Integrating Bounded Contexts

Not only does the bounded context pattern protect the consistency of a ubiquitous

language, it also enables modeling. You cannot build a model without specifying its

purpose—its boundary. The boundary divides the responsibility of languages. A language

in one bounded context can model the business domain to solve a particular

problem. Another bounded context **can represent the same business entities but**

**model them to solve a different problem.**

Moreover, models in different bounded contexts can **be evolved and implemented**

**independently**. That said, bounded contexts **themselves are not independent**. Just as a

system cannot be built out of independent components—**the components have to**

**interact with one another** to achieve the system’s overarching goals—so, too, do the

implementations in bounded contexts. Although they can evolve independently, they

have to integrate with one another. As a result, **there will always be touchpoints**

**between bounded contexts. These are called *contracts*.**

The need for contracts results from differences in bounded contexts’ models and languages.

Since each contract affects more than one party, they need to be defined and

coordinated.

Also, by definition, **two bounded contexts are using different ubiquitous**

**languages. Which language will be used for integration purposes?**

These integration concerns should be evaluated and addressed by the solution’s design.

We will learn about domain-driven design patterns **for defining relationships** and **integrations between bounded contexts**.

These patterns **are driven by the nature of collaboration between teams** working on bounded contexts. We will divide the patterns into three groups, each representing a type of team collaboration:

cooperation, customer–supplier, and separate ways.

# Cooperation

**Cooperation patterns relate to bounded contexts implemented by teams with well-established communication.**

**In the simplest case, these are bounded contexts implemented by a single team.**

**This also applies to teams with dependent goals, where one team’s success depends on the success of the other, and vice versa.**

**Again, the main criterion here is the quality of the teams’ communication and collaboration.**

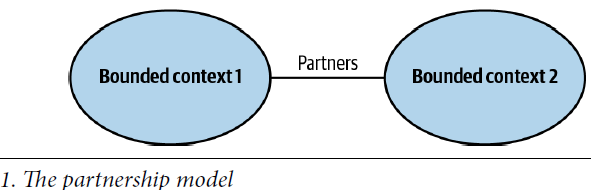
Let’s look at two **DDD patterns suitable for cooperating teams**: the partnership and

shared kernel patterns.

## Partnership

**In the partnership model, the integration between bounded contexts is coordinated in an ad hoc manner. One team can notify a second team about a change in the API,**

**and the second team will cooperate and adapt—no drama or conflicts.**



**The coordination of integration here is two-way**. **No one team dictates the language**

**that is used for defining the contracts.** The teams can work out the differences and

choose the most appropriate solution. Also, both sides cooperate in solving any integration

issues that might come up. Neither team is interested in blocking the other one.

**Well-established collaboration practices**,

**high levels of commitment,**

**and frequent synchronizations between teams** are required for successful integration in this manner.

From a technical perspective, continuous integration of the changes applied by

both teams is needed to further minimize the integration feedback loop.

**This pattern might not be a good fit for geographically distributed teams since it may**

**present synchronization and communication challenges.**

## Shared Kernel

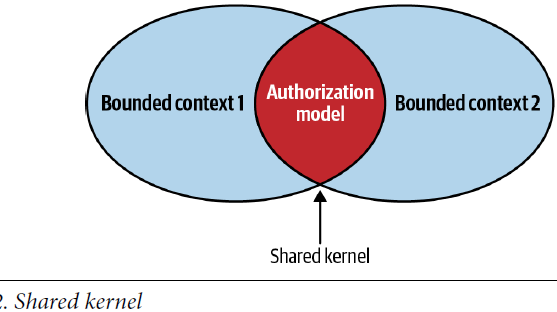
***This section needs to be revisited later-on***

Despite bounded contexts being model boundaries, **there still can be cases when the**

**same model of a subdomain, or a part of it, will be implemented in multiple bounded contexts.**

It’s crucial to stress that the shared model is designed according to the needs of all of the bounded contexts. Moreover, the shared model has to be consistent across all of the bounded contexts that are using it.

As an example, consider an enterprise system that uses a tailor-made model for managing users’ permissions. Each user can have their permissions granted directly or inherited from one of the organizational units they belong to. **Moreover, each bounded context can modify the authorization model, and the changes each bounded context applies have to affect all the other bounded contexts using the model (what does this mean exactly?)**



### Shared Scope

**The overlapping model couples the lifecycles of the participating bounded contexts.**

A change made to the shared model has an immediate effect on all the bounded contexts.

Hence, to minimize the cascading effects of changes**, the overlapping model should be limited**, exposing only that part of the model that has to be implemented by both bounded contexts. Ideally, the shared kernel will consist only of integration contracts and data structures that are intended to be passed across the bounded contexts’ boundaries.

### Implementation

**The shared kernel is implemented so that any modification to its source code is**

**immediately reflected in all the bounded contexts using it.**

If the organization uses the mono-repository approach, these can be the same source

files referenced by multiple bounded contexts.

If using a shared repository is not possible, the shared kernel can be extracted into a dedicated project and referenced in the bounded contexts as a linked library.

**Either way, each change to the shared kernel must trigger integration tests for all the affected bounded contexts.**

The continuous integration of changes is required because the shared kernel belongs

to multiple bounded contexts. Not propagating shared kernel changes to all related

bounded contexts leads to inconsistencies in a model: bounded contexts may rely on stale implementations of the shared kernel, leading to data corruption and/or runtime issues.

### When to Use Shared Kernel

The overarching applicability criterion for the shared kernel pattern is the cost of

duplication versus the cost of coordination. Since the pattern introduces a strong

dependency between the participating bounded contexts, it should be applied only

when the cost of duplication is higher than the cost of coordination—in other words,

only when integrating changes applied to the shared model by both bounded contexts

will require more effort than coordinating the changes in the shared codebase.

The difference between the integration and duplication costs depends on the volatility

of the model. The more frequently it changes, the higher the integration costs will

be. Therefore, the shared kernel will naturally be applied for the subdomains that

change the most: the core subdomains.

In a sense, the shared kernel pattern contradicts the principles of bounded contexts

introduced in the previous chapter. If the participating bounded contexts are not

implemented by the same team, introducing a shared kernel contradicts the principle

that a single team should own a bounded context. The overlapping model—the

shared kernel—is, in effect, being developed by multiple teams.

That’s the reason why the use of a shared kernel has to be justified. It’s a pragmatic

exception that should be considered carefully. A common use case for implementing

a shared kernel is when communication or collaboration issues prevent implementing

the partnership pattern—for example, because of geographical constraints or organizational

politics. Implementing a closely related functionality without proper coordination

will result in integration issues, desynchronized models, and arguments about

which model is better designed. Minimizing the shared kernel’s scope controls the

scope of cascading changes, and triggering integration tests for each change is a way

to enforce early detection of integration issues.

Another common use case for applying the shared kernel pattern, albeit a temporary

one, is the gradual modernization of a legacy system. In such a scenario, the shared

codebase can be a pragmatic intermediate solution for gradually decomposing the

system into bounded contexts.

Finally, a shared kernel can be a good fit for integrating bounded contexts owned and

implemented by the same team. In such a case, an ad hoc integration of the bounded

contexts—a partnership—can “wash out” the contexts’ boundaries over time. A

shared kernel can be used for explicitly defining the bounded contexts’ integration

contracts.