Experiencing Domain-Driven Design
Workshop with Mathias Verraes

http://value-object.com/

Handouts licensed only for participants of the "Experiencing Domain-Driven Design" workshop by Mathias Verraes

Version **2016-04-15**
Latest version:
http://verraes.net/workshops/handouts/expddd.pdf

Contact:
mathias@verraes.net @mathiasverraes verraes.net

Contents

Intro to Domain-Driven Design

Bounded Set vs Centered Set

A Bounded Set are static, they have a clear boundary, and elements in them are uniform. For example, any fruit that satisfies certain characteristics is an apple, and all fruits are not.

Bounded Set vs Centered Set

Domain-Driven Design on the other hand is a Centered Set. It consists of principles and methodologies, but they are not clearly bound. Applying more of the principles to your work, makes you "more DDD", but there is no single delineation. This was chosen on purpose: it allows DDD to evolve and question its own fundamentals.[1]

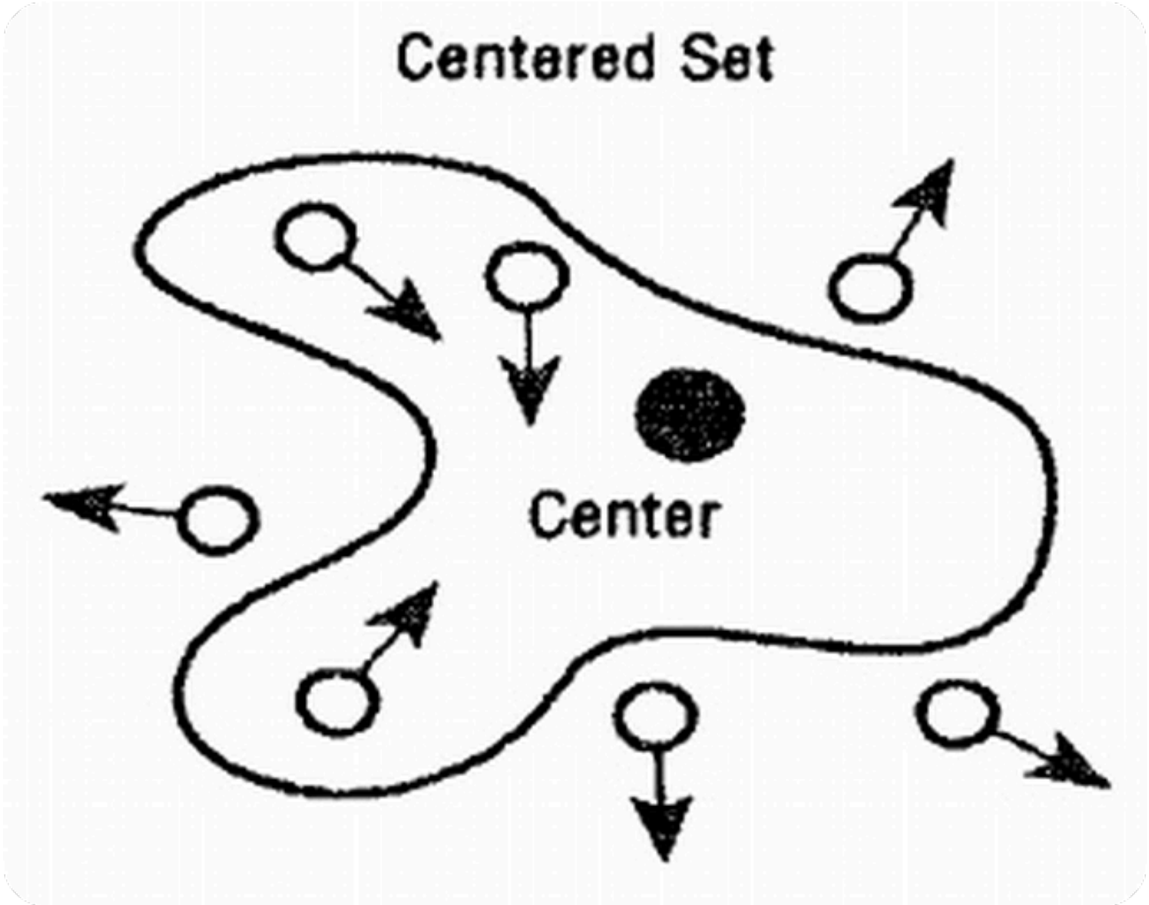
[1] "Rethinking System Analysis and Design" — Gerald M. Weinberg

Bounded Set vs Centered Set

An example of Domain-Driven Design ability to evolve and adapt, are Domain Events. They were not mentioned in the blue book[2], but are now considered by many an essential aspect of DDD. Similarly, CQRS, Event Sourcing, Event Storming, … are newer additions, and DDD has branched into Functional Programming and other paradigms.

[2] Domain-Driven Design, Eric Evans 2004

Bounded Set vs Centered Set



DDD Principles

1. **Design Consciously**
2. Grasp the Domain
3. Obsess over Language
4. Iterate and Synchronise Models
5. Worry about Boundaries
6. Accept Complexity

1. Design Consciously

All software has a design. There is no option not to have design. By not thinking about design, we often end up with bad design.

