DDD 하지않겠는가..?

혼돈 속에서 질서를 찾아서

(+ CQRS)

BeomSeom Kim, https://github.com/tigerwest

시작하기 전에... 옛날 프로그래밍 농담

의사, 엔지니어, 프로그래머 셋이서 누가 제일 가장 오래된 직업인가 토론했습니다.

의사: "하나님이 아담의 갈비뼈로 이브를 만들었으니, 의사가 가장 오래된 직업이다!"

엔지니어: "그 전에 혼돈속에서 세상의 질서를 만들고 건축했으니, 엔지니어가 먼저다!"

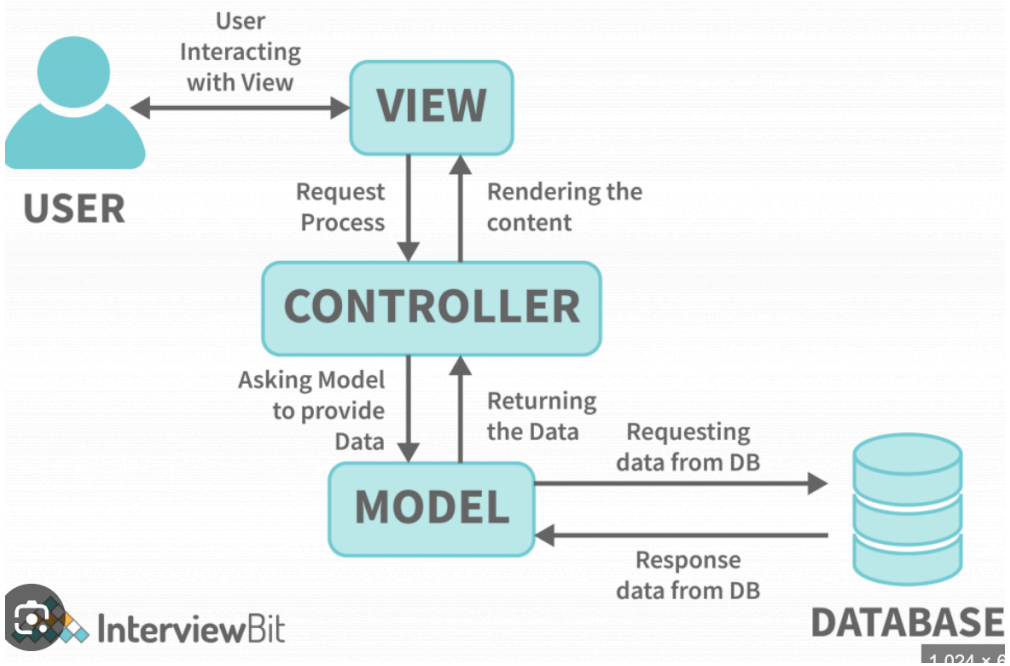
프로그래머: "그 혼돈을 누가 만들었죠?"

