Event-driven Architectures

Tom Van Cutsem

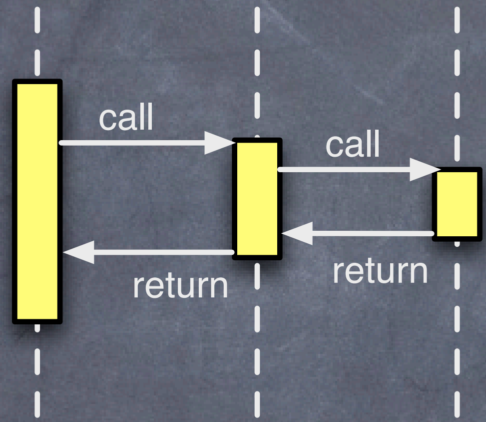
Programming Technology Lab

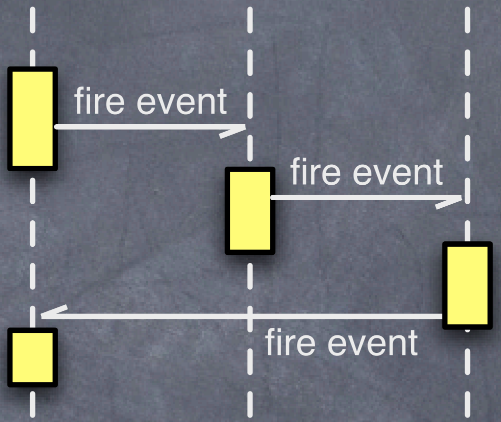
Vrije Universiteit Brussel

Overview

- Event-driven Programming Model
- Event-driven Programming Techniques
- Event-driven Architectures

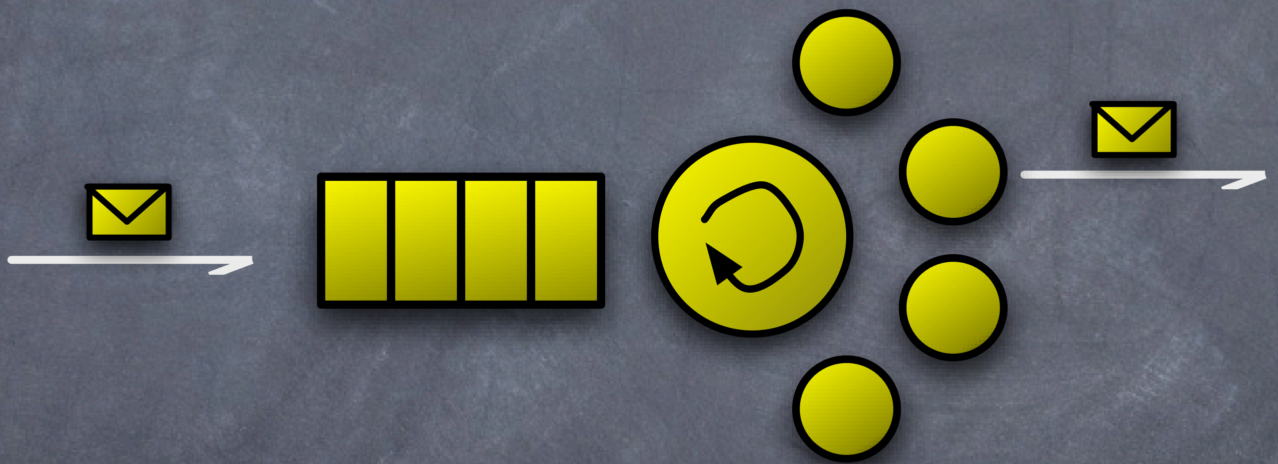
Call versus Event



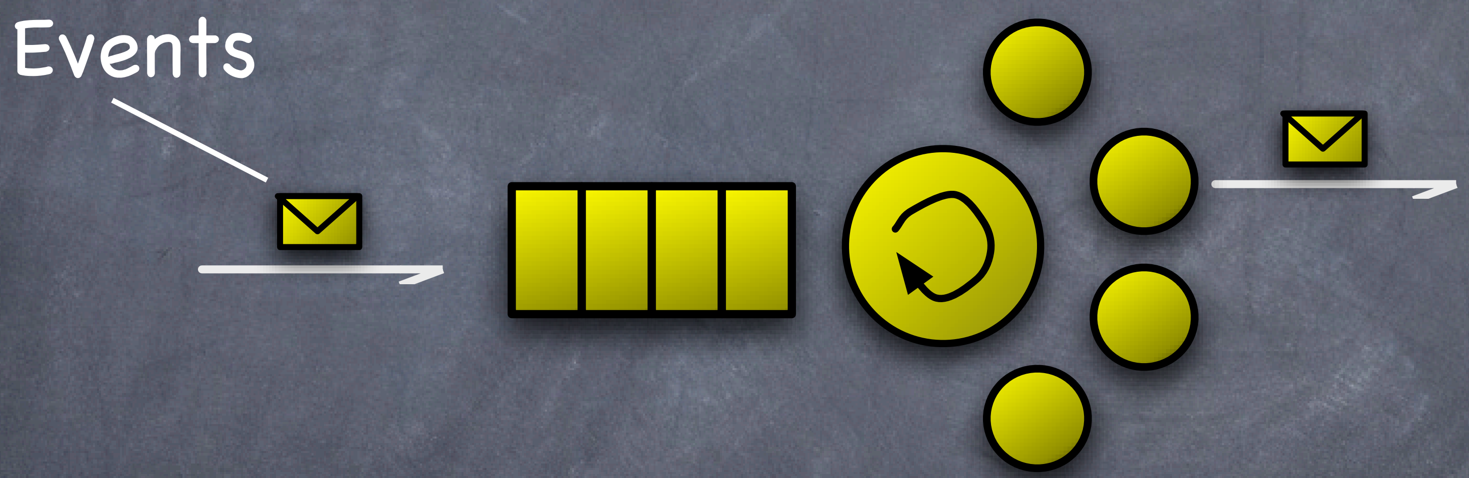


- Programming without a call stack
  - Much more flexible interactions
  - But... free synchronization & context are gone

