Domain Driven Design

Prof. Dr. Raphael Herding

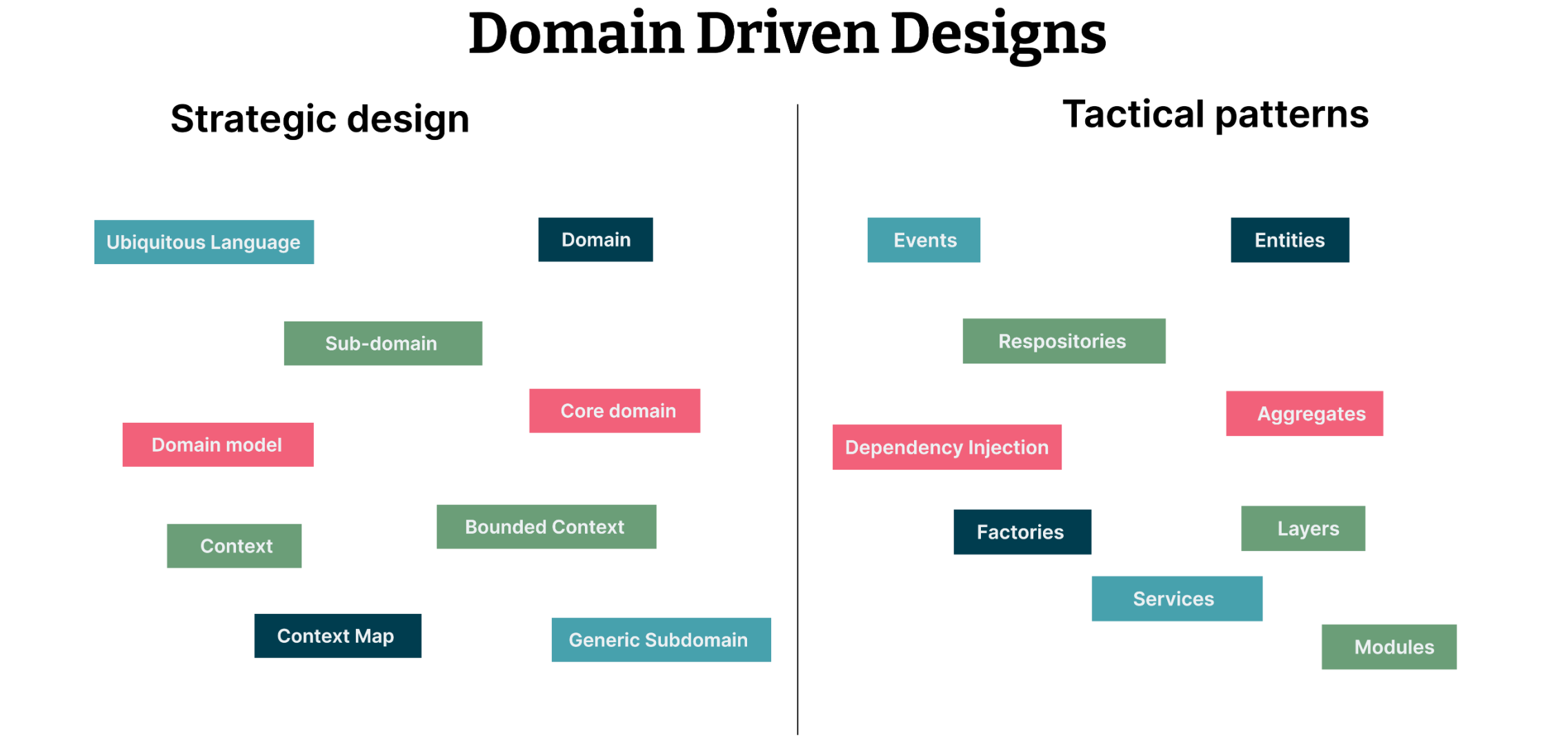
Domain Driven Design
Definition

- DDD (Domain-Driven Design) is a software development methodology for building complex systems by focusing on the business domain.

