Contact us:
https://www.linkedin.com/in/alina-liburkina/
https://www.linkedin.com/in/oliver-zihler/

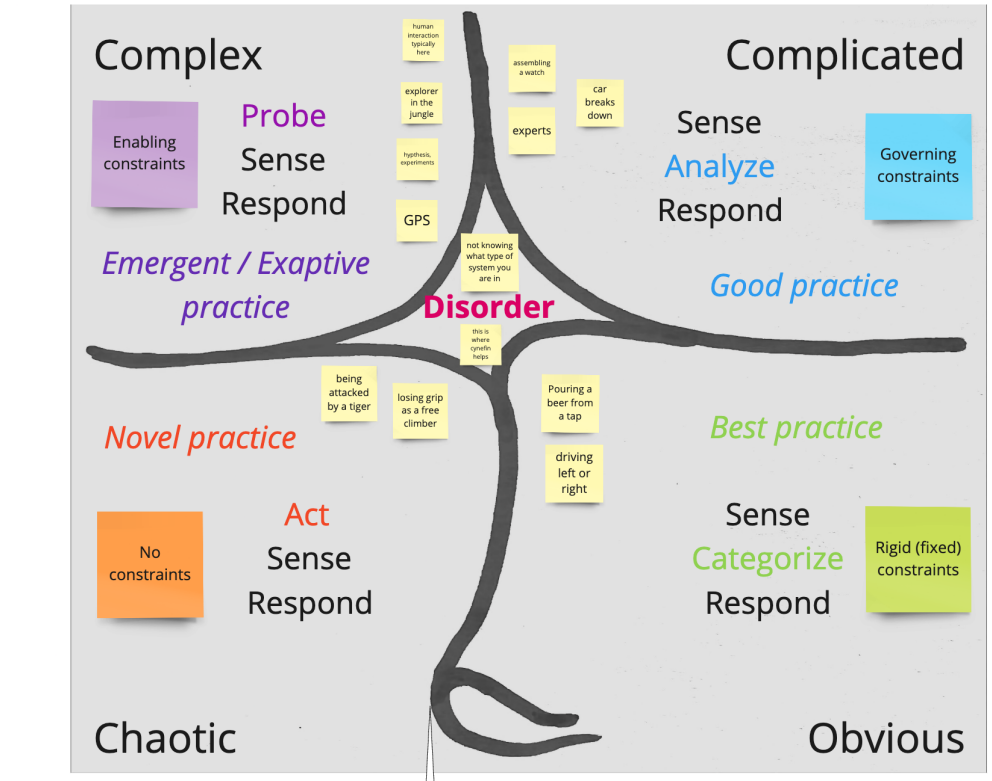
Our International Guest: Vlad Khononov, **Possible only this yea**r
DDD with O'Reilly author of "Learning Domain-Driven Design"
https://www.letsboot.ch/en-gb/course-date/vlad-ddd-2023-12-07

Our In-House Courses:
https://www.letsboot.ch/en-gb/course/event-storming-ddd-architecture (extended version of the current workshop)
https://www.letsboot.ch/en-gb/course/clean-code
https://www.letsboot.ch/en-gb/course/clean-architecture-spring

Join Our Meetup:
https://meetu.ps/c/4Xxs0/NL64k/a
https://www.techexcellence.io

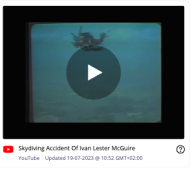
Cynefin

Cynefin Framework
A sense making framework - a way to look at reality
5 domains / states a system can be in



Ordered Systems

Boundary between Obvious and Chaos is a "cliff": Zone of complacency
quick to get into, hard (if at all) to get out again



Waterfall

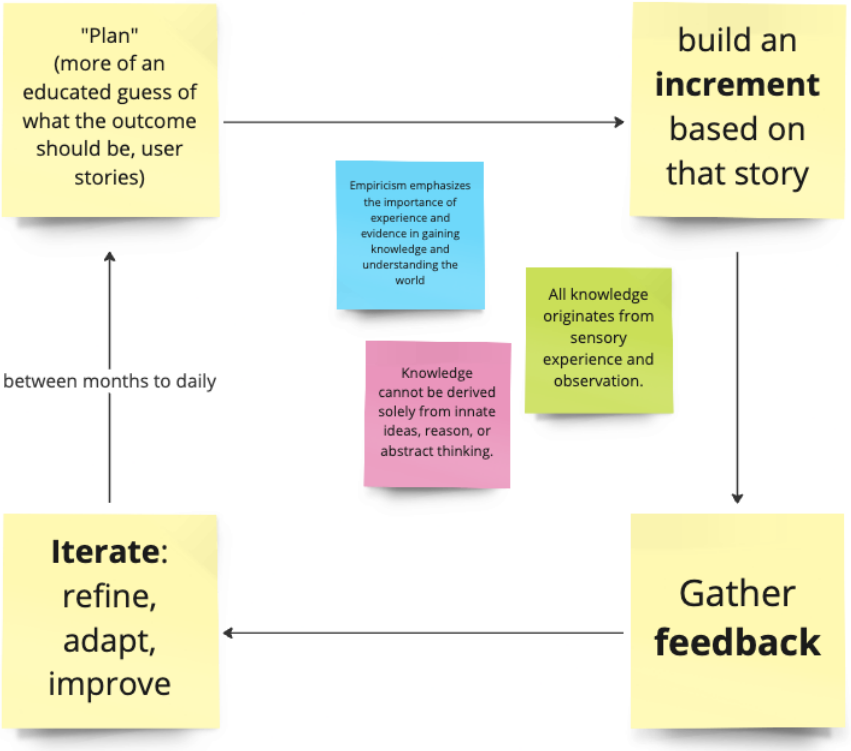
Agile

Resources:
https://thecynefin.co/about-us/about-cynefin-framework/
https://youtu.be/ogtpxA6brGo
https://doras.dcu.ie/20770/1/icssp15main-mainid50-p-29621-129c376-24234-preprint.pdf

Agile, Feedback, Iterative & Incremental Development

Agile

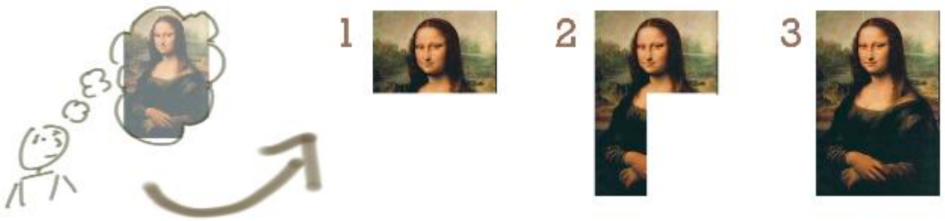
Agile is empirical: not abstract thinking, but validating in the real world