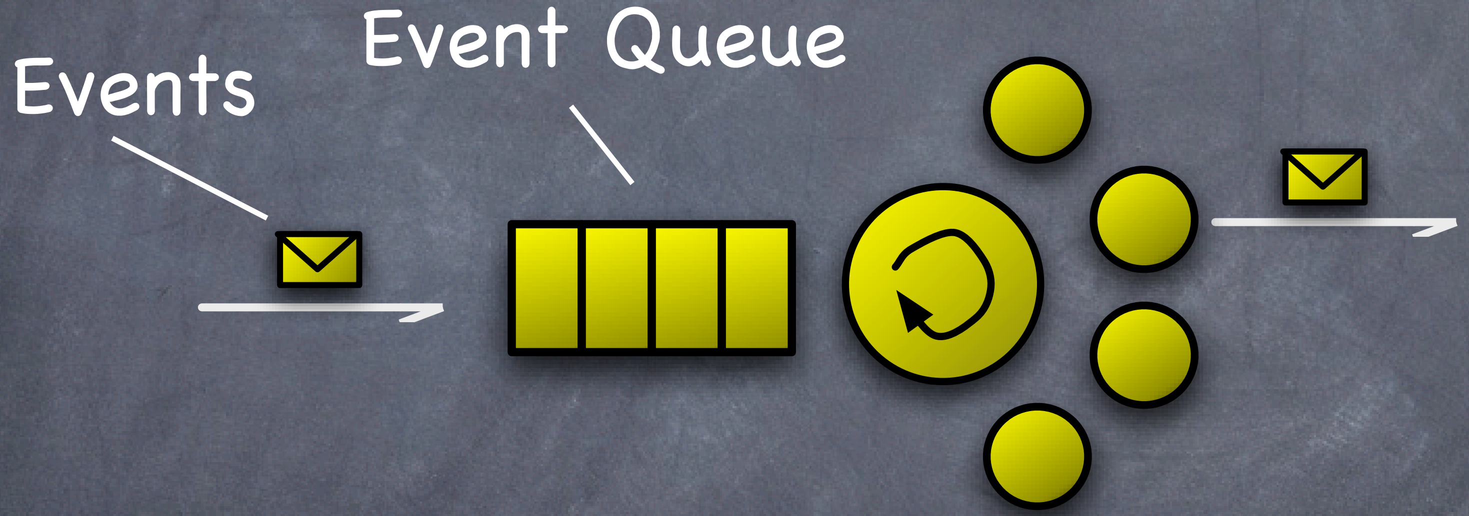
Event-driven Model



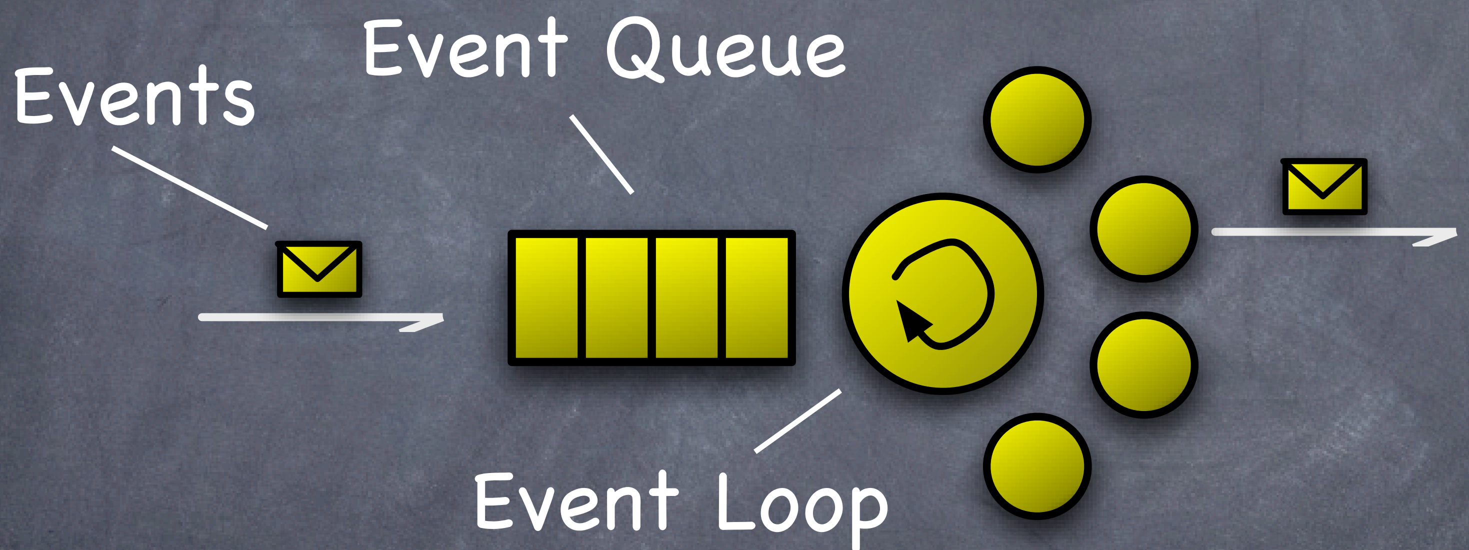
Event-driven Model



Event-driven Model

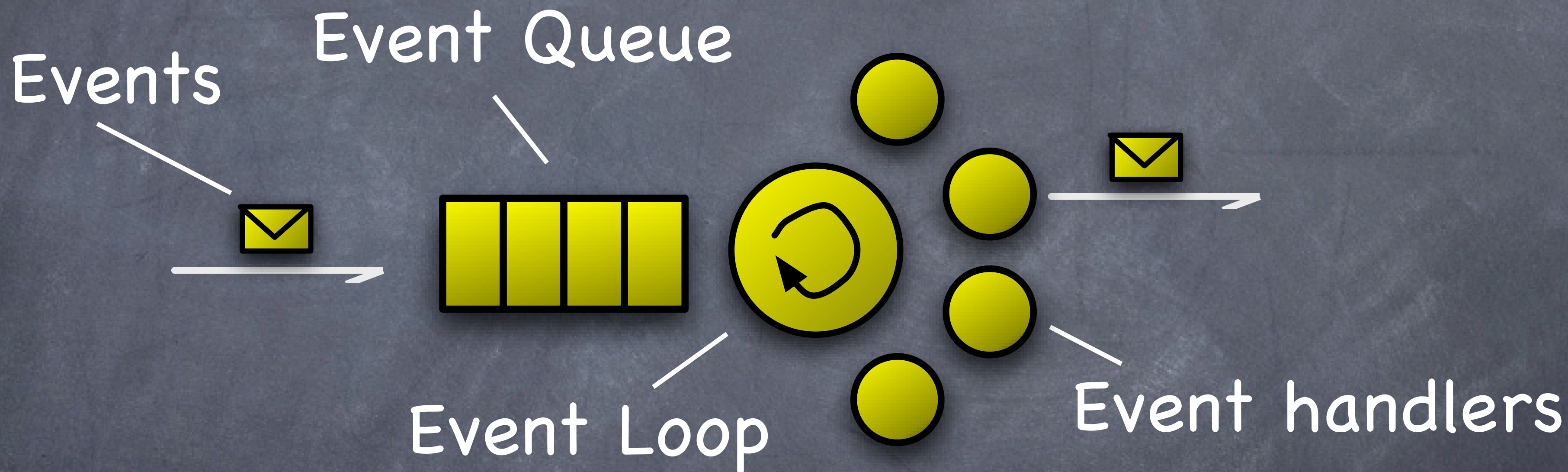


Event-driven Model




```
while (true) {
  Event e = eventQueue.next();
  switch (e.type) {
    ...
  }
}
```

Event-driven Model

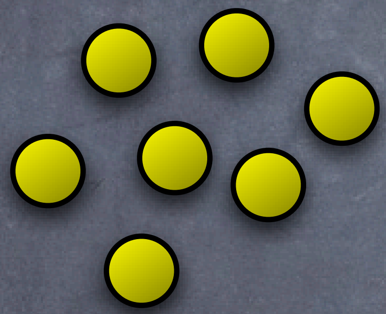



```
void onKeyPressed(KeyEvent e) {
  // process the event
}
```

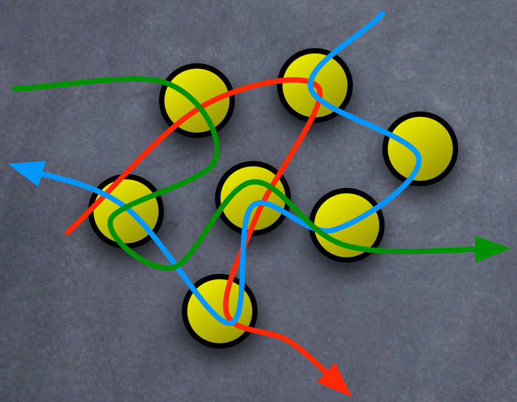
Examples

- GUI Frameworks (e.g. Java AWT)
- Highly interactive applications (e.g. games)
- Operating Systems
- Discrete Event Modelling (e.g. simulations)