Good design balances **purpose** and **tradeoffs** in a **context**. We need to be aware of all three, make them explicit, and evolve the design to find an optimal balance.

1. Design Consciously

Agile shifted the mindset from Big Design Up Front (BDUF) to No Design Up Front.

Domain-Driven Design prefers lots of small design up front.

1. Design Consciously

"Perfect" design scares away others. As the design can not be perfect anyway, the appearance of perfection blocks evolution of the design.

Small bits of imperfect design attracts collaboration.

DDD Principles

1. Design Consciously
2. **Grasp the Domain**
3. Obsess over Language
4. Iterate and Synchronise Models
5. Worry about Boundaries
6. Accept Complexity

2. Grasp the Domain

It is not the desires of the customer that get built into software, or the insights of the analysts. It's the understanding of the developers that ultimately becomes the software.

Creating software is a learning activity, with working code as a side effect.

2. Grasp the Domain

Although in many people intuitively understand that developers with domain knowledge build better software, in practice it is rare for developers to spend a lot of effort learning the domain. They are already burdened with creating, running, and supporting the software, and need to continuously lear fast evolving technologies.

2. Grasp the Domain

There are also institional blockers. Developers are "shielded" from the customers and the domain, and only get second- or third-hand information from other roles, such as analysts and domain experts. Often, developers are given features to build, but lack context to understand the bigger picture. This leads to inflexible, incomplete, or even faulty models, and ad hoc "spaghetti" code that doesn't communicate intent.

2. Grasp the Domain

Often the best developers are assigned to highly technical tasks, leaving the Core Domain to juniors. DDD advocates to invert this trend.

2. Grasp the Domain

Grasping the Domain involves more than just acquiring superficial knowledge. Ideally, developers spend a lot of time **interviewing Domain Experts**, for example using techniques like **Event Storming**.

2. Grasp the Domain

A useful questions is asking how people in the domain solve problems. Often they have done this for years (or even centuries in mature domains) without software.

2. Grasp the Domain

Furthermore, the developers can do their own research, read specialised literature about the domain, visit the work place and talk to the end users. Some companies go as far as to having the developers do the actual work they will be writing software for.

2. Grasp the Domain

Over time, the developers become so knowledgeable in the domain, that they require fewer assistance from the domain experts to make decisions. Thanks to their analytical skills and modelling efforts, the developers learn to **see opportunity in the domain**, that was previously invisible. The mindset shifts: instead of just a cost, **developers become an appreciated asset to the business**.

2. Grasp the Domain

"Knowledge crunching is an exploration, and you can’t know where you will end up."

— Eric Evans

DDD Principles

1. Design Consciously
2. Grasp the Domain
3. **Obsess over Language**
4. Iterate and Synchronise Models
5. Worry about Boundaries
6. Accept Complexity

3. Obsess over Language

The primary means for acquiring domain knowledge is language: visual, textual, verbal, …

In many environments, the domain language gets translated as it moves through different roles in the organisation, until what ends up in the code, is highly technical and bears no resemblance to the original domain.

3. Obsess over Language

These translations bring an important cost. When changes to the software are required, the developers need to reverse engineer the code to figure out not just what it is doing, but also why. Often the original developers or domain experts are no longer available.

Translation from business language to technical language is lossy.

3. Obsess over Language

DDD proposes to grow a Ubiquitous Language. This is a **shared language between all stakeholders**: business, analysts, testers, documentation, UI, developers, ...

3. Obsess over Language

If everybody shares the same language, and if the model and the code reflect this, something important happens: a lot of energy that went into dealing with translation, can now be refocused on the domain.

3. Obsess over Language

The domain is our primary source of language, but it's not all-knowing. Human language is messy, full of nuance, duplication, and redundancy. The context defines the meaning, but it is implicit, hidden like the mass of iceberg. The language is not formal. It has grown historically over time, to fill need. Many concepts don't have a name, indeed because there never was a need to name them.

3. Obsess over Language

A Ubiquitous Language that only consists of concepts that are explicit in the domain, cannot be complete. On the other hand, **forcing a language on the stakeholders, is often tricky**. It might not stick, or worse, it might be used differently than you intended.

3. Obsess over Language

We make concepts explicit, name them, define them, and disambiguate them. We make compromises: if the wrong definition is universal, we should stick to it. If we need to introduce a new term to make our model work, that's ok as well. Most **languages grow organically**, whether we like that or not. What we can do is water it, guide it, weed it, and enjoy the fruits.

3. Obsess over Language

A Ubiquitous Language is **not a dictionary** of terms. A language has a grammar. It has words, phrases, sentences, scenarios, and dialogues. A phrase can have all the right ingredients, combined in a syntactically correct way, yet still not be idiomatic.

A change in the language is a change in the model.

DDD Principles

1. Design Consciously
2. Grasp the Domain
3. Obsess over Language
4. **Iterate and Synchronise Models**
5. Worry about Boundaries
6. Accept Complexity

4. Iterate and Synchronise Models

→ **Domain**: The problem space. The world of the business.