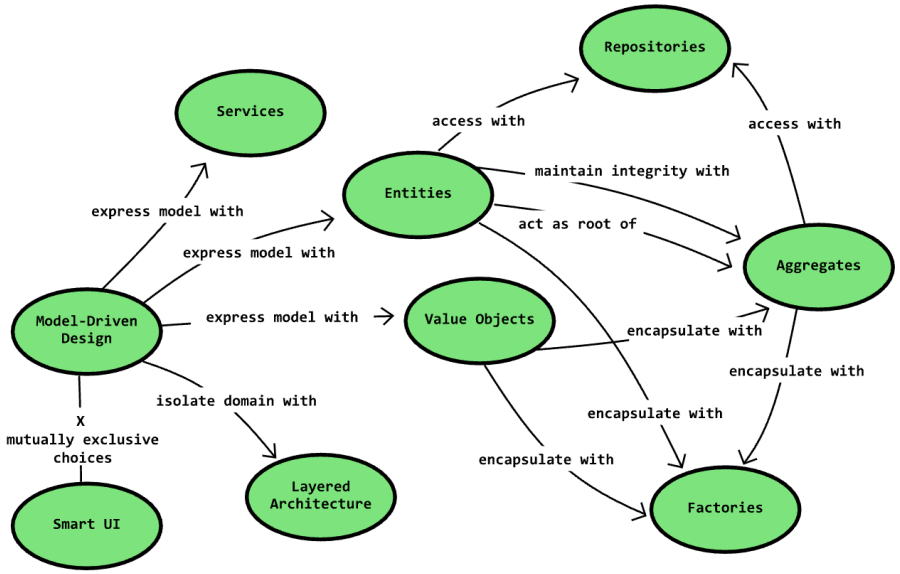
"카오스메이커" 여러분 👩‍💻👨‍💻

이제 혼돈 속 엔지니어로서 질서를 정립할 때

간단하게 DDD를 시작해보세요.

레거시 MVC 패턴의 현실





mvc vs ddd -> mvc가 단순해 보이지만... 실제로는?

비대해진 코드의 현실

전형적인 MVC 구조

```
class UserService {
  getUserList() {} // 조회로직을 무의미한 중개만 해주는 서비스
}
// 비대한 Repository 메서드들
class UserRepository {
  findAll() {}
  findById(id: string) {}
  findByEmail(email: string) {}
  findByRole(role: string) {}
  findByStatus(status: string) {}
  findByCreatedDate(date: Date) {}
  findWithProfile(id: string) {}
  findWithOrders(id: string) {}
  findWithPermissions(id: string) {}
  findActiveUsersWithRecentOrders() {}
  findInactiveUsersOlderThan(months: number) {}
  // ... 끝없이 늘어나는 조회 메서드들
}
```

```
// 1. 도메인 엔티티: 데이터만 보유, 로직 없음
class User {
  id: string;
  email: string;
  password: string;
  status: string;
  // 단순 데이터 컨테이너 (getter/setter만 존재)
}
```

문제점 분석

- Repository의 조회 로직 집중
- Service의 의미 없는 중개
- 도메인 모델이 아무것도 하지않음

DDD의 핵심 개념중 하나: Aggregate

데이터 집합체 (여러 Entity나 ValueObject를 소유할 수 있음)

"함께 변경되어야 하는 것들의 경계"(ACID)

"소유와 참조가 구분되는 경계"

Aggregate 설계 예시

```
type CustomerId = string;
// ✅ Order Aggregate
class Order {
  private items: OrderItem[] = [];
  private totalAmount: Money;

  private customerId: CustomerId; // 주문자 id

  // Aggregate Root를 통해서만 수정 가능
  addItem(product: Product, quantity: number) {
    const item = new OrderItem(product, quantity);
    this.items.push(item);
    this.recalculateTotalAmount(); // 불변성 보장
  }

  removeItem(itemId: string) {
    this.items = this.items.filter((item) => item.id !== itemId);
    this.recalculateTotalAmount(); // 항상 일관성 유지
  }

  private recalculateTotalAmount() {
    this.totalAmount = this.items.reduce(
      (sum, item) => sum.add(item.getSubtotal()),
      Money.ZERO
    );
  }
}
```

Aggregate 작성주의점

1️⃣ 작게 유지하라

```
// ❌ 너무 큰 Aggregate
// 예를들어 유저가 모든걸 소유한다고 연결된것들을 aggregate로 묶지않기
// 너무 큰 aggregate는 유지보수하기 어려움
class Customer {
  orders: Order[] = []; // 수천 개가 될 수 있음
  reviews: Review[] = []; //
  wishlist: Product[] = [];
}
```

2️⃣ 소유가 아니면 ID로 참조하라

```
// ✅ ID 참조
class Order {
  customerId: CustomerId; // ID만
}
```

DAO(Data Access Object) vs Repository 패턴

데이터 접근의 두 가지 철학

그들은 같은 것일까? 🤔

DAO (Data Access Object)

