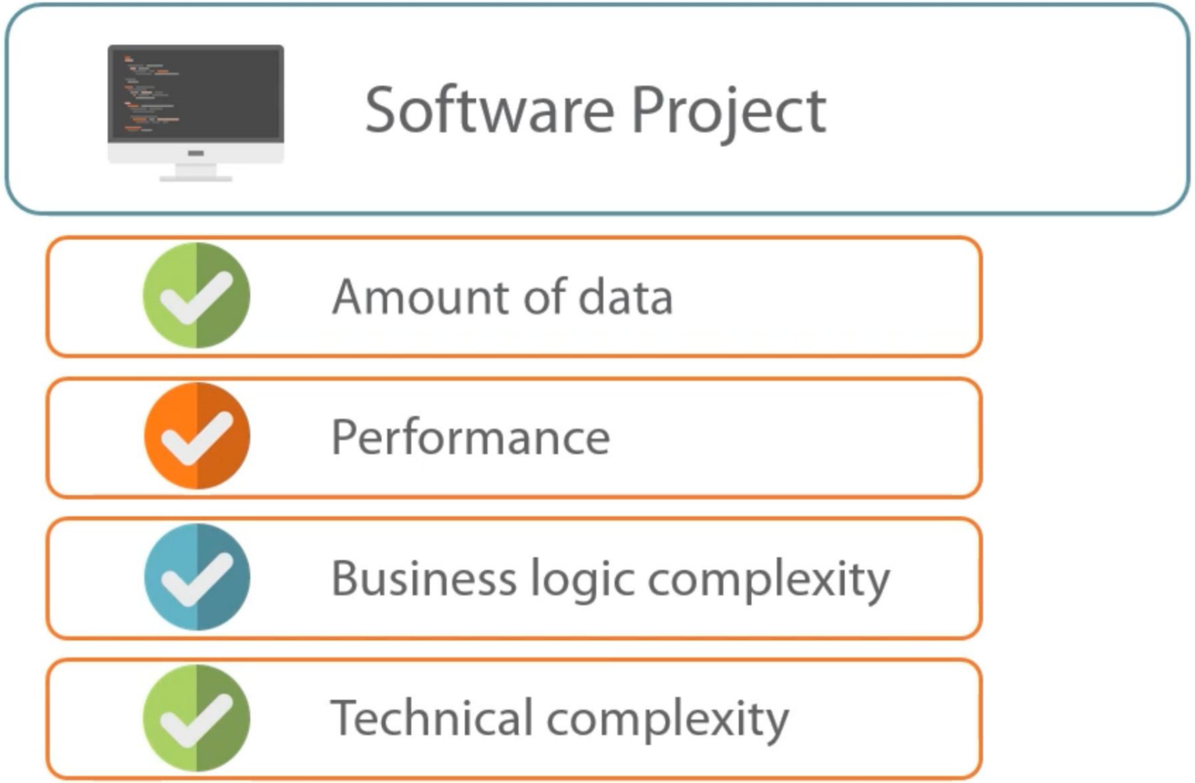
Strategic Design: the big picture

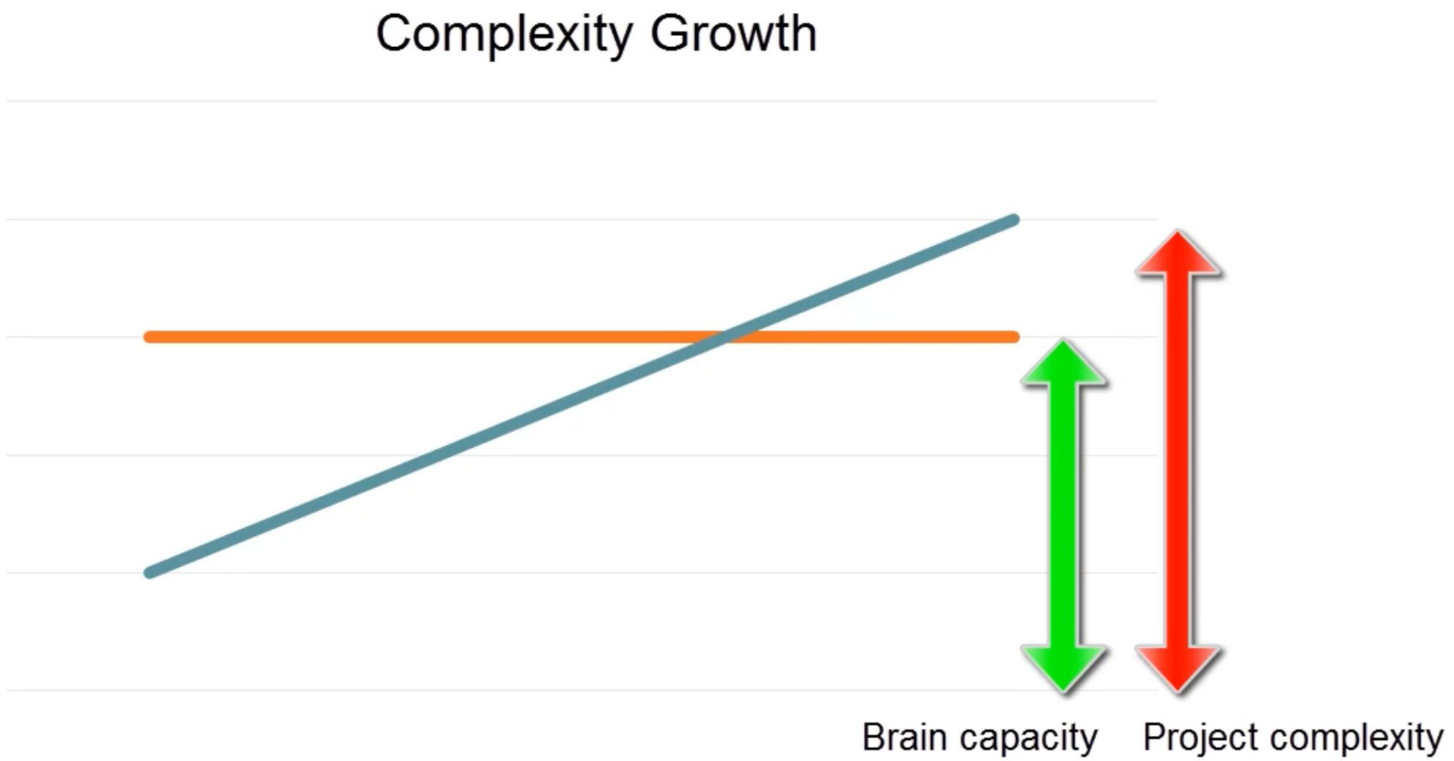
Complexity



Software Project

- ✓ Amount of data
- ✓ Performance
- ✓ Business logic complexity
- ✓ Technical complexity

Complexity



Complexity

YAGNI