→ **Domain Model**: The solution space. Our choices and abstractions to represent solutions to business problems.

4. Iterate and Synchronise Models

"All models are wrong, but some are useful."
— George E.P. Box

4. Iterate and Synchronise Models

We judge models not by how elegant they are, or how much they reflect reality. We judge them by how useful they are to solve problems.

In one context, an elaborate model might be useful, in another some simple reductionist abstractions might be all we need.

4. Iterate and Synchronise Models

The usefulness of a model depends on who's using it. As a customer, I prefer a cardboard model of my house, but as a builder, I prefer a blueprint.

Having multiple representations (visual, text, diagrams, code...) of the same model helps to communicate and weed out nuances.

4. Iterate and Synchronise Models

An idea that was radical at the time Domain-Driven Design was published, is to eliminate the separation between an Analysis Model and an Implementation Model.

4. Iterate and Synchronise Models

"Domain-driven design calls for a model that doesn't just aid early analysis but is the very foundation of the design."

— Eric Evans

4. Iterate and Synchronise Models

When something changes in the understanding during analysis, implementation must be adapted.

When new insights are found during implementation, analysis models must be adapted.

4. Iterate and Synchronise Models

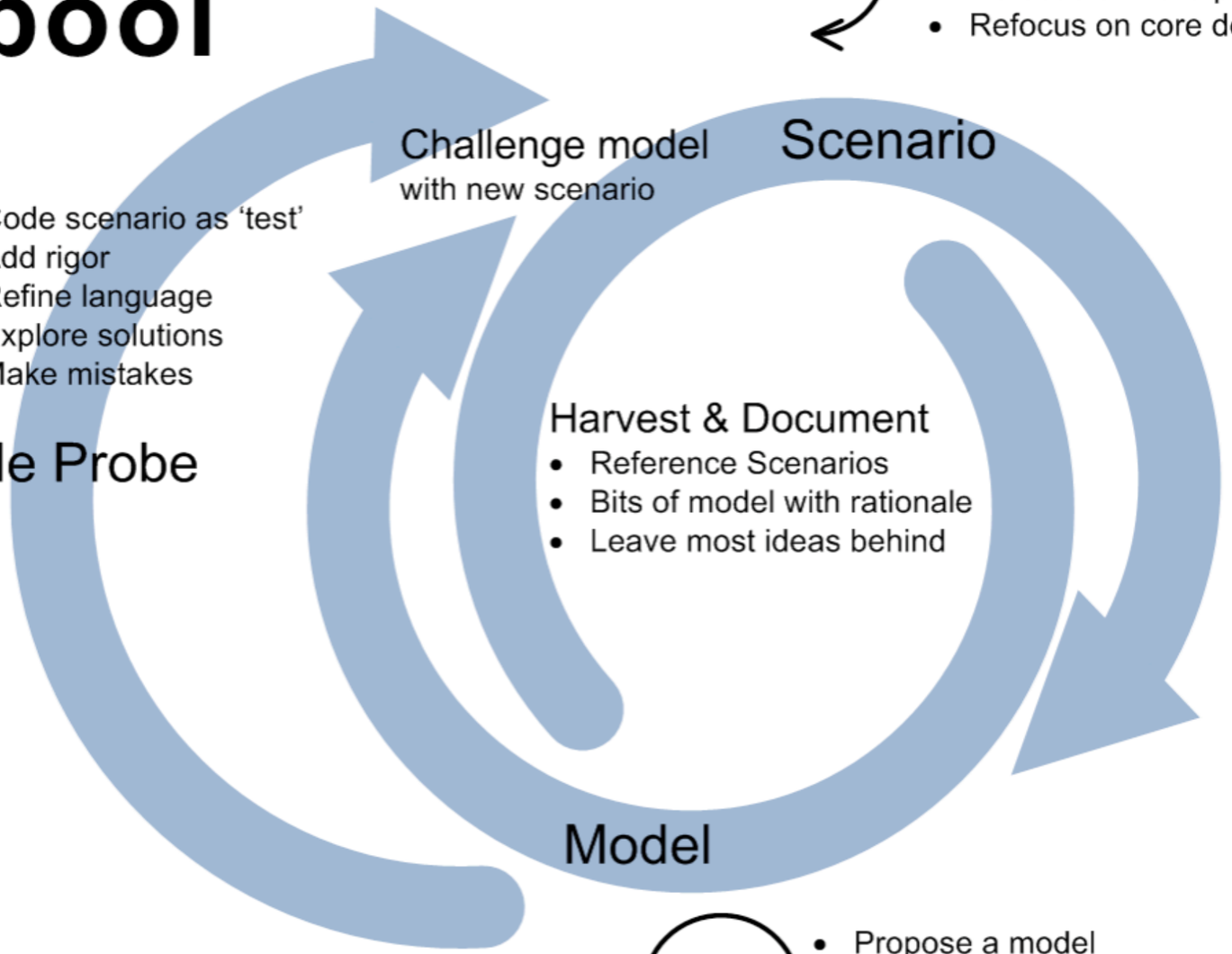
Model Exploration Whirlpool[3] (see next slide)

