Kapitel 12. Stil der Pipeline-Architektur

Einer der grundlegenden Stile in der Softwarearchitektur ist die *Pipeline-Architektur* (auch bekannt als *Pipes-and-Filter-Architektur*). Sobald Entwickler und Architekten beschlossen, Funktionen in einzelne Teile aufzuteilen, folgte dieser Architekturstil. Die meisten Entwicklerinnen und Entwickler kennen diese Architektur als das Grundprinzip von Unix-Terminal-Shell-Sprachen wie Bash und Zsh.

Entwickler in vielen funktionalen Programmiersprachen werden Parallelen zwischen Sprachkonstrukten und Elementen dieser Architektur erkennen. Tatsächlich folgen viele Tools, die das MapReduce-Programmiermodell verwenden, dieser grundlegenden Topologie. Die Beispiele in diesem Kapitel zeigen zwar Low-Level-Implementierungen der Pipeline-Architektur, sie kann aber auch für anspruchsvollere Geschäftsanwendungen verwendet werden.

Topologie

Die Topologie der Pipeline-Architektur besteht aus zwei Hauptkomponententypen: Pipes und Filter. *Filter* enthalten die Systemfunktionalität und führen eine bestimmte Geschäftsfunktion aus, und *Pipes* übertragen Daten an den nächsten Filter (oder die nächsten

Filter) in der Kette. Sie koordinieren sich auf eine bestimmte Art und Weise, wobei Pipes eine einseitige, meist Punkt-zu-Punkt-Kommunikation zwischen Filtern bilden, wie in Abbildung 12-1 dargestellt.

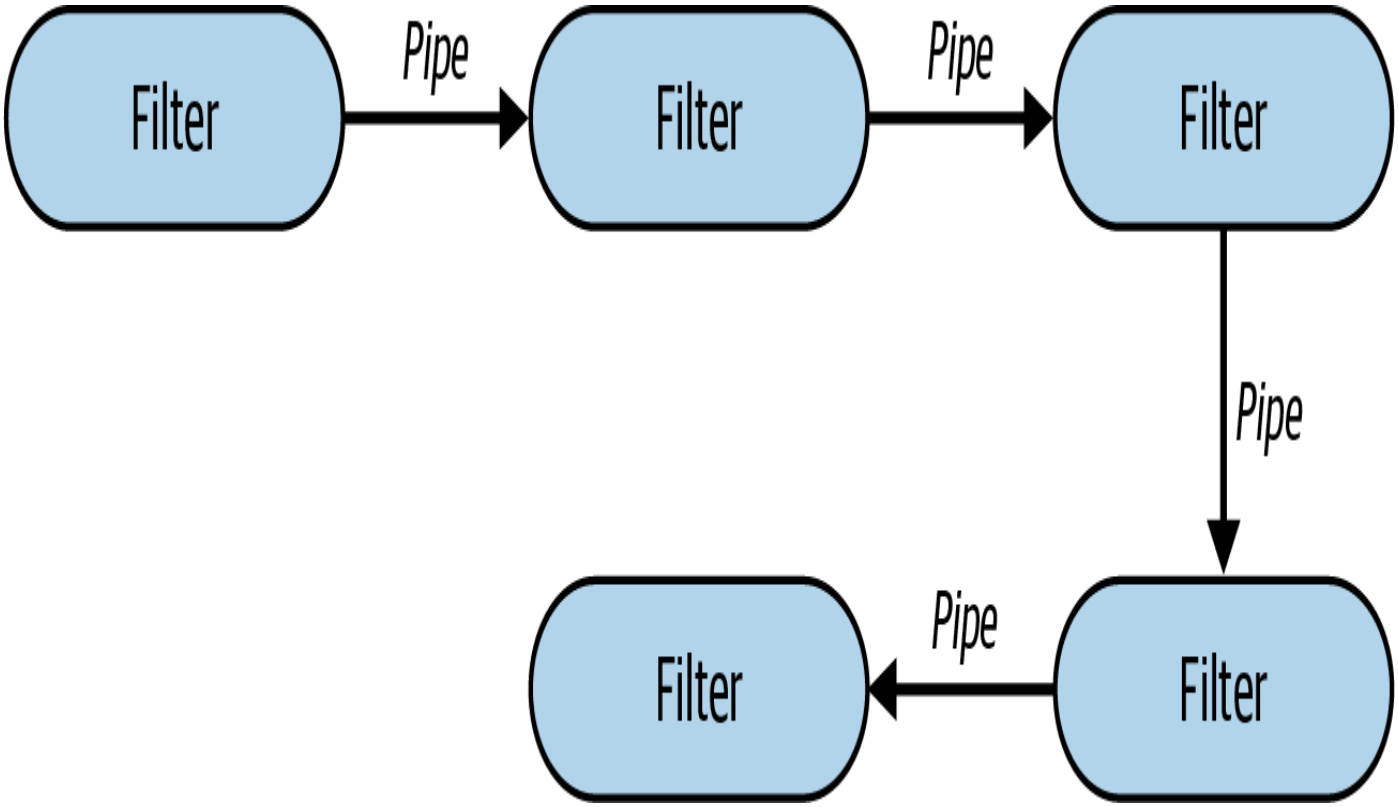

Abbildung 12-1. Grundtopologie für die Pipeline-Architektur

Die isomorphe "Form" der Pipeline-Architektur ist also eine *einzige Einsatzeinheit, deren Funktionen in Filtern enthalten sind, die durch unidirektionale Pipes verbunden sind.*

