# Code organization

Vitalii Lakusta
July 25, 2022 | Golang Estonia

PALO ALTO
CLUB

*paloalto.club*

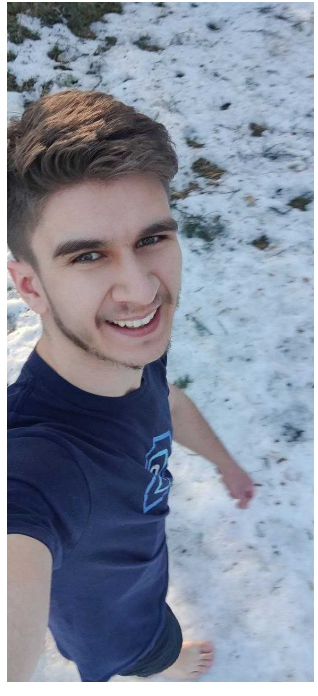
# lasn.digital

veljo@lasn.ee
+372 5803 9202

# About me

- Currently, CTO & co-founder at *BetterMedicine.ai* - we help radiologists do cancer diagnostics more accurately and faster.
- Previously in *Starship*, fleet team, where we optimized robot deliveries in real time .
- Before Starship, in fintech (*Transfer)Wise*. Worked there on public API design and implementation, integration with banks and partners. Later in anti-money laundering team 💸.

---



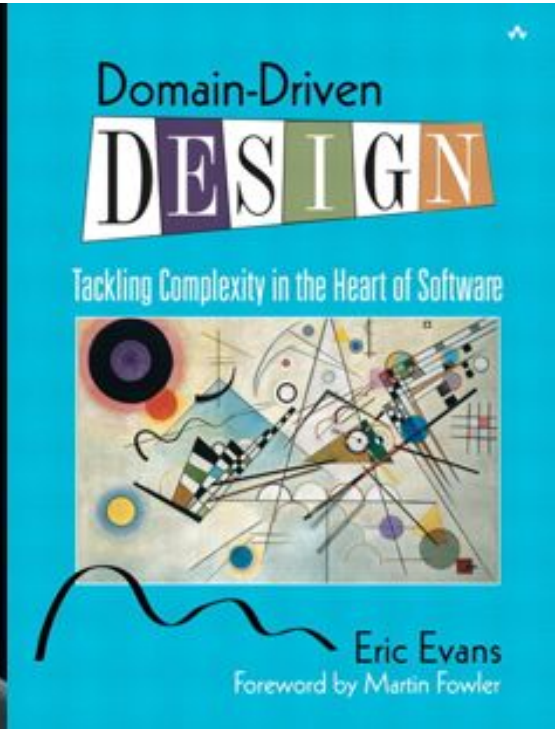
- I blog at *vlakusta.com*
- In my free time: brazilian jiu-jitsu, barefoot running (in snow too), gymnastic rings training, raising my 4-year old son Luka

# Topics covered

- Domain-driven design
- Code org in the context of microservices vs modular monolith
- Hexagonal architecture (Robert Martin)
- Dependency tree. Avoiding circular dependencies
- Code organization for prototyping vs long-term maintenance
- Further recommended readings
- Q & A

# History

DDD concepts first described by Eric Evans in his book (year 2003).



- DDD does not require a specific architecture—only one that can separate the **technical (infrastructure) concerns** from the **business concerns**.

- DDD centers the development on programming a domain model that has a rich understanding of the processes and rules of a domain.

Technical concerns (infrastructure layer)

- Database session
- AWS Session
- Configuration management
- Logging
- Metrics
- Messaging (Kafka, NanoMsg, ROS messaging etc.)

Business concerns

- Auth Logic: Does this user have authorized access to feature *ABC*?
- Pricing logic: have we applied the discount for this customer?