[3] Model Exploration Whirlpool

Model Exploration

Whirlpool

DRAFT 0.3



- Tell us a story.
- Flesh it out.
- Refocus on hard part.
- Refocus on core domain

- Code scenario as ‘test’
- Add rigor
- Refine language
- Explore solutions
- Make mistakes

- Propose a model
- Walkthrough states
- Walkthrough solutions
- Explore language
- Make mistakes

DDD Principles

1. Design Consciously
2. Grasp the Domain
3. Obsess over Language
4. Iterate and Synchronise Models
5. **Worry about Boundaries**
6. Accept Complexity

5. Worry about Boundaries

Attempting to build a single Ubiquitous Language and a single enterprise-wide model, often leads to problems:

→ High coupling

→ Reductionist models in some places

→ Overly complex models in other places

→ Rigid models that do not age well

5. Worry about Boundaries

Domain-Driven Design aims to find smaller boundaries.

→ Problems become smaller, without losing essential complexity.

→ Important problems can be solved in isolation, receive more attention than other areas.

→ Parts of the system can be more easily replaced.

5. Worry about Boundaries

In Domain-Driven Design, we look for boundaries everywhere:

→ in the small, eg well-defined objects, transaction boundaries, etc

→ in the large, eg Bounded Contexts

5. Worry about Boundaries

A Bounded Context is an explicit context in which a model applies. Ideally, Bounded Contexts coincide with the Domains we identified, but other constraints (such as existing legacy) may have an effect as well.

5. Worry about Boundaries

A model and its Ubiquitous Language should be consistent within a Context.

There may be other models and Ubiquitous Languages in the organisation, with overlapping concepts and terminology. Because these models and languages are in separate, explicitly delineated contexts, they do not conflict.

DDD Principles

1. Design Consciously
2. Grasp the Domain
3. Obsess over Language
4. Iterate and Synchronise Models
5. Worry about Boundaries
6. **Accept Complexity**

6. Accept Complexity

All too often we attempt to hide complexity. In search of simpler abstractions, we fall in the trap of reductionism: lossy models that do not capture the real language and problems.

6. Accept Complexity

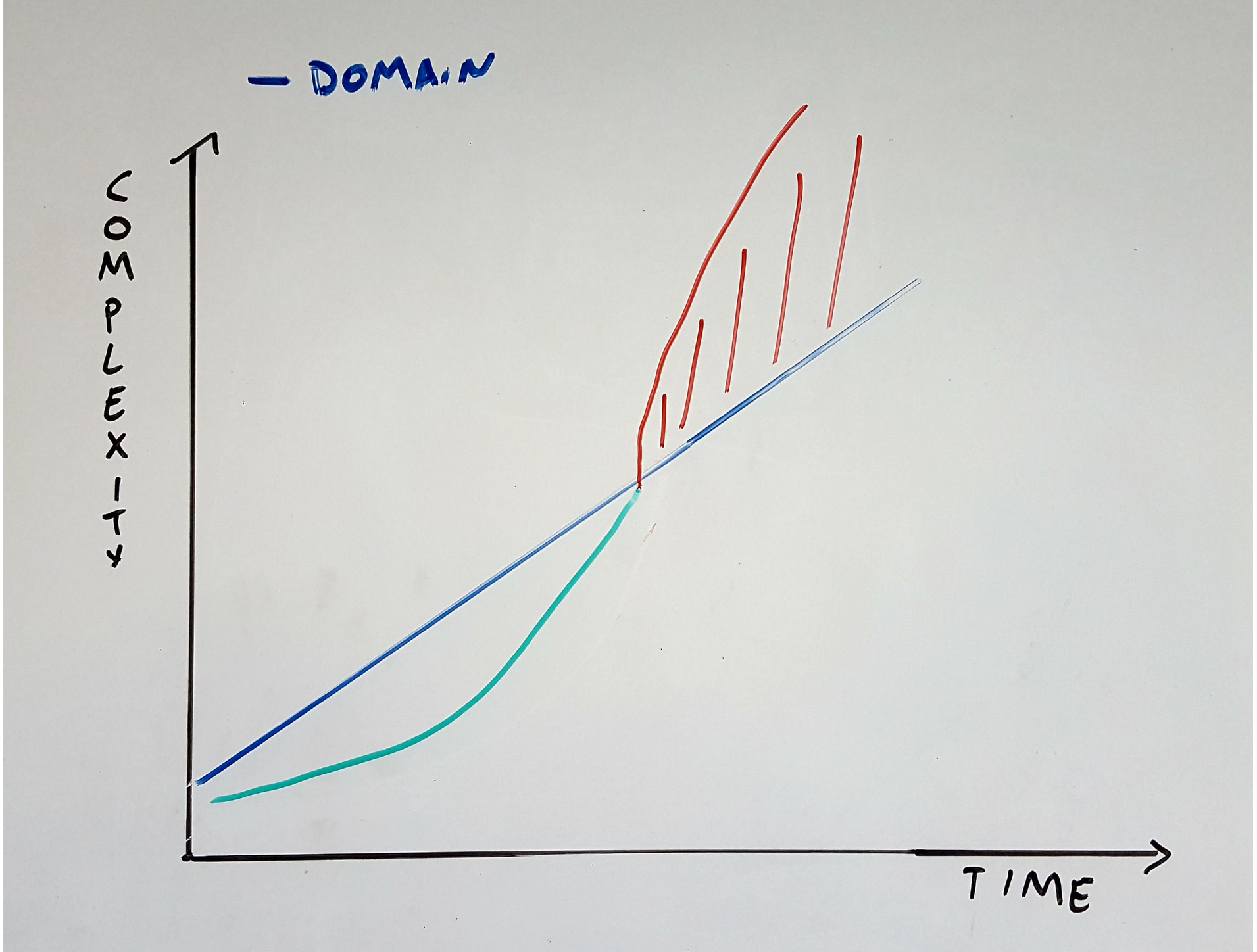
→ Essential Complexity[4]: Inherent in the problem or domain, cannot be eliminated through technology or methodology.