Stil Besonderheiten

Während die meisten Implementierungen der Pipeline-Architektur monolithisch sind, ist es möglich, jeden Filter (oder eine Reihe von

Filtern) als Dienst einzusetzen und so eine verteilte Architektur mit synchronen oder asynchronen Fernaufrufen zu jedem Dienst zu schaffen. Unabhängig von der Einsatztopologie besteht die Architektur aus nur zwei Komponenten, den Filtern und den Pipes, die wir in den folgenden Abschnitten ausführlich beschreiben.

Filter

Filter sind in sich geschlossene Funktionseinheiten, die von anderen Filtern unabhängig sind. Sie sind in der Regel zustandslos und sollten nur eine Aufgabe erfüllen. Zusammengesetzte Aufgaben werden in der Regel von einer Reihe von Filtern und nicht von einem einzigen bearbeitet.

Da Filter durch mehr als eine Klassendatei implementiert werden können, werden sie als *Komponenten* der Architektur betrachtet (siehe Kapitel 8). Auch wenn ein Filter einfach ist und nur durch eine einzige Klassendatei implementiert wird, ist er dennoch eine Komponente.

Es gibt vier Arten von Filtern im Stil der Pipeline-Architektur:

Produzent
Der Startpunkt eines Prozesses, Erzeugerfilter sind nur ausgehend. Sie werden manchmal auch als *Quelle* bezeichnet. Eine Benutzeroberfläche und eine externe Anfrage an das System sind beides Beispiele für Erzeugerfilter.

Transformator

Transformer-Filter nehmen Eingaben entgegen, führen optional eine Transformation an einigen oder allen Daten durch und leiten die Daten dann an die Outbound-Pipe weiter. Verfechter der funktionalen Programmierung werden diese Funktion als *Map* kennen. Transformer-Filter können z. B. Daten verbessern, umwandeln oder eine Berechnung durchführen.

Tester
Prüffilter nehmen Eingaben entgegen, prüfen sie nach einem oder mehreren Kriterien und erzeugen dann optional eine Ausgabe, die auf dem Test basiert. Funktionale Programmierer/innen werden dies als ähnlich zu *reduzieren* erkennen. Ein Prüffilter kann prüfen, ob alle Daten gültig sind und korrekt eingegeben wurden, oder als Schalter fungieren, um zu bestimmen, ob die Verarbeitung fortgesetzt werden soll (z. B. "die Daten nicht an den nächsten Filter weiterleiten, wenn der Bestellbetrag weniger als fünf Dollar beträgt").

Verbraucher
Als Endpunkt des Pipeline-Flusses speichern Verbraucherfilter manchmal das Endergebnis des Pipeline-Prozesses in einer Datenbank oder zeigen die Endergebnisse auf einem UI-Bildschirm an.

Die unidirektionale Natur und die Einfachheit von Pipes und Filtern begünstigen die kompositorische Wiederverwendung. Viele Entwickler haben diese Fähigkeit mit Shells entdeckt. Eine berühmte Geschichte aus dem Blog "More Shell, Less Egg" veranschaulicht, wie mächtig diese Abstraktionen sind. Donald Knuth wurde gebeten, ein Programm zu schreiben, das ein Textverarbeitungsproblem löst: Er sollte eine

Textdatei lesen, die *n* am häufigsten verwendeten Wörter ermitteln und eine nach Häufigkeit sortierte Liste dieser Wörter ausgeben. Er schrieb ein Programm, das aus mehr als 10 Seiten Pascal bestand, und entwickelte (und dokumentierte) dabei einen neuen Algorithmus. Dann demonstrierte Doug McIlroy ein Shell-Skript, das so klein ist, dass es problemlos in einen Social-Media-Post passt und das Problem elegant löst:

```
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed ${1}q
```

Selbst die Designer von Unix-Shells sind oft überrascht, wie einfallsreich die Entwickler ihre einfachen, aber leistungsstarken zusammengesetzten Abstraktionen einsetzen.

Rohre

Pipes bilden in dieser Architektur den Kommunikationskanal zwischen den Filtern. Jede Pipe ist in der Regel unidirektional und Punkt-zu-Punkt, sie nimmt Eingaben von einer Quelle entgegen und leitet die Ausgaben an eine andere weiter. Die Nutzdaten können in jedem beliebigen Format vorliegen, aber die Architekten bevorzugen in der Regel kleinere Datenmengen, um eine hohe Leistung zu erzielen.

