Welcome

Domain Driven Design

Prakash Badhe

prakash.badhe@vishwasoft.in

Copyright Notice

Survival of the Software

- Every software system has a model at its heart.
- If this model matches the underlying domain well, the software will accept enhancements more easily, and it has a much better chance of surviving and thriving intact for years.

Software Development

- The software development is a kind of manufacturing activity.
- A lot of projects get messed up for one simple reason - software development is *all* design and not considering the domain aspect fully.
- This design should be focused more on the domain.
- During the design if everyone spent some time modelling the domain they are working in, discover the different concepts that model required, and every one in the team had that knowledge; then time and money can be saved to avoid future iterations in application because of inferior knowledge about the domain.

Domain Driven Design

- Domain-driven design (DDD) is an approach to software development for complex needs by connecting the implementation to an evolving model.
- It supports the following goals:
  - Place the project's primary focus on the core domain and domain logic.
  - Base complex designs on a model of the domain
  - Initiate a creative collaboration between technical and domain experts to iteratively refine a conceptual model that addresses particular domain problems.

Domain Driven Design

- Domain-driven design is the design with focus upon and application of the domain concept, as it applies to the development of software.
- It aims to ease the creation of complex applications by connecting the related pieces of the software into an ever-evolving model.

DDD Focus

The DDD focuses on three core principles:

- Focus on the core domain and domain logic.
- Base complex designs on models of the domain.
- Constantly collaborate with domain experts, in order to improve the application model and resolve any emerging domain-related issues.

Purpose of DDD

The purpose of DDD is to model the *behavior* of a system such that the result is a useful *abstraction* of its functional requirements. This means that a properly designed domain is modeled according to the *behavior* of the system, not the data.

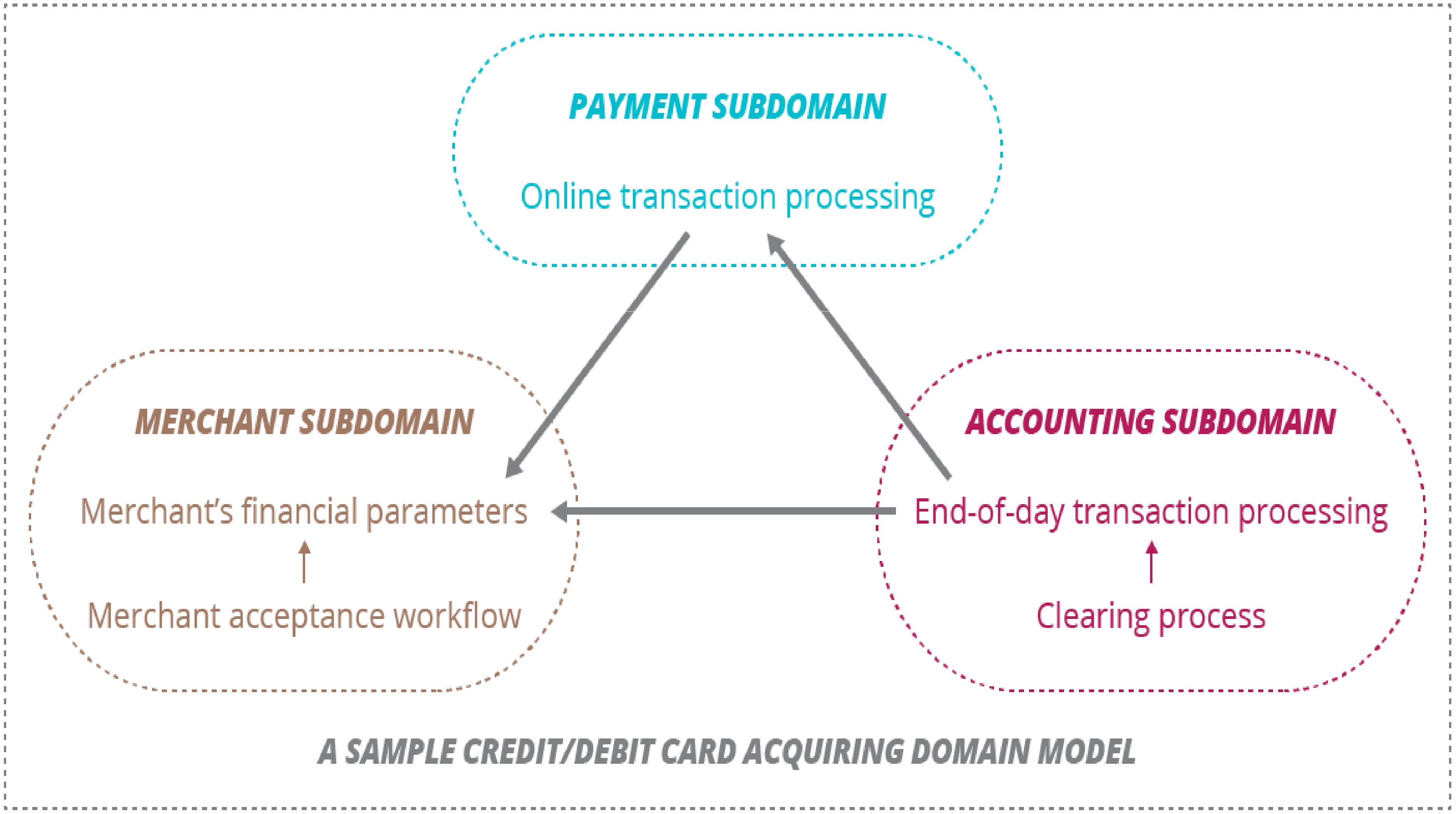
What domain is all about ?