→ Accidental Complexity: Unnecessary compelexity introduced by our technology, methodology, (lack of) design choices.

[4] The Mythical Man-Month, Fred Brooks

6. Accept Complexity

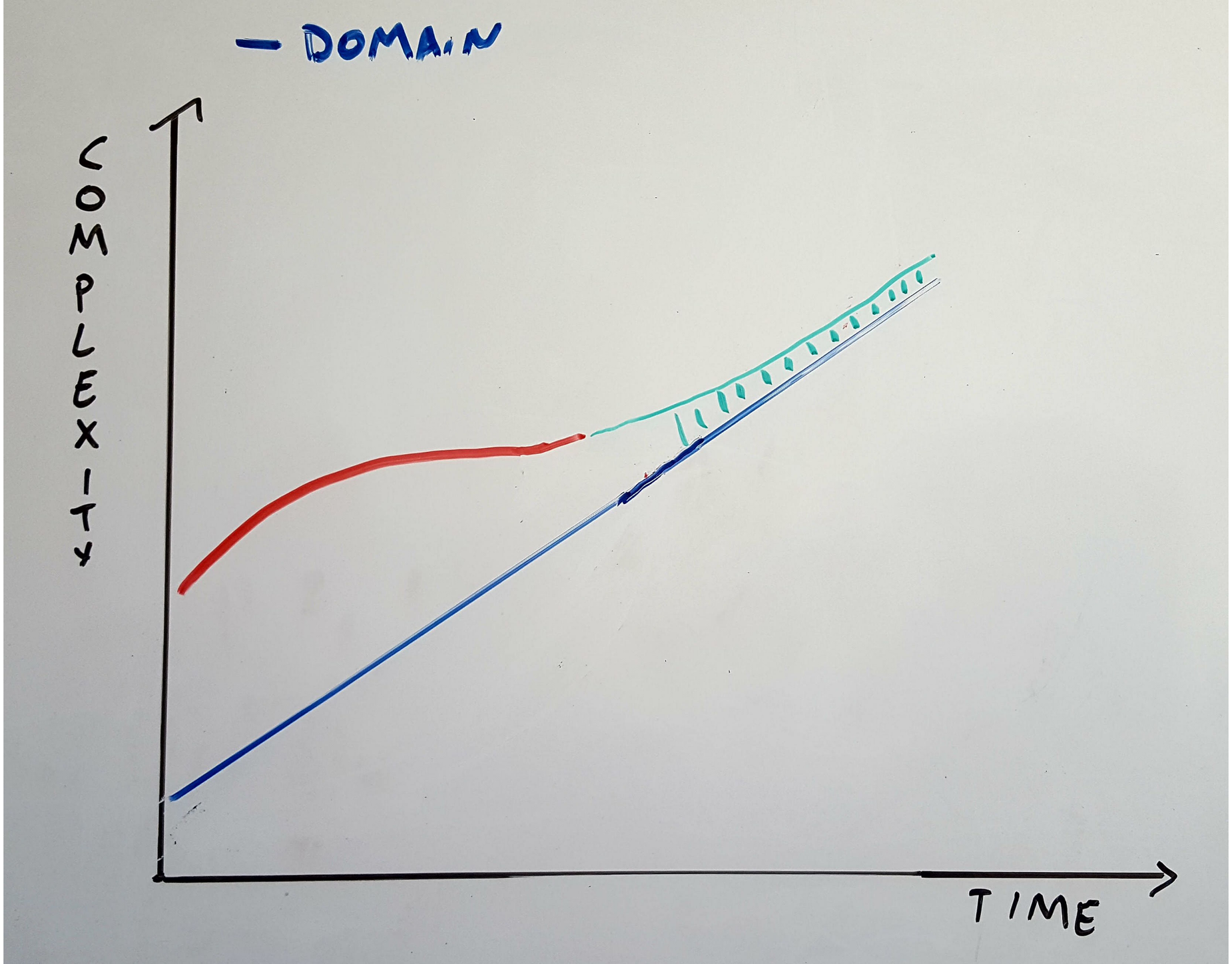
Reductionist design styles, such as CRUD, have a low initial cost, but as essential complexity grows, they can't cope and cause high essential complexity.