- ☐ You are not gonna need it
- ☐ Shortening development time

KISS

- ☐ Keep it short and simple
- ☐ Maintainable code

DDD

- ☐ Focus on essential parts
- ☐ Simplifying the problem

Value Proposition of DDD

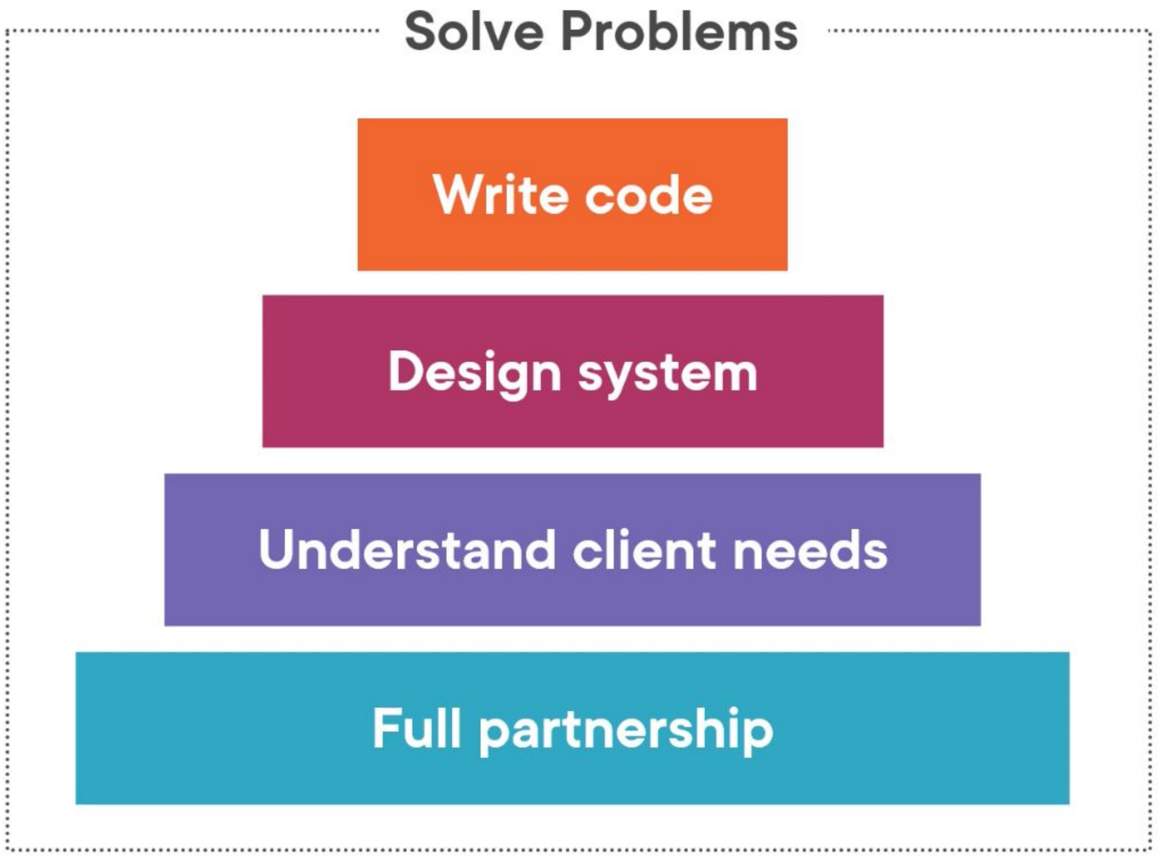
Principles & patterns to
solve difficult problems

