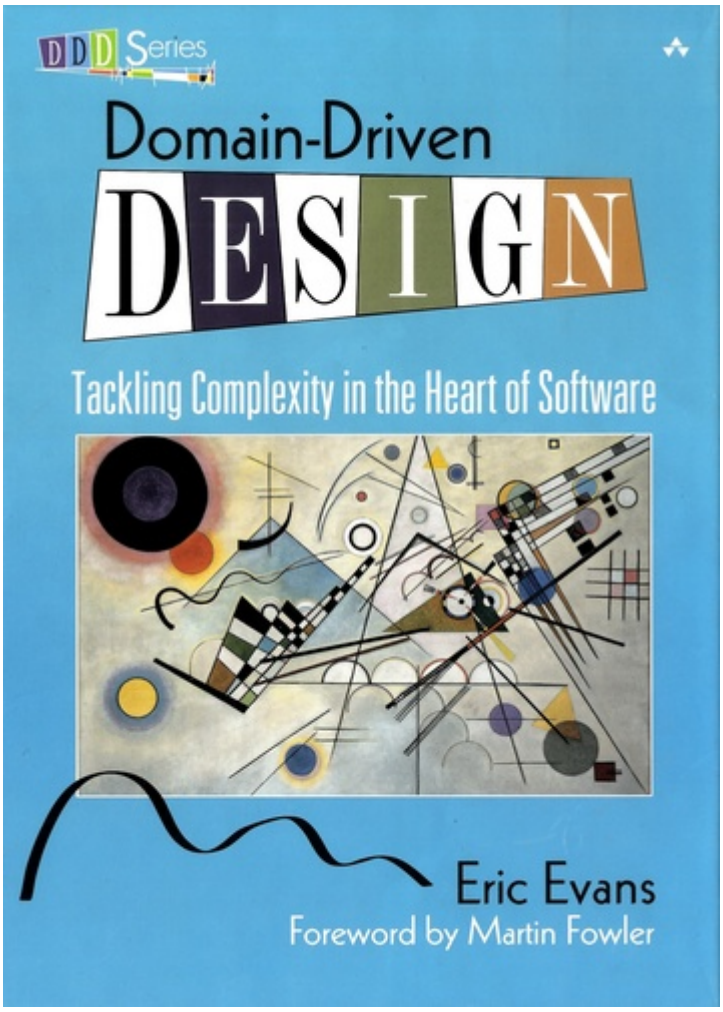
Back to the future: How a 2004 book helps us design cloud native software

Table of Contents

Domain Driven Design



Introduction

- Speaker of the day: **Konrad Renner**
- Profession: **Software Architecture**
- Some personal things: **Linux** / **Java** / **DDD** / **Open Source** / **OpenHab** / **Star Wars** / **BBQ** enthusiast
- Direct link to digital life: **GitHub - konradrenner**

Agenda

- What is it about?
- How can this time travel help us?
- Are there other crazy ideas Doc?
- Sounds pretty heavy. How does this all come together?
- Let me know what you think about all this

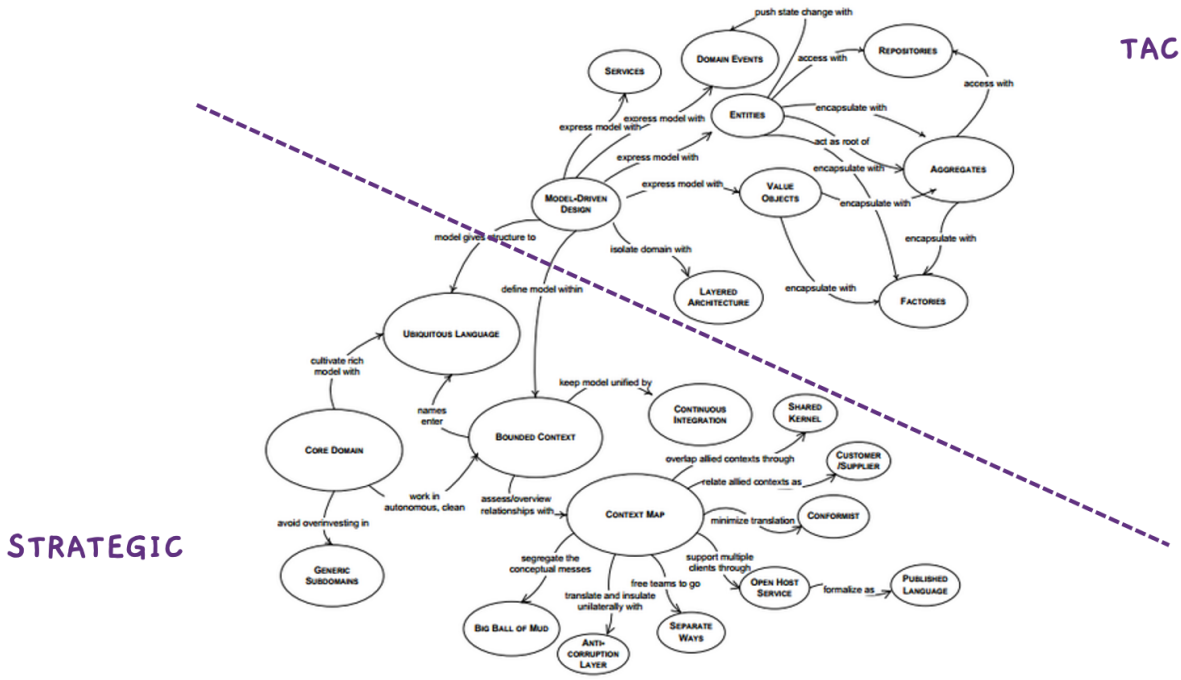
What is it about?