Wenn ein Filter (oder eine Gruppe von Filtern) als separater Dienst in einer verteilten Umgebung eingesetzt wird, rufen die Pipes einen unidirektionalen Fernzugriff über REST, Messaging, Streaming oder ein anderes Fernkommunikationsprotokoll auf. Unabhängig davon, ob es sich um eine monolithische oder verteilte Bereitstellung handelt, können Pipes entweder synchron oder asynchron sein. In monolithischen Implementierungen verwenden Architekten Threads oder eingebettetes Messaging für die asynchrone Kommunikation mit einem Filter.

Daten-Topologien

Da die meisten Pipeline-Architekturen als Monolithen implementiert werden, sind sie mit einer einzigen monolithischen Datenbank verbunden. Das ist jedoch nicht immer der Fall. Die Datenbanktopologie kann bei diesem Architekturstil stark variieren, von einer einzigen Datenbank bis hin zu einer Datenbank pro Filter.

Das Beispiel in Abbildung 12-2 zeigt eine Pipeline-Architektur für eine typische kontinuierliche Fitnessfunktion (Architekturtest), die in einer Produktionsumgebung läuft und ein bestimmtes Betriebsmerkmal (wie Reaktionsfähigkeit oder Skalierbarkeit) analysiert. Beachte, dass der Filter "Rohdaten erfassen" eine separate Datenbank mit Rohdaten lädt. Dieser Filter sendet die Rohdaten über eine Pipe an den Filter Time Series Selector, der Konfigurationsinformationen (z. B. den zu analysierenden Zeitraum) aus einer separaten Datenbank liest. Die umgewandelten Daten werden dann über eine weitere Pipe an den

Trend Analyzer-Filter gesendet, der die Daten analysiert und diese Analysen in einer separaten Datenbank speichert. Dieser Filter sendet die Analysedaten über eine letzte Pipe an den Graphing Tool-Filter, der die Pipeline abschließt, indem er einen grafischen Bericht über die Daten erstellt.

