Glosarium Ubiquitous Language & Rancangan Model

II3160 - Teknologi Sistem Terintegrasi

Disusun Oleh:

Kenzie Raffa Ardhana - 18223127

Program Studi Sistem dan Teknologi Informasi

Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung

Jl. Ganesha 10, Bandung 40132

2025

DAFTAR ISI

DAFTAR GAMBAR

1. PENDAHULUAN

1.1 Tujuan Dokumen

Dokumen ini bertujuan untuk:

- Mendefinisikan Glosarium Ubiquitous Language yang komprehensif untuk Sistem Reservasi Hotel

- Merancang model taktis dengan detail Entities, Value Objects, dan Aggregates

- Menyajikan Class Diagram yang menggambarkan struktur dan relasi antar komponen domain

- Memberikan panduan implementasi berdasarkan prinsip Domain-Driven Design (DDD)

1.2 Ruang Lingkup

Dokumen ini fokus pada **Reservation Context** yang telah dipilih sebagai Core Domain pada Milestone 2.

Desain taktis mencakup:

- Identifikasi lengkap Entities dan Value Objects

- Definisi Aggregates dengan boundary yang jelas

- Relasi antar komponen domain

- Business rules dan invariants

1.3 Hubungan dengan Milestone Sebelumnya

Milestone 1 (Analisis Domain):

- Mengidentifikasi Reservation sebagai Core Subdomain

- Memetakan kapabilitas bisnis utama

Milestone 2 (Desain Batasan):

- Mendefinisikan Reservation Context sebagai Bounded Context
- Memilih Reservation Context sebagai konteks inti untuk desain detail
- Memetakan relasi dengan context lain

Milestone 3 (Desain Taktis) - Dokumen Ini:

- Mendetailkan model domain Reservation Context

- Merancang Entities, Value Objects, dan Aggregates

- Membuat Class Diagram lengkap

Milestone 3 ini melanjutkan hasil analisis dan desain strategis pada Milestone 1 dan 2. Pada M01 telah dipilih **Reservation & Booking Management** sebagai salah satu core subdomain, dan pada M02 konteks ini ditetapkan sebagai **bounded context inti** melalui analisis komparatif.

Oleh karena itu, M03 memfokuskan seluruh desain taktis pada **Reservation Context**, dan menjadi jembatan langsung menuju Milestone 4, di mana implementasi kode hanya mencakup aggregate dan

domain logic penting yang sudah diprioritaskan sejak M01–M02. Dengan demikian, seluruh alur dari identifikasi domain → pemilihan bounded context → desain taktis → implementasi tetap konsisten dan terstruktur.

---

2. GLOSARIUM UBIQUITOUS LANGUAGE

Ubiquitous Language adalah bahasa bersama yang digunakan oleh domain experts dan developers untuk menghindari ambiguitas dan memastikan pemahaman yang sama.

2.1 Core Business Terms

2.1.1 Reservation (Reservasi)

Definisi: Pemesanan kamar hotel oleh tamu dengan detail tanggal check-in/check-out, tipe kamar, jumlah tamu, dan harga yang telah disepakati.

Sinonim: Booking, Pemesanan

Contoh Penggunaan: - “Tamu membuat *reservation* untuk kamar Deluxe dari tanggal 1-5 Maret 2025” - “Reservation dapat di-modify hingga 24 jam sebelum check-in”

Business Rules:

- Satu reservation minimal untuk 1 malam

- Reservation memiliki unique confirmation code

- Dapat memiliki multiple special requests

2.1.2 Guest (Tamu)
Definisi: Individu atau pihak yang melakukan pemesanan dan/atau menginap di hotel.

Tipe:

- **Primary Guest:** Tamu utama yang namanya tercatat dalam reservation

- **Additional Guest:** Tamu tambahan yang menginap dalam satu kamar

- **Loyalty Member:** Tamu yang terdaftar dalam program loyalitas

Business Rules:

