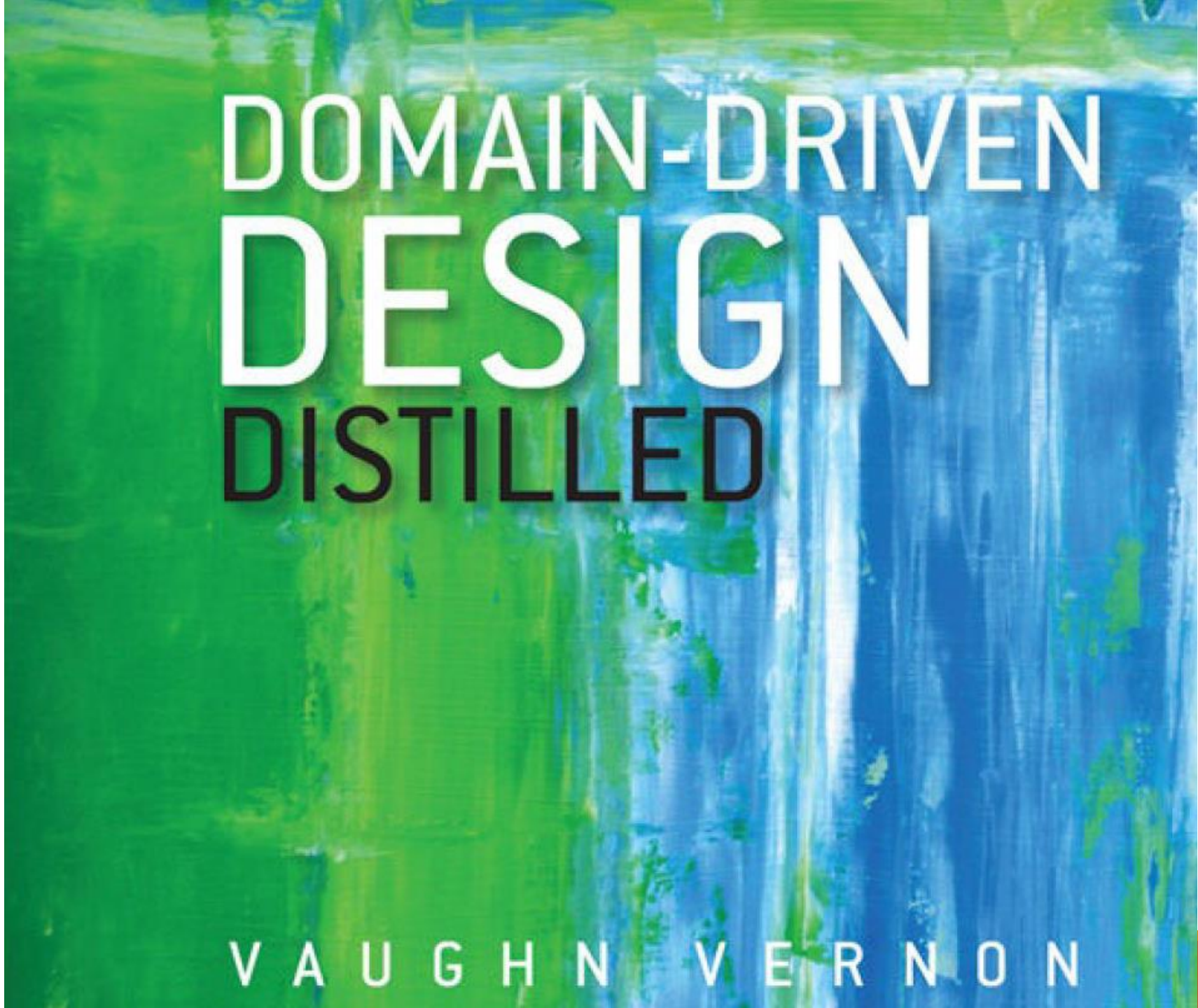
Implementing Domain Driven Design

VAUGHN VERNON

Domain-Driven Design Distilled, **Vaughn Vernon, 2016**



Chapter 1. DDD for Me

- DDD is a set of tools that assist you in designing and implementing software that delivers high value, both strategically and tactically.
- 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.

Good, Bad, and Effective Design

- Often people talk about good design and bad design.
- What kind of design do you do?
  - In many cases: “the task-board shuffle.” - move a sticky note from the “**To Do**” column to the “**In Progress**” column.
  - This often happens due to the pressure to deliver software releases on a relentless schedule
- *Effective design* meets the needs of the business organization to the extent that it can distinguish itself from its competition by means of software.

Most people make the mistake of thinking design is what it looks like. People think it’s this veneer—that the designers are handed this box and told, “Make it look good!” That’s not what we think design is. It’s not just what it looks like and feels like. Design is how it works.

—Steve Jobs

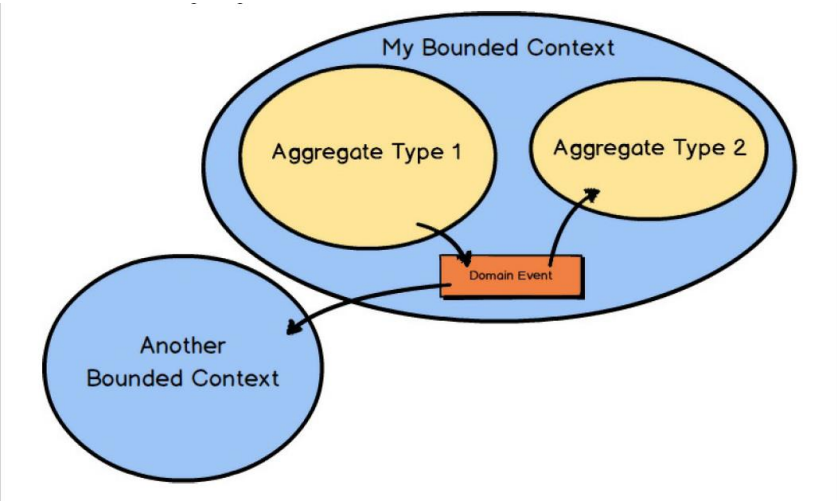
Strategic Design

- You really cannot apply tactical design in an effective way unless you begin with strategic design.
- It highlights what is strategically important to your business, how to divide up the work by importance, and how to best integrate as needed.
- *Developers + Domain Experts = DDD team.*
- *Bounded Contexts* – segregation of domain models using the strategic design pattern.
- *Ubiquitous Language* - the language developed by DDD team together through collaboration.
- *Subdomains* – deal with the unbounded complexity of legacy systems

Context Mapping – both team relationships and technical mechanisms that exist between two integrating *Bounded Contexts*

Tactical Design

- Tactical design is like using a thin brush to paint the fine details of your domain model.
- *Aggregate* - one of the more important tools is used to aggregate entities and value objects together into a right-sized cluster.
- *Domain Events* - something that happened in the domain that you want other parts of the local (or remote) *Bounded Context* to be aware of.

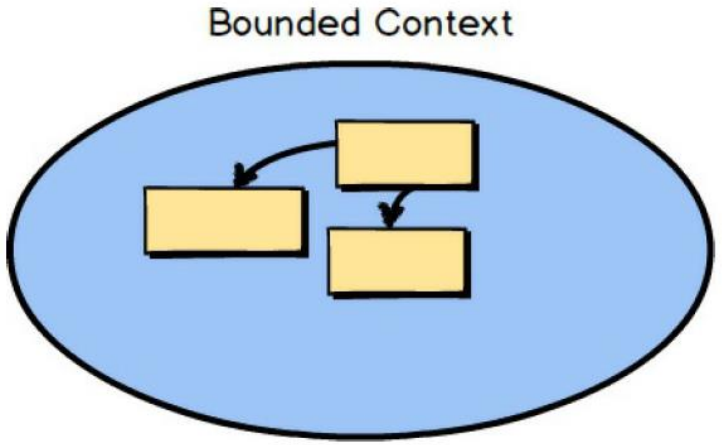


Strategic Design with Bounded Contexts and the Ubiquitous Language

CHAPTER 2

Bounded context

- *Bounded Context* is a semantic contextual boundary.
- Within the boundary each component of the software model has a specific meaning and does specific things.
- The components inside a *Bounded Context* are context specific and semantically motivated.
- *Bounded Context* is where a model is implemented, and you will have separate software artifacts for each *Bounded Context.*



Ubiquitous Language

- The software model inside the context boundary reflects a language that is developed by the team working in the *Bounded Context.*
- It is spoken by every member of the team that creates the software model that functions within that *Bounded Context.*
- ***The language is called the Ubiquitous Language because it is both spoken among the team members and implemented in the software model.***
- *Ubiquitous Language* must be rigorous—strict, exact, stringent, and tight.
- When someone on the team uses expressions from the Ubiquitous Language , everyone on the team understands what is meant with precision and constraints.
- **Active participation** from the domain experts is absolutely essential for domain driven design to succeed.

Ubiquitous Language

- You can document the ubiquitous language in various ways.
  - glossary of terms
- Business processes can be described graphically using e.g. swimlane diagrams and flow charts.
- UML - the relationship between things
- State diagrams - how state changes as different things move through different processes
- Maybe different "dialects" of the language for different subdomains.
- The domain model is not the same as a data model or a UML class diagram.

Creating the Ubiquitous Language

Developer: We want to monitor air traffic. Where do we start?

Expert: Let's start with the basics. All this traffic is made up of **planes**. Each plane takes off from a **departure** place, and lands at a **destination** place.

Developer: That's easy. When it flies, the plane can just choose any air path the pilots like? Is it up to them to decide which way they should go, as long as they reach destination?

Expert: Oh, no. The pilots receive a **route** they must follow. And they should stay on that route as close as possible.

Developer: I'm thinking of this **route** as a 3D path in the air. If we use a Cartesian system of coordinates, then the **route** is simply a series of 3D points.

Expert: I don't think so. We don't see **route** that way. The **route** is actually the projection on the ground of the expected air path of the airplane. The **route** goes through a series of points on the ground determined by their **latitude** and **longitude**.

Developer: OK, then let's call each of those points a **fix**, because it's a fixed point of Earth's surface. And we'll use then a series of 2D points to describe the path. And, by the way, the **departure** and **destination** are just **fixes**. We should not consider them as separate concepts. The **route** reaches destination as it reaches any other **fix**. The plane must follow the route, but does that mean that it can fly as high or as low as it likes?

Expert: No. The **altitude** that an airplane is to have at a certain moment is also established in the **flight plan.**

Developer: Flight plan? What is that?

Expert: Before leaving the airport, the pilots receive a detailed **flight plan** which includes all sorts of information about the **flight**: the **route**, cruise **altitude**, the cruise **speed**, the type of **airplane**, even information about the crew members.

Developer: Hmm, the **flight plan** seems pretty important to me. Let's include it into the model.

Developer: That's better. Now that I'm looking at it, I realize something. When we are monitoring air traffic, we are not actually interested in the planes themselves, if they are white or blue, or if they are Boeing or Airbus. We are interested in their **flight**. That's what we are actually tracking and measuring. I think we should change the model a bit in order to be more accurate.

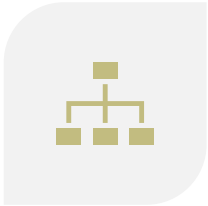
Bounded Contexts, Teams, and Source Code Repositories

THERE SHOULD BE *ONE TEAM* ASSIGNED TO WORK ON ONE *BOUNDED CONTEXT*.

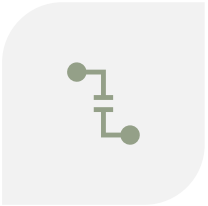
SEPARATE *SOURCE CODE REPOSITORY* FOR EACH *BOUNDED CONTEXT*.

ONE TEAM **CAN** WORK ON MULTIPLE BOUNDED CONTEXTS.

MULTIPLE TEAMS **SHOULD NOT** WORK ON A SINGLE BOUNDED CONTEXT.



CLEANLY SEPARATE THE SOURCE CODE AND DATABASE SCHEMA FOR EACH BOUNDED CONTEXT IN THE SAME WAY THAT YOU SEPARATE THE UBIQUITOUS LANGUAGE.