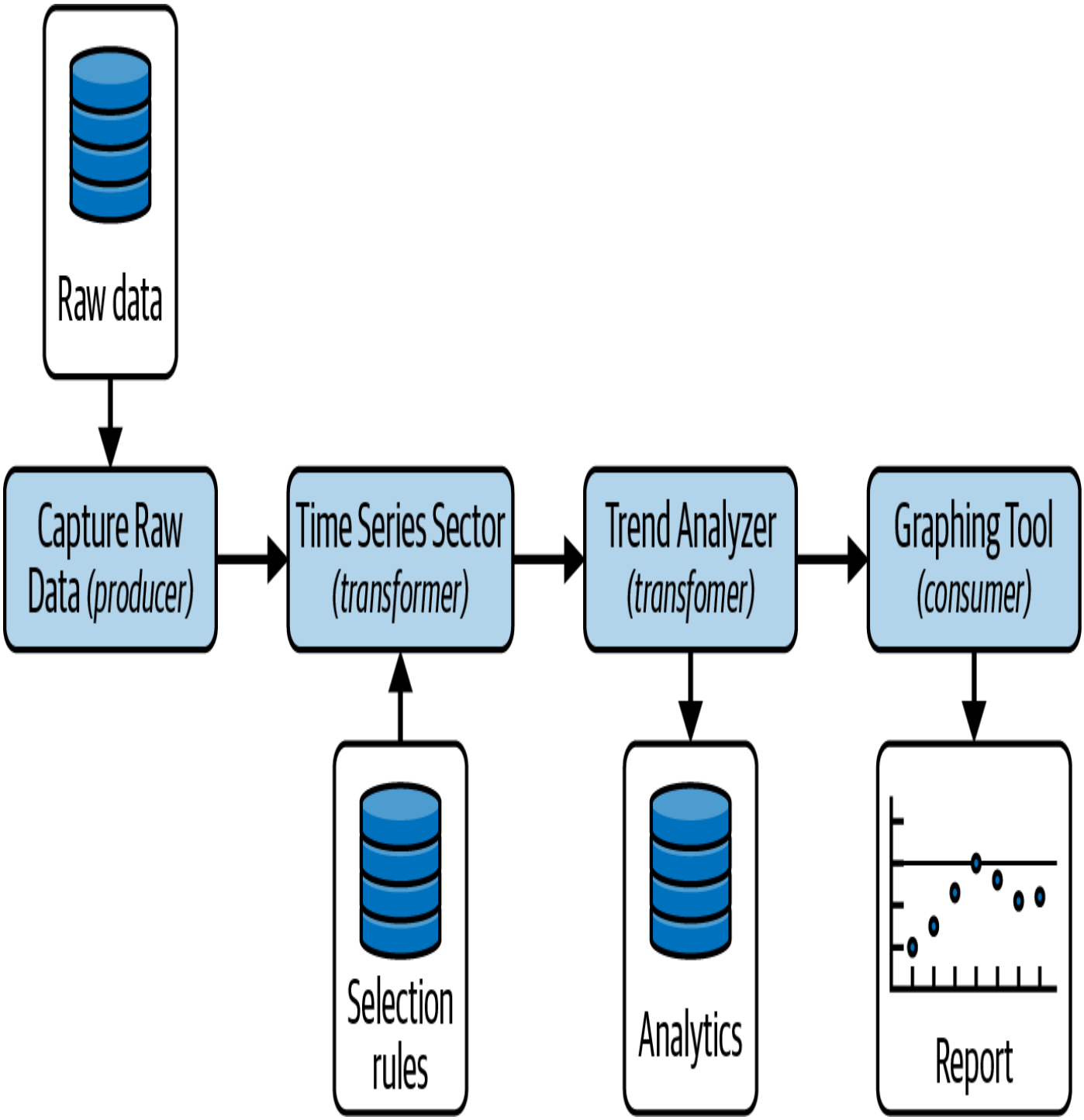

Abbildung 12-2. Pipeline-Architekturen können eine einzige Datenbank oder viele

Überlegungen zur Cloud

Die Pipeline-Architektur eignet sich aufgrund ihres hohen Maßes an Modularität und der separaten Filtertypen gut für cloudbasierte Einsätze.

Da die meisten Pipeline-Architekturen nicht übermäßig komplex und umfangreich sind, eignen sie sich gut für den Einsatz als monolithische Architekturen, bei denen alle Filter in derselben Einsatzeinheit eingesetzt werden.

Filter funktionieren aber auch gut in Cloud-Umgebungen als verteilte Funktionen. In AWS kann die Pipeline-Architektur als AWS Step Functions eingesetzt werden, wobei jeder Filter als separater Lambda im Workflow eingesetzt wird. AWS Step Functions bieten zwei Arbeitsabläufe: *Standard*, bei dem jeder Schritt genau einmal ausgeführt wird, und *Express*, bei dem jeder Schritt mehr als einmal ausgeführt werden kann. Die Pipeline-Architektur funktioniert mit beiden. Der folgende Code zeigt das Beispiel der kontinuierlichen Fitnessfunktion in Abbildung 12-2 als typische Cloud-Bereitstellung für die Pipeline-Architektur, die als AWS Step Function implementiert wurde:

```
{
  "Comment": "Measure and analyze scalability trends.",
  "StartAt": "Capture Raw Data",
  "States": {
    "Capture Raw Data": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:region:account_id:function:ra
    "Next": "Time Series Selector"
   },
    "Time Series Selector": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:region:account_id:function:ti
    "Next": "Trend Analyzer"
```

```
    },
     "Trend Analyzer": {
     "Type": "Task",
     "Resource": "arn:aws:lambda:region:account_id:function:tr
     "Next": "Graphing Tool"
     },
     "Graphing Tool": {
     "Type": "Task",
     "Resource": "arn:aws:lambda:region:account_id:function:gr
     "End": true
     }
   }
}
```

Dieses Beispiel ist nicht die einzige Möglichkeit, die Pipeline-Architektur in Cloud-Umgebungen zu nutzen. Filter können auch als serverlose Funktionen, als containerisierte Funktionen oder sogar als ein einzelner Dienst mit allen vier Filterkomponenten in einer monolithischen Bereitstellung eingesetzt werden.

Gemeinsame Risiken

Das Hauptziel der Pipeline-Architektur ist es, die Funktionalität in Einzweck-Filter aufzuteilen, wobei jeder Filter *eine* bestimmte Aktion mit den Daten durchführt und sie dann an einen anderen Filter zur weiteren Verarbeitung weitergibt. Eines der häufigsten Risiken ist daher die Überlastung der Filter mit zu viel Verantwortung. Eine gute Governance,

auf die wir im nächsten Abschnitt eingehen, hilft den Teams, dieses Risiko zu mindern, indem sie den Zweck hinter jeder Filterkomponente ermittelt.

Ein weiteres Risiko bei dieser Architektur ist die Einführung einer bidirektionalen Kommunikation zwischen Filtern. Pipelines sollen *nur unidirektional* sein und eine klare Trennung zwischen den Filtern gewährleisten, um eine Zusammenarbeit zwischen ihnen zu vermeiden. Wenn eine bidirektionale Kommunikation notwendig ist, ist das ein guter Hinweis darauf, dass die Pipeline-Architektur nicht der richtige Stil ist oder dass die Filter zu komplex sind; die Funktionen sind nicht richtig abgegrenzt.

Der Umgang mit Fehlerzuständen ist eine weitere Komplikation, die bei diesem Architekturstil ein erhebliches Risiko darstellen kann. Wenn in einer Pipeline ein Fehler auftritt, ist es oft schwierig zu bestimmen, wie die Pipeline ordnungsgemäß beendet und wiederhergestellt werden kann, sobald die Pipeline gestartet wurde. Aus diesem Grund ist es für Architekten wichtig, mögliche fatale Fehlerbedingungen innerhalb der Pipeline zu bestimmen, bevor sie die Architektur festlegen.