**FIGURE 3-2:** The domain of an online auction site.

*reference: Patterns, Principles, and Practices of Domain-Driven Design, by Scott Millett*

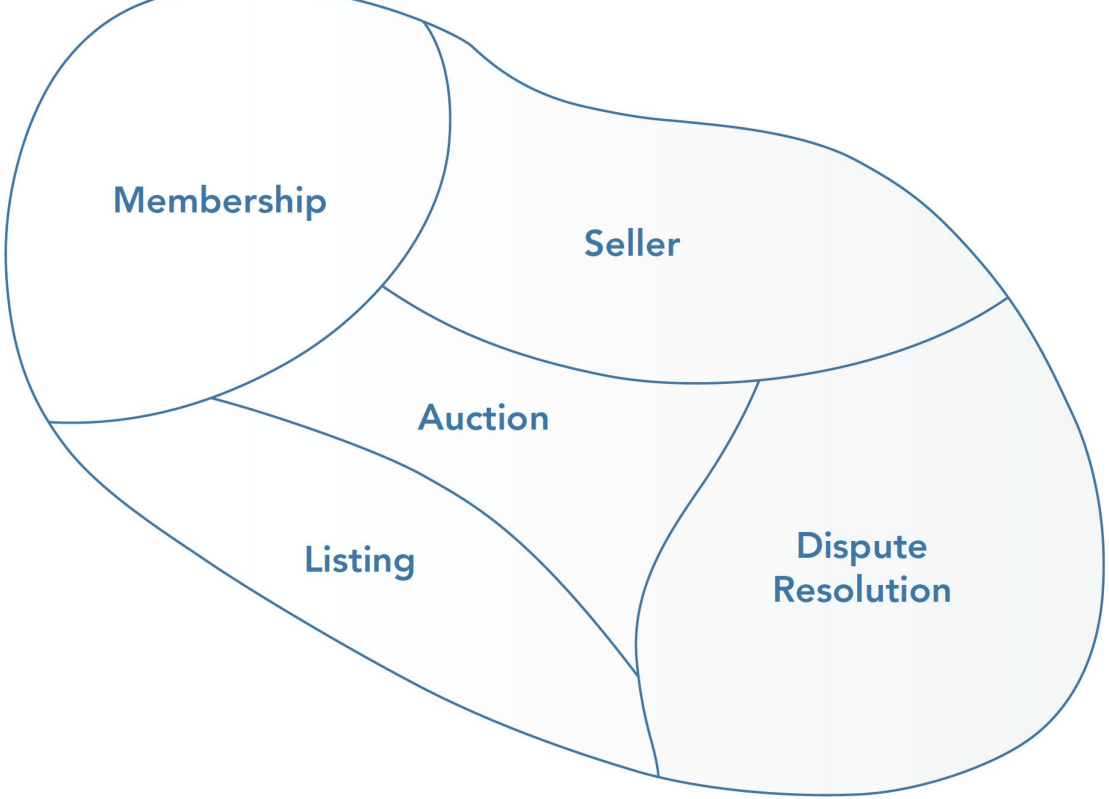



**FIGURE 3-3:** The domain of an online auction site distilled into subdomains.

*reference: Patterns, Principles, and Practices of Domain-Driven Design, by Scott Millett*

# Monolith

- Start with a monolith.
- Once your sub-domains are becoming explicit, it’s a sign to split.

# Bounded context

- Contains a well defined sub domain.
- Has as **little coupling** to other bounded contexts as possible
- **Fulfills business use-cases** in its sub domain (**service-oriented**, not just CRUD).
- Usually has its own data source, if necessary to fulfill the use-cases

- Each bounded context of the system **integrates through well-thought out interfaces and boundaries.**

# Bounded contexts are like countries

Treat bounded contexts like the **borders of a country**. Nothing should pass in to the bounded context unless it goes through the border control and is valid. Just like **countries where people speak a different language**, so does the code within your bounded context.

Be on your guard in case people try to bypass your borders and don’t adhere to your rules and language!

# What is Contribution of DDD?

**Strategic patterns** - how do we **split monolith into bounded contexts**, and how do these **bounded contexts talk to each other**?