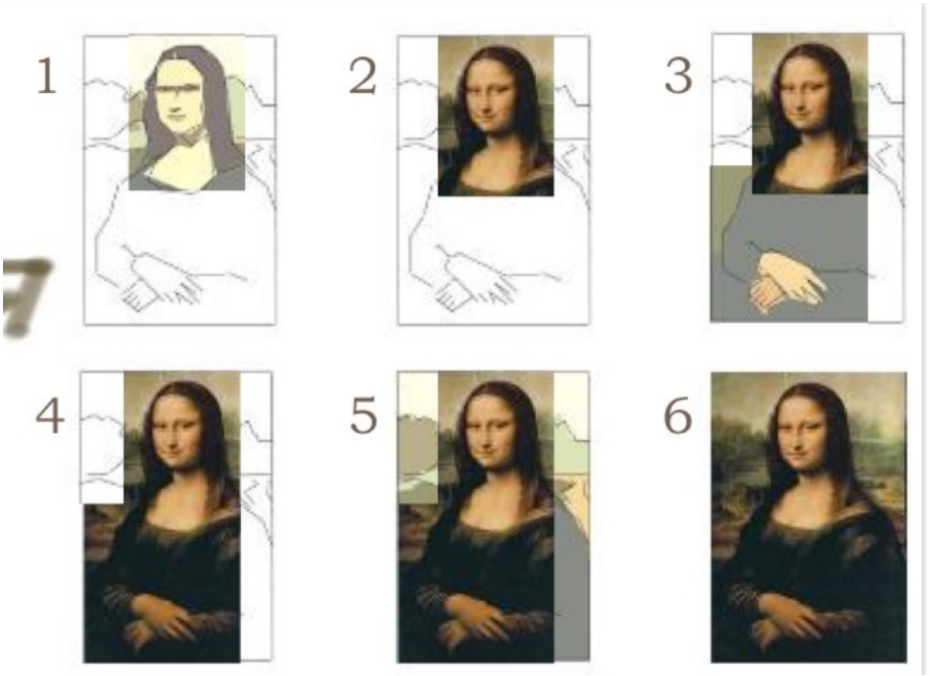
Agile *is the ability to create and respond to* ***change***

Iterative Development

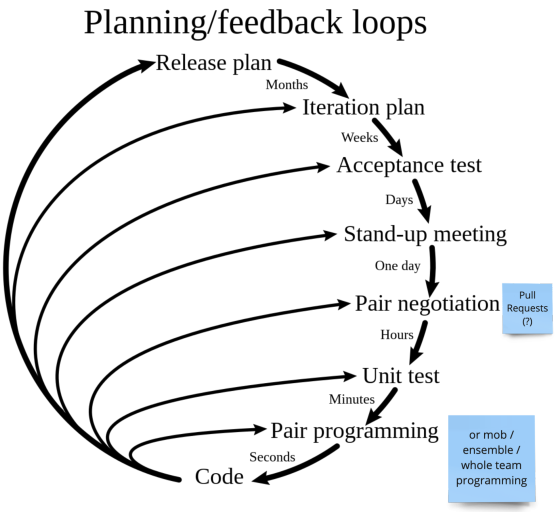
Incremental Development



Incremental & Iterative Development



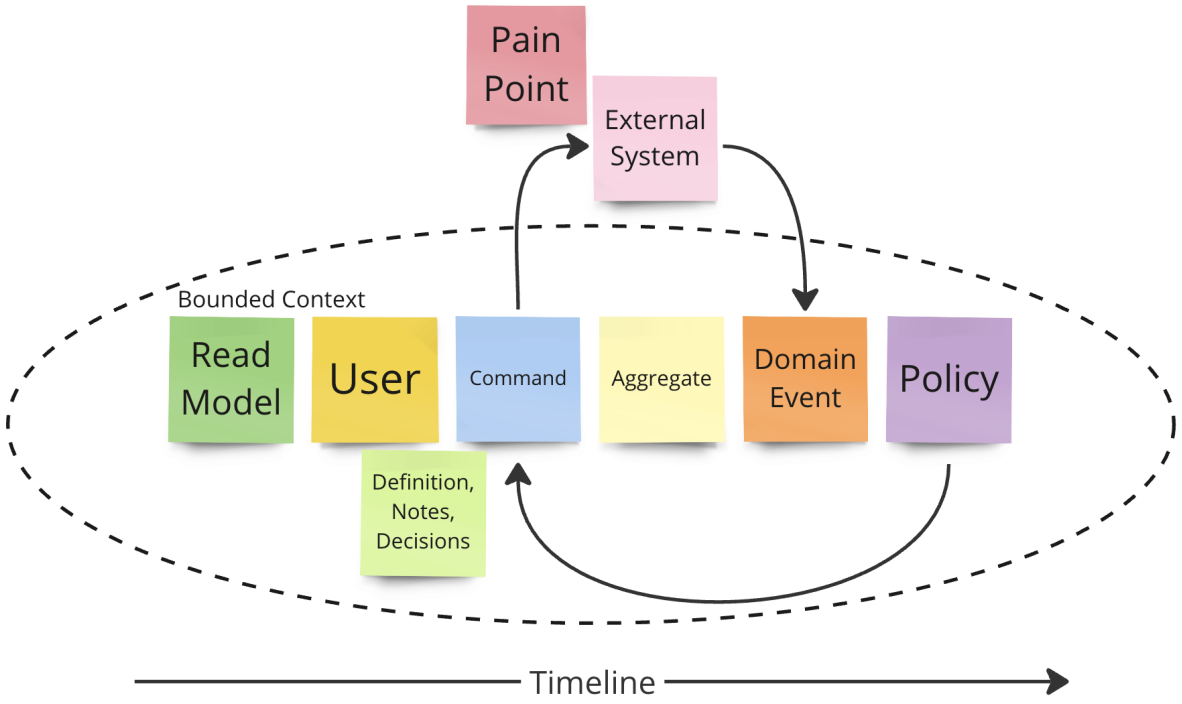
Feedback Opportunities in XP



Resources

https://www.stickyminds.com/article/neglected-practice-iteration
https://de.wikipedia.org/wiki/Extreme_Programming

Event Storming



Application

Note: ES is LIGHTWEIGHT - you can't do it wrongly!

As long as everybody involved understands the system better than before, the goal of event storming is achieved.

1. Start with gathering Domain Events (orange --> Event Storming)
2. If there are no obvious Domain Events anymore: start to add commands iteratively
3. For each command, decide
    1. if a user can carry it out (and what data (read model) they need to carry it out)
    2. if an automatic policy / rule (when - then) executes it (and maybe also what data (read model) is required
4. Commands typically invoke external systems. They can be introduced rather early
5. Pain Points can be added to be discussed also later, but should be resolved or mitigated at some point
6. User light green stickies to note down any definition, decision, or other notes that are not risks or pain points

The next part typically is more for devs and technical people

1. Once you think you have your algorithms in place, you can start to think about what data/state they require. Write them down as aggregates on pale yellow sticky notes and attach commands on the left and produced domain events on the right (these are DDD aggregates)
    1. Aggregates are transaction boundaries. They enforce consistency rules.

