Takeaways summary

EventStorming MasterClass

Workshop Formats

There is not a single EventStorming format. EventStorming is actually an *open platform for designing your own workshop,* eventually diving in from big picture to fine grained design of a software oriented solution.

Big Picture

This is the best format to explore complexity in an intricate domain. You'd like to involve all the different perspectives in the workshop, that will progressively build structure upon a chaotic structure.

Variations on this theme allow to handle scenarios like ***enterprise reboot*** (gathering understanding and consensus around a new organisational vision); critical ***project kick-off*** (exploring the scope and the surroundings of a new enterprise software project) or ***organisation retrospective*** (beyond the boundaries of a typical team retro).

The actual outcomes usually go beyond the official promise: collective learning is the first outcome (usually with no negative side-effects), but the achieved clarity usually triggers interesting reactions.

Process Modelling

This is tailored around the needs of designing processes and services as a synthesis of different disciplines and perspectives, like *business* (focusing on value delivery, but also managing costs and revenues), *user experience* (focusing on the needs and feelings of users and internal stakeholders) and *software development* (focusing on the responsibility boundaries and implementation constraints).

A sophisticated language is created step by step injecting precision in the discussion. No hand-off will be necessary: different tribes are already in.

Software Design

Domain Events allow the discussion to get fine grained, exploring the forces behind a given software structure. Independent software components emerge around the need for consistent behaviour. Outcomes of complex processes map very well with end-to-end testing strategies such as BDD.

The divergent attitudes of the modelling team will require a specific discipline in leading the discussion: *branching*, *drilling* and the tendency to *solve everything* are powerful forces that need to be tamed in order to deliver successful results.

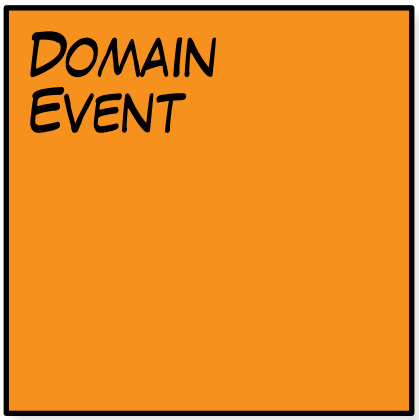
Other Formats

Just start with events, make sure you have an extra supply of sticky notes, with no predefined meaning, apply **incremental notation** and surprise yourself!

Interesting experiments have been

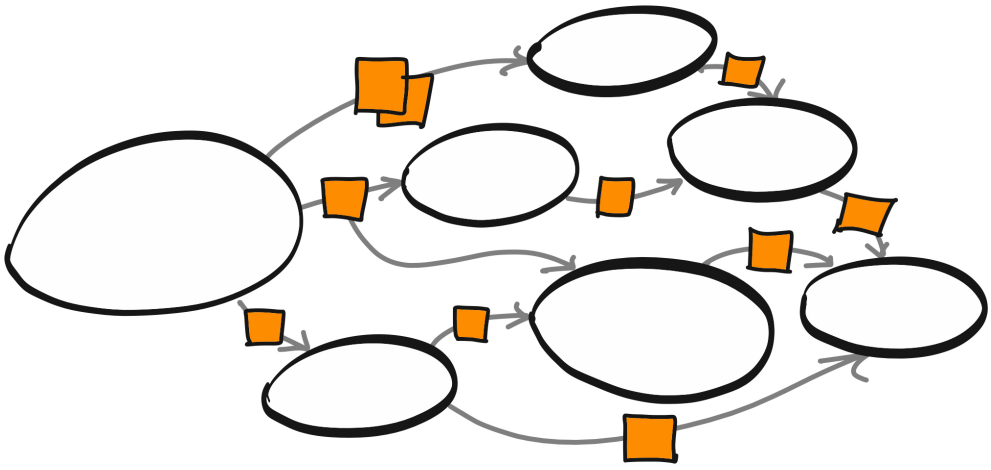
- merging with **Value Stream Mapping**, looking explicitly to value and measuring delays.
- Visualising **alternative futures**: take a business hypothesis (or more than one) and flesh it out!
- **Book Storyboarding**, visualising chapters structure and takeaways.

The power of Events



Events are the glue that make the magic possible.

- They're building blocks of **consistent storytelling**, and this will allow your team to double check the emerging narrative.
- They're **semantically robust**, a fact that happened at a given moment in time carrying less ambiguity than a *process* or a *phase*.
- They're **fine grained**. Once we get rid of spatial limitations, we can see the forest *and* the trees. Turns out it's often just a matter of space.
- They **trigger human reactions**. Getting deep into details we implicitly trigger emotional responses from participants, challenging the fallacies of the current narrative.
- They **trigger system reactions**. Policies are organisation's behaviour made explicit. And they're hooked onto given events.
- They **allow precise software modelling**. The DDD community struggled around the concept of Ubiquitous Language, mostly looking for nouns, we now know that Events are a way better building block for such a language.
- They allow for **more efficient software architectures**. Software monoliths are not a good choice for the long term, loosely coupled bounded contexts propagating information like domain events are a much more resilient choice.



Domain Events are a more natural and efficient way to propagate information across bounded contexts.

Enriching your model with Hot Spots

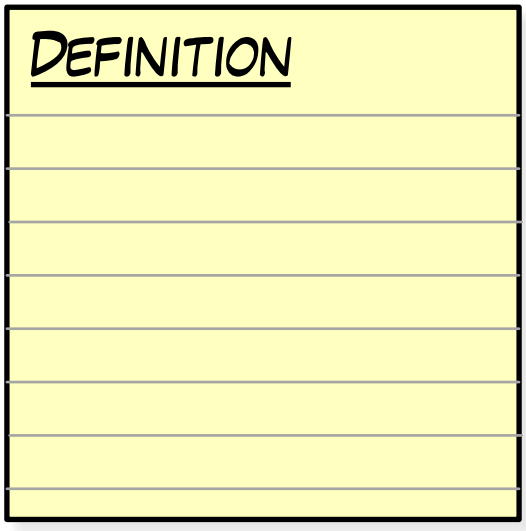
A lot of the content during exploration and modelling doesn't strictly fit *in* the model. But it's a vital part of our learning! So ...let's make it visible.

A **Hot Spot** is an annotation we can add in realtime to our model, with the explicit purpose to *make everything visible*, including disagreements, and questions.

In Big Picture, typically hot spots start to emerge once we start to *enforce the timeline,* and people are asked to build a consistent model of the whole instead of just a locally ordered cluster of their own silo.

In Process Modelling and Software Modelling, Hot Spots also serve as a tool to keep the modelling going without getting swamped by the possibile alternatives and the ongoing objections.

Adding definitions where needed



Sometimes there are terms that seem to be clear to everybody except you. This is where making the definition visible might come in handy.

Unlike hotspots, definitions are supposed not to be controversial, it's just a visible *clarification* for a specific term that doesn't carry too much ambiguity.

They're great for acronyms, making the actual meaning accessible to everybody.

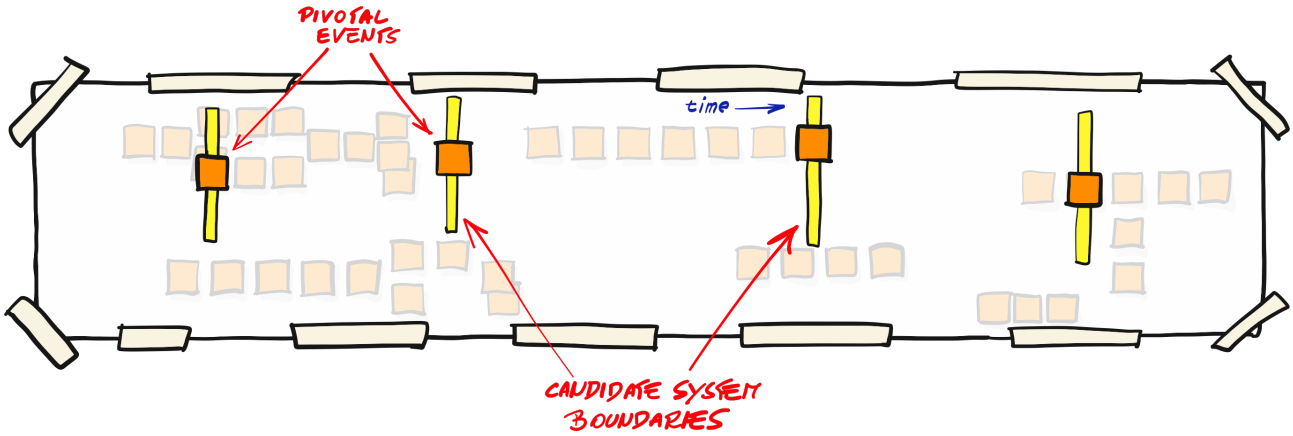
Warning: don't fall into the trap of *trying to provide a definition for everything*, discussing about the meaning of *names* will severely cripple the workshop, turning the discussion into a database design session in disguise, exactly what we're trying to avoid.