History of success
with complex projects

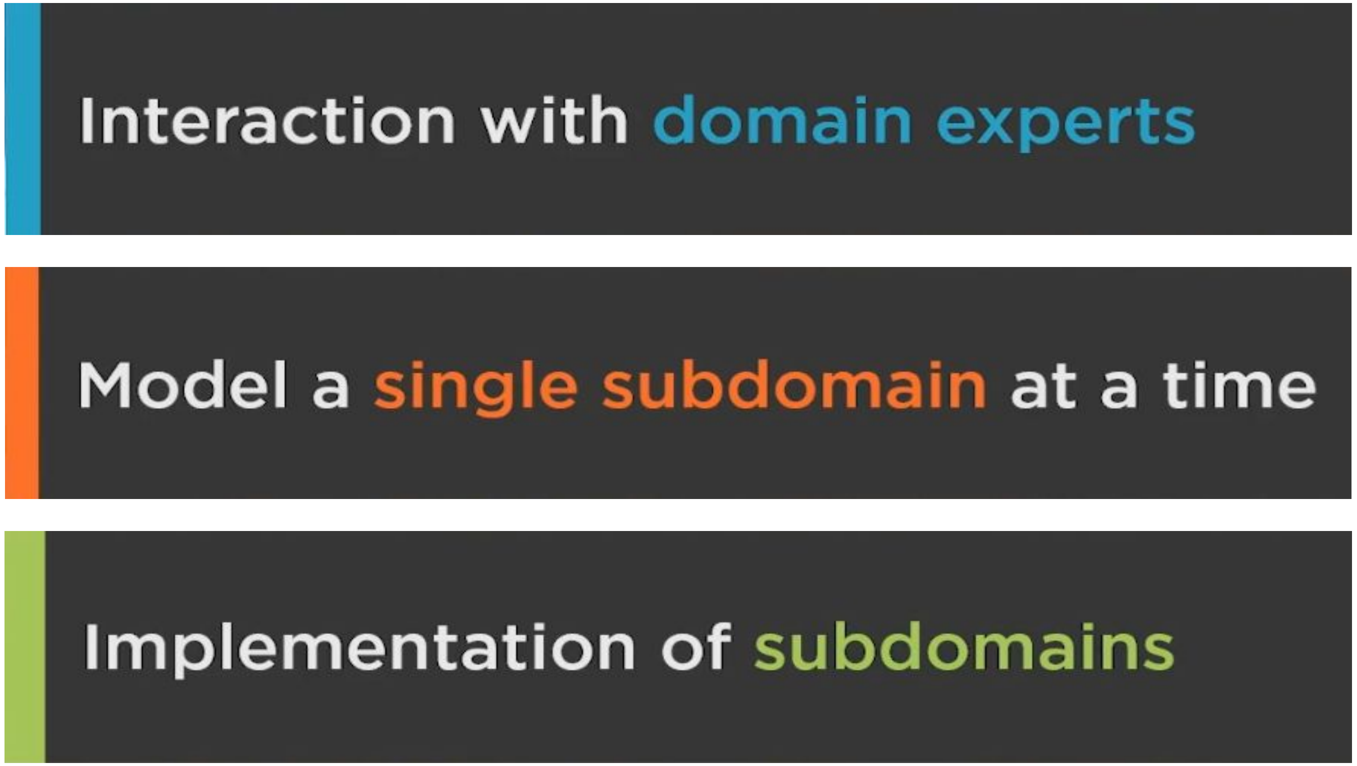
Aligns with practices
from our own experience