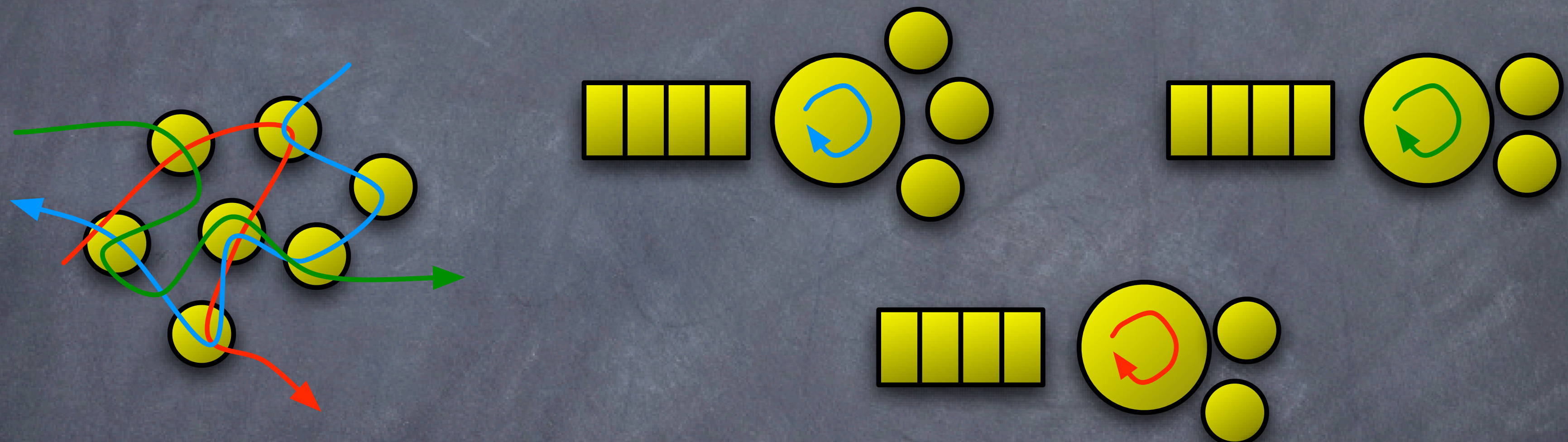
Event-loop Concurrency



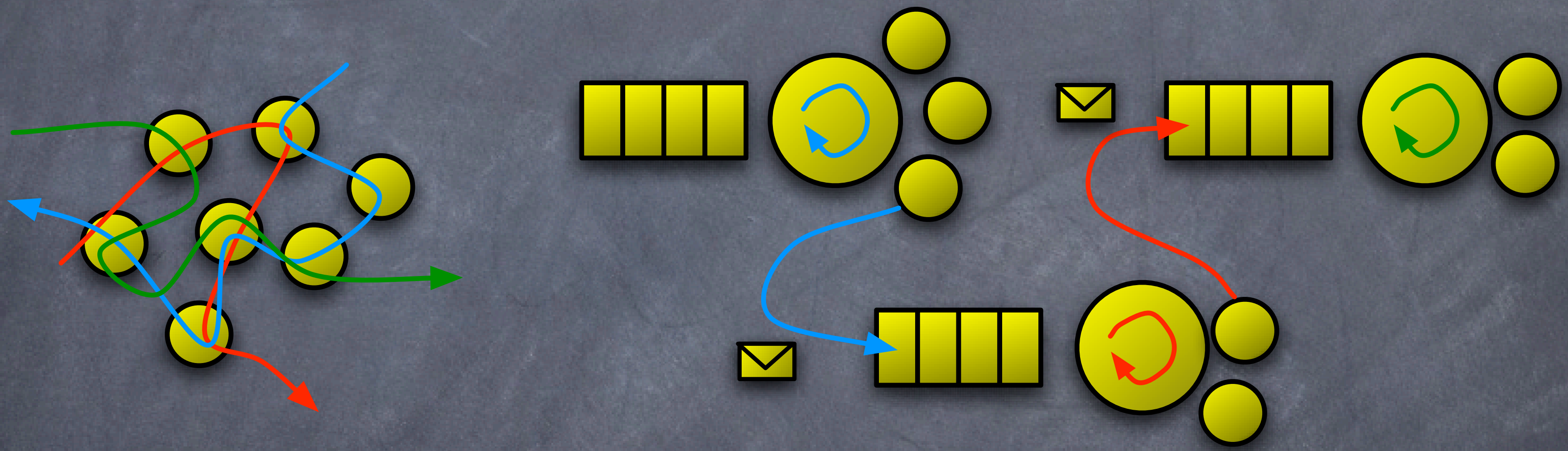
Event-loop Concurrency



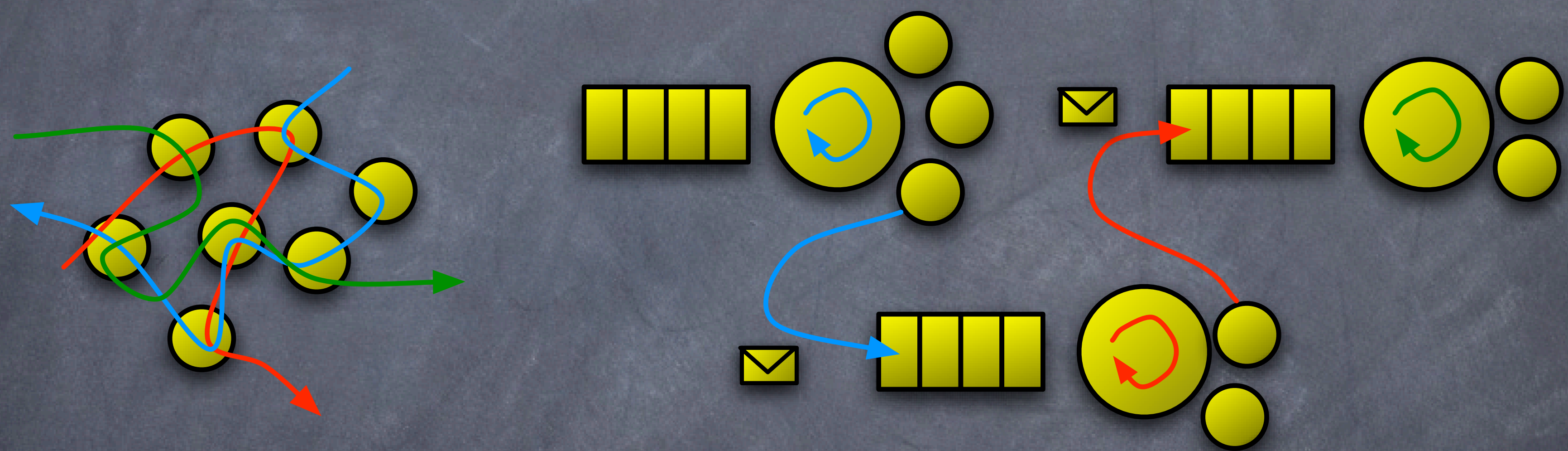
Event-loop Concurrency



Event-loop Concurrency



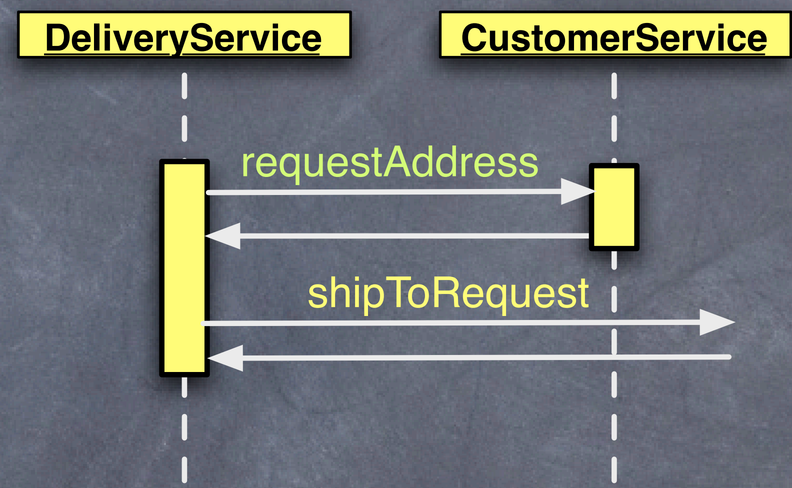
Event-loop Concurrency