EMERGING STRUCTURE

While enforcing the timeline, we should choose a strategy to speed up the sorting operations. We'll have a few options.

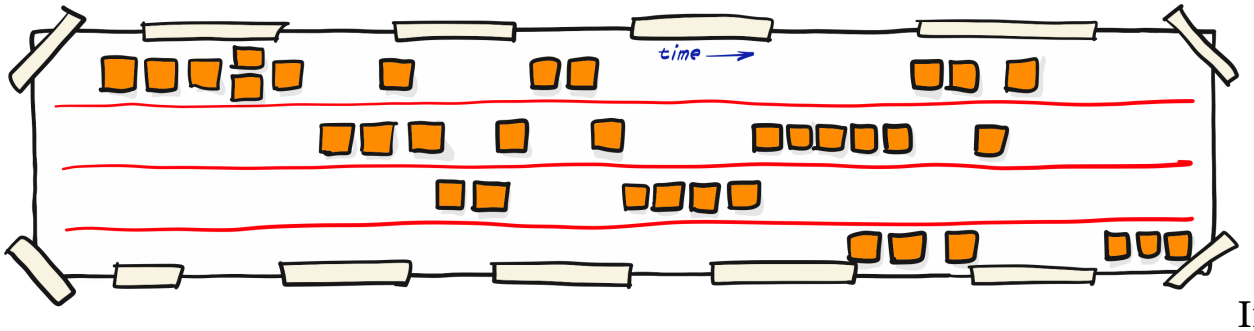
PIVOTAL EVENTS

Some events are more important than others, and are actually defining key moments in the flow. It's fairly easy to have participants agree on what those events are, and we just need 3-4 of them to speed up sorting operations.



SWIM-LANES

Swim-lanes work great for parallel processes and responsibilities. Unfortunately, readability comes at a price: we'll quickly fall short of space.

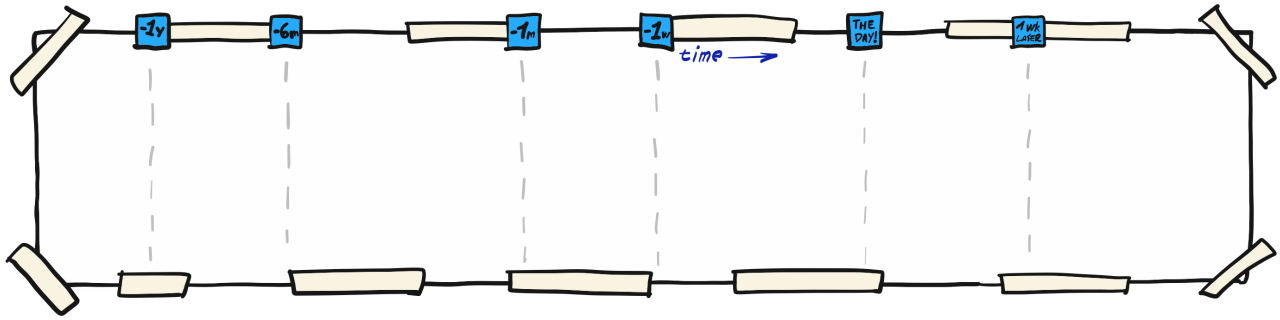


In general I tend to use swim lanes *after* defining vertical chunks with Pivotal Events or Temporal Milestones, and place a yellow label to name the swim-lane.

I also often use a variation on this format - called Us and Them - to do retrospectives on a collaboration between two teams and organisations: same story, different perspectives.

Temporal Milestones

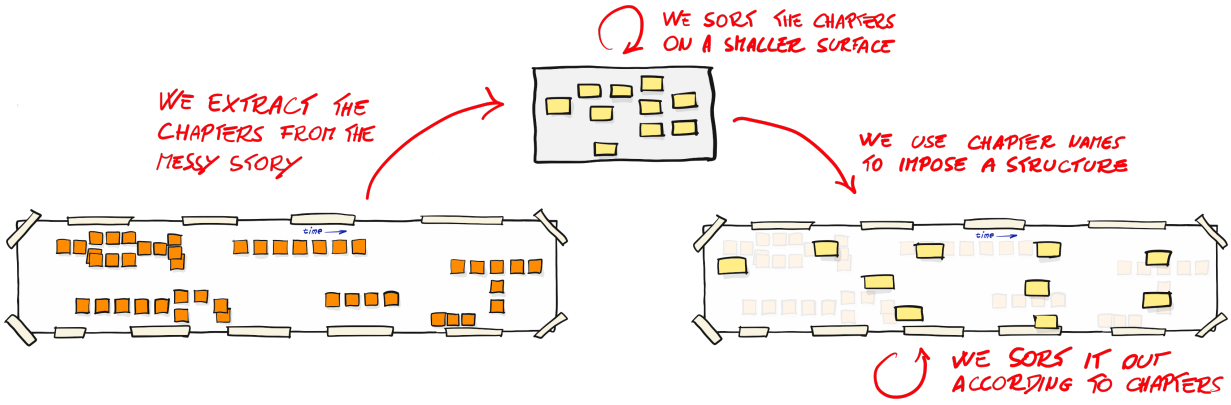
Some businesses are structured in a *Big Bang fashion* (like organising conferences or big events) or are driven by seasons (like making wine, for example). In these scenarios it may be convenient to prepare the paper roll in advance with some temporal notation (I usually put blue stickies above the paper roll).



This way the exploration scope would be clearer and the sorting won't take ages.

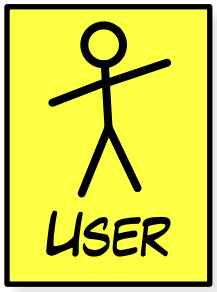
Chapter Sorting

A few times, the modelling surface simply isn't enough to sort everything. My last resort is to *extract the chapters* from the story - usually with different the colours on a smaller surface - sort the chapters, and then apply the order on top of the existing messy workflow.



This way sorting becomes a little faster.

People and systems



Showing **people** is the most obvious answer to the need to see *who is doing what*.

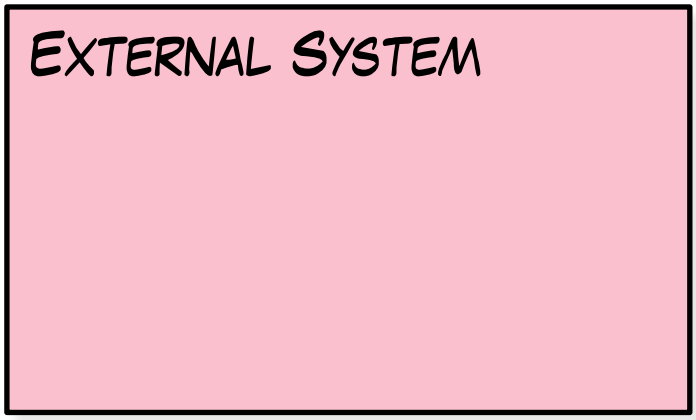
I prefer the term *people* instead of *actors, roles,* or *personas,* because of its fuzziness. The lack of precision allows different points of view to be visualised at the same time.

People can be roles, customer segments, a specific person with name and surname, … it depends!

It also allows to augment our notation without choosing a winning party: *this is not UML, this is not Customer Journey*.

We don't need to match every single orange sticky note with the corresponding sticky figure. In some places, picking the right people will feel trivial, in some other places, you'll find out that the system behaviour is dependent on different persons/roles, and so on.

That's just great! 👍 Now make it visible!



External Systems are the other big counterpart in our system.