Clear, readable, testable code
that represents the domain

Value Proposition of DDD



High Level View of DDD

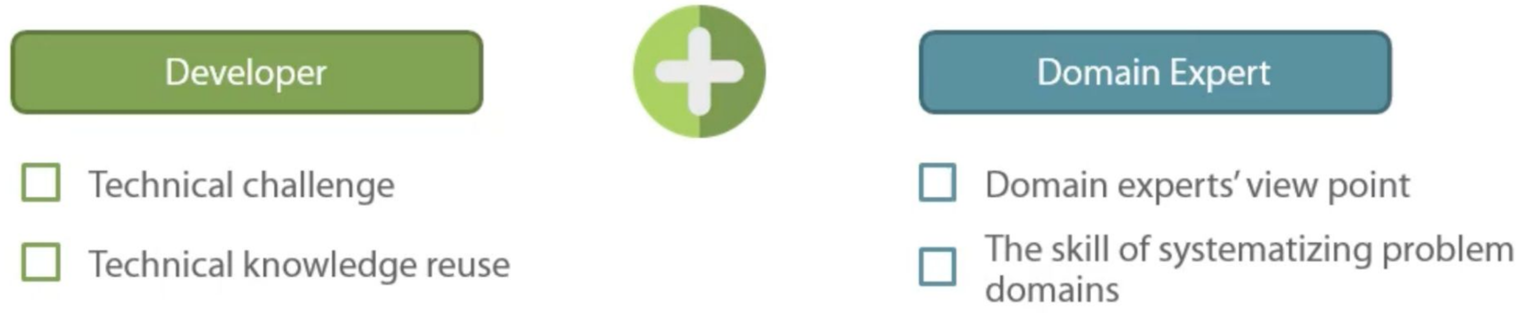


Interaction with domain experts

Model a single subdomain at a time

Implementation of subdomains

High Level View of DDD



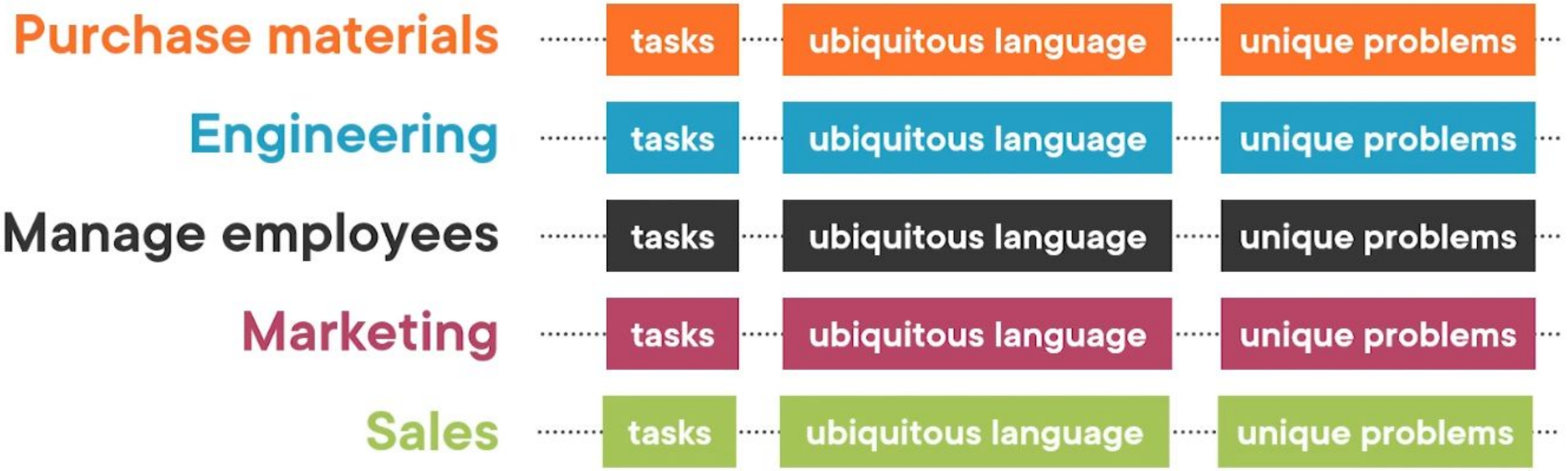
Developer

- ☐ Technical challenge
- ☐ Technical knowledge reuse

Domain Expert

- ☐ Domain experts' view point
- ☐ The skill of systematizing problem domains

High Level View of DDD



Benefits of Domain Driven Design

Flexible

Customer's vision/perspective of the problem

Path through a very complex problem

Well-organized and easily tested code

Business logic lives in one place

Many great patterns to leverage

