SOACON 2007
Where IT Aligns With Business

Gregor Hohpe | Google

Programming Without a Call Stack: Event-driven Architectures

www.eaipatterns.com

Who's Gregor?

- **Distributed systems, enterprise integration, service-oriented architectures**

- **MQ, MSMQ, JMS, TIBCO, BizTalk, Web Services**

- **Write code every day. Share knowledge through patterns.**

eaipatterns.com
- Patterns
- Articles
- Blog

Enterprise Integration Patterns
Addison-Wesley

Integration Patterns
Microsoft Press

Enterprise Solution Patterns
Microsoft Press

Best Software Writing I
APress

SOA Experten-wissen
dpunkt Verlag

1

Agenda

- **It's All About Coupling**

- **Events Everywhere**

- **Event-driven Architectures**

- **Developing in an EDA**

- **Case Study: Building an EDA in Java™**

2

In A Connected World It's All About Coupling
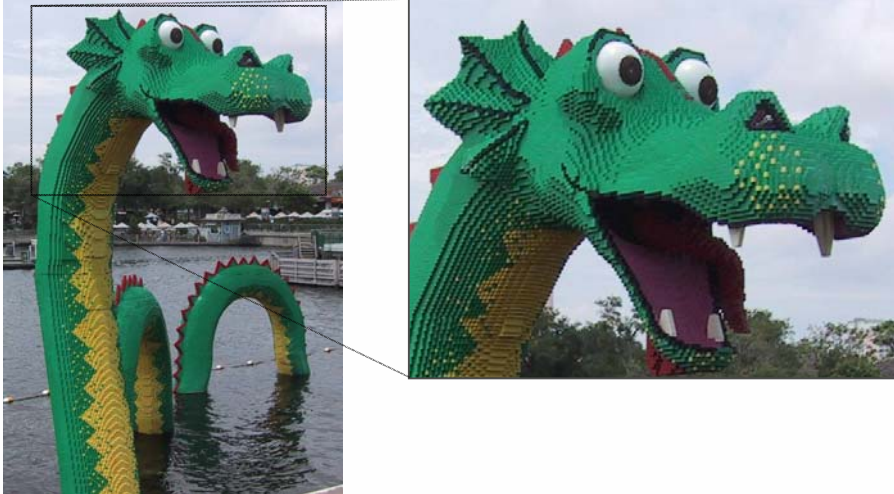

"Measure of dependency between components"

3

Dynamic Composability

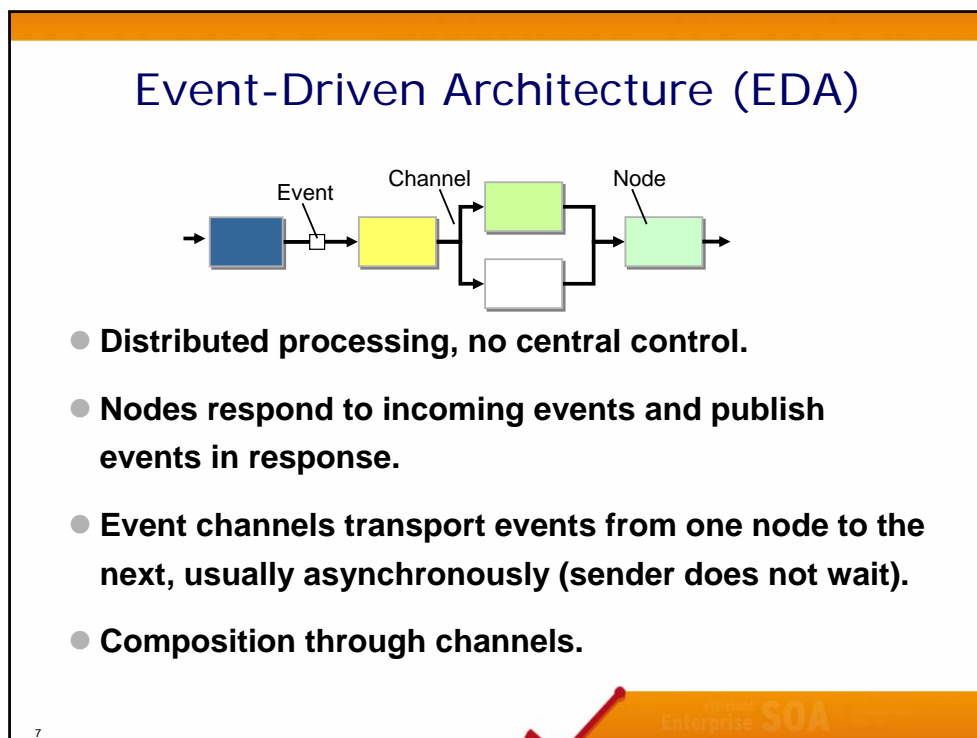**"The ability to build new things from existing pieces."**

4

Interaction Takes Center Stage

"The lines are becoming boxes now."