- Minimal ada 1 adult guest per reservation

- Guest profile disimpan untuk future reservations

2.1.3 Room (Kamar)
Definisi: Unit akomodasi fisik di hotel dengan nomor unik dan karakteristik tertentu.

Atribut:

- Room Number (Nomor Kamar): Identifier unik

- Room Type: Kategori/tipe kamar

- Floor: Lantai lokasi kamar

- Status: Status operasional kamar

2.1.4 Room Type (Tipe Kamar)
Definisi: Kategori kamar berdasarkan ukuran, fasilitas, dan harga.

Contoh:

- Standard Room: Kamar standar dengan fasilitas dasar

- Deluxe Room: Kamar dengan fasilitas lebih lengkap

- Suite: Kamar dengan ruang terpisah (bedroom dan living room)

- Family Room: Kamar besar untuk keluarga

Karakteristik:

- Max Occupancy: Kapasitas maksimal tamu

- Base Price: Harga dasar per malam
- Amenities: Fasilitas yang tersedia

2.1.5 Availability (Ketersediaan)
Definisi: Status ketersediaan kamar untuk tanggal tertentu, menunjukkan berapa kamar yang dapat dipesan.

Formula:

Available Rooms = Total Rooms - Reserved Rooms - Blocked Rooms

Status:

- Available: Ada kamar yang dapat dipesan
- Limited: Sisa kamar sedikit (< 3 kamar)
- Sold Out: Tidak ada kamar tersedia
- On Request: Perlu konfirmasi manual

2.2 Reservation Lifecycle Terms

2.2.1 Reservation Status (Status Pemesanan)

Definisi: Status saat ini dari suatu reservation dalam lifecycle-nya.

Status Values:

1. **PENDING**
   - Reservation baru dibuat, menunggu pembayaran
   - Dapat di-cancel tanpa penalty
   - Auto-cancel jika tidak dibayar dalam 24 jam

2. **CONFIRMED**
    - Pembayaran berhasil, reservation dikonfirmasi
    - Confirmation email sudah dikirim
    - Dapat di-modify dengan terms & conditions
    - Dapat di-cancel dengan penalty sesuai policy
3. **CHECKED_IN**
    - Tamu sudah check-in dan menempati kamar
    - Tidak dapat di-cancel
    - Dapat extend stay jika ada availability
4. **CHECKED_OUT**
    - Tamu sudah check-out
    - Final bill sudah di-settle
    - Status terminal (tidak dapat berubah lagi)
5. **CANCELLED**
    - Reservation dibatalkan oleh tamu atau hotel
    - Refund sudah diproses (jika applicable)
    - Status terminal
6. **NO_SHOW**
    - Tamu tidak datang tanpa pemberitahuan
    - Charged sesuai no-show policy
    - Status terminal

State Transitions:

PENDING → CONFIRMED (payment received)
PENDING → CANCELLED (cancelled before payment) CONFIRMED → CHECKED_IN (guest arrives) CONFIRMED → CANCELLED (cancelled with refund) CONFIRMED → NO_SHOW (guest didn't arrive) CHECKED_IN → CHECKED_OUT (guest departs)

2.2.2 Check-in
Definisi: Proses kedatangan tamu ke hotel, verifikasi identitas, dan menerima akses kamar.

Komponen:

- Guest verification (ID, passport)

- Room assignment (nomor kamar spesifik)

- Key card issuance

- Deposit collection (jika belum dibayar)

- Welcome amenities delivery

Standard Time: 14:00 (2 PM)

Business Rules:

- Early check-in subject to availability (fee may apply)

- Identity verification mandatory

- Deposit required untuk charges during stay

2.2.3 Check-out
Definisi: Proses keberangkatan tamu, penyelesaian bill, dan pengembalian akses kamar.

Komponen:

- Bill settlement (review charges)
- Payment for extras (minibar, room service)
- Key card return
- Deposit refund
- Express check-out option available

Standard Time: 12:00 (12 PM)

Business Rules:

- Late check-out subject to availability (fee may apply)
- All charges must be settled
- Room inspection before final settlement

2.3 Pricing Terms

2.3.1 Rate (Tarif)

Definisi: Harga per malam untuk tipe kamar tertentu pada tanggal tertentu.