📊 데이터베이스 중심 사고

```
// DAO는 데이터경계없이 자유롭게 조회하는 클래스
class UserDAO {
  async findById(id: string): Promise<UserRow> {
    return db.query("SELECT * FROM users WHERE id = ?", [id]);
  }

  async findByEmail(email: string): Promise<UserRow> {
    return db.query("SELECT * FROM users WHERE email = ?", [email]);
  }

  async insert(data: UserRow): Promise<void> {
    return db.query("INSERT INTO users VALUES (?)", [data]);
  }

  async update(id: string, data: Partial<UserRow>): Promise<void> {
    return db.query("UPDATE users SET ? WHERE id = ?", [data, id]);
  }
}
```

Repository 패턴

🏛️ 도메인 중심 사고

```
// Repository는 Aggregate를 다룸
class OrderRepository {
  async findById(id: OrderId): Promise<Order> {
    // 여러 테이블에서 데이터 조회
    const orderData = await db.orders.findById(id);
    const itemsData = await db.orderItems.findByOrderId(id);

    // Aggregate로 재구성
    return Order.reconstitute(orderData, itemsData);
  }

  async save(order: Order): Promise<void> {
    // Aggregate 전체를 저장
    const snapshot = order.getSnapshot();

    await db.transaction(async (trx) => {
      await trx.orders.upsert(snapshot.order);
      await trx.orderItems.deleteByOrderId(order.id);
      await trx.orderItems.insertMany(snapshot.items);
    });
  }
}
```

핵심 차이점

DAO vs Repository

dao는 데이터의 경계없이 자유롭게 crud

vs

repository는 aggregate를 crud

CQRS(Command Query Responsibility Segregation)패턴

명령과 조회의 분리

CQRS를 통해서 ApplicationService의 메소드를 클래스 핸들러로 분리하기

구현시 파일 트리 예시

```
src/
   ├── application/
   │      ├── command/
   │      │      ├── CreateOrder.ts
   │      │      └── ...
   │      └── query/
   │             ├── GetProductList.ts
   │             └── GetProductDetail.ts
   │
```

리팩토링: 조회는 CQRS + DAO로!

이제 이 지식을 활용해서 정리해볼시간

리마인드: 기존 MVC Repository의 함정

😰 조회를 위한 메서드 폭발

```
class UserRepository {
  // 조회 메서드가 계속 늘어남...
  findById(id: string): Promise<User>;
  findByEmail(email: string): Promise<User>;
  findActiveUsers(): Promise<User[]>;
  findByRole(role: string): Promise<User[]>;
  findWithOrders(id: string): Promise<User>;
  findWithRecentActivity(): Promise<User[]>;
  findByAgeRange(min: number, max: number): Promise<User[]>;
  // ... 끝없는 find 메서드들

  // 정작 중요한 도메인 저장 로직
  save(user: User): Promise<void>; // 이게 Repository의 본질!
}
```

문제점

- ddd에서 말하는 repository에서 벗어나는 책임
- Repository가 비대해짐
- 의미없는 서비스의 중개

임시 해결책: CQRS + DAO

```
// 1 Command: Repository는 일반 ui 조회에서 사용x, 비즈니스 로직에서 사용o
class OrderRepository {
  constructor(private prisma: PrismaClient) {}
  async findById(id: string): Promise<Order> {}
  async save(order: Order): Promise<void> {}
}

// 2 Query: Prisma 직접 사용 (화면로직을 위한 조회)
class GetOrderQueryHandler {
  constructor(private prisma: PrismaClient) {}

  async execute(customerId: string) {
    return this.prisma.order.findMany({
      where: { customerId },
      select: { id: true, total: true, createdAt: true },
    });
  }
}
```

조회로직이 추가되더라도 안심

```
src/
   ├── application/
   │      └── query/
   │             ├── GetProductList.ts
   │             └── GetProductDetail.ts
   │             └── GetProductListByCategory.ts
   │             └── GetProductListByPriceRange.ts
   │             └── GetProductListByStock.ts
```

잠깐, Service를 안 거쳐도 되나요?