-- Ralf Westphal

5

The World is Full of Events

6

Event-Driven Architecture (EDA)

- **Distributed processing, no central control.**

- **Nodes respond to incoming events and publish events in response.**

- **Event channels transport events from one node to the next, usually asynchronously (sender does not wait).**

- **Composition through channels.**

7

EDA Defining Qualities

- **Timeliness**. Publish events as they occur instead of waiting for the next batch cycle.

- **Asynchrony**. The publishing system does not wait for the receiving system(s) to process the event.

- **Fine Grained**. Publish single events as opposed to large aggregated event.

- **Ontology**. A nomenclature to classify and express interest in certain groups of events.

- **Complex Event Processing**. Understanding the relationships between events, for example aggregation and causality.
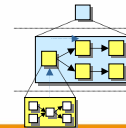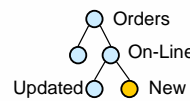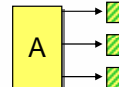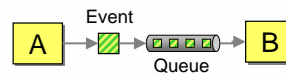
8

Composition via Channels

System A — publish → Channel X — subscribe → System B
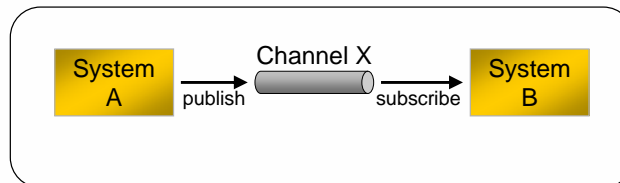
- **Nodes communicate via Channels**

- **Sender and receiver need to agree to a common channel. This is a form of coupling.**

- **Sender and receiver have to decide which channel is "right". The burden can be shifted between sender and receiver.**

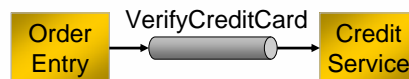- **Channel namespace provides some structure.**
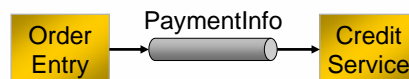
9

5

Channel Naming and Semantics
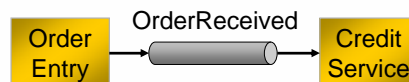
● **Target component**

| Order Entry | → CreditService → | Credit Service |

● **Action / Operation**

| Order Entry | → VerifyCreditCard → | Credit Service |

● **Document**

| Order Entry | → PaymentInfo → | Credit Service |

● **Event**

| Order Entry | → OrderReceived → | Credit Service |

Enterprise SOA

10

How Do A and B Connect?

Structured

● **Channel Name / Instance**
  - **Common in message queue systems.**
  - **Limited expressiveness.**

```
MessageQueue q = new
MessageQueue("foo");

q.Send("Msg");
```

● **Topic Hierarchy**
  - **Allows wildcard subscription**
  - **Requires mapping of topic space onto a tree. Forces prioritization.**

Orders
On-Line
Updated   New
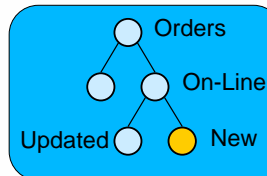
● **Content-based**
  - **Flexible, but difficult to implement efficiently in widely distributed systems.**

```
Channel.Subscribe(
"/element/foo='bar'"
);
```

Unstructured

Enterprise SOA

11

Composition Strategies

● **Implicit Composition**

```
channel = Channel.byName("orders");
channel.publish(message);
```

● **Explicit Composition**

```
A(Channel ch) { this.channel = ch; }
…
channel.publish(message);
```

A → orders → **B**

A ← Channel, <<create>> "Composer", Channel ← **B**

```
channel = Channel.byName("orders");
channel.subscribe(eventHandler);
```

```
B(Channel ch) { this.channel = ch; }
…
channel.subscribe(eventHandler);
```

12

Event Collaboration

Multiple Components work together by communicating with each other by sending events when their internal state changes. (Fowler)

Request Collaboration

Order Management — Order → RequestFor Address → Customer Management → Address — SendShipment → Shipping

Event Collaboration

Order Management — Order → Customer Management — Address Changed → Shipping

13

7

Event Collaboration

- **Adding Consumers is Side-Effect Free**
  - **Debugging / logging / sniffing**
  - **Parallel implementations**

- **Simple components, more complex interactions**

- **Robust against unstable connections**

- **Can be difficult to manage and /or debug**
  - **Need to understand relationship between events**
  - **Shift burden from design-time to run-time**

- **Components may operate on stale data**

14

Event-sourced Systems

Capture all changes to an application state as a sequence of events. (Fowler)

Persisted
State
(Snapshot)