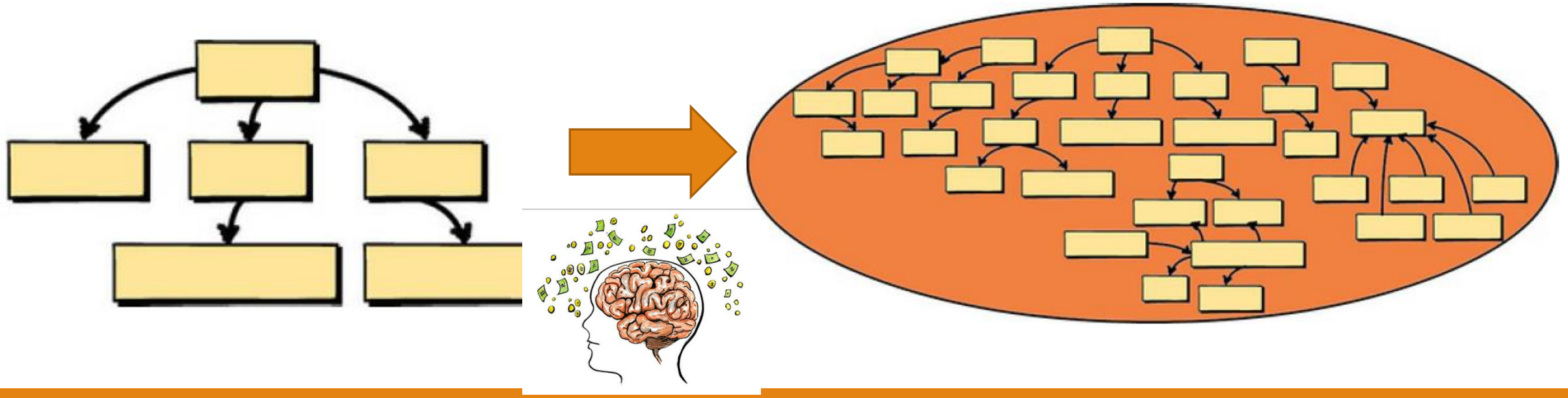


KEEP ACCEPTANCE TESTS AND UNIT TESTS TOGETHER WITH THE MAIN SOURCE CODE.

YOUR TEAM OWNS THE SOURCE CODE AND THE DATABASE AND DEFINES THE OFFICIAL INTERFACES THROUGH WHICH YOUR BOUNDED CONTEXT MUST BE USED.

Big Ball of Mud

This is where a system has multiple tangled models **without explicit boundaries**. It probably also **requires multiple teams** to work on it, which is **very problematic**. Furthermore, various **unrelated concepts are blown** out over many modules and **interconnected with conflicting elements**. If this **project has tests**, it probably takes a **very long time to run** them, and so the **tests** may be **bypassed** at especially **important times**.

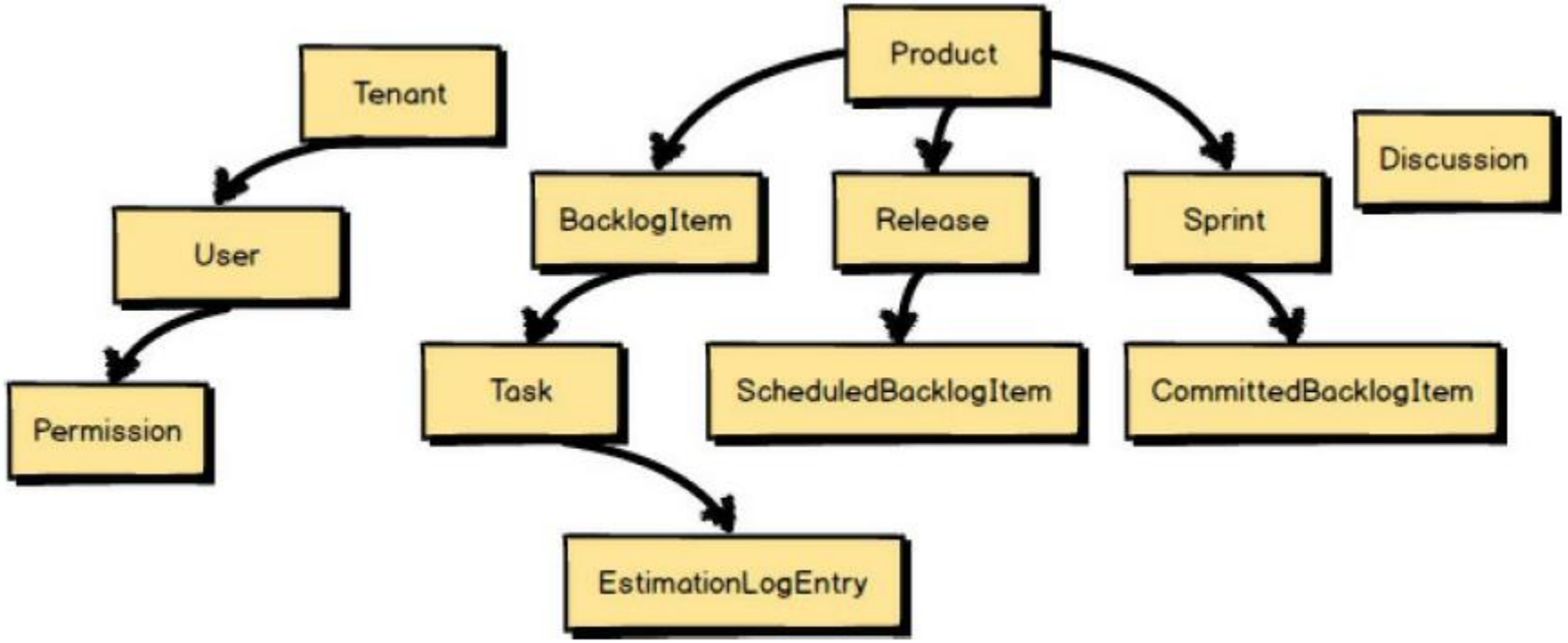


Case Study

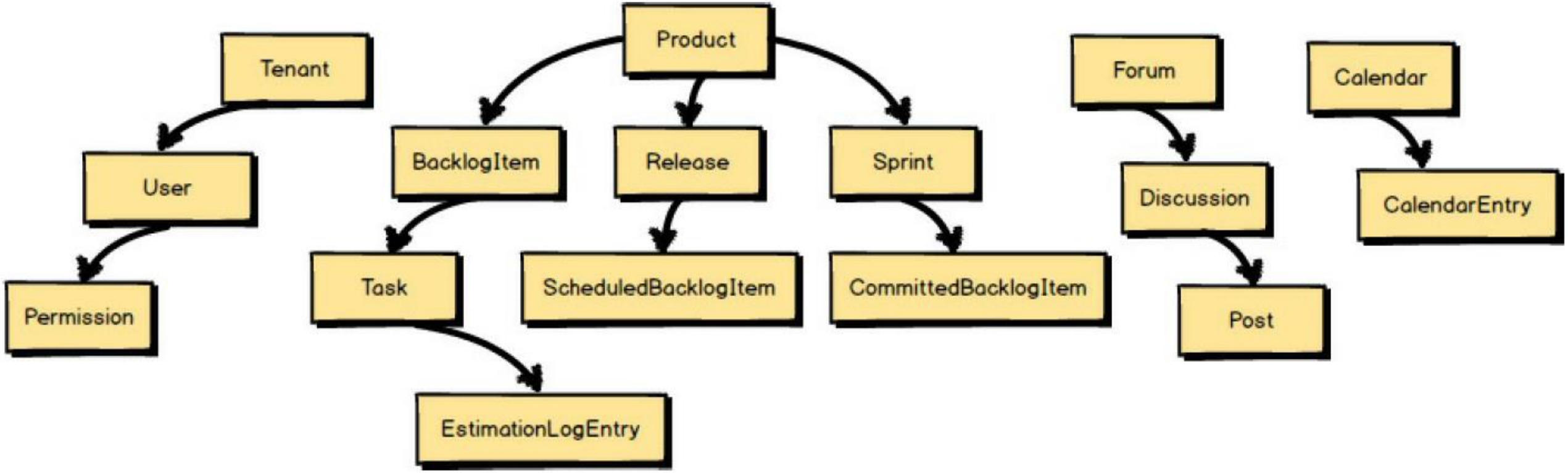
SCRUM-BASED AGILE PROJECT MANAGEMENT APPLICATION

Scrum-based agile project management application Iteration 1

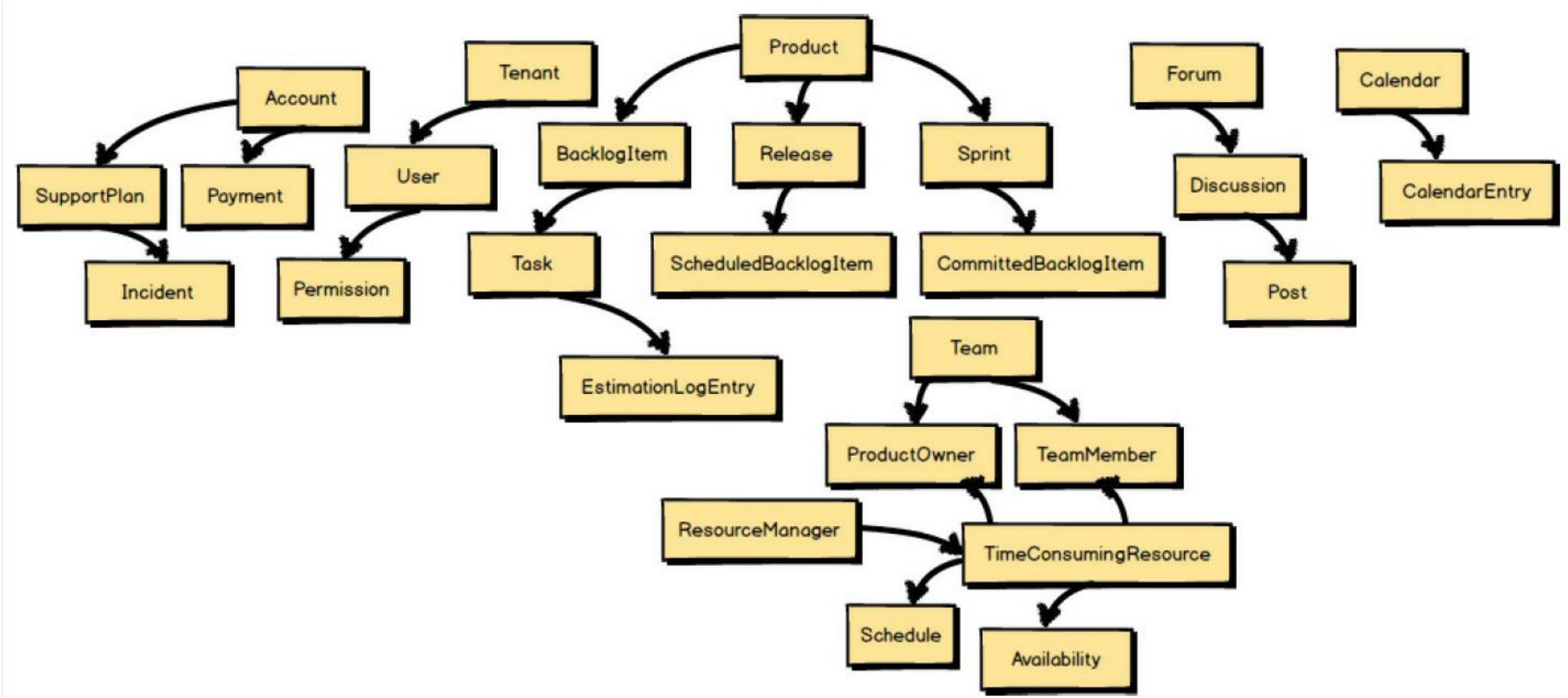
- central or core concept is Product



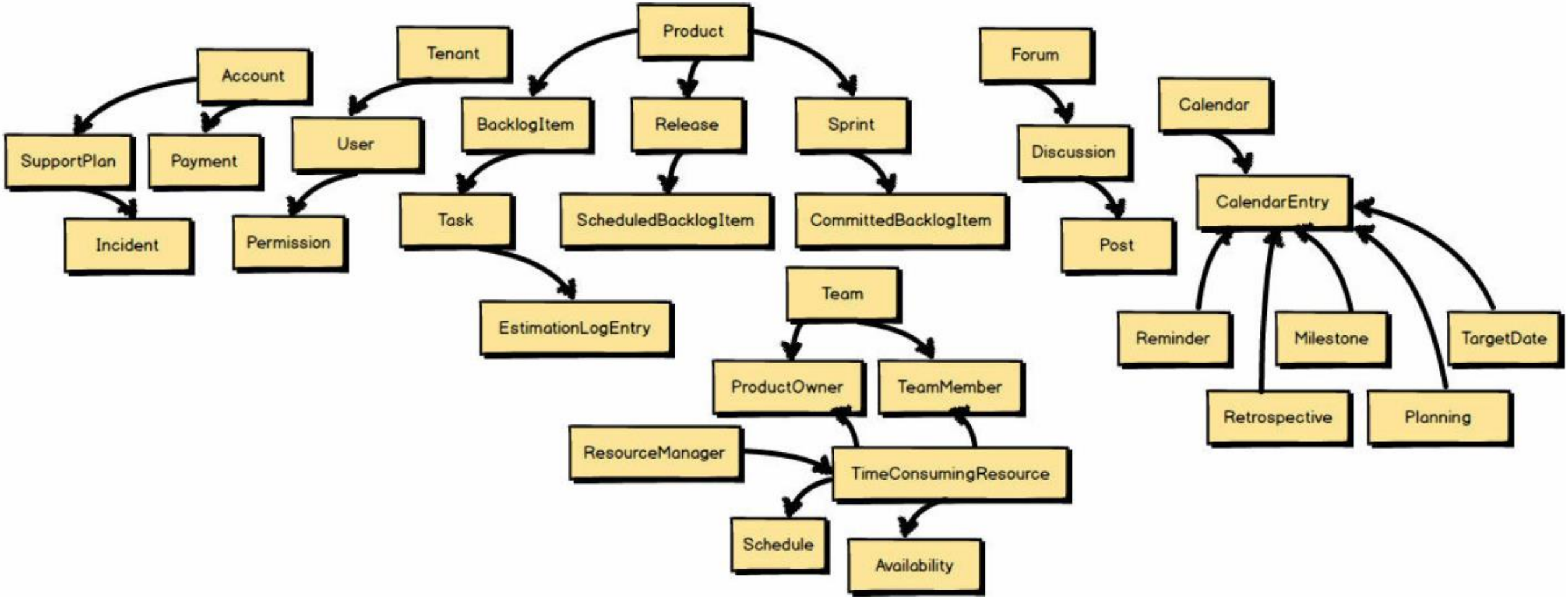
Scrum-based agile project management application Iteration 2