- No locks, no deadlocks
- No shared state, no race conditions

Event-driven Programming

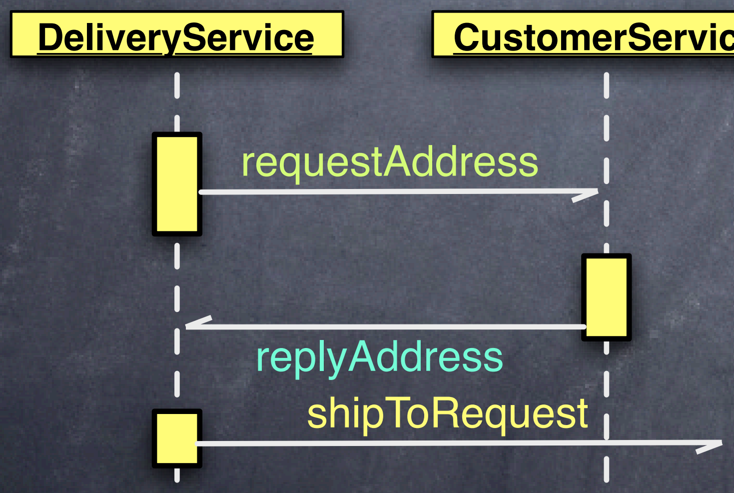
Return values




```
void processDelivery(Order o) {
  // request customer's address
  Address a = customerService.requestAddress(o.customerId));
  courier.shipToRequest(o, a);
}
```