Local
State

Domain
Objects

State
Changes

Event Bus

Event
Log

15
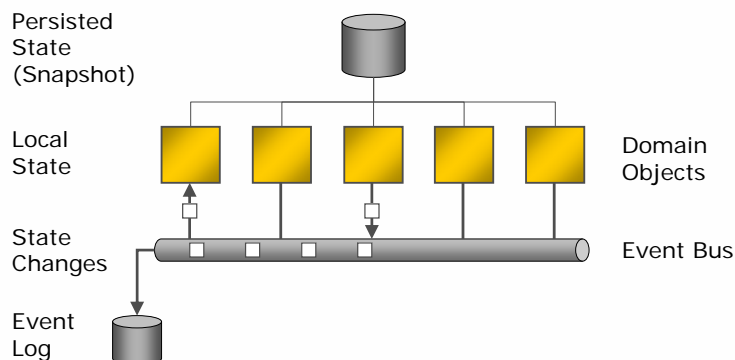
Event-sourced Systems

- **More than an event log or "temporal database"**

- **Rebuild state based on events by re-executing behavior**

- **Temporal Query**
  - **Determine application state at specific point in time**

- **Event replay**
  - **Run "what if" scenarios or corrections to past events**

- **Limitation: code changes**

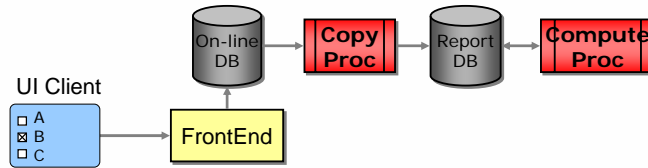16

Composite Events Processing

- **Understand causality**

- **Some events are the result a of a sequence of events**

- **CEP = Complex Event Processing**

- **Pattern matching languages can be challenging**

17
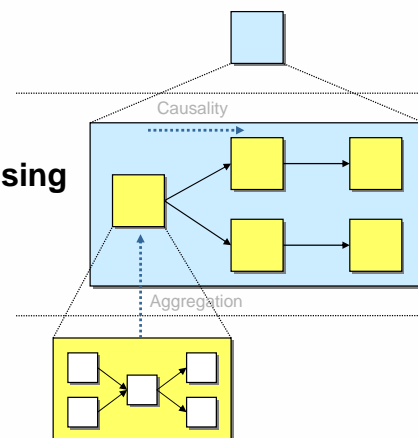
Case Study – Existing System

UI Client

On-line DB → **Copy Proc** → Report DB → **Compute Proc**
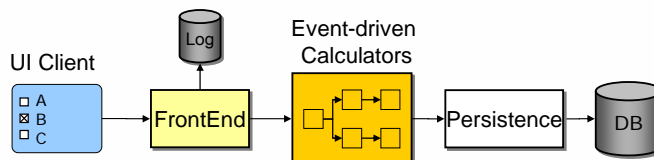
A
B
C

FrontEnd

- **Compute statistics based on responses to on-line questionnaires**
- **Responses stored in database**
- **At the end, stored procedure computes "scores" based on user responses**
  - **Load on RDBMS**
  - **Single thread, monolithic, synchronous**
  - **Poor response time at end of user session**
- **Goal: scalable, extensible architecture**

18

Case Study – New Architecture

Log

Event-driven Calculators

UI Client

A
B
C

FrontEnd → [Calculators] → Persistence → DB

- **Decompose logic into individual "calculators"**
- **Calculators precompute results as response events arrive**
- **Channels connect calculators**
- **Calculators do not updates database**
- **Persist results into database once all scores computed**
- **Pure Java (1.4) implementation**

19

10

Design Decisions

- **Point-to-Point vs. Publish-Subscribe Channels**
- **Distributed vs. Distributable**
- **Asynchronous vs. One-Way**
- **Technology Specific vs. Technology Neutral**
- **Explicit vs. Implicit Composition**
- **Channel Naming "ontology"**
  - **String match**
  - **Hierarchy (Class Hierarchy)**
  - **Content-based**
- **Automated Dispatch vs. Manual Dispatch**

20

Implementation

```
public interface Channel {

  public void send(Event event);

  public void subscribe(EventRecipient recipient,
                        Class eventClass);
}
```

Marker Interface

- **Multiple calculators subscribe to abstract Channel**
- **Channel stores subscribers by event type (hierarchy)**
- **For each incoming event, channel looks up all subscribers for the event type and its superclasses**
- **For each subscribing class, figure out the overriding onEvent method with the most specific matching argument**
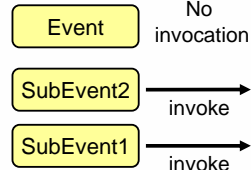
21

