HANSER

Leseprobe

Jürgen Dunkel, Andreas Eberhart, Stefan Fischer, Carsten Kleiner, Arne Koschel

Systemarchitekturen für Verteilte Anwendungen

Client-Server, Multi-Tier, SOA, Event-Driven Architectures, P2P, Grid, Web 2.0

ISBN: 978-3-446-41321-4

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-41321-4

sowie im Buchhandel.

Kapitel 6

Event-Driven Architecture (EDA)

Die im vorigen Kapitel betrachteten Service-orientierten Architekturen (SOA) bieten ein Konzept, um Software auf Basis der von ihnen angebotenen Dienste (*Services*) zu strukturieren. Insbesondere können aus bestehenden Anwendungssystemen Services extrahiert und dann zu komplexen Geschäftsprozessen zusammengesetzt werden.

Dieser Architekturansatz ist sehr gut geeignet, wenn die zu unterstützenden Prozesse Ablauf-orientiert sind, d.h. durch einzelne Arbeitsschritte, Abfragen, Schleifen usw. beschrieben werden können. Eine solche Ablauf-orientierte Sicht auf Geschäftsprozesse stößt aber in vielen Anwendungsbereichen an ihre Grenzen.

Denn in der Realität sind viele Geschäftsprozesse ereignisgesteuert: immer mehr Detailinformationen werden elektronisch in Form von Ereignissen bereitgestellt. Dabei handelt es sich um fein-granulare Daten, die häufigen Updates unterworfen sind, und auf die in angemessener Weise möglichst in Echtzeit reagiert werden muss. Oder, wie es in einer Gartner-Studie heißt: *'the real world is mostly event driven'* [77]. Beispiele für solche ereignisgesteuerten Geschäftsprozesse gibt es viele:

- Logistische Prozesse, die eine zentrale Rolle in allen Wirtschaftsbereichen spielen, sind durch die Verarbeitung von Ereignissen bestimmt. Sie müssen bspw. auf den Eingang eines Werkstücks, das Beenden eines Fertigungsschritts oder das Auftreten eines Fehlers unmittelbar reagieren. Von besonderer Bedeutung ist dabei die Lokalisierung und Identifizierung von Waren und Gegenständen, wie sie mittels RFID-Technologie (*Radio Frequency Identification*) erfolgen kann.
- Aber auch in vielen betriebswirtschaftlichen Anwendungen sind Ereignisse von zentraler Bedeutung: die Steuerung von Arbeitsabläufen (*Workflows*) ist durch den Eingang von Bestellungen, Aufträgen, Buchungen usw. bestimmt.

Ein klassisches Beispiel ist der Wertpapierhandel: die Kauf-Entscheidung für eine bestimmte Aktie hängt von der Entwicklung von Dollarkurs, Goldpreis, Dow Jones usw. ab. Jede einzelne Kursänderung kann als ein Ereignis aufgefasst werden und muss ggf. bei der Entscheidungsfindung berücksichtigt werden.

- Im *Business Activity Monitoring* (BAM) werden die für alle Geschäftsprozesse relevanten Ereignisse in Echtzeit gesammelt und zu Daten verdichtet, um so kontinuierlich den aktuellen Status der kritischen Unternehmensprozesse bestimmen zu können. Auf Basis geeigneter Indikatoren lassen sich dann Entscheidungen fundierter treffen und die Prozesse dynamisch den Unternehmenszielen anpassen [17].

Wegen der Vielzahl der potenziell auftretenden Ereignisse und ihrer komplexen Wechselwirkungen lässt sich in den beschriebenen Szenarien kein vordefinierter Ablauf für einen Geschäftsprozess festlegen. Das prozessorientierte Konzept von SOA greift hier also nicht.

Speziell für solche Ereignis-getriebene Systeme wurde das Konzept der *Event-Driven Architecture* (EDA) von David Luckham an der Stanford University entwickelt [55]. Im Wesentlichen handelt es sich dabei um eine Softwarearchitektur, die auf die Ereignisverarbeitung ausgerichtet ist, also das Erzeugen, Entdecken und Verarbeiten einzelner Ereignisse oder ganzer Ereignisströme als zentrale Architekturkomponenten beinhaltet.

Von Gartner wurde für eine Kombination von SOA und EDA der Begriff SOA 2.0 oder *Advanced SOA* geprägt.[1] Als Hauptanwendungsgebiete werden dort u.a. Echtzeithandel in der Finanzbranche oder Verwaltung von RFID-Netzen identifiziert.

6.1 Architekturkonzept

Für die Betrachtung von Event-Driven-Architectures möchten wir nun schrittweise zwei verschiedene Ansätze unterscheiden:

- Zuerst betrachten wir Ereignis-Orientierung in Softwarearchitekturen als generelles Konzept, d.h. wir diskutieren, wie Ereignisse zur Kommunikation zwischen Komponenten verwendet werden können.
- Anschließend beschäftigen wir uns mit *Complex Event Processing* (CEP), einem Ansatz zur flexiblen, regelbasierten Verarbeitung von Ereignissen. Das CEP-Konzept ist auf Architekturen ausgelegt, deren Kontrollfluss vorwiegend Ereignis-gesteuert ist.

[1] http://www.computerwoche.de/nachrichten/577497/

6.1.1 Ereignis-orientierte Softwarearchitektur

Event-Driven-Architecture soll nicht den SOA-Architekturansatz ersetzen, sondern ihn ergänzen. Nach wie vor hat SOA seine Berechtigung und wird für viele Anwendungsgebiete die erste Wahl sein. Doch wie lässt sich EDA genauer gegen SOA abgrenzen?

Service-Orientierung. Zunächst werfen wir noch einmal einen kurzen Blick auf die Design-Prinzipien von SOA: Zentrales Konzept ist ein *Service*, der von einer Softwarekomponente (*Service Provider*) angeboten und von einem Client (*Service Requester*) aufgerufen wird. Es ist durch folgende Eigenschaften gekennzeichnet:

- Die Kommunikation zwischen Client und einem Service-Anbieter erfolgt nach dem Request/Reply-Muster. Service-Aufrufe sind in der Regel synchron, d.h. der Client wartet auf die Beendigung des Service und ggf. auf einen Rückgabewert. Damit ein Service aufgerufen werden kann, müssen sein Name, die Syntax (Parameterstrukturen) und die technischen Mechanismen (Nachrichtenformate, Transportprotokolle, Server und Ports) bekannt sein.[1] Insgesamt entsteht somit eine relative starke Kopplung des Client an den Service-Anbieter
- Services können zu komplexeren Services oder Geschäftsprozessen zusammengesetzt bzw. orchestriert werden. Das heißt, es wird ein linearer Ablauf in einer Programmier- oder einer Prozessbeschreibungssprache, bspw. in BPEL (*Business Process Execution Language)*, definiert. Eine Service-orientierte Architektur ist somit durch eine hierarchische, funktionale Zerlegung gekennzeichnet.

Abbildung 6.1 verdeutlicht den Ansatz: Die von den Komponenten angebotenen Services können aus der Geschäftsprozess-Schicht oder von Services anderer Komponenten aufgerufen werden. Die Komponenten sind dabei stark gekoppelt, weil für einen Service-Aufruf detaillierte Kenntnisse über den Service (Parameter-Struktur, Protokolle, Service-Endpunkte) erforderlich sind.

