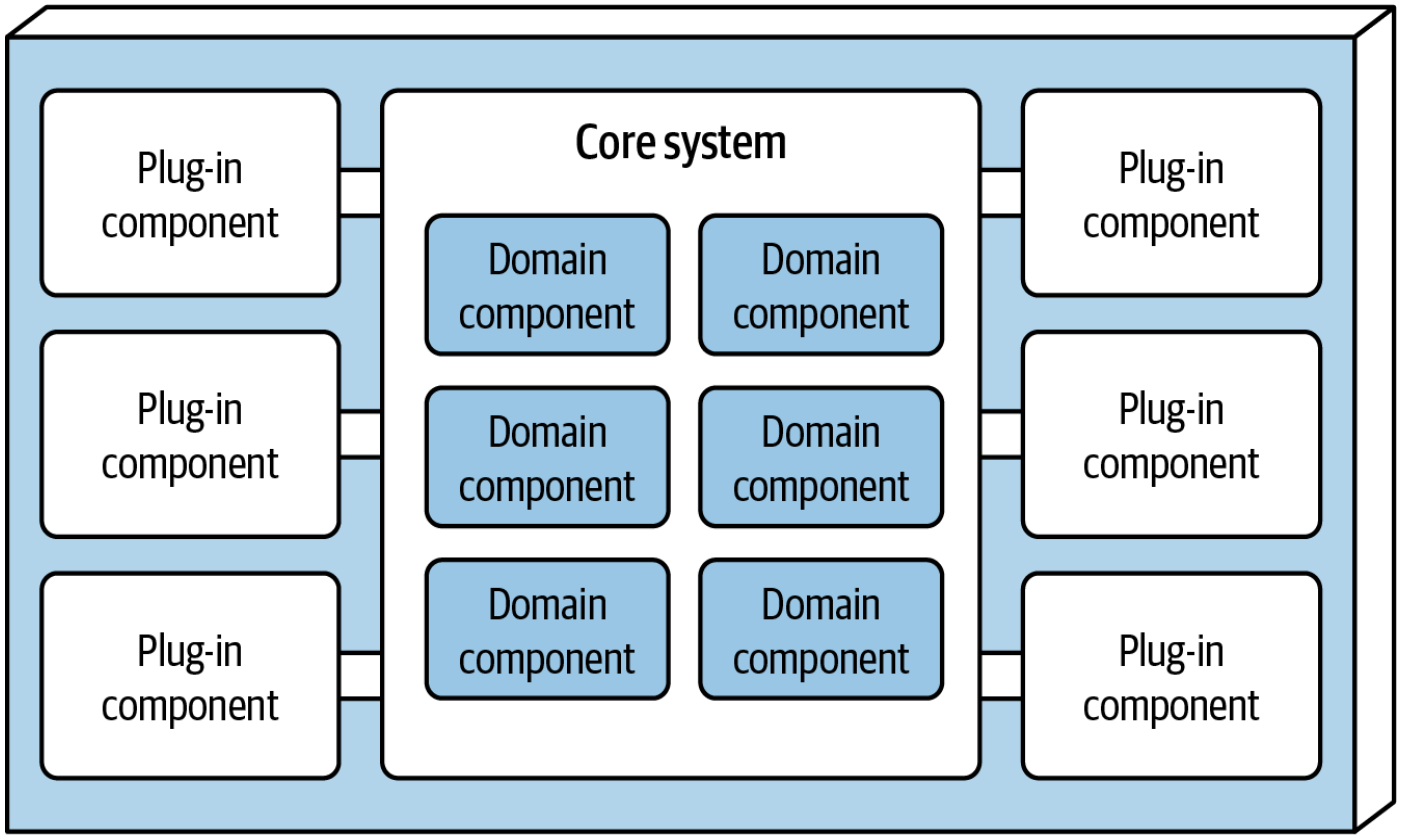
In diesem Beispiel sind alle komplexen Regeln und Anweisungen für die Bewertung eines bestimmten elektronischen Geräts in einer eigenständigen, unabhängigen Plug-in-Komponente enthalten, die generisch vom Kernsystem ausgeführt werden kann.

Je nach Größe und Komplexität kannst du das Kernsystem in einer Schichtenarchitektur oder als modularen Monolithen implementieren (wie in Abbildung 13-2 dargestellt). In manchen Fällen könntest du das Kernsystem sogar in separat bereitgestellte Domänenservices aufteilen, wobei jeder Domänenservice spezifische Plug-in-Komponenten für die jeweilige Domäne enthält. In diesem Beispiel gehen wir davon aus, dass du das System für Going Green in einer Schichtenarchitektur implementierst.

Nehmen wir an, `Payment Processing` ist der Domain-Service, der das Kernsystem darstellt. Für jede Zahlungsmethode (Kreditkarte, PayPal, Kundenkredit, Geschenkkarte und Bestellung) gäbe es eine eigene Plug-in-Komponente, die speziell für diese Zahlungsdomäne gilt.



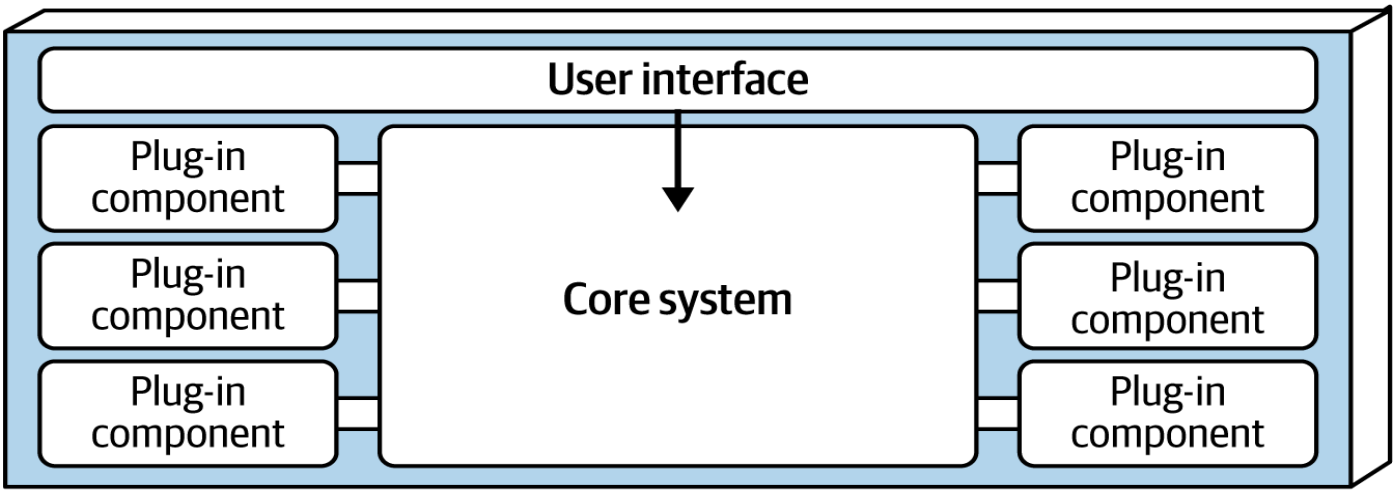
Layered core system (technically partioned)



