FLEX - Modularization

Simon Härer

Created: 2025-12-02 Tue 13:19

Modularization

Modularization is the key to up-scaling development:

We ultimately want independent teams to parallelize the entire development and release process.

Module

Part of a software system that is:

- tightly coupled ("cohesive") internally
- loosely coupled externally

Often: separate interface from implementation

Modularization at Source Level

- Split code into source files
- Integrate files during compilation
- Significant coupling

Code is compiled and tested as a whole. Strong dependencies between teams in the process.

@ctive group

Modularization at Source Level

- `#include` in C/C++ projects
- macros in Racket/C++/Rust etc. (less coupling if used properly)
- source-code assembly and code generation (Xtend, generation from models)
- processes in Erlang/Elixir

Modularization into Compilation Units

- Split code into separate compilation units
- Offer code as libraries
- Enforce interface/implementation separation
- Allows semi-independent development

This level of modularization helps some teams develop and test code independently.

However, at some point dependencies need to be resolved and errors in one compilation unit affects releases and can delay features created by other teams.

Modularization into Compilation Units

- `.o` / `.a` / `.lib` / `.so` files for C/C++ projects
- `.class` / `.jar` files in Java projects
- OSGi bundles
- ”plugins”
- module systems with imports/exports in any number of languages
- Haskell type classes
- parametric module systems in SML/OCaml

Modularization into Executables

- Can co-exist with other levels of modularization
- Each unit is an independent application on its own
- Each unit is developed by one team
- Interaction between units is mapped entirely at run time

With this strategy, most of the dependencies can be removed from development.

Testing, compilation and deployment are independent and teams can design their application regarding their specific needs and requirements.

Modularization into Executables

- OS executables
- OS packages
- OS service
- appropriately annotated `jar` files with any of the above
- VM images
- Erlang applications
- Docker containers

Labelling of Building Blocks

"Source file", "compilation unit", "executable" are only coarse-grained labels for building blocks. As are "library", "web service", "microservice".

Making decisions about the use of a certain technical solution for modularization requires more characterization.

For example, "module systems" differ vastly in capabilities. (Separate interfaces, independent compilation, type abstraction, higher-order modules etc.)

On the other hand, "microservice" and "Erlang process" may share many pragmatic similarities.

@ctive group

Consequences

- Level of independence between developers/teams
- Integration effort
- Effort to move units between boundaries
- Performance
- Automatic checking of interface conformance
- Effort involved in distribution

Autonomy of Modularization Units

Strategy	At development time	At run time
Source Code	low	non-existent
Compilation Units	medium	non-existent
Executables	high	depends on interface coupling

@ctive group

Integration Effort

Strategy	Effort	Comment
Source Code	low	compiler can spot mistakes early
Compilation Units	medium	type system can spot mistakes early
Executables	high	interfaces need to be generated,
		checked, or tested

Effort needs be to be offset by advantages!

Cost of Moving Boundaries

Strategy	Cost	Comment
Source Code	low	just move the code
Compilation Units	medium	must change interface
Executables	high	must move to other project,
		adjust dependencies,
		adjust API etc.

Macro Architecture

Decisions affecting all systems

- operating platform
- communication infrastructure
- protocol-description format
- documentation standards

Too many or too strict decisions restrict the applicability of the macro architecture, and the suitability for certain problems

@ctive group

Micro Architecture

Architecture of a single system, independent from architecture of other systems.

Functional or Technological

Functional split:

- Requirements can often be handled by a single expert team.
- As a result, we have independent development of most features.
- As a result, we have independent deployments of the services, enabling a higher rate of deployment.
- End-to-end responsibility leads to higher motivation in teams and strong domain experts.

Functional or Technological

In Erlbank we have a technological modularization: A classical layer architecture. Each team is responsible for one layer and most probably has their own technology stack to work with.

If there is a requirement, most probably *all* teams need to be involved, since all layers have additions or need to be changed. We want to avoid that. Thus, a functional split is the recommendation.

How do we find a good functional split?