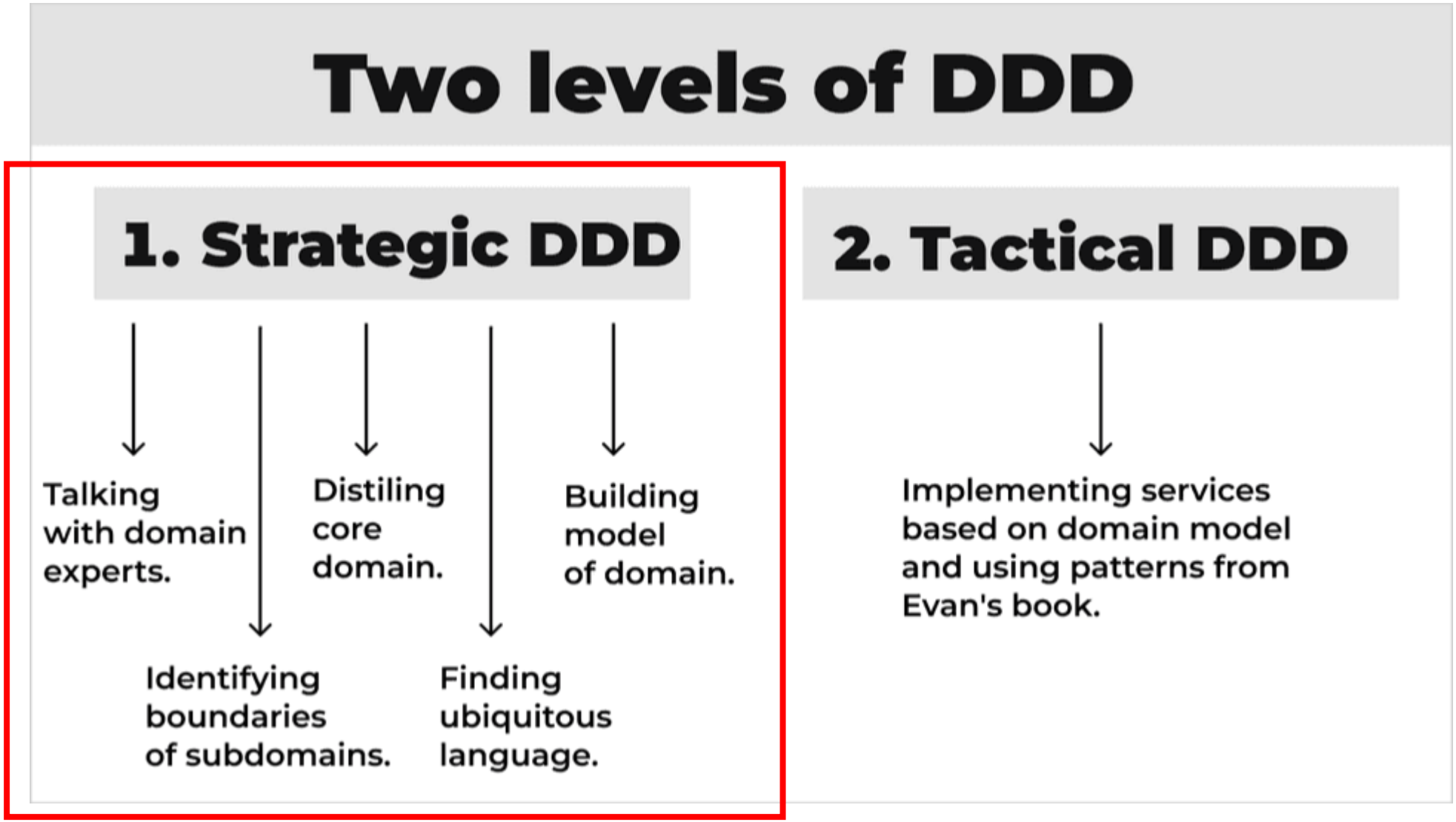
- Its core idea is to tightly integrate the code structure with real business needs.

- **In one sentence**: DDD is about using code to reflect the essence of the business, rather than just implementing functionality.

Domain Driven Design
Tactical. Vs. Strategical



Domain Driven Design
Tactical. Vs. Strategical



Domain Driven Design
Definition

- In traditional development, we follow requirement documents and write if-else logic accordingly.
  - how the database is designed determines how the code is written.

- In DDD, we work together with **business stakeholders** to build domain models.
- The code mirrors the business (**when business changes, code adapts accordingly**).

Domain Driven Design
Ubiquitous Language

- Creation of a common language between all stakeholders involved in a project, such as domain experts, developers, and end-users.