They have an even fuzzier definition: they can be pieces of software, like external products, or they can be an external entity or another department inside your organisation.

My definition is:

"Whatever you can put the blame on"

…which opens the possibility to have a broader set of external systems, including things like *'Bad Luck', 'The Weather', 'The European Community'* and so on.

Interestingly, this tends to trigger extra thinking about corner cases, which are part of your systems and of your business too. Or about constraints that can be discussed and challenged once they've been visualised.

Explicit Walkthrough

Once we've added People and Systems we badly need to sort out everything.

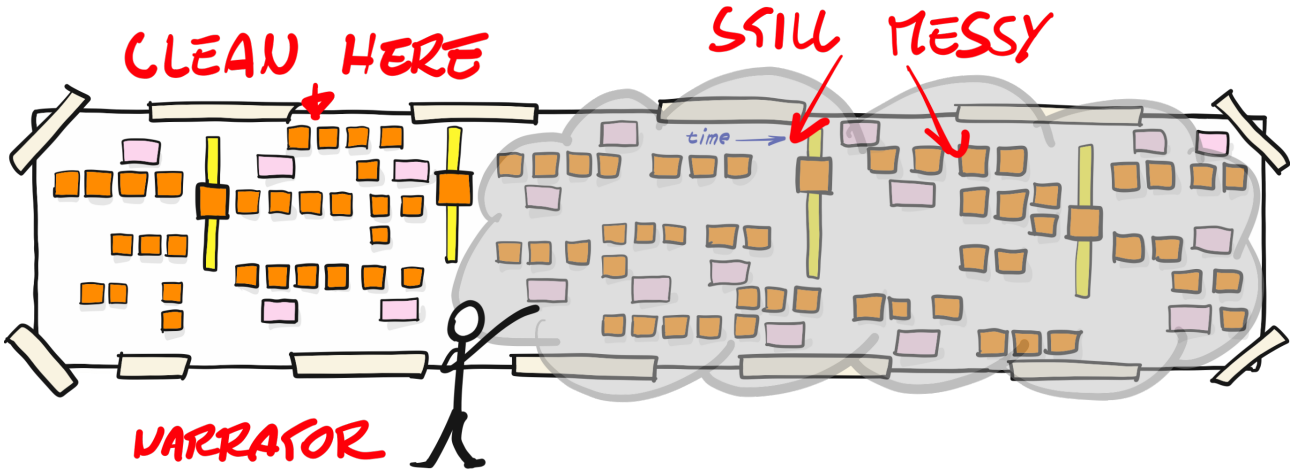
This is when parallel exploration is not efficient anymore and we should constraint the conversation bandwidth to one.

In order to validate this understanding we need three distinct responsibilities: ***Telling the story***, ***Validating the story***, ***Make sure the model is in sync with the conversation***.

But this isn't that easy, so we gotta play it smart.

Walkthrough Kick-Off

- **Facilitators will tell the story**: it's going to be hard, since they're also the less experts about the problem domain. As a consequence, they'll try to tell the story (often finding out that some important building blocks are missing, or that the story can start in several different ways).
- **Audience will validate the story**: when facing an external narrator naively describing their work, experts usually tend to riot. If you warned the audience about it, and you're not worried about being frequently corrected, then we're fine! The facilitator has no authority, so the audience shouldn't be scared about correcting.
- **Facilitator will keep the model in sync**: when learning new stuff in the process, the real facilitator duty is to make sure the model is in sync with the conversation that just happened. This means adding more events, splitting swim-lanes, rewriting stuff and do some clean-up.



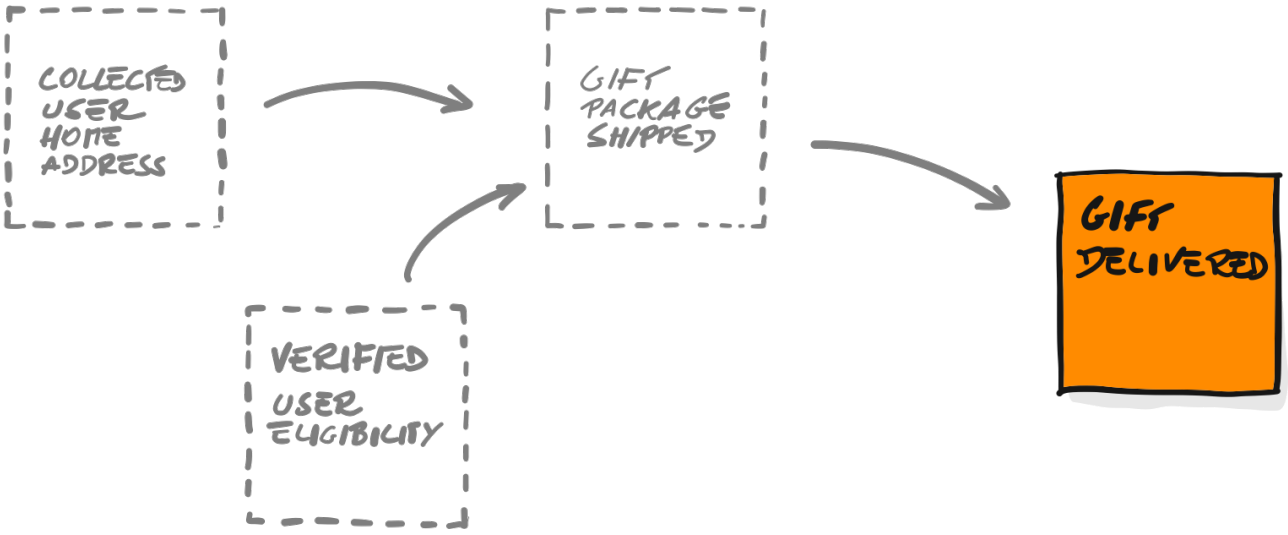
Usually, at the first pivotal events the narrator can pass the narrator responsibility to a domain expert, and focus on the model. But now the audience is ready to challenge the narrator's voice regardless of the authority.

Reverse Narrative

An even more powerful tool to enforce consistency of our model is to try to tell the story in reverse, picking an event from the end, and checking the steps needed to get there.

The narrator usually picks an event from the end of the flow and start asking questions like: *"which events need to happen in order to make this one possible?",* sometimes we have the foundation event in the past, other times we discover that we forgot something important, and we need to add it to the model.

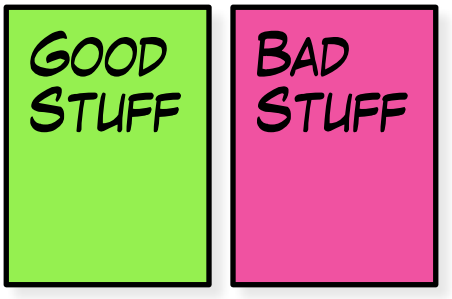
Reverse narrative usually unveils a lot more complexity from the story, it helps detecting events that we forgot, or alternatives that we didn't explore (*this needs to happen here, but ...what if it doesn't?)*



Picking the event

Usually we start from some event that looks "terminal" like a *happy state at the end of the flow*, or straight from *pivotal events*: they tend to have a very interesting nature of ambiguity and complexity.

Exploring Value



Why are people doing the things they do? What are they expecting to get in return?

The moment we start exploring the value created in a business process, the moment we open infinite possibilities.

The first obvious value to look at is Money, but quickly we'll discover that money is not the only currency in play.

We might waste ***time*** of our users with useless steps, or we might discover that lack of engagement, reduces the conversation rate of a given page (thus making us lose ***money***, ...again).

We might discover how many emotions are involved in a typical business interaction: *stress, anger, fear, pride, safety, anxiety,* just to name a few.

We might discover that different users are driven by different emotions and so we cannot treat them exactly in the same way.

Or we might discover that some key steps in our flow lack a motivation, making the underlying business hypothesis, an unrealistic dream.

Problems and Opportunities

A good way to wrap up a Big Picture workshop is to call for an explicit round of **Problems** and **Opportunities**.

If you've been performing the Exploring Value step there's a good chance some inspiration for issues and ideas is already in the air.