Bemerkung: Man kann in einer Service-orientierten Architektur die Kopplung von Komponenten durch verschiedene Tricks abschwächen.

- Zum einen kann man asynchrone Service-Aufrufe verwenden, damit der Client nicht durch einen Service-Aufruf blockiert wird. (Dazu wird in Java der Service in einem eigenen Thread – einem leichtgewichtigen Prozess – aufgerufen.)
- Andererseits kann man Dokumenten-orientierte Service-Parameter verwenden, d.h. die zwischen Services ausgetauschten Daten werden in einem Dokumenten-ähnlichen Austauschformat gekapselt. Dabei bietet sich XML als Programmiersprachen- und Plattform-unabhängiges Standardformat an. Da-

[1] durch WSDL beschrieben

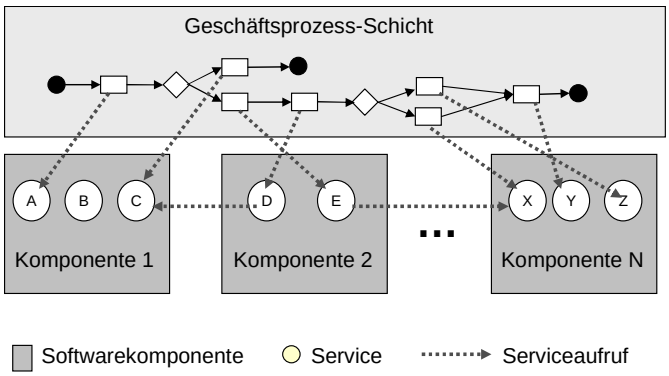

Abbildung 6.1: Service-Orientierung

bei entstehen syntaktisch stabilere Schnittstellen, weil sich auch bei Service-Modifikationen die Dokumenten-Parameter meist nicht ändern müssen. Allerdings sind solche Parameter nicht Typ-sicher. Verwendet man bspw. XML-Dokumente, so kann man diese lediglich zur Laufzeit gegen ein entsprechendes (XML-)Schema validieren.

Durch diese Maßnahmen verliert jedoch das Service-Paradigma einen Großteil seiner Attraktivität: seine Einfachheit; man ruft einen Service auf, übergibt strukturierte und wohl-definierte Parameter und wartet auf das Ergebnis.

Ereignis-Orientierung. Ereignisse sind das zentrale Konzept von Ereignis-orientierten Architekturen. Jedes Ereignis wird in Form einer Nachricht an eine Middleware[2] gesendet. Die Middleware leitet die Nachricht weiter, d.h. sie macht das Ereignis allen Softwarekomponenten bekannt, die sich für Ereignisse dieses Typs registriert haben. Dieser Ansatz impliziert folgende Eigenschaften:

- Weil der Austausch des Ereignisses über die Middleware als Mediator erfolgt, weiß die Ereignis-erzeugende Komponente nicht, wohin ein Ereignis weitergereicht und wie es weiterverarbeitet wird. Die über Ereignisse kommunizierenden Komponenten kennen sich gegenseitig nicht, sie müssen lediglich die Semantik der ausgetauschten Nachrichten verstehen. Die Kommunikation erfolgt somit asynchron und führt zu einer äußerst losen Kopplung zwischen den beteiligten Komponenten.
- Jede Komponente ist autonom und entscheidet selbst, wie sie mit den empfangenen Ereignissen umgeht. Komponenten-übergreifende Prozesse sind nicht explizit als Ganzes beschrieben, sondern die Prozess-Beschreibung ist in Form von Ereignisbehandlungs-Routinen auf verschiedene Komponenten aufge-

[2] meist eine Message-oriented Middleware (MOM)

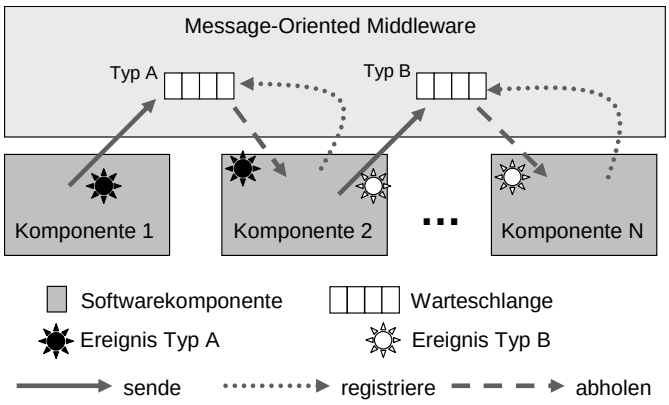

Abbildung 6.2: Ereignis-Orientierung

teilt. Das hat zur Folge, dass Komponenten-übergreifende Geschäftsprozesse nicht gut voneinander abzugrenzen und somit schlecht änderbar und wartbar sind.

Die Struktur Ereignis-gekoppelter Systeme zeigt Abbildung 6.2 genauer: Komponente 1 sendet Ereignisse von Typ *A* als Nachricht an die Message-Oriented Middleware (MOM), wo sie in einer entsprechenden Warteschlange[3] verwaltet werden. Jede Komponente, die sich bei der Middleware für Ereignisse dieses Typs registriert hat, wird ggf. über einen Ereigniseingang benachrichtigt und kann sich dann das Ereignis abholen.

Die Komponenten sind lose gekoppelt: sie kennen sich nicht gegenseitig, sondern stellen Ereignisse der Außenwelt zur Verfügung und setzen voraus, dass andere Komponenten sich dafür interessieren und die Ereignis-Semantik verstehen.
Das Konzept des Austausches von Ereignissen über Nachrichten ist schon lange im Bereich der Anwendungsintegration (EAI – *Enterprise Application Integration*, [14]) etabliert. MOM-Produkte wie IBM/MQSeries werden seit langem genutzt, um Mainframe-Systeme an andere Anwendungen anzubinden. In [29] wird die auf Ereignisaustausch basierende Anwendungsintegration detaillierter vorgestellt.

SOA versus EDA. Wie bereits erwähnt, haben sowohl Service-Orientierung wie auch Ereignis-Orientierung ihre Berechtigung.

- Service-Orientierung arbeitet nach dem Request/Reply-Paradigma und ist das Mittel der Wahl, wenn Ablauf-orientierte Prozesse implementiert werden, insbesondere, wenn synchrone Aufrufe – wie bei Benutzerabfragen – oder Trans-

[3] Eine Warteschlange kann auch als Postfach betrachtet werden, wie man es von normalen E-Mail-Systemen her kennt.

aktionen ins Spiel kommen. Klassische Geschäftsprozesse werden zumeist Service-orientiert realisiert.

- Ereignis-Orientierung wird dann eingesetzt, wenn die Entkopplung der Komponenten im Vordergrund steht. Insbesondere bei B2B-Anwendungen, wenn über Unternehmensgrenzen hinweg kommuniziert werden soll, wird ein Datenaustausch über Ereignisse durchgeführt. Ein anderes Einsatzgebiet ist die Unternehmens-interne Anwendungsintegration, wenn Komponenten keine gemeinsame technische Basis besitzen, die einen Serviceaufruf ermöglichen würde [42].
  Ereignis-Orientierung impliziert, dass Daten und Funktionalität redundant in mehreren Systemen vorgehalten werden. Dies scheint ein Nachteil zu sein, ist aber eine logische Folge der losen Kopplung, denn ein Aufheben von Redundanz verstärkt automatisch die Kopplung, weil Komponenten voneinander abhängig werden.