🤔 "Controller에서 바로 QueryHandler를?"

조회 로직이 Service를 안 타고 이렇게 호출해도 되나요?

네 하셔도 됩니다.

- 기존에는 Service가 단순 중개자 역할
- 불필요한 레이어 추가
- 코드 복잡도만 증가

최근 다시 돌아가고 있는 접근 방식: UI와 조회의 밀접성

GraphQL의 철학

```
// GraphQL Resolver - UI 요구사항과 직접 연결
const resolvers = {
  Product: {
    // 필요할 때만 추가 조회
    reviews: async (product, _, { prisma }) => {
      return prisma.review.findMany({
        where: { productId: product.id },
      });
    },
  },
};
```

React Server Components의 접근

🎯 UI와 데이터 조회의 공동 배치 (Co-location)

```
// app/products/page.tsx
async function ProductList({ category }: Props) {
  // UI 컴포넌트에서 직접 데이터 조회
  const products = await prisma.product.findMany({
    where: { category },
    select: {
      id: true,
      name: true,
      price: true,
      thumbnail: true,
    },
  });

  return (
    <div className="grid grid-cols-3 gap-4">
      {products.map((product) => (
        <ProductCard key={product.id} product={product} />
      ))}
    </div>
  );
}
```

응집도(Cohesion)의 관점

📦 관련된 것들을 함께 두기

ui와 데이터조회가 가까울수록 응집도가 높아요.

불필요한 서비스 중개자를 줄일수 있어요.

항상 나오는 Application Service의 현실

😥 빈약한 도메인 모델의 증상(transactional script pattern)

```
// 전형적인 Application Service
// 비즈니스로직이 전부 ApplicationService에 노출되어 있음
class OrderService {
  async createOrder(dto: CreateOrderDto) {
    if (!dto.items || dto.items.length === 0) {
      throw new Error("주문 항목이 비어있습니다");
    }

    const totalAmount = dto.items.reduce(
      (sum, item) => sum + item.price * item.quantity,
      0
    );

    // 3. 할인 계산도 Service에
    const discount = this.calculateDiscount(totalAmount, dto.couponCode);

    // 4. 도메인 객체는 단순 데이터 홀더
    const order = new Order();
    order.customerId = dto.customerId;
    order.items = dto.items;
    order.totalAmount = totalAmount - discount;

    return this.orderRepository.save(order);
  }

  private calculateDiscount(amount: number, couponCode?: string) {
    // 비즈니스 로직이 Service에 산재
  }
}
```

도메인 중심으로 리팩토링

✅ 풍부한 도메인 모델 (Rich Domain Model)

```
// Domain Entity가 비즈니스 로직을 소유
class Order {
  private items: OrderItem[] = [];
  private status: OrderStatus;
  private totalAmount: Money;

  // 정적 팩토리 메서드
  static create(customerId: CustomerId, items: OrderItem[]): Order {
    if (items.length === 0) {
      throw new EmptyOrderError();
    }

    const order = new Order(customerId);
    items.forEach((item) => order.addItem(item));
    return order;
  }

  submit(): DomainEvent[] {
    this.validateCanSubmit();
    this.status = OrderStatus.SUBMITTED;
    return [new OrderSubmittedEvent(this.id)];
  }
}
```

ApplicationService를 제거하고 CommandHandler로 메소드를 이동

📋 얇은 조정자 (Thin Coordinator)

```
// Command Handler는 조정만
class CreateOrderCommandHandler {
  constructor(
    private orderRepo: OrderRepository,
    private couponRepo: CouponRepository,
    private inventoryService: InventoryDomainService
  ) {}

  async handle(command: CreateOrderCommand): Promise<OrderId> {
    // 1. 필요한 도메인 객체 조회
    const items = await this.inventoryService.reserveItems(command.items);

    // 2. 도메인 로직 실행
    const order = Order.create(command.customerId, items);
    if (command.couponCode) {
      const coupon = await this.couponRepo.findByCode(command.couponCode);
      order.applyCoupon(coupon); // 도메인이 검증
    }
    // 3. 저장
    await this.orderRepo.save(order);

    return order.id;
  }
}
```

근데 Aggregate가 소유하기 적합하지 않은 로직은 어떻게?

