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.**

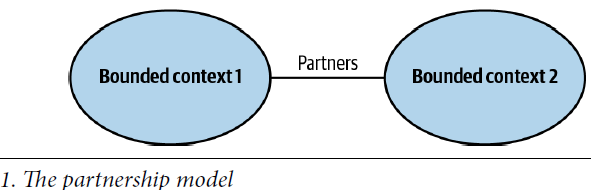
Let’s look at two **DDD patterns suitable for cooperating teams**: the partnership and

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

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