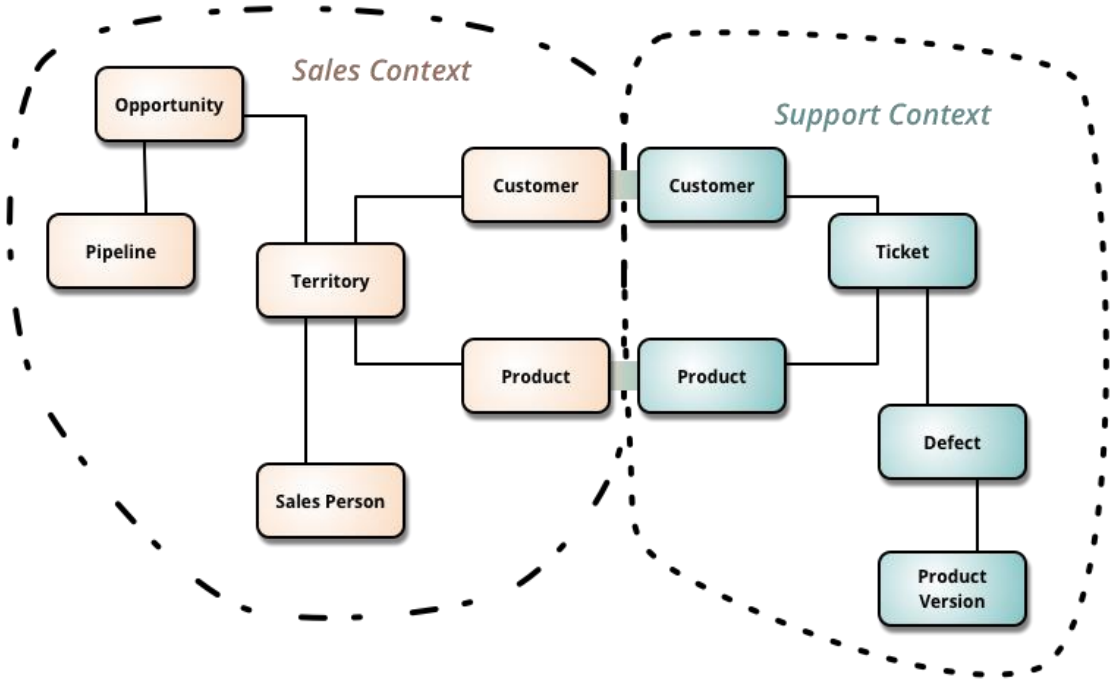
Define common terminology

The same word can vary from context

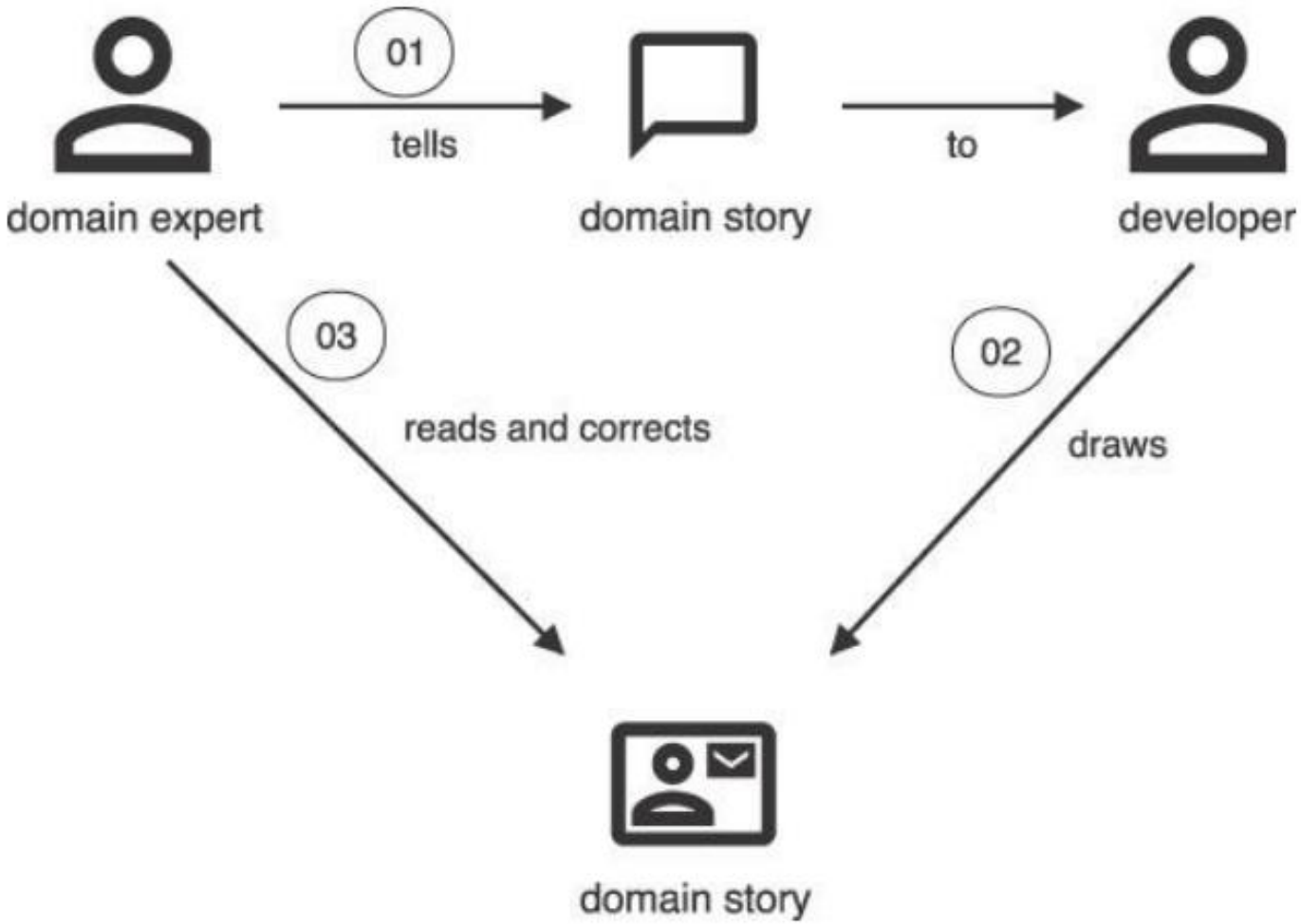
The common language between experts and the engineering team.