Callbacks

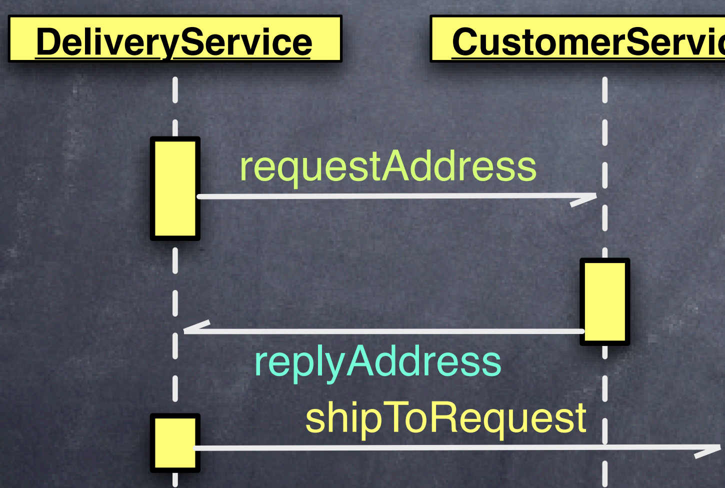
- Dealing with asynchronous 'return values'




```
void processDelivery(Order o) {
  // store order to retrieve it later
  orders.add(o);
  // request customer's address
  customerService.receive(
    new RequestAddress(o.orderId, o.customerId));
}
```

Callbacks

- Dealing with asynchronous 'return values'

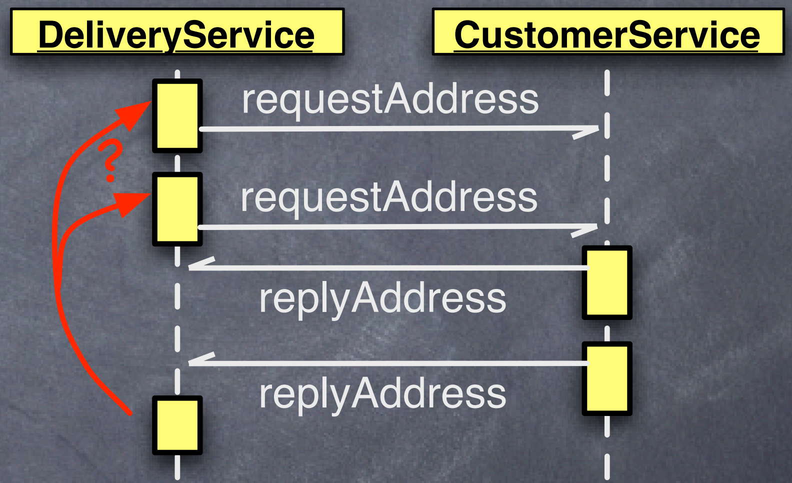



```
void processDelivery(Order o) {
  // store order to retrieve it later
  orders.add(o);
  // request customer's address
  customerService.receive(
    new RequestAddress(o.orderId, o.customerId));
}
```

```
void replyAddress(AddressReply reply) {
  // retrieve order again
  Order o = orders.get(reply.orderId);
  Address a = reply.address;
  courier.receive(new ShipToRequest(o, a));
}
```

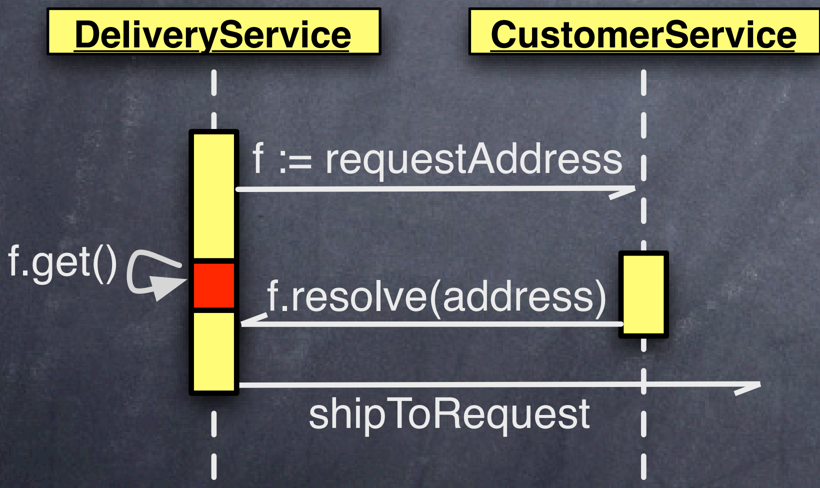
Issues with Callbacks

- Fragmented Code
- Callback is out of context:
  - what is its originating call?
  - what was the state (e.g. local variables) when call was made?