- Ubiquitous Language
- Distillation and Context
- Refactoring toward deeper insight

NOTE

- **Ubiquitous Language most important part**
  - Ubiquitous: appearing everywhere ⇒ users, architects, product owner, developer and of course in code too
  - The meanings of words are "context sensitive"
- **Distillation and Context**
  - You can think about Problem space and Solution Space
  - Example: Problem Space - How to build a time machine; Solution Space - How a time machine is actually built
  - *Distillation*: Distill the core domain out of your business domain
  - Put most of your effort in your core domain
  - Example Distillation: Doc Browns DeLorean DMC-12
    - Core Domain is the timetravel functionality
    - Subdomain is, that the DeLorean is possible to drive
  - *Context*: The area in which a word or some kind of "structure" has the same meaning everywhere
  - Example Context: Doc Browns DeLorean DMC-12
    - When Doc Brown talks about a timemachine, he means his DMC-12
    - When a mechanic gets his fingers on the DMC-12, he is repairing a car
- **Refactoring toward deeper insight**
  - Design and implementation is an ongoing process
  - Agile and DDD are a perfect match
  - Think of products, not projects

Strategic and Tactical Design



NOTE

- **Strategic Design**
  - "Big Picture"
  - Communication paths betweens contexts
- **Tactical Design**
  - Model within a Bounded Context
  - Aggregates not just encapsulate, they are also important for consistency
    - Aggregates map nicely to the concept of entities as described in the position paper Life beyond distributed transactions
- *Examples for technical communication possibilities will come in the next slides!*

How can this time travel help us?

NOTE

- There is a tragedy that not only concerns Marty McFly and Doc Brown, but also a galaxy far, far away
- First part of the tragedy could be a misunderstanding of the domain because: ***"It's developer's understanding, not expert knowledge that gets released into production"*** - Alberto Brandolini
- The second part of the tragedy could be, that the cut of Microservices was based on an inappropriate approach
- Inappropriate approaches would be:
  - Pure technical
  - Based on organizational circumstances

What would it look like?

NOTE

- Because **inappropriate cut Microservices can lead to unnecessary or even dangerous remote communication**
- One might think that the "smaller" a microservice is, the less complex it is
  - This is true for the local complexity of this specific microservice, but it is not true for the whole system
  - The smaller a microservice is cut, the more communication with other services is necessary and this in turn increases the complexity of the overall system
    - The much more important type of complexity is global complexity (the complexity of the whole system) because it has a much higher impact on different non-functional requirements on the whole system, than one part ot the whole system
    - It’s less about black and white thinking (monolith vs microservice) and more about creating a balance

- In the worst case you transform a "local" monolithic app (local from a transactional view), to a distributed monolithic app (distributed transactions)
  - If you are faced with the need of distributed transactions, there is already a great comparison about different distributed transaction patterns
  - As stated above: too high global complexity is worse then local complexity
  - Sooner or later this will lead to a real resilience tragedy (e.g. Deadlocks)
  - Beware: Local monoliths do not necessarily have to be bad, but distributed monoliths are problematic most of the time!
- Service Mesh Tooling (e.g. Istio, Linkerd, Consul) and similar solutions are often only symptom treatments, but do not solve the problems at the cause
  - But of course Service Mesh Tooling can solve many security problems (e.g. Zero Trust with mTLS) and resilience problems on the infrastructure layer
- So this "time travel" to the 2004 book, can help us find more effective approach
  - As the book subtitle states: Tackling complexity in the heart of software
  - In the next couple of slides I will show you some of the concepts, to minimize the propability that such tragedies will occur