@ctive group

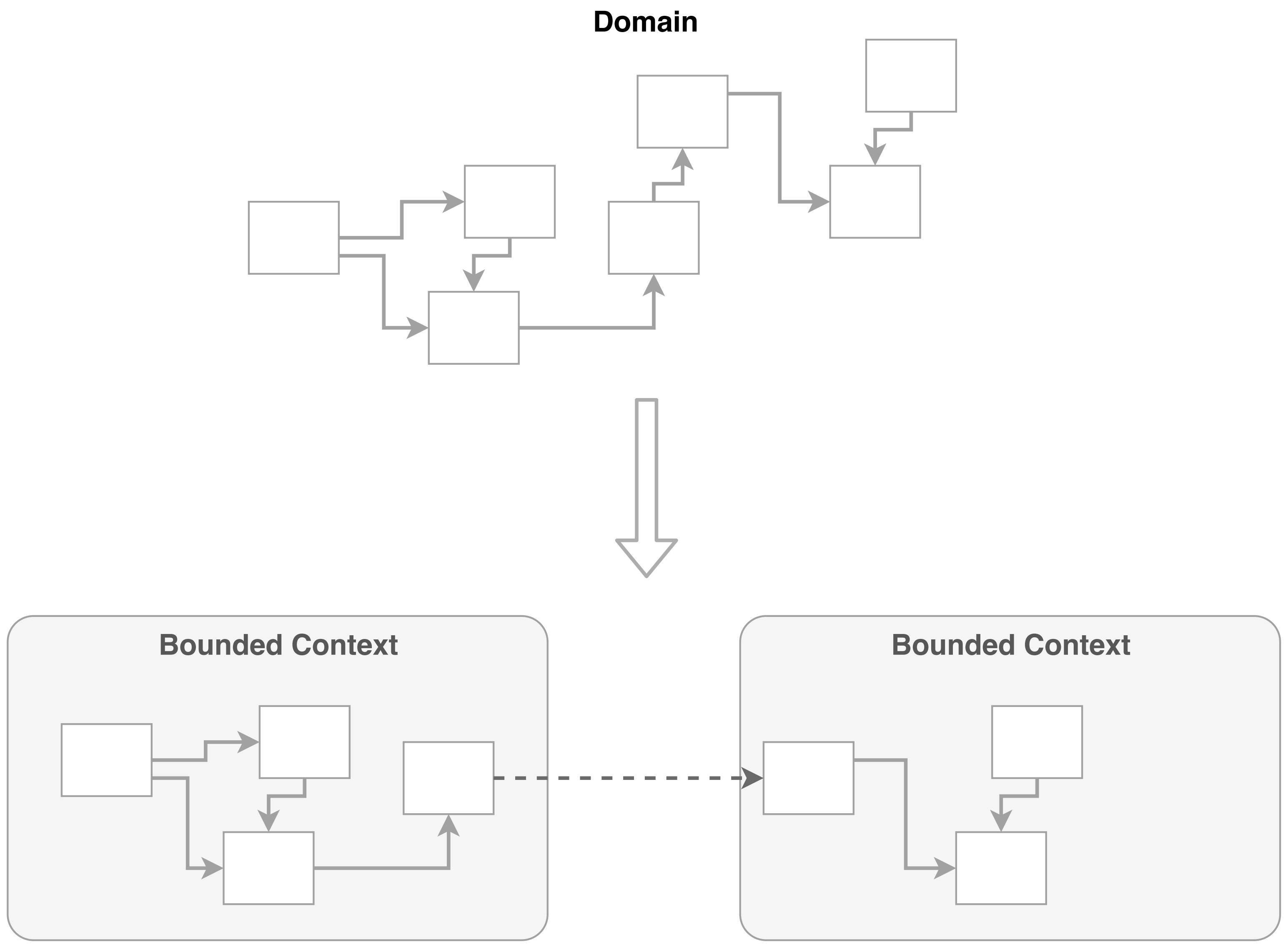
Bounded Contexts

- Multiple Models exist in a large project.
- Bounded Context is the part of the project where one model applies.
- Possibly separate code base, database schema, development process

Eric Evans: *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley Professional, 2003.

@ctive group

Bounded Contexts



@ctive group

Domain Storytelling

Story-based method to model a domain & find bounded contexts

- Listen to the story told by domain experts
- Participants learn terminology from domain experts
- Stories are visualized using pictographic language

https://domainstorytelling.org/

Domain Storytelling

1. Every sentence starts with an actor who initiates an activity
2. A work object or a piece of information is something the actor does something with. A work object is visualized by an apposite pictogram
3. Connect the actor and the work object with an arrow. Name the arrow according to the activity

Domain Storytelling

1. If multiple actors are involved, draw another arrow from the work object to the other actors. Mark them with prepositions such as "with".

   The basic pattern is *subject - verb - object*.

2. Tell a story by numbering the arrows and giving the story an order. Each actor should only appear once in a story. If a work object appears in multiple sentences, draw it multiple times.

Exercise: Domain Storytelling on Erlbank

Try domain story telling in Erlbank. Either take a partner or the trainer as a domain expert that is interviewed and tells the story.

- Identify Bounded Contexts
- What was easy/hard?
- Where are shortcomings of Domain Storytelling?