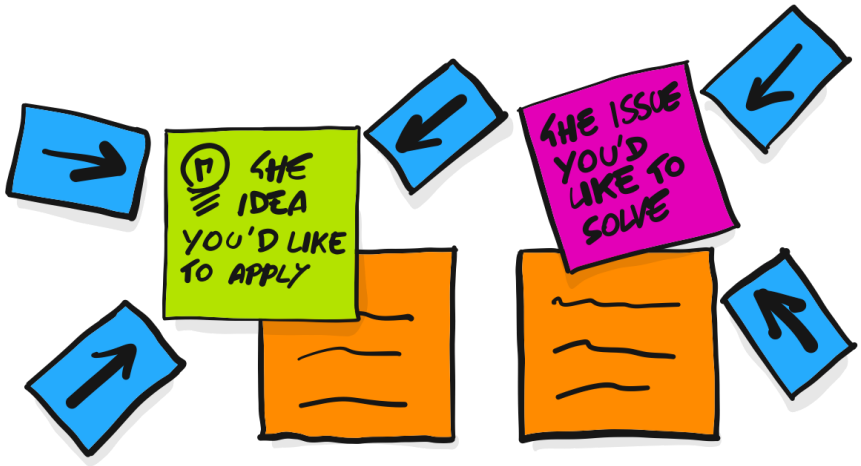
I usually keep the same colour combination: *Green* for good things —> Opportunities and Ideas, and *Magenta* for bad things.

We've already been exploring bad things as a side activity during the workshop, whenever they were spontaneously emerging during the discussion. But this does not guarantee the exploration was complete: maybe the facilitation didn't leave enough space to dissonant voices, or maybe there's simply something which isn't visible yet.

A quick round to highlight problems and opportunities makes sure that everybody is finally on the same page.

Arrow Voting

A tons of problems, albeit mitigated by the presence of some cool opportunities, is not an actionable outcome. If the combination of people in the room is safe you may want to call for an explicit round of arrow voting, with little blue stickies.

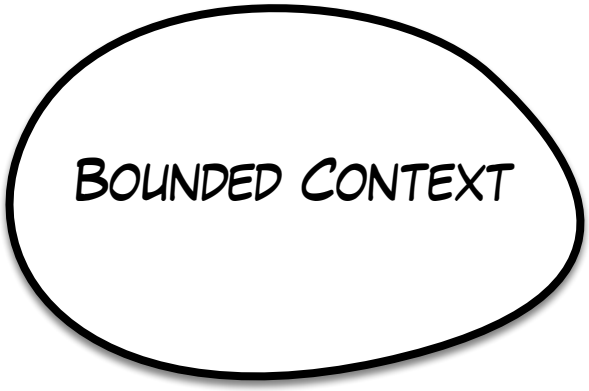


Everybody gets 2 votes, and they can choose what they think is the most important issue to attack.

Given the people in the room (selection bias plays a strong role here) you now end up with a very clear indication of what to do the next morning.

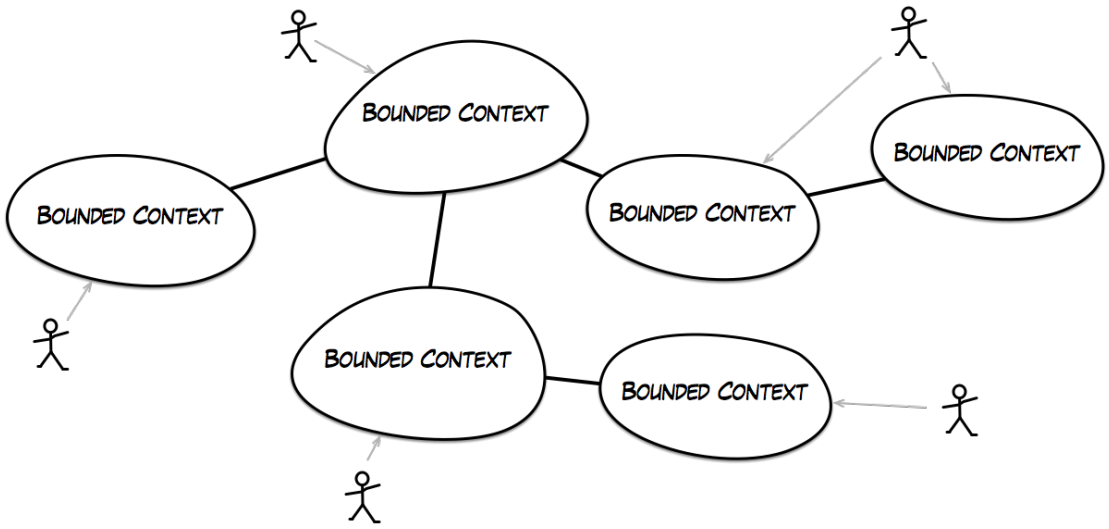
Languages and Bounded Contexts

Organisations don't usually have consistent languages to describe their domain. Silos actively fragment knowledge into locally consistent bubbles, but end-to-end language consistency is usually a myth. The larger the organisation, the more specialised jargons thrive around a specific function.



A **Bounded Context** is a unit of language consistency, a portion of the model that guarantees that every term has one and only one meaning, greatly improving our modelling precision.

A good software architecture is the result of the combination of multiple specialised models, implemented around a specific purpose/business need.



An actionable Heuristic

Language consistency is like a canary in a coal mine, since it's sensible to variations coming from different sources. A more practical rule of thumb would be to:

- **minimise the number of involved stakeholders,** the fewer the people, the higher the language uniformity;
- **focus on behaviour,** since *data* would easily lead into ambiguities;
- **focus on purpose** but keeping in mind that it would be hard to grasp for some stakeholders (*i.e. "Managing is not a purpose")*.

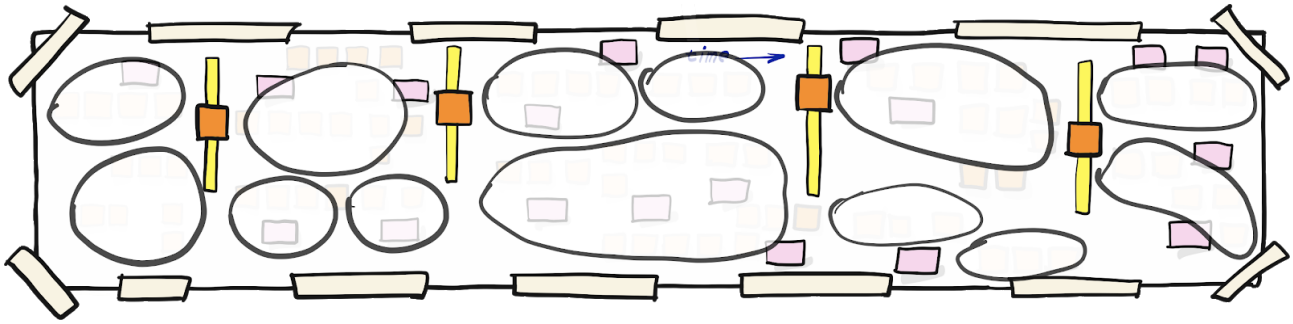
Deriving Bounded Context from Big Picture

During big picture exploration we'll have a few clues about how to effectively partition a system:

- we'll see where **people** tend to be, and usually different people tend to have different specific *needs*;
- We'll see **boundaries**, between different *phases* and **swimlanes**, often pointing to different modelling needs, like *the expert on the left and the novice on the right*.

However, we should be aware that defining the right context boundaries, despite being vital for a healthy software architecture, is not necessarily crucial for the business. Well... it *is* crucial, but they don't know.