- **Domain : I**n the realm of software engineering domain is the subject area on which the application is intended to apply.
- During the application development, the domain is the “sphere of knowledge and activity around which the application logic revolves.”

Model in a context

- **Model**: A system of abstractions that describes selected aspects of the domain and used to solve problems related to that domain.
- **Context**: The setting in which a word or statement appears that determines its meaning. Statements about a model can only be understood in a context.

Domain Model

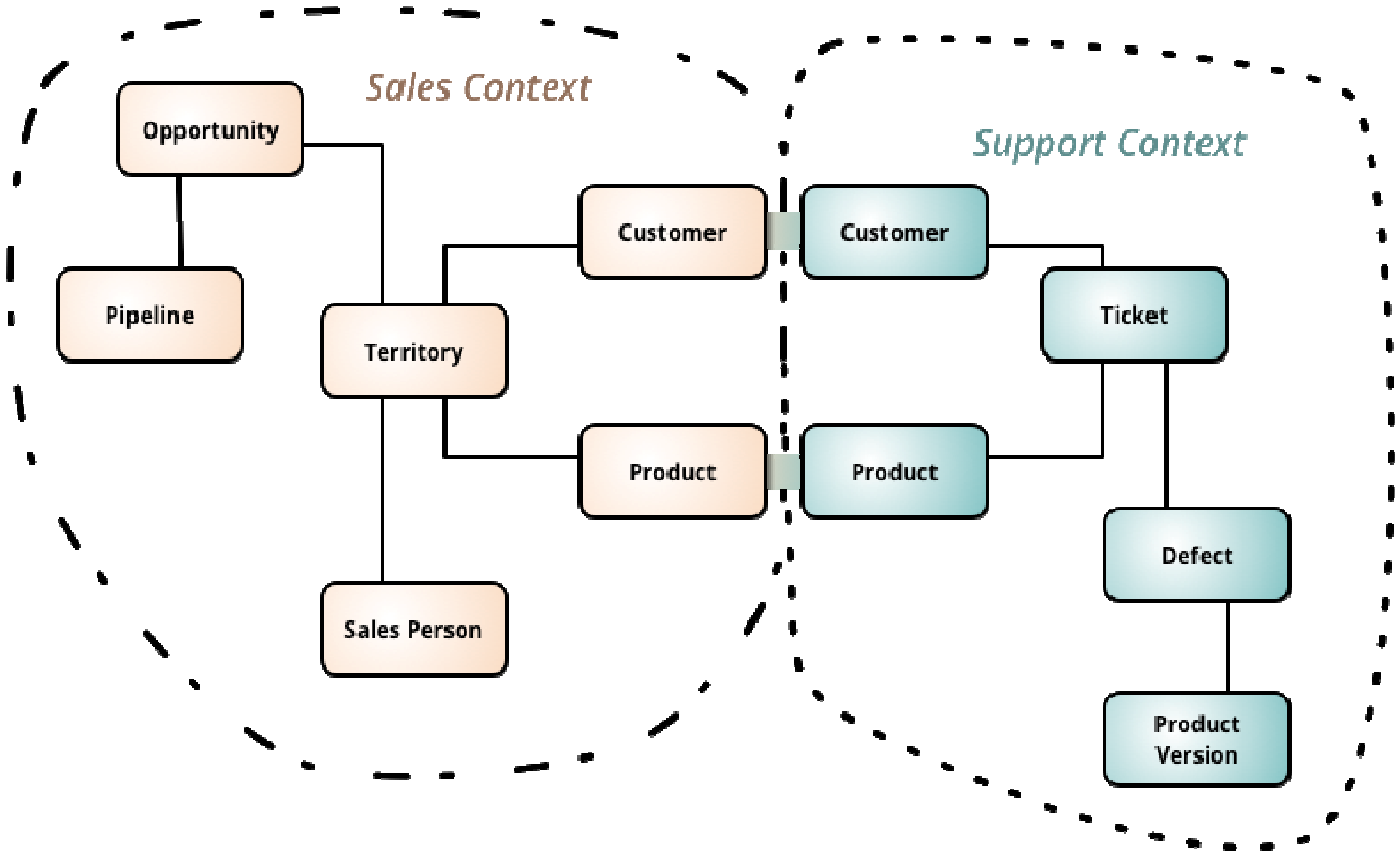

PAYMENT SUBDOMAIN
Online transaction processing
MERCHANT SUBDOMAIN
Merchant's financial parameters
Merchant acceptance workflow
ACCOUNTING SUBDOMAIN
End-of-day transaction processing
Clearing process
A SAMPLE CREDIT/DEBIT CARD ACQUIRING DOMAIN MODEL

Model Boundaries

Bounded Context: A description of a boundary (typically a subsystem, or the work of a specific team) within which a particular model is defined and applicable.

DDD deals with large models by dividing them into different Bounded Contexts and aware about their interrelationships.

Model in a context



Language of DDD

Ubiquitous Language: A language structured around the domain model and used by all team members to connect all the activities of the team with the software.

Analogous to UML for class design

DDD Building Blocks

- Used to create and modify domain models
- **Entity**: An object that is identified by its consistent thread of continuity, as opposed to traditional objects, which are defined by their attributes.
- **Value Object**: An immutable (unchangeable) object that has attributes, but no distinct identity.
- **Domain Event**: An object that is used to record a discrete event related to model activity within the system. While *all* events within the system could be tracked, a domain event is only created for event types which the domain experts care about.