Event Storming

Brain-storming-based method to model a domain & find bounded contexts

- Without a computer, but sticky notes
- Urges developers to ask questions
- Different colors for different stickies:
    - Domain event
    - Command
    - Aggregate
    - View
    - …

Event Storming: Process

1. Create domain events: What can happen in the domain?
2. Add commands to domain events: What is the trigger for the events?
3. Add actors to commands: Who gives the commands?
4. Add aggregate to domain-command-actor triples: An aggregate is a cluster or group that pools what can be treated as a single unit.

Boundaries & Communication in DDD

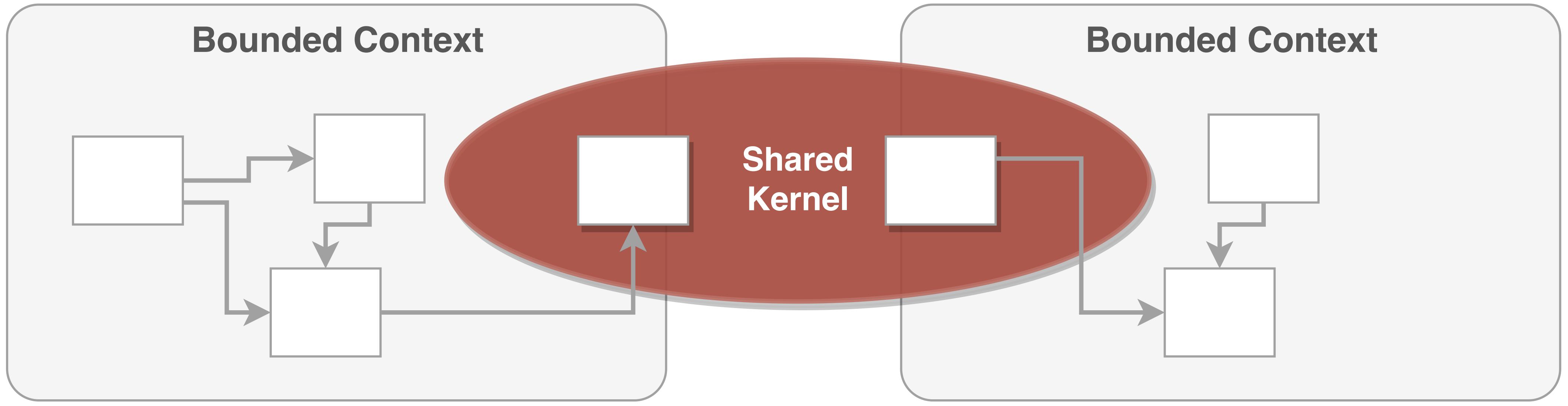
Bounded contexts group a domain in different contexts. These contexts still have to be connected somehow. That is, your system's contexts have to communicate and share data. Strategic design patterns are a way to describe how the communication can be coordinated.

Strategic Design Patterns

- Shared Kernel
- Customer/Supplier
- Conformist
- Anti-corruption Layer
- Separate Ways
- Open Host Service

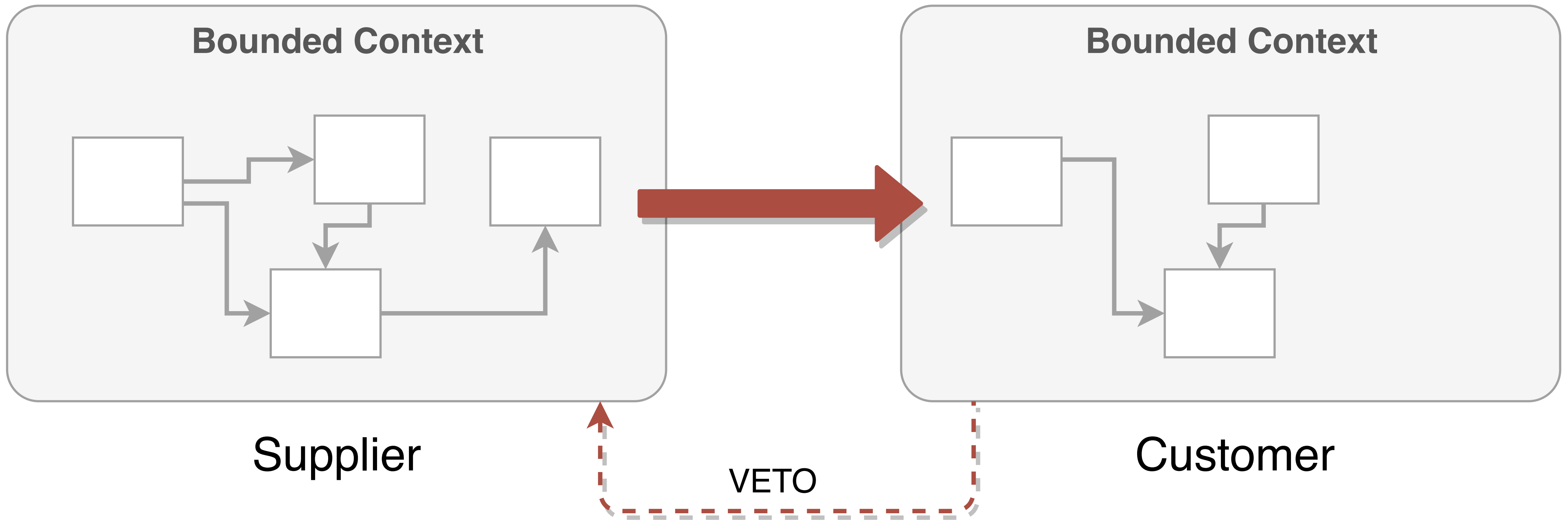
Scott Wlaschin: *Domain Modeling Made Functional*, The Pragmatic Programmers, 2018.