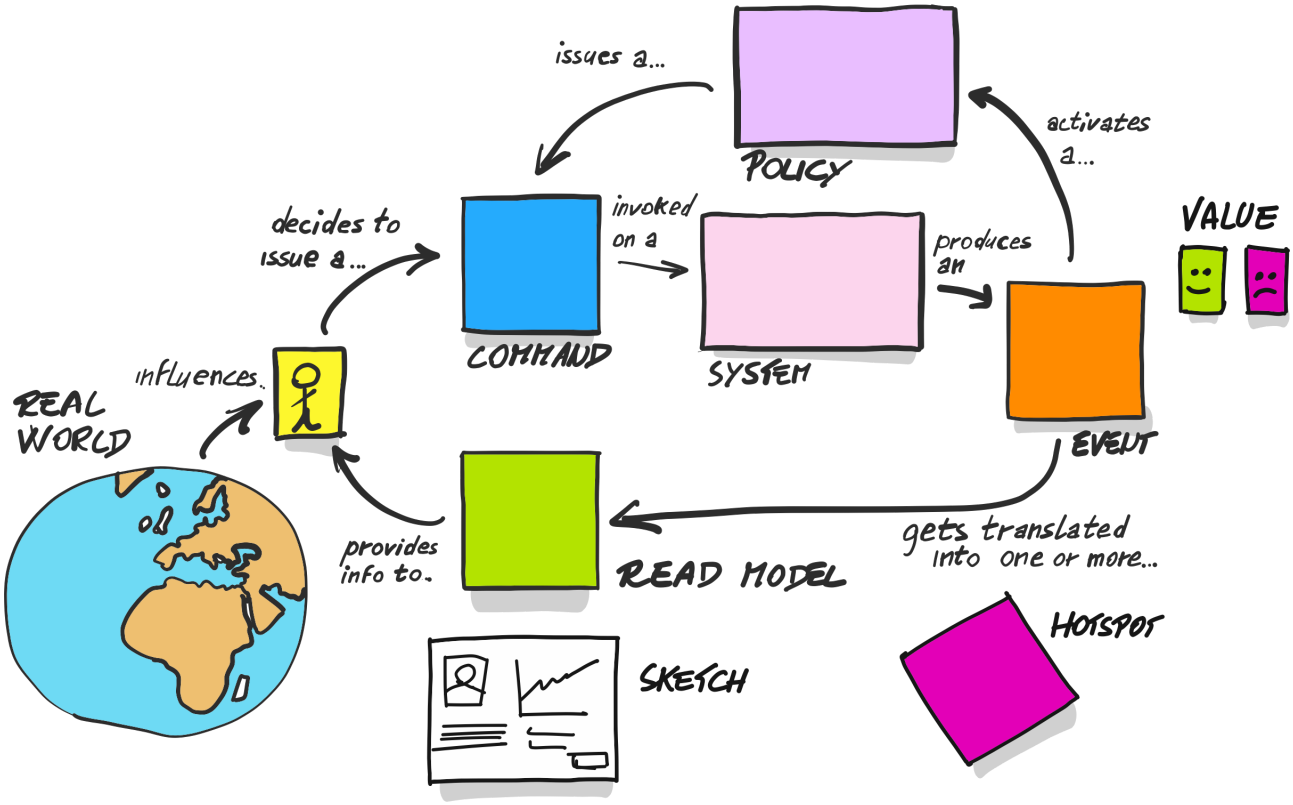
So, I keep drawing the boundaries as a homework activity, to be completed by the technical team once the workshop is officially finished, without the need of business participants: they already gave us all the information we needed.



PROCESS MODELLING

THE PICTURE THAT EXPLAINS EVERYTHING

This will probably come in handy when doing **process modelling**.



Users take decisions (*blue*) according to their real world experience, and the information that they see on the screen (the Read Model, *green*).

The user command is processed by a system that triggers a domain event as an outcome. In order to be readable for different users, the Domain Event will have to be translated into one or more read models.

At the same time, Events may trigger some Policies (*lilac*) representing the business rules, or better the organisation reactive logic:

*"Whenever [a given **event**] happens, we do [**command** / **action**]".*

A COOPERATIVE GAME

Collaboratively Modelling processes and software is tough: providing better tools for doing that is only half of the job, specially when your model needs to be the synthesis of contribution of many different expert specialists.

We probably need a different metaphor too, and I really like the idea of cooperative games, the ones where the players are supposed to win a challenge by *cooperating*, instead of competing.

THE RULES OF THE GAME

Here are the rules we can use when modelling processes and/or the underlying software.

1) Every process might end in a stable state, usually a combination of an Event and a Read Model.
2) The colour grammar must be respected.
3) Every involved stakeholder should be reasonably happy (using value stickies to visualise it).
4) Every hotspot that should arise during the modelling session should be addressed.

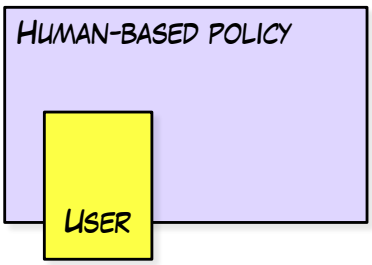
This are the *rules*, then our job will be to figure out the winning strategies at the game.

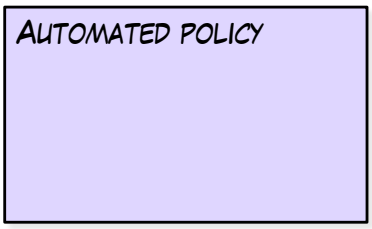
...but let's have a look to the building blocks first.

POLICIES

Policies are probably the most interesting thing to model in EventStorming. They cover all the spectrum of organisation reactions from human behaviours to automation including:

- **habits**
- **unconscious reactions**
- **Tacit agreements**
- **Explicit agreements**
- **Codified or corporate rules**
- **Inconsistent rules** (different actors implementing the same rule in different ways)
- **Listeners** (stateless listeners performing easy tasks, like sending notifications)
- **Process Managers and Sagas**





DISCOVERING POLICIES

The link between an event and its consequences can be encapsulated by a policy, but when the event-reaction combination is too obvious, we might miss the link.

Forcing ourself to obey the rule:

*There has to be a **lilac** between the **orange** and the **blue***

...is a good way to force yourself into asking deeper questions about the nature of the policy.

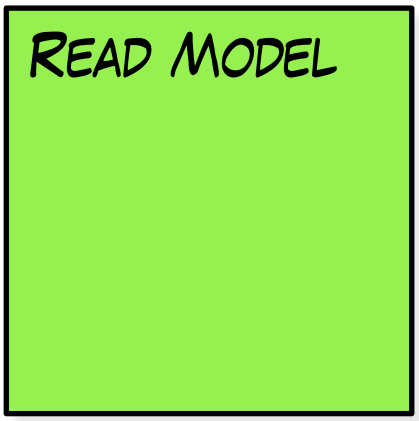
CHALLENGING POLICIES

Policies are places where the business is never telling the whole story. To challenge the first round of explanation you may want to repeat the explanation louder, adding the words **"always"** and **"immediately"**, and watch your experts going in auto-correction mode.

CHALLENGING ASSUMPTIONS

The implicit one - for developer, mostly - is that *automation is the only way*. In practice, many times the real process improvement is in making a human step *easier*, instead of *faster*.

Read Models



Read models are an interesting twist in modelling style: they provide ***the information needed in order to take a given decision****.*

It's not infrequent to find blockers in processes which are related to decisions which are hard to take because the information needed is scattered in multiple systems.

Example*: in order to fill my timesheet, I won't need much information if I track my hours on a daily basis, a mobile app would be just fine. Things become harder if I don't do the tracking daily, but do it at the end of the month. In this scenario the read model becomes a combination of my mailbox (did I write any emails?), my google calendar (where was I supposed to be?), my GitHub commit history or my phone log.*

The big flip in design has to do with the fact that Read Models are designed ***driven by the needs of the consequential decision****. Given that the decision is important - we"re on the bottleneck after all - implementation of the read model is subordinated to the following decision.*

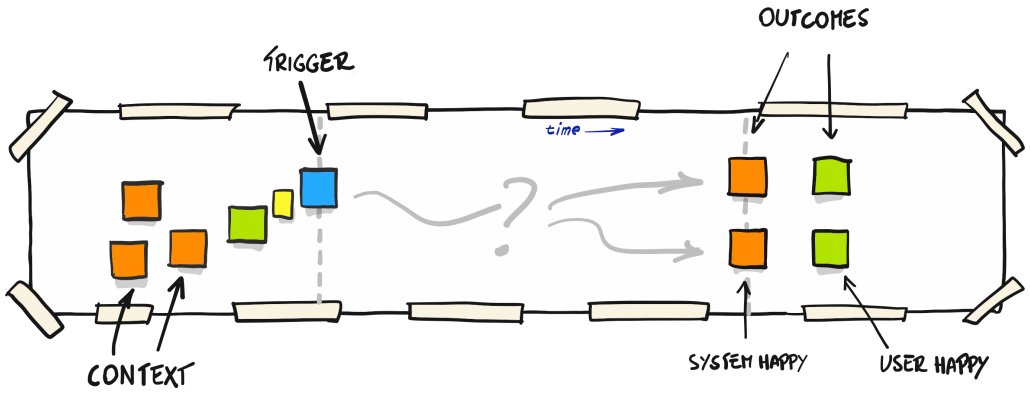
Possible implementations

Avery strategy is welcome, here are a few:

- ***Projections*** *listening to event streams and creating a dedicated read models. Some may be really aggressive on the loading time like a single-document-per page (for example on a product page on an online shop).*
- ***Composite UI*** *mixing information from multiple sources.*
- ***Mashup pages*** *if the information is external, like Google Maps, Twitter activity and so on.*
- ***Good old queries and store procedures*** *are still valid, they're just a little uncool if your team is strong on CQRS and Event Sourcing.*
- *...*

KICKING-OFF OUR MODEL

Apparently, our goal is simple: just connecting events from the starting point to the final state, using a consistent sequence of commands, systems, events, policies and read models.