Der letzte Risikobereich ist die Verwaltung der Verträge zwischen den Filtern. Jede Pipe hat einen Vertrag, der die Daten (und möglicherweise die entsprechenden Typen) enthält, die sie an den nächsten Filter sendet. Die Änderung eines Vertrags zwischen Filtern erfordert eine strenge Steuerung und Prüfung, um sicherzustellen, dass die anderen Filter, die den Vertrag erhalten, nicht zusammenbrechen.

Governance

Die Steuerung allgemeiner betrieblicher Merkmale - wie Reaktionsfähigkeit, Skalierbarkeit, Verfügbarkeit usw. - ist spezifisch für jeden einzelnen Anwendungsfall und kann stark variieren. Vom *strukturellen* Standpunkt aus betrachtet, nutzen Architekten jedoch die Rolle und Verantwortung jedes Filtertyps, um Pipeline-Architekturen zu steuern.

Jeder der vier grundlegenden Filtertypen (Producer, Transformer, Tester und Consumer) erfüllt eine bestimmte Aufgabe. Es ist jedoch allzu leicht für Entwickler, die Filter mit zu viel Verantwortung zu überfrachten und die Pipeline-Architektur in einen unstrukturierten Monolithen zu verwandeln.

Es ist schwierig, eine automatisierte Fitnessfunktion zu schreiben, um zu bestimmen, ob ein Producer-Filter tatsächlich der Startpunkt der Pipeline ist oder ob ein Tester-Filter tatsächlich eine bedingte Prüfung durchführt, um zu bestimmen, ob der Fluss fortgesetzt oder beendet werden soll. Governance-Techniken helfen Architekten dabei, das Entwicklungsteam dabei zu unterstützen, die Rolle der einzelnen Filtertypen und die Verantwortung eines bestimmten Filtertyps einzuhalten.

Eine dieser Techniken ist der Einsatz von *Tags*. (In Java werden Tags als *Annotationen* implementiert, in C# als *benutzerdefinierte Attribute*). Tags erfüllen keine eigentliche Funktion, sondern liefern programmatisch

Metadaten über die Komponente oder den Dienst. Durch die Verwendung von Tags in der Startklasse der Filterkomponente erfahren die Entwickler, welche Art von Filter sie erstellen oder ändern, und können so verhindern, dass sie dem Filter zu viel Verantwortung übertragen oder eine Funktion ausführen, die nicht zu seinem Typ passt.

Um diese Technik zu veranschaulichen, betrachten wir den folgenden Quellcode, der Tags für die vier grundlegenden Filtertypen definiert. Hier ist die Definition der Java-Tag-Annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Filter {
   public FilterType[] value();

   public enum FilterType {
      PRODUCER,
      TESTER,
      TRANSFORMER,
      CONSUMER
   }
}
```

Und hier ist die Definition des benutzerdefinierten Attributs für den C#-Tag:

```
[System.AttributeUsage(System.AttributeTargets.Class)]
class Filter : System.Attribute {
```

```
    public FilterType[] filterType;

    public enum FilterType {
        PRODUCER,
        TESTER,
        TRANSFORMER,
        CONSUMER
    };
}
```

Da Filter im Wesentlichen eine Architekturkomponente sind, können sie durch mehrere Klassendateien implementiert werden. In unserem Beispiel für eine kontinuierliche Fitnessfunktion in Abbildung 12-2 ist der Trend-Analyzer-Filter ziemlich komplex und könnte aus mehreren Klassendateien bestehen. Aus diesem Grund definieren wir ein zusätzliches Tag, um die Klasse zu identifizieren, die den Einstiegspunkt des Filters darstellt, damit wir andere Tags an sie anhängen können.

Hier ist die Definition des Java-Einstiegspunkt-Tags:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FilterEntrypoint {}
```

Und die C#-Einstiegspunkt-Tag-Definition:

```
[System.AttributeUsage(System.AttributeTargets.Class)]
class FilterEntrypoint : System.Attribute {}
```

Jetzt können die Entwickler der Klasse `Entrypoint` das Tag `FilterType` hinzufügen, das ihren Typ und die entsprechende Rolle in der Architektur kennzeichnet. Die folgenden Anmerkungen kennzeichnen zum Beispiel den Typ und die Rolle des Trend Analyzer Transformationsfilters. In Java:

```
@FilterEntrypoint
@Filter(FilterType.TRANSFORMER)
public class TrendAnalyzerFilter {
    ...
}
```

Und in C#:

```
[FilterEntrypoint]
[Filter(FilterType.TRANSFORMER)]
class TrendAnalyzerFilter {
    ...
}
```

Diese Technik kann zwar nicht jeden Entwickler davon abhalten, einen Transformator-Filtertyp für die Ausführung von Prüflogik zu verwenden (was von einem Tester-Filter erledigt werden sollte), aber sie bietet zumindest zusätzlichen Kontext.