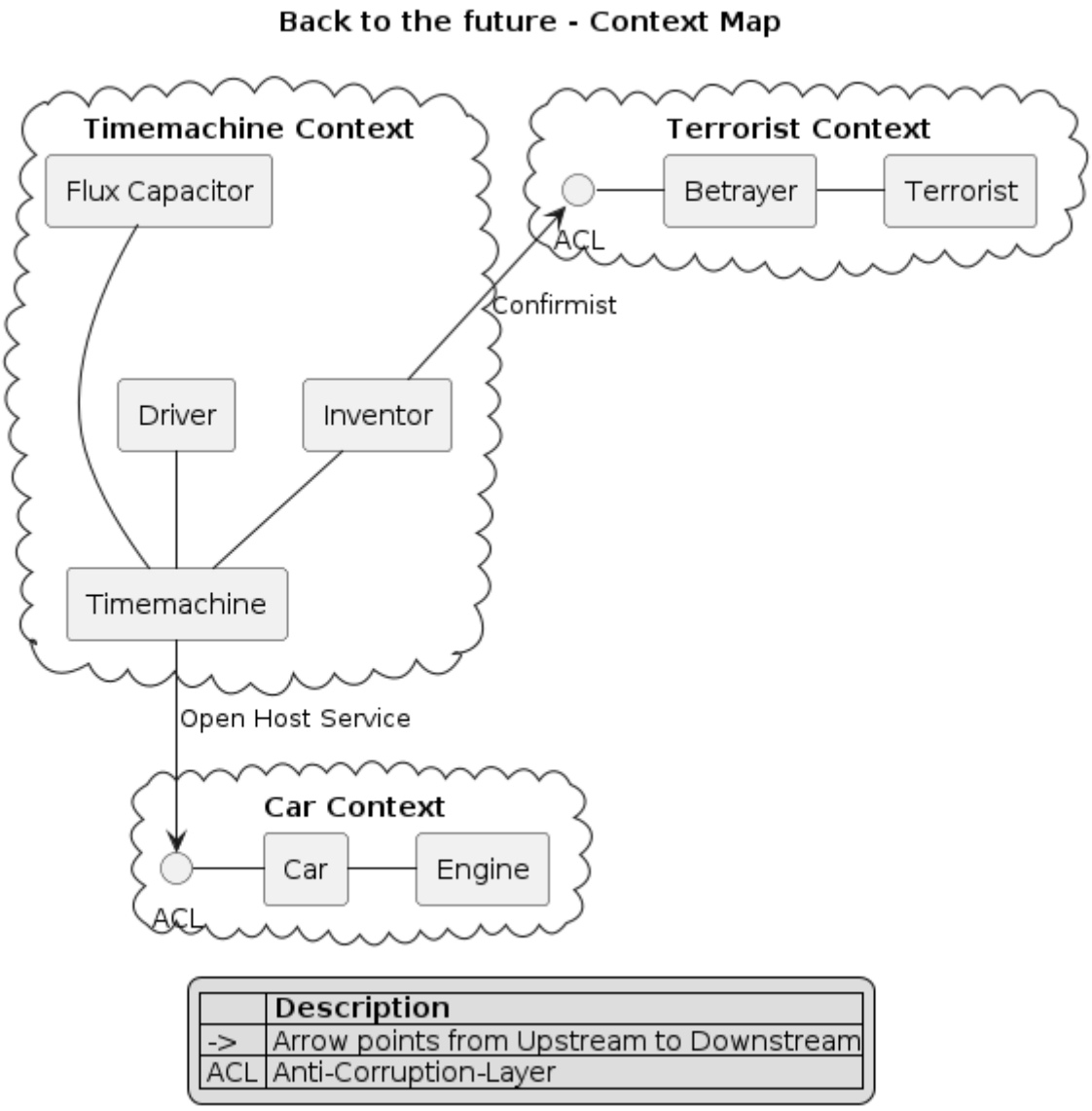
DDD for "cloud native software architecture"

- **Focus on your core domain, not technical aspects**
- Establish a common understanding of strategic AND tactical design
  - **Merge the people, split the software**
- Build Microservices or Self-Contained-Systems based on Bounded Context
  - Maybe a Bounded Context can also help by defining K8s Namespaces ;-)