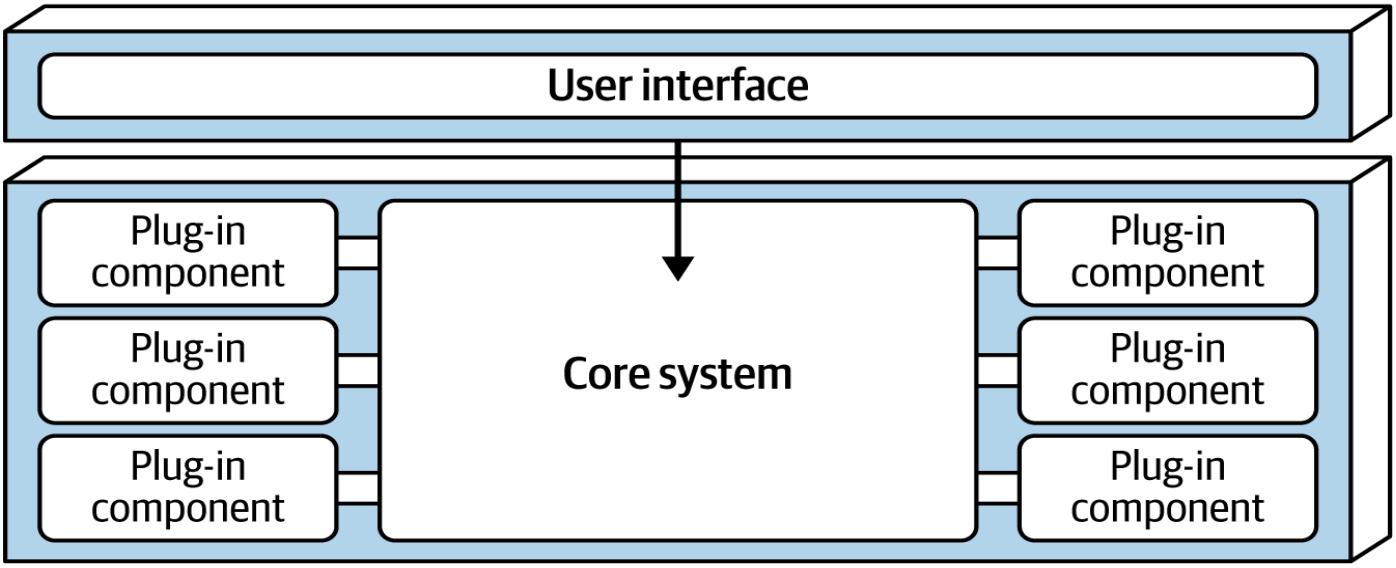
Modular core system (domain partioned)

Abbildung 13-2. Variationen des Kernsystems der Mikrokernel-Architektur

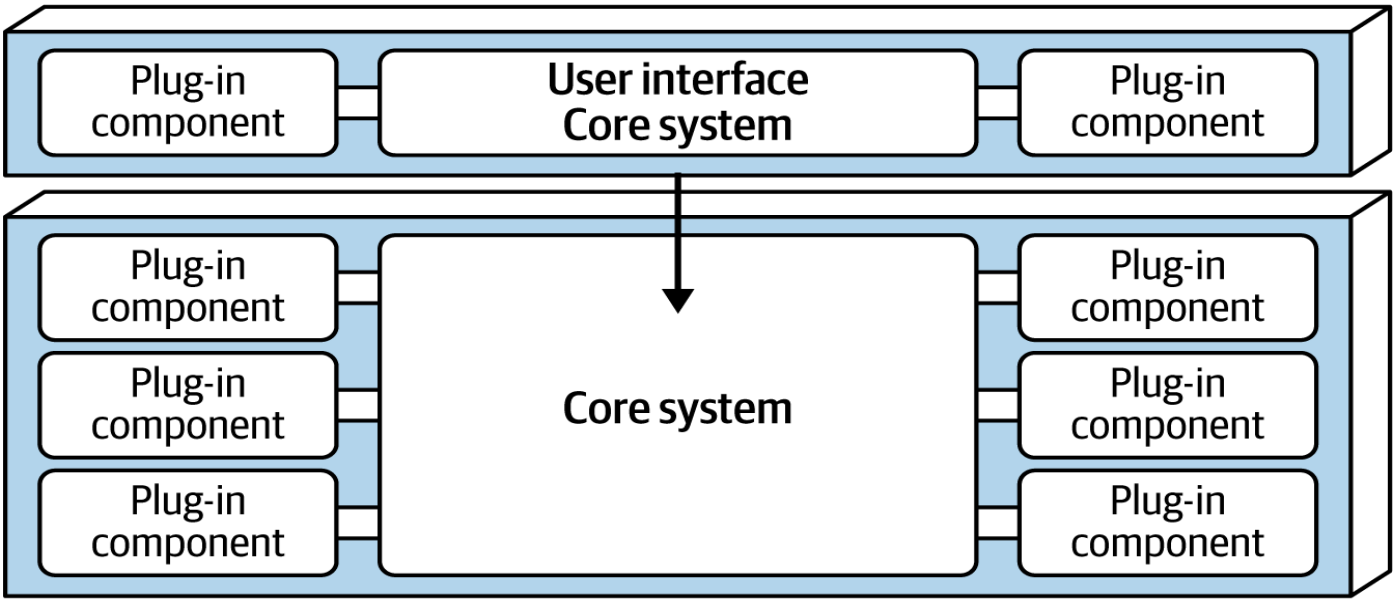
Die Präsentationsschicht des Kernsystems kann in das Kernsystem eingebettet oder als separate Benutzeroberfläche implementiert werden, wobei das Kernsystem Backend-Dienste bereitstellt. Du könntest natürlich auch eine separate Benutzeroberfläche im Stil einer Mikrokernel-Architektur implementieren. Abbildung 13-3 veranschaulicht diese Varianten der Präsentationsschicht im Verhältnis zum Kernsystem.



Embedded user interface (single deployment)



Separate user interface (multiple deployment units)



Separate user interface (multiple deployment units, both microkernel)

Abbildung 13-3. Varianten der Benutzeroberfläche

Plug-In Komponenten

Plug-in-Komponenten sind eigenständige, unabhängige Komponenten, die spezielle Verarbeitungsprozesse, zusätzliche Funktionen und benutzerdefinierten Code enthalten, um das Kernsystem zu verbessern oder zu erweitern. Außerdem isolieren sie hochvolatilen Code und sorgen so für eine bessere Wartbarkeit und Testbarkeit innerhalb der Anwendung. Idealerweise sollten Plug-in-Komponenten keine Abhängigkeiten untereinander haben.