11

Subscription / Dispatching

Event
Hierarchy

```
Event
  △
SomeEvent
  △
SubEvent1    SubEvent2
```
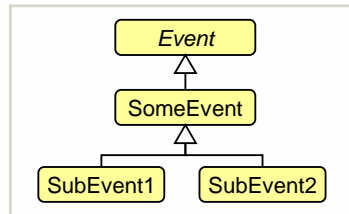
```
Class SomeEvent extends Event {}
class SubEvent1 extends SomeEvent {}
class SubEvent2 extends SomeEvent {}

class SomeSubscriber {
  public SomeSubscriber {
    channel.subscribe(SomeEvent.class);
  }

  public void onEvent (Event e) {}

  public void onEvent (SubEvent1 se) {}
}
```

Incoming
Event

```
Event        No
           invocation

SubEvent2  ──────▶
           invoke

SubEvent1  ──────▶
           invoke
```

22

Channel Implementation

```
public void send(Event event) {
  Set<EventRecipient> subscribers =
      getSubscribersForEventTypeAndItsSuperTypes(event.getClass());
  for (EventRecipient recipient : subscribers) {
    EventProcessorHelper.invokeEventHandler(event, recipient);
  }
}
```

```
Map<Class, Set<EventRecipient>> subs;

Set<EventRecipient> getSubscribersForEventTypeAndItsSuperTypes
    (Class eventClass) {
  Set<EventRecipient> allSubscribers = new HashSet<EventRecipient>();

  for (Map.Entry<Class, Set<EventRecipient>> entry : subs.entrySet()) {
    Class subscriberEventClass = entry.getKey();
    if (subscriberEventClass.isAssignableFrom(eventClass)) {
      allSubscribers.addAll(entry.getValue());
    }
  }
  return allSubscribers;
}
```

23

Channel Implementation (Cont'd)

```
boolean invokeEventHandler(Event event, EventRecipient recip)
{
  for (Class eventClass = event.getClass();
       eventClass != null;
       eventClass = eventClass.getSuperclass()) {
    Method eventHandler = recip.getClass().getMethod
           ("onEvent", new Class[]{eventClass});
    try {
      eventHandler.invoke(recip, new Object[] {event});
      return true;
    } catch (…) {…}

    if (Event.class.equals(eventClass))
      return false;
  }
  return false;
}
```

24

Channel Behaviors

```
public void testEachSubscriberReceivesMessage() {…}
public void testSubscribeTwiceReceiveOnce() {…}
public void testBaseClassSubscriberReceivesDerivedClassEvents() {…}
public void testSubscribingForNonEventTypeThrows() {…}

public void testInvokesExactlyMatchingMethodForBaseEventType() {…}
public void testInvokesExactlyMatchingMethodForEventSubType() {…}
public void testDoesNothingForOverlySpecificEventHandler() {…}
public void testInvokesMostSpecificMethodIfBothAreAvailable() {…}
```

25

Cool Things

- **Testing components in isolation**

- **Publish-subscribe enables adding rather than replacing**

- **Replay of events to recover transient state**

- **Tracing / logging trivial, almost aspect-like**

```
public class DebugCalculator extends Calculator
{
  public DebugCalculator(Channel channel) {
    super(channel);
    channel.subscribe(this, Event.class);
  }

  public void onEvent(Event event) {
    System.out.println("event = " + event);
  }
}
```

> Base class of all events

26

(Tough) Lessons Learned

- **Must keep architectural big picture in mind**
- **Integration testing more critical – less compile time validation (the price of loose coupling)**
- **Tools essential**
  - **Event logger**
  - **Dependency visualization ("reverse MDA")**
- **Shared state not always avoidable. Can lead to hidden dependencies**
- **Make minimum necessary assumptions about sequence of events**
- **Loosely coupled systems harder to diagnose**

27

Side-By-Side

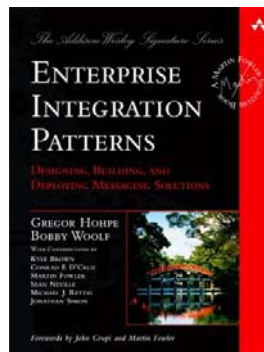| Call Stack	Event-Driven
Top-down	Bottom-up
Design-time composition	Run-time composition
Sequential	Parallel
Synchronous	Asynchronous
Predictive	Reactive
Transactional (Pessimistic)	Compensation / Retry (Optimistic)
Centralized state	Distributed state
Error handling simple	Error handling more complex
	Diagnostics more complex

28

29

For More Information

- **Enterprise Integration Patterns**
  - **Addison-Wesley, 0-321-20068-3**

- **www.eaipatterns.com**
  - **Article: Programming without a Call Stack**
  - **Blog ("Ramblings")**

- **http://www.martinfowler.com/eaaDev/EventCollaboration.html**

- **http://www.martinfowler.com/eaaDev/EventSourcing.html**

30