**Tactical patterns** - how do we organize code **inside of a bounded context**?

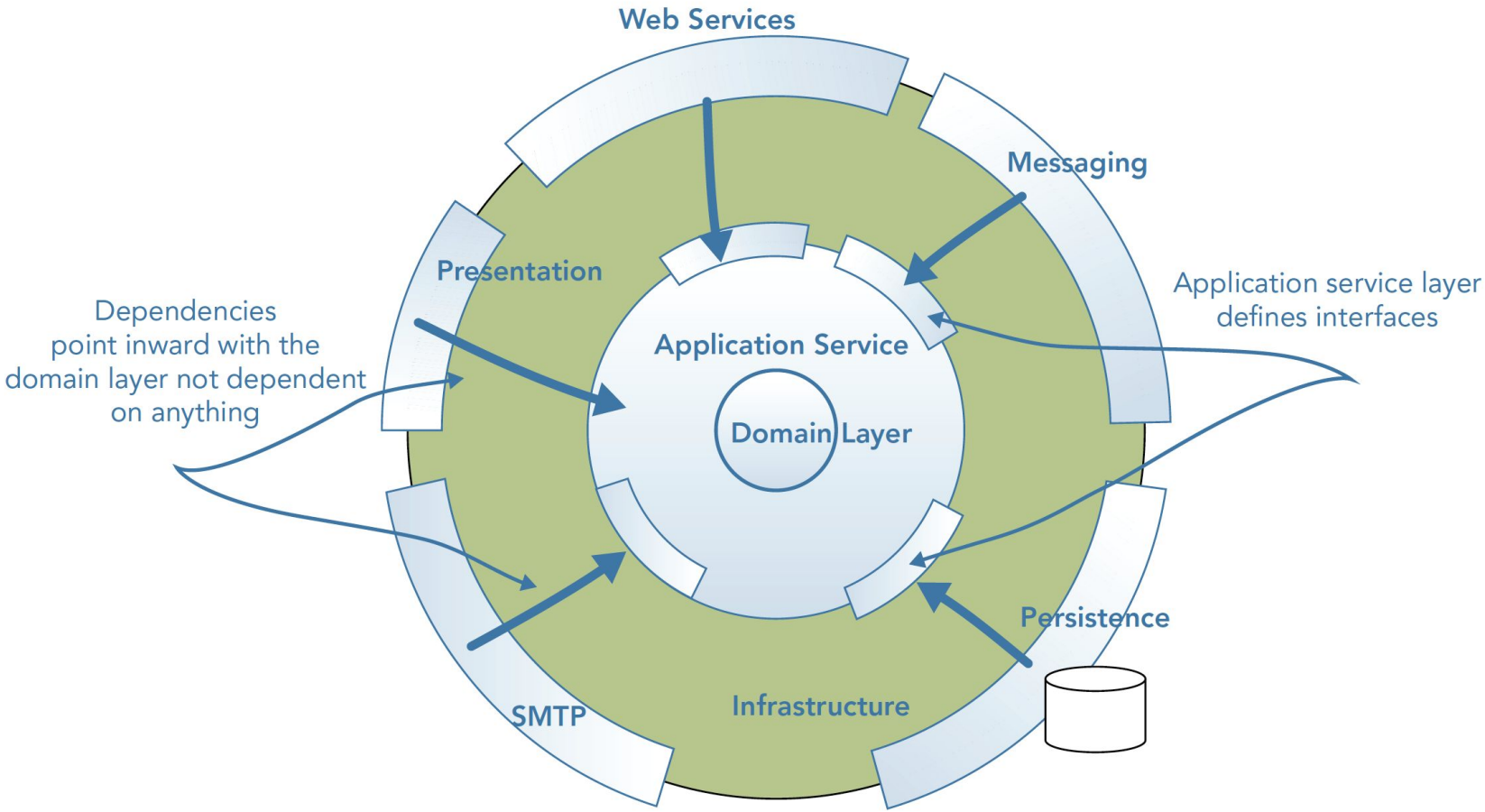
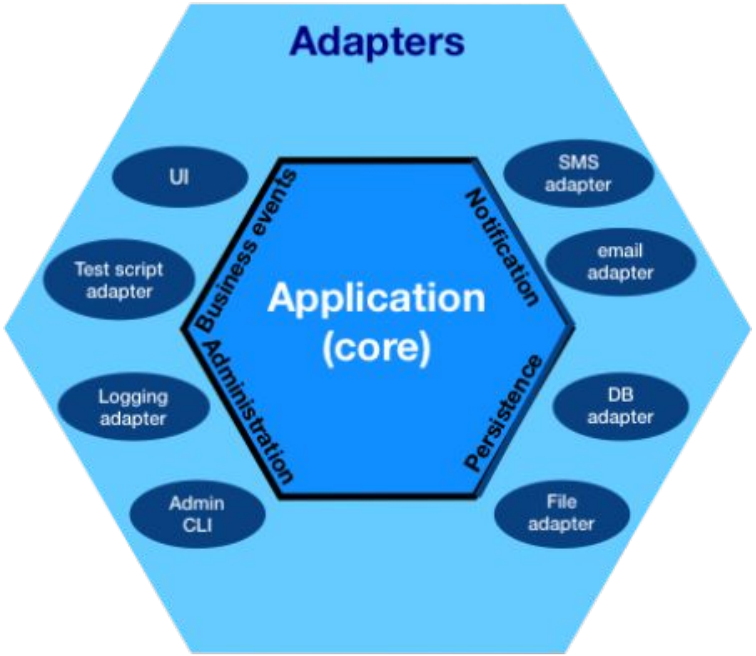