NOTE

- One of the most common questions in my day to day work is, how to size Microservices or siblings (e.g. SCS)
  - Just use the Bounded Contexts

Strategic Design



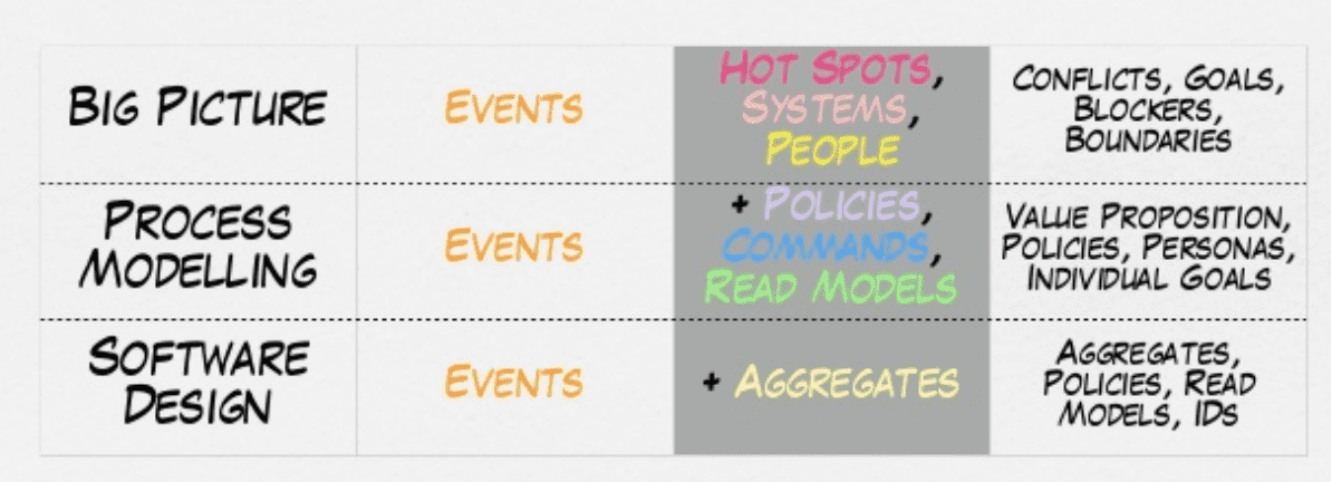
NOTE

- **The Context Map helps to understand how communication flows through the system**
  - The relationship types helps in discusions about the technical communication
    - Confirmist
      - Upstream has no motivation to provide for the downstream team's need
      - Maybe a lib, which is developed without regard to the downstream (maybe because it was create for another downstream in form of a customer-supplier relationship)
    - Open Host Service
      - Access to a system is provided by clearly defined services, using a clearly defined protocol
      - Maybe RESTful services with OpenAPI powered Published Language

Are there other crazy ideas Doc?

- *Disclaimer:* The following tooling are just my personal favorites
- Start with Event Storming
- Document architecture with arc42 template
- Take out the pain of documentation with Documentation As Code
- Structure code on basis of Clean Architecture