Domain Driven Design
Bounded Context

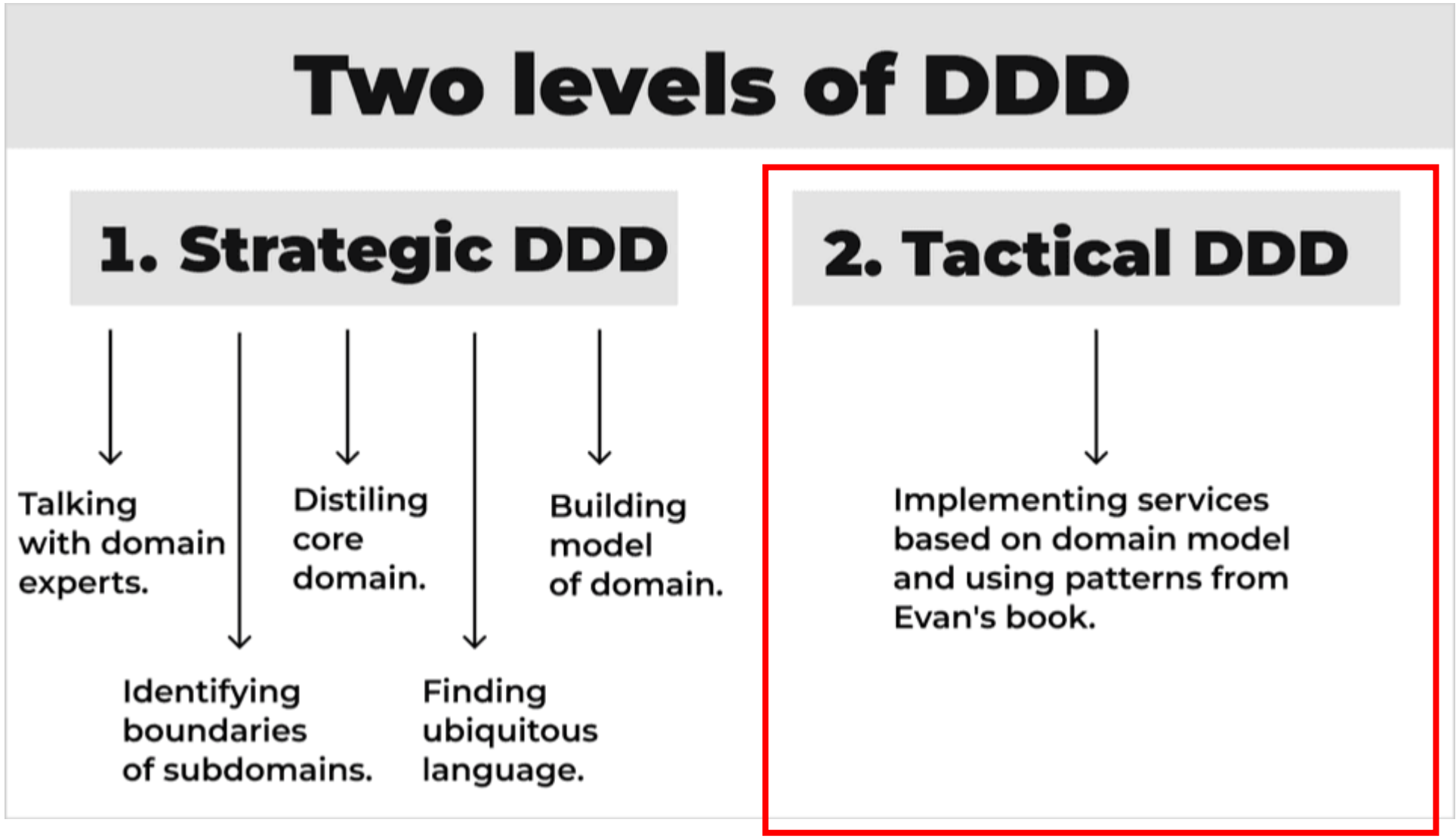
- Dividing the large domain model in smaller pieces.
- **Bounded Context**:
  - Submodel tightening to one business domain (e.g. Sales or Support)
  - Consists of unique domain objects (*Pipeline*, *Opportunity*, *Ticket*, *Defect*) and shared domain objects (*Product*, *Customer*)