2. Once you are done with that, you can start to group those aggregate combinations into so called bounded context (again, a DDD concept). The typical criterion to decide what belongs to the same BC is to think from a domain language perspective. If an aggregate name denotes multiple different concepts, it belongs into two different bounded contexts.

At any point, new domain events may emerge or people may go back to refine certain parts once new information is revealed

Domain Event

Events within the domain
Happen after one another
Past tense
In domain expert's language

Command

Lead to 1 or more events
Carried out by

- a user (yellow) using a read model (green)
- a software (policy, purple)

User / Actor

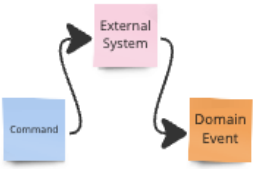
A human user interacting with the system
Executes commands
Requires read models (GUI, descriptions, PDFs, other data) to know how to carry out a command

Policy

Reactive logic to a domain event
Automatic policies: whenever (if) [domain event], then do [command]
When neither a user nor a system triggers a command, it's a policy
"Business" and other rules in the system
Use 2 policies for if - else
Read Models can be added if data is needed for the decision

External System

Separate, 3rd party or internal system
Receives commands (blue)
Triggers commands or produces domain events (orange) in the system



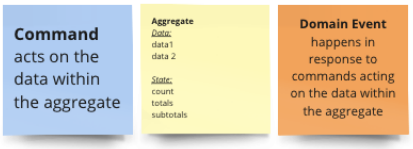
Read Model

Data to help a user invoke a command (can even add a wire frame or anything that helps)
Highlight *specific* required data
Required data can be anything, e.g.:

- data on screen
- data from a SQL query
- data from any data source
- specific part of the UI

Aggregate

Data and state
Commands act on the aggregate's data
Aggregates may or may not emit domain events in response
Noun for a name
Grouping commands and events



Bounded Context

Group related aggregates
- related by concepts
- related by (domain) language

Use as (preliminary) boundary for microservice scope
- consolidate for transaction boundaries
- decompose for performance

Can also simply be used as package names in a monolithic service before the domain is well understood.

Pain Point

Concerns, risks, issues
Need to be resolved!
Should be raised transparently

Definition, Notes, Decisions

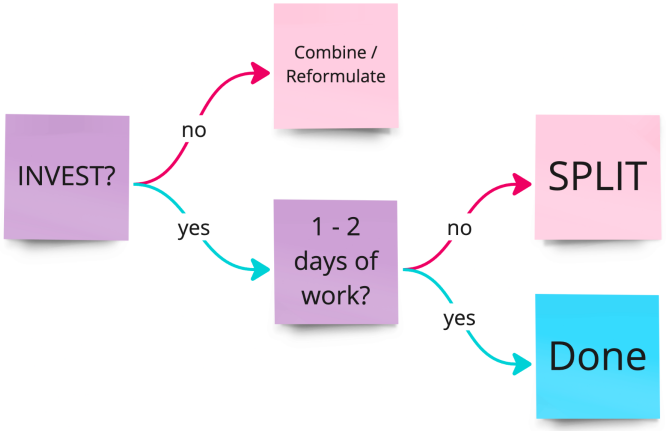
For unknown / confused terms
Aid consistent discussions

Time Triggered Domain Event → Domain Event

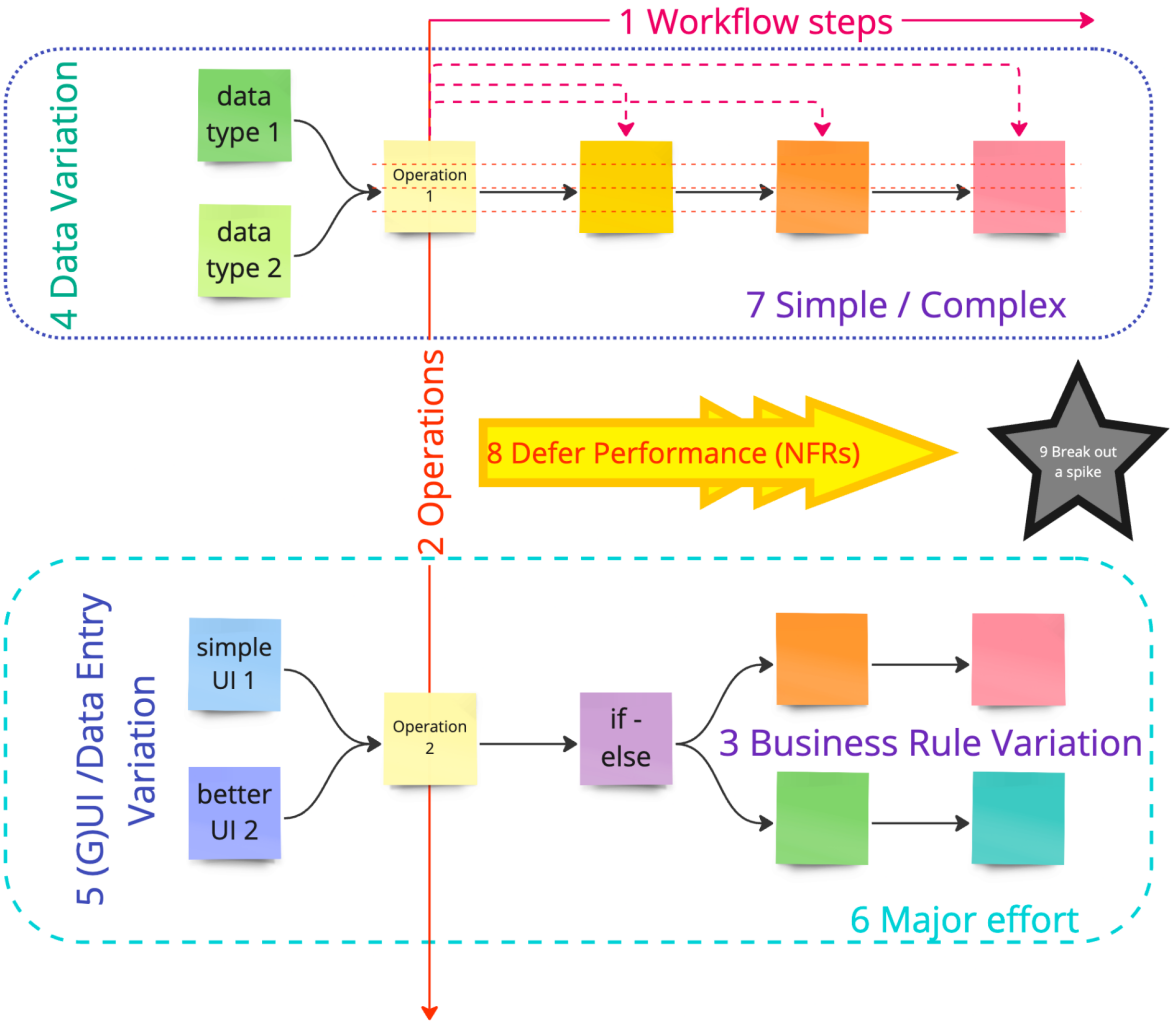
No user involved
Not a software-based decision
Just the passing of time decides this