명백한 비즈니스규칙이고

aggregate에 넣기 애매하고

또 비즈니스로직이라 repository에 넣기도 애매하고

그럴때 DomainService (어렵다)

언제 로직이 DomainService에 들어가고 언제 Aggregate나 Entity에 들어가야할까?

1. 단일 Aggregate 내부 상태만 다룬다 [엔티티/밸류에 넣는다]
2. 둘 이상의 Aggregate(또는 외부 도메인 개념)를 동시에 다룬다 [Domain Service 후보]
3. 도메인-규칙이지만 상태를 갖지 않는다 (stateless). [Domain Service 적합]
4. 순수 계산·판단이 아니라 외부 I/O(레포지토리, API 호출 등)까지 필요 [Domain Service이지만 Application Service로 착각하기 쉬움]
5. “행위 주체(주어)”가 모호하다 (예: 환율 계산, 통화 변환,다수 창고 간 재고 재배치)

- 엔티티/밸류에 넣기 불가능 → Domain Service.

DomainService 예시 1: 계좌이체

두 계좌의 잔액을 변경해야 하는데,
어떻게 보면 한 Aggregate가 소유하기 적합하지 않은 로직일수도 있음

```
class AccountTransferService {
  async transfer(
    fromAccount: Account,
    toAccount: Account,
    amount: Money
  ): Promise<void> {
    // 두 계좌의 잔액을 변경
    fromAccount.withdraw(amount);
    toAccount.deposit(amount);
    return {
      fromAccount,
      toAccount,
    };
  }
}
```

DomainService 예시 2: 환율계산

행위 주체가 없는 비즈니스 로직

```
// domains/fx-conversion-service.ts
export interface FxConversionService {
  convert(src: Money, targetCurrency: string, at?: Date): Money;
}

// infra/default-fx-conversion-service.ts
export class DefaultFxConversionServiceImpl implements FxConversionService {
  constructor(private readonly rateRepo: ExchangeRateRepository) {}

  convert(src: Money, target: string, at: Date = new Date()): Money {
    if (src.currency === target) return src;

    const rate = this.rateRepo.findRate(src.currency, target, at);
    if (!rate) throw new NotFoundError("rate not found");

    return new Money(src.amount * rate.rate, target);
  }
}
```

DomainService 예시 3: 배송비 계산

외부 io가 필요한 경우

```
// domain/shipping-cost-calculator.ts
export interface ShippingCostCalculator {
  estimate(p: ShipmentParams): Money;
}

// infra/default-shipping-cost-calculator.ts
export class DefaultShippingCostCalculator implements ShippingCostCalculator {
  constructor(
    private rateRepo: CarrierRateRepository,
    private distanceSvc: DistanceService
  ) {}

  estimate(p: ShipmentParams): Money {
    const km = this.distanceSvc.distanceBetween(p.originZip, p.destZip);
    const rate = this.rateRepo.findRate(p.carrierCode, p.weightKg, km);
    if (!rate) throw new Error("rate not found");
    return new Money(p.weightKg * km * rate);
  }
}
```

필요하다면 Repository, 다른 infra service 의존가능, domain service 의존가능

가급적 entity나 aggregate만 받아서 로직을 처리해야하지만 재사용성이 필요하거나 표현이 자연스럽다면 의존해도 괜찮다.

단, 상태를 저장한다던가 사이드이펙트가 발생하면 도메인 서비스가 아니기에 주의

그래도 혼란스럽다면?

“이 메서드가 없어지면, 내 도메인 모델이 비즈니스 규칙을 잃어버리나?”