Die Kommunikation zwischen den Plug-in-Komponenten und dem Kernsystem erfolgt in der Regel *Punkt-zu-Punkt*, d.h. die "Pipe", die das Plug-in mit dem Kernsystem verbindet, ist in der Regel ein Methoden- oder Funktionsaufruf an die Einstiegsklasse der Plug-in-Komponente. Darüber hinaus kann die Plug-in-Komponente entweder kompilierungs- oder laufzeitbasiert sein. *Laufzeit-Plug-in-Komponenten* können zur Laufzeit hinzugefügt oder entfernt werden, ohne dass das Kernsystem oder andere Plug-ins neu bereitgestellt werden müssen. Sie werden in der Regel über Frameworks wie Open Service Gateway Initiative (OSGi) für Java, Penrose (Java), Jigsaw (Java) oder Prism (.NET) verwaltet. *Compile-basierte* Plug-in-Komponenten sind viel einfacher zu verwalten, aber wenn du sie änderst, entfernst oder hinzufügst, musst du die gesamte monolithische Anwendung neu bereitstellen.

Punkt-zu-Punkt-Plug-in-Komponenten können als gemeinsam genutzte Bibliotheken (z. B. als JAR, DLL oder Gem), Paketnamen in Java oder Namensräume in C# implementiert werden. In unserer Going Green-Anwendung kann jedes Plug-in für ein elektronisches Gerät als JAR, DLL oder Ruby Gem (oder jede andere Shared Library) geschrieben und implementiert werden, wobei der Name des Geräts dem Namen der unabhängigen Shared Library entspricht, wie in Abbildung 13-4 dargestellt.

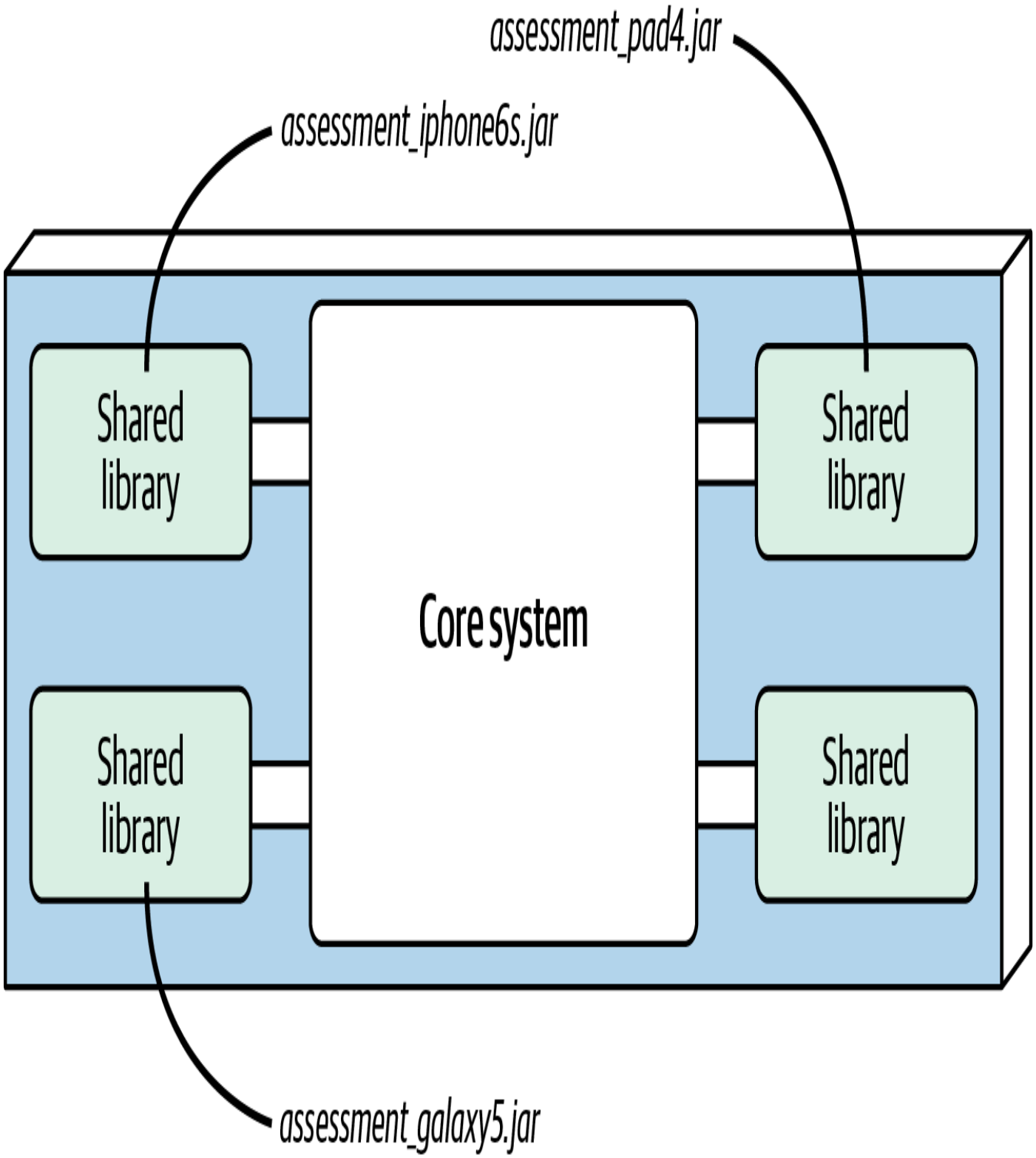


Abbildung 13-4. Implementierung eines Shared Library Plug-ins

Eine einfachere Methode, die in Abbildung 13-5 dargestellt ist, besteht darin, jede Plug-in-Komponente als eigenen Namensraum oder

Paketnamen innerhalb derselben Codebasis oder desselben IDE-Projekts zu implementieren. Bei der Erstellung des Namensraums empfehlen wir die folgende Semantik: *app.plug-in.<Domäne>.<Kontext>*. Betrachten wir zum Beispiel den Namensraum *app.plug-in.assessment.iphone6s*. Der zweite Knoten (`plug-in`) macht deutlich, dass es sich bei dieser Komponente um ein Plug-in handelt und sie sich daher streng an die Grundregeln halten sollte (sie muss in sich geschlossen und von anderen Plug-ins getrennt sein). Der dritte Knoten beschreibt den Bereich (in diesem Fall `assessment`), so dass die Plug-in-Komponenten nach einem gemeinsamen Zweck organisiert und gruppiert werden können. Der vierte Knoten (`iphone6s`) beschreibt den spezifischen Kontext für das Plug-in, so dass es einfach ist, das Geräte-Plug-in für Änderungen oder Tests zu finden.