Story Narrowing / Splitting

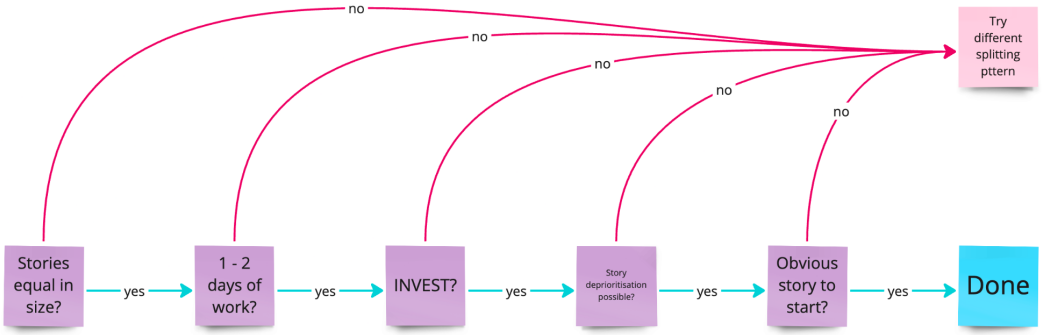
1. Prepare input story for split



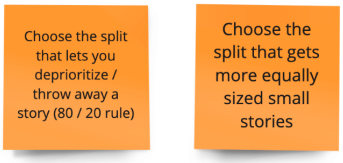
2. Apply a splitting pattern



3. Evaluate the split



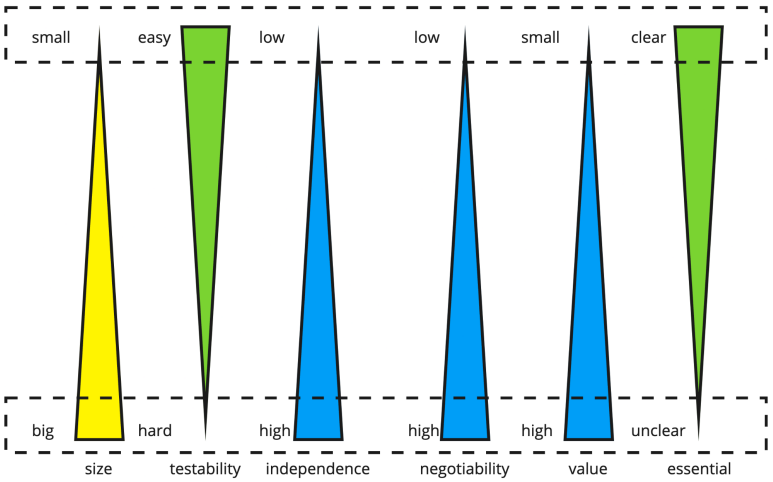
Rules of Thumb:



4. Meta Splitting Pattern

5. INVEST

Independent - **N**egotiable - **V**aluable - ~~**E**stimable~~ **E**2e / **E**ssential - **S**mall - **T**estable

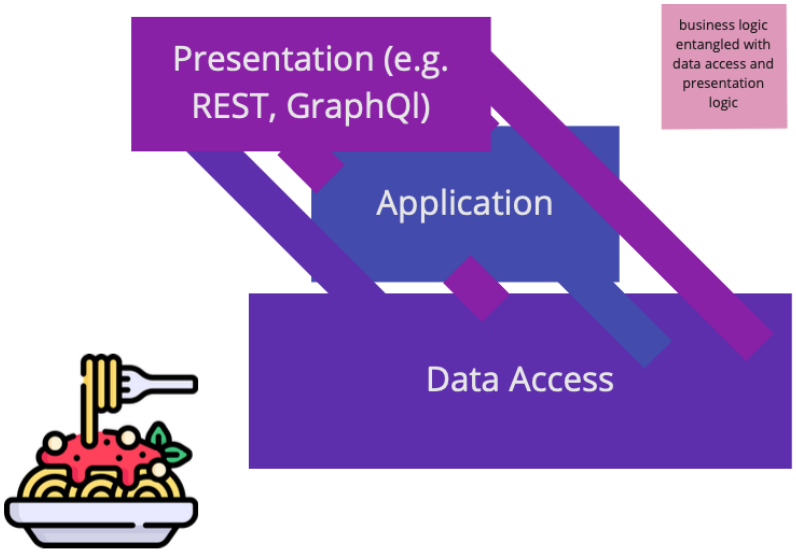


Resources

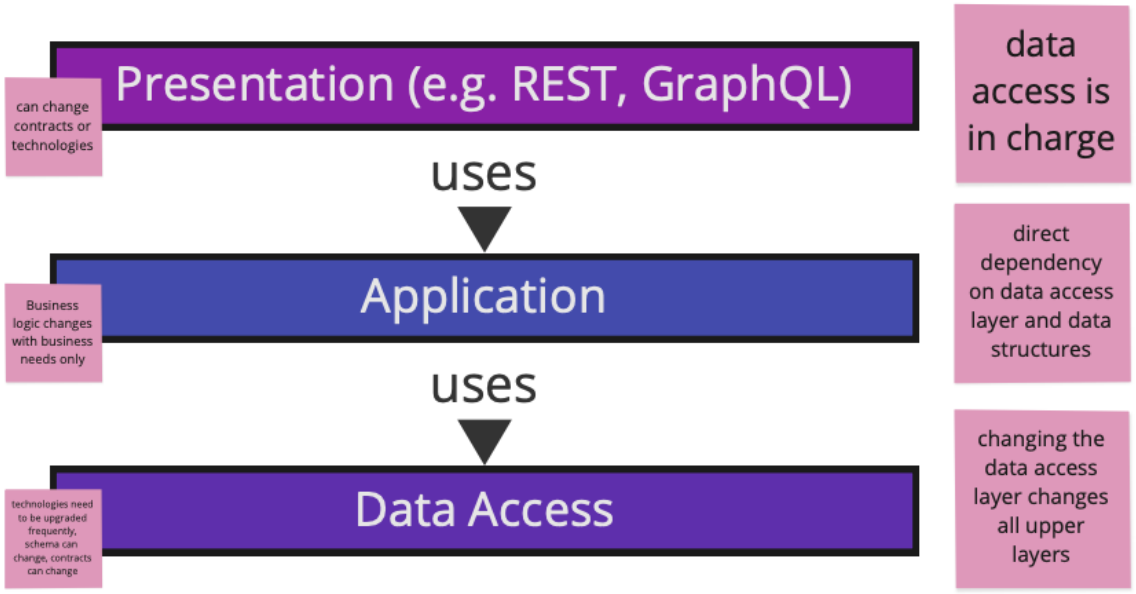
https://www.humanizingwork.com/the-humanizing-work-guide-to-splitting-user-stories/
https://learning.oreilly.com/videos/event-storming/0636920362104/