However there are multiple strategies here, let's them.

START FROM THE BEGINNING

Find the starting point, and build a sequence up to the desired final state.

- 🙂: easy to tell the story from the beginning.
- 😥: easy to get swamped with alternatives.
- 😊: easier to discover unexpected outcomes.

START FROM THE END

Start from the desired outcome, and build your way back to the starting point, respecting the colour grammar.

- 😎: really lean implementation, usually with no redundant steps.
- 😐: a little tough on the beginners.
- 😕: not so much discovery.

BRAINSTORM FIRST

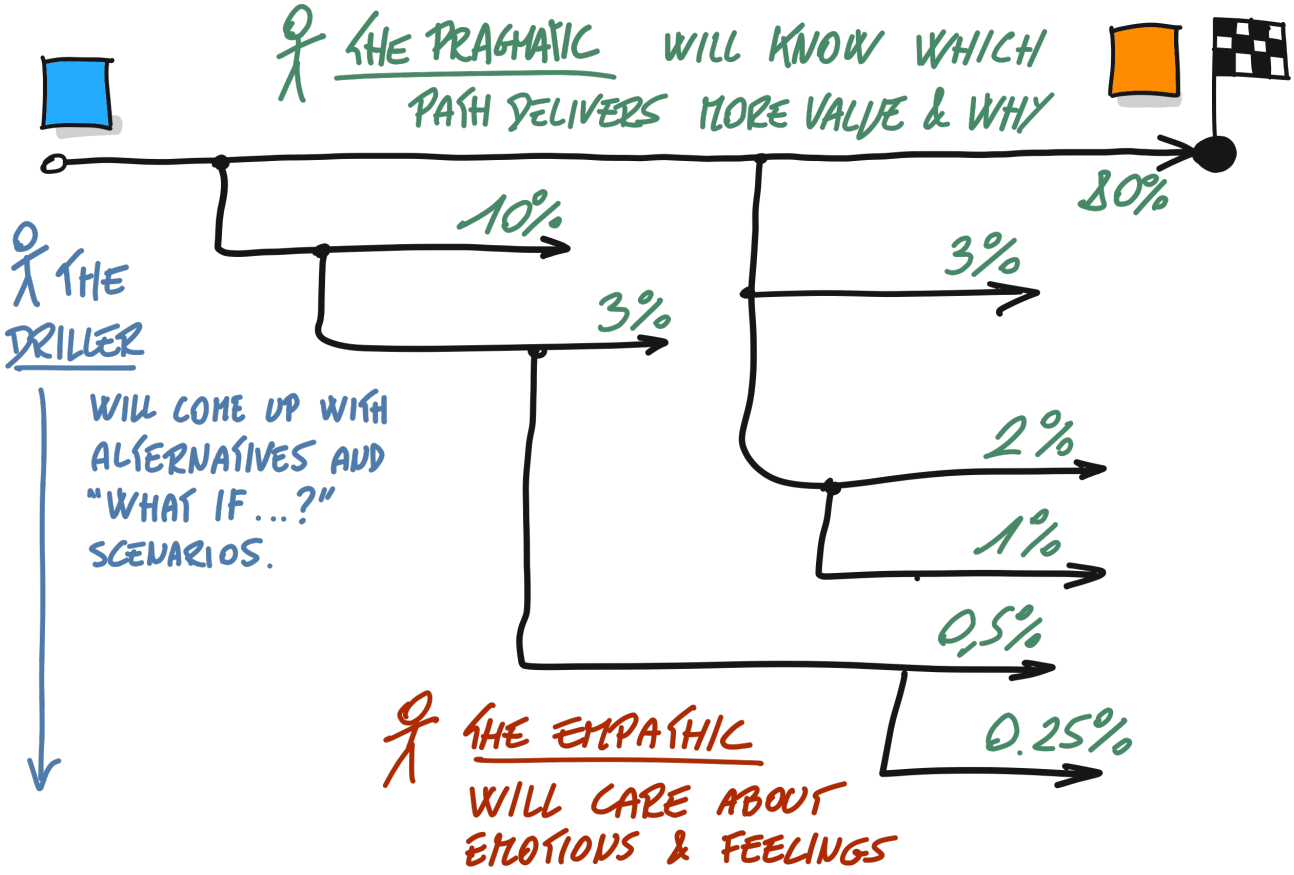
Start by exploring the process area with events first, then find a way to connect them.

- 🤯: highest discovery (including useless things)
- 😝: tough to sort out.

Balancing perspectives

Multiple perspectives in the room will bring salt to the discussion. Here you can se a possible map of the conflicting perspectives, attached to common behavioural stereotypes.

Nobody is right upfront, but every contribution is necessary to come up with good model.



Sometimes, some alternatives are not going to happen anytime soon, so it would be ok to dismiss them.

Sometimes, it's ok to be pragmatic about implementing or not implementing stuff, according to the numbers. But we need to explore in order to make an informed choice.

Some times are not worth exploring... but we're making a risky call. Risk is fine, as long as we're making explicit calls and we have similar #SkinInTheGame.

Modelling Conversational Systems

A common trap, while modelling processes lies in the boundary line between a *fuzzy* and *conversational media* towards a more mechanical executable part.

Multiple output events

A phone conversation may actually produce a set of events non necessarily captured by the following business flow. A bad salesperson can produce a **Customer Contacted**, **Offer Proposed**, **Customer Insulted** and **Customer Lost** even in the same sentence.

In general, a conversational step may trigger multiple outcomes, and it's a good idea to visualise them. You may also want to explicitly embrace a *downstream perspective*: in the ticket cancellation exercise, the process for freeing up the cancelled seat started once we were sure that the participant was not coming, possibly hinting for a **Cancellation Confirmed**.

Focusing on the termination condition

Conversational media can also produce a lot of irrelevant events: some people have the tendency to model a conversation like a sequence of scripted steps: **Option A Proposed**, **Option A Rejected**, **Option B Proposed**, and so on.

This is usually not really interesting, it is sometimes *creepy*, and usually tends to use a lot of precious space.

The usual advice is to focus on the termination condition of the conversation: *What event will complete the conversation?* In our cancellation scenario it's probably something like **Agreement Reached** (no further conversation is necessary).

The type of agreement will then be an interesting information for the following steps.