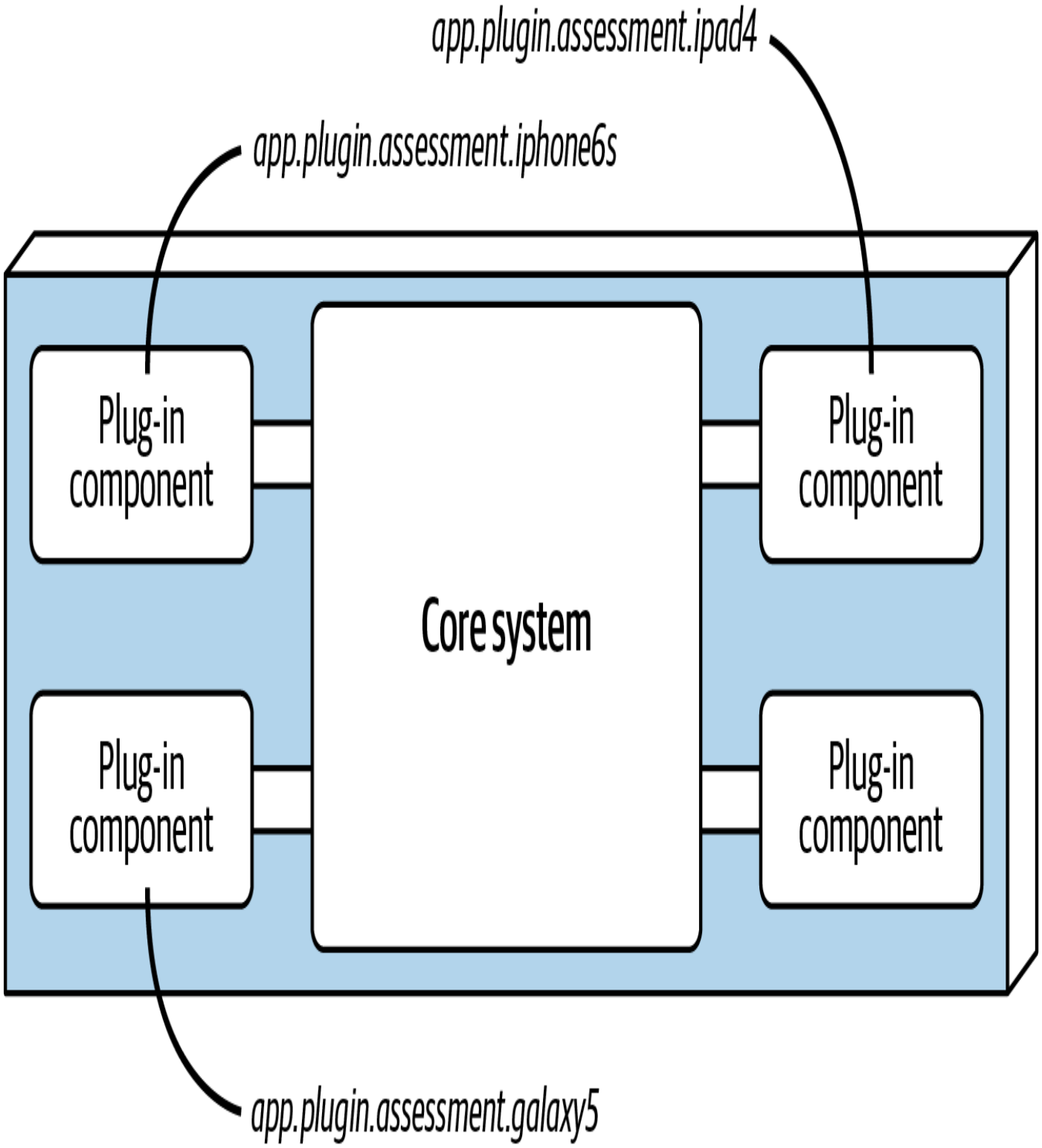


Abbildung 13-5. Implementierung eines Paket- oder Namensraum-Plug-ins

Plug-in-Komponenten müssen nicht immer eine Punkt-zu-Punkt-Kommunikation mit dem Kernsystem nutzen. Alternativen sind die Verwendung von REST oder Messaging zum Aufrufen von Plug-in-

Funktionen, wobei jedes Plug-in ein eigenständiger Dienst ist (oder vielleicht sogar ein Microservice, der in einem Container implementiert ist). Auch wenn dies nach einer guten Möglichkeit klingt, die allgemeine Skalierbarkeit zu erhöhen, ist zu beachten, dass diese Topologie (siehe Abbildung 13-6) aufgrund des monolithischen Kernsystems immer noch nur ein einziges Architekturquantum darstellt. Jede Anfrage muss zunächst das Kernsystem durchlaufen, um zum Plug-in-Dienst zu gelangen.

Die Vorteile des Remote-Access-Ansatzes für Plug-in-Komponenten, die als einzelne Dienste implementiert sind, liegen in der besseren Entkopplung der Komponenten insgesamt, der besseren Skalierbarkeit und dem höheren Durchsatz sowie in Laufzeitänderungen ohne spezielle Frameworks (wie OSGi, Jigsaw oder Prism). Außerdem ermöglicht es die asynchrone Kommunikation mit Plug-ins, was je nach Szenario die Reaktionsfähigkeit der Benutzer erheblich verbessern kann. Im Beispiel von Going Green könnte das Kernsystem eine asynchrone *Anfrage* stellen, um eine Bewertung für ein bestimmtes Gerät zu starten, anstatt auf die Bewertung der elektronischen Geräte zu warten. Wenn die Prüfung abgeschlossen ist, kann das Plug-in das Kernsystem über einen anderen asynchronen Nachrichtenkanal benachrichtigen, das wiederum den Nutzer über den Abschluss der Prüfung informiert.