Layered-, Hexagonal- and Clean Architecture

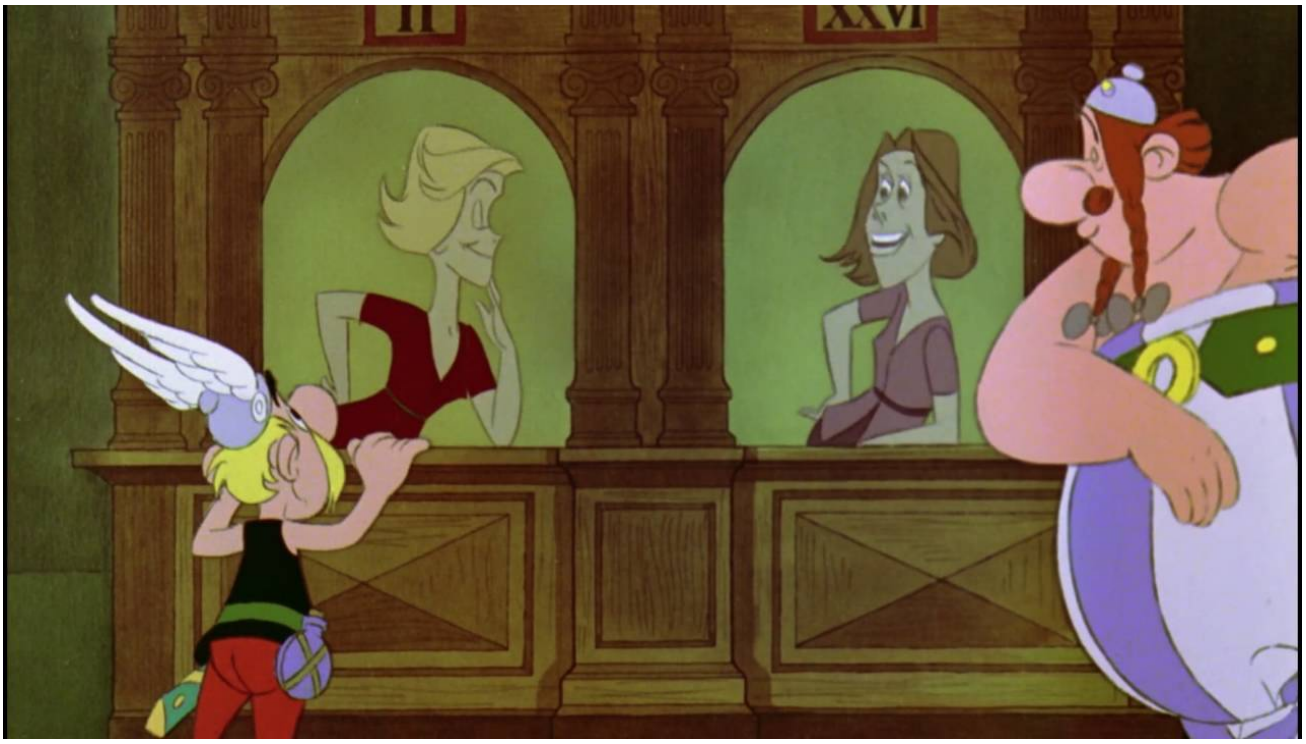
No architecture



Layered Architecture



Dependency Inversion and Interface Segregation

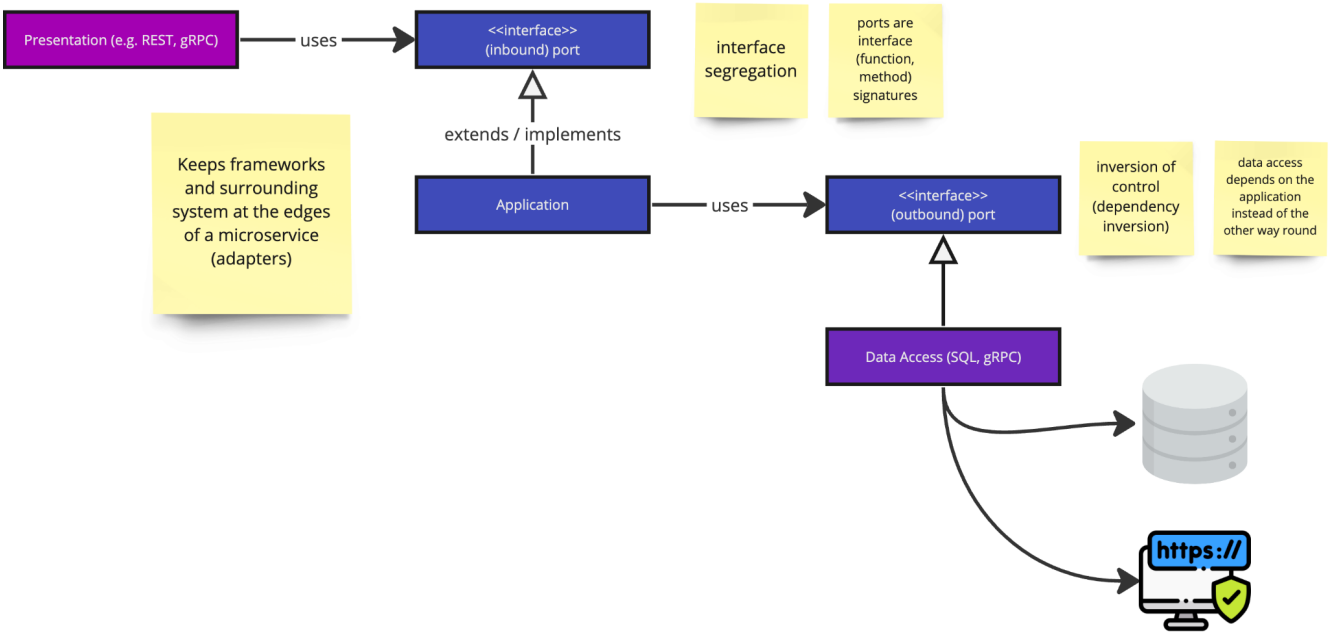


How does the input/output look like? Who decides?

Do we know *how* the output is produced? Who decides?

Do we know *how many different* actions each adapter can perform?

Who *depends* on whom?