Meist werden Ereignis-Orientierung und Service-Orientierung gemeinsam eingesetzt. Stark abhängige Komponenten kommunizieren mittels Serviceaufrufen, aber lose gekoppelte oder Unternehmens-externe Komponenten über Ereignisse. Ereignis-Orientierung kommt immer dann ins Spiel, wenn im Geschäftsfeld eine sehr große Menge von Ereignissen auftritt, für deren Verarbeitung es keinen definierten linearen Ablauf gibt. Speziell mit dieser Situation beschäftigt sich das *Complex Event Processing*.

6.1.2 Complex Event Processing

In fast allen herkömmlichen IT-Systemen werden in bestimmten Bereichen Ereignisse ausgelöst und verarbeitet. Ereignis-Orientierung ist meist aber nicht das zentrale Konzept, auf dem die Softwarearchitektur basiert. Das von David Luckham propagierte Paradigma des *Complex Event Processing* (CEP) stellt die Verarbeitung komplexer Ereignisströme jedoch in den Mittelpunkt der Architektur [55]. CEP ist ein Konzept, um sehr große Mengen von Ereignissen flexibel verarbeiten und für den Kontrollfluss von Anwendungen nutzen zu können. Ein Beispiel ist ein System für den Aktienhandel, das Millionen von Ereignissen betrachtet, die durch alle aktuellen und historischen Kursänderungen verursacht sind.

Als wesentliche Erkenntnis betrachtet CEP die aufgetretenen Ereignisse nicht voneinander unabhängig, sondern deren Abhängigkeiten und Korrelationen. Erst die Betrachtung vieler Ereignisse über einen längeren Zeitraum gewinnt an Bedeutung und lässt entsprechende Schlussfolgerungen zu. Deshalb betrachtet man sogenannte Ereignisströme (*event stream*), d.h. die kontinuierlich eintreffenden Ereignisse sowie Ereignisse der Vergangenheit. Das Hauptziel von CEP ist es, innerhalb einer großen Ereigniswolke Muster von zusammenhängenden Ereignissen (*event patterns*) zu erkennen.

In unserem Beispiel des Aktienhandels bringt die isolierte Betrachtung eines einzelnen Ereignisses – wie die momentane Änderung eines Aktienkurses – keinen Erkenntnisgewinn. Das Ziel ist vielmehr in der riesigen Menge aller Ereignisse ein Muster zu erkennen, aus dem sich eine Kaufs- oder Verkaufsempfehlung ableiten lässt.

Im Folgenden stellen wir die einzelnen Bestandteile der Architektur vor.

Ereignisse

Ereignisse kann man als die Änderungen eines Systemzustandes verstehen. Sie lassen sich in der Terminologie der Domäne beschreiben. Beispiele für Ereignisse sind eine Flugbuchung, der Eingang eines Auftrags, eine Geldtransaktion, das Ausliefern einer Ware.

Damit Ereignisse sinnvoll genutzt und zwischen beliebigen Komponenten ausgetauscht werden können, müssen sie alle notwendigen Informationen in Form eines *Ereignis-Objektes* enthalten. Dazu gehören:

- Allgemeine Metadaten des Ereignisses, die den Ereignistyp, die Ereignisquelle, den Auftrittszeitpunkt und eine eindeutige Ereignis-ID enthalten. Diese Daten müssen für jedes Ereignis angegeben werden.
- Alle Ereignis-spezifischen Daten, die zu einer Verarbeitung benötigt werden, also bspw. die Flugdaten für ein Flugbuchungs-Ereignis oder den Geldbetrag und die Konten für eine Geld-Transaktion.

Zwischen Ereignissen bestehen Abhängigkeiten und Wechselwirkungen: bspw. gehört zu jedem Bestellungs-Ereignis ein Warenausgang-, sowie Lieferungs-Ereignis.

Fluss der Ereignisse

Einen ersten Überblick über die Funktionsweise des Complex Event Processing erhält man, wenn man den Fluss der Ereignisse verfolgt: Im CEP durchlaufen die Ereignisse verschiedene Stationen, die Abbildung 6.3 darstellt.

- Ereignis-Quellen (*Event Sources*): Ereignisse werden durch *Quellen* erzeugt, bspw. durch einen Service-Aufruf, eine Benutzerinteraktion, eine eingehende E-Mail oder ein RFID-Ereignis. Die Ereignisse sollten idealerweise in einem einheitlichen Standard-Format verschickt werden.
- Ereignis-Kanal (*Event-Channel*): Der Ereignis-Kanal bietet die technische Infrastruktur für den Transport von Ereignissen von den Quellen zu den Ereignisverarbeitenden Komponenten, insbesondere zur zentralen CEP (*Complex Event Processing*)-Komponente. Technisch wird ein Ereignis-Kanal meist durch die bereits erwähnte Message-Oriented Middleware (MOM) realisiert, die es ermöglicht, Ereignisse in Form von Nachrichten zu verschicken.

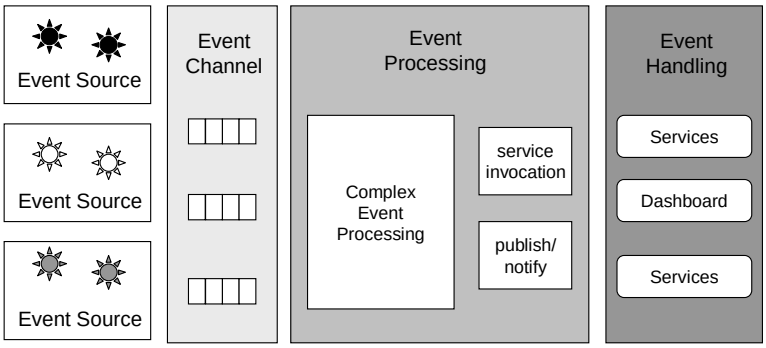

Abbildung 6.3: Fluss der Ereignisse

- Ereignis-Verarbeitung (*Event Processing*): Zur Ereignis-Verarbeitung wird der Strom der eingehenden Ereignisse mit den Techniken des Complex Event Processing analysiert (s.u.). Dabei werden anhand von Regeln Ereignismuster erkannt, auf die entsprechend reagiert werden kann. Beispielsweise soll beim Wertpapierhandel eine bestimmte Kurssituation erkannt werden, die ein Kaufs- oder Verkaufssignal darstellt.

  Es gibt verschiedene Arten, wie auf vorliegende Ereignismuster reagiert werden kann: in der Regel wird der Service einer Anwendung aufgerufen. In unserem Beispiel wird in den Backend-Systemen ein Dienst zum Kauf oder Verkauf einer Aktie ausgeführt. Zum andern können aufgetretene Ereignisse auch einfach veröffentlicht (*publish*) oder anderen Komponenten bzw. interessierten Personen bekanntgegeben (*notify*) werden.
- Ereignis-Behandlung (*Event Handling*): Die eigentliche Behandlung der Ereignisse erfolgt nicht in der Event-Processing-Komponente, sondern in den produktiven Anwendungen, den Backend-Systemen. Dies sind in der Regel Geschäftsprozesse oder Services, die von Ereignissen angestoßen werden (*event-triggered*). Alternativ werden Ereignisse einfach in der graphischen Benutzungsschnittstelle angezeigt, bzw. in Dashboards.

