**Appunti stage**

**Event-Driven architecture**

Events have very different source data, since they can represent many different things. However two important things that events should have are [Time Points](https://martinfowler.com/eaaDev/TimePoint.html). There are two distinct [Time Points](https://martinfowler.com/eaaDev/TimePoint.html) that we should always consider: the time the event occurred and the time the system noticed the event. (In a complex system there may even be several noticing times.) These [Time Points](https://martinfowler.com/eaaDev/TimePoint.html) correspond closely to the notions of actual and record that I talk about in the temporal patterns.

The most common way of making them collaborate is a request/response style. If a customer object wants some data from a salesman object, it invokes a method on the salesman object to ask it for that data.  
Another style of collaboration is Event Collaboration. In this style you never have one component asking another to do anything, instead each component signals an event when anything changes. Other components listen to that event and react however they wish to. The well-known observer pattern is an example of Event Collaboration. [Event Collaboration](https://martinfowler.com/eaaDev/EventCollaboration.html) isn't mandatory when using events in applications, also the choice between [Event Collaboration](https://martinfowler.com/eaaDev/EventCollaboration.html) and request/response isn't an exclusive choice (questa affermazione si discosta dalla definizione di evento di Fowler).  
[Event Collaboration](https://martinfowler.com/eaaDev/EventCollaboration.html) results in very loose coupling which makes it particularly easy to add new components to a system without needing to modify existing components. The downside of [Event Collaboration](https://martinfowler.com/eaaDev/EventCollaboration.html) is that it's very hard to understand the collaboration. Request/response collaborations are specified in some form of code that shows the entire flow, [Event Collaboration](https://martinfowler.com/eaaDev/EventCollaboration.html) is much more implicit - which makes it much harder to debug when something unexpected happens.  
Events just communicate that something happened - with an event you let a system know that Y has happened. Another difference is that you think of broadcasting events to everyone who may be interested but sending commands only to specific receiver.

**Introduzione**

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

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.  
Possiamo usare più pattern allo stesso tempo per ottenere migliori performance senza rendere troppo complicato il nostro sistema?   
Cos’è un evento?

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**

Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not just can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes.

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.

The key to Event Sourcing is that we guarantee that all changes to the domain objects are initiated by the event objects. This leads to a number of facilities that can be built on top of the event log:

* **Complete Rebuild**: We can discard the application state completely and rebuild it by re-running the events from the event log on an empty application.
* **Temporal Query**: We can determine the application state at any point in time. Notionally we do this by starting with a blank state and rerunning the events up to a particular time or event. We can take this further by considering multiple time-lines (analogous to branching in a version control system).
* **Event Replay**: If we find a past event was incorrect, we can compute the consequences by reversing it and later events and then replaying the new event and later events. (Or indeed by throwing away the application state and replaying all events with the correct event in sequence.) The same technique can handle events received in the wrong sequence - a common problem with systems that communicate with asynchronous messaging.

The simplest way to think of using Event Sourcing is to calculate a requested application state by starting from a blank application state and then applying the events to reach the desired state. It's equally simple to see why this is a slow process, particularly if there are many events.

**Structuring the Event Handler Logic**

There are a number of choices about where to put the logic for handling events. The primary choice is whether to put the logic in [Transaction Scripts](https://martinfowler.com/eaaCatalog/transactionScript.html) or [Domain Model](https://martinfowler.com/eaaCatalog/domainModel.html). As usual [Transaction Scripts](https://martinfowler.com/eaaCatalog/transactionScript.html) are better for simple logic and a [Domain Model](https://martinfowler.com/eaaCatalog/domainModel.html) is better when things get more complicated.  
A good way to think of this is that there are two responsibilities involved. *Processing domain logic* is the business logic that manipulates the application. *Processing selection logic* is the logic that chooses which chunk of processing domain logic should run depending on the incoming event. You can combine these together, essentially this is the [Transaction Script](https://martinfowler.com/eaaCatalog/transactionScript.html) approach, but you can also separate them. Next is whether to put the processing selection logic in the event object itself, or have a separate event processor object. The problem with the processor is that it necessarily runs different logic depending on the type of event. All things being equal you want the processing selection logic in the event itself, since that's the thing that varies with the type of event.