Futures

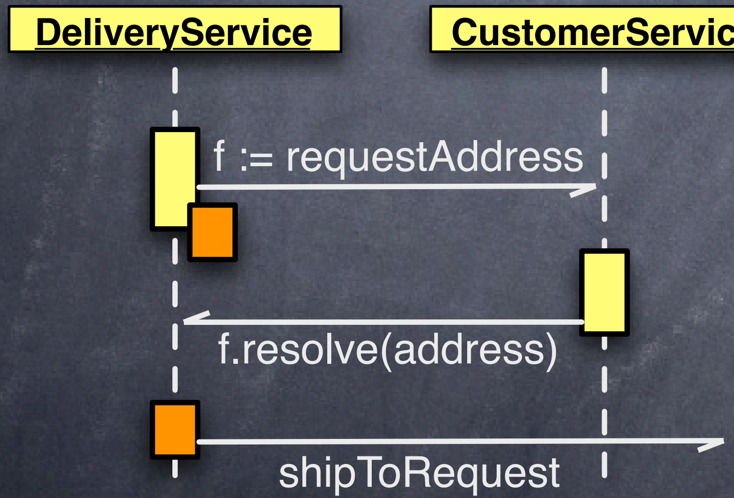
- Placeholders for asynchronous return values
- Typically synchronize when used




```
void processDelivery(Order o) {
  Future addressFuture = customerService.receive(
      new RequestAddress(o.customerId));
  // do things that don't require address
  Address adr = (Address) addressFuture.get();
  courier.receive(new ShipToRequest(o, adr));
}
```

Asynchronous Futures

- Subscription of listeners that are executed when return value is available

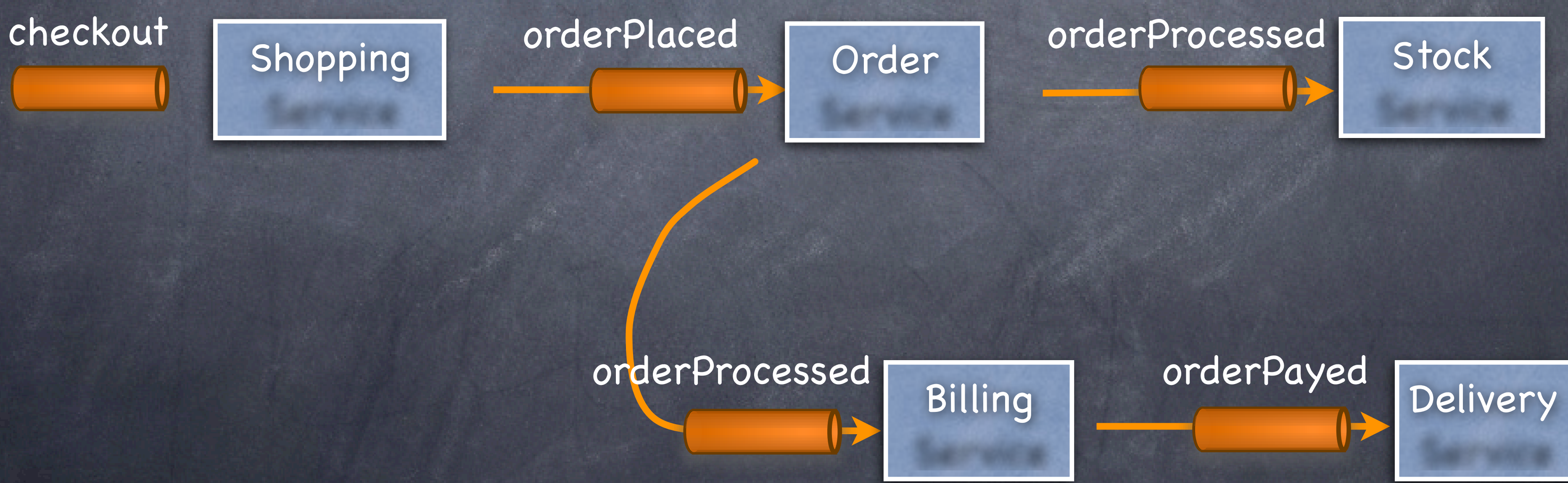



```
void processDelivery(Order o) {
  Future addressFuture = customerService.receive(
    new RequestAddress(o.customerId));
  addressFuture.addListener(new FutureListener() {
    void whenComputed(Result r) {
      Address adr = (Address) r;
      courier.receive(new ShipToRequest(o, adr));
    }
  });
}
```

Event-driven Architecture

Event-driven Architecture

- A program is composed of services
- Services communicate via channels



Channels

- Point-to-point: fixed endpoints
- Publish-subscribe: very loose coupling
- Example: Model-View-Controller

Channels

- Point-to-point: fixed endpoints
- Publish-subscribe: very loose coupling
- Example: Model-View-Controller

Model

Votes

yes	41
no	44
abstain	15

Channels

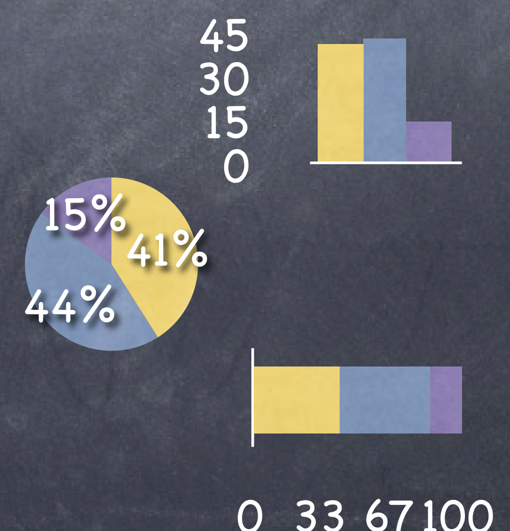
- Point-to-point: fixed endpoints
- Publish-subscribe: very loose coupling
- Example: Model-View-Controller