Komponenten zur Implementierung der CEP-Architektur

Der zentrale Schritt erfolgt in der Ereignis-Verarbeitung, die wir uns etwas genauer im Folgenden ansehen. Abbildung 6.4 zeigt die grundlegenden Bausteine eines Complex-Event-Processing-Systems; sie verfeinert den Block „Event Processing“ in Abbildung 6.3.

- **Event Specification:** Damit Ereignisse automatisch verarbeitet werden können, müssen sie präzise definiert werden. Nur mit einem definierten Ereignisformat können Ereignisse zwischen beliebigen Anwendungen und Software-

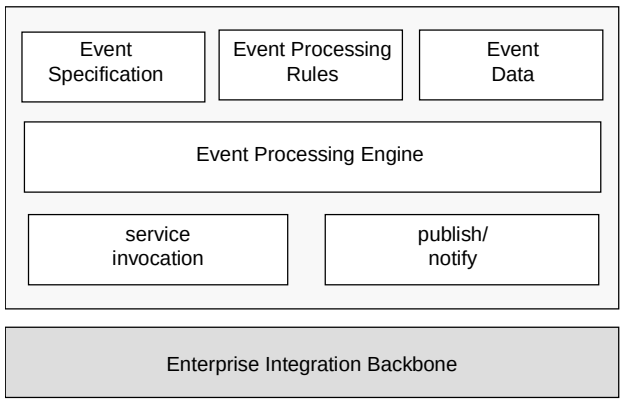

Abbildung 6.4: CEP-Architektur

Komponenten ausgetauscht werden. Die Ereignis-Spezifikation beschreibt die möglichen Ereignis-Typen mit ihren erforderlichen Daten sowie deren Abhängigkeiten und Wechselwirkungen. Oft kann eine Ereignis-Hierarchie definiert werden: Von einem abstrakten Ereignistyp, der die allgemeinen Metadaten (Typ, ID, Zeitstempel usw.) enthält, sind die konkreten Domänenspezifischen Ereignisse abgeleitet.

Mögliche technische Formate sind Schlüssel-Werte-Paare, POJOs (*Plain Old Java Objects*) in einer reinen Java-Systemlandschaft, oder XML in einer heterogenen Systemlandschaft. Die Syntax der Ereignisse, d.h. die zulässigen Elemente und Strukturen, müssen formal definiert werden, bspw. mit XML Schema für ein definiertes XML-Austauschformat.

Darüber hinaus sollte die Semantik der Ereignisse genau festgelegt werden, damit alle beteiligten Komponenten die Ereignisse korrekt interpretieren können. Die Semantik von Ereignissen, insbesondere eine genauere Klassifizierung und Beschreibung von Zusammenhängen, kann mit Ontologien definiert werden.[4] Insgesamt kann die Ereignis-Spezifikation als Metamodell für die betrachteten Ereignisse aufgefasst werden.

- **Event Data:** Die Ereignisdaten (*event data*) beschreiben die in der realen Welt tatsächlich aufgetretenen Ereignisse und sind Instanzen der in der Event Specification definierten Ereignistypen.
- **Event Processing Rules**: Auf Basis der Ereignis-Spezifikation werden Regeln zur Verarbeitung von Ereignissen, beispielsweise eine Transformation in andere Ereignis-Formate, das Filtern und Aggregieren von Ereignissen sowie das Erzeugen neuer Ereignisse, definiert. Eine *Ereignisregel* definiert Aktionen, die ausgeführt werden, wenn ein vordefiniertes Muster in einem Ereignisstrom

[4] Dabei können Technologien des *Semantic Web* wie RDF oder OWL genutzt werden.

erkannt wird. Das Ausführen einer Regel erfordert somit zunächst eine solche Mustererkennung, nämlich den Abgleich des Ereignisstroms mit einem bestimmten Muster. Ereignisregeln bestehen aus zwei Teilen:

- Ein Trigger, der aus einem oder mehreren miteinander verknüpften Mustern besteht.
- Eine Aktion, die ausgeführt wird, wenn der Trigger ausgelöst wird. Oft spricht man auch vom „Feuern" einer Regel.

Zur Beschreibung eines Musters werden Beziehungen zwischen Ereignissen ausgedrückt. Dabei lassen sich grundsätzlich folgende Beziehungstypen unterscheiden:

- *Zeitliche Zusammenhänge*: Ereignis A ereignet sich vor Ereignis B.
- *Ursächliche Zusammenhänge*: Ereignis A verursacht Ereignis B, d.h. B ist von A abhängig.
- *Aggregation von Ereignissen*: Ereignis A ist eine Gruppierung aller Ereignisse B_i, was bedeutet, dass B_i Teile von A sind.

In der Praxis werden Ereignismuster und -regeln in dafür vorgesehene Sprachen EPL (*Event Pattern Languages*) formuliert [99], [75]. Ein Beispiel für eine einfache EPL ist die Rapide-EPL, die im Buch von Luckham [55] vorgestellt wird. Eine andere viel eingesetzte Open Source EPL ist Esper [22], die wir später noch etwas genauer betrachten werden. Beispielsweise können in Rapide-EPL auf Basis der Operatoren *and*, *or* und $\rightarrow$ Ereignismuster wie folgt definiert werden:

- A *and* B *and* C: Die Regel feuert, wenn die Ereignisse A, B, und C aufgetreten sind.
- A $\rightarrow$ (B *or* C): Die Regel feuert, wenn zuerst das Ereignis A und danach das Ereignis B oder das Ereignis C aufgetreten sind.

Zurzeit gibt es jedoch noch keinen Standard zur formalen Beschreibung von Ereignissen oder für Regelsprachen, sodass diese Aufgabe in jedem Anwendungskontext neu bewältigt werden muss.

- **Event Processing Engine:** Den Kern der eigentlichen Ereignis-Verarbeitung bildet eine Regelmaschine (*event processing engine*), die die Ereignisdaten lädt und die darauf definierten Ereignisregeln ausführt.

  Weil Pattern Matching auch in der Vergangenheit stattgefundene Ereignisse berücksichtigen, müssen Ereignisdaten permanent gespeichert werden. Normalerweise lassen sich nicht alle Ereignisse abspeichern, weil deren Anzahl riesengroß werden kann – z.B. alle Aktienkursänderungen eines vergangenen Jahres.

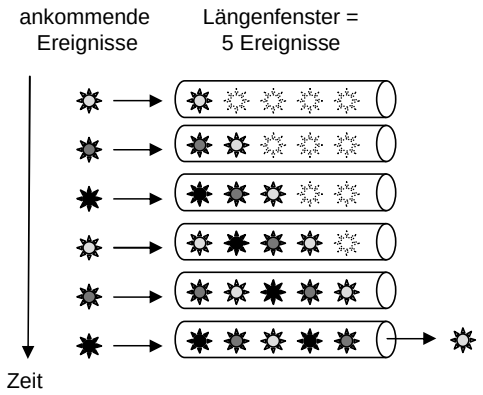

Abbildung 6.5: Längenfenster zum Abspeichern von Ereignissen

Als praktikablen Ansatz speichern die meisten Event Processing Engines nur diejenigen Ereignisse, die in einem bestimmten Zeit- oder Längenfenster (*sliding window*) stattgefunden haben. In einem Zeitfenster hat jedes Ereignis eine bestimmte Gültigkeitsdauer (*lease time*). Wenn diese abgelaufen ist, wird es aus dem Speicher entfernt. Abbildung 6.5 zeigt ein Längenfenster mit einem Puffer der Länge 5. Kommt bei vollem Puffer ein neues Ereignis an, so wird das älteste Ereignis gemäß einer FIFO (*First-In-First-Out*)-Strategie aus dem Puffer geschoben.