Shared Kernel



- Parts of the infrastructure are shared
- Changes in shared parts must be coordinated between teams

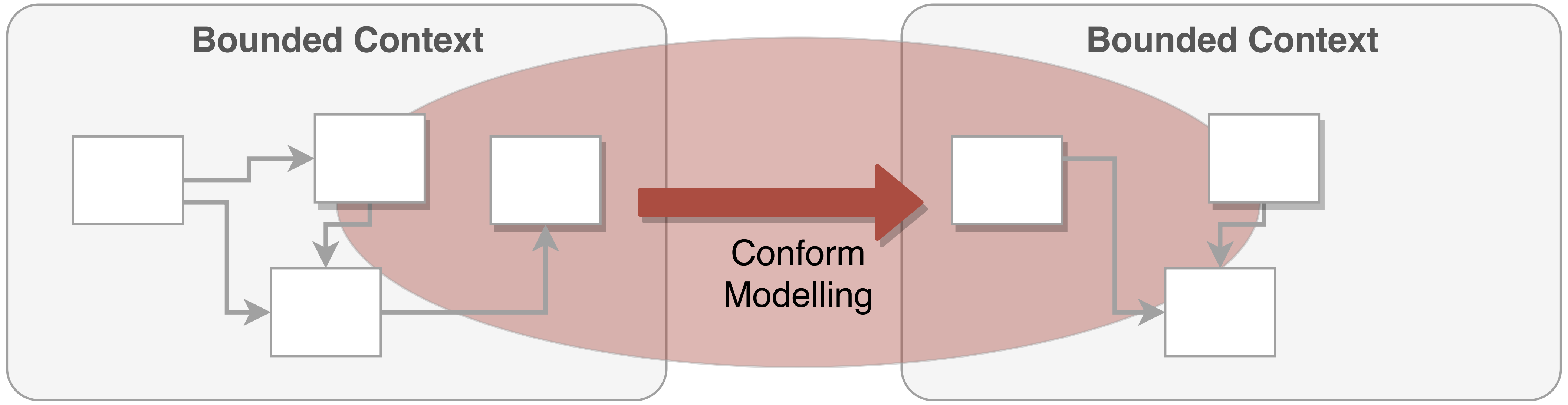
Customer/Supplier



- Supplier team works independently
- Offers their domain to customer teams
- Customers have the power of veto
- Acceptance testing can help limit unintended effects