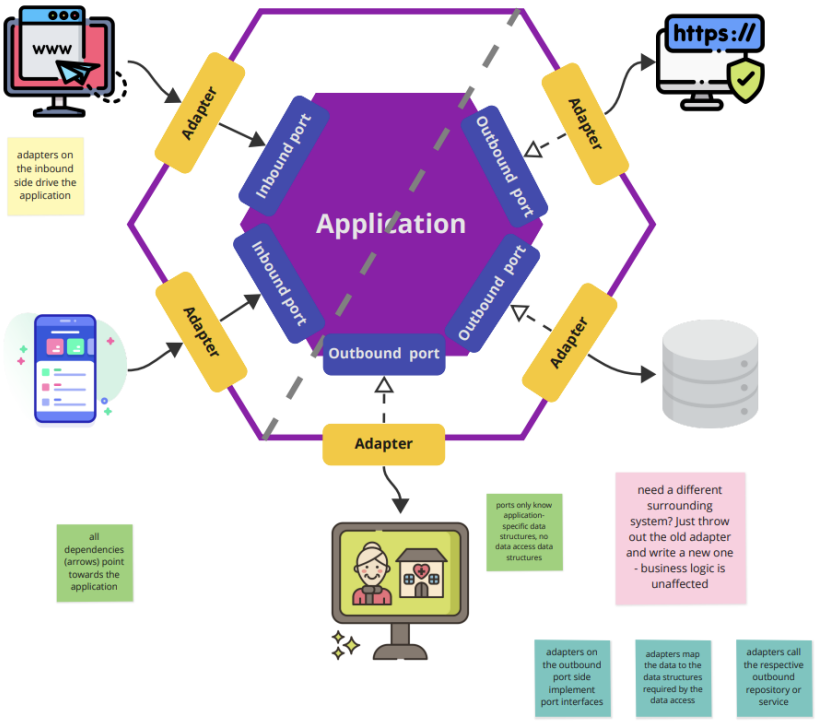
Hexagonal Architecture / Ports and Adapters

Input port

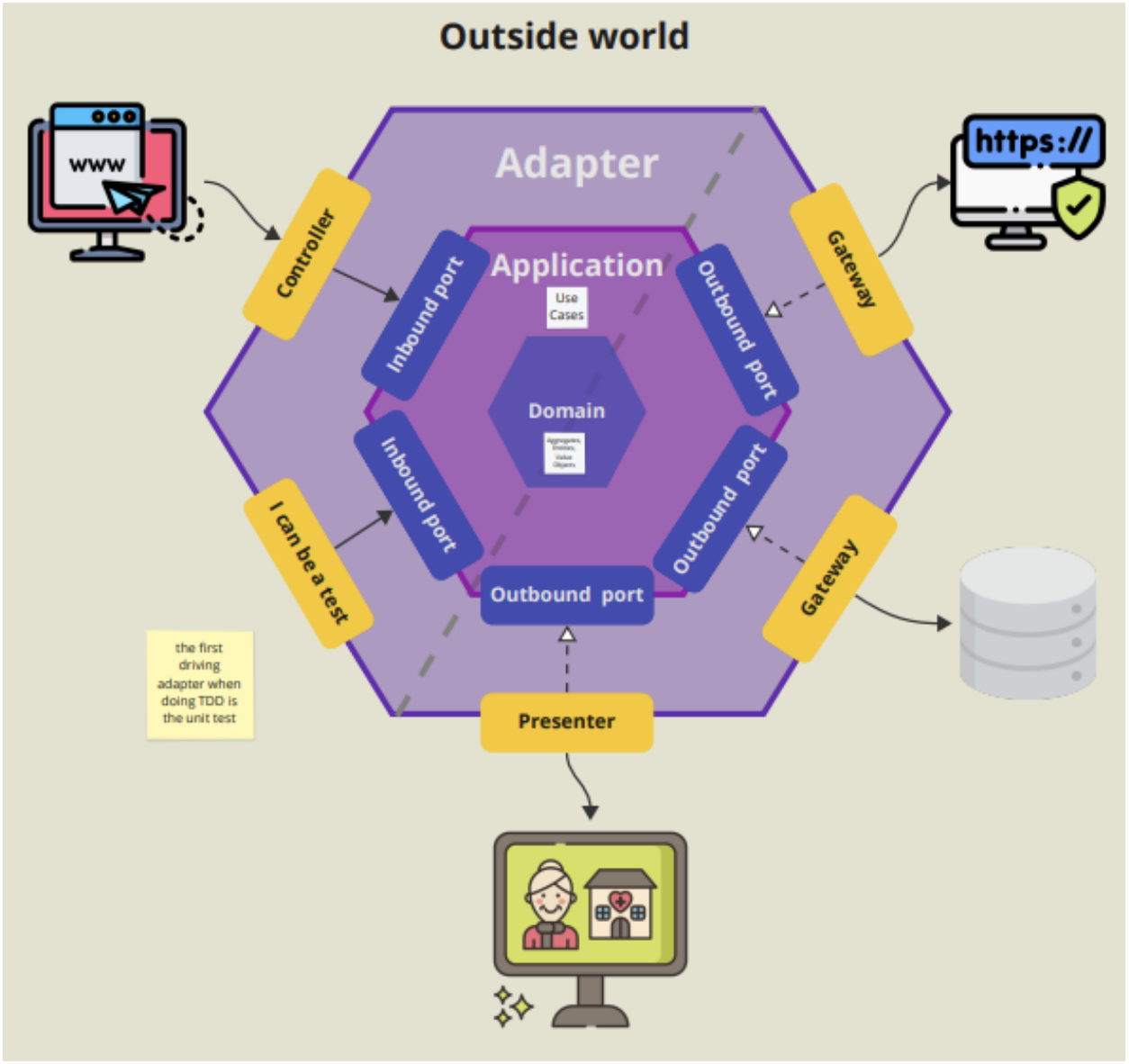
Inbound port /
Driving port /
Primary port