Model

Votes

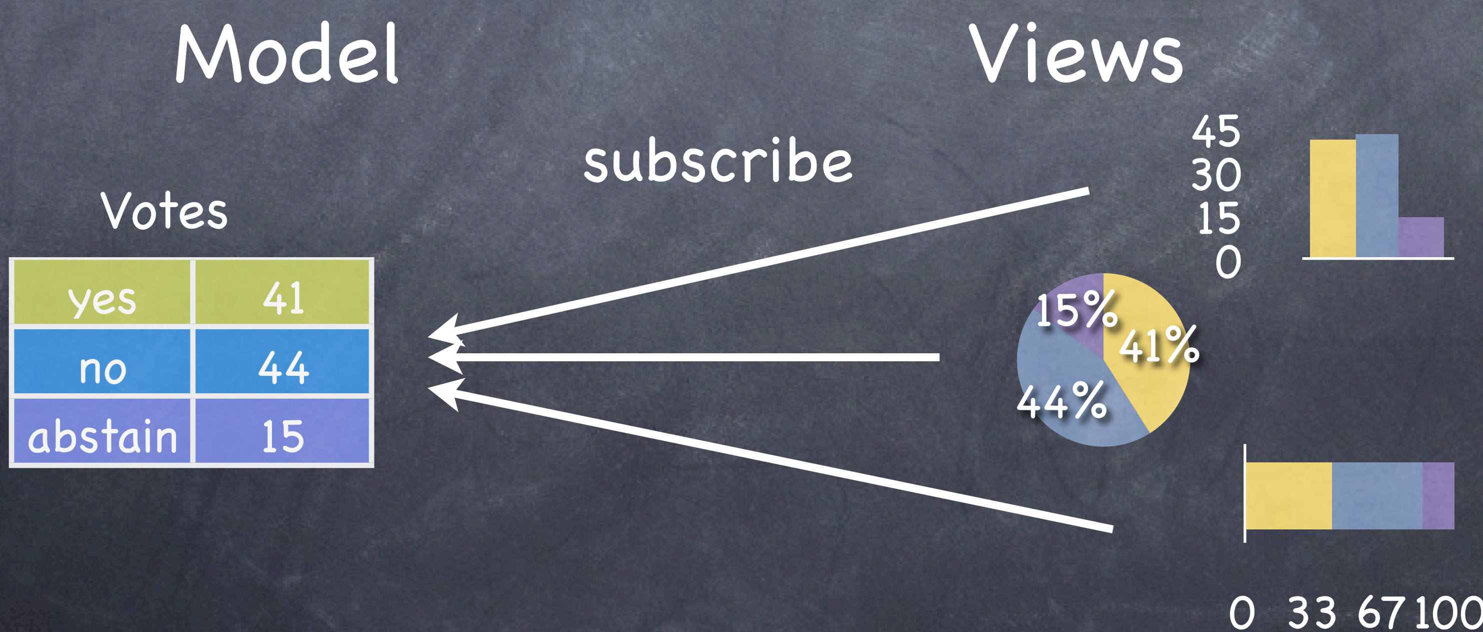
yes	41
no	44
abstain	15

Views



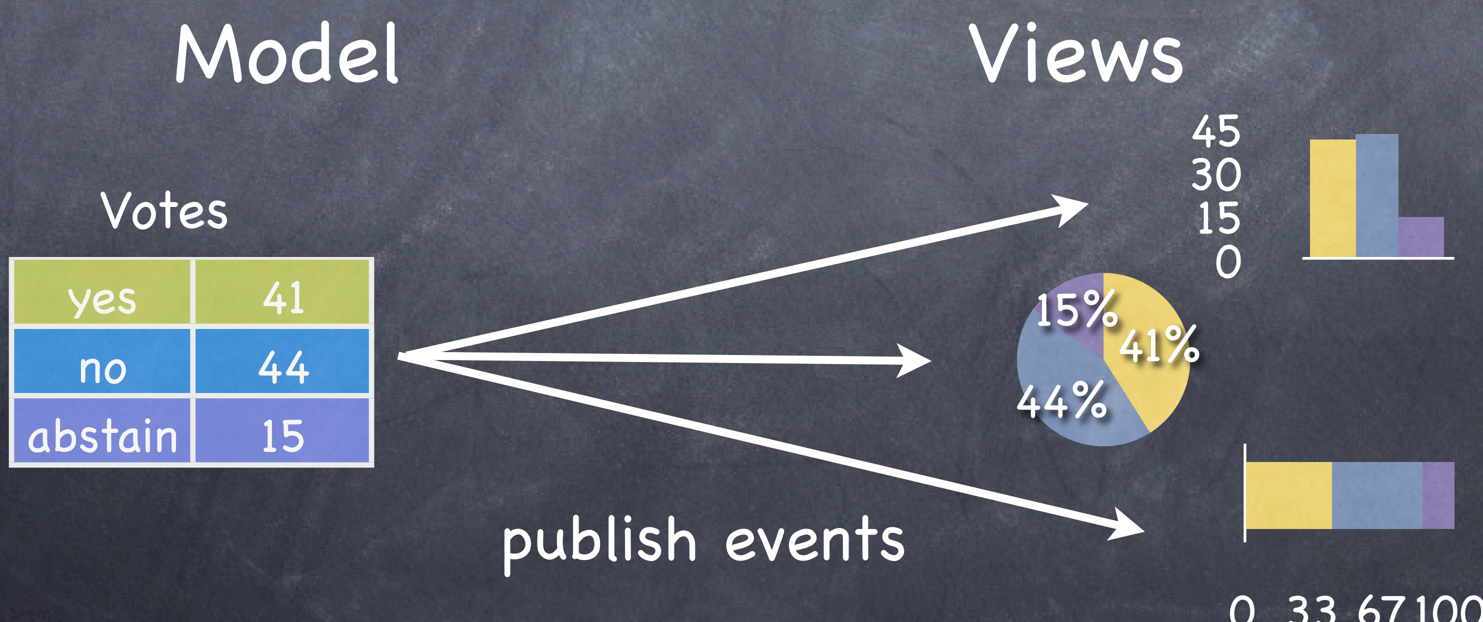
Channels

- Point-to-point: fixed endpoints
- Publish-subscribe: very loose coupling
- Example: Model-View-Controller