Scrum-based agile project management application Iteration 3

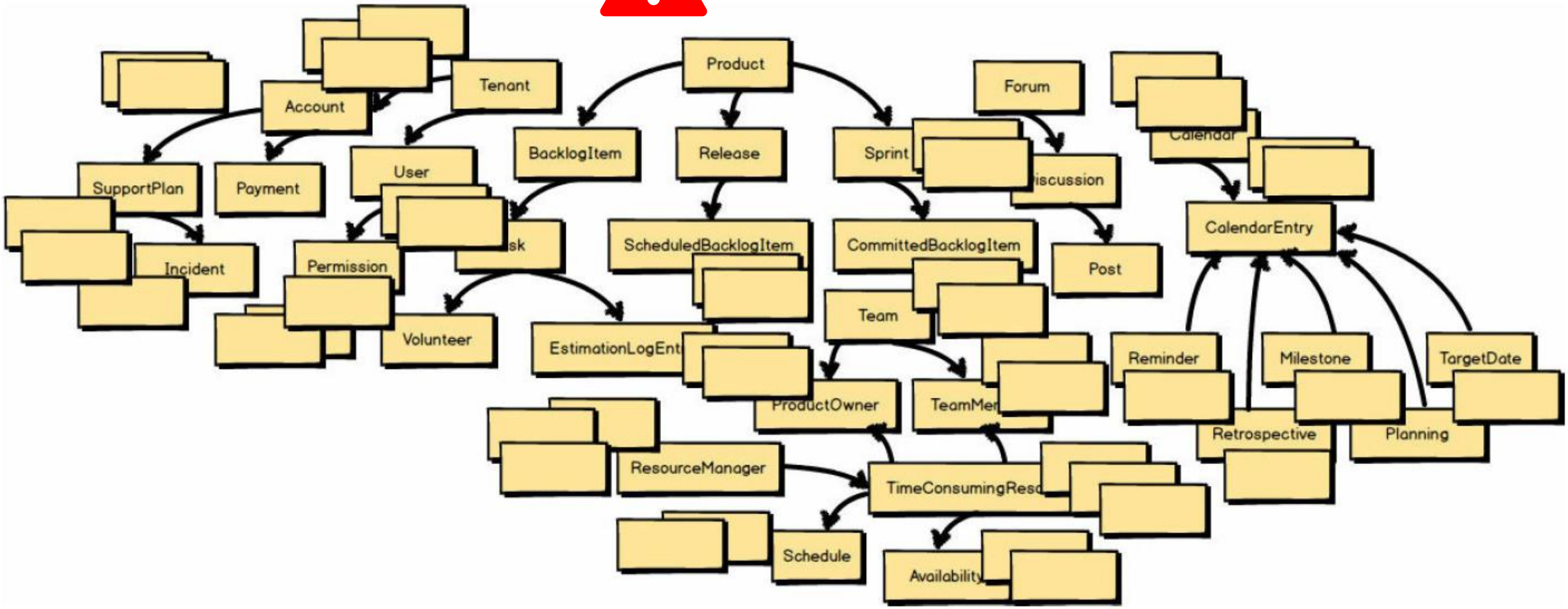


Scrum-based agile project management application Iteration 4



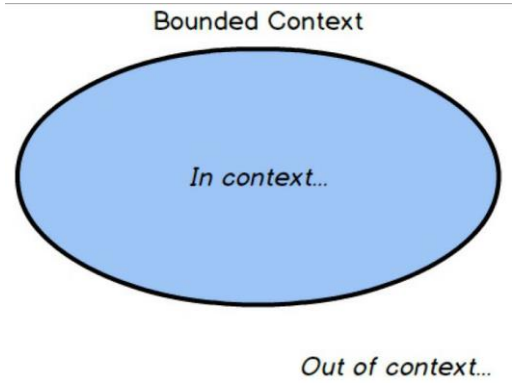
Scrum-based agile project management application Iteration 4

Fundamental Strategic Design Needed!!



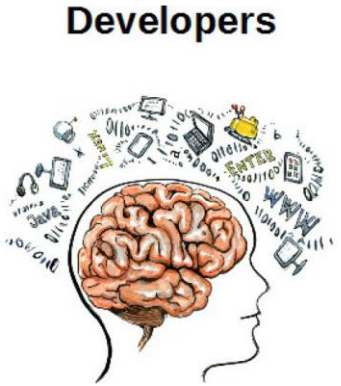
What tools are available with DDD to help us avoid such pitfalls?

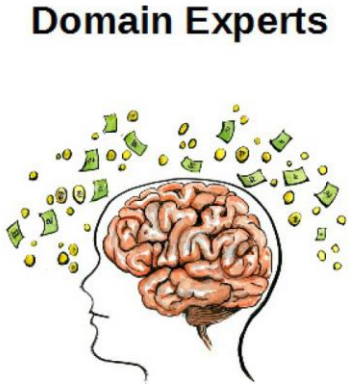
- *Bounded Context*
  - *The Bounded Context should hold closely all concepts that are core to the strategic initiative and push out all others.*



The concepts that survive this stringent application of core-only filtering are part of the *Ubiquitous Language* of the team that owns the *Bounded Context.* The boundary emphasizes the rigor inside.

So, what is core? This is where we have to bring together two vital groups of individuals into one cohesive, collaborative team: *Domain Experts* and software developers.