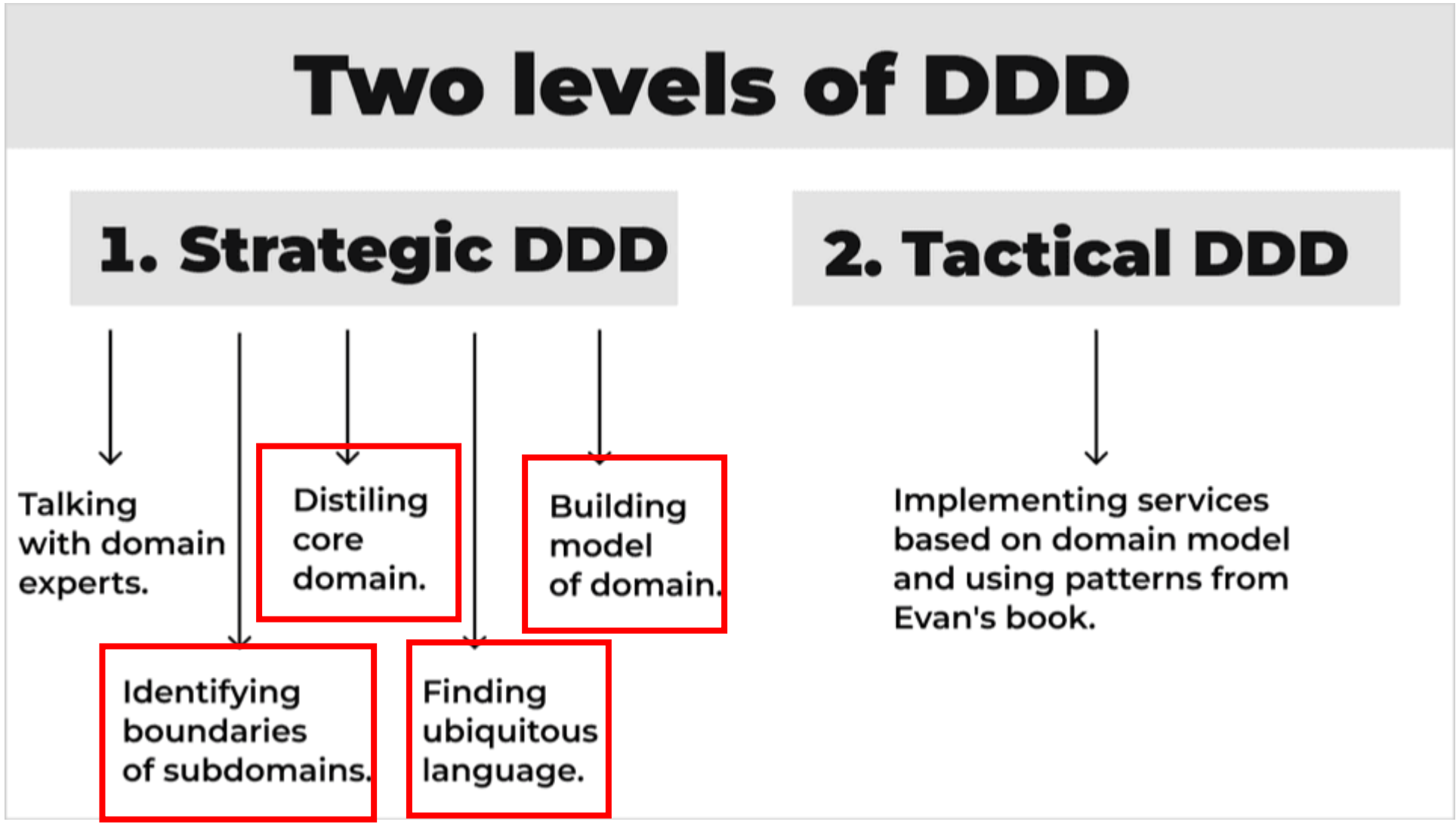
Domain Driven Design
Collaboration



Domain Driven Design
Tactical. Vs. Strategical



Domain Driven Design
Tactical. Vs. Strategical



Lab

- Lab ddd-01 Task 1 and task 2 only

Domain Driven Design
Traditional Example

- Suppose we're building a user registration feature with the following business rules:
  - The username must be unique.
  - The password must meet complexity requirements.
  - A log must be recorded after registration.

```
@Controller
public class UserController {
    public void register(String username, String password) {
        // Validate password
        // Check username
        // Save to database
        // Record log
        // All logic mixed together
    }
}
```

Domain Driven Design
Traditional Example

- Suppose we're building a user registration feature with the following business rules:
  - The username must be unique
  - The password must meet complexity requirements
  - A log must be recorded after registration

```
@Controller
public class UserController {
    public void register(String username, String password) {
        // Validate password
        // Check username
        // Save to database
        // Record log
        // All logic mixed together
    }
}
```

Domain Driven Design
Traditional Example

- … we should separate concerns using layers like controller, service, and DAO.

```
// Service layer: only controls the flow, business rules are scattered
public class UserService {
    public void register(User user) {
        // Validation Rule 1: implemented in a utility class
        ValidationUtil.checkPassword(user.getPassword());
        // Validation Rule 2: implemented via annotation
        if (userRepository.exists(user)) { ... }
        // Data is passed directly to DAO
        userDao.save(user);
    }
}
```

- Flow is much clearer and **layered** as well!

Domain Driven Design
Traditional Example

- … we should separate concerns using layers like controller, service, and DAO.

```
// Service layer: only controls the flow, business rules are scattered
public class UserService {
    public void register(User user) {
        // Validation Rule 1: implemented in a utility class
        ValidationUtil.checkPassword(user.getPassword());
        // Validation Rule 2: implemented via annotation
        if (userRepository.exists(user)) { ... }
        // Data is passed directly to DAO
        userDao.save(user);
    }
}
```

Validation Layer??

Data Layer??

Storage Layer??

- Flow is much clearer and **layered** as well!