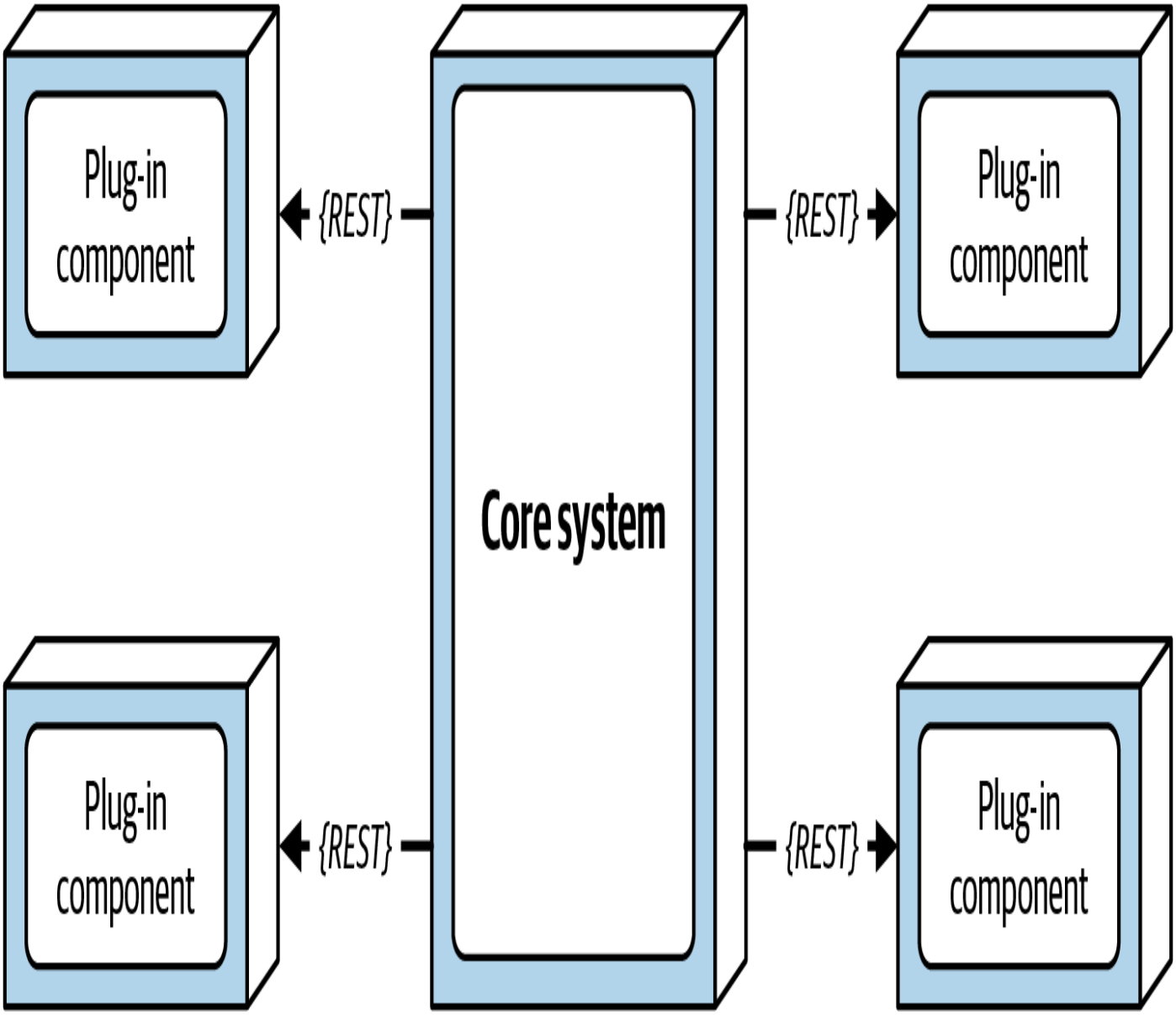


Abbildung 13-6. Fernzugriff auf das Plug-in über REST

Diese Vorteile bringen Kompromisse mit sich. Durch den Remote-Plug-in-Zugriff wird die Mikrokernel-Architektur zu einer verteilten statt zu einer monolithischen Architektur, was die Implementierung und Bereitstellung für die meisten On-Prem-Produkte von Drittanbietern erschwert. Außerdem führt dies zu mehr Komplexität und Kosten und verkompliziert die gesamte Bereitstellungstopologie. Wenn ein Plug-in nicht mehr reagiert oder nicht mehr läuft, insbesondere wenn das System REST verwendet, kann die Anfrage nicht abgeschlossen werden.

Dies wäre bei einer monolithischen Bereitstellung nicht der Fall. Die Entscheidung zwischen Punkt-zu-Punkt- und Remote-Kommunikation sollte auf der Grundlage der spezifischen Anforderungen des Projekts und einer sorgfältigen Analyse des Kompromisses getroffen werden.

Das Spektrum der "Mikrokern-alität"

Nicht alle Systeme, die Plug-ins unterstützen, sind Mikrokernel, aber alle Mikrokernel unterstützen Plug-ins. Der Grad der "Mikrokernalität", wie wir es nennen, hängt davon ab, wie viel eigenständige Funktionalität im Kernsystem vorhanden ist. Dies spiegelt sich in dem in Abbildung 13-7 dargestellten Spektrum wider.

Abbildung 13-7. Das Spektrum der "Mikrokern-alität"

In Abbildung 13-7 haben "reine" Mikrokernel-Architekturen (wie die oben erwähnte Eclipse IDE oder Linter-Tools) nur sehr wenige Kernfunktionen. Ein Linter analysiert zum Beispiel den Quellcode und liefert den abstrakten Syntaxbaum, damit ein Entwickler Regeln für die Sprachverwendung schreiben kann. Der Kern analysiert den Code, aber solange niemand ein Plug-in schreibt, um ihn zu nutzen, ist er von geringem Nutzen. Im Gegensatz dazu unterstützt ein Webbrowser zwar Plug-ins, ist aber auch ohne sie vollkommen funktionsfähig.

Die Bestimmung der Volatilität des Kerns hilft Architekten bei der Entscheidung zwischen einem System, das nur Plug-ins unterstützt, und einem "reinen" Mikrokernel.

Registry

Das Kernsystem muss wissen, welche Plug-in-Module verfügbar sind und wie man sie erreichen kann. Eine gängige Methode, dies zu realisieren, ist eine *Plug-in-Registrierung*. Diese Registry enthält Informationen über jedes Plug-in-Modul, z. B. den Namen, den Datenvertrag und die Details zum Fernzugriffsprotokoll (je nachdem, wie das Plug-in mit dem Kernsystem verbunden ist). Ein Plug-in für Steuersoftware, das risikobehaftete Steuerprüfungsposten markiert, könnte zum Beispiel einen Registry-Eintrag haben, der den Namen des Dienstes (AuditChecker), den Datenvertrag (Eingangs- und Ausgangsdaten) und das Vertragsformat (XML) enthält.

Die Registry kann so einfach sein wie eine interne Map-Struktur, die dem Kernsystem gehört und einen Schlüssel und die Referenz der Plug-in-Komponente enthält, oder sie kann so komplex sein wie ein Registry- und Discovery-Tool, das in das Kernsystem eingebettet ist oder extern eingesetzt wird (wie Apache ZooKeeper oder Consul). Anhand des Beispiels des Elektronikrecyclings implementiert der folgende Java-Code eine einfache Registry innerhalb des Kernsystems und zeigt Beispiele für einen Punkt-zu-Punkt-Eintrag, einen Messaging-Eintrag und einen RESTful-Eintrag zur Bewertung eines iPhone 6S-Geräts:

```
Map<String, String> registry = new HashMap<String, String>();
```

```
Map<String, String> registry = new HashMap<String, String>();
static {
  //point-to-point access example
  registry.put("iPhone6s", "Iphone6sPlugin");

  //messaging example
  registry.put("iPhone6s", "iphone6s.queue");

  //restful example
  registry.put("iPhone6s", "https://atlas:443/assess/iphone6s
}
```

Verträge

Die Verträge zwischen Plug-in-Komponenten und dem Kernsystem sind in der Regel standardmäßig in einer Domäne von Plug-in-Komponenten und umfassen das Verhalten, die Eingabedaten und die von der Plug-in-Komponente zurückgegebenen Ausgabedaten. Benutzerdefinierte Verträge werden in der Regel verwendet, wenn die Plug-in-Komponenten von einem Dritten entwickelt werden und der Architekt keine Kontrolle über den Vertrag hat, den das Plug-in verwendet. In solchen Fällen ist es üblich, einen Adapter zwischen dem Plug-in-Vertrag und deinem Standardvertrag zu erstellen, damit das Kernsystem nicht für jedes Plug-in einen eigenen Code benötigt.