Es gibt eine ganze Reihe kommerzieller Anbieter von Event Processing Engines,[5] darüber hinaus einige Open-Source-Produkte, bspw. Esper [22].

- **Enterprise Integration Backbone**: Der EIB liegt außerhalb der eigentlichen Ereignis-Verarbeitung und gehört deshalb nicht mehr zur CEP-Komponente. Er bietet lediglich die Infrastruktur, um die CEP-Komponente an die vorhandenen unternehmensweiten Anwendungssysteme (bspw. Informationssysteme oder Legacy-Systeme) anzubinden. Dabei erfolgt die Kommunikation bidirektional:
  - Einerseits können Geschäftsprozesse durch Ereignisse gesteuert werden, d.h. das Auftreten eines Ereignisses stößt mithilfe des EIB den Aufruf eines Services an (*event-driven actions*). [6]
  - Umgekehrt kann eine Komponente oder ein Service der Anwendungssysteme ein neues Ereignis generieren. Ein solches Ereignis kann unterschiedliche Bedeutung besitzen und eine bestimmte Situation im Unternehmen oder ein aufgetretenes Problem anzeigen. Dieses Ereignis wird dann über

[5] bspw. IBM, BEA, Tibco, Coral8, StreamBase

[6] Diese direkte Abbildung von Ereignissen auf Services wird manchmal auch *event-driven SOA* [60] genannt. Genau genommen handelt es sich hier aber um eine spezielle Form von SOA.

den Enterprise Integration Backbone an alle interessierten Parteien verteilt, die dann autonom in angemessener Weise reagieren können.

Normalerweise ist der Enterprise Integration Backbone eine Standard-Middleware, die als Mediator [30] dient, um Ereignisse zwischen Komponenten auszutauschen. In klassischen Anwendungslandschaften mit Legacy-Systemen ist dies meist eine MOM (Message-Oriented Middleware), in moderneren Service-orientierten Architekturen ein ESB (Enterprise Service Bus).

Muster der Ereignis-Verarbeitung

Nachdem wir die einzelnen Bausteine für die Implementierung einer Complex-Event-Processing-Komponente kennengelernt haben, wollen wir nun etwas genauer auf die möglichen Verarbeitungsschritte in einer CEP-Komponente schauen, die von sogenannten Agenten (*event processing agents* – EPAs) durchgeführt werden.

- *Event Filtering*: Event Processing Agents beobachten den Strom der aufgetretenen Ereignisse, um Ereignismuster zu finden. Damit sie mit einer ggf. riesigen Menge von Ereignissen umgehen können, werden die für eine Fragestellung bedeutenden Ereignisse herausgefiltert (siehe Abbildung 6.6). Dabei könnten bspw. nur Ereignisse eines bestimmten Typs betrachtet werden (z.B. ein Ereignis vom Typ „Auftragseingang“) oder Ereignisse mit einem bestimmten Inhalt (alle Auftragseingänge mit einem Wert $\geq$ 1000 Euro). Das Filtern reduziert die Menge der betrachteten Ereignisse und führt so zu einer effizienteren Verarbeitung.
- *Content-based Routing*: Der Ereignistyp oder -inhalt entscheidet über das genaue Ziel, an das ein Ereignis geschickt wird, bspw. die Warteschlange eines Ereignis-Kanals. So könnte ein Auftragseingangs-Ereignis an das Warenlager geschickt werden, das für diesen Auftrag zuständig ist.
- *Event Splitting and Event Aggregation*: Manchmal müssen komplexe Ereignisse in einzelne Ereignisse aufgebrochen werden; bspw. wird das Ereignis einer umfangreichen Bestellung zur Weiterverarbeitung in mehrere Ereignisse für jeweils eine Bestellposition aufgeteilt. Umgekehrt können fein-granulare Ereignisse zu einem Ereignis zusammengefasst werden, um so die Gesamtzahl der Ereignisse zu reduzieren; siehe Abbildung 6.7 (links).

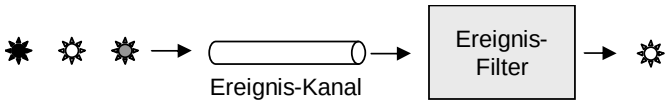

Abbildung 6.6: Filtern von Ereignissen mithilfe von EPAs

Abbildung 6.7: Ereignis-Aggregation und Content-Enrichment

- *Event Transformation*: Darüber hinaus können EPAs Ereignisse für die weitere Verarbeitung modifizieren. Die einfachste Art einer Transformation besteht darin, die Ereignisbeschreibung in ein anderes Format zu übersetzen (*event translation*). Dies ist dann erforderlich, wenn die Ereignis-Quellen keine Ereignis-Objekte im definierten Standard-Format erzeugen können. Ein fortgeschritteneres Konzept ist die Anreicherung des Ereignisinhalts (*content enrichment*), d.h. dem Ereignisobjekt werden weitere Daten hinzugefügt. Dazu müssen Dienste der Applikationsschicht genutzt werden; siehe Abbildung 6.7 (rechts). Bspw. kann ein Ereignis, das eine Personalnummer enthält, nach Zugriff auf ein Informationssystem mit genaueren Personaldaten angereichert werden.
- *Synthesis of Complex Events*: Schließlich können neue, komplexe Ereignisse (*complex events*) aus einfachen Ereignissen erzeugt werden. Hierin liegt die große Mächtigkeit von CEP, denn dieser Schritt berücksichtigt die Wechselwirkungen und Abhängigkeiten der aufgetretenen Ereignisse: Aus bestimmten Mustern des vorliegenden Ereignisstroms werden Ereignisse auf einer höheren Abstraktionsebene generiert, siehe Abbildung 6.8. Ein Beispiel ist das oben genannte System zum Wertpapierhandel: einfache Kursänderungsereignisse werden auf ein komplexes, aus vielen Ereignissen abgeleitetes Kauf- oder Verkaufsereignis abgebildet.

Die genannten Verarbeitungsschritte sind nicht grundsätzlich neu, sondern werden in ähnlicher Form als sogenannte *Messaging Patterns* auch in der Anwendungsintegration eingesetzt. Eine umfassende Darstellung dieses Themas findet sich in [29].

Event Processing Networks

Die Ereignis-Verarbeitung in einer CEP-Komponente lässt sich als Abfolge einzelner Verarbeitungsschritte darstellen, die von jeweils einem EPA ausgeführt werden. Graphisch kann dies durch ein *Event Processing Network* (EPN) beschrieben werden.

Ein EPN ist ein Netzwerk, das aus mehreren EPAs gebildet wird, sodass die Ausgabe eines EPAs als Eingabe eines anderen dient. Ein einzelner EPA über-

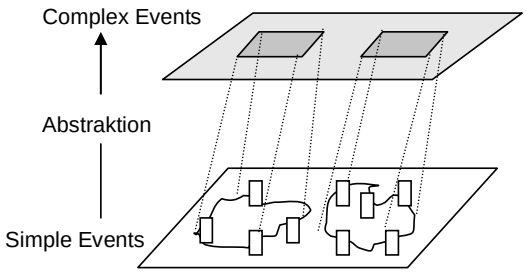

Abbildung 6.8: Complex Events

nimmt dabei nur einen oder wenige der oben dargestellten Schritte zur Ereignis-Verarbeitung (Event Filtering, Event Splitting, Event Agregation, Event Transformation, Synthesis of Complex Events, usw.). Die durch Pfeile dargestellten Verbindungen zwischen den EPAs repräsentieren den Ereignisfluss im Netzwerk.