— DOMAIN
COMPLEXITY
TIME

6. Accept Complexity

Explicit design styles like Domain-driven Design may bring a higher (learning) cost for small projects, but as essential complexity grows, the growth of accidental complexity slows and equals growth in essential complexity.

(Some accidental complexity is unavoidable.)


— DOMAIN
COMPLEXITY
TIME

6. Accept Complexity

Make the implicit explicit.

The reason DDD deals with complexity better, is by makin git very visible and explicit. This way, it can be reasoned about, and the model can change as needed.

6. Accept Complexity

In implicit design styles, there is no model reflecting the complexity. Changes become harder, so workarounds and hacks are employed, making changes even harder, etc.


Event
Storming

Event Storming

The grand dichotomy of systems[1]:
Things vs Processes

Most modelling efforts focus on things/artifacts/state/structure...

[1] "Rethinking System Analysis and Design" — Gerald M. Weinberg

Event Storming

Temporal modelling on the other hand focusses on events, processes, relation in time, cause and effects, behaviour, variability in paths (happy path, divergence, …)

Event Storming

A collaborative method to visualise complex business processes as events over time.

It is found to aid communication between business and development teams.

Big Picture Event Storming

Exploration — Ubiquitous Language — Processes …

Typically used at the kick off of a new project, when onboarding team members, make strategic decisions, find bottlenecks, …

Big Picture Event Storming

Involve customers, domain experts, analysts, … as well developers, testers, UI…

Design Level Event Storming

Precision — Constraints — Processes — Aggregates — Dependencies — Messaging — Event Sourcing — Bounded Contexts — Concurrency — Race Conditions — …

Design Level Event Storming

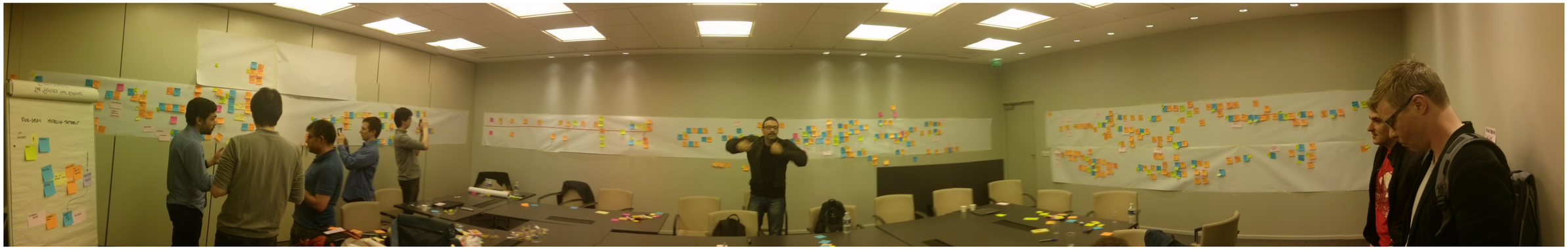
Typically used regularly throughout the project, to explore and design small parts of the model (eg per story).

Design Level Event Storming

Typically only technical roles are involved. Often happens in an ad hoc fashion, as an alternative/addition to drawing other model diagrams.

Event Storming

Create a large modelling space. Don't allow the physical space to constrain the logical solution.



Event Storming

Get attendees to stand up. It increases blood flow, creative, attention span.

Event Storming

When facilitating[5], mind the body language of the participants. Keep sure they are entertained. Avoid technical terminology. Make sure all participants are heard.

[5] http://verraes.net/2013/08/facilitating-event-storming/

Event Storming

Avoid upfront consensus. Allow chaos. Be mindful of divergent or conflicting ideas. Visualise all opinions, ask for more options.

Event Storming

Stick to a consistent color coding style. Make a legend when creating new color codes or symbols. Prefer visual elements, symbols, or icons, where possible.

Event Storming

Put an arrow pointing right to indicate the direction of time. Hang the first event. **Domain Events**:

→ Something that happened in the past

→ That is relevant to the business

→ Expressed as past tense

eg ** Order Was Paid**

Event Storming

Prefer using past tense verbs: **Order Was Paid** is more expressive than **Order Paid** or **Order Payment**.

Using natural languages enables a shared Ubiquitous Language, and later enables direct back and forth translation from the domain mdoels to code and tests. (eg Given-When-Then scenarios).

Event Storming

Starting with the first event, keep asking questions:

→ What happens before? What happens after?

→ What happens in between?

→ Are there any consequences?

→ Does the business care about this?

Event Storming

After a while, start adding **Commands**:

→ An instruction that someone sends to the system

→ Expressed as imperative.

eg **Pay For Order**

Event Storming

To find **Commands**, ask:

→ What causes this event?

→ What does the user want the system to do?