Event Granularity

You may find yourself asking: *"Should we model one big event - like Agreement Reached - and dig into the details? Or emit multiple fine-grained events?"*

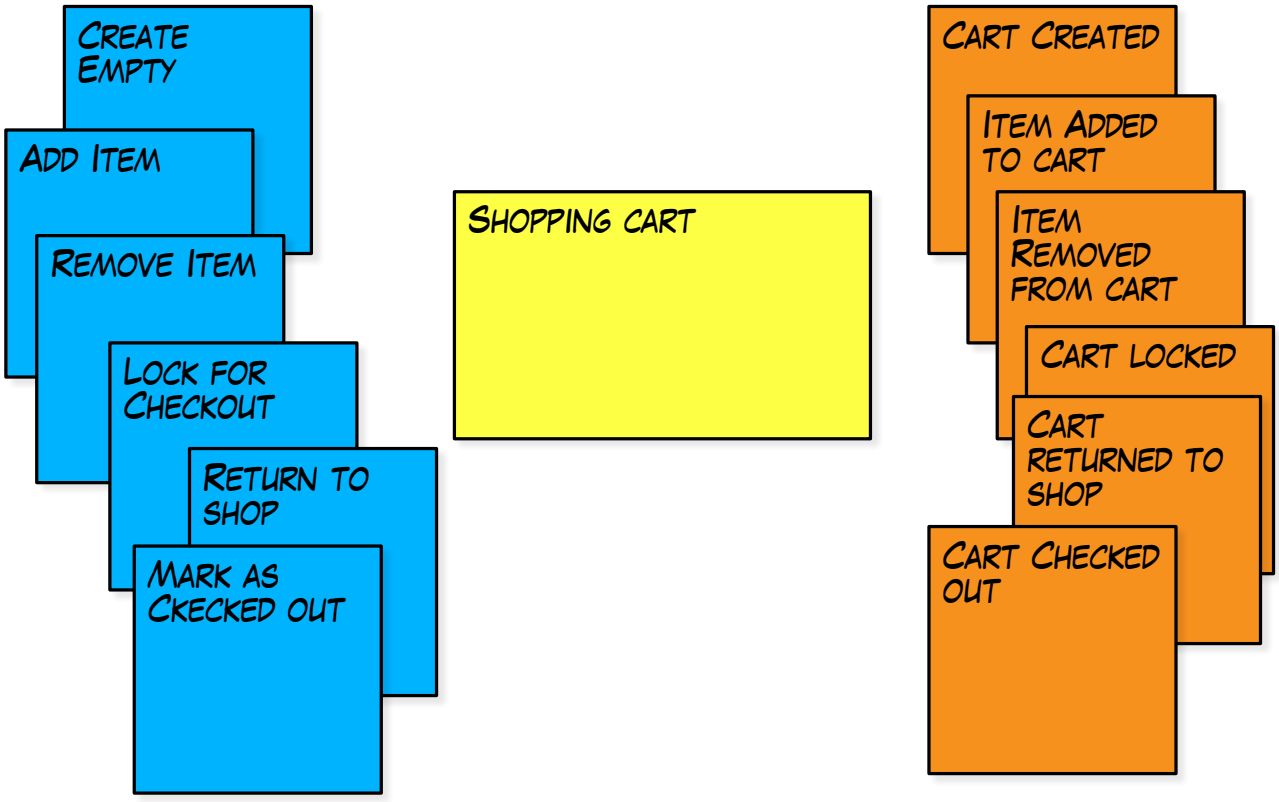
Well... it depends! But you have emitters and listeners modelling together, so finding an agreement shouldn't be hard. ;-)

SOFTWARE DESIGN

The dynamics for a software design workshop build on top of what we've already experienced for process design. The main difference is that we're now opening the box and looking at the internals of our software components.

"THE AGGREGATE"

I kept the name 'aggregate' mostly for historical reasons, EventStorming was born around Domain-Driven Design and Aggregates, so it makes little sense to look for another name. Aggregates are unit of transactional consistency within a software system. More interestingly they are little state machines that expose consistent behaviour.



My EventStorming definition is ***aggregates are the yellow sticky note between the blue commands and the orange events***.

We shouldn't be too much worried about the implementation details: aggregates boundaries are more business boundaries than technical ones.

PATTERNS AND ANTIPATTERNS

SOLVE EVERYTHING -> RUSH TO THE GOAL

One of your biggest impediments would be to try to solve every problem at once, in a scenario that starts apparently linear, but that quickly becomes a tree of possible branches. Solving everything will be hardly achievable if we consider a few special condition:

1) We're probably modelling **in an area of high complexity**,
2) There's already a "working solution" meaning that somebody did their best in the past, but it's no longer enough in different working condition.

Put in another way, *finding a better solution won't be a 5 minutes trip*.

However, half of the people in your team would be *professional what if machines*, and you want to be sure to be able to deliver without being swamped by complexity.

The recommended strategy is to try reach the end of the process *no matter what*. Every time a new alternative appears in our exploration, we can use a **hot spot** to mark the path we're not exploring right now.

This way we'll not be forgetting the problem. While we'll still be able to deliver the simplest path while exploring more complex alternatives.

Keep it simple -> Raise the bar

This is another unconventional strategy that might not work in the general purpose world, but makes a lot of sense in a high-complexity high-value scenario.

When dealing with complex problems, learning the intricacies is paramount. But *a simple use case won't be enough to learn the complexity with the needed degree of confidence*. A simple solution might work on the base scenario, but more complex scenario can make the simple solution collapse under its weight.

In general we'd like to include also corner cases in the exploration, better if driven by examples[1]. As we've seen in Rush to the Goal, we can't deal with all the complexity at once, but pretending corner cases didn't happen is not a good modelling strategy here.

Keep it Simple -> Listen and Track Terms

A common dentition would be to *try to solve the modelling problem with the smallest problem of moving parts*.

Nothing wrong with this strategy: is good common sense, except that it probably didn't work well with the problem we're dealing with right now, especially if we're on the bottleneck of a business flow, that was originated by a legacy monolith.

However, simply *listening* to the words used during in the discussion can provide a lot of clues on the actual complexity of the underlying problem: during the ticketing exercise I collected quite a few words: Purchase, Order, Ticket, Reservation, Seat, Spot, Payment, Training, and so on.

Some of them will probably *be really synonyms*, but some others aren't. Listing the words and playing a little with them

Watch Out for Symmetries

Software design will be a playing ground for competing forces in visualisation: usually the business stakeholder prefer a timeline representation, since it better matches the storytelling, while developers will need to see the emerging software shape in terms of *roles and responsibilities*, but clustering possible commands and event for the same aggregate would possibly break the timeline.

---

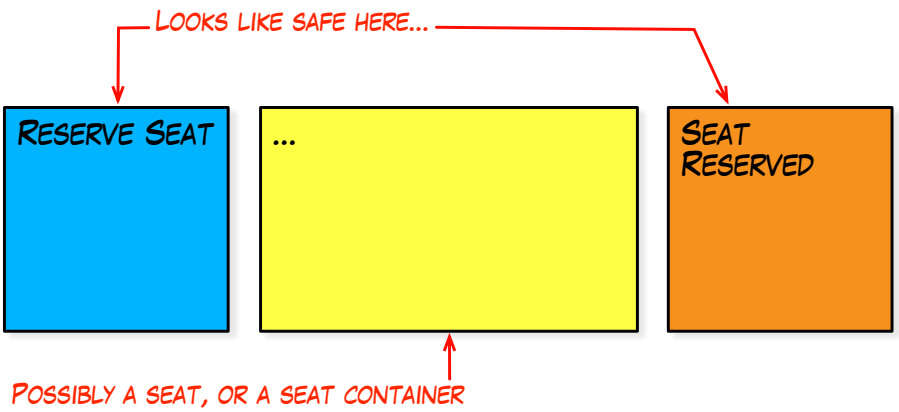
[1] *I like to collect stories about the most extreme situations an organisation faced, and check whether we're able to support them with our model. Extreme situations are not exceptions, if our model is flexible enough.*

In general, I try to keep the model readable for the business as much as I can, but this means that I need to find different clues to check that I am doing a good job in modelling. This is where symmetries come into play.

Cause and effect symmetries

Some commands need to have an effect that looks exactly like the effect of the command. I am actively looking for this language consistency.

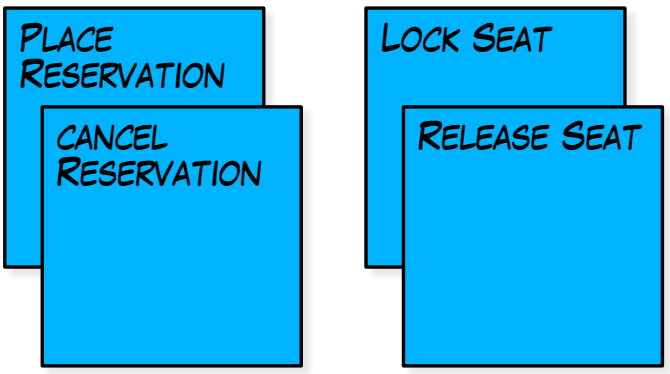
Unfortunately, when this modelling step becomes obvious, developers tend to get annoyed, because the problem is no longer matching their brain capacity.