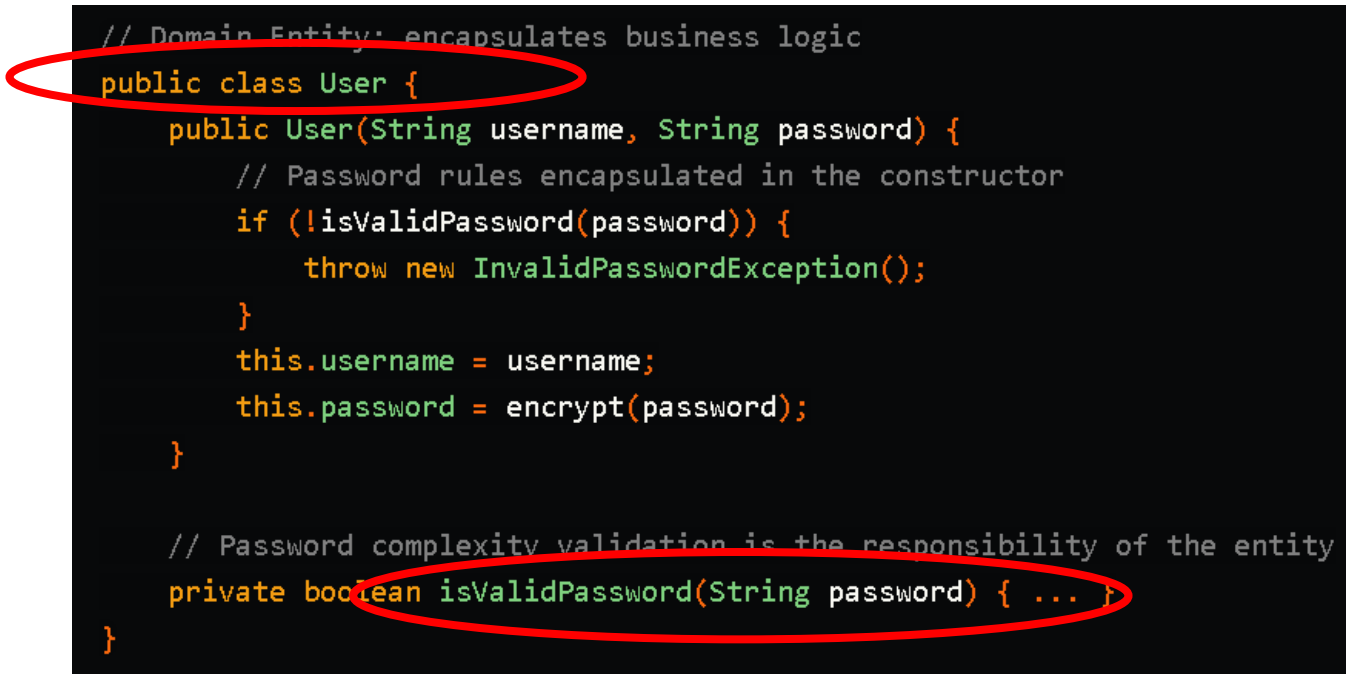
Domain Driven Design
Definition

- Is DDD Layered Architecture?
  - **No**!
  - Although the code is layered and structurally divided, it is not DDD.
  - …but we can use layered architecture for DDD.
- In traditional layered code
  - User object is just a data carrier (anemic model).
  - Its business logic is offloaded elsewhere.

```java
// Service layer: only controls the flow, business rules are scattered
public class UserService {
    public void register(User user) {
        // Validation Rule 1: implemented in a utility class
        ValidationUtil.checkPassword(user.getPassword());
        // Validation Rule 2: implemented via annotation
        if (userRepository.exists(user)) { ... }
        // Data is passed directly to DAO
        userDao.save(user);
    }
}
```

Domain Driven Design
Definition

- In DDD, **most** logic is encapsulated within the domain object (e.g. like password validation).



```
// Domain Entity: encapsulates business logic
public class User {
    public User(String username, String password) {
        // Password rules encapsulated in the constructor
        if (!isValidPassword(password)) {
            throw new InvalidPasswordException();
        }
        this.username = username;
        this.password = encrypt(password);
    }

    // Password complexity validation is the responsibility of the entity
    private boolean isValidPassword(String password) { ... }
}
```

- “password validation” represents the **business logic** of the User **domain object**.
- The User object is no longer a simple “data bag” or data transfer object (DTO).

Domain Driven Design
Definition

- … but DDD is more than just putting business logic into the domain object.

- Besides **layering**, the essence of DDD lies in deepening business expression through the following patterns:
  - Aggregate Root
  - Domain Service vs. Application Service
  - Domain Events

Domain Driven Design
Aggregate Root

- **Scenario**: A user (*User*) is associated with shipping addresses (*Address*)

- **Traditional approach**: manage *User* and *Address* separately in the **Service layer**

- **DDD approach**: treat *User* as the **aggregate root** and control the addition/removal of *Address* through it.

Domain Driven Design
Aggregate Root

- **Scenario**: A user (*User*) is associated with shipping addresses (*Address*)

```java
public class User {
    private List<Address> addresses;

    // The logic to add an address is controlled by the aggregate root
    public void addAddress(Address address) {
        if (addresses.size() >= 5) {
            throw new AddressLimitExceededException();
        }
        addresses.add(address);
    }
}
```

- **DDD approach**: treat *User* as the **aggregate root** and control the addition/removal of *Address* through it.

Lab

- Refactor the drawn model from ddd-01 Task 1 and 2 and define the aggregate roots.

Domain Driven Design
Domain Events

- Use **events** to explicitly express business state changes.
- **Example**: After a user successfully registers, trigger a *UserRegisteredEvent*

```
public class User {
    public void register() {
        // ...registration logic
        this.events.add(new UserRegisteredEvent(this.id)); // Record domain eve
    }
}
```

Lab

- Lab ddd-01 Task 3 only

- Don’t forget to implement tests!

Domain Driven Design
Domain Service vs. Application Service

- **Domain Service**:
  - Handles business logic that spans multiple entities (e.g., transferring money between two accounts)

- **Application Service:**
  - Coordinates the overall process (e.g., calling domain services + sending messages)

Domain Driven Design
Domain Service vs. Application Service

```
// Domain Service: handles core business logic
public class TransferService {
    public void transfer(Account from, Account to, Money amount) {
        from.debit(amount); // Debit logic is encapsulated in Account entity
        to.credit(amount);
    }
}

// Application Service: orchestrates the process, contains no business logic
public class BankingAppService {
    public void executeTransfer(Long fromId, Long toId, BigDecimal amount) {
        Account from = accountRepository.findById(fromId);
        Account to = accountRepository.findById(toId);
        transferService.transfer(from, to, new Money(amount));
        messageQueue.send(new TransferEvent(...)); // Infrastructure operation
    }
}
```

Lab

- Lab ddd-01 Task 4 and task 5 only

- Don’t forget to implement tests!