Event Storming



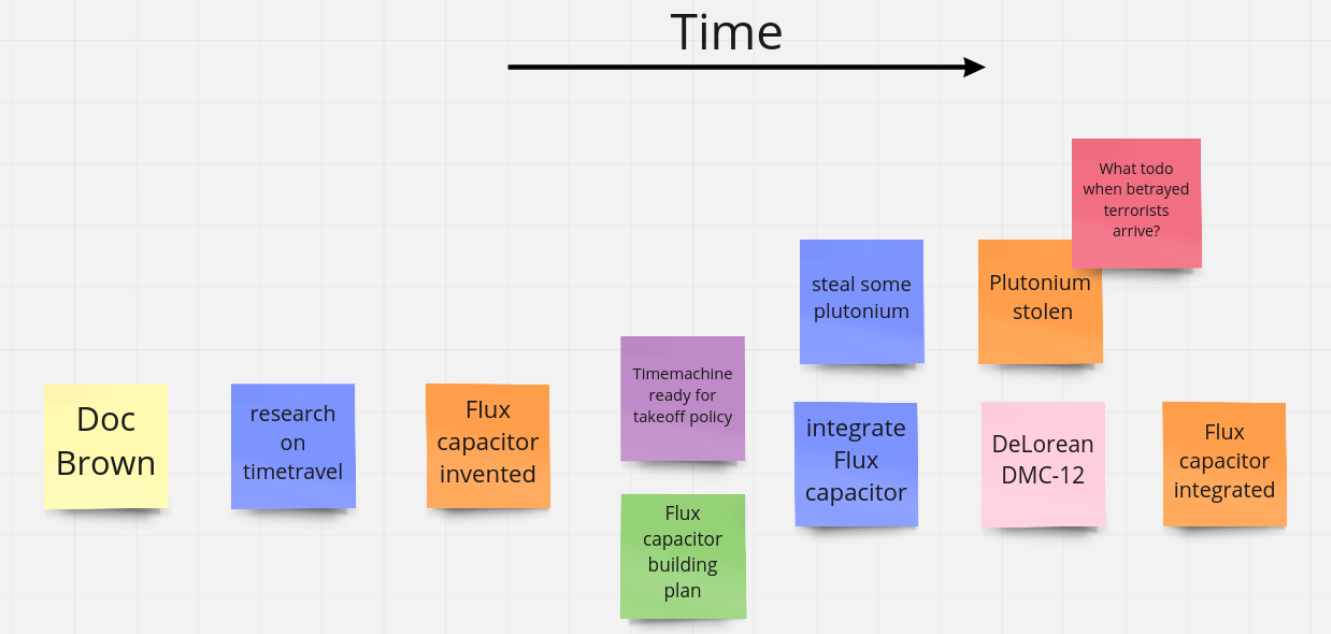
- **The key idea of EventStorming is**
  1. **See the system as a whole**
  2. Find a problem worth solving (Distillation)
  3. Gather the best immediately available information
  4. Start implementing a solution from the best possible starting point (Context)
- You just need a room with a long enough wall, many coloured stickies, something to write, the "right" people (and no table in the middle)
- Invite all relevant stakeholder in the room
  - They put their view in brain storming fashion on an "endless" wall, in form of events
  - Events are always past tense
  - They discuss the outcomes

NOTE

    - Consensus is not required, it could be a signal for different meanings of an event; mark heavy discussion with a hotspot sticky
- **Start with a Big Picture workshop**
  - Helps crossing knowledge silo boundaries
  - You get many hints about possible Bounded Contexts
- **Then you can start modelling your processes** in the contexts with the integration of commands, policies and read models
  - Picture that explain (nearly) everthing (see picture in next slide)
- **And then you could dive even deeper into Software Design** (for discovering/designing Aggregates)
  - Aggregates are the "state machines" between commands and events
  - It is not just Process Modelling with Aggregates because many processes can be connected with an Aggregate (think of a combination of processes with focus on Aggregates)

    - Think of behavior, not data!
- But be aware, that every time you dive deeper, the required person’s will change. And maybe you have to step back at some point of time
    - Have a look at chapters "system scope and context" and "building block views" and "runtime views" of arc42, if you are interested in how to document outcomes