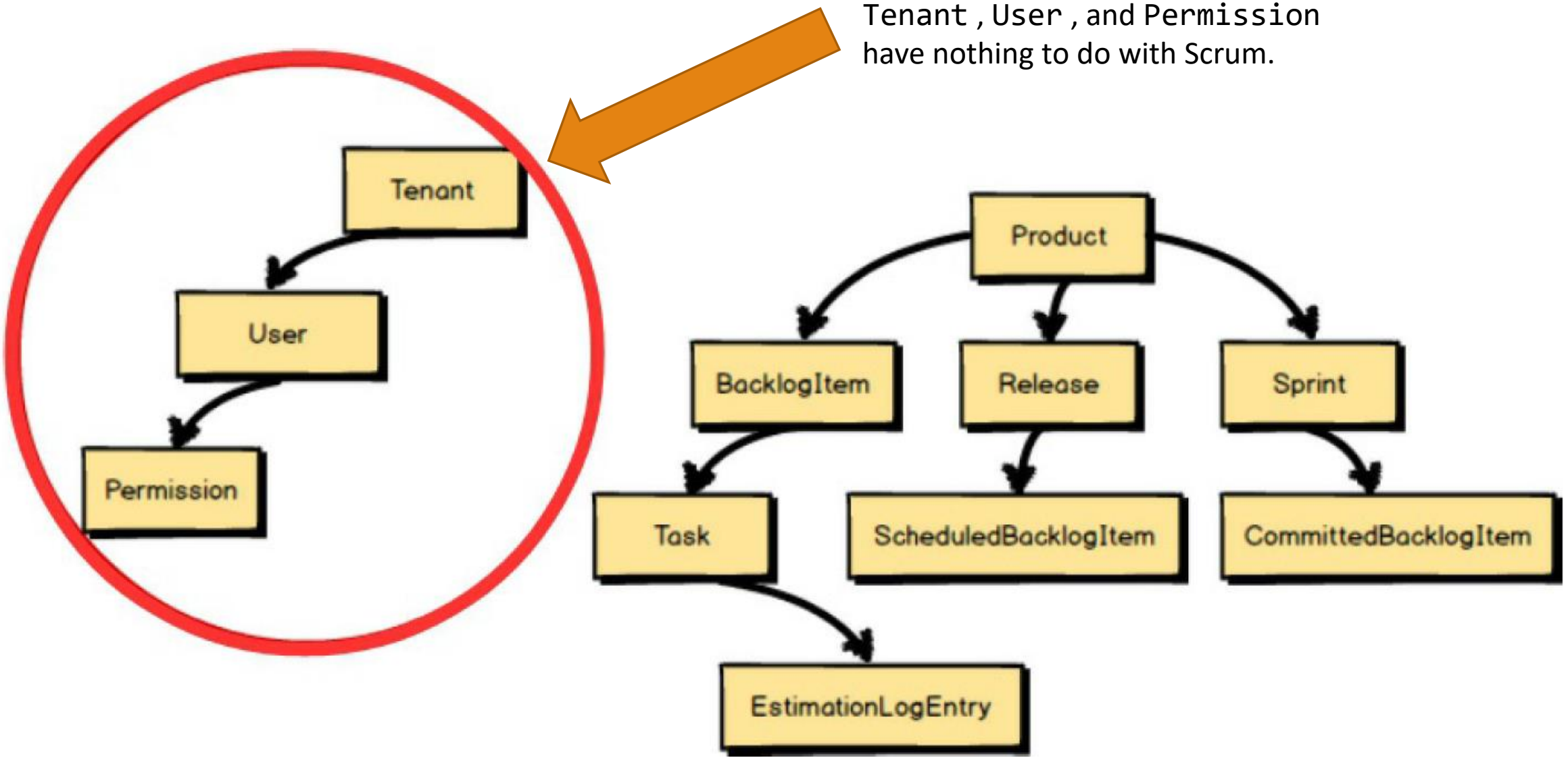


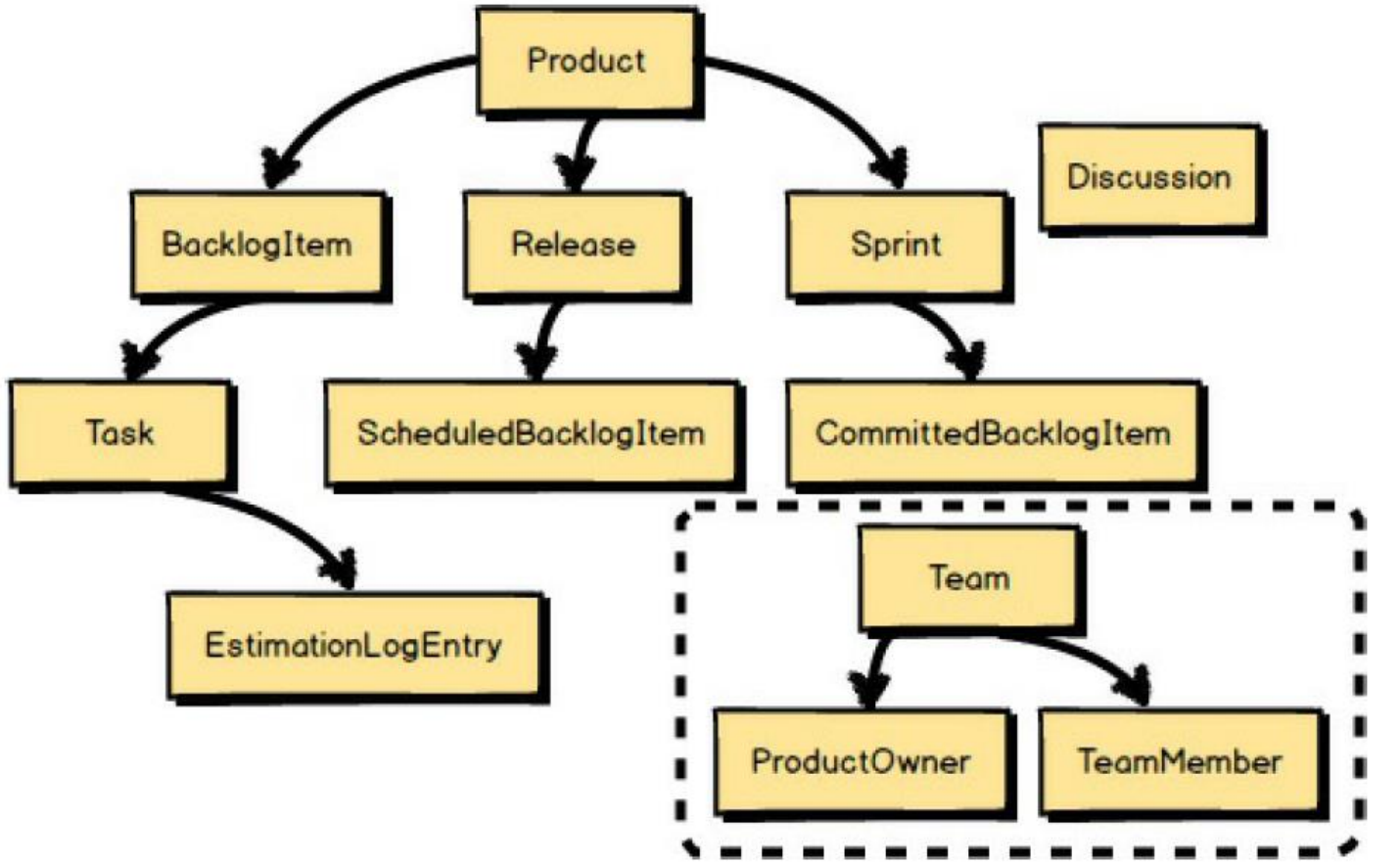


Focus on Business Complexity, Not Technical Complexity

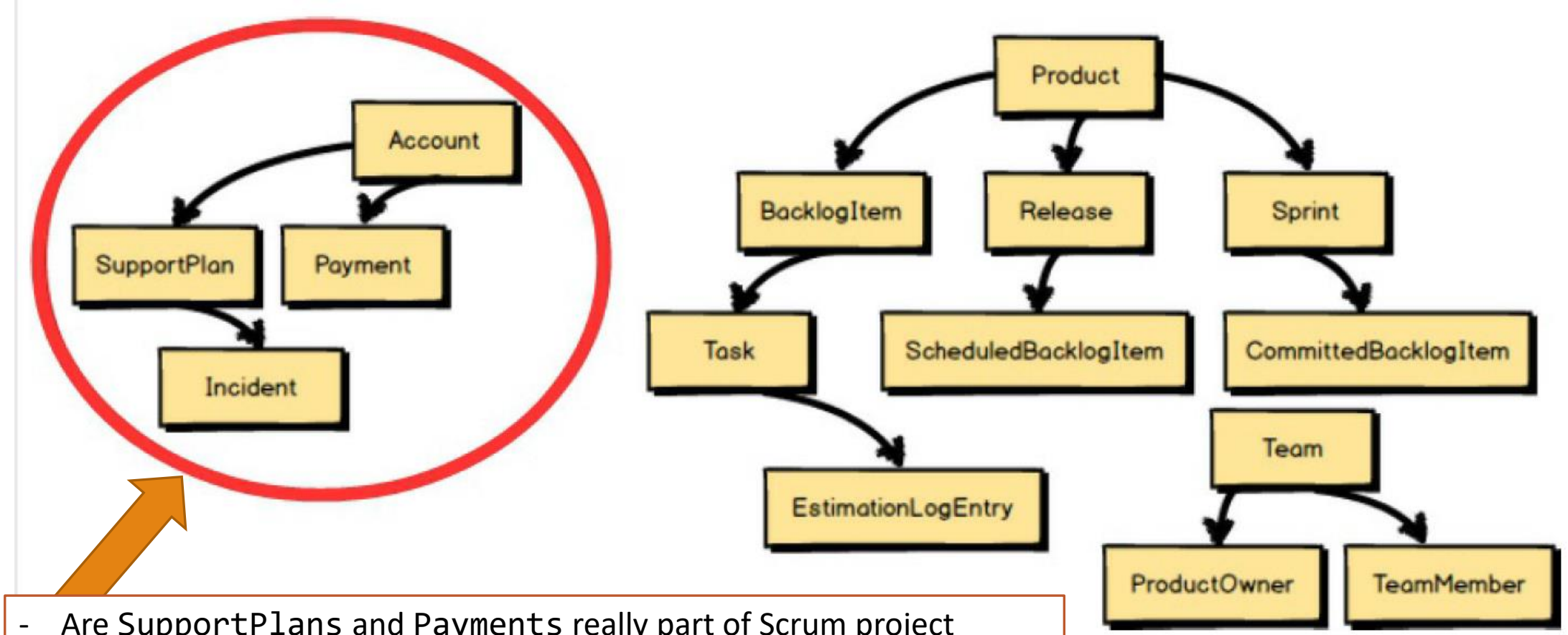
You are using DDD because the business model complexity is high. We never want to make the domain model more complex than it should be. Still, you are using DDD because the business model is more complex than the technical aspects of the project. That's why the developers have to dig into the business model with *Domain Experts* !

`Tenant` , `User` , and `Permission` have nothing to do with Scrum.

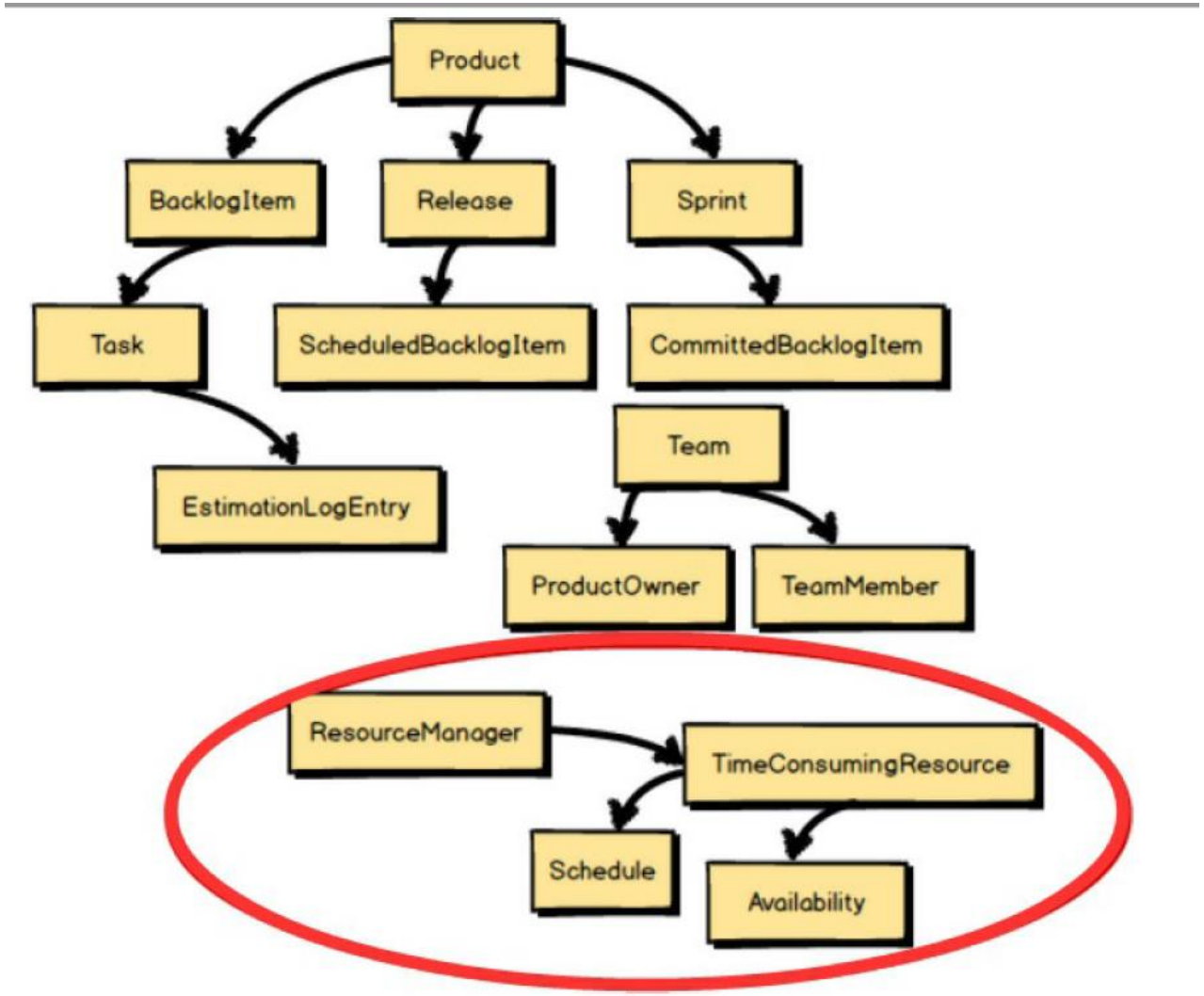




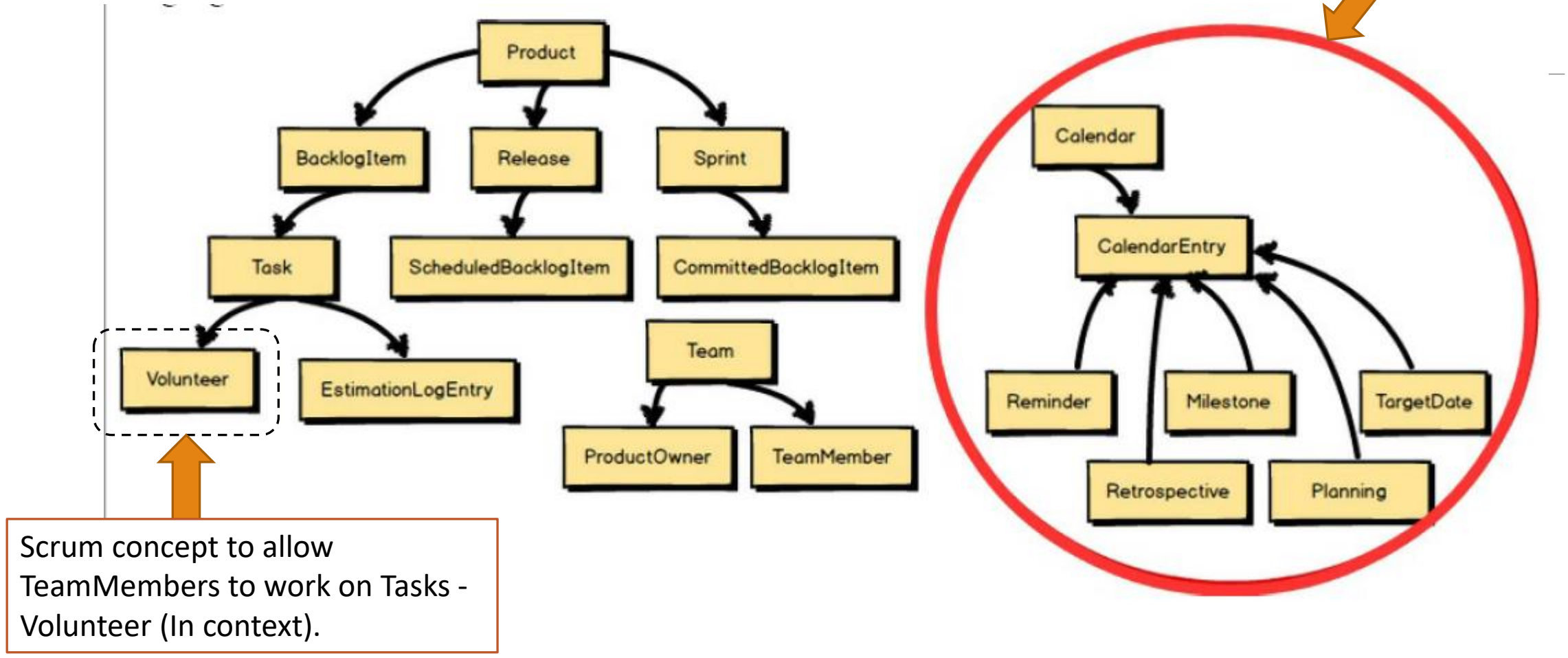
- **Tenant** , **User** , and **Permission** should be replaced by **Team** , **ProductOwner** , and **TeamMember**
- A **ProductOwner** and a **TeamMember** are actually **Users** in a **Tenancy**
- with **ProductOwner** and **TeamMember** we adhere to the *Ubiquitous Language* of **Scrum**.