Output port

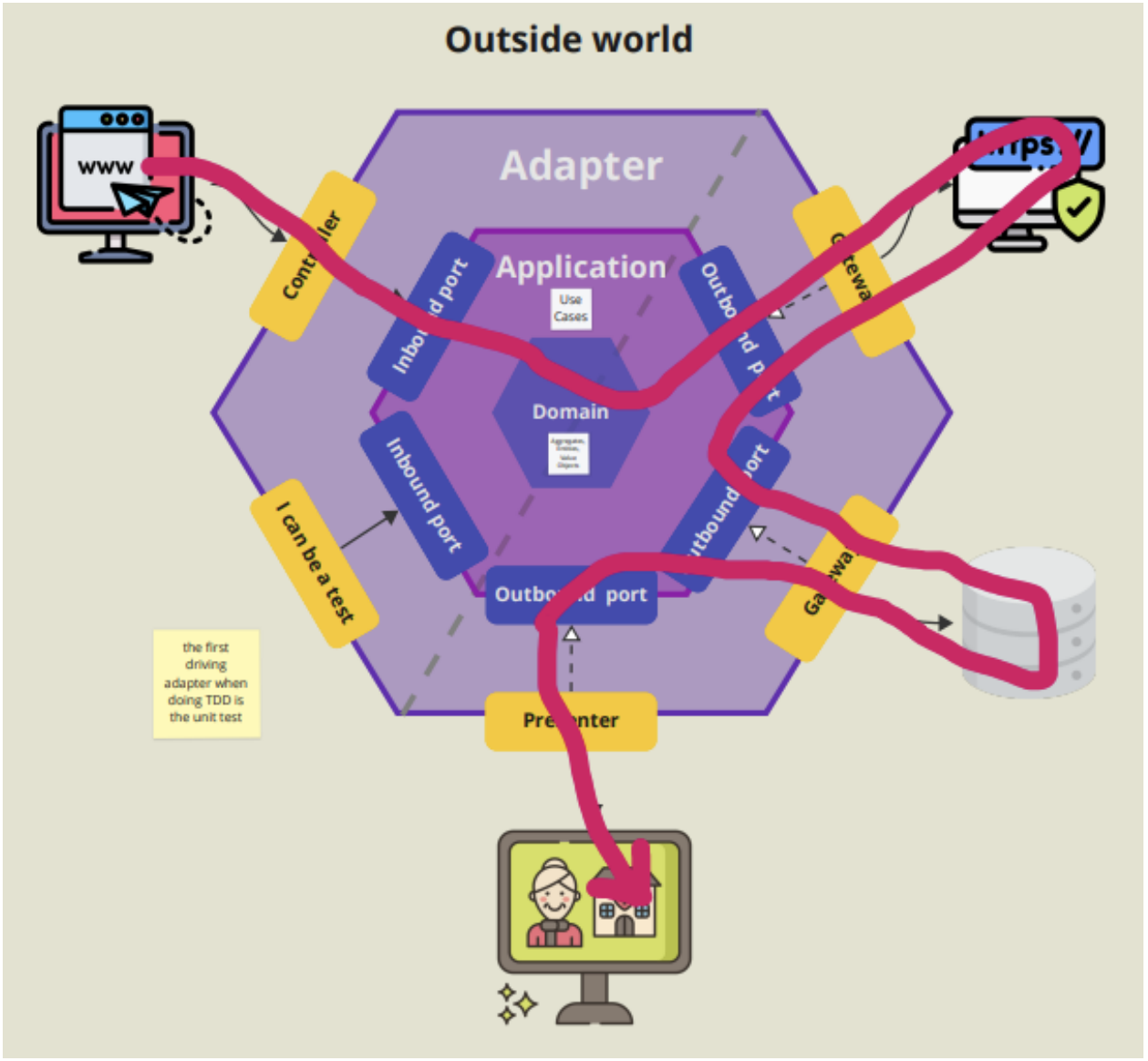
Outbound port /
Driven port /
Secondary port



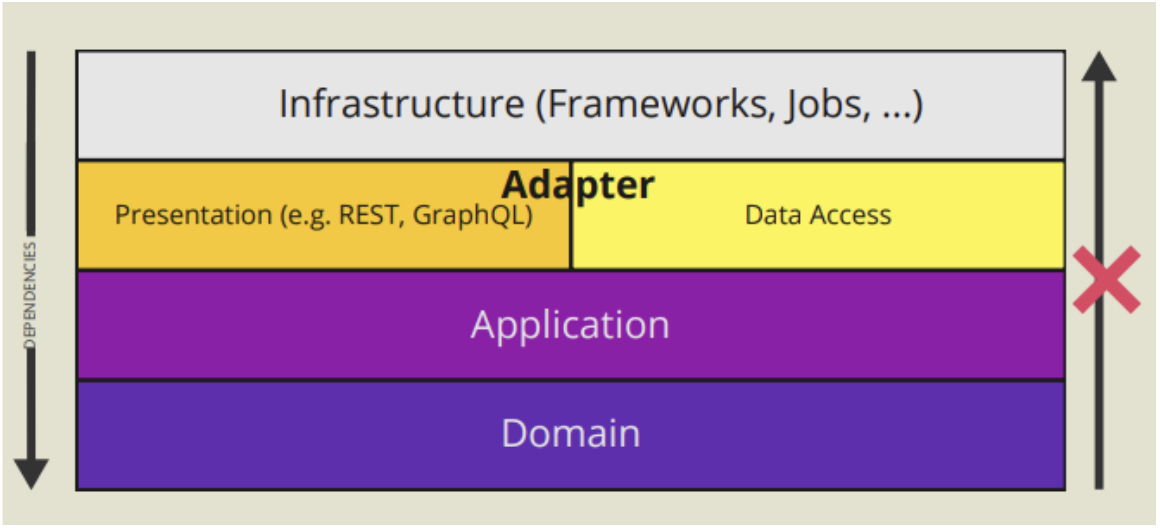
Clean Architecture (Use Case driven)



Clean Architecture Control Flow



Clean Architecture Dependencies Between Layers



Use Case Code Example

```
public class CreateDancingEventUseCase implements CreateDancingEvent {

    private final FetchUnpublishedDancingEvents fetchUnpublishedDancingEvents;
    private final StoreUnpublishedDancingEvents storeUnpublishedDancingEvents;
    private final PrevalidateUnpublishedDancingEvent prevalidateUnpublishedDancingEvent;


    public CreateDancingEventUseCase(FetchUnpublishedDancingEvents fetchUnpublishedDancingEvents, StoreUnpublishedDancingEvents storeUnpublishedDancingEvents, PrevalidateUnpublishedDancingEvent prevalidateUnpublishedDancingEvent) {
        this.fetchUnpublishedDancingEvents = fetchUnpublishedDancingEvents;
        this.storeUnpublishedDancingEvents = storeUnpublishedDancingEvents;
        this.prevalidateUnpublishedDancingEvent = prevalidateUnpublishedDancingEvent;
    }

    @Override
    public void execute(CreateDancingEventInput input, PresentDancingEventsCreation presenter) {
        try {
            EventOrganizer eventOrganizer = new EventOrganizer(input.eventOrganizerId());

            UnpublishedDancingEvents unpublishedDancingEvents = fetchUnpublishedDancingEvents.fetchAllOfEventOrganizer(eventOrganizer);

            UnpublishedDancingEvent unpublishedDancingEvent = createFrom(input);

            prevalidateUnpublishedDancingEvent.prevalidate(unpublishedDancingEvent);

            unpublishedDancingEvents.add(unpublishedDancingEvent);

            storeUnpublishedDancingEvents.store(eventOrganizer, unpublishedDancingEvents);

            presenter.presentSuccess(unpublishedDancingEvent);
        } catch (Exception e) {
            presenter.presentFailure(e);
        }
    }

    private static UnpublishedDancingEvent createFrom(CreateDancingEventInput input) {
        Description description = new Description(input.description());
        Title title = new Title(input.title());
        EventDate eventDate = new EventDate(input.eventDate());

        return new UnpublishedDancingEvent(title, description,eventDate);
    }

}
```

Resources

Repository: https://github.com/ozihler/chopen-workshop
TDD in Hexagonal & Clean Architecture: https://youtu.be/WAoqGzVDHc0
Clean Architecture: Principles, Patterns, Practices: https://youtu.be/g2gwmqZQUkQ

Test-Driven Development in a Nutshell

Golden rule: *no test, no prod code!*

Mechanism

- [Red] - start with a failing test - [add] new constraints (assertion/test case)
- [Green] - make the test pass - [transform] behaviour of prod code to fulfil new constraint
- [Refactor] - improve prod code - [preserve] behaviour of prod code

How to move forward in TDD?