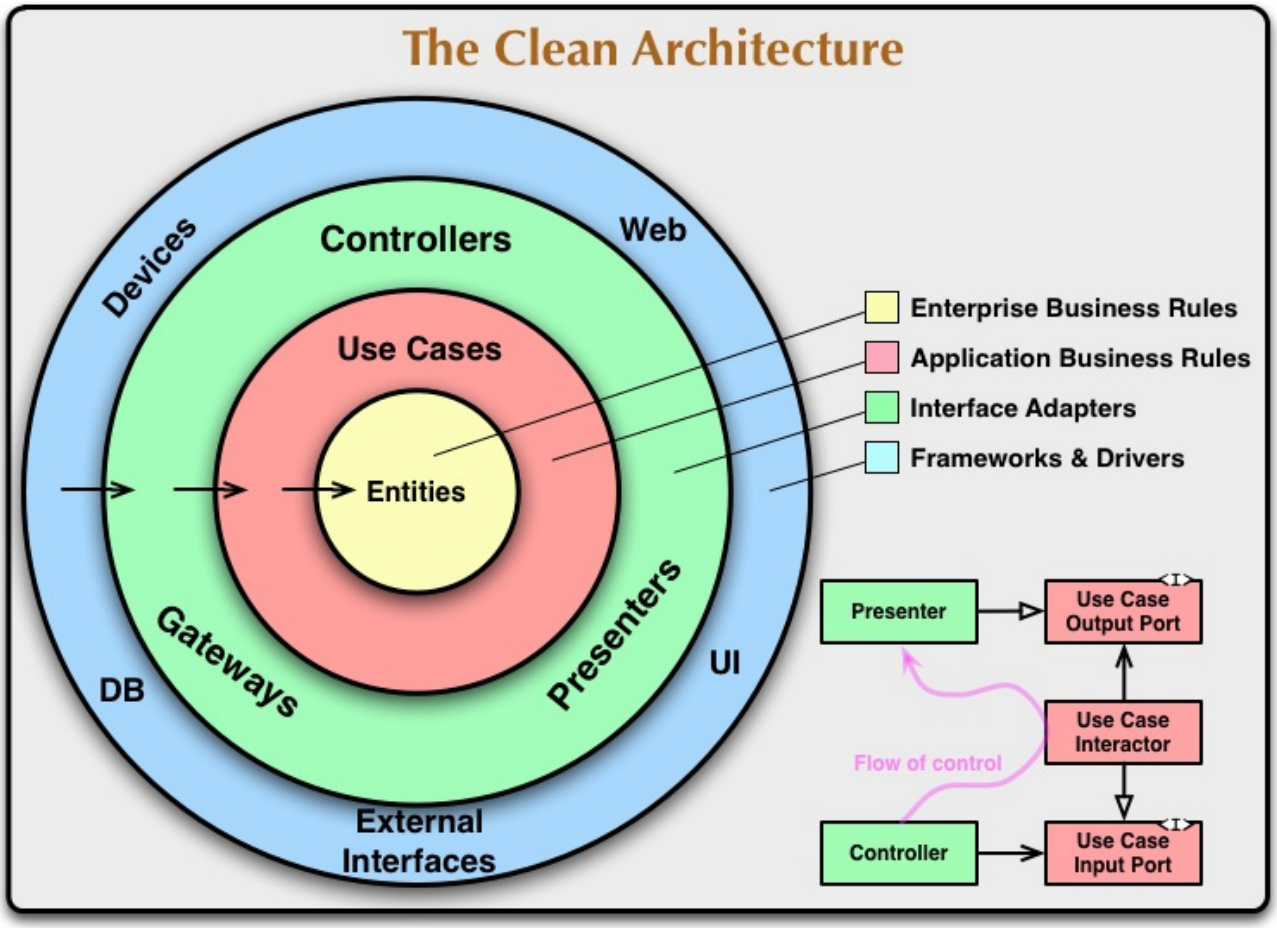
Event Storming



NOTE

- *Yellow*: People, Actor or Persona
- *Blue*: Command or Action (triggered from people, system or time based event)
- *Orange*: Event (consists at least of a noun and past tense verb)
- *Purple*: Policy or Business Rule, glue between event and thereafter command(Whenever [event(s)] the [command(s])
- *Green*: Read Model (information/data that needs to be available to take a given decision)
- *Pink*: (External) System or part of a system
- *Red*: HotSpot (open question, noticed for later discussion)
- Precise Notation or explorations are not required and could harm creativity (e.g. it is not important if the yellow means people or Persona)

Clean Architecture



NOTE

- The most important part is flow of control
  - **Never ever make inner circles depend on outer!**
  - Technical aspects must never enter the domain logic
    - If so: your code will e.g. not be unit testable (you cannot mock away technical aspects sufficient)
- This architecture perfectly fits with the "Layered Architecture" and Tactical design as described in the DDD book
  - **Enterprise Business Rules**: *Entities and Aggregates*
  - **Application Business Rules**: *Domain Services, Repository contracts* (e.g. Java Interface)
  - **Interface Adapters**: *Repository implementations*
- An example is just 2 slides away

Sounds pretty heavy. How does this all come together?

- publishing-company example
- Uses Quarkus as *"Kubernetes native Java stack"*
- Boundary-Control-Entity pattern for implementing "lightweight" Clean Architecture on top of DDD
- Architecture automatically checked with ArchUnit

NOTE

- Some think, Java is not the cool or hip enough nowadays
  - They did not try Quarkus yet
  - rock solid tooling, massive community, native performance and state of the art dev experience
- DDD and Clean Architecture are a perfect match
  - Use BCE and you also get a standardized und clear structuring of your projects
    - **Boundary**: *Interface Adapters*
    - **Control**: *Application Business Rules*
    - **Entity**: *Enterprise Business Rules*
- Let the tooling do the "boring" work for you
  - Automatic versioning and releasing
  - Automatic publishing
  - Automatic testing
- The (Git) Repo is the single source of truth for all aspects
  - Architecture, Security, Code, Config
  - Every change is tracked in your favorite VCS and absolutly traceable
  - Maybe you **use GitOps to further improve automation**