Komponen:

- Base Rate: Harga dasar
- Seasonal Adjustment: Penyesuaian berdasarkan musim
- Dynamic Pricing: Penyesuaian berdasarkan demand
- Final Rate: Harga final yang ditampilkan ke guest

2.3.2 Total Amount (Total Harga)

Definisi: Total biaya yang harus dibayar oleh guest untuk seluruh periode menginap.

Perhitungan:

Total Amount = (Rate per Night × Number of Nights) + Taxes + Service Charge

Breakdown:

- Room Charge: Biaya kamar
- Tax (PB1): Pajak 10%
- Service Charge: Biaya layanan 10%
- Additional Charges: Biaya tambahan (jika ada)

2.3.3 Discount (Diskon)

Definisi: Pengurangan harga berdasarkan promo, membership, atau strategi pricing.

Tipe:

- Early Bird: Reservation jauh hari (min 30 hari)
- Last Minute: Reservation mendadak (< 3 hari)
- Loyalty Discount: Untuk member setia
- Promotional Code: Kode promo khusus
- Corporate Rate: Tarif korporat

Business Rules:

- Satu reservation hanya dapat menggunakan satu discount
- Beberapa discount tidak dapat digabung
- Discount memiliki validity period

2.3.4 Refund (Pengembalian Dana)

Definisi: Pengembalian sebagian atau seluruh pembayaran kepada guest saat cancellation.

Formula:

Refund Amount = Total Paid - Cancellation Penalty

Faktor yang Mempengaruhi:

- Cancellation Policy

- Waktu cancellation (berapa hari sebelum check-in)

- Special rate terms (non-refundable rates)

2.4 Policy Terms

2.4.1 Cancellation Policy (Kebijakan Pembatalan)

Definisi: Aturan yang mengatur pembatalan reservation dan perhitungan refund.

Policy Types:

7. **FLEXIBLE**
    - Free cancellation hingga 24 jam sebelum check-in
    - 100% refund jika dibatalkan dalam periode free cancellation
    - 0% refund jika dibatalkan < 24 jam
8. **MODERATE**
    - Free cancellation hingga 7 hari sebelum check-in
    - 50% refund jika dibatalkan 3-7 hari sebelum
    - 0% refund jika dibatalkan < 3 hari

9. **STRICT**
    - Free cancellation hingga 14 hari sebelum check-in
    - 25% refund jika dibatalkan 7-14 hari sebelum
    - 0% refund jika dibatalkan < 7 hari
10. **NON_REFUNDABLE**
    - Tidak dapat dibatalkan
    - 0% refund dalam kondisi apapun
    - Biasanya harga paling murah

2.4.2 No-Show Policy

Definisi: Kebijakan yang diterapkan ketika guest tidak datang tanpa pemberitahuan.

Charges:

- 100% dari first night charge

- Atau sesuai cancellation penalty (yang lebih tinggi)

Grace Period: 2 jam setelah check-in time untuk menghubungi hotel

2.4.3 Minimum Stay

Definisi: Durasi minimum menginap yang required untuk periode tertentu.

Contoh:

- Weekend: Min 2 nights (Friday-Saturday)

- Peak Season: Min 3 nights

- Special Events: Min 4-7 nights

- Regular Days: Min 1 night

2.4.4 Overbooking Policy

Definisi: Strategi menerima reservasi melebihi kapasitas untuk mengantisipasi no-show dan cancellation.

Parameters:

- Overbooking Limit: Max 10% dari total rooms

- Historical No-Show Rate: Average 3-5%

- Risk Threshold: High-demand periods only

Procedures:

- Relocation ke hotel lain (jika terjadi actual overbooking)

- Compensation: Upgrade atau discount untuk next stay

- Priority: Relocate non-loyalty, OTA reservations first

2.5 Guest Request Terms

2.5.1 Special Request (Permintaan Khusus)

Definisi: Permintaan tambahan dari guest terkait kamar atau layanan.