Benefits of Domain Driven Design

DDD is for handling complexity in business problems

Not just CRUD or data-driven applications

Not just technical complexity without business domain complexity

Benefits of Domain Driven Design

Understand client's business

Identify processes beyond project scope

Look for subdomains we should include

Look for subdomains we can ignore

Benefits of Domain Driven Design

Important to get on the “same page” with the domain expert

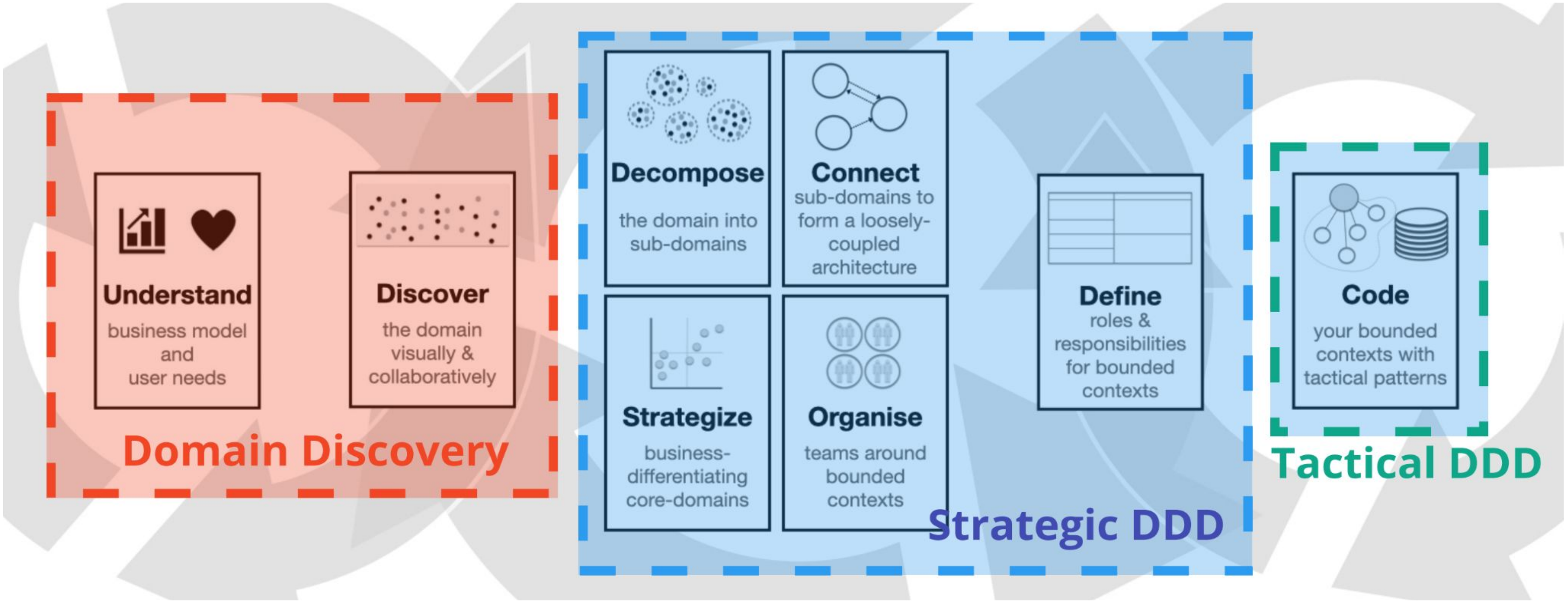
The customer gained better understanding of their own business process by describing it in terms we could understand and model

Avoid speaking in programmer terms

At this stage, the focus is on how the domain works, not how the software will work