- Yes ➔ Domain Service (infra에 의존하더라도)
- No ➔ Application Service or Infrastructure Service

예시1: 주문 생성

```
application
  |-- OrderService or
  |-- PlaceOrderCommandHandler
domain
  |-- DiscountCalculatorService  <- 사라지면 도메인내 비즈니스 규칙이 의미를 잃을수 있어요.
  |-- Order
infra
  |-- EmailService
```

예시2: User 생성

```
application
  |-- CreateUserCommandHandler <-- 없어져도 도메인 모델이 의미가 사라지지 않아요.
domain
  |-- User
  |-- UniqueEmailPolicy (interface only)
infra
  |-- EmailService
```

(선택)필요하다면 domain service를 aggregate method에 주입 할 수 있다. (생성자에 주입 X)

예시1: 주문 추가 로직에서 service를 주입받아 사용하는 형태

```
// Aggregate에서 method에 Domain Service를 주입해서 사용
class Order {
  private items: OrderItem[] = [];

  // Domain Service를 매개변수로 받아 사용
  async addItem(
    product: Product,
    quantity: number,
    pricingService: PricingDomainService,
    inventoryService: InventoryDomainService
  ): Promise<void> {
    // 재고 확인 (Domain Service 사용)
    const isAvailable = await inventoryService.checkAvailability(
      product.id,
      quantity
    );

    if (!isAvailable) {
      throw new InsufficientInventoryError();
    }

    // 가격 계산 (Domain Service 사용)
    const discount = await pricingService.calculateDiscount(
      product,
      this.customer,
      quantity
    );

    const finalPrice = product.price.subtract(discount);

    const item = new OrderItem(product, quantity, finalPrice);
    this.items.push(item);
    this.recalculateTotalAmount();
  }
}
```

예시2: 최종 주문 계산 로직에서 domain service를 주입받아 사용 하는 형태

```
// 도메인 계층
export interface PromotionEngine {
  /** 주문 + 고객 정보를 주면 최종 할인 금액을 돌려준다 */
  discountFor(order: Order, customer: Customer): Money;
}

export class Order {
  private readonly lines: OrderLine[];
  private discount: Money = Money.zero();

  /** 서비스·협력 객체를 인자로 받는 것이 핵심 */
  calculateTotal(promo: PromotionEngine, customer: Customer): Money {
    const gross = this.lines
      .map((l) => l.subtotal())
      .reduce((a, b) => a.add(b), Money.zero());

    this.discount = promo.discountFor(this, customer); // ← Domain Service 호출
    return gross.subtract(this.discount);
  }

  discountAmount(): Money {
    return this.discount;
  }
}
```

예시3: 유저생성시에 이메일 중복 체크 로직에서 domain service를 주입받아 사용 하는 형태

```
export interface UniqueEmailPolicy {
  isUnique(email: Email): Promise<boolean>;
}
// 유저 생성시에 이메일 중복 체크 로직에서 domain service를 주입받아 사용 하는 형태
class User {
  private email: Email;

  private constructor(email: Email) {}

  static async create(
    email: Email,
    uniqueEmailPolicy: UniqueEmailPolicy
  ): User {
    if (!(await uniqueEmailPolicy.isUnique(email))) {
      throw new Error("이메일이 중복되었습니다");
    }
    return new User(email);
  }
}
```

예시4: 방 예약 및 가능 여부 확인

```
export interface ReservationAvailabilityPolicy {
  /** true → 아직 아무 예약과도 겹치지 않는다 */
  isAvailable(roomId: RoomId, desired: DateRange): Promise<boolean>;
}
/* reservation.ts */
export class Reservation {
  /* …생략… */

  static async create(
    roomId: RoomId,
    period: DateRange,
    policy: ReservationAvailabilityPolicy,
    id: ReservationId = uuid()
  ): Promise<Reservation> {
    if (!(await policy.isAvailable(roomId, period))) {
      throw new Error("date range overlaps with another reservation");
    }
    return new Reservation(id, roomId, period);
  }
}
```

인프라 엔티티와 도메인 엔티티 매핑

🔄 왜 수동 매핑이 필요한가?