**Reversing Events**As well as events playing themselves forwards, it's also often useful for them to be able to reverse themselves. Reversal is the most straightforward when the event is cast in the form of a difference. An example of this would be "add $10 to Martin's account" as opposed to "set Martin's account to $110". In the former case I can reverse by just subtracting $10, but in the latter case I don't have enough information to recreate the past value of the account.   
If the input events don't follow the difference approach, then the event should ensure it stores everything needed for reversal during processing. You can do this by storing the previous values on any value that is changed, or by calculating and storing differences on the event.  
It's worth remembering that all the capabilities of reversing events can be done instead by reverting to a past snapshot and replaying the event stream. As a result reversal is never absolutely needed for functionality. However it may make a big difference to efficiency since you may often be in a position where reversing a few events is much more efficient than using forward play on a lot of events.

**External Updates**

One of the tricky elements to Event Sourcing is how to deal with external systems that don't follow this approach (and most don't). You get problems when you are sending modifier messages to external systems and when you are receiving queries from other systems.

Many of the advantages of Event Sourcing stem from the ability to replay events at will, but if these events cause update messages to be sent to external systems, then things will go wrong because those external systems don't know the difference between real processing and replays.

An easy way to help deal with this is to ensure that your system calls external systems through gateways that can be configured to ensure that no messages go out unless you are processing an event 'for real'.

To handle this you'll need to wrap any external systems with a [Gateway](https://martinfowler.com/eaaCatalog/gateway.html). This in itself isn't too onerous since it's a thoroughly good idea in any case. The gateway has to be a bit more sophisticated so it can deal with any replay processing that the Event Sourcing system is doing. For rebuilds and temporal queries it's usually sufficient for the gateways to be able to be disabled during the replay processing. You want to do this in a way that's invisible to the domain logic.  
  
The problem with external queries is that the data that they return has an effect on the results on handling an event. If I ask for an exchange rate on December 5th and replay that event on December 20th, I will need the exchange rate on Dec 5 not the later one.

**Code Changes**We can think as three broad kinds of code changes here: new features, defect fixes, and temporal logic.

* **New features:** essentially add new capabilities to the system but don't invalidate things that happened before. These can be added pretty freely at any time. If you want to take advantage of the new features with old events you can just reprocess the events and the new results pop up.
* **Bug fixes:** occur when you look at past processing and realize it was incorrect. For internal stuff this is really easy to fix, all you need to do is make the fix and reprocess the events. Your application state is now fixed to what it should have been.
* **Temporal logic:** the third case is where the logic itself changes over time, a rule along the lines of "charge $10 before November 18 and $15 afterwords". This kind of stuff needs to actually go into the domain model itself. The domain model should be able to run events at any time with the correct rules for the event processing. You can do this with conditional logic, but this will get messy if you have much temporal logic. The better route is to hook strategy objects into a [Temporal Property](https://martinfowler.com/eaaDev/TemporalProperty.html)

**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.

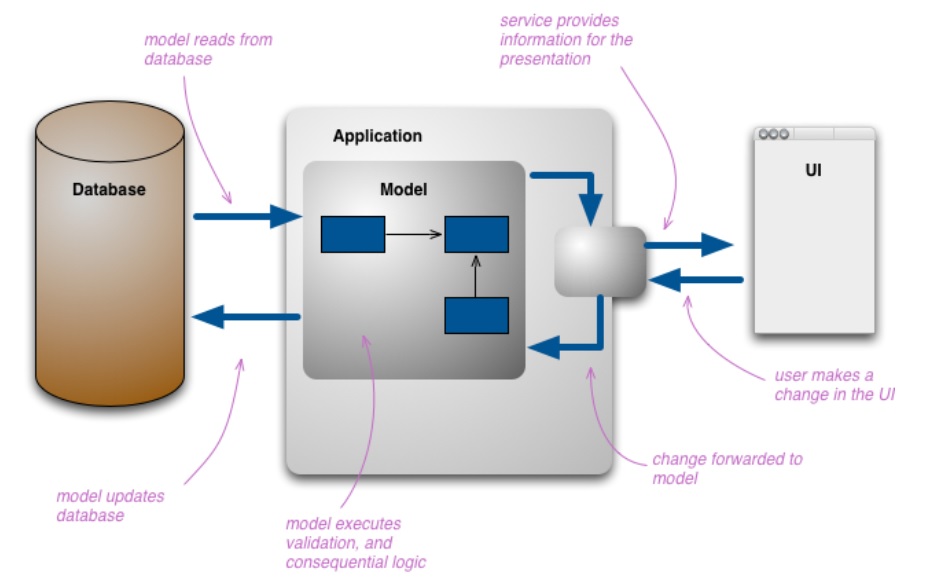
The change that CQRS introduces is to split that conceptual model into separate models for update and display, which it refers to as Command and Query.