Domain Driven Design
Difference to Traditional Development

- **Traditional Development:**
  - **Ownership of Business Logic**: Scattered across Services, Utils, Controllers.
  - **Role of the Model**: Data carrier (anemic model).
  - **Impact on Technical Implementation**: Schema is driven by database table design.

- **Domain Driven Design**:
  - **Ownership of Business Logic**: Encapsulated in domain entities or domain services.
  - **Role of the Model**: Business model that carries behavior (rich model).
  - **Impact on Technical Implementation**: Schema is driven by business needs.

Domain Driven Design
Full Example – Placing an Order in E-Commerce

```java
// Service layer: bloated order placement logic
public class OrderService {
    @Autowired private InventoryDAO inventoryDAO;
    @Autowired private CouponDAO couponDAO;

    public Order createOrder(Long userId, List<ItemDTO> items, Long couponId)
        // 1. Stock validation (scattered in Service)
        for (ItemDTO item : items) {
            Integer stock = inventoryDAO.getStock(item.getSkuId());
            if (item.getQuantity() > stock) {
                throw new RuntimeException("Insufficient stock");
            }
        }

        // 2. Calculate total amount
        BigDecimal total = items.stream()
                .map(i -> i.getPrice().multiply(i.getQuantity()))
                .reduce(BigDecimal.ZERO, BigDecimal::add);
```

```java
        // 2. Calculate total amount
        BigDecimal total = items.stream()
                .map(i -> i.getPrice().multiply(i.getQuantity()))
                .reduce(BigDecimal.ZERO, BigDecimal::add);

        // 3. Apply coupon (logic hidden in utility class)
        if (couponId != null) {
            Coupon coupon = couponDAO.getById(couponId);
            total = CouponUtil.applyCoupon(coupon, total); // Discount logic
        }

        // 4. Save order (pure data operation)
        Order order = new Order();
        order.setUserId(userId);
        order.setTotalAmount(total);
        orderDAO.save(order);
        return order;
    }
}
```

Domain Driven Design
Full Example – Placing an Order in E-Commerce

```java
// Service layer: bloated order placement logic
public class OrderService {
    @Autowired private InventoryDAO inventoryDAO;
    @Autowired private CouponDAO couponDAO;

    public Order createOrder(Long userId, List<ItemDTO> items, Long couponId)
        // 1. Stock validation (scattered in Service)
        for (ItemDTO item : items) {
            Integer stock = inventoryDAO.getStock(item.getSkuId());
            if (item.getQuantity() > stock) {
                throw new RuntimeException("Insufficient stock");
            }
        }

        // 2. Calculate total amount
        BigDecimal total = items.stream()
                .map(i -> i.getPrice().multiply(i.getQuantity()))
                .reduce(BigDecimal.ZERO, BigDecimal::add);
```

```java
// 2. Calculate total amount
BigDecimal total = items.stream()
        .map(i -> i.getPrice().multiply(i.getQuantity()))
        .reduce(BigDecimal.ZERO, BigDecimal::add);

// 3. Apply coupon (logic hidden in utility class)
if (couponId != null) {
    Coupon coupon = couponDAO.getById(couponId);
    total = CouponUtil.applyCoupon(coupon, total); // Discount logic
}

// 4. Save order (pure data operation)
Order order = new Order();
    }
}
```

- Stock validation and coupon logic are scattered across Service, Util, DAO
- The Order object is just a data carrier (anemic); no one owns the business rules
- When requirements change, developers have to "dig" through the Service layer

Domain Driven Design
Full Example – Placing an Order in E-Commerce

```java
// Aggregate Root: Order (carries core logic)
public class Order {
    private List<OrderItem> items;
    private Coupon coupon;
    private Money totalAmount;

    // Business logic encapsulated in the constructor
    public Order(User user, List<OrderItem> items, Coupon coupon) {
        // 1. Stock validation (domain rule encapsulated)
        items.forEach(item -> item.checkStock());

        // 2. Calculate total amount (logic resides in value objects)
        this.totalAmount = items.stream()
                .map(OrderItem::subtotal)
                .reduce(Money.ZERO, Money::add);

        // 3. Apply coupon (rules encapsulated in entity)
        if (coupon != null) {
            validateCoupon(coupon, user); // Coupon rule encapsulated
            this.totalAmount = coupon.applyDiscount(this.totalAmount);
        }
    }

    // Coupon validation logic (clearly owned by the domain)
    private void validateCoupon(Coupon coupon, User user) {
        if (!coupon.isValid() || !coupon.isApplicable(user)) {
            throw new InvalidCouponException();
        }
    }
}
```

```java
// Domain Service: orchestrates the order process
public class OrderService {
    public Order createOrder(User user, List<Item> items, Coupon coupon) {
        Order order = new Order(user, convertItems(items), coupon);
        orderRepository.save(order);
        domainEventPublisher.publish(new OrderCreatedEvent(order)); // Domain
        return order;
    }
}
```

- **Stock validation**: Encapsulated in the *OrderItem* value object.
- **Coupon logic**: Encapsulated within methods of the *Order* entity.
- **Calculation logic**: Ensured precision by the *Money* value object.
- **Business changes**: Only require changes to the domain object.