ORM 엔티티 vs 도메인 엔티티의 차이

ORM 엔티티의 한계

📊 데이터베이스 구조에 종속된 모델

```
// Prisma/TypeORM 등 ORM 엔티티
@Entity("users")
class UserEntity {
  @PrimaryColumn()
  id: string;

  @Column()
  email: string;

  @OneToMany(() => OrderEntity, (order) => order.user)
  orders: OrderEntity[]; // 관계 매핑

  @Column()
  password_hash: string; // DB 컬럼명

  @Column()
  created_at: Date; // DB 컬럼명
}
```

🚨 그대로 쓸때 문제점

- 도메인과 인프라 관심사가 섞임
- 객체의 표현력이 (Value Object, Entity, Design Pattern) orm의 제약사항에 걸림
- 불필요한 관계 로딩 및 표현(소유와 참조가 구분되지 않을 수 있음)

도메인 엔티티의 순수성

🎯 비즈니스 로직에 집중

```
// 순수한 도메인 엔티티
class User {
  private constructor(
    private readonly id: UserId,
    private email: Email, // 풍부한 도메인 객체 표현
    private password: HashedPassword,
    private status: UserStatus,
    private registeredAt: DateTime
  ) {}

  // 비즈니스 메서드만 노출
  changeEmail(newEmail: Email): void {
    if (this.status.isDeleted()) {
      throw new DeletedUserError();
    }
    this.email = newEmail;
    this.addEvent(new EmailChangedEvent(this.id, newEmail));
  }
}
```

수동 매핑의 구현

🔧 Repository혹은 Mapper에서 변환 담당

```
class UserRepositoryImpl implements UserRepository {
  constructor(private prisma: PrismaClient) {}

  // 도메인 → DB 엔티티 매핑
  private toEntity(user: User): PrismaUserEntity {}

  // DB 엔티티 → 도메인 매핑
  private toDomain(entity: PrismaUserEntity): User {}
}
```

수동매핑 전략의 장점

도메인 개념 표현을 풍부하게 가져가고
orm의 제약사항에 묶이지 않는다.

```
class Money {
  constructor(
    private readonly amount: number,
    private readonly currency: Currency
  ) {}

  add(other: Money): Money {
    if (!this.currency.equals(other.currency)) {
      throw new CurrencyMismatchError();
    }
    return new Money(this.amount + other.amount, this.currency);
  }
}
```

세부사항 미루기 원칙

🎯 비즈니스가 먼저, 기술은 나중에

잘못된 접근: 기술부터 시작

😰 흔한 실수들

```
// ❌ 개발자의 첫 질문들
"어떤 DB를 쓸까요? PostgreSQL? MongoDB?";
"어떤 프레임워크가 좋을까요? Express? NestJS?";
"메시지 큐는 RabbitMQ? Kafka?";
"ORM은 TypeORM? Prisma?";

// 🤔 정작 중요한 질문은...
"우리 비즈니스에서 '주문'이란 정확히 무엇인가요?";
"고객이 주문을 취소할 수 있는 조건은 무엇인가요?";
"재고가 부족할 때 어떻게 처리하나요?";
```

문제점

- 비즈니스 이해 없이 기술 스택 결정
- 기술에 비즈니스를 맞추는 역전 현상
- 나중에 바꾸기 어려운 구조

올바른 접근: 비즈니스 이해관계자와의 대화

```
// 👔 비즈니스 전문가와의 대화
비즈니스: "VIP 고객은 주문 후 24시간 내 무료 취소가 가능해요";
개발자: "VIP 고객의 기준은 무엇인가요?";
비즈니스: "최근 1년간 누적 구매액이 100만원 이상인 고객이에요";
// ✅ 이 대화가 바로 도메인 모델이 됨
class Customer {
  isVIP(): boolean {
    const oneYearAgo = dateSubtract(DateTime.now(), { years: 1 });
    const recentPurchaseAmount = this.calculatePurchaseAmountSince(oneYearAgo);
    return recentPurchaseAmount.isGreaterThanOrEqual(Money.KRW(1000000));
  }
}

class Order {
  canBeCancelledBy(customer: Customer): boolean {
    if (customer.isVIP() && this.hoursElapsedSinceCreation() <= 24) {
      return true;
    }
    return this.status.isCancellable();
  }
}
```