Categories:

1. **Room Preferences:**
   - High Floor: Lantai tinggi
   - Low Floor: Lantai rendah
   - Quiet Room: Jauh dari lift/tangga
   - Corner Room: Kamar pojok
   - Adjoining Rooms: Kamar bersebelahan
2. **Bed Configuration:**
   - King Bed: 1 kasur besar
   - Twin Beds: 2 kasur terpisah
   - Extra Bed: Kasur tambahan
   - Crib/Baby Cot: Kasur bayi
3. **Check-in/out:**
   - Early Check-in: Sebelum jam 14:00

- Late Check-out: Setelah jam 12:00

4. **Special Occasions:**
   - Birthday Celebration: Dekorasi ulang tahun
   - Anniversary: Honeymoon setup
   - Romantic Package: Setup romantis
5. **Accessibility:**
   - Wheelchair Accessible: Akses kursi roda
   - Hearing Impaired: Visual alarm
   - Walk-in Shower: Shower tanpa bathtub

Fulfillment:

- Subject to Availability

- Some requests may incur additional charges

- Guaranteed vs Best Effort basis

2.5.2 Waitlist Entry (Entri Daftar Tunggu)
Definisi: Pendaftaran guest pada waiting list untuk kamar yang fully booked.

Attributes:

- Priority: NORMAL, HIGH, VIP

- Requested Dates: Tanggal yang diinginkan

- Expiry: Automatically expires after 14 days

- Status: ACTIVE, CONVERTED, EXPIRED

Priority Factors:

- Loyalty Tier: Platinum > Gold > Silver > Regular

- Previous Stays: Frequent guests higher priority

- ReservationValue: Higher total amount = higher priority

- Request Date: First come, first served (within same tier)

Notification:

- Auto-notify when room becomes available

- 48-hour window to confirm reservation

- Convert to reservation if accepted

2.6 Operational Terms

2.6.1 Room Status (Status Kamar)

Definisi: Status operasional fisik dari kamar.

Status Values:

1. **AVAILABLE**
   - Kamar bersih dan siap ditempati
   - Inspeksi sudah dilakukan
   - Can be assigned to check-in guest
2. **OCCUPIED**

- Kamar sedang ditempati oleh guest
- Do not disturb unless requested
- Housekeeping can service if allowed

3. **CLEANING**
   - Sedang dibersihkan oleh housekeeping
   - Not ready for check-in
   - Typical duration: 30-45 minutes
4. **DIRTY**
   - Guest sudah check-out, perlu dibersihkan
   - High priority untuk cleaning
   - Cannot be assigned
5. **MAINTENANCE**
   - Sedang dalam perbaikan
   - Blocked dari reservation system

- Repair work in progress

6. **OUT_OF_ORDER**
   - Tidak dapat digunakan (major issue)
   - Blocked untuk extended period
   - Requires management approval untuk reactivate
7. **BLOCKED**
   - Di-block untuk alasan tertentu
   - Tidak tersedia untuk reservation
   - Reasons: VIP, renovation, inspection

2.6.2 Confirmation Code

Definisi: Kode unik yang digenerate untuk setiap reservation sebagai identifier.

Format: PREFIX + DATE + RANDOM

- Example: RES-20250301-A7B9C2

- PREFIX: “RES”
untuk Reservation
- DATE: Tanggal pembuatan (YYYYMMDD)
- RANDOM: 6 karakter alphanumeric

Usage:

- Guest reference untuk modifikasi

- Check-in identification

- Customer service inquiry

- Refund processing

2.6.3 Room Assignment

Definisi: Proses assign nomor kamar spesifik ke reservation.

Timing:

- Pre-Assignment: Beberapa hari sebelum check-in

- At Check-in: Saat guest tiba

- Auto-Assignment: Berdasarkan algorithm

Factors:

- Guest preferences (special requests)

- Loyalty tier (better rooms untuk VIP)

- Room status availability

- Previous stay history (same room if liked)

- Operational efficiency (minimize walking)

2.7 Date and Time Terms

2.7.1 Date Range (Rentang Tanggal)

Definisi: Periode menginap dari check-in hingga check-out date.