Überlegungen zur Team-Topologie

Im Gegensatz zu einigen der in diesem Buch beschriebenen Architekturstile ist der Pipeline-Architekturstil im Allgemeinen unabhängig von der Team-Topologie und funktioniert mit jeder Team-Konfiguration:

Auf den Strom ausgerichtete Teams
Da die Pipeline-Architektur in der Regel klein und in sich geschlossen ist und eine einzige Reise oder einen einzigen Fluss durch das System darstellt, funktioniert sie gut mit Teams, die auf einen Fluss ausgerichtet sind. Bei dieser Team-Topologie besitzen die Teams in der Regel den Fluss durch das System von Anfang bis Ende, was gut zur Form der Pipeline-Architektur passt.

Teams befähigen
Da die Pipeline-Architektur hochgradig modular und nach technischen Gesichtspunkten getrennt ist, lässt sie sich gut mit Team-Topologien kombinieren. Spezialisten und übergreifende Teammitglieder können Vorschläge machen und Experimente durchführen, indem sie zusätzliche Filter in die Pipeline einfügen, ohne den Rest des Ablaufs zu beeinträchtigen. In unserem Beispiel mit der kontinuierlichen Fitnessfunktion könnte ein Enabling Team zum Beispiel einen Transformationsfilter nach dem Time Series Selector Filter hinzufügen, um eine alternative Trendanalyse durchzuführen. Dieser neue Filter würde dieselben Daten wie der bestehende Trendanalysefilter verwenden, ohne den normalen Ablauf der Pipeline zu unterbrechen.

Teams mit komplizierten Subsystemen

Da jeder Filter eine ganz bestimmte Aufgabe erfüllt, funktioniert die Pipeline-Architektur gut mit der Team-Topologie des komplizierten Teilsystems. Verschiedene Teammitglieder können sich unabhängig voneinander (und von anderen Filtern) auf die Bearbeitung komplizierter Filter konzentrieren. Die unidirektionalen Übergaben, die mit dieser Architektur einhergehen, ermöglichen es den Mitgliedern von Teams für komplizierte Teilsysteme, sich genau auf die Komplexität zu konzentrieren, die in ihrer spezifischen Filterverarbeitung enthalten ist.

Plattform-Teams
Plattformteams, die an einer Pipeline-Architektur arbeiten, können den hohen Grad an Modularität nutzen, indem sie gemeinsame Tools, Dienste, APIs und Aufgaben verwenden.

Stilmerkmale

Eine Ein-Stern-Bewertung in der Tabelle mit den Merkmalen(Abbildung 12-3) bedeutet, dass ein bestimmtes 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 in der Scorecard enthaltenen Merkmale werden in Kapitel 4 beschrieben und definiert.

Die Pipeline-Architektur ist eine *technisch partitionierte* Architektur, da die Anwendungslogik in Filtertypen aufgeteilt ist. Da die Pipeline-

Architektur in der Regel als monolithischer Einsatz implementiert wird, ist ihr Architektur-Quantum immer 1.

	Architectural characteristic	Star rating
	Overall cost	$
Structural	Partitioning type	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	★

Abbildung 12-3. Pipeline-Architektur Merkmale Bewertungen

Gesamtkosten, Einfachheit und Modularität sind die wichtigsten Stärken der Pipeline-Architektur. Da Pipeline-Architekturen monolithisch sind, weisen sie nicht die Komplexität auf, die mit verteilten Architekturen verbunden ist - sie sind einfach und leicht zu verstehen und relativ kostengünstig zu erstellen und zu warten. Die Modularität der Architektur wird durch die Trennung der verschiedenen Filtertypen und Transformatoren erreicht: Jeder Filter kann geändert oder ersetzt werden, ohne die anderen Filter zu beeinflussen. In dem in Abbildung 12-4 gezeigten Kafka-Beispiel kann die `Duration Calculator` beispielsweise so verändert werden, dass die Berechnung der Dauer geändert wird, ohne dass ein anderer Filter verändert wird.

Einsetzbarkeit und Testbarkeit sind zwar nur durchschnittlich, aber aufgrund der Modularität, die Filter erreichen können, etwas besser als bei der Schichtenarchitektur. Pipeline-Architekturen sind jedoch immer noch *typischerweise* monolithisch. Zu ihren Nachteilen gehören daher die Zeremonie, das Risiko, die Häufigkeit der Bereitstellung und die Vollständigkeit der Tests.

Elastizität und Skalierbarkeit werden bei der Pipeline-Architektur sehr niedrig bewertet (ein Stern), was vor allem auf die monolithischen Implementierungen zurückzuführen ist. Die Implementierung dieses Architekturstils als verteilte Architektur mit asynchroner Kommunikation kann diese Eigenschaften deutlich verbessern, aber der Kompromiss ist ein Schlag für die Gesamtkosten und die Einfachheit.

Pipeline-Architekturen unterstützen keine Fehlertoleranz, da sie in der Regel als monolithische Systeme implementiert werden. Wenn bei einem kleinen Teil einer Pipeline-Architektur ein Out-of-Memory-Zustand auftritt, ist die gesamte Anwendung davon betroffen und stürzt ab. Darüber hinaus wird die Gesamtverfügbarkeit durch die hohe mittlere Wiederherstellungszeit (MTTR) beeinträchtigt, die bei den meisten monolithischen Anwendungen üblich ist, wobei die Startzeiten in Minuten gemessen werden. Wie bei der Elastizität und Skalierbarkeit kann die Implementierung dieses Architekturstils als verteilte Architektur mit asynchroner Kommunikation die Fehlertoleranz erheblich verbessern, was wiederum mit Kosten und Komplexität erkauft wird.