Make the implicit knowledge of domain experts explicit

Strategic Design vs Tactical Design



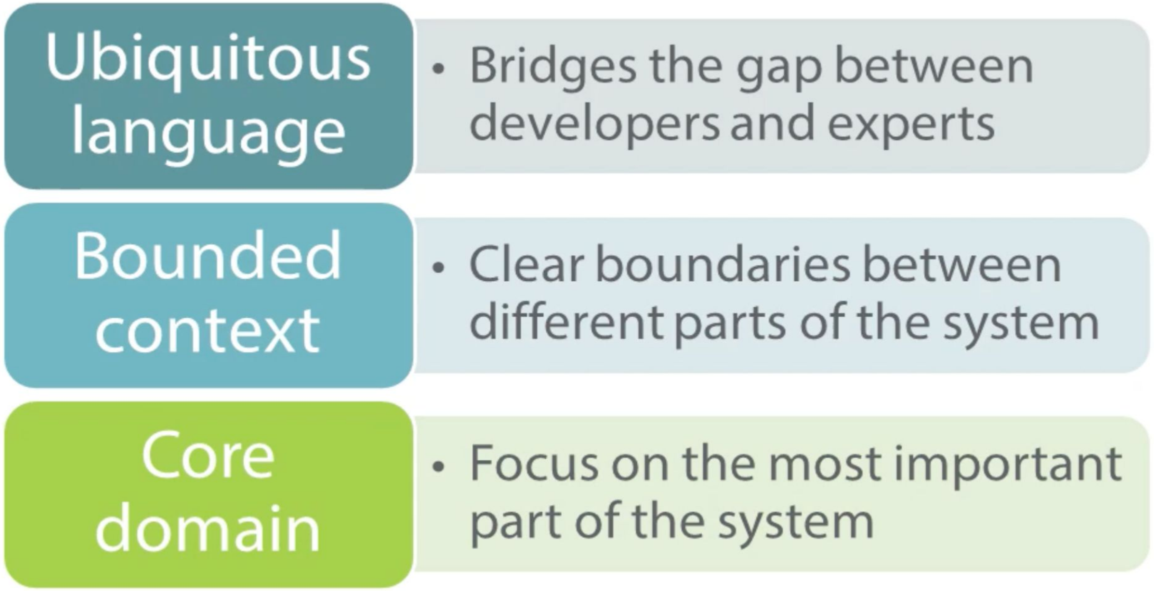
Understand
business model and user needs
Discover
the domain visually & collaboratively
Domain Discovery
Decompose
the domain into sub-domains
Connect
sub-domains to form a loosely-coupled architecture
Strategize
business-differentiating core-domains
Organise
teams around bounded contexts
Define
roles & responsibilities for bounded contexts
Strategic DDD
Code
your bounded contexts with tactical patterns
Tactical DDD

Strategic Design vs Tactical Design

- The DDD **strategic development** tools help you and your team make the competitively best **software design choices** and **integration decisions** for your business.

- The DDD **tactical development** tools can help you and your team design useful software that **accurately models the business’s unique operations**.

Strategic Design Tools



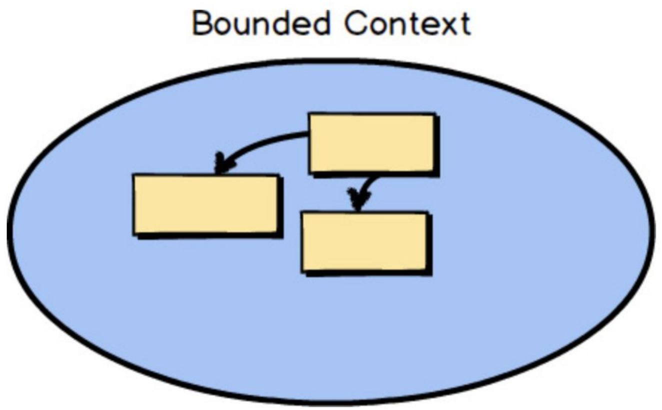
Ubiquitous language	• Bridges the gap between developers and experts
Bounded context	• Clear boundaries between different parts of the system
Core domain	• Focus on the most important part of the system

Bounded Context

In short, DDD is primarily about modeling a Ubiquitous Language in an explicitly Bounded Context.

The components inside a Bounded Context are context specific.

You develop your solution in the Bounded Context as code, both main source and test source.

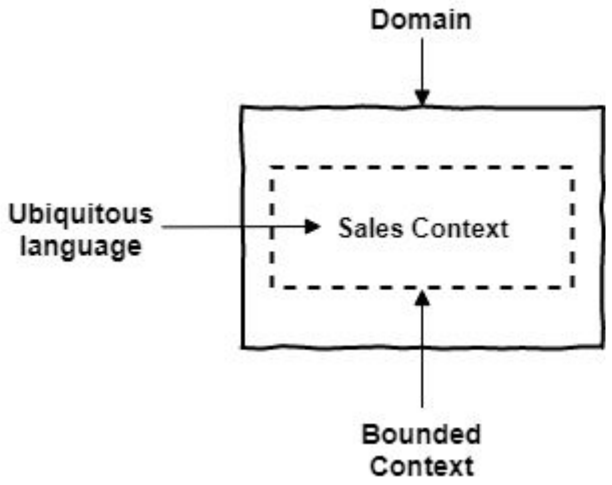


Characteristics of Bounded Context

- There should be **one team assigned** to work on one Bounded Context.
- There should also be a **separate source code repository** for each Bounded Context.
- There should also be a **separate database schema** for each Bounded Context.
- There should also be **separate unit test cases** for each Bounded Context.
- There should also be **separate acceptance test cases** for each Bounded Context.

Ubiquitous Language

Ubiquitous Language is modeled within a Limited context, where the terms and concepts of the business domain are identified, and there should be no ambiguity.