Components:

- Check-in Date: Tanggal tiba

- Check-out Date: Tanggal berangkat

- Number of Nights: Duration dalam malam

Validation:

- Check-out date must be after check-in date

- Minimum: 1 night

- Maximum: 30 nights (system limit)

Business Logic:

Number of Nights = Check-out Date

- Check-in Date Example: Check-in Mar 1, Check-out Mar 5 = 4 nights

2.7.2 Reservation Source (Sumber Pemesanan)
Definisi: Channel atau sumber dari mana reservation dibuat.

Channels:

11. **Direct Reservations:**
    - WEBSITE: Hotel website langsung
    - MOBILE_APP: Mobile application hotel

- PHONE: Telephone reservation
- WALK_IN: Langsung di front desk

12. **Online Travel Agencies (OTA):**
    - OTA_RESERVATION_COM: Reservation.com
    - OTA_AGODA: Agoda
    - OTA_TRAVELOKA: Traveloka
    - OTA_EXPEDIA: Expedia
13. **Other Sources:**
    - CORPORATE: Corporate contract reservation
    - TRAVEL_AGENT: Melalui travel agency
    - GDS: Global Distribution System

Impact:

- Commission rates (OTA: 15-25%, Direct: 0%)

- Modification policies (lebih flexible untuk direct)

- Customer data access

- Marketing attribution

2.8 Value Objects Terminology

2.8.1 Money (Uang)

Definisi: Representasi nilai moneter dengan currency.

Attributes:

- Amount: Nilai numerik (Decimal untuk precision)

- Currency: ISO code (IDR, USD, EUR)

Operations:

- Addition, Subtraction (same currency only)

- Multiplication by scalar

- Percentage calculation

- Currency conversion (dengan exchange rate)

Validation:

- Amount cannot be negative untuk prices

- Currency must be supported

- Precision: 2 decimal places

2.8.2 Guest Count (Jumlah Tamu)

Definisi: Jumlah tamu yang akan menginap.

Components:

- Adults: Jumlah tamu dewasa (18 tahun)

- Children: Jumlah anak (0-17 tahun)

- Infants: Bayi (0-2 tahun, biasanya free)

Validation:

- Minimum 1 adult required

- Total tidak exceed room capacity

- Children count affects bed requirements

2.9 Domain Events Terminology

2.9.1 Reservation Created

Event: Fired when new reservation successfully created

Payload:

- Reservation ID
- Guest ID
- Room Type
- Date Range
- Total Amount
- Timestamp

Subscribers:

- Inventory Context: Update availability
- Pricing Context: Record reservation pattern
- Notification Service: Send confirmation email

2.9.2 Reservation Modified

Event: Fired when reservation details changed

Payload:

- Reservation ID
- Changed Fields (dates, room type, guests)
- Old Values
- New Values
- Modified By
- Timestamp

Subscribers:

- Inventory Context: Adjust availability
- Pricing Context: Recalculate amount
- Notification Service: Send modification notification

2.9.3 Reservation Cancelled

Event: Fired when reservation is cancelled

Payload:

- Reservation ID
- Cancellation Reason
- Refund Amount
- Cancelled By
- Timestamp

Subscribers:

- Inventory Context: Release rooms
- Payment Context: Process refund
- Guest Context: Update history
- Notification Service: Send cancellation confirmation

2.9.4 Guest Checked In

Event: Fired when guest completes check-in

Payload:

- Reservation ID
- Guest ID
- Room Number Assigned
- Actual Check-in Time
- Times-tamp

Subscribers:

- Operations Context: Update room status to OCCUPIED

- Guest Context: Start active stay tracking

- Notification Service: Send welcome message

2.9.5 Guest Checked Out

Event: Fired when guest completes check-out

Payload:

- Reservation ID

- Guest ID

- Room Number

- Actual Check-out Time

- Final Bill Amount

- Timestamp

Subscribers:

- Operations Context: Mark room as DIRTY

- Guest Context: Update stay history, award points

- Notification Service: Send thank you & feedback request

2.10 Repository Patterns Terminology

2.10.1 Repository

Definisi: Abstraction untuk data access, menyediakan collection

- like interface untuk aggregates.

