**Appunti stage**

**Introduzione**

The biggest outcome of the summit was recognizing that when people talk about “events”, they actually mean some quite different things.

1. **Event notification**

This happens when a system sends event messages to notify other systems of a change in its domain. A key element of event notification is that the source system doesn't really care much about the response. There would be a marked separation between the logic flow that sends the event and any logic flow that responds to some reaction to that event. Event notification is nice because it implies a **low level of coupling.** The problem is that it can be hard to see such a flow as it's not explicit in any program text. Often the only way to figure out this flow is from monitoring a live system (this can make it hard to debug and modify such a flow). The danger is that it's very easy to make nicely decoupled systems with event notification, without realizing that you're losing sight of that larger-scale flow.

**N.B.** An event need not carry much data on it, often just some id information and a link back to the sender that can be queried for more information. The receiver knows something has changed, may get some minimal information on the nature of the change, but then issues a request back to the sender to decide what to do next.

**Vantaggi**: disaccoppiamento, flessibilità  
**Svantaggi**: difficile debug e capire il flusso degli eventi nei sistemi GUI

1. **Event-Carried State Transfer**

This pattern shows up when you want to update clients of a system in such a way that they don't need to contact the source system in order to do further work. A customer management system might fire off events whenever a customer changes their details (such as an address) with events that contain details of the data that changed. A recipient can then update it's own copy of customer data with the changes, so that it never needs to talk to the main customer system in order to do its work in the future.  
An obvious down-side of this pattern is that there's lots of data schlepped around and lots of copies. What we gain is greater resilience, since the recipient systems can function if the customer system is becomes unavailable. We reduce latency, as there's no remote call required to access customer information. We don't have to worry about load on the customer system to satisfy queries from all the consumer systems, but it does involve more complexity on the receiver since it has to sort out maintaining all the state, when it's usually easier just to call the sender for more information when needed.

**Vantaggi**: disaccopiamento, migliori performance, minori richieste al supplier, availability(disponibilità)  
**Svantaggi**: lack of consistency

1. **Event-Sourcing**

The core idea of [event sourcing](https://martinfowler.com/eaaDev/EventSourcing.html) is that whenever we make a change to the state of a system, we record that state change as an event, and we can confidently rebuild the system state by reprocessing the events at any time in the future. The event store becomes the principal source of truth, and the system state is purely derived from it. There's no need for event processing to be asynchronous. Another common mistake is to assume that everyone using an event-sourced system should understand and access the event log to determine useful data. But knowledge of the event log can be limited. When working with an event log, it is often useful to build snapshots of the working copy so that you don't have to process all the events from scratch every time you need a working copy. Indeed, there is a duality here, we can look at the event log as either a list of changes, or as a list of states. We can derive one from the other. Version-control systems often mix snapshots and deltas in their event log in order to get the best performance (vedi Git). Event-sourcing has many interesting benefits, which easily come to mind when thinking of the value of version-control systems. The event log provides a strong audit capability (accounting transactions are an event source for account balances). We can recreate historic states by replaying the event log up to a point. We can explore alternative histories by injecting hypothetical events when replaying. Event sourcing does have its problems. Replaying events becomes problematic when results depend on interactions with outside systems. We have to figure out how to deal with changes in the schema of events over time.

**Vantaggi**: audit (revisioni guardando in modo retrospettivo gli eventi), debugging (riportando il sistema ad un determinato istante), storico degli stati/eventi, stati alternativi(non posso fare l’undo ma posso ricreare il sistema ad partire da un preciso istante), memory image(log di eventi al posto di usare database per persistenza dei dati)

**Svantaggi**: difficile comunicazione con servizi esterni, event schema (come salvare gli eventi)

1. **CQRS**

[Command Query Responsibility Segregation (CQRS](https://martinfowler.com/bliki/CQRS.html)) is the notion of having separate data structures for reading and writing information. Strictly CQRS isn't really about events, since you can use CQRS without any events present in your design. The justification for CQRS is that in complex domains, a single model to handle both reads and writes gets too complicated, and we can simplify by separating the models. But the gain for using CQRS has to be balanced against the additional complexity of having separate models.

**Vantaggi**:  
**Svantaggi**:

**Note Conferenza 2017, Chicago**

Ci sono 4 pattern Event-Driven, un sistema Event-Driven c’è quando viene applicato almeno uno di questi pattern.   
Differenze tra il concetto di event e command: command dico al sistema remoto “questo è quello che dovresti fare”, event dico al sistema remoto “questo è cambiato” indicando il cambiamento. Entrambi possono essere wrappati in singoli oggetti.