Time for an example

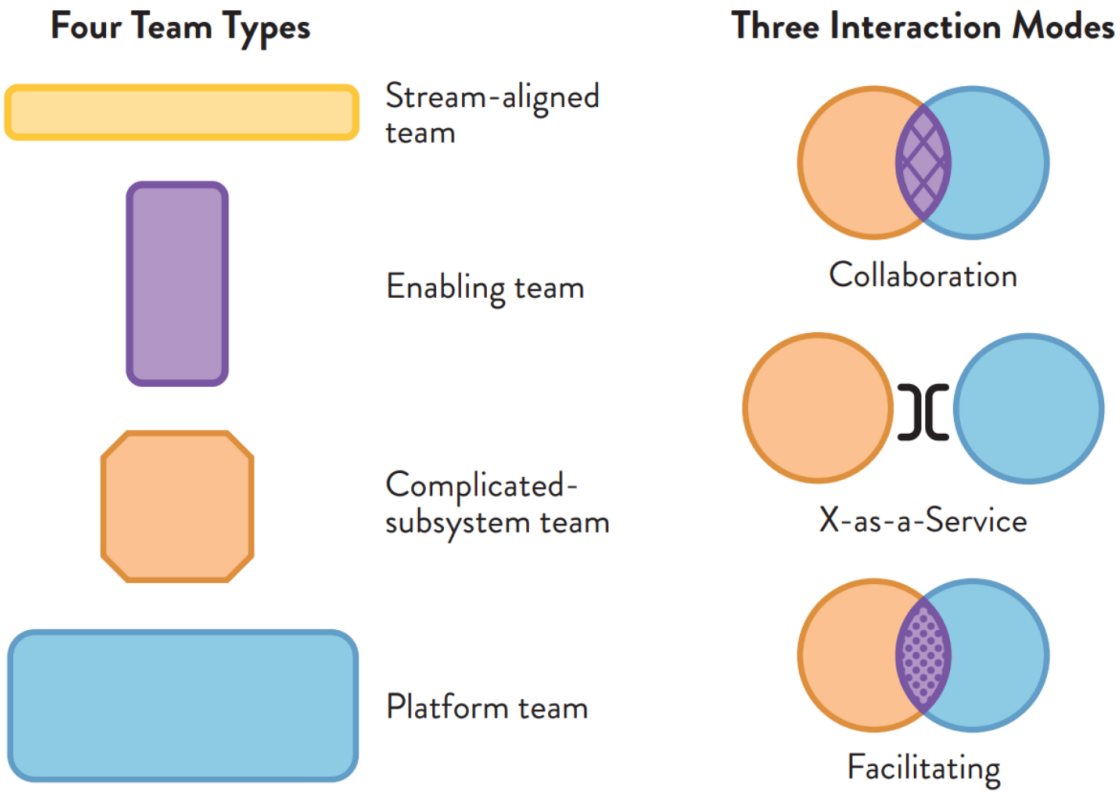
NOTE

- Disclaimer: The publishing-company example has just little todo with back to the future :-) (one book entry)
- This example "lives", so it is in parts unfinished an will change from time to time
- It demonstrates all of the tools discussed, except context mapping
  - It just contains the "Author Aggregate" from the "Author Context" (1:1 mapping)
- It consists of an Web UI (JSF), REST API and an Cross Compiled Mobile/Desktop Companion App

But…

- *Organizations which design systems […] are constrained to produce designs which are copies of the communication structures of these organizations.* - Melvin E. Conway
- Have a look at Team Topologies
  - Approach to modern software delivery with awareness of
    - Conway's Law, team cognitive load and responsive organization evolution