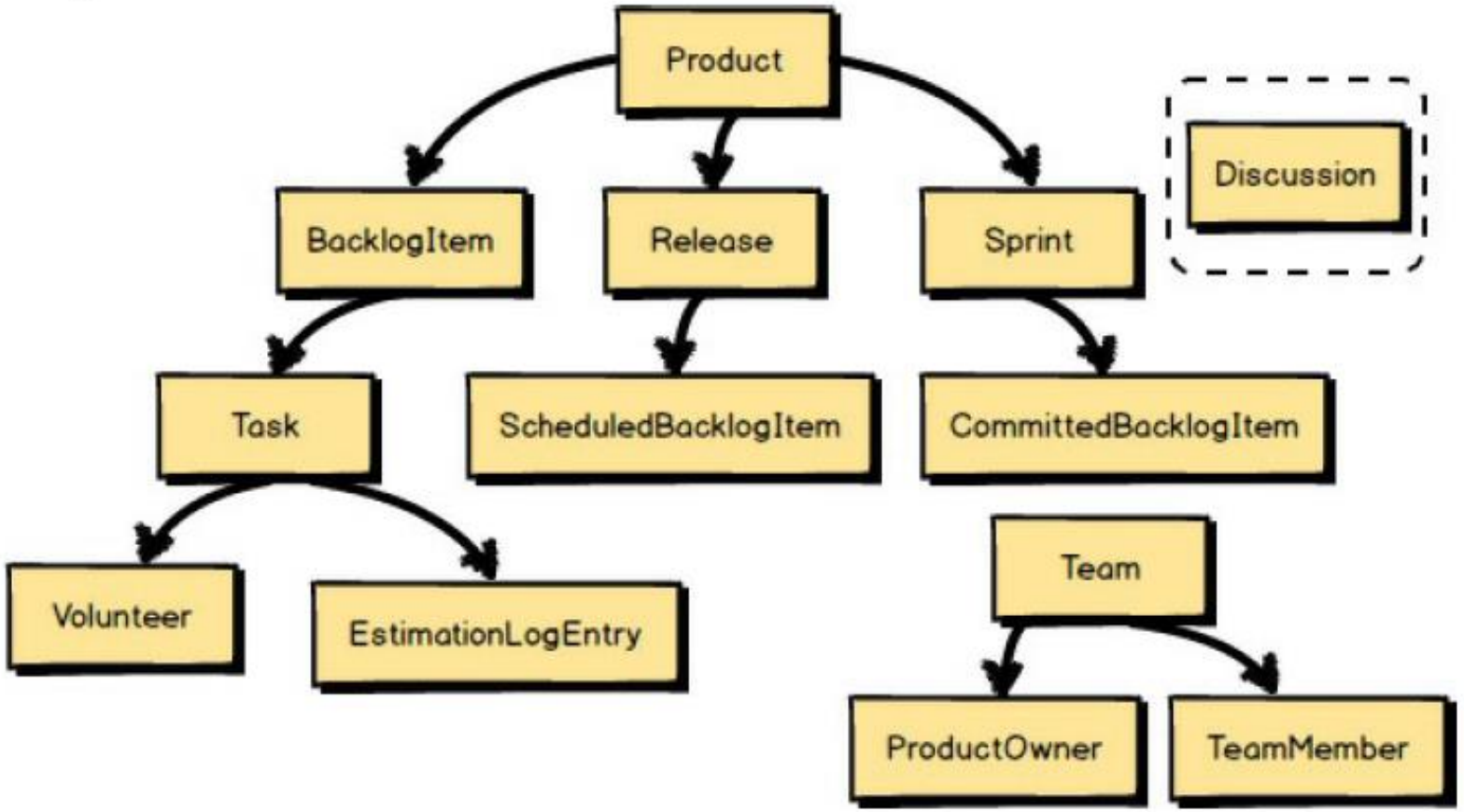


- Are `SupportPlans` and `Payments` really part of Scrum project management?
- No. True, both `SupportPlans` and `Payments` will be managed under a `Tenant` 's `Account` , but **these are not part** of our core Scrum language. They are **out of context** and are **removed** from this model.

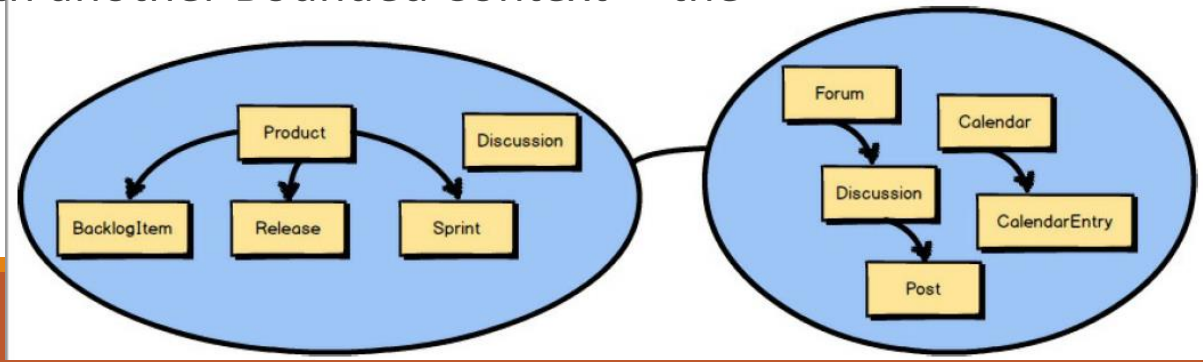


- What about introducing **Human Resource Utilization** concerns?
- It's probably useful to someone, but it's not going to be directly used by TeamMember Volunteers who will work on BacklogItemTasks . It's **out of context**.

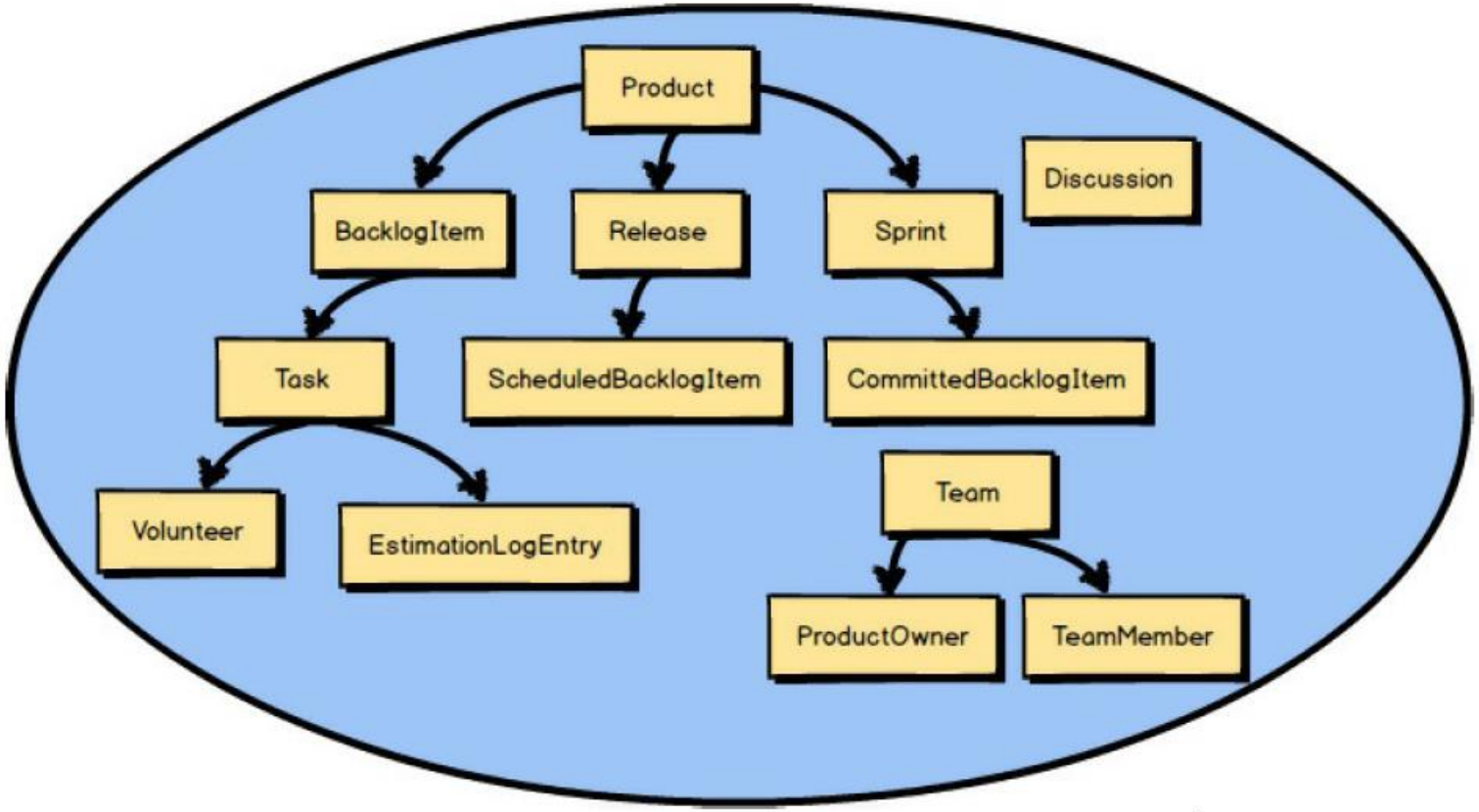

In context, but for now they are out of scope
Product
BacklogItem
Release
Sprint
Task
ScheduledBacklogItem
CommittedBacklogItem
Volunteer
EstimationLogEntry
Team
ProductOwner
TeamMember
Calendar
CalendarEntry
Reminder
Milestone
TargetDate
Retrospective
Planning
Scrum concept to allow TeamMembers to work on Tasks - Volunteer (In context).



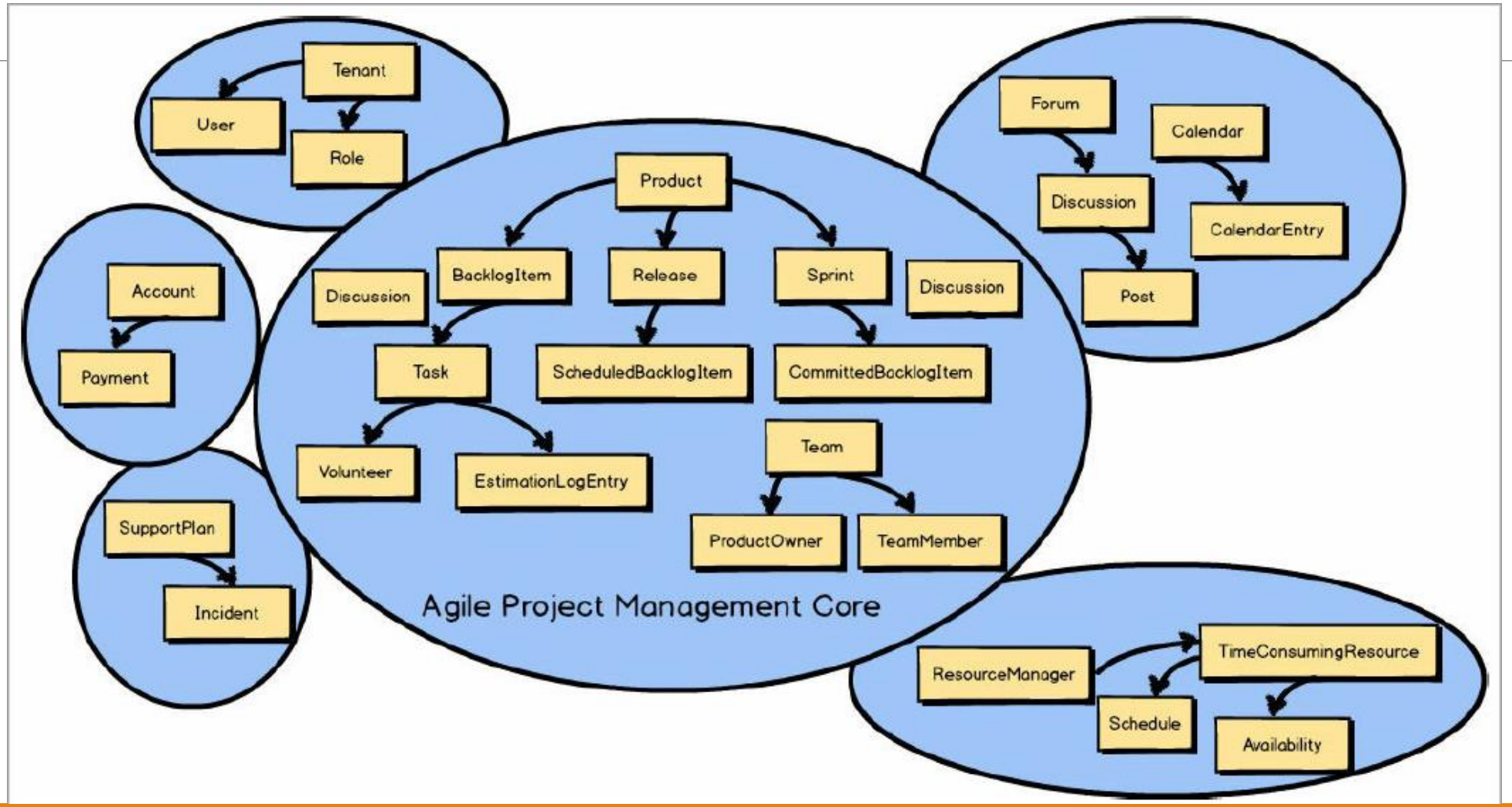
- Discussion is part of the team's *Ubiquitous Language* , and thus inside the *Bounded Context.*
- The Discussion will be supported by integrating with another Bounded Context —the Collaboration Context.