@ctive group

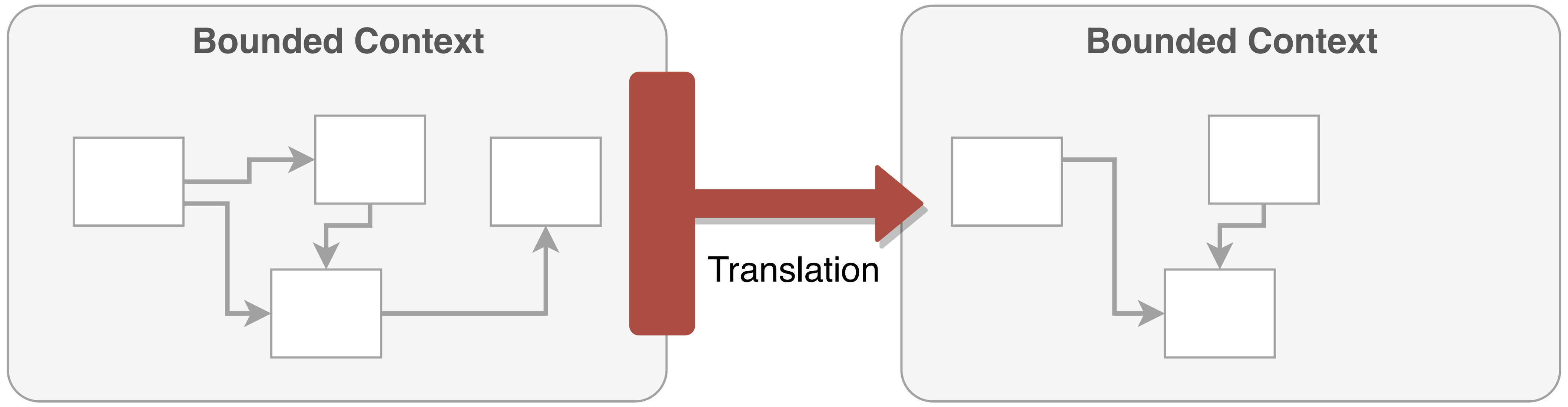
Conformist



- A context is based on another
- Decisions are transferred

@ctive group

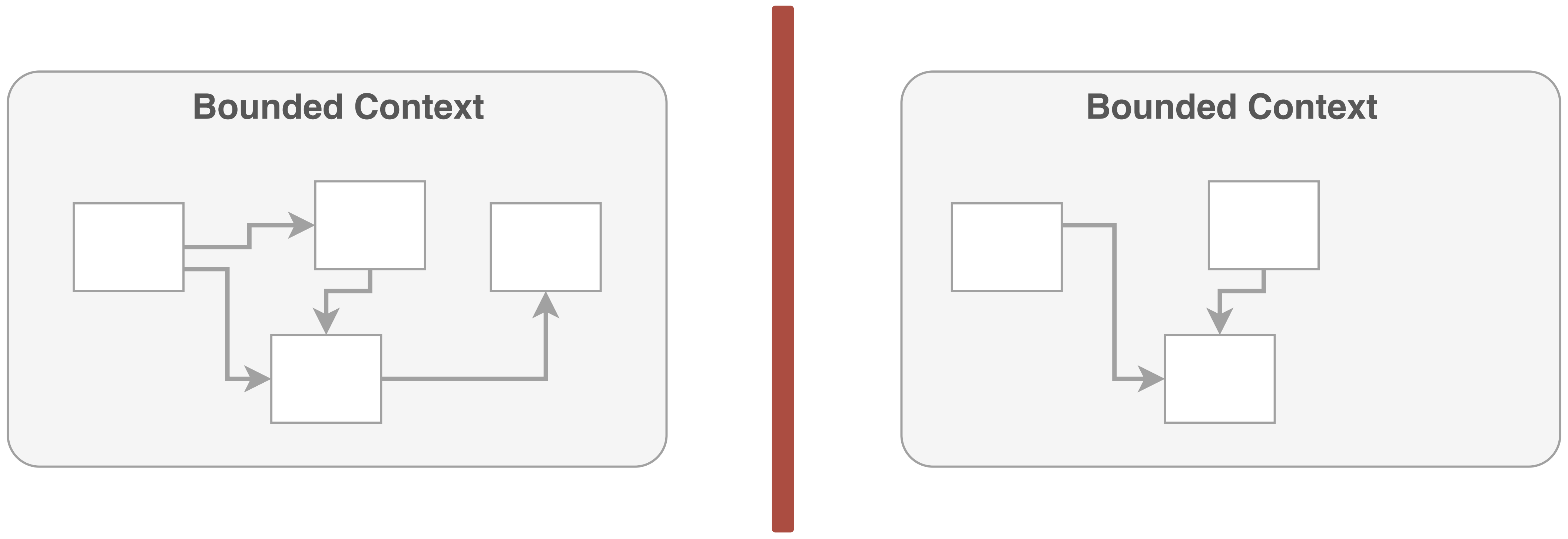
Anti-Corruption Layer



- well-defined translation between contexts
- very helpful when migrating legacy systems