**FIGURE 8-2:** Dependency inversion within a layered architecture.

*reference: Patterns, Principles, and Practices of Domain-Driven Design, by Scott Millett*

# Hexagonal Architecture (Robert Martin)




- Uses the same principle of dependency inversion
- Divides a system into several loosely-coupled interchangeable components, such as the application core, the database, the user interface, test scripts and interfaces with other systems.

*Clean Architecture*, by Robert Martin

# Bounded Context <> Ubiquitous Language

Bounded contexts are important because they allow us to define an **ubiquitous language** that is shared and **valid within a boundary.**

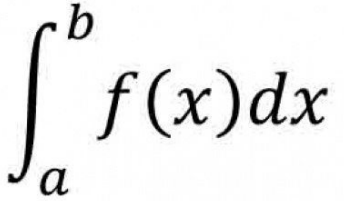
# Ubiquitous language

- Create a **glossary of domain terms** with the domain expert to avoid confusion and to help make concepts explicit.
- Carve out a language by working with concrete biz use-cases.
- Ensure that you have **linguistic consistency**. If you are using a term in code that the domain expert doesn’t say, you need to check it with her. It could be that you have found a concept that was required, so it needs to be added to the UL and understood by the domain expert. Alternatively, maybe you misunderstood something that the domain expert said; therefore,you should **rectify the code with the correct term**.

# Ubiquitous language: evolution, cognitive load

- As you gain a deeper understanding of the domain you are working in, your **UL will evolve**. **Refactor** your code to embrace the evolution by using more intention-revealing method names. If you find a **grouping of complex logic** starting to form, talk through what the code is doing with your domain expert and see if you can **define a domain concept for it**.

Example from math: integral represents a grouping of complex logic! This **domain concept reduces cognitive load** in the long term significantly!



Bounded Context = Microservice

Bounded Context = Microservice

*or start with …*

Bounded Context = a separate module in a modular monolith

*… and once certain, feel free to move a bounded context into a separate microservice*

# Context mapping

Answers questions:

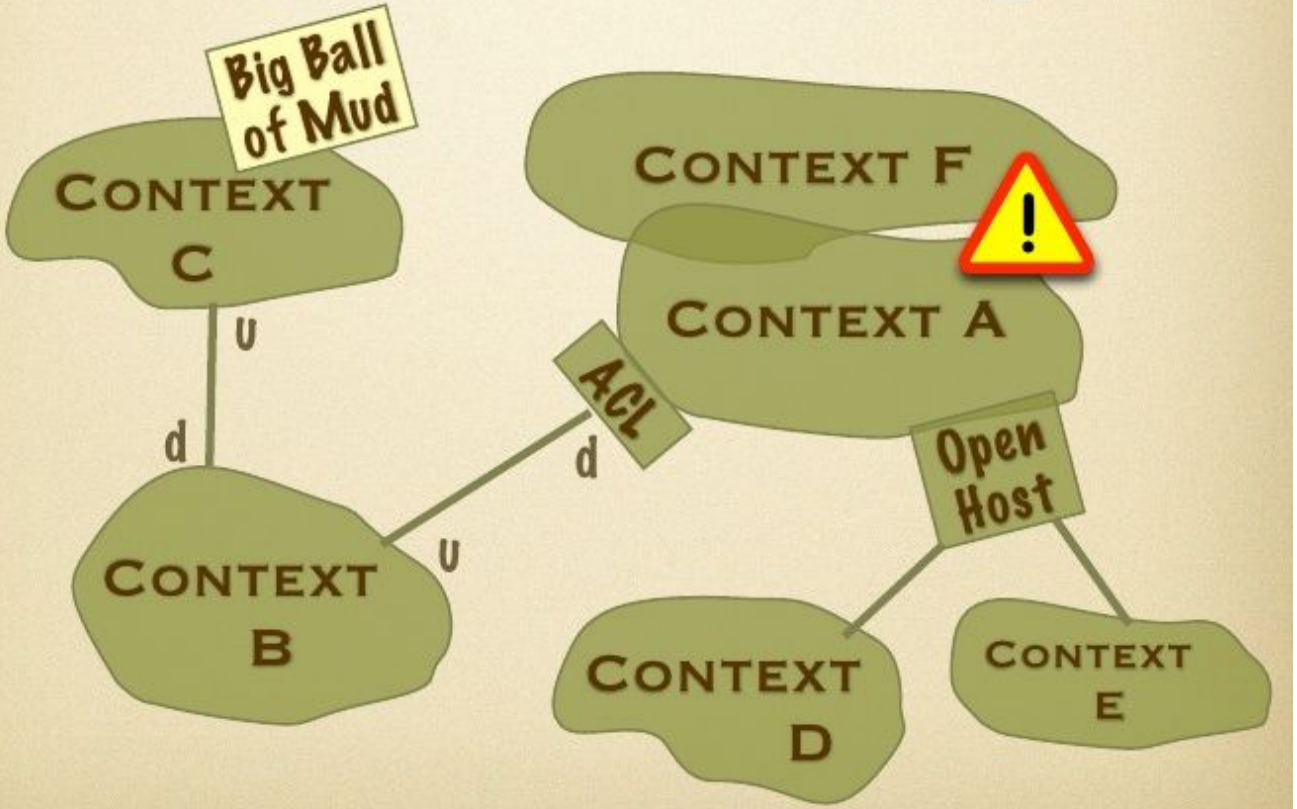
- **What** other bounded contexts do I speak to?
- What is a **relationship** with the bounded contexts I speak to?

Your context map represents your **not perfect reality**. This is the most important thing.

Next step? Look at your map, and decide what to do next - **strategize**.

- Do you need to **change relationship** with the services you speak to?
- Maybe the map gives you a hint for the next split from the monolith?

# Our context map



# Context mapping: relationships, terms

- **Partnership** - when teams in 2 contexts succeed or fail together
- **Customer/Supplier** - upstream-downstream
- **Shared Kernel** - sharing a part of the model
- **Conformist** - upstream has no motivation to provide for downstream team's needs
- **Open-Host** - open for everyone that needs to integrate with your system
- Anti Corruption Layer (**ACL**) - isolating layer to limit impact of changes in an upstream system.
- Further reading - Crisp View of Context Map

# Code organization for prototyping vs long-term maintenance

- When prototyping - experiment, fail-fast, less layers.
  - Still optimize for necessary testability, strive for very flat, minimal hierarchy
- Long-term maintenance - refactor your MVP/prototype continuously into a structure that separates application-logic concerns from technical/infra concerns. Use principles from DDD, hexagonal architecture etc.
  - For example, start by refactoring direct DB access into a repository layer with abstract interfaces

```
type Repository interface {
    FindAll(ctx context.Context) ([]Study, error)
    FindById(ctx context.Context, id int) (*Study, error)
    Create(ctx context.Context, data Study) (*Study, error)
}
```

```
studies, err := r.repository.FindAll(ctx)
```

# Wrapping everything up so far:

- Microservice = **bounded context.** Or module in monolith = bounded context.
- **Define boundary (API)** of your bounded context
- Integrate with other bounded contexts through **well-thought out interfaces**
- Use **ubiquitous language** inside your bounded context to crystalize your domain terms in your context.
- Use **context mapping** to see the current reality, and strategize for further actions!
- Inside a bounded context, use **dependency inversion** to separate infrastructure concerns from business concerns.