Eine typische EPN-Struktur mit den EPAs als Netzknoten zeigt Abbildung 6.9. Um irrelevante Ereignisse herauszufiltern, werden auf der Adapter-Ebene die in den Anwendungen aufgetretenen Ereignisse in Ereignisobjekte transformiert und an die Filter-EPAs übergeben. Die Adapter stellen somit die Verbindung zu den Ereignisquellen her. Die Filter-EPAs sind in der Regel einfach und enthalten keine rechenintensiven Regeln. Sie filtern lediglich die für den Kontext der Applikation unbedeutenden Ereignisse heraus und leiten nur die relevanten Ereignisse an die Aggregation-EPAs der nächsten Ebene weiter.

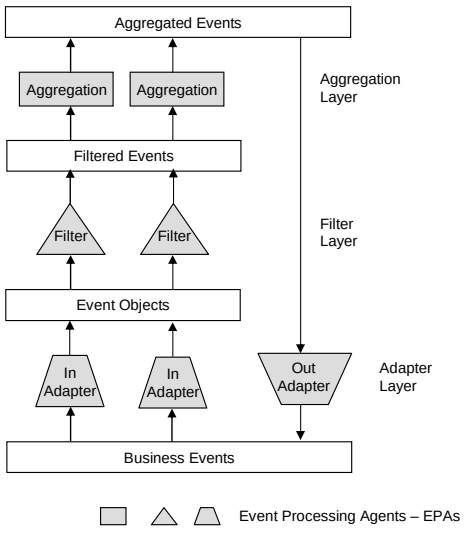

Abbildung 6.9: Event Processing Networks

Event Processing Networks veranschaulichen die komplexe Verarbeitung von Ereignisströmen. Die einzelnen EPAs sind leichtgewichtig und übernehmen nur einzelne Transformationen. Jeder EPA kann dazu eine eigene Rule Engine mit entsprechend wenigen Regeln besitzen. Die Verarbeitung der Ereignisse in kleineren Schritten macht das CEP verständlicher und somit besser wartbar. Viele und komplexe Regeln sind in einer einzelnen Rule Engine oft schwer zu verstehen und nachzuvollziehen. Wenn man das komplette EPN stattdessen in nur einer einzigen Rule Engine abbildet, so kann diese als einzelner, schwergewichtiger EPA aufgefasst werden.

6.1.3 EDA-Referenzarchitektur

Abbildung 6.10 zeigt das komplette Bild einer Referenzarchitektur mit den wesentlichen Komponenten und führt die bisher vorgestellten Architekturbausteine zusammen [18].

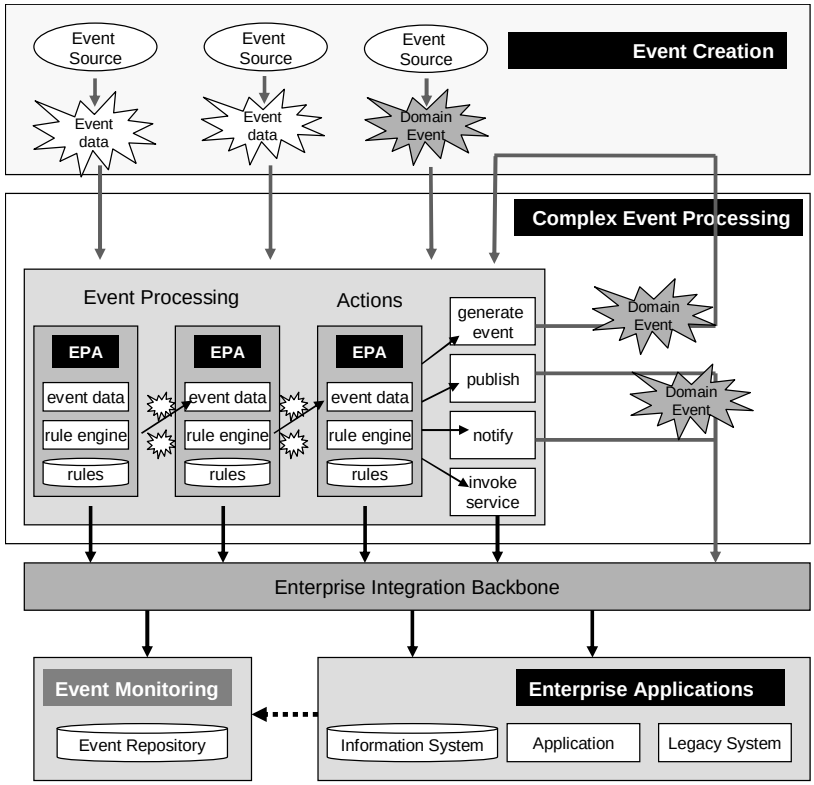

Abbildung 6.10: EDA-Referenzarchitektur

- In der obersten Schicht der Architektur (*Event Creation*) befinden sich die Ereignisquellen, in denen die Ereignisse erzeugt werden. Diese können von externen Geräten stammen, bspw. RFID-Lesegeräten, oder auch durch Zustandsänderungen des Systems verursacht werden, bspw. die Änderung eines Datensatzes oder das Eintreffen einer E-Mail. Jedes auftretende Ereignis wird ggf. schon in dieser Schicht in ein Standard-Datenformat transformiert, bspw. in einen XML-Datensatz oder ein Java-Objekt.
- Zentrale Steuerungseinheit ist die *Complex Event Processing-Komponente*, die das EPN zur Ereignisverarbeitung enthält. Das EPN besteht aus einem oder mehreren Ereignis-verarbeitenden Agenten (EPAs), die die auftretenden Ereignisse transformieren, modifizieren und zusammenführen. Dazu nutzen sie jeweils eine eigene Rule Engine mit einer eigenen Rule Base.

  Welche Verantwortung und Aufgabenbereiche man den einzelnen Agenten zubilligt, entspricht der Abwägung von Kohäsion und loser Kopplung in Softwarearchitekturen und Agentensystemen [20], [19]: Zusammenhängende Aufgaben sollten von einem einzelnen Agenten bearbeitet werden. Übernimmt ein Agent zu viele Aufgaben, wird er zunehmend komplex und unübersichtlich. Baut man stattdessen ein komplexes Netz aus sehr vielen einfachen Agenten auf, wird die Komplexität in das Netzwerk verlagert.
- Der *Enterprise Integration Backbone* stellt eine Schnittstelle oder Fassade [30], [20] zur Verfügung, damit die CEP-Komponente auf die Anwendungssysteme zugreifen kann. Dies ist zum Beispiel erforderlich, wenn Ereignisse mit zusätzlichen Informationen angereichert werden sollen (content enrichment), bspw. um aus einer Bestellnummer weitere Details zu erhalten. Die Kommunikation ist unidirektional, d.h. die Anwendungssysteme kennen die CEP-Komponente nicht.
- Die *Event-Monitoring*-Komponente ist optional und enthält ein Repository zum Speichern der Events. Die Speicherdauer von Ereignissen hängt von ihrem jeweiligen Typ ab. In der Regel werden Ereignisse höherer Komplexität länger gespeichert als einfache atomare Ereignisse. Die Event-Monitoring-Komponente ist die einzige EDA-Komponente, die für die Anwendungssysteme sichtbar ist. Sie kann von den Anwendungssystemen verwendet werden, um Ereignisse anzuzeigen, auszuwerten oder in einer Business-Activity-Monitoring (BAM)-Komponente zu visualisieren.