Wie wir bereits erwähnt haben, können die meisten der schlecht bewerteten Betriebseigenschaften verbessert werden, indem wir diese Architektur zu einer verteilten Architektur mit asynchroner Kommunikation machen, bei der jeder Filter eine separate Einsatzeinheit ist und die Pipes Fernaufrufe sind. Dies wirkt sich jedoch negativ auf andere Eigenschaften wie Einfachheit und Kosten aus und verdeutlicht einen der klassischen Kompromisse der Softwarearchitektur.

Wann verwendet man

Die Pipeline-Architektur eignet sich für beliebig komplexe Systeme mit eindeutigen, geordneten und deterministischen Verarbeitungsschritten

in eine Richtung. Dank ihrer Einfachheit eignet sie sich auch gut für Situationen, in denen der Zeit- und Kostenrahmen eng gesteckt ist.

Wann man nicht verwenden sollte

Da diese Architektur monolithisch ist, eignet sie sich nicht für Systeme, die eine hohe Skalierbarkeit, Elastizität und Fehlertoleranz benötigen. Ein verteilter Architekturansatz kann diese Probleme jedoch entschärfen.

Die Pipeline-Architektur eignet sich auch nicht für Szenarien, in denen eine Hin- und Her-Kommunikation zwischen Filtern stattfindet, da die Pipes nur unidirektional sein sollen. Es ist zwar möglich, diesen Architekturstil für nicht-deterministische Workflows zu verwenden, indem man viele Prüffilter in der Pipeline einsetzt, aber wir raten davon ab. Es verkompliziert diesen relativ einfachen Architekturstil zu sehr und kann sich negativ auf die Wartbarkeit, die Testbarkeit, die Verteilbarkeit und folglich auf die allgemeine Zuverlässigkeit auswirken. Die ereignisgesteuerte Architektur (siehe Kapitel 15) eignet sich viel besser für Situationen, in denen es um nicht-deterministische Workflows geht.

Die Pipeline-Architektur kommt in einer Vielzahl von Anwendungen vor, insbesondere bei Aufgaben, die eine einfache, einseitige Verarbeitung ermöglichen. Zum Beispiel nutzen viele Tools für den elektronischen Datenaustausch (EDI) dieses Muster, indem sie mithilfe von Pipes und Filtern Transformationen von einem Dokumententyp in einen anderen

vornehmen. Tools zum Extrahieren, Transformieren und Laden von Datenbanken (ETL) nutzen Pipeline-Architekturen, um Daten zu verändern und von einer Datenbank oder Datenquelle in eine andere zu übertragen. Orchestratoren und Mediatoren wie Apache Camel nutzen die Pipeline-Architektur, um Informationen von einem Schritt in einem Geschäftsprozess zu einem anderen zu übertragen.

Beispiele und Anwendungsfälle

Um zu veranschaulichen, wie die Pipeline-Architektur genutzt werden kann, betrachten wir das folgende Beispiel, bei dem Service-Telemetriedaten von den Diensten per Streaming an Apache Kafka gesendet werden(Abbildung 12-4).