Characteristics of the Ubiquitous Language

- Ubiquitous Language must be expressed in the Domain Model.
- Ubiquitous Language unites the people of the project team.
- Ubiquitous Language eliminates inaccuracies and contradictions from domain experts.
- Ubiquitous Language is not a business language imposed by domain experts.
- Ubiquitous Language is not a language used in industries.
- Ubiquitous Language evolves over time, it is not defined entirely in a single meeting.
- Concepts that are not part of the Ubiquitous Language should be rejected.

Common Problems

- The lack of a common language, generating “translations”, which is bad for the Domain Model, and causes the creation of wrong Domain Models.
- Team members using terms differently without realizing it, for lack of a common language.
- Communication without using Ubiquitous Language, even if it exists.
- Creation of abstraction by the technical team for the construction of the Domain Model, which is not understood by domain experts.
- Technical team disregarding the participation of domain experts in the Domain Model, considering it too abstract for domain experts.

How to develop a Ubiquitous Language?

Draw

Express your Domain on a whiteboard, do not worry if they are formal designs or not.

Create a glossary

Develop a glossary of all terms with definitions.

Use Event storming

Domain experts and developers can achieve a fast cycle of business process learning using Event Storming, which facilitates the development of Ubiquitous Language.

Review and update

Be ready to review and update what has been generated in an agile way.

Core Domain & SubDomains

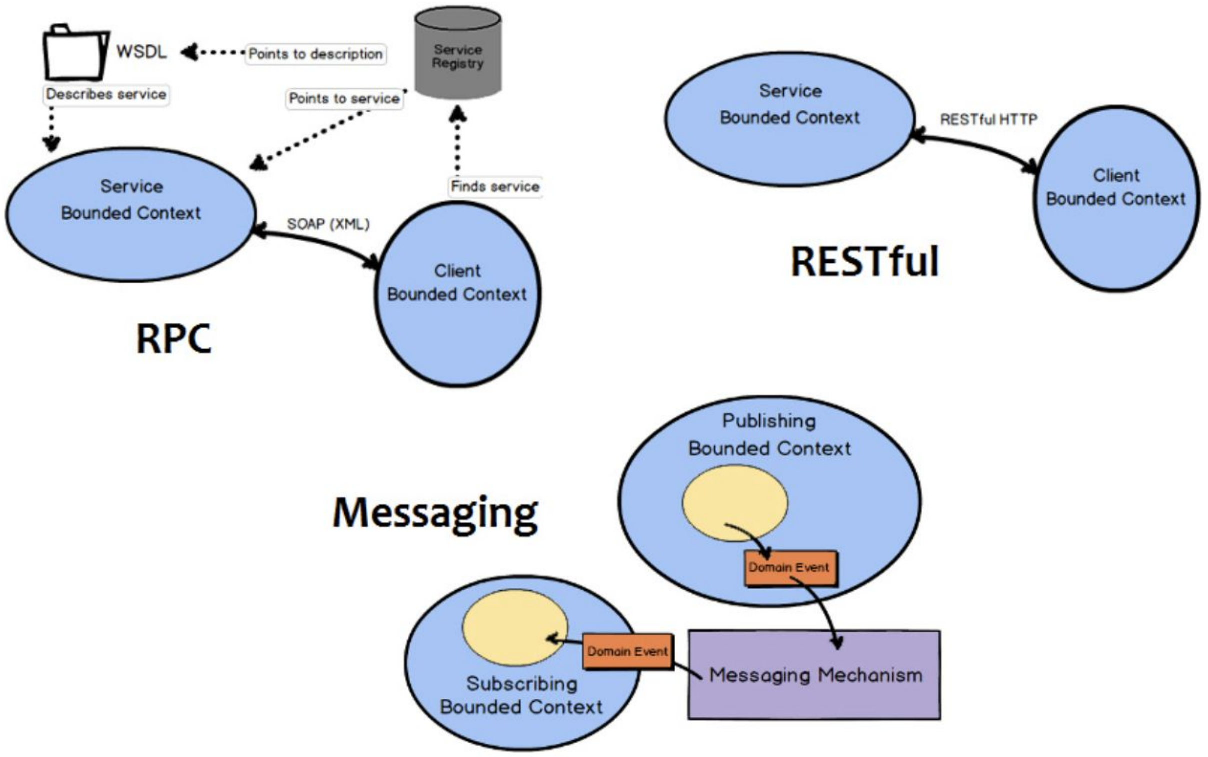
Subdomains can be used to logically break up your whole business domain so that you can understand your problem space on a large, complex project.