Responsibilities:

- Persist aggregates ke database

- Retrieve aggregates by ID

- Query aggregates dengan specifications

- Manage transactions

Pattern:

```
# Interface
class ReservationRepository:
    def save(reservation: Reservation) -> None
    def find_by_id(reservation_id: str) -> Reservation
    def find_by_confirmation_code(code: str) -> Reservation
    def delete(reservation_id: str) -> None
```

2.10.2 Unit of Work

Definisi: Pattern untuk manage transactions dan track changes.

Responsibilities:

- Track objects changed during business transaction
- Coordinate writing out changes
- Manage commit/rollback

Usage:

```
with unit_of_work:
    reservation = repository.find_by_id(id)
    reservation.confirm()
    repository.save(reservation)
    unit_of_work.commit()
```

---

3. MODEL TAKTIS - RESERVATION CONTEXT

3.1 Aggregates

Aggregates adalah cluster of entities dan value objects yang memiliki consistency boundary. Setiap aggregate memiliki satu root entity yang menjadi entry point untuk akses ke aggregate.

3.1.1 Reservation Aggregate

Root Entity: Reservation

Bounded Components:

- Reservation (Entity - Root)
- DateRange (Value Object)
- GuestCount (Value Object)
- Money (Value Object)
- ReservationStatus (Value Object/Enum)
- Cancellation
- Policy (Value Object)
- ReservationSource (Enum)
- SpecialRequest (Entity - child)

Aggregate Boundary Rationale:

- Semua komponen ini harus consistent dalam single transaction
- Perubahan dates mempengaruhi pricing dan availability
- Business invariants harus protected atomically
- Lifecycle tightly coupled

Consistency Rules (Invariants):

1. **Date Validity**

   check_out_date > check_in_date
   check_in_date >= today
   nights >= 1
   nights <= 30 (max stay)

2. **Guest Count Capacity**

   adults >= 1
   children >= 0
   total_guests <= room_type.max_capacity

3. **Status Transitions**

   Valid state machine transitions only
   Cannot modify after CHECK_IN (except extend) Terminal
   states: CHECKED_OUT, CANCELLED, NO_SHOW

4. **Amount Validity**

total_amount > 0
total_amount = (rate × nights) + taxes + fees
refund_amount <= total_paid_amount

5. Modification Rules

Can modify only if status in [PENDING, CONFIRMED]
Cannot modify if check_in_date < today + 1 day
Modification triggers repricing

Entity Details:

Reservation Entity (Root)

Identity: reservation_id (UUID)

Attributes:

```
class Reservation:
    # Identity
    reservation_id:  UUID
    confirmation_code: str

    # References to other contexts
    guest_id:  UUID
    # Reference to Guest Context
    room_type_id: str
    # Reference to Inventory Context

    # Value Objects
    date_range: DateRange
    guest_count: GuestCount
    total_amount: Money
    cancellation_policy: CancellationPolicy

    # Enums/Status
    status: ReservationStatus
    reservation_source:
    ReservationSource

    # Collections (child entities)
    special_requests: List[SpecialRequest]

    # Metadata created_at:
    datetime modified_at:
    datetime created_by:
    str
    version:  int
    # For optimistic locking
```