But... not every step is complex. Actually, the majority of them is probably trivial, and there's nothing wrong about it.

Complementary commands

Usually a command has a reverse/complementary action, like *open* and *close*. It's not guaranteed, and some domain actions are irreversible (you would like an *untweet* sometimes). However looking for reverse and complementary actions is usually a good way to get into a more consistent model.



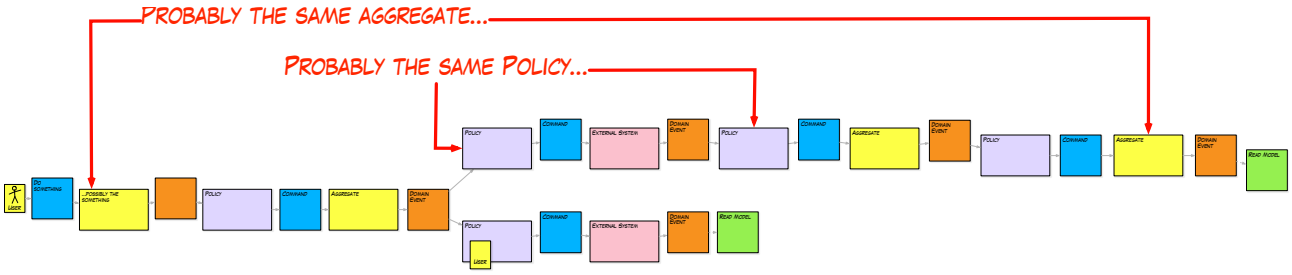
Once more, I usually say it out loud, checking whether I sound stupid in the process. This way I usually find more appropriate verbs, or I simply trigger better insights.

PLACE RESERVATION and CANCEL RESERVATIOn sounds like a better wording compared to LOCK SEAT and RELEASE SEAT (I picture myself locking seat in real life, and yes, it *does* feel a little stupid).

Open-Close Symmetries

This is something usually visible at the whole process size: it still builds on the need for complementary actions, but it's especially useful on the main aggregate.

It won't be possible to read the picture below... and that doesn't matter, because sometimes the information is in the shape, or in the colour pattern.



There's usually something that we open/start at the beginning, that needs to be closed at the end. If I don't see the same aggregate at the end there might be something left in an inconsistent state.

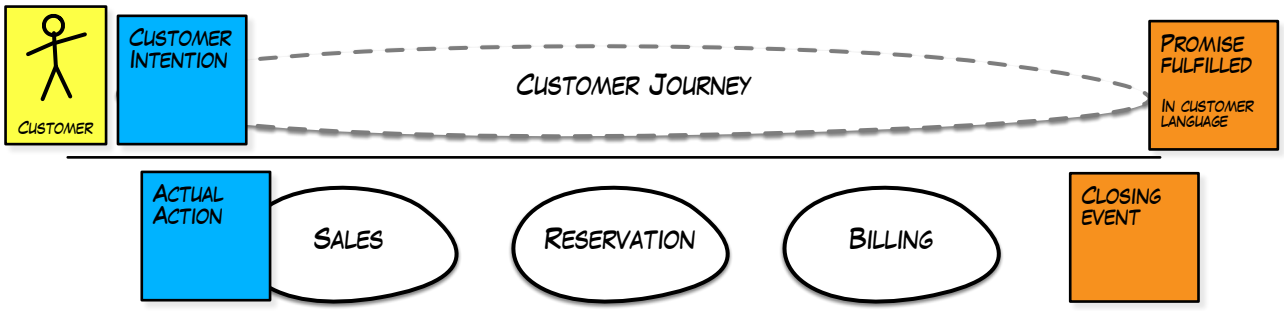
Some policies follow the same pattern, especially if they're *wrapping* an external component, or holding some story.

THE LANGUAGE MISMATCH

Our search for symmetry around the aggregates tend to fail miserably once we hit the borders. The BUY TICKET command doesn't really match with the target and the resulting immediate event.

The main reason is that the **Customer** is very likely to speak a different language. It doesn't happen with every user: *internal users* are usually the people we can invite in a workshop and whose language we can interact with, while external users are usually only explored by UX experts with tools like Customer Journeys and Personas.

As a result, customer language is seldom mapped as a separate bounded context. But it is a different one, with different rules: BUY TICKET needs to be a good call to action, while the most honest underlying command INITIATE PURCHASE PROCESS, doesn't.



I usually end up representing the translation happening between the customer world and the complexity of the actual implementation, by putting two parallel swim lanes.

It's interesting to notice that symmetry is preserved at the edges too: a customer wants to buy a ticket, and a ticket would be the reward at the end of the purchase process. Providing a RESERVATION to the customer, after promising a TICKET would have left a feeling of unfulfilled promise.

Format Cheat Sheet

EventStorming MasterClass

Big Picture

1. Invite the Right People

1.1. Invitations won't be perfect

1.2. Invite the ones who cares

2. Set up the Room

2.1. No big Tables, poisonous seats aside

2.2. Unlimited modelling tools (surface, stickies, markers, space)

---

3. Explore the whole business line with Domain Events along a timeline

3.1. Ice breakers

3.2. Ignite chaos / break anti patterns

4. Enforce the timeline

4.1. Breaking clusters & Global consistency

4.2. Triggering conversations

4.3. Emergent structure

5. Add Users and External Systems

5.1. explore motivation

5.2. Explore near the boundaries

6. Explicit walkthrough

6.1. Deeper consistency

6.2. No blind spots

6.3. Reverse narrative

7. Value / Problem & Opportunities

8. Pick the problem

8.1. Consensus or voting

8.2. Managing the consequences

---

9. Wrap up

9.1. Pictures and special folding

9.2. Call to action

9.3. Leave it around

9.4. Aftermaths

9.5. Homework (Bounded Contexts)

> **Facilitator:** Capture definitions, asking the obvious

> **Facilitator:** Highlight Hot Spots on interesting conversations

> **Facilitator:** Make sure model is in sync with storytelling

> **Facilitator:** Add more space when needed

Process Modelling

1. Invite the Right People
   1.1. only the needed few
2. Set up the Room
   2.1. No big Tables, poisonous seats aside
   2.2. Unlimited modelling tools (surface, stickies, markers, space)

- -

3. Define exploration scope (...or challenge it!)
   3.1. We are where it matters (Big Picture's output)
   3.2. Choose beginning and the end
4. Explore the process with Domain Events
   4.1. Hard copy or restart?
   4.2. Capture scenarios.
5. Where are Events coming from?
   5.1. User Decisions -> Commands
   5.2. External Systems
   5.3. Time
   5.4. Cascading reactions -> Policies
6. Enforce process consistency
   6.1. Pick one scenario
   6.2. model it end to end
   6.3. Pick a more complex one -> Repeat
7. Explore Value & Motivation
   7.1. Money is not the only currency
   7.2. Make dilemmas visible
8. Are we done?
   8.1. ROI - Based selection.
   8.2. Are we happy?

Facilitator:

Incremental notation with newcomers

Facilitator:

Picture that Explains Everything is your best

Facilitator:

Only decide on visible alternatives.

- -

9. Wrap up: no, don't
   9.1. Just code it!

Software Modelling

1. Invite the Right People

1.1. Mostly technical + Product Owner

2. Set up the Room

2.1. No big Tables, poisonous seats aside

2.2. Unlimited modelling tools (surface, stickies, markers, space)

---

3. Define exploration scope

3.1. Clarify the set of features to support

3.2. redefine beginning and the end (Given, When, Then...)

4. Explore the process with Domain Events

4.1. Hard copy or restart?

4.2. Capture scenarios.

5. Enforce process consistency

5.1. Pick one scenario

5.2. model it end to end

5.3. Pick a more complex one -> Repeat

6. Cluster around state

6.1. Postpone naming

6.2. Look for state machines

7. Shake the system

7.1. Can we manage more complex scenarios

7.2. Bruce Lee's attitude: "One more requirement"

8. Are we done?

8.1. ROI - Based selection.

8.2. Are we happy?

---

9. Wrap up: no, don't

9.1. Just code it!

9.2. Enjoy the whirlpool!

Facilitator:

Incremental notation with newcomers

Facilitator:

Picture that Explains Everything is your best friend

Facilitator:

Add more space when needed

Facilitator:

Only decide on visible alternatives.