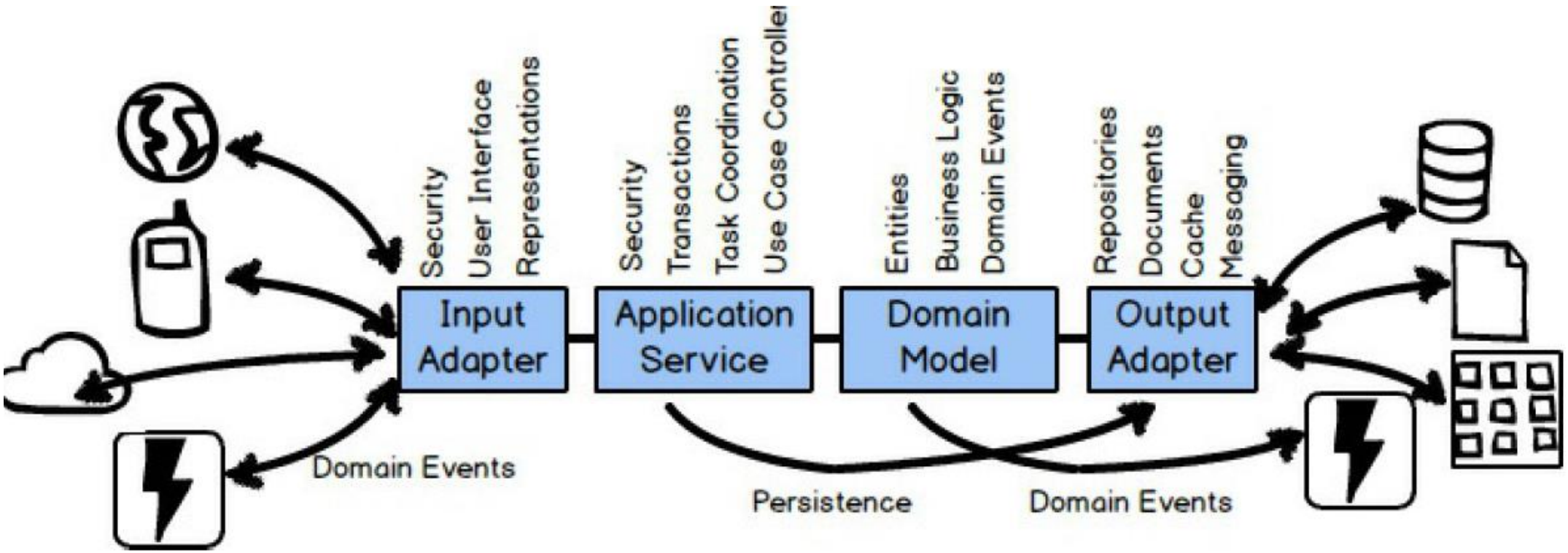


Core Domain


Product
BacklogItem
Release
Sprint
Discussion
Task
ScheduledBacklogItem
CommittedBacklogItem
Volunteer
EstimationLogEntry
Team
ProductOwner
TeamMember

What about all the other modeling concepts that have been removed from the Core Domain?





What's inside a Bounded Context ?

Technology-Free Domain Model

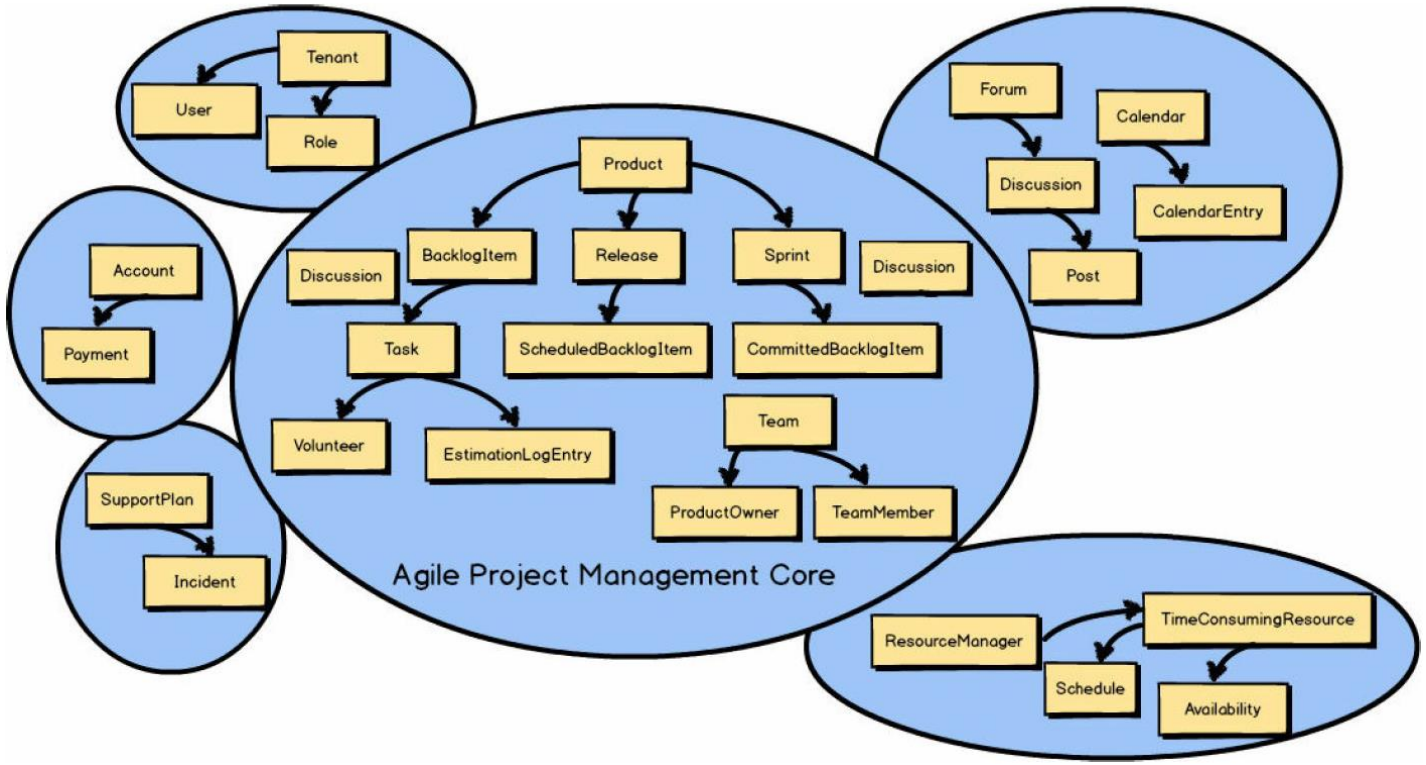
Although there will be technology scattered throughout your architecture, the domain model should be free of technology. For one thing, that's why transactions are managed by the application services and not by the domain model.

Strategic Design with Subdomains

CHAPTER 3

Typical DDD project

- There are always multiple Bounded Contexts in play
- One of the Bounded Contexts is the Core Domain
- Subdomains in other Bounded Contexts
  - *One Subdomain per Bounded Context*
  - Multiple Subdomains in one Bounded Context



6 Bounded Contexts and 6 Subdomains

What Is a Subdomain?

- *Subdomain* is a sub-part of your overall business domain.
- 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.

Types of Subdomains

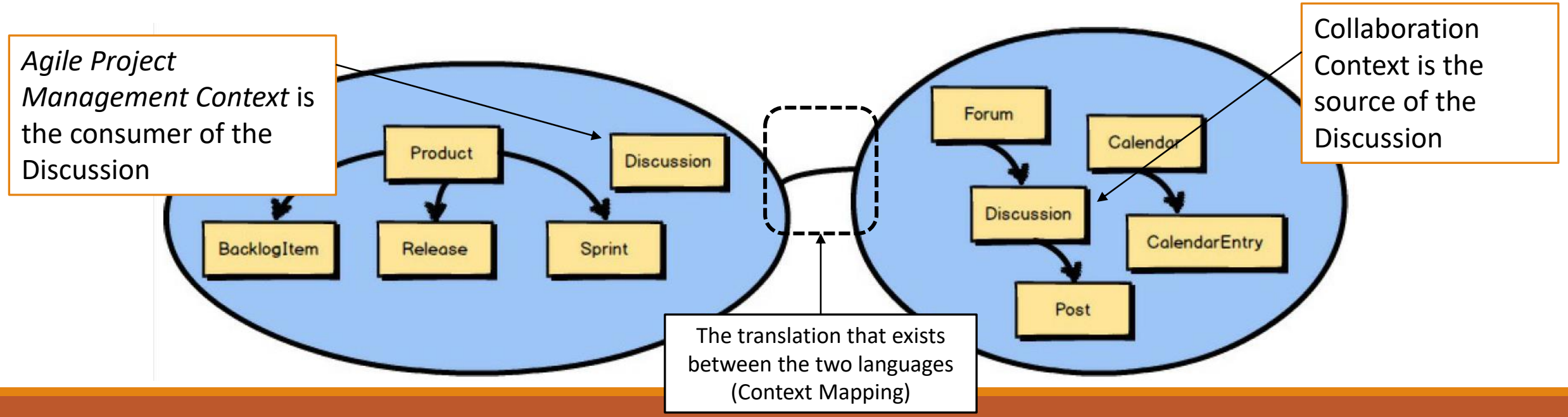
- **Core Domain**: This is where you are making a strategic investment in a single, well-defined domain model, committing significant resources for carefully crafting your Ubiquitous Language in an explicit Bounded Context.
- **Supporting Subdomain***:* This is a modeling situation that calls for custom development, because an off-the-shelf solution doesn't exist. However, you will still not make the kind of investment that you have made for your Core Domain. Your Core Domain cannot be successful without it.
- **Generic Subdomain**: This kind of solution may be available for purchase off the shelf but may also be outsourced or even developed in house by a team that doesn't have the kind of elite developers that you assign to your Core Domain or even a lesser Supporting Subdomain.

Strategic Design with Context Mapping

CHAPTER 4

Recap…

- In previous chapters you learned that in addition to the *Core Domain* , there are multiple *Bounded Contexts* associated with every DDD project. All concepts that didn't belong in the *Agile Project Management Context* —the *Core Domain* —were moved to one of several other *Bounded Contexts.*
- *You also learned that the Agile Project Management Core Domain would have to integrate with other Bounded Contexts (Context Mapping)*

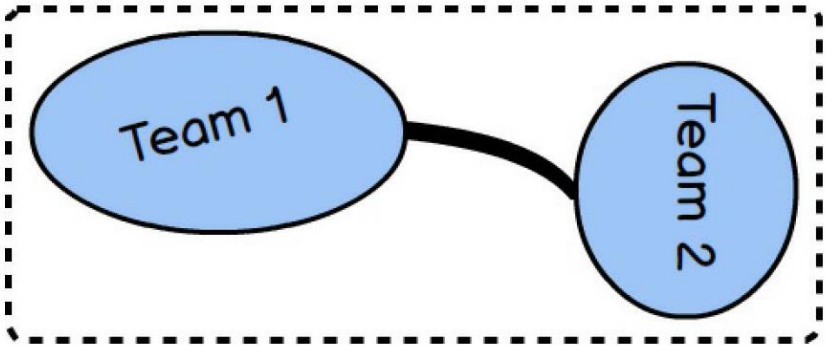


Kinds of Mappings

- **Partnership**
- **Shared Kernel**
- **Customer-Supplier**
- **Conformist**
- **Anticorruption Layer**
- **Open Host Service**
- **Published Language**
- **Separate Ways**

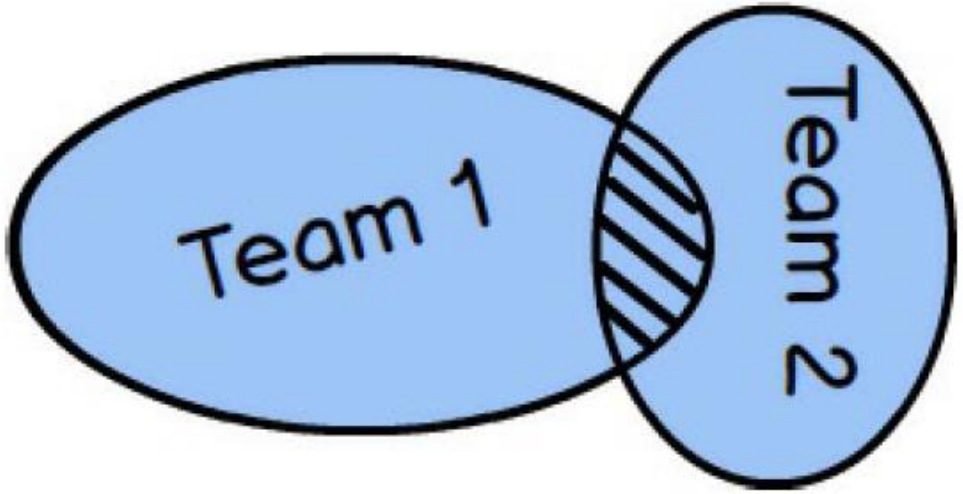
Partnership

- A *Partnership* relationship exists between two teams. Each team is responsible for one Bounded Context. They create a Partnership to align the two teams with a dependent set of goals.
- It is said that the two teams will succeed or fail together.
- Since they are so closely aligned, they will meet frequently to synchronize schedules and dependent work, and they will have to use continuous integration to keep their integrations in harmony.
- The synchronization is represented by the thick mapping line between the two teams.
- The Partnership should last only as long as it provides an advantage, and it should be remapped to a different relationship