Domain Aggregate as group

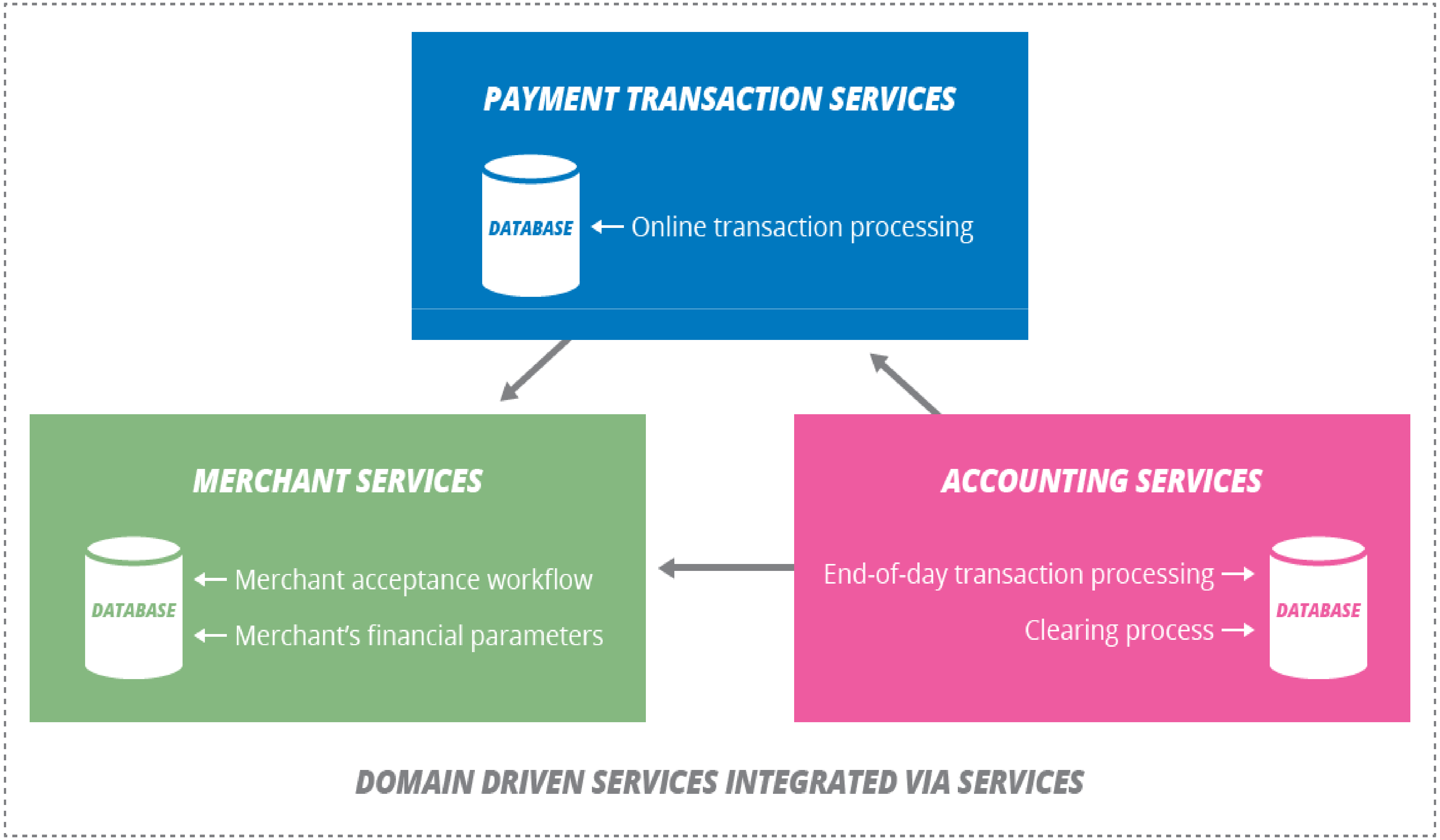
- **Aggregate**: A cluster of entities and value objects with defined boundaries around the group.
- Instead of allowing every single entity or value object to perform all actions on its own, the collective aggregate of items is assigned a singular aggregate root item.
- The external objects no longer have direct access to every individual entity or value object within the aggregate, but instead only have access to the single aggregate root item, and use that to pass along instructions to the group as a whole.
- The aggregates serve two purposes: domain simplification, and technical improvements.

- The aggregates serve two purposes: domain simplification, and technical improvements.
- All aggregates are eventually consistent with each other.