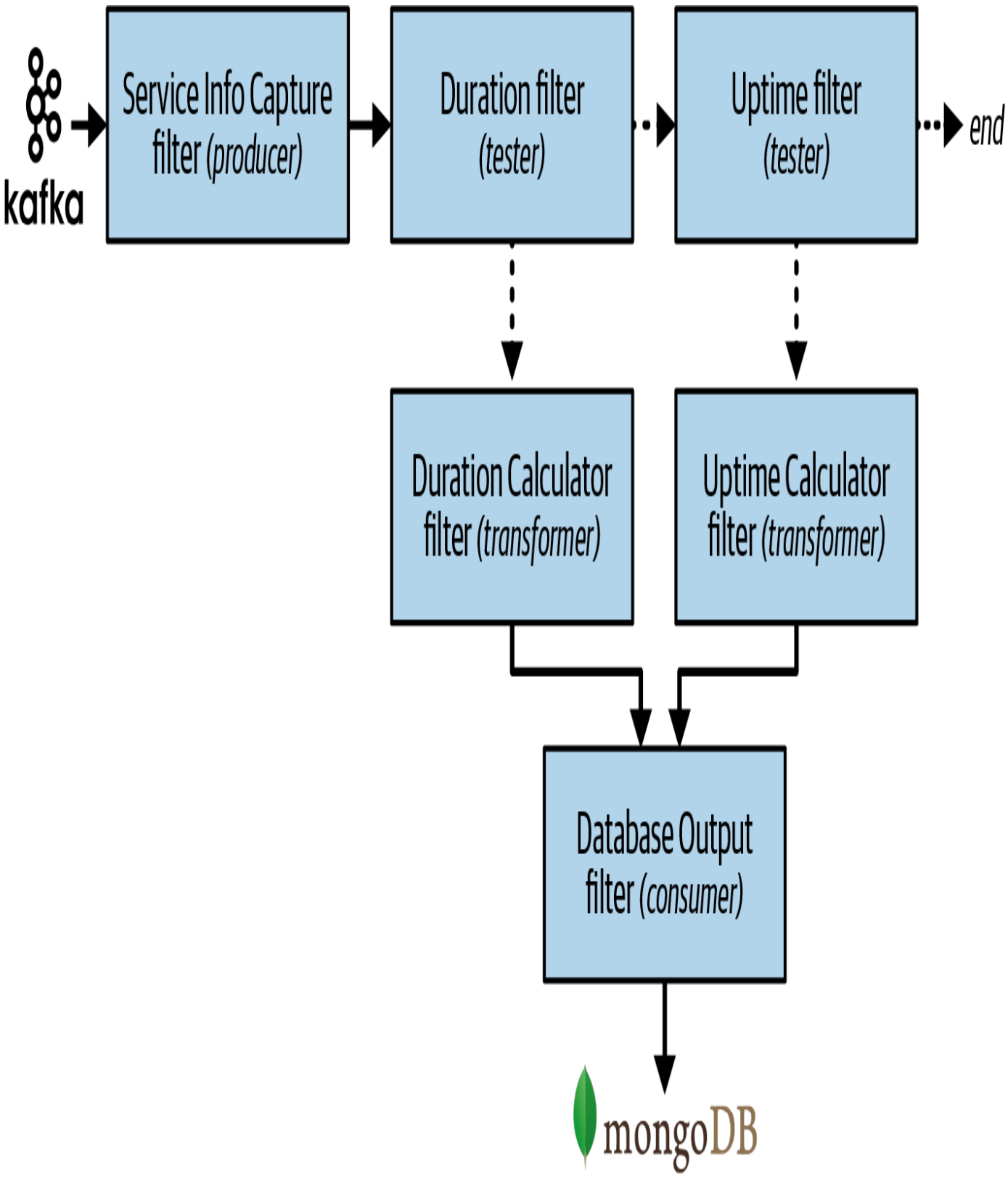

Abbildung 12-4. Beispiel für eine Pipeline-Architektur

Das in Abbildung 12-4 dargestellte System verwendet die Pipeline-Architektur, um verschiedene Arten von Daten zu verarbeiten, die an Kafka gestreamt werden. Der Filter "Service Info Capture" (Erzeugerfilter) abonniert das Kafka-Topic und empfängt Service-Informationen. Anschließend sendet er diese erfassten Daten an einen Prüffilter, den Duration-Filter, um festzustellen, ob sie mit der Dauer (in Millisekunden) der Serviceanfrage zusammenhängen.

Beachte die Trennung der Anliegen zwischen den Filtern: Der Filter Service Info Capture befasst sich nur mit der Verbindung zu einem Kafka-Topic und dem Empfang von Streaming-Daten, während der Filter Duration nur die Daten qualifiziert und bestimmt, ob sie an die nächste Pipe weitergeleitet werden sollen. Wenn sich die Daten auf die Dauer der Serviceanfrage beziehen, werden sie an den Transformer-Filter Duration Calculator weitergeleitet; andernfalls werden die Daten an den Filter Uptime tester weitergeleitet, um zu prüfen, ob sie sich auf die Betriebszeitmetriken beziehen. Ist dies nicht der Fall, wird die Pipeline beendet - die Daten sind für diesen speziellen Verarbeitungsablauf nicht von Interesse. Wenn sie für die Uptime-Metriken relevant sind, leitet der Uptime-Tester-Filter sie an `Uptime Calculator` weiter, das sie zur Berechnung der Uptime-Metriken verwendet. Die `Uptime Calculator` gibt die geänderten Daten an den Database Output Consumer weiter, der sie in einer MongoDB-Datenbank speichert.

Dieses Beispiel zeigt, wie erweiterbar die Pipeline-Architektur ist. In Abbildung 12-4 könnte ein neuer Filter nach dem Uptime-Filter

hinzugefügt werden, um die Daten für eine neue Metrik zu verwenden, z. B. für die Wartezeit der Datenbankverbindung.

Der Pipeline-Architekturstil ist zwar typischerweise eine monolithische Architektur, weist aber dank der Verwendung von technisch partitionierten Filtern ein gutes Maß an Modularität auf. Sie eignet sich gut für Situationen, in denen ein Workflow-basierter, schrittweiser Ansatz zur Verarbeitung von Daten innerhalb eines Systems erforderlich ist.

Bevor wir zu komplizierteren verteilten Architekturen übergehen, wollen wir im nächsten Kapitel eine weitere monolithische Architektur beschreiben, die durch die Verwendung von Plug-in-Komponenten ebenfalls ein hohes Maß an Modularität unterstützt: die *Mikrokernel-Architektur*.