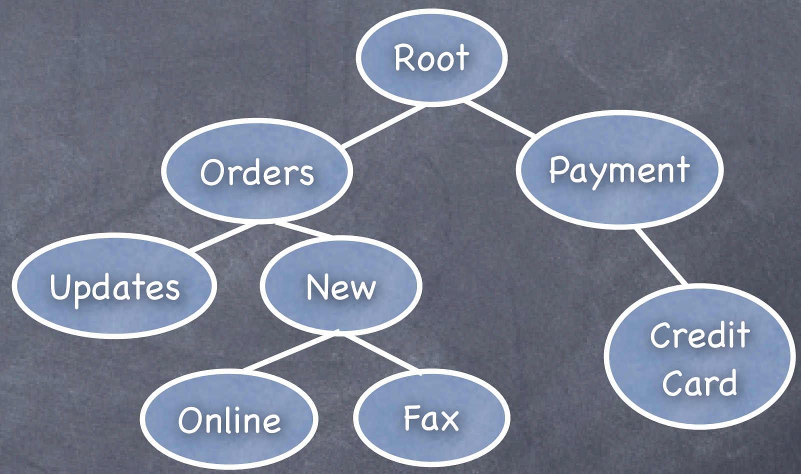
Channels

- Point-to-point: fixed endpoints
- Publish-subscribe: very loose coupling
- Example: Model-View-Controller



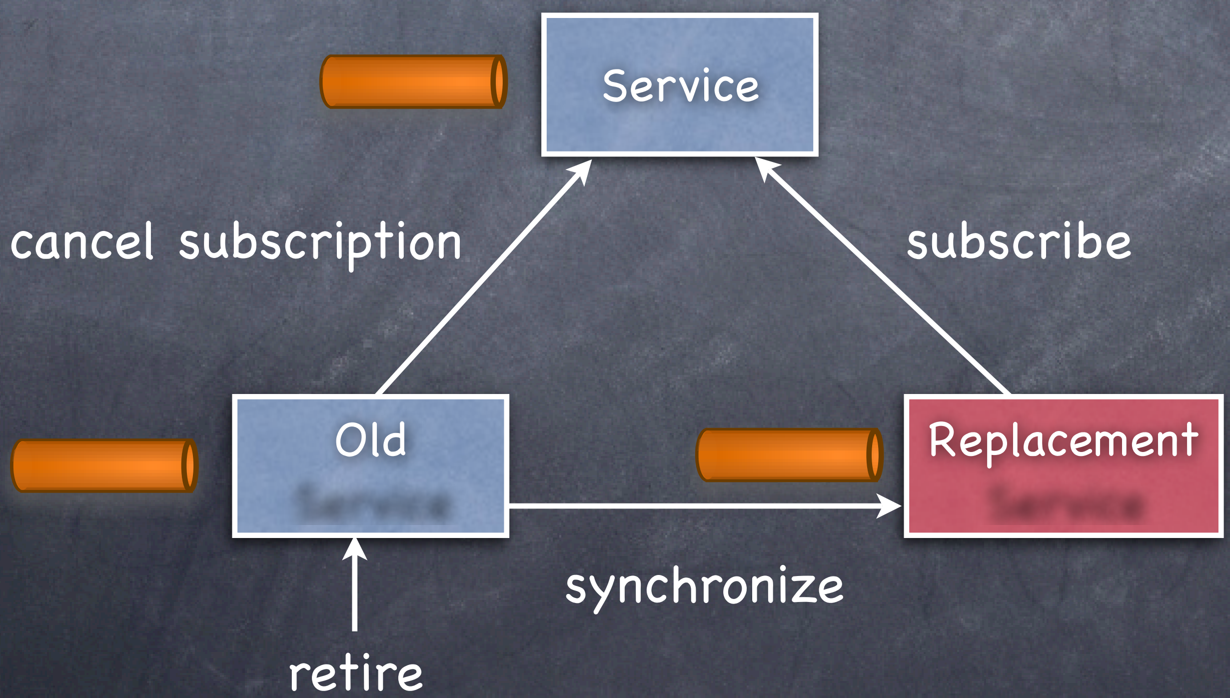
Composing Services

- Service Repository
- Topic hierarchy:
  - Wildcard subscriptions
  - Additional level of abstraction



EDA: Benefits

- Services are highly reusable
- Highly reconfigurable (e.g. upgrades)



EDA: Benefits

- Unit Testing: testing services in isolation

Provides test data

Verifies outgoing events

EDA: Benefits

- Temporal decoupling:
  - services cannot block one another
  - more responsive applications

EDA: Benefits

- Adaptor services easily introduced:
  - logging events
  - authenticating events
  - matching events to an updated interface
  - ...

EDA: Drawbacks

- Loose coupling: implicit control flow
  - makes source code harder to understand
  - less compile-time checks, unit testing even more critical
  - tool support required for easy visualization and composition validation

EDA: Drawbacks

- Temporal decoupling: non-determinism
  - Events may arrive in arbitrary order
  - make as little assumptions as possible on ordering

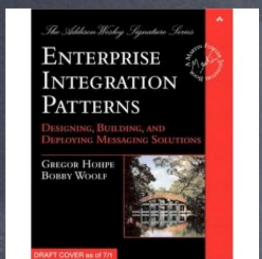
Failure Handling

- Pessimistic synchronization (e.g. 2PC protocol)
  - strong guarantees but...
  - kills asynchrony in the system
- Optimistic synchronization (e.g. compensating actions)
  - works entirely asynchronously but...
  - system (temporarily) in inconsistent state

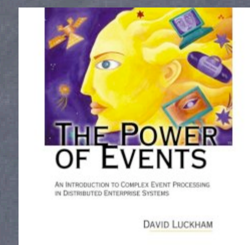
Conclusions

- Event-driven programming = programming without a call stack
- With flexibility comes more responsibility: return values, local state, ordering, ...
- EDA: emphasis on loose coupling
  - Services easily reused
  - Concurrency becomes manageable

References



Enterprise Integration Patterns
Gregor Hohpe and Bobby Woolf
Addison-Wesley



The Power of Events
David Luckham
Addison-Wesley

Concurrency among Strangers
Miller, Tribble and Shapiro
In Symposium on Trustworthy global computing, LNCS Vol 3705, pp. 195-229, 2005

Programming without a call stack
Gregor Hohpe
Available online: www.enterpriseintegrationpatterns.com

Concurrent Object-oriented Programming
Gul Agha
In Communications of the ACM, Vol 33 (9), p. 125, 1990