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.

- Enterprise Integration Patterns
  - Addison-Wesley
- Integration Patterns
  - Microsoft Press
- Enterprise Solution Patterns
  - Microsoft Press
- Best Software Writing I
  - APress
- SOA Expertenwissen
  - dpunkt Verlag
Agenda

- It's All About Coupling
- Events Everywhere
- Event-driven Architectures
- Developing in an EDA
- Case Study: Building an EDA in Java™

In A Connected World It's All About Coupling

"Measure of dependency between components"
Dynamic Composability

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

Interaction Takes Center Stage

"The lines are becoming boxes now."

-- Ralf Westphal
The World is Full of Events

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.
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.

Composition via Channels

- **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.**
Channel Naming and Semantics

- **Target component**
  - Order Entry
  - CreditService

- **Action / Operation**
  - Order Entry
  - VerifyCreditCard

- **Document**
  - Order Entry
  - PaymentInfo

- **Event**
  - Order Entry
  - OrderReceived

How Do A and B Connect?

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

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

- **Content-based**
  - Flexible, but difficult to implement efficiently in widely distributed systems.

```
MessageQueue q = new MessageQueue("foo");
q.Send("Msg");
```

```
Orders
  On-Line
  Updated
  New
```

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

- Implicit Composition
  ```java
  channel = Channel.byName("orders");
  channel.publish(message);
  channel = Channel.byName("orders");
  channel.subscribe(eventHandler);
  ```

- Explicit Composition
  ```java
  A(channel ch) { this.channel = ch; }
  ... channel.publish(message);
  
  B(channel ch) { this.channel = ch; }
  ... channel.subscribe(eventHandler);
  ```

Event Collaboration

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

(Fowler)
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

Event-sourced Systems

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

Composite Events Processing

- Understand causality
- Some events are the result of a sequence of events
- CEP = Complex Event Processing
- Pattern matching languages can be challenging
Case Study – Existing System

- 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

Case Study – New Architecture

- 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
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

Implementation

```java
public interface Channel {
    public void send(Event event);
    public void subscribe(EventRecipient recipient, Class eventClass);
}
```

- 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
Subscription / Dispatching

Event Hierarchy

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) {}
}

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;
}
Channel Implementation (Cont'd)

```java
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;
}
```

Channel Behaviors

```java
public void testEachSubscriberReceivesMessage() {...}
public void testSubscribeTwiceReceiveOnce() {...}
public void testBaseClassSubscriberReceivesDerivedClassEvents() {...}
public void testSubscribingForNonEventTypeThrows() {...}
public void testInvokesExactlyMatchingMethodForBaseEventType() {...}
public void testInvokesExactlyMatchingMethodForEventSubType() {...}
public void testDoesNothingForOverlySpecificEventHandler() {...}
public void testInvokesMostSpecificMethodIfBothAreAvailable() {...}
```
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

```java
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);
    }
}
```

(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
Side-By-Side

**Call Stack**
- Top-down
- Design-time composition
- Sequential
- Synchronous
- Predictive
- Transactional (Pessimistic)
- Centralized state
- Error handling simple

**Event-Driven**
- Bottom-up
- Run-time composition
- Parallel
- Asynchronous
- Reactive
- Compensation / Retry (Optimistic)
- Distributed state
- Error handling more complex
- Diagnostics more complex
For More Information

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

- [www.eaipatterns.com](http://www.eaipatterns.com)
  - Article: Programming without a Call Stack
  - Blog ("Ramblings")