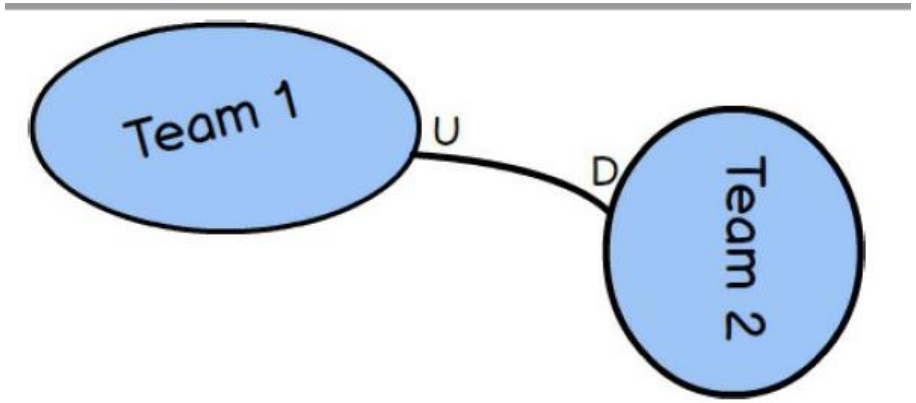
Shared Kernel

- A Shared Kernel by the intersection of the two Bounded Contexts , describes the relationship between two (or more) teams that share a small but common model.
- The teams must agree on what model elements they are to share.
- It’s possible that only one of the teams will maintain the code, build, and test for what is shared.
- Very difficult to conceive and maintain.
- Still, it is possible to be successful if all involved are committed to the idea that the kernel is better than going Separate Ways.
- If a lot of model code ends up in the shared kernel, it may be a sign that the contexts should, in fact, be merged into one big context.



Customer-Supplier

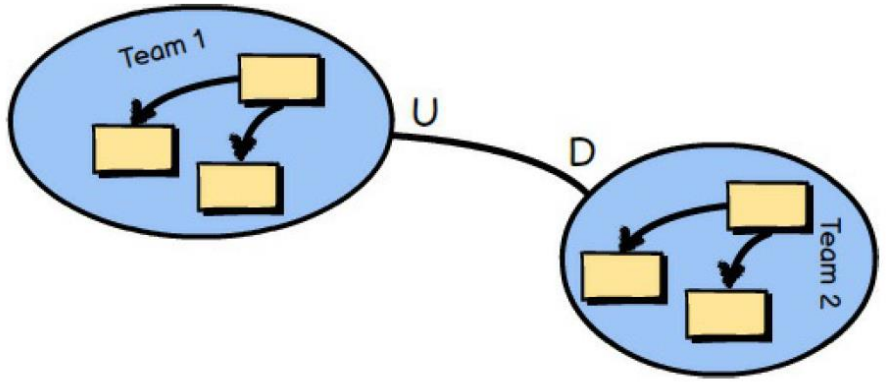
- A Customer-Supplier describes a relationship between two Bounded Contexts and respective teams, where the Supplier is upstream (the U in the diagram) and the Customer is downstream (the D in the diagram).
- A Customer-Supplier relationship is viable when both teams are interested in the relationship.
- The customer is very dependent on the supplier, while the supplier is not.
- The Supplier holds sway in this relationship because it must provide what the Customer needs.
- It's up to the Customer to plan with the Supplier to meet various expectations, but in the end the Supplier determines what the Customer will get and when.



Conformist

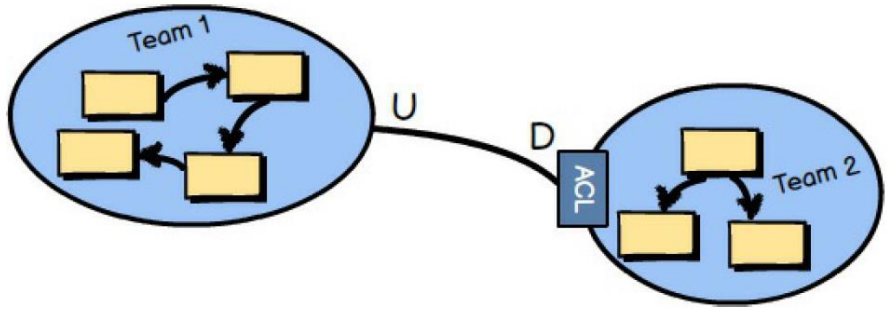
- A Conformist relationship exists when there are upstream and downstream teams, and the **upstream team is not motivated to support** the specific needs of the downstream team.
- Team conforms to the upstream model as is.

  Example: Consider the need to conform to the Amazon.com model when integrating as one of Amazon's affiliate sellers.



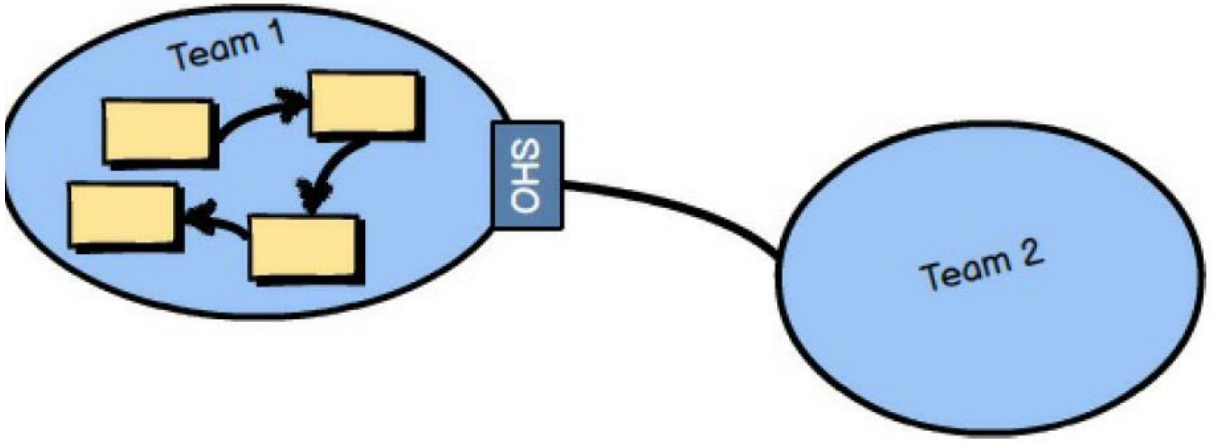
Anticorruption Layer

- Downstream team creates a translation layer between its Ubiquitous Language (model) and the Ubiquitous Language (model) that is upstream to it.
- The layer isolates the downstream model from the upstream model and translates between the two.
- When the **upstream** context **changes**, the **anticorruption** layer must also **change**, but **the rest** of the downstream context can **remain unchanged**.
- Create an Anticorruption Layer so that you can produce model concepts on your side of the integration that specifically fit your business needs and that keep you completely isolated from foreign concepts.



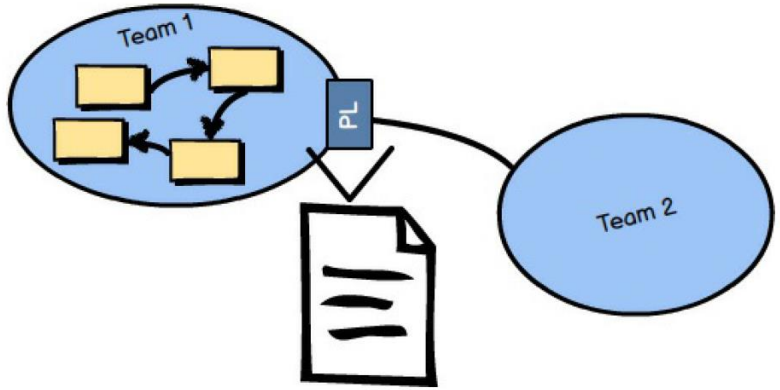
Open Host Service

- An Open Host Service defines a protocol or interface that gives access to your Bounded Context as a set of services.
- The protocol is “open” so that all who need to integrate with your Bounded Context can use it with relative ease.
- Well documented API



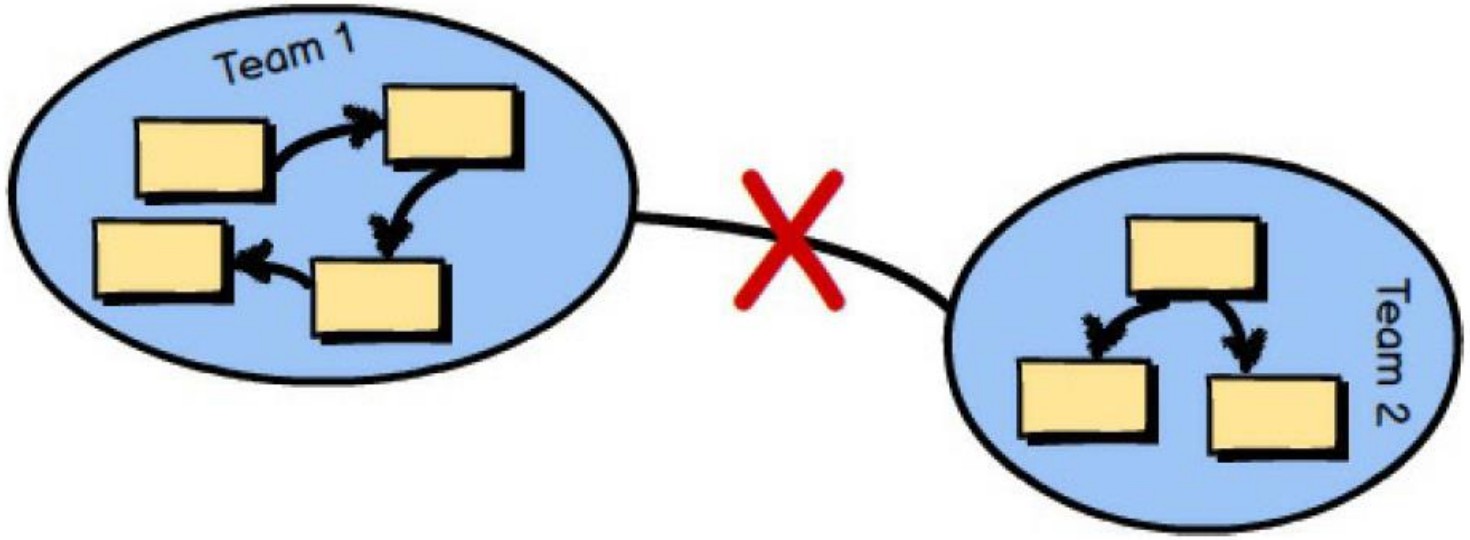
Published Language

- A Published Language is a well-documented information exchange language enabling simple consumption and translation by any number of consuming Bounded Contexts.
- XML Schema, JSON Schema…
- Often an Open Host Service serves and consumes a Published Language.
- This combination makes the translations between two Ubiquitous Languages very convenient.