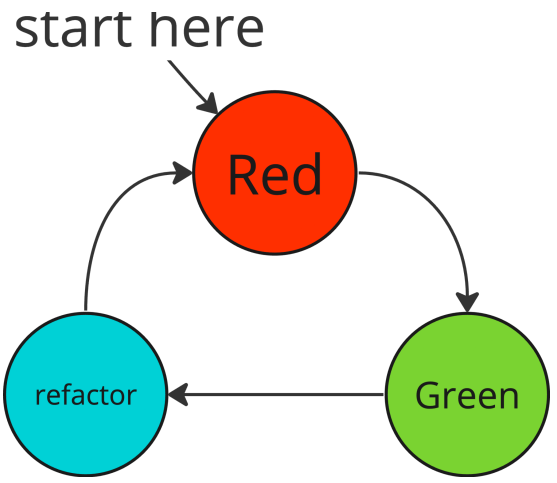
- Fake it]: just return what the test expects
- [Obvious implementation] - apply [TPP] (Transformation Priority Premise)
- [Triangulation]: write a more [specific test] to enforce more [generic prod code]

Use [TPP] to generalise prod code:

1. [{} => nil]: from no code at all to a compiling and failing test
2. [nil => constant]: return a constant instead of null/0
3. [constant => variable]: return a variable instead of a constant
4. [unconditional => selection]: add branching to code (e.g., if)
5. [value => list]: turn a single into multiple values
6. [statement => recursion]: turn a single statement into a recursive statement
7. [selection => iteration]: turn a single if into a loop
8. [value => mutated value]: mutate the value of a variable

Create FIRST Unit tests!

- [F]ast
- [I]solated
- [R]epeatable
- [S]elf-validating
- [T]imely



Structure your tests as

- AAA: [A]rrange – [A]ct – [A]ssert
- GWT: [G]iven – [W]hen –[T]hen

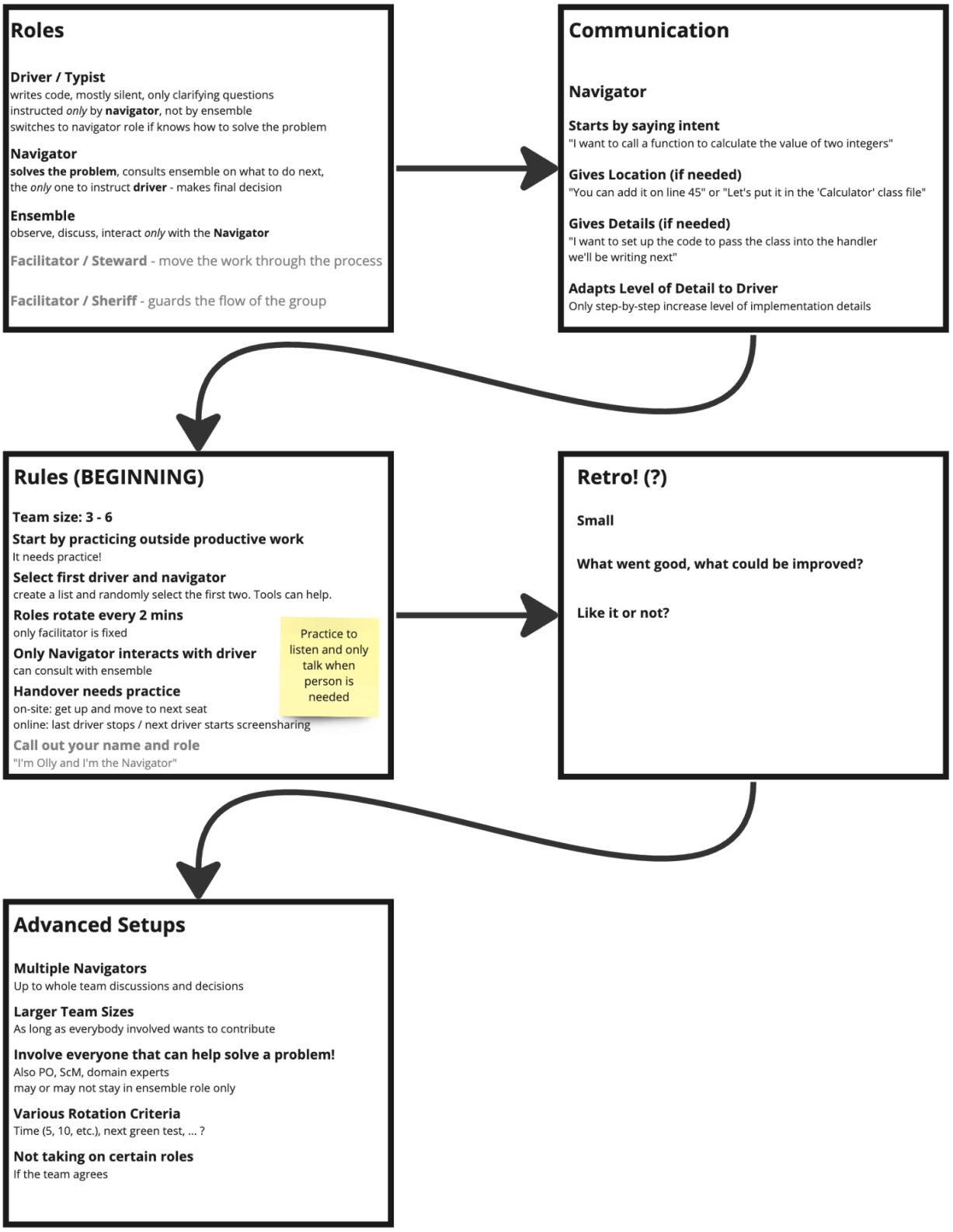
Test names should

- describe [business domain] [behaviour]
- not contain or describe [technical] details

Resources

https://www.youtube.com/watch?v=Q2qLql0dvz4
https://www.youtube.com/watch?v=TzkuDJ6ef44
Clean Craftsmanship (Uncle Bob)

Ensemble / Whole Team / Mob Programming



Resources

https://www.youtube.com/watch?v=3BcegnvAaag
https://www.youtube.com/watch?v=kyG_ljVF3cA