Service and Repository in DDD

- **Service**: service is an operation or form of business logic that doesn’t fit within the realm of objects, rather it is outside the context.
- If some functionality must exist, but it cannot be related to an entity or value object, it’s a service.
- **Repositories**: Its not the common version control repositories.
- The DDD repository is a service that uses a global interface to provide access to all entities and value objects that are within a particular aggregate collection.
- Methods should be defined to allow for creation, modification, and deletion of objects within the aggregate.
- However, by using this repository service to make data queries, the goal is to *remove such data query capabilities from within the business logic of object models.*

Services in DDD


PAYMENT TRANSACTION SERVICES
DATABASE
Online transaction processing
MERCHANT SERVICES
DATABASE
Merchant acceptance workflow
Merchant's financial parameters
ACCOUNTING SERVICES
End-of-day transaction processing
Clearing process
DATABASE
DOMAIN DRIVEN SERVICES INTEGRATED VIA SERVICES

Factory in DDD

- **Factories**: DDD suggests the use of a factory, which encapsulates the logic of creating complex objects and aggregates.
- Generally the client has no knowledge of the inner-workings of object manipulation in the factory.

Patterns in DDD

- SPECIFICATION PATTERN Use the specification pattern when there is a need to model rules, validation and selection criteria.
- The specification implementations test whether an object satisfies all the rules of the specification

Strategy pattern

- Also known as the Policy Pattern is used to make algorithms interchangeable.
- In this pattern, the varying “part” is factored out.
- An example to determine the success of a project, based on two calculations: (1) a project is successful if it finishes on time, or (2) a project is successful if it does not exceed its budget.

```
public class Project {
boolean isSuccessfulByTime();
boolean isSuccessfulByBudget();
}
```

Success strategy

- We encapsulate the specific calculations in policy implementation classes that contain the algorithm for the two different calculations.

-
  ```
  interface ProjectSuccessPolicy {
    Boolean isSuccessful(Project p);
  }
  ```
- `class SuccessByTime implements ProjectSuccessPolicy { ... }`
- `class SuccessByBudget implements ProjectSuccessPolicy { ... }`

Project with strategy

- Refactoring the Project class to use the policy, encapsulate the criteria for success in the policy implementations and not the Project class itself.
- class Project {

```
 boolean isSuccessful(ProjectSuccessPolicy policy)
 {
  return policy.isSuccessful(this); }
}
```

Applying DDD

- At application level, the DDD is applied at domain design and major focus is on it.
- The domain design is understanding completely the business process involved with model objects.
- The services and repositories serve as external components to work with aggregates and domain models.

DDD with frameworks

- Hibernate working with domain entities to manage them as persistent entities.
- The Grails framework : High level on top of Spring and Hibernate supports high level abstractions about domain model, where you can focus on domain design and the framework takes care of the wrapper boiler plate coding.
- The Spring Data an implementation of JPA works in similar mode as ORM to manage the domain entities.

DDD Advantage

- **Ease of Communication**
- **Improves Flexibility in design**
- **Emphasizes Domain Over Interface**
- The practice of building around the concepts of domain and what the domain experts within the project advise, DDD often produces applications that are accurately suited for and representative of the domain, as opposed to those applications which emphasize the *UI/UX first and foremost*.

DDD advantage..

- **Improved Patterns:** The DDD gives software developers the principles and patterns to solve tough problems in software and, at times, in business.
- **Reduced Risk of Misunderstandings.**
- **Better Team Coordination**
  The coordination is more people driven (since everyone is using the same terminology) that is less top driven, ensuring fewer bottlenecks.

Challenges of Applying DDD

- More time and effort required to create a Ubiquitous Language to describe actions on domain objects
- Involving domain experts at the outset and continuously with the project
- It changes the way developers think about solutions in their domain

DDD disadvantages

- **Requires Robust Domain Expertise**
- **Encourages Iterative Practices**: DDD practices strongly rely on constant iteration and continuous integration in order to build a malleable project that can adjust itself when necessary.
- **Not suitable for Highly Technical Projects**
- DDD is not very well-suited for applications that have marginal domain complexity, but conversely have a great deal of technical complexity.