Core Domain: foundational concept behind the business

Generic Subdomain: these types of pieces can be purchased from a vendor or outsourced

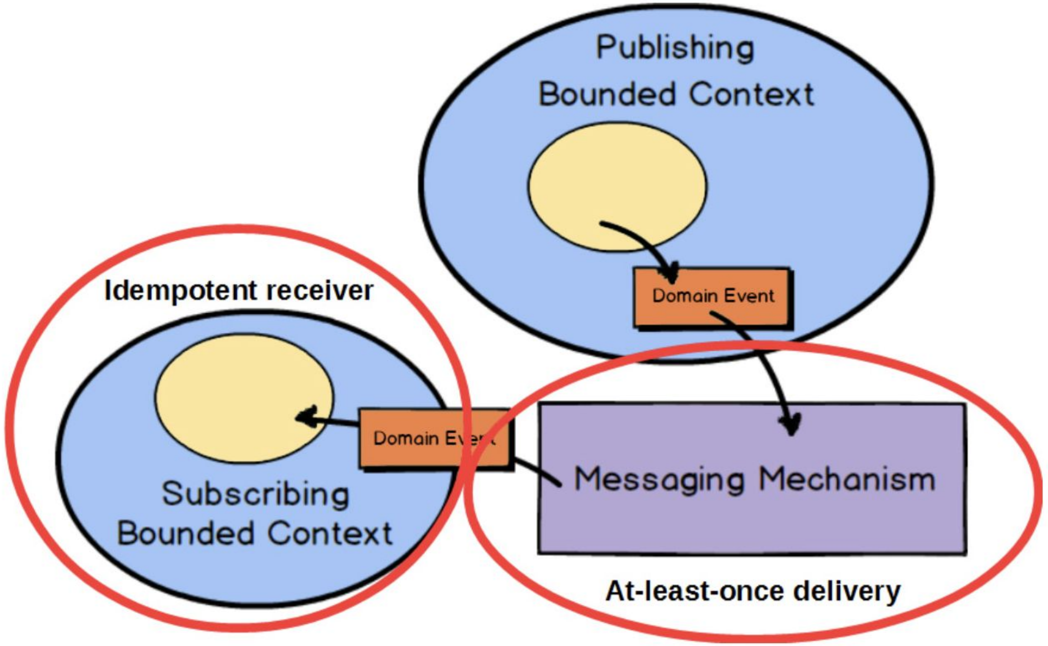
Supporting Subdomain: supporting functions related directly to what the business does

Types of Context Mapping



Messaging

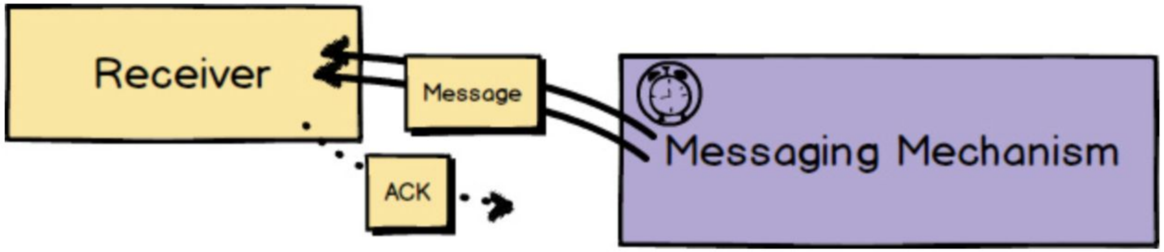
Messaging is one of the **most robust** forms of integration



At-Least-Once Delivery

At-Least-Once Delivery is a messaging pattern where the messaging mechanism will **periodically redeliver** a given message.

This will be done in cases of **message loss**, **slow-reacting** or **downed receivers**, and **receivers failing to acknowledge receipt**.

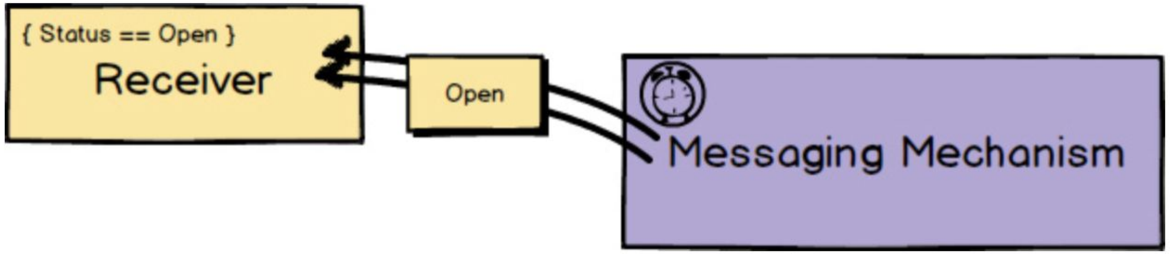


Idempotent Receiver

Whenever a message could be delivered more than once, the receiver should be designed to deal correctly with this situation.

It produces the same result even if it is performed multiple times.

Some amount of latency, non-blocking


{ Status == Open }
Receiver
Open
Messaging Mechanism