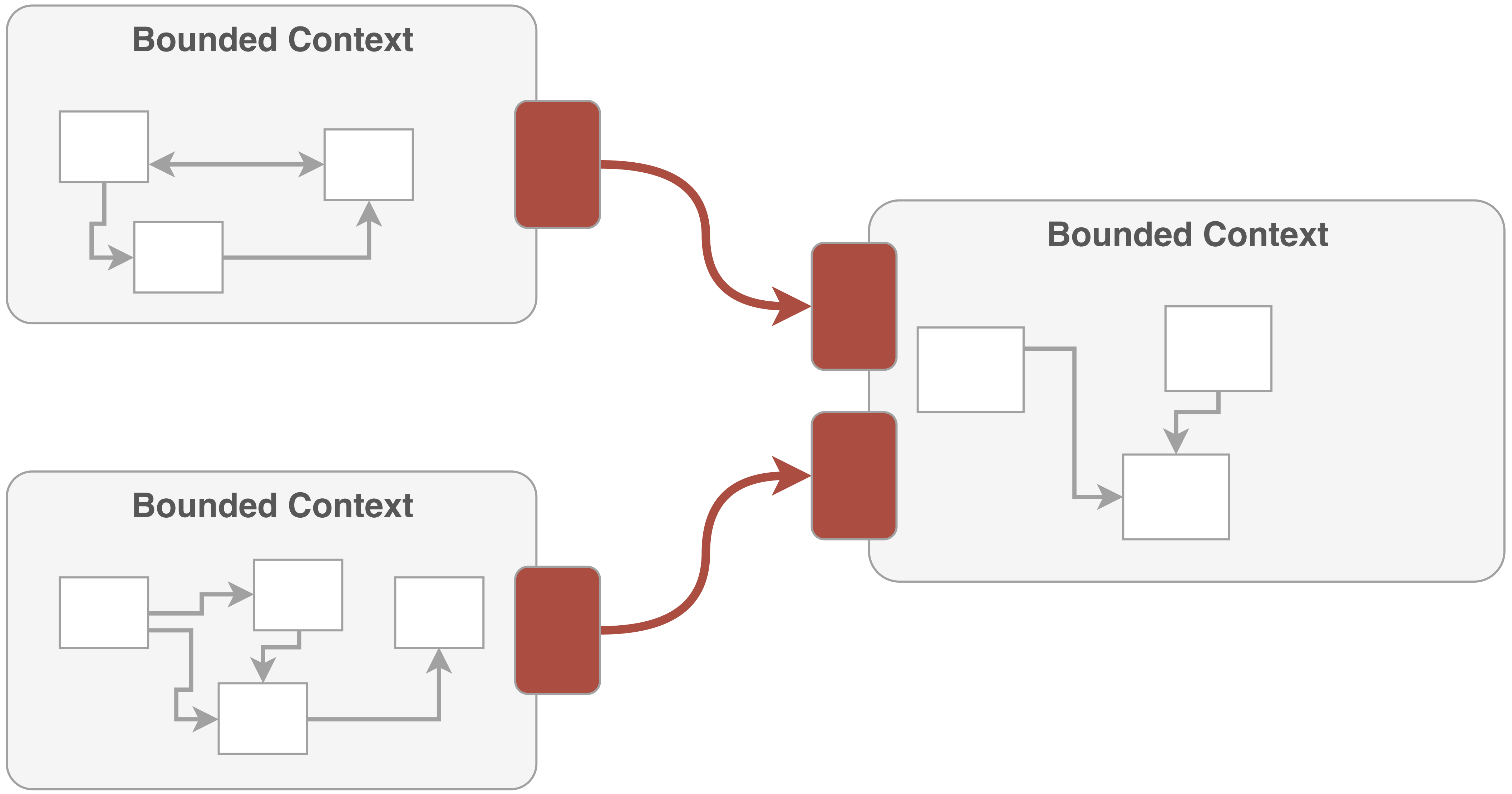
@ctive group

Separate Ways



- Entirely separated contexts
- No dependencies, neither on technological level nor on domain level

Open Host Service



- A bounded context defines interfaces for communication
- Allows definitions of many different interfaces for diverse client contexts

@ctive group

Strategic Design Patterns, which to choose?

Pick strategy based on quality goals, e.g.

- Support for transactions needed?
- Share code as much as possible?
- Robust, independent runtime?
- ...

From Modularization to Services: Anatomy of a Microservice

A microservice is a standalone service that offers interfaces other services can interact with:

- has its own architecture
- is owned & developed by exactly one team
- can contain a GUI

Wolff, Eberhard. *Microservices: Grundlagen flexibler Softwarearchitekturen*. dpunkt. verlag, 2018.

@ctive group

Self Contained Systems (SCS)

1. autonomous web application
2. owned by one team
3. communication asynchronous whenever possible
4. optional service API
5. includes data and logic
6. own UI
7. shares no business code with other SCSes
8. shared infrastructure is minimized

Web site: Self-Contained Systems

Exercise: To share or not to share between microservices & SCS?

What should be shared between microservices/SCS? Discuss e.g.:

- data storage
- data schemas
- code
- infrastructure
- developer

Teams and Ownership

- Domain Storytelling is used to find natural splits along domain boundaries
- A company structure may differ, e.g. it comes from a non-digital era
- Teams have ownership of one part of the domain
- Rather split along companies structure (Conway's Law)
- Team should not have to handle the domain of several departments. This results in an communication overhead and a discrepancy of domain boundaries

What about Erlbank?

We already know from domain story telling:

1. A customer may open an account
2. Send money to e.g. a friend
3. Finally prints a bank statement to get an overview of the transfers.