기술 결정을 미루는 이유

비즈니스로직을 검증하는게 먼저다.

```
// 1 도메인부터 시작 (기술 독립적으로 인터페이스만 정의하고 임시 구현체로 연결)
interface OrderRepository {
  save(order: Order): Promise<void>;
  findById(id: OrderId): Promise<Order | null>;
}

interface InventoryService {
  reserve(order: Order): Promise<void>;
}
```

```
// 2 비즈니스 로직 구현 및 응용서비스 구현
class PlaceOrderUseCase {
  constructor(
    private orderRepo: OrderRepository, // 인터페이스만 의존
    private inventoryService: InventoryService
  ) {}

  async execute(command: PlaceOrderCommand): Promise<OrderId> {
    // 순수한 비즈니스 로직
    const order = Order.create(command.customerId, command.items);
    order.validate();

    await this.inventoryService.reserve(order);
    await this.orderRepo.save(order);

    return order.id;
  }
}

// 3 나중에 기술 결정하여 구현
class PostgresOrderRepository implements OrderRepository {}
class MongoOrderRepository implements OrderRepository {}
class InMemoryOrderRepository implements OrderRepository {}
```

<

인메모리 Repository 패턴

💾 영속성 없이 시작하기

```
// Kent Beck, Martin Fowler 등이 권장하는 방식
class InMemoryOrderRepository implements OrderRepository {
  private orders: Map<string, Order> = new Map();

  async save(order: Order): Promise<void> {
    this.orders.set(order.id.value, order);
  }

  async findById(id: OrderId): Promise<Order | null> {
    return this.orders.get(id.value) || null;
  }

  async findByCustomerId(customerId: CustomerId): Promise<Order[]> {
    return Array.from(this.orders.values()).filter((order) =>
      order.customerId.equals(customerId)
    );
  }
}
```

실제 사례: 유명 개발자들의 접근

Uncle Bob (Robert C. Martin)

> 18개월의 개발 기간 동안 데이터베이스를 실행하지 않았다는 사실은, 18개월 동안 스키마 문제, 쿼리 문제, 데이터베이스 서버 문제,
> 비밀번호 문제, 연결 시간 문제, 그리고 데이터베이스를 가동할 때 나타나는 모든 다른 성가신 문제들이 없었다는 것을 의미했습니다.

Kent Beck

> 작동하는 소프트웨어를 빠르게 만들기 위해 우리는 매우 간단한 인메모리 구현을 만들 수 있고, 나중에 우리가 선호하는 기술로 구현할 수 있습니다.

올바른 비즈니스 모델링의 영향

✅ 정확한 이해가 만드는 견고한 시스템

```
// ✅ 올바른 비즈니스 이해
class Order extends AggregateRoot {
  // "배송 시작 전, VIP는 24시간 내, 일반 고객은 1시간 내 취소 가능"
  requestCancellation(
    customer: Customer,
    currentTime: DateTime
  ): CancellationResult {
    if (this.status.isShipped()) {
      return CancellationResult.failed("이미 배송이 시작되었습니다");
    }

    const hoursElapsed = this.calculateHoursElapsedSince(currentTime);
    const maxCancellableHours = customer.isVIP() ? 24 : 1;

    if (hoursElapsed > maxCancellableHours) {
      return CancellationResult.failed(
        `주문 후 ${maxCancellableHours}시간이 지났습니다`
      );
    }

    this.status = OrderStatus.CANCELLATION_REQUESTED;
    this.addEvent(new OrderCancellationRequestedEvent(this.id, customer.id));

    return CancellationResult.success();
  }
}
```

여기서 얘기하기에는 긴 내용들이 있어요.

Bounded Context, Value Object, Domain Event, Domain Integration Event, Saga(Choreography, Orchestration), Shared Kernel ...등등

QNA