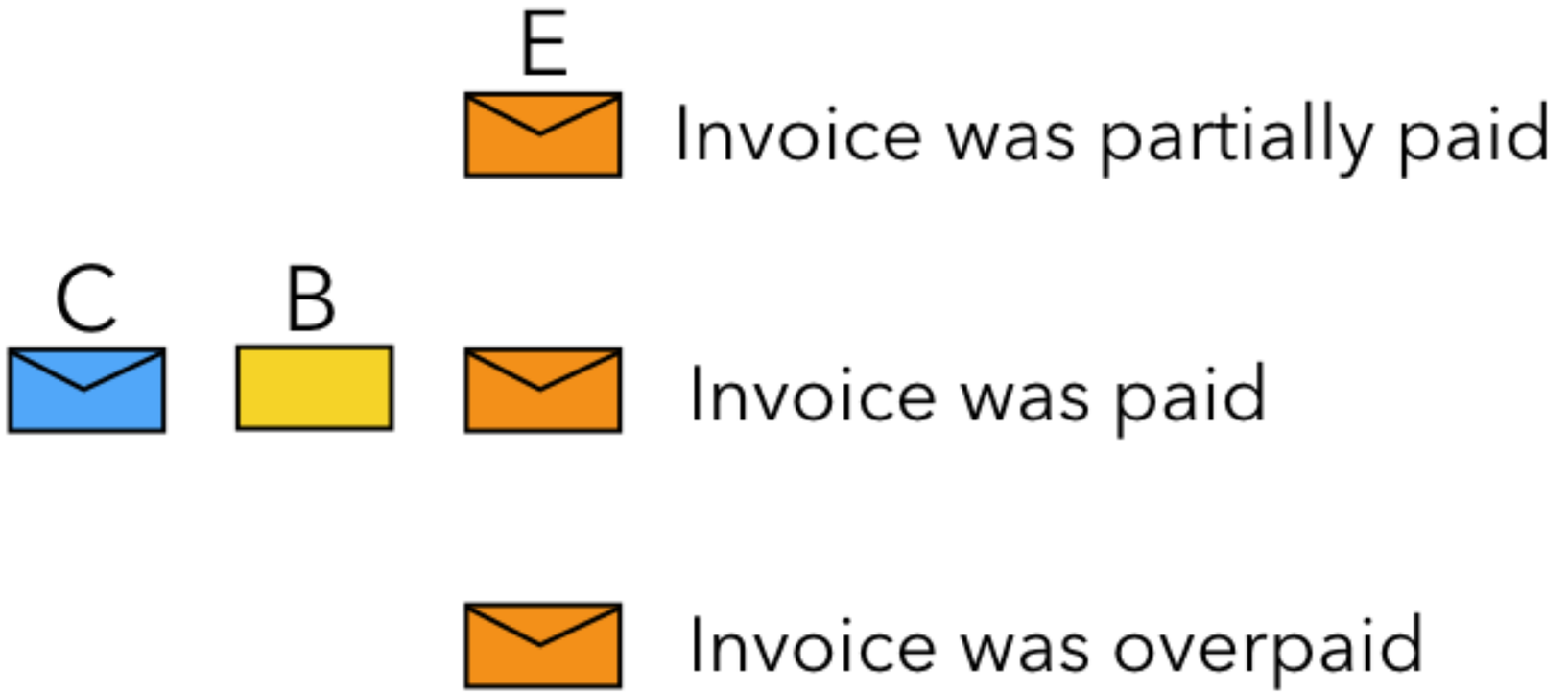
Event Storming

If it's ambiguous, identify **Roles** or **Actors** who send the Commands.

Identify Commands that might be sent from another system, as opposed to a human user.

Event Storming

Identify **Invariants**, **Constraints**, or **Business Rules**.
(The name Business Rule seems to work better with non-technical people.)



Paid amount < Invoice amount

Paid amount = Invoice amount

Event Storming

Use either natural language, or mathematical notation:

"The paid amount must equal the invoice amount"

vs

paid amount = invoice amount

Event Storming

To identify **Business Rules***, ask:

→ What can cause this Command to fail?

→ When do we want to prevent this from happening?

→ Can the user do this in any circumstance?

→ Are there outside regulations?

→ ...

Event Storming

A single Command can have multiple Business Rules affecting it, and can have multiple possible outcomes (Events).

Event Storming

As the model gets bigger, clusters of events naturally emerge. Often these are a first indicator for candidate **Bounded Contexts**.

Use the Event Storming map to indicate Bounded Context, and to decide which events might be sent to or received from neighbouring Bounded Contexts.

Event Storming

Typically, Big Picture Event Storming stops here. When doing Design Level Event Storming with technical people, we tend to get more precise, zoom in on specific parts of the model, and work towards a design that might end up in the code.

Event Storming Patterns

External System -> Event

Something happened elsewhere, and we are being notified, and record it in our system. There are no constraints, it is accepted as fact.

Eg **Car Has Crashed**.

Event Storming Patterns

Event -> Event

An Event that causes another Event to happen. Often this is a direct translation, for example when receiving an Event from another Bounded Context and translating to our Ubiquitous Language.

Event Storming Patterns

Command -> Event

A user or external system instructs the system to do something.