Plug-in-Verträge können in XML, in JSON oder sogar in Objekten implementiert werden, die zwischen dem Plug-in und dem Kernsystem hin- und hergereicht werden. In der Elektronik-Recycling-Anwendung

definiert der folgende Vertrag (implementiert als Java-Standardschnittstelle mit dem Namen `AssessmentPlugin`) das Gesamtverhalten (`assess()`, `register()` und `deregister()`) sowie die entsprechenden Ausgabedaten, die von der Plug-in-Komponente erwartet werden (`AssessmentOutput`):

```
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;
}
```

In diesem Vertragsbeispiel wird erwartet, dass das Gerätebewertungs-Plug-in den Bewertungsbericht mit zurückgibt:

- Eine formatierte Zeichenkette
- Ein Wiederverkaufs-Flag (wahr oder falsch), das angibt, ob dieses Gerät auf einem Drittmarkt weiterverkauft oder sicher entsorgt werden kann
- Wenn der Artikel weiterverkauft werden kann, seinen berechneten Wert und den empfohlenen Wiederverkaufspreis

Beachte das Rollen- und Verantwortungsmodell zwischen dem Kernsystem und der Plug-in-Komponente in diesem Beispiel, insbesondere beim Feld `assessmentReport`. Es liegt nicht in der Verantwortung des Kernsystems, die Details des Beurteilungsberichts zu formatieren und zu verstehen, sondern nur, ihn entweder auszudrucken oder dem Nutzer anzuzeigen.

Daten-Topologien

In der Regel implementieren Teams Mikrokernel-Architekturen als monolithische Architekturen, die eine einzige (in der Regel relationale) Datenbank verwenden.

Es ist unüblich, dass sich Plug-in-Komponenten direkt mit einer zentral freigegebenen Datenbank verbinden. Stattdessen übernimmt das Kernsystem diese Aufgabe und gibt die benötigten Daten an die einzelnen Plug-ins weiter. Der Hauptgrund für diese Praxis ist die Entkopplung. Eine Änderung der Datenbank sollte sich nur auf das Kernsystem auswirken, nicht auf die Plug-in-Komponenten. Dennoch können Plug-ins separate Datenspeicher haben, auf die nur das jeweilige Plug-in Zugriff hat. So könnte z. B. jedes Plug-in für die Gerätebewertung im Going Green-System über eine eigene einfache Datenbank oder Regelmaschine verfügen, die alle spezifischen Bewertungsregeln für das jeweilige Produkt enthält. Der Datenspeicher, der der Plug-in-Komponente gehört, kann extern sein (wie in Abbildung 13-8 dargestellt) oder als Teil der Plug-in-Komponente oder der monolithischen

Bereitstellung eingebettet sein (wie im Fall einer In-Memory- oder eingebetteten Datenbank).

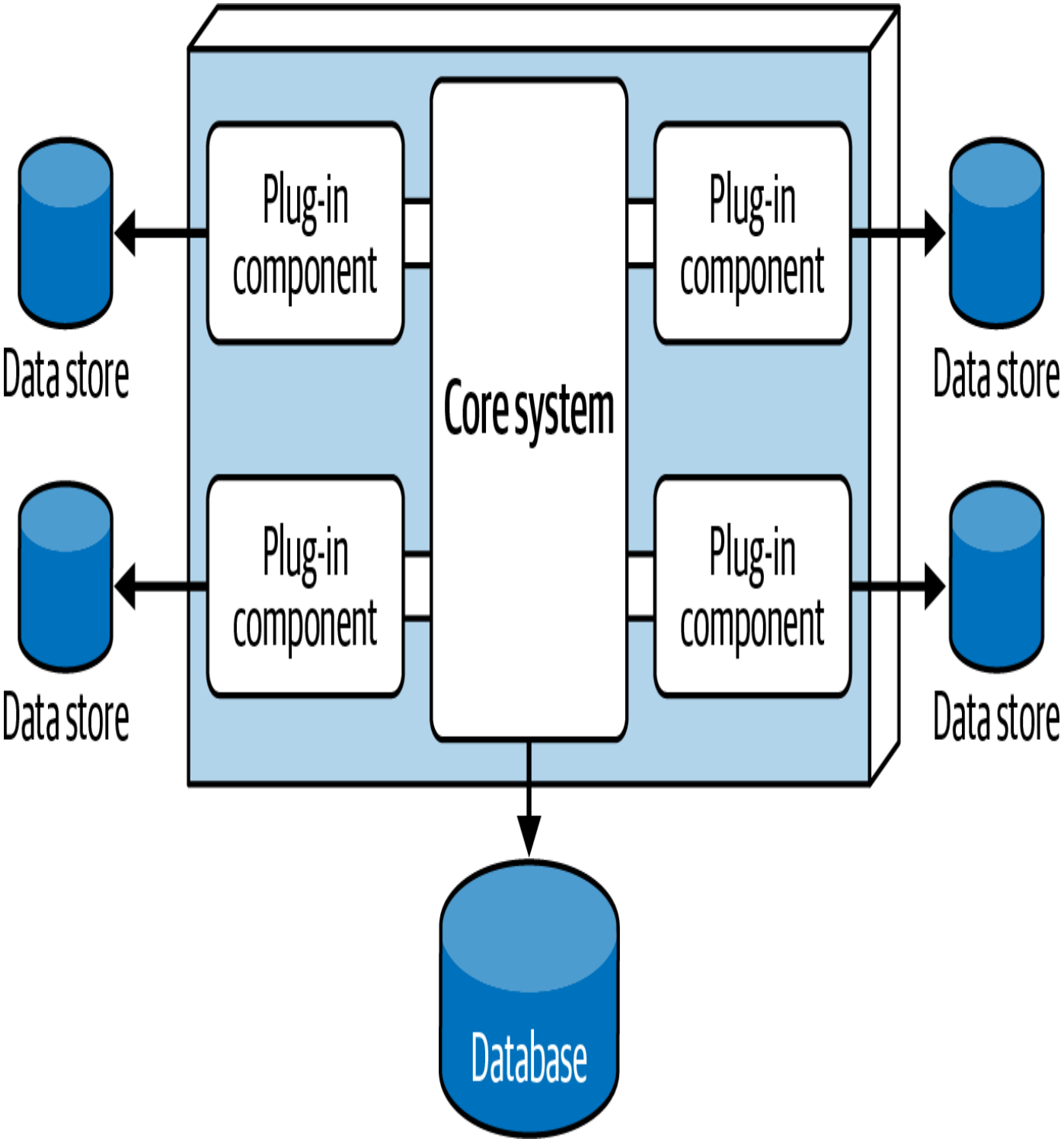


Abbildung 13-8. Eine Plug-in-Komponente kann ihren eigenen Datenspeicher besitzen

Überlegungen zur Cloud

Da Mikrokernel-Architekturen in der Regel monolithisch sind, bietet die Cloud ein paar grobkörnige Optionen. Die erste Option besteht darin, die gesamte Anwendung in der Cloud bereitzustellen, indem Cloud-Einrichtungen oder Container verwendet werden. Die zweite Möglichkeit besteht darin, nur die Daten in der Cloud bereitzustellen und den Mikrokernel als On-Premises-System zu implementieren. Die dritte Option trennt das Kernsystem als On-Premises-System ab und platziert die Plug-ins in der Cloud. Auch wenn dies unter dem Gesichtspunkt der Modularität eine gute Option zu sein scheint, bringt sie einige Herausforderungen für die Reaktionsfähigkeit mit sich. In einer Mikrokernel-Architektur werden Plug-ins häufig aufgerufen, und bei jedem Aufruf wird eine große Menge an Informationen übertragen, da die Teams wichtige Arbeitsabläufe mit Hilfe von Plug-ins implementieren. Die Latenzzeit, die durch die Trennung von Kern und Plug-ins entsteht, kann zu unerwünschtem Overhead führen.