Domain Driven Design
Full Example – Placing an Order in E-Commerce

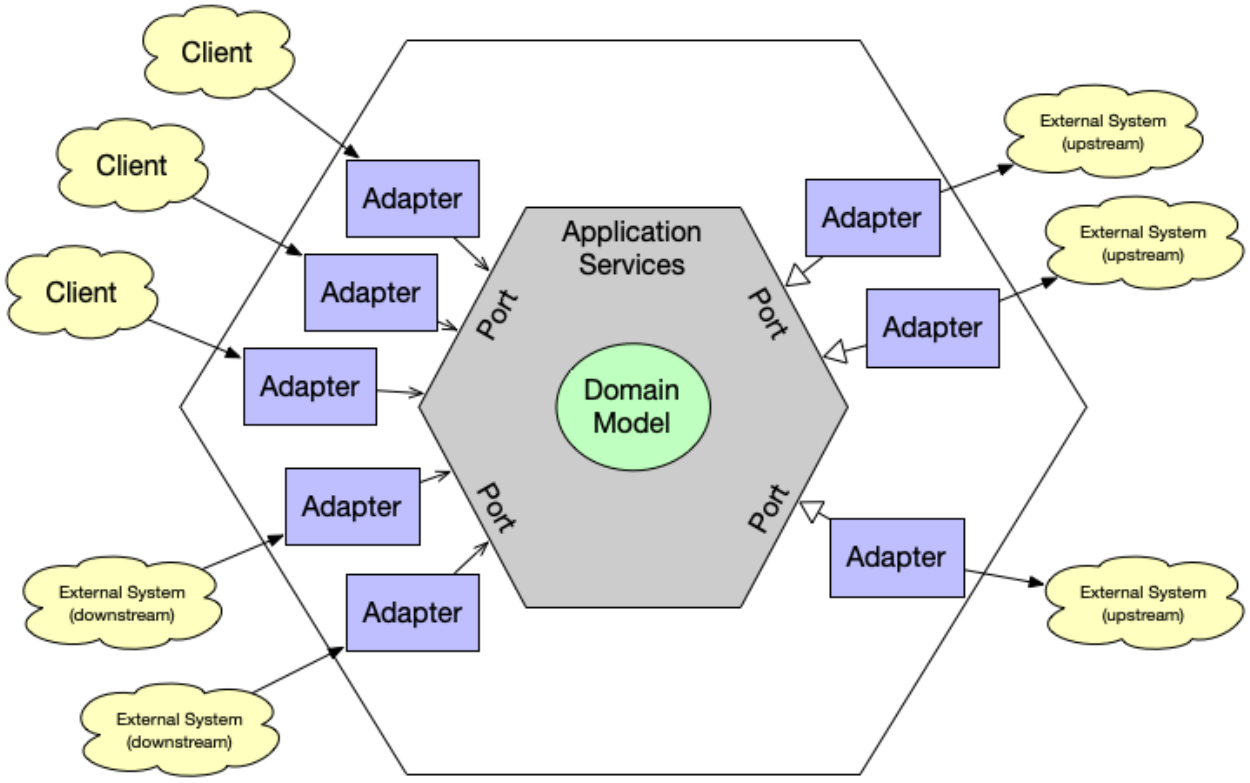
- Suppose there's a new product requirement:
  - Coupons must offer 20€ off for orders over 100€ and apply only to new users.

- **Traditional development:**
  - Modify CouponUtil.applyCoupon() logic.
  - Modify the Service layer to add new-user validation.

- **With DDD**:
  - Modify *Order.validateCoupon()* method in the domain layer.

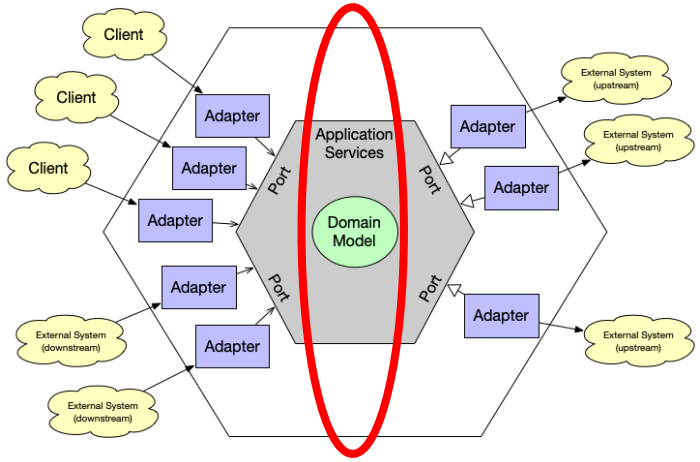
Domain Driven Design
Complex Systems

- Layered Architecture and Hexagonal Architecture support DDD.



Domain Driven Design
Hexagonal Architecture

- Core of hexagonal architecture is the **domain model**.
- Domain model consists of **core business logic**.
- **Application services** of DDD coordinates the process.
- **Ports** are used to expose application service logic to outer applications.
- **Adapters** use that ports to integrate them into the clients application.
- **Adapters** are client-specific.



Lab

- Lab ddd-02
- Don’t forget to implement tests!