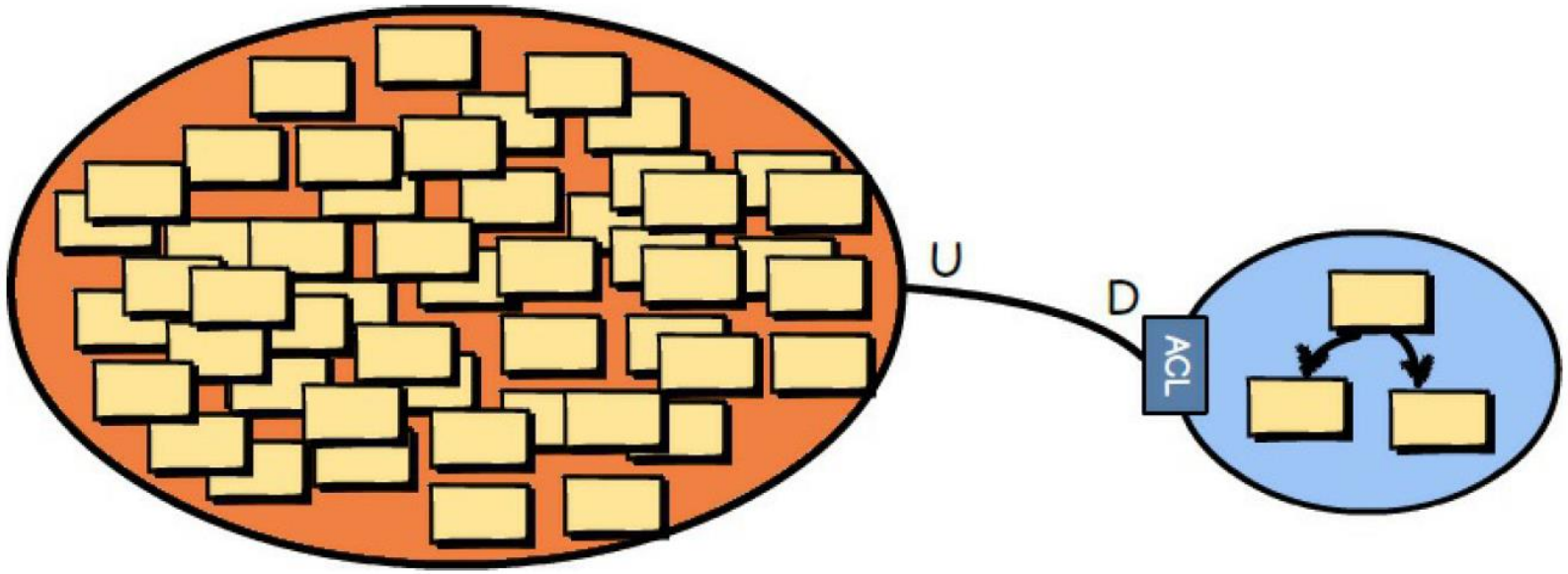
Separate Ways

- Separate Ways describes a situation where integration with one or more Bounded Contexts will not produce significant payoff through the consumption of various Ubiquitous Languages.
- In this case produce your own specialized solution in your Bounded Context and forget integrating for this special case.

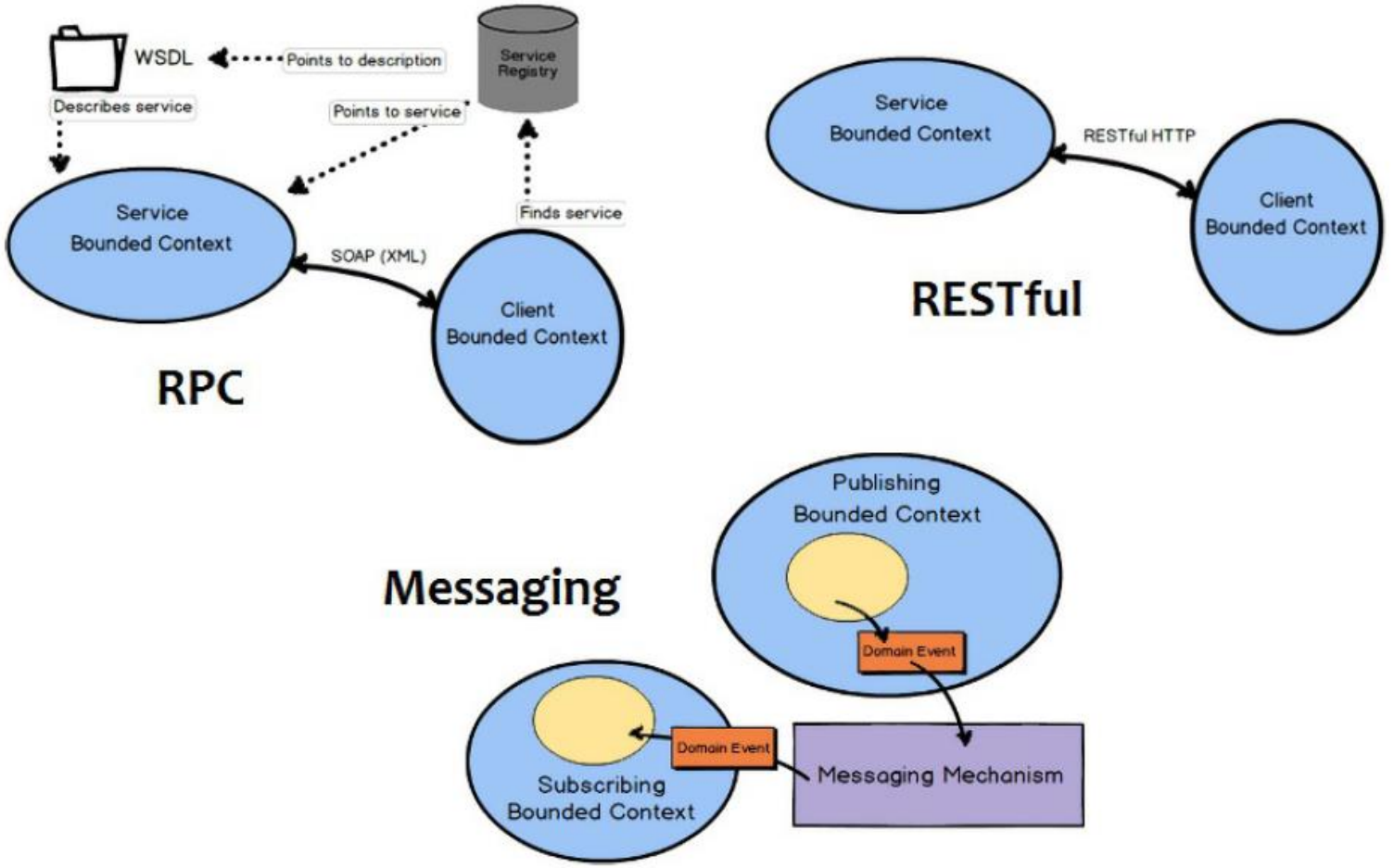


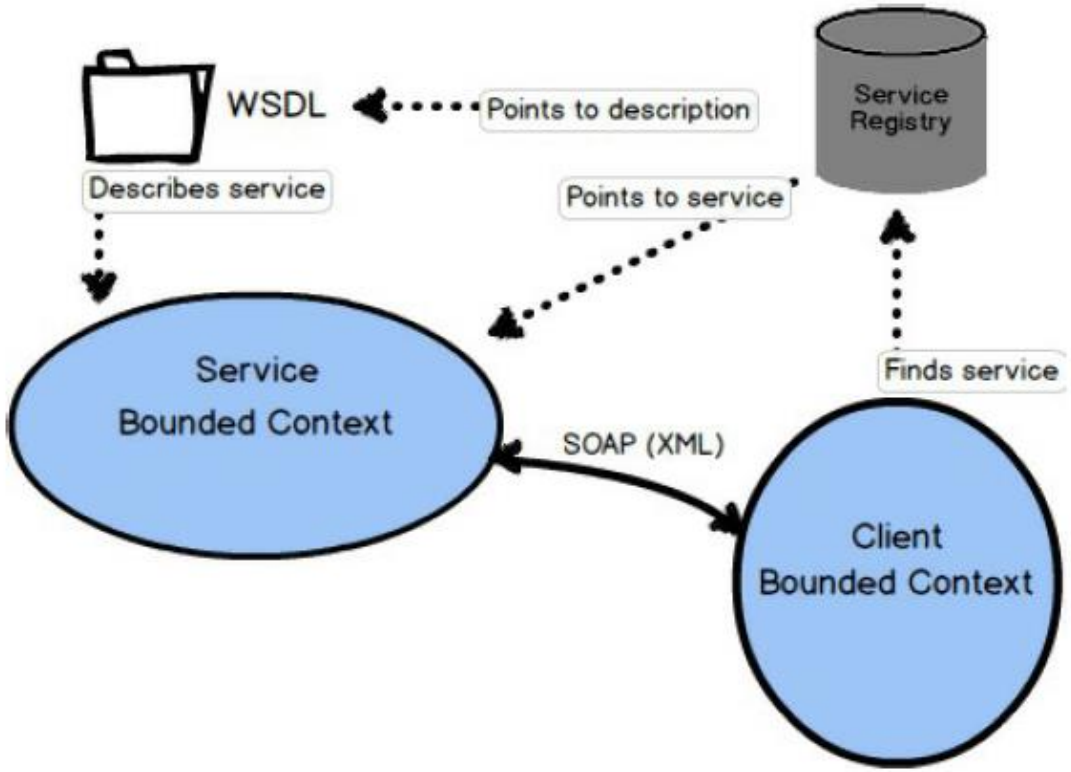
How to integrate with “Big Ball of Mud”?

- Even if you are able to avoid creating a Big Ball of Mud by employing DDD techniques, you may still need to integrate with one or more.
- Try to create an Anticorruption Layer against each legacy system.
- Whatever you do, *don’t speak that language!*



Making Good Use of Context Mapping





Remote Procedure Calls (RPC) with SOAP

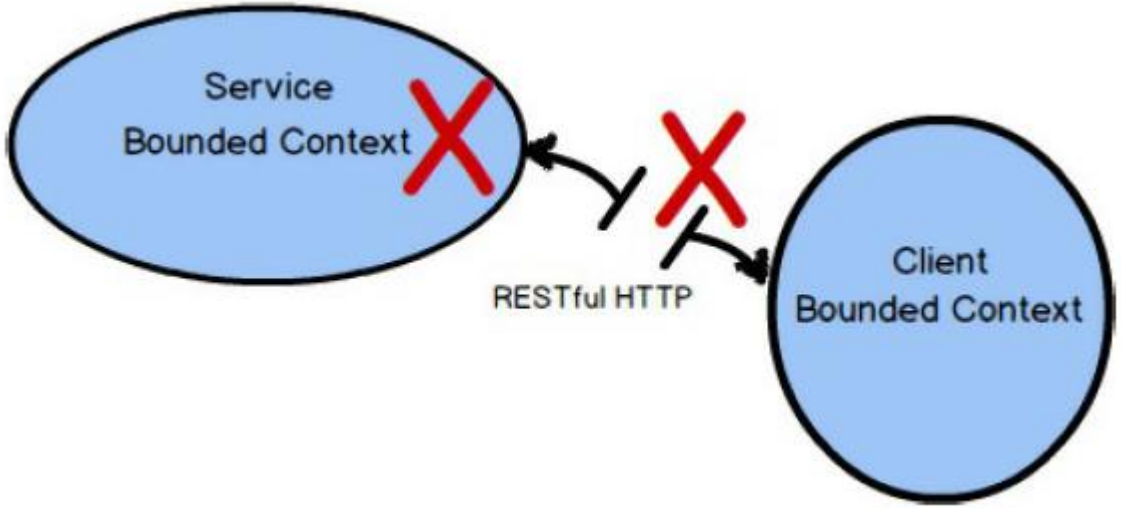
- RPC over SOAP also implies strong coupling between a client *Bounded Context* and the *Bounded Context* providing the service.
- The idea behind RPC with SOAP is to make using services from another system look like a simple, local procedure or method invocation.
- The SOAP request must travel over the network, reach the remote system, perform successfully, and return results over the network.
- It is not fault-tolerant.
- **Isolate your model with Anticorruption Layer**

RESTful HTTP

- Integration using RESTful HTTP focuses attention on the resources that are exchanged between Bounded Contexts , as well as the four primary operations: POST, GET, PUT, and DELETE.
- Many find that the REST approach to integration works well because it helps them define good APIs for distributed computing.
- A service Bounded Context that sports a REST interface should provide an Open Host Service and a Published Language.
- Resources deserve to be defined as a Published Language , and combined with your REST URIs they will form a natural Open Host Service.

RESTful - Problems

- RESTful HTTP will tend to fail for many of the same reasons that RPC does—network and service provider failures.
- A common mistake made when using REST is to design resources that directly reflect the Aggregates in the domain model.
- Doing this forces every client into a *Conformist* relationship, where if the model changes shape the resources will also.



Messaging

- Publish – Subscribe concept.
- Message transfer between Publishing Bounded Context and Subscribing Bounded Context
- Overcome the problem with **blocking forms** such as RPC and REST.
- *At-Least-Once Delivery* - messaging mechanism will periodically redeliver a given message in cases of message loss, slow-reacting or downed receivers.
- *Idempotent Receiver* - receiver of a request performs an operation in such a way that it produces the same result even if it is performed multiple times (receiver uses de-duplication and ignores the repeated message).

Are Domain Event Consumers Conformists?

You may be wondering how *Domain Events* can be consumed by another *Bounded Context* and not force that consuming *Bounded Context* into a *Conformist* relationship. As recommended in *Implementing Domain-Driven Design* [IDDD] , and specifically in Chapter 13, "Integrating Bounded Contexts," consumers should not use the event types (e.g., classes) of an event publisher. Rather, they should depend only on the schema of the events, that is, their *Published Language*. This generally means that if the events are published as JSON, or perhaps a more economical object format, the consumer should consume the events by parsing them to obtain their data attributes.