6.1.4 Vorgehen bei der Entwicklung von EDA-Anwendungen

Der Entwicklungsprozess von EDA-Anwendungen muss die Ereignisverarbeitung entsprechend berücksichtigen. Neben den üblichen Aufgaben des Software-Entwicklungsprozesses müssen generell die folgenden EDA-spezifischen Schritte durchgeführt werden:

1. *Ereignisquellen und -typen*: Zunächst müssen die Ereignisquellen und die von ihnen erzeugten Ereignistypen identifiziert werden. Diese Ereignisse sind einfach (*simple events*) und beschreiben eine Zuständsänderung oder das Eintreten einer bestimmten Tatsache. In der Regel wird man eine Ereignishierarchie definieren, die durch Spezialisierung von Ereignissen entsteht. Zum Beispiel ist der Kauf eines Produktes mit Kreditkarte eine spezielle Ausprägung eines normalen Kauf-Ereignisses.
2. *Beziehungen zwischen Ereignissen*: Anschließend werden die Abhängigkeiten und Beziehungen zwischen den Ereignistypen untersucht, um daraus ggf. komplexe Ereignisse (*complex events*) abzuleiten. Komplexe Ereignisse spiegeln einen Sachverhalt in der Domäne wider, der durch eine bestimmte Zustandskonstellation charakterisiert ist. Dieser Zustand ist eingetreten, wenn ein bestimmtes Muster von Ereignissen aufgetreten ist.
3. *Ereignisformat*: Es wird ein einheitliches Ereignisformat definiert, in das ggf. die ursprünglichen Ereignisse transformiert werden. Dazu werden zunächst die allgemeinen[7] und Ereignistyp-spezifischen Ereignis-Metadaten festgelegt. Anschließend wird in Abhängigkeit von der eingesetzten Rule Engine das Darstellungsformat festgelegt, bspw. XML oder Java-Objekte.
4. *Muster und Regeln*: Mithilfe einer EPL werden die relevanten Ereignismuster und -regeln spezifiziert. Dabei werden ggf. einfache Ereignisse transformiert, mit Inhalten angereichert oder zu komplexen Ereignissen zusammengeführt.
5. *Ereignis-gesteuerte Aktionen*: Schließlich wird die Ereignisbehandlung in den Anwendungssystemen realisiert. Die beim Auftreten eines Musters ausgeführten Aktionen werden implementiert und von der CEP-Komponente angestoßen.

6.1.5 Aktueller Entwicklungsstand

Insgesamt führt Event-Driven-Architecture in Verbindung mit Complex-Event-Processing zu einem neuen Paradigma für Softwarearchitekturen. Im Gegensatz zu klassischen Service-orientierten Architekturen (SOA) ist der Kontrollfluss nicht vordefiniert und imperativ als Folge von Befehlen beschrieben, sondern reagiert dynamisch und regelbasiert auf auftretende Ereignis-Muster. Viele Anwendungsbereiche – insbesondere in Logistik, Medizin und Finanzwesen – sind Ereignisgetrieben und können durch EDA-basierte Softwaresysteme besser und flexibler abgebildet werden. In diesen Bereichen kann schon heute EDA klassische IT-Systeme ablösen oder zumindest ergänzen.

Allerdings stehen dem schnellen Erfolg von EDA einige Faktoren im Wege:

- Es fehlen allgemeine Standards, insbesondere Event Processing Languages zur Beschreibung von Ereignismustern und -regeln. Die vorhandenen Rule Engi-

[7] Ereignistyp, ID, Zeitstempel,...

nes verwenden proprietäre EPLs und APIs, die eine unternehmensweite Entscheidung für ein Produkt erschweren. Zurzeit sind allerdings einige Standards auf den Weg gebracht, die zwar nicht unmittelbar auf EDA ausgerichtet sind, aber zumindest eine Standardisierung für Regelsprachen vorantreiben: die Java Rule Engine API [39] und die Regelsprache RuleML [89].

- Darüber hinaus gibt es noch keine etablierten Methoden für die Entwicklung von EDA-Systemen. Insbesondere Guidelines, Entwurfsmuster oder wiederverwendbare Event Driven Agents existieren zurzeit noch nicht.
- Aktuell gibt es noch keine etablierten Produkte am CEP-Markt. Darüber hinaus ist die Werkzeugunterstützung noch nicht annähernd so weitgehend wie bei klassischen Architekturen. Beispielsweise gibt es kaum Hilfen beim Debugging von Regelsystemen.
- Es gibt noch wenig Erfahrungen mit wirklich komplexen EDA-Systemen. Insbesondere die Entwicklung komplexer Regelsysteme ist für viele Unternehmen völliges Neuland.

Insgesamt bleiben also noch viele Fragen und Probleme offen. Weil aber der mögliche Einsatzbereich immer wichtiger wird und mittlerweile auch führende Hersteller auf den EDA-Zug aufspringen, ist zu erwarten, dass sich EDA-Architekturen auf Dauer durchsetzen werden. Sobald sehr viele korrelierende Ereignisse verarbeitet werden, bietet EDA den zurzeit besten Architekturansatz.

6.2 Realisierungsplattformen

Bisher wurde die CEP-Arbeitsweise lediglich konzeptionell dargestellt. Doch für die Verarbeitung von Ereignissen, insbesondere für das Pattern Matching, benötigt man eine entsprechende Realisierungsplattform.

Die ersten EDA-Plattformen entstanden an Hochschulen und Forschungseinrichtungen, bspw. an den amerikanischen Hochschulen Stanford und Berkley, und wurden meist nur in einzelnen Projekten eingesetzt. Mittlerweile gibt es aber einige kommerzielle Produkte, die meist von kleinen Spezialanbietern (wie Coral8, GemStone, StreamBase) stammen. Und inzwischen haben sich auch die etablierten Middleware-Anbieter (wie IBM, TIBCO, BEA, Siemens, HP) des Themas angenommen.

Leider existieren zurzeit noch keine Standards, sodass die Produkte recht unterschiedlich sind. Im Wesentlichen stellen alle Produkte jedoch die folgenden Komponenten zur Verfügung:

- Es wird eine proprietäre *Event Processing Language (EPL)* definiert, mit deren Hilfe sich Ereignisse, Ereignismuster und Regeln beschreiben lassen.

  Die meisten EPLs sind an SQL angelehnt und nutzen aus SQL bekannte Konstrukte wie `SELECT`, `WHERE` und `INSERT INTO`, die zum Filtern, Aggregieren

und Erzeugen komplexer Ereignisse benötigt werden. Statt Tabellen werden jedoch Eingabeströme sowie *Views* verwendet. Dazu lassen sich durch Zeitfenster (*Time Window*) und Längenfenster (*Length Window*) die Menge der in einem Ereignisstrom betrachteten Ereignisse einschränken. Die in EPL formulierten Abfragen werden kontinuierlich – nicht einmalig, wie bei einer Datenbank – ausgeführt und deshalb auch *continuous queries* genannt [4].