How does this fit the potential organizational structure?

We may have the following departments

- Customer management
- Banking backend, including transfers, fraud detection, ...
- ...

Maybe customer management takes care of offering bank statements for customers.

How convert a monolith into separate services?

- rewrite
- split
- duplicate

Split

Divide modules into separate services.

Note: A remote-procedure call is different from a local call. See section on distributed systems.

Duplicate

Give copy to each separate team, with a mission to only preserve and develop part of the functionality.

Give license to delete unneeded parts.

Appropriate when the different services already communicate via external means such as database, but code bases themselves are entangled.

How to split Erlbank? Customer Journey

For *Erlbank* we could split into:

- Accounts Application
- Transfers Application
- Bank Statements Application

A split along these boundaries would likely not interfere with the organizational boundaries.

@ctive group

Context Mapping

"Identify each model in play on the project and define its bounded context. [...] Name each bounded context, and make the names part of the ubiquitous language.

Describe the points of contact between the models, outlining explicit translation for any communication, highlighting any sharing, isolation mechanisms, and levels of influence."

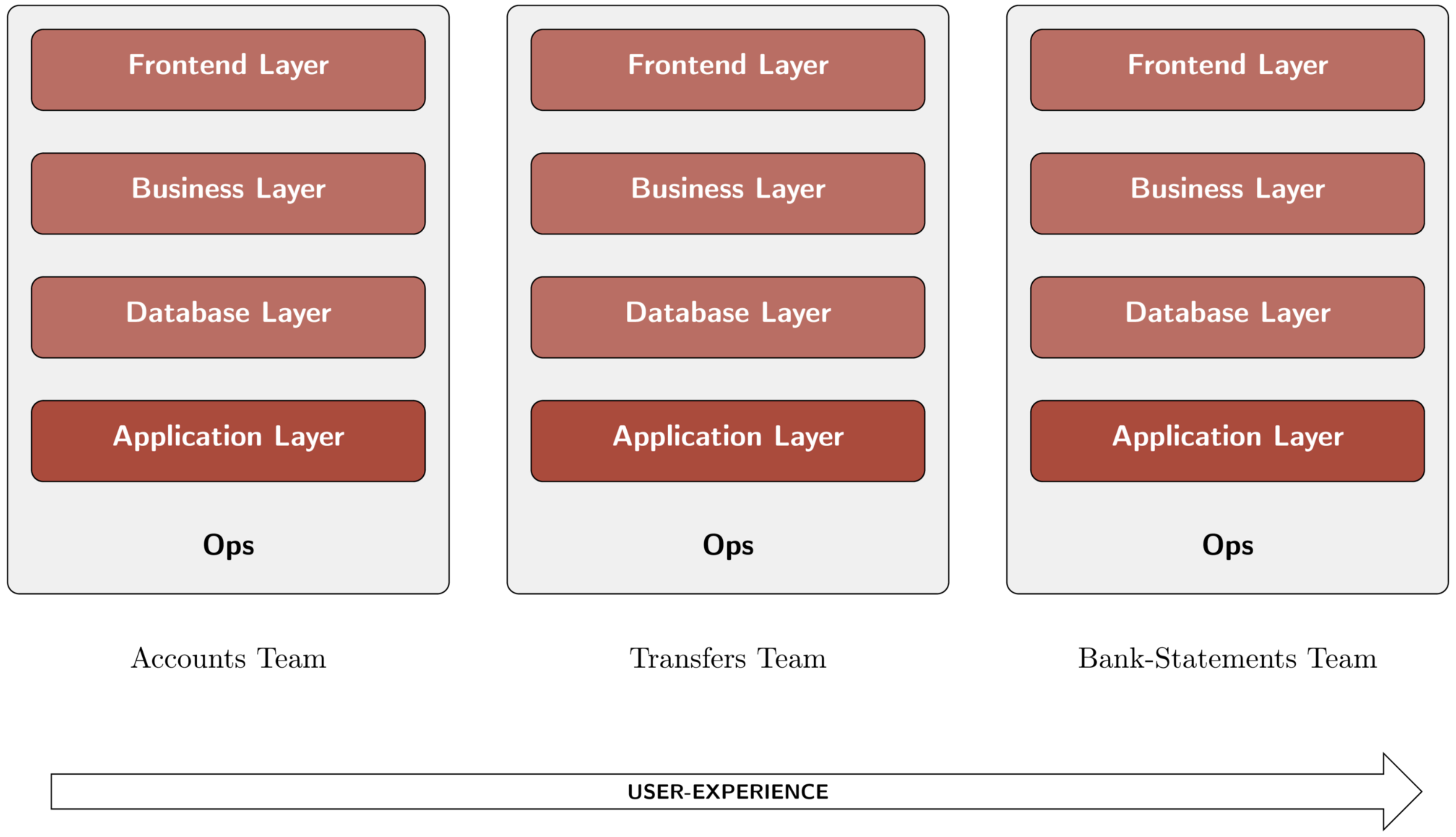
E. Evans: Domain-Driven Reference, 2015

Code Session - Solve the problems! Take 2

Participants now have the prerequisites to design a better architecture.