# Avoiding circular dependencies in Go

- Good news! Go won't let you have circular dependencies : )
- Result - engineers are forced to think about a better/simpler software design that avoids circular dependencies.

---

*Ref: https://www.reddit.com/r/golang/comments/ajpj53/whats_an_idiomatic_way_to_solve_this_circular/*

```
global/
    env.go
router/
    router.go
data/
    data.go
main.go
```

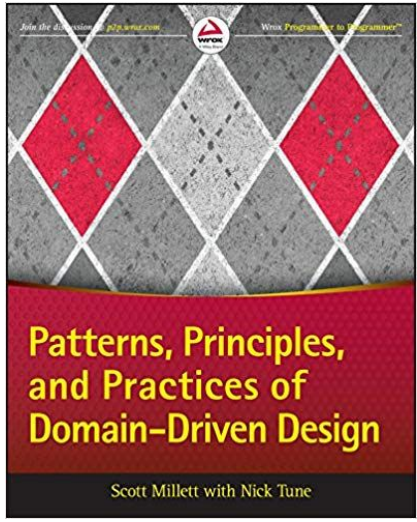
the dependencies are as follows

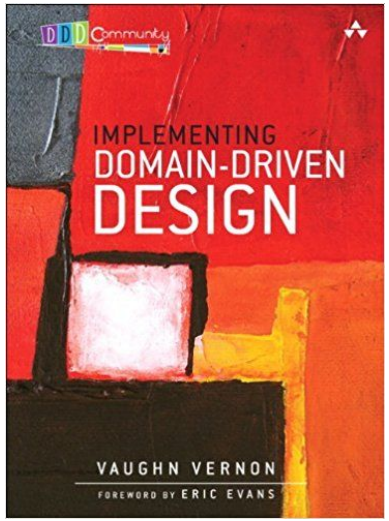
```
main -> global/ data/ router/
global/ -> data/ router/
router/ -> global/
```

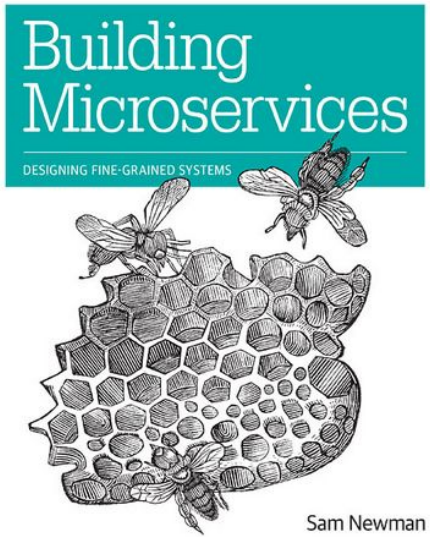
- The issue is that *global/* depends on *router/* to get some structs definitions.
- But *router/,* in turn, depends on *global/* to get the database connection.
- → bad design

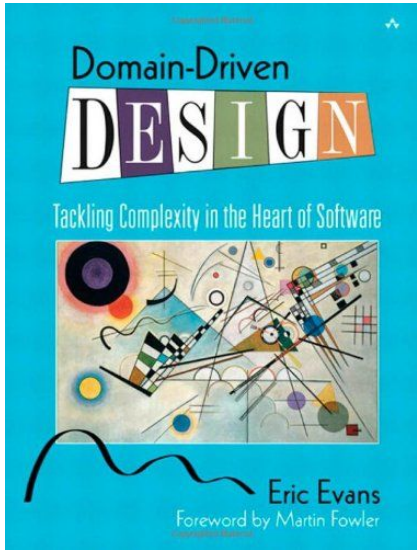
# Further reading

1. https://martinfowler.com/articles/break-monolith-into-microservices.html
2. https://martinfowler.com/bliki/BoundedContext.html
3. https://martinfowler.com/bliki/UbiquitousLanguage.html



# Further reading

1. **https://github.com/golang-standards/project-layout**
2. https://dzone.com/articles/ddd-thinking-in-terms-of-context-map
3. https://egonelbre.com/thoughts-on-code-organization/
4. https://egonelbre.com/psychology-of-code-readability/

Thank you. Q & A time!