Event Storming Patterns

Command -> Business Rule -> Event|Failure

Certain preconditions are not met, so the Command is refused and the user is notified of the failure.

Event Storming Patterns

Command -> Business Rule
-> EventA|EventB|...|Failure

Various outcomes are possible depending on a complex set of conditions.

Event Storming Patterns

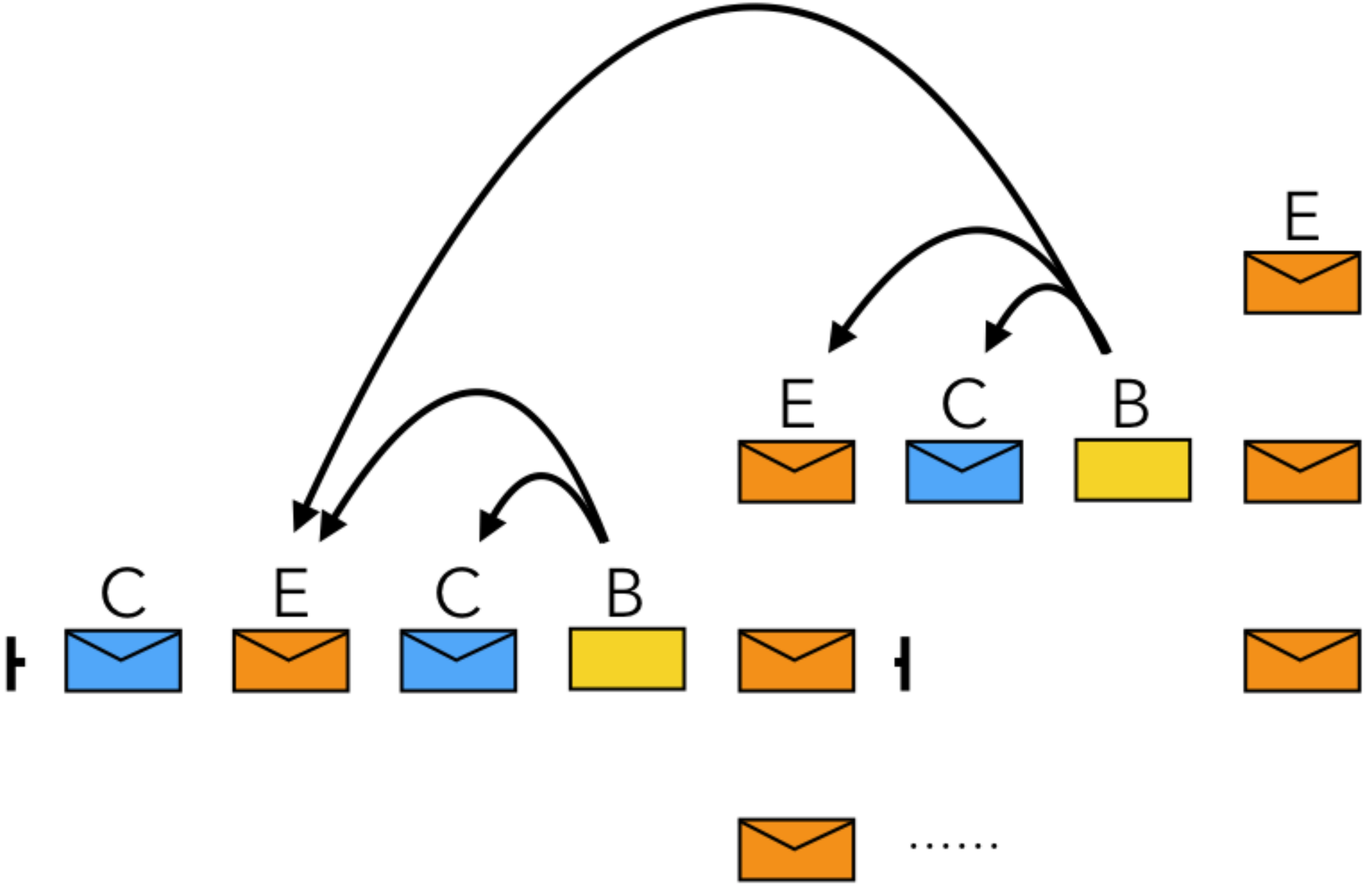
Events -> Business Rule -> EventX

A process where a set of Events over time cause something else to happen. For example, when EventX happens whenever a certain number of Events have occurred.

Event Storming

Indicate the beginning and end of business processes.

Use arrows to indicate what knowledge of past Events and Commands a Business Rule needs to make decisions.


C
E
C
B
E
C
B
E
.....

Event Storming

These clusters of connected elements can serve as good indicators for fionding Aggregates, Process Managers, Sagas, …

Learning More

Learning More

Blog:
http://verraes.net

Reading List:
http://verraes.net/2015/12/reading-list/

Domain-Driven Design by Eric Evans

Software Consultancy for Complex Environments

http://value-object.com/


DOMAIN
DRIVEN
DESIGN
EUROPE
JAN 31ST – FEB 3RD 2017
AMSTERDAM