Zur Definition von Ereignismustern können in der `WHERE`-Klausel Ereignistypen, Ereignisattribute und temporale Aspekte, also die zeitliche Abfolge von Ereignissen, betrachtet werden. Insbesonder müssen sich bestimmte Ereignisse auch ausschließen lassen, d.h. man spezifiziert, dass bestimmte Ereignisse in einem definierten Zeitraum nicht stattgefunden haben.

Die Anforderungen an Sprachen zur Definition von Ereignissen werden bspw. in [99] ausführlich diskutiert. Ein konkretes Beispiel stellen wir im folgenden Abschnitt vor.

- Darüber hinaus steht eine *Rule Engine* zur Verfügung, um den Strom der eingehenden Ereignisse gemäß der in der EPL definierten Ereignismuster und -regeln zu verarbeiten. Die Rule Engine ist speziell auf die schnelle Verarbeitung großer Ereignisströme ausgerichtet. Dabei werden zwei Entwurfsziele angestrebt: zum einen muss ein hoher Durchsatz erreicht werden, d.h. bis zu 100000 Ereignisse sollen pro Sekunde verarbeitet werden, zum andern sollen die Latenzzeiten für die Reaktion auf Ereignisse möglichst kurz sein [23].

Im Folgenden möchten wir eine praktische Umsetzung des EDA-Ansatzes anhand der Open-Source-Plattform *Esper* vorstellen [22]. Esper ist die zurzeit einzige Open Source Rule Engine, die speziell auf die Verarbeitung von Ereignissen ausgerichtet ist. Die Plattform steht als Java-Implementierung oder unter dem Namen *Nesper* auch für .NET zur Verfügung.

6.3 Code-Beispiele

Die erforderlichen Esper-Elemente wollen wir nun schrittweise anhand eines Beispiels vorstellen.

Ereignisse

Zunächst müssen die im Anwendungsszenario relevanten Ereignisse definiert werden. Um die Konzepte zu verdeutlichen, betrachten wir im Folgenden ein Beispiel aus dem Bankenumfeld. Abbildung 6.11 zeigt die möglichen Ereignisse in Form einer Klassenhierarchie.

- Allgemeine Ereignisse werden in der abstrakten Klasse `AbstractEvent` modelliert, die die allgemeinen Metadaten aller Events beschreibt: hier den Typ, eine ID und einen Zeitstempel. Von dieser Klasse sind nun alle domänen-

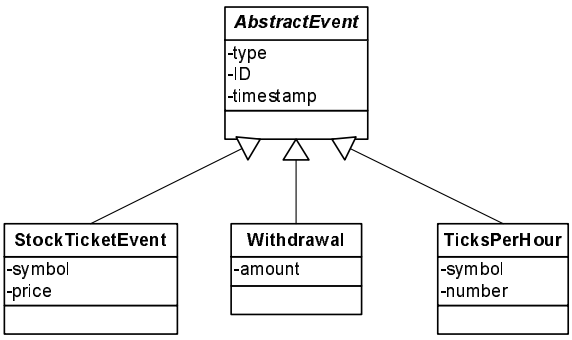

Abbildung 6.11: Ereignis-Hierarchie

spezifischen Ereignisse abgeleitet, die jeweils einen bestimmten Zustand anzeigen.

- Das Ereignis `StockTicketEvent` zeigt die Änderung eines Aktienkurses an und ist durch das Symbol der Aktie und den entsprechenden Preis gekennzeichnet.
- Das `Withdrawal`-Ereignis zeigt das Abheben eines Geldbetrags an.
- Das `TicksPerHour`-Ereignis ist ein komplexes Ereignis; es zeigt die Anzahl der Kursänderungen an, die eine Aktie innerhalb der letzten Stunde erfahren hat.

Ereignisse können in Esper nicht nur durch Java-Objekte (POJOs), sondern auch durch Schlüssel/Werte-Paare (`java.util.Map`) oder XML-Dokumente beschrieben werden.

Event Processing Language – EPL

Als EPL bietet Esper mit der *Event Processing Query Language* (EQL) eine SQL-ähnliche Abfragesprache für Ereignisströme. Mithilfe von EQL können Ereignismuster definiert und neue, komplexe Ereignisse erzeugt werden. Die folgenden Beispiele demonstrieren die Möglichkeiten von EQL:

```
select * from StockTicketEvent
```

Dieses EQL-Statement selektiert aus allen Ereignissen, die an Esper gesendet werden, diejenigen, die vom Typ `StockTicketEvent` sind. Das Statement definiert ein sehr einfaches Ereignis-Pattern, das nur den Ereignistyp betrachtet.

Für jedes EQL-Statement kann ein Listener-Objekt registriert werden, dessen Ereignisbehandlungs-Methode aufgerufen wird (s.u.), sobald das Pattern erfüllt ist – für unser Beispiel, wenn ein `StockTicketEvent` auftritt.

```
select avg(price) from StockTicketEvent.win:time(30sec)
```

Das Beispiel zeigt die *Aggregation* von `StockTicket`-Ereignissen: Es wird der Durchschnittspreis aller Aktien, die in den letzten 30 Sekunden gehandelt wurden, ausgerechnet. Dazu wird ein entsprechendes Sliding Time Window für die `StockTicket`-Ereignisse definiert: `StockTicketEvent.win:time(30sec).`

```
select * from Withdrawal.win:time(60min)
             where amount >= 1000
```

Mit diesem EQL-Statement werden alle `Withdrawal`-Ereignisse selektiert, deren Betrag (`amount`) größer als 1000 ist. Durch diese Abfrage findet somit ein *Filtern* der Ereignisse statt.

```
insert into TicksPerHour
        select symbol, count(*) as number
        from StockTicketEvent.win:time(60min)
        group by symbol}
```

Dieses Beispiel zeigt, wie ein neues, komplexes Ereignis generiert wird. Das `insert`-Statement erzeugt neue Ereignisse vom Typ `TicksPerHour` mit den Attributen `symbol` und `number`. Dabei werden von allen `StockTicket`-Ereignissen der letzten Stunde (`from StockTicket-Event.win:time(60min)`) das Kurs-Symbol und die Anzahl der Kursänderungen betrachtet (`select symbol, count(*) as number`).

Ereignis-Erzeugung und -Behandlung

Die Beispiele veranschaulichen, wie Ereignisströme nach einfachen Mustern untersucht werden können. Für ein Gesamtbild bleiben aber noch zwei Fragen offen: Wie können Ereignisse initial erzeugt werden, und wie lassen sich Ereignisse behandeln? Für beide Aufgaben gibt es in Esper ein entsprechendes Java-API:

```
StockTicketEvent evt = new StockTicketEvent(IBM, 74.50);
....getEPRuntime().sendEvent(event);
```

erzeugt ein Ereignis als Java-Objekt und sendet es an Esper.

```
EPStatement statement =
..getEPAdministrator().createEPL("select␣*␣from␣Withdrawal");
statement.addListener(listener);
```

registriert ein Listener-Objekt für ein EOL-Statement. Die Listener-Klasse muss eine Update-Methode implementieren, die von Esper aufgerufen wird, sobald das in `statement` beschriebene Pattern eintritt.