Gemeinsame Risiken

Die üblichen Risiken, die mit dieser Architektur verbunden sind, ergeben sich hauptsächlich aus ihrer falschen Anwendung.

Flüchtiger Kern

Der Kern in einer Mikrokernel-Architektur soll nach der anfänglichen Entwicklung so stabil wie möglich sein; ein Vorteil dieses Stils ist, dass er Änderungen an Plug-ins isoliert. Die Entwicklung eines Kerns, der sich ständig ändert, untergräbt die Philosophie dieser Architektur, ist aber ein häufiger Fehler. Oftmals liegt es daran, dass die Architekten die Volatilität des Kerns falsch einschätzen; sie müssen diese Volatilität dann durch Refactoring ausgleichen.

Plug-In-Abhängigkeiten

Mikrokernel funktionieren am besten, wenn die Plug-ins nur mit dem Kernsystem und nicht untereinander kommunizieren. Die meisten Systeme, die Plug-ins verwenden, die keine Mikrokerne sind, verwenden *abhängigkeitsfreie Plug-ins*, d. h. Plug-ins, die keine anderen Abhängigkeiten als die des Kerns haben. Mit anderen Worten: Die Plug-ins kommunizieren nicht miteinander und haben daher keine gemeinsamen Abhängigkeiten, die vom Kern aufgelöst werden müssen. Komplexe Anwendungen der Mikrokernel-Architektur, wie z. B. die Eclipse IDE, bauen jedoch manchmal Abhängigkeiten zwischen Komponenten auf, so dass der Kern transitive Abhängigkeitskonflikte zwischen ihnen auflösen muss.

Es ist also möglich, Abhängigkeiten zu haben, aber sie führen zu unzähligen Komplexitäten bei der Verwaltung transitiver Abhängigkeiten. Was passiert, wenn zwei Plug-ins von verschiedenen Versionen der gleichen Kernbibliothek abhängen? Der Kern muss diese Abhängigkeiten auflösen und die Kommunikation zwischen den

verschiedenen Plug-in-Versionen ermöglichen. Jeder, der schon einmal mit dem Hinzufügen von Plug-ins in einer Umgebung mit transitiven Abhängigkeiten zu kämpfen hatte, weiß, wie schwer es ist, widersprüchliche Versionen zu entwirren. Am besten ist es, Abhängigkeiten zwischen Plug-ins so weit wie möglich zu vermeiden.

Governance

Bei der Steuerung einer Mikrokernel-Architektur geht es darum, zu überprüfen, wie gut die Architekten die Philosophie einhalten.

Zu den üblichen Kontrollen der Governance gehören:

- Volatilitätsprüfungen für den Kern - das sind Fitnessfunktionen, die in der Versionskontrolle für die Überprüfung von Abwanderung zuständig sind, und keine spezielle Codeprüfung
- Rate der Veränderung im Kern
- Vertragstests (vor allem, wenn einige Plug-ins aufgrund der schrittweisen Weiterentwicklung andere Versionen als andere unterstützen)
- Andere strukturelle Überprüfungen für die Topologie

Überlegungen zur Team-Topologie

Die offensichtliche Aufteilung der Teams in dieser Architektur ist zwischen Kern und Plug-ins, was die Topologie widerspiegelt:

Auf den Strom ausgerichtete Teams
Der Kern ist ein offensichtlicher "Sweet Spot" für Teams, die sich an den Streams orientieren und die Kernfunktionen des Systems entwickeln. Je nach Art der Anwendung können auch Plug-ins in dieses Team fallen.

Teams befähigen
Die Mikrokernel-Architektur eignet sich hervorragend für Enabling-Teams, da sie einige Verhaltensweisen in Plug-ins auslagert, um A/B-Tests und andereExperimente zu ermöglichen.

Teams mit komplizierten Subsystemen
Mikrokernel eignen sich auch gut für Teams, die mit komplizierten Subsystemen arbeiten, weil sie spezielle Funktionen auf Plug-ins verlagern. So können z. B. spezialisierte Verarbeitungen wie Analysen in Plug-ins isoliert werden, so dass das Stream-fähige Team am Kernverhalten arbeiten kann und für spezielles Verhalten auf ausgefeilte Plug-ins zurückgreifen kann.

Plattform-Teams
Wie bei anderen monolithischen Architekturen kümmern sich die Plattformteams auch bei dieser Architektur hauptsächlich um die betrieblichen Details.

Architektur Merkmale Bewertungen

Eine Ein-Stern-Bewertung bei den Merkmalen in Abbildung 13-9 bedeutet, dass das betreffende Architekturmerkmal in der Architektur nicht gut unterstützt wird, während eine Fünf-Stern-Bewertung bedeutet, dass das Architekturmerkmal eines der stärksten Merkmale des Architekturstils ist. Die Definition für jedes Merkmal in der Scorecard findest du in Kapitel 4.

Bei der Mikrokernel-Architektur sind, ähnlich wie bei der Schichtenarchitektur, Einfachheit und Gesamtkosten die Hauptstärken, während Skalierbarkeit, Fehlertoleranz und Elastizität die Hauptschwächen sind. Diese Schwächen sind auf die typischen monolithischen Implementierungen zurückzuführen, die für die Mikrokernel-Architektur typisch sind. Wie bei der Schichtenarchitektur ist das Architekturquantum immer singulär (1), da alle Anfragen das Kernsystem durchlaufen müssen, um zu den unabhängigen Plug-in-Komponenten zu gelangen. Hier enden die Ähnlichkeiten.

Der Mikrokernel ist einzigartig, weil er der einzige Architekturstil ist, der sowohl eine Domänenpartitionierung *als auch eine* technische Partitionierung ermöglicht. Während die meisten Microkernel-Architekturen technisch partitioniert sind, kommt der Aspekt der Domänenpartitionierung vor allem durch einen starken Isomorphismus zwischen Domänen und Architekturen zustande. Zum Beispiel passen Probleme, die unterschiedliche Konfigurationen für jeden Standort oder Client erfordern, sehr gut zu diesem Architekturstil. Das Gleiche gilt für Produkte oder Anwendungen, bei denen die Benutzeranpassung und die

Erweiterbarkeit von Funktionen im Vordergrund stehen (z. B. Jira oder eine IDE wie Eclipse).