CQRS fits with some other architectural patterns:

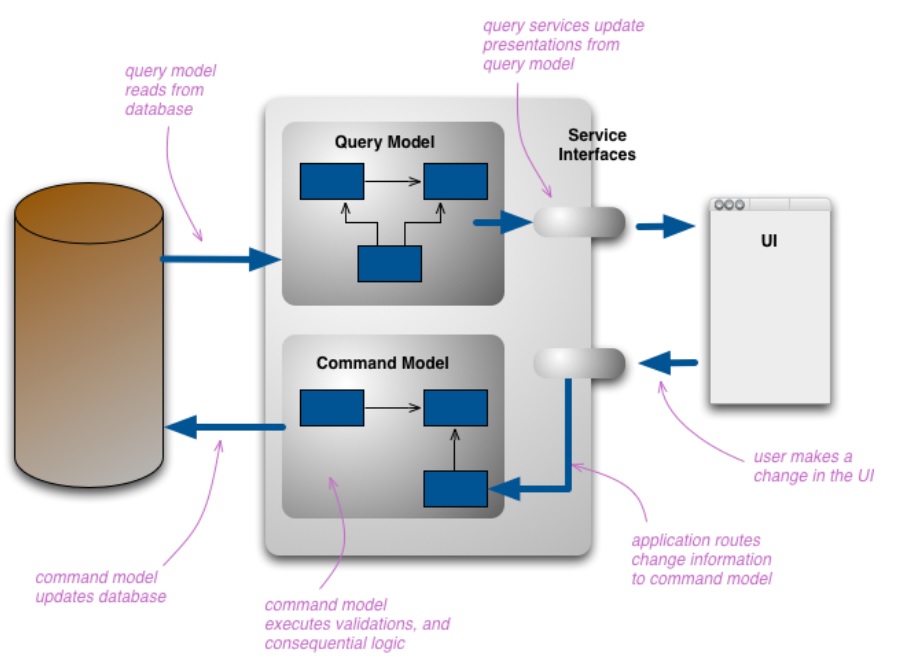
* interact with via CRUD, we can easily move to a task-based UI
* CQRS fits well with [event-based programming models](https://martinfowler.com/eaaDev/EventNarrative.html). It's common to see CQRS system split into separate services communicating with [Event Collaboration](https://martinfowler.com/eaaDev/EventCollaboration.html). This allows these services to easily take advantage of [Event Sourcing](https://martinfowler.com/eaaDev/EventSourcing.html).
* Having separate models raises questions about how hard to keep those models consistent, which raises the likelihood of using [eventual consistency](http://www.allthingsdistributed.com/2008/12/eventually_consistent.html).
* If the write model generates events for all updates, you can structure read models as [EventPosters](https://martinfowler.com/bliki/EventPoster.html), allowing them to be [MemoryImages](https://martinfowler.com/bliki/MemoryImage.html) and thus avoiding a lot of database interactions.
* CQRS is suited to complex domains, the kind that also benefit from [Domain-Driven Design](https://www.amazon.com/gp/product/0321125215?ie=UTF8&tag=martinfowlerc-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=0321125215)

In particular CQRS should only be used on specific portions of a systemand not the system as a whole.  
Firstly is that a few complex domains may be easier to tackle by using CQRS. The other main benefit is in handling high performance applications. CQRS allows you to separate the load from reads and writes allowing you to scale each independently. If your application sees a big disparity between reads and writes this is very handy. Even without that, you can apply different optimization strategies to the two sides. An example of this is using different database access techniques for read and update.

PRIMA



DOPO



**Vantaggi**: adatto per operazioni CRUD, separazione tra scrittura e lettura  
**Svantaggi**: non adatto a sistemi troppo complessi ma può aggiungere complessità in sistemi semplici

**Appunti da unipd**