Team Topologies



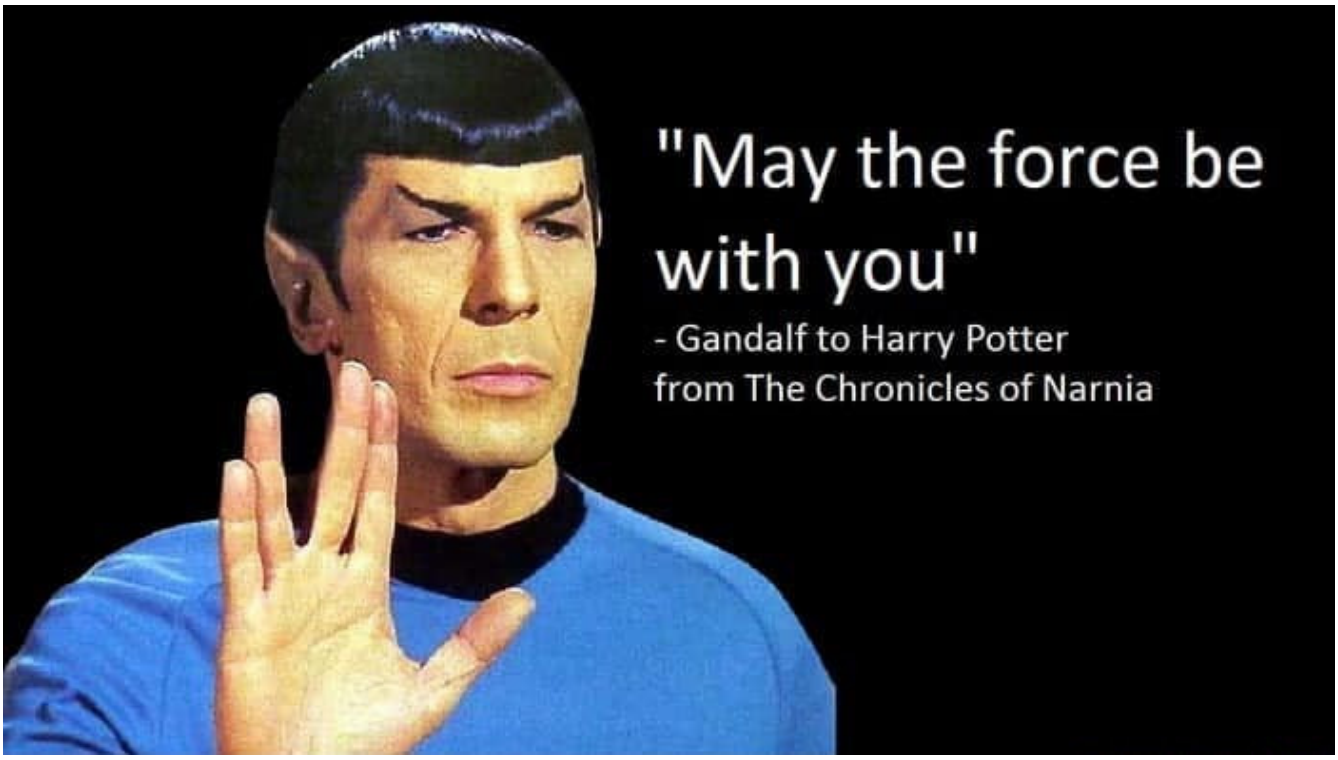
© Matthew Skelton and Manuel Pais from *Team Topologies*

NOTE

- **Like DDD it "just" formalizes some good practices and ideas**
- **Stream aligned teams** are the "heart" because the are aligned on value streams
  - These are based on top of the DevOps ideas
  - The other teams are "just" supporting them in which they take away cognitive load
  - So the other team types are just required, if the cognitive load will get to high for a stream aligned teams
  - The other teams may consists "internally" also of stream aligned teams
- **Complicated subsystem team**:
  - Parts of the system which not directly mapped to the value stream, but are a requirement "to function"
  - Think on the flux capacitor: one team just focuses on this complicated part, whereas the stream aligned teams will do improvements on the integration with the Delorean
- **Enabling team**:
  - Disclaimer: This is not Architecture Department, but a team of specialists
  - They help to spread knowledge about new things in the organization and tech world

    - They also evaluate if "trends" are applyable and how
- **Platform team**:
    - They are building and maintain e.g. the tools which are required, so that stream aligned teams can work effective AND efficient
    - Think on the Delorean: A Platform team would have built it and will repair things, whereas the stream aligned teams will focus on the time travel functionalities
- The interaction modes helps visualising and so understanding the dependencies between teams
    - **Collaboration**: strong delivery dependencies (e.g. stream aligned and complicated subsystem team)
    - **X as a Service**: Decoupling and standardization (mostly used when interaction with a platform team is needed)
    - **Facilitating**: helping or being helped by another team (mostly the case when a stream aligned teams "gets knowledge" from an enabling team)

Let me know what you think about all this



NOTE

- Thank you for the possibilty to share my thoughts on this topic
- In closing, I have only two things to say
  - Never stop refactoring, there is no "perfect" or "everlasting" solution
    - *Software development is a learning process, working code is a nice side effect*
  - And: **may the force be with you**