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

Abbildung 13-9. Merkmale der Mikrokernel-Architektur Bewertungen

Testbarkeit, Einsetzbarkeit und Zuverlässigkeit liegen etwas über dem Durchschnitt (drei Sterne), vor allem weil die Funktionen in unabhängigen Plug-in-Komponenten isoliert werden können. Wenn dies richtig gemacht wird, verringert sich der Gesamtumfang der Tests bei Änderungen und das Gesamtrisiko des Einsatzes, insbesondere wenn die Plug-in-Komponenten zur Laufzeit eingesetzt werden.

Auch die Modularität und die Evolvierbarkeit liegen etwas über dem Durchschnitt (drei Sterne). Bei der Mikrokernel-Architektur können zusätzliche Funktionen durch unabhängige, in sich geschlossene Plug-in-Komponenten hinzugefügt, entfernt und geändert werden. Das macht es relativ einfach, Anwendungen zu erweitern und zu verbessern, und ermöglicht es den Teams, viel schneller auf Änderungen zu reagieren. Nimm das Beispiel der Steuererstellungssoftware aus dem vorherigen Abschnitt. Wenn sich das US-Steuergesetz ändert (und das tut es ständig) und ein neues Steuerformular erforderlich wird, kann dieses neue Steuerformular als Plug-in-Komponente erstellt und der Anwendung ohne großen Aufwand hinzugefügt werden. Wenn ein Steuerformular oder Arbeitsblatt nicht mehr benötigt wird, kann das Plug-in einfach aus der Anwendung entfernt werden.

Die Reaktionsfähigkeit ist immer eine interessante Eigenschaft, wenn es um die Bewertung der Mikrokernel-Architektur geht. Wir haben ihr drei Sterne gegeben (etwas über dem Durchschnitt), vor allem weil Mikrokernel-Anwendungen in der Regel klein sind und nicht so groß

werden wie die meisten Schichtenarchitekturen. Außerdem leiden sie nicht so sehr unter demArchitektur-Sinkhole-Antipattern, das wir in Kapitel 10 besprochen haben. Schließlich können Mikrokernel-Architekturen durch das Herausnehmen nicht benötigter Funktionen optimiert werden, wodurch die Anwendung schneller läuft. Ein gutes Beispiel dafür ist WildFly (der frühere JBoss Application Server). Wenn unnötige Funktionen wie Clustering, Caching und Messaging entfernt werden, läuft der Anwendungsserver viel schneller als mit diesen Funktionen.

Beispiele und Anwendungsfälle

Die meisten Tools für die Entwicklung und Veröffentlichung von Software werden mit der Microkernel-Architektur implementiert. Einige Beispiele sind die Eclipse IDE, PMD, Jira und Jenkins, um nur einige zu nennen. Auch Internetbrowser wie Chrome und Firefox nutzen häufig die Microkernel-Architektur: Viewer und andere Plug-ins fügen zusätzliche Funktionen hinzu, die im Basisbrowser (dem Kernsystem) nicht enthalten sind. Wir könnten endlose Beispiele für produktbasierte Software anführen, aber was ist mit großen Unternehmensanwendungen? Die Mikrokernel-Architektur gilt auch für diese Situationen.

Um diesen Punkt zu verdeutlichen, betrachten wir unser früheres Beispiel der Software für die Steuererstellung. Die US-Steuerbehörde, der Internal Revenue Service, hat ein zweiseitiges Steuerformular, das sogenannte 1040-Formular, das eine Zusammenfassung aller

Informationen enthält, die zur Berechnung der Steuerschuld einer Person benötigt werden. Jede Zeile des Steuerformulars 1040 enthält eine einzelne Zahl, z. B. das Bruttoeinkommen, und um diese Zahlen zu ermitteln, sind viele weitere Formulare und Arbeitsblätter erforderlich. Jedes zusätzliche Formular und Arbeitsblatt kann als Plug-in-Komponente implementiert werden, wobei das zusammenfassende Steuerformular 1040 das Kernsystem (der Treiber) ist. Auf diese Weise können Änderungen am Steuerrecht in einer unabhängigen Plug-in-Komponente isoliert werden, was Änderungen einfacher und weniger riskant macht.

Ein weiteres Beispiel für eine große, komplexe Geschäftsanwendung, die die Microkernel-Architektur nutzen kann, ist die Bearbeitung von Versicherungsansprüchen. Die Bearbeitung von Versicherungsansprüchen ist ein sehr komplizierter Prozess. In jedem Land gibt es unterschiedliche Regeln und Vorschriften dafür, was in einem Versicherungsfall erlaubt ist und was nicht. In einigen Ländern (z. B. in den USA) ist es zum Beispiel erlaubt, dass die Versicherung die Windschutzscheibe kostenlos austauscht, wenn sie durch einen Stein beschädigt wurde, in anderen nicht. Dadurch ergeben sich fast unendlich viele Bedingungen für einen Standardschadenprozess.

Die meisten Anwendungen für Versicherungsansprüche nutzen große, komplexe Rules Engines, um einen Großteil dieser Komplexität zu bewältigen. Eine *Rules Engine* ist ein Framework oder eine Bibliothek, die es Entwicklern (oder Endbenutzern) ermöglicht, eine Reihe von Regeln oder Schritten zu definieren, um einen Arbeitsablauf deklarativ

festzulegen, entweder mit visuellen Tools oder einer domänenspezifischen Sprache. Diese Regel-Engines können sich jedoch zu einem großen Schlammball auswachsen, bei dem eine einfache Regeländerung ein Heer von Analysten, Entwicklern und Testern erfordert, um sicherzustellen, dass nichts kaputt geht. Die Verwendung des Mikrokernel-Architekturmusters kann viele dieser Probleme lösen.

Die Anspruchsregeln für die einzelnen Gerichtsbarkeiten können in separaten, eigenständigen Plug-in-Komponenten enthalten sein, die als Quellcode oder als eine spezielle Instanz der Rules-Engine implementiert sind, auf die die Plug-in-Komponente zugreift. Auf diese Weise können Regeln für ein bestimmtes Rechtsgebiet hinzugefügt, entfernt oder geändert werden, ohne dass ein anderer Teil des Systems davon betroffen ist. Außerdem können neue Gerichtsbarkeiten hinzugefügt und entfernt werden, ohne dass sich dies auf andere Teile des Systems auswirkt. Das Kernsystem wäre in diesem Beispiel der Standardprozess für die Einreichung und Bearbeitung eines Anspruchs - etwas, das sich nicht oft ändert.

Der Stil der Mikrokernel-Architektur ist sehr verbreitet; wenn du ihn einmal gesehen hast, fällt er dir überall auf. Es ist ein Fall, in dem eine Architekturstruktur (Kern + Plug-ins) mit dem allgemeinen Problem der Anpassung übereinstimmt, und es stellt sich heraus, dass die Anpassung in der Software sehr häufig vorkommt.