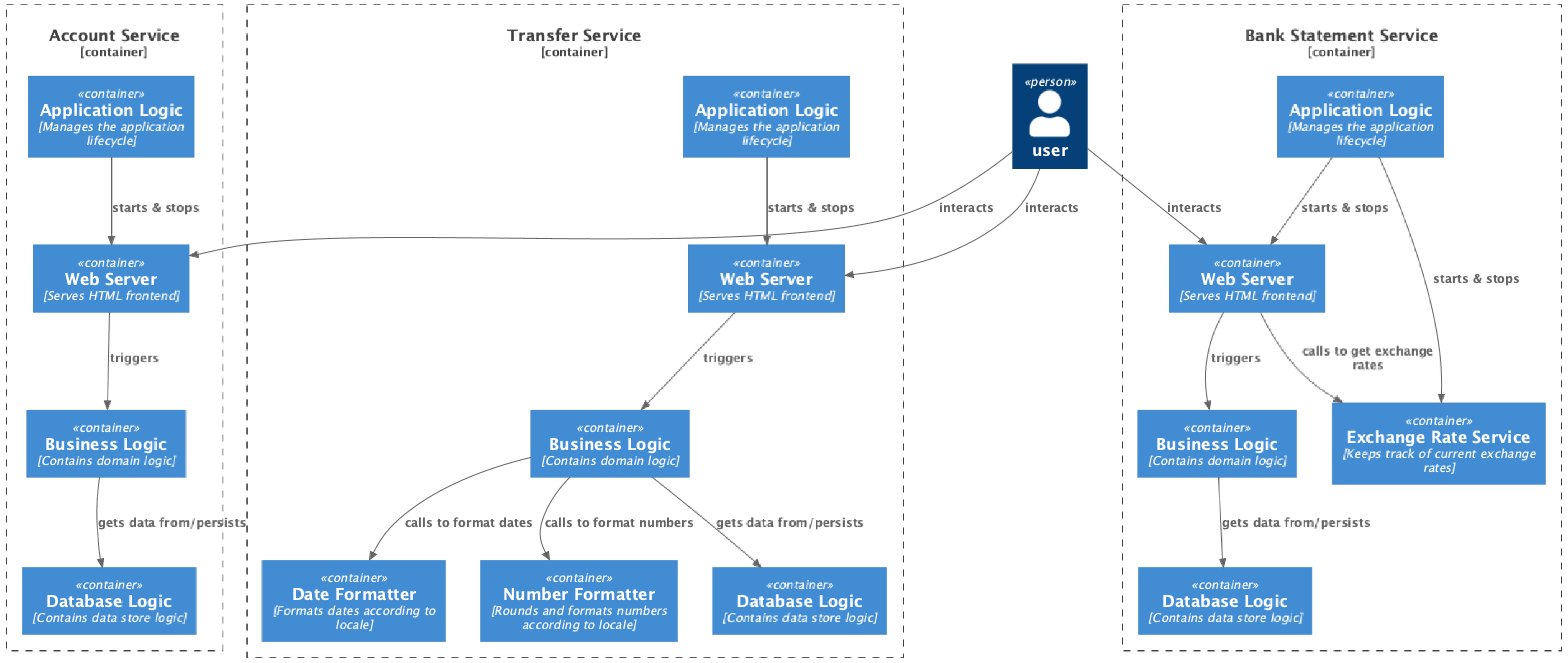
1. They will reevaluate, take a look at their goals and create a concept to split the application
2. Participants will form 3 teams
3. Each team will create a microservice/SCS for their domain

Erlbank Architecture



@ctive group

Component Diagram



@ctive group

Exercise: Diffculties when splitting up into services?

Participants discuss what the hardest part in the refactoring process was.

Additionally:

- What happens when we do wrong cuts? How can boundaries be re-adjusted?

Discussion and Retrospective: What we achieved

Independent development of source code

It reduces the amount of necessary communication and leads to more flexibility.

@ctive group

Discussion and Retrospective: What we achieved

Independent compilation

It requires less communication, leads to more flexible processes and less complex build-setups.

Discussion and Retrospective: What we achieved

Independent deployment

We can get rid of versioning and complex coordination, paving the way for continuous deployment.

What is missing?

Integration

- Data
- Frontend
- What else?