Key Methods:

```
# Creation

def create(
    guest_id: UUID,
    room_type_id: str, date_range:
    DateRange, guest_count:
    GuestCount, total_amount:
    Money,
    cancellation_policy: CancellationPolicy,
```

```
        reservation_source: ReservationSource
    ) -> Reservation:
        """Create new reservation with validation"""
        # Validate business rules
        # Generate confirmation code
        # Set status to PENDING
        # Publish ReservationCreated event

    # Modification
    def modify(
        new_date_range: Optional[DateRange],
        new_guest_count: Optional[GuestCount],
        new_room_type_id: Optional[str]
    ) -> None:
        """Modify reservation details"""
        # Check if modifiable
        # Validate new values
        # Recalculate amount
        # Update modified_at
        #  Publish  ReservationModified  event

    def add_special_request(
        request_type: RequestType,
        description: str
    ) -> None:
        """Add special request from guest"""
        # Validate request type
        # Add to collection
        # Publish event if needed

    # State Transitions
    def confirm(payment_confirmed: bool) -> None:
        """Confirm reservation after payment"""
        # Validate can transition to CONFIRMED
        # Change status
        # Publish ReservationConfirmed event

    def check_in(room_number: str) -> None:
        """Mark guest as checked in"""
        # Validate can check in
        # Change status to CHECKED_IN
        # Record actual check-in time
        # Publish GuestCheckedIn event

    def check_out() -> Money: """Process
        guest check-out""" # Calculate
        final bill
        # Change status to CHECKED_OUT
```

```
        # Publish GuestCheckedOut event
        # Return final amount

def cancel(reason: str) -> Money:
        """Cancel reservation"""
        # Validate can cancel
        # Calculate refund using policy
        # Change status to CANCELLED
        # Publish ReservationCancelled event
        # Return refund amount

def mark_no_show() -> None:
        """Mark guest as no-show"""
        # Change status to
        NO_SHOW
        # Publish NoShowRecorded event

# Queries
def is_modifiable(self) -> bool:
        """Check if reservation can be modified"""
        return (
                self.status in [ReservationStatus.PENDING, ReservationStatus.CONFIRMED]
                and  self.date_range.start  >=  date.today()  +  timedelta(days=1)
        )

def is_cancellable(self) -> bool:
        """Check if reservation can be cancelled"""
        return self.status in [
                ReservationStatus.PENDING,
                ReservationStatus.CONFIRMED
        ]

def calculate_refund(self, cancellation_date: date) -> Money:
        """Calculate refund amount based on policy"""
        return self.cancellation_policy.calculate_refund(
                self.total_amount,
                cancellation_date, self.date_range.start
        )

def get_nights(self) -> int:
        """Get number of nights"""
        return self.date_range.nights()
```

SpecialRequest Entity (Child)

Identity: request_id (UUID)

Attributes:

```
class SpecialRequest:
```

request_id: UUID
request_type: RequestType
 # Enum
description: str
fulfilled: bool notes:
Optional[str]
created_at: datetime

Methods:

def fulfill(self, notes: str) -> None: *"""Mark request as fulfilled"""* self.fulfilled = True
self.notes = notes

def is_fulfilled(self) -> bool:
return self.fulfilled

3.1.2 Availability Aggregate

Root Entity: Availability

Bounded Components:

- Availability (Entity - Root only, denormalized)

Aggregate Boundary Rationale:

- Independent lifecycle dari Reservation

- Eventual consistency acceptable

- Optimized untuk high-frequency queries

- Separate locking granularity

Consistency Rules (Invariants):

total_rooms >= 0
reserved_rooms >= 0
blocked_rooms >= 0
reserved_rooms + blocked_rooms <= total_rooms + overbooking_threshold available_rooms = total_rooms - reserved_rooms - blocked_rooms

Entity Details:

Availability Entity (Root)

Composite Identity: (room_type_id, date)

Attributes:

class Availability:
 # Composite Identity
room_type_id: str date:
date

 # Capacity Tracking
total_rooms: int
reserved_rooms: int

blocked_rooms: int

 # Configuration
overbooking_threshold: int

```
    # Metadata last_updated:
    datetime version: int
```

Computed Properties:

```
@property
def available_rooms(self) -> int:
    """Calculate available rooms"""
    return self.total_rooms - self.reserved_rooms - self.blocked_rooms

@property
def is_fully_booked(self) -> bool: """Check if
    no rooms available""" return
    self.available_rooms <= 0

@property
def can_overbook(self) -> bool:
    """Check if can accept overbooking""" current_overbooking = max(0,
    -self.available_rooms) return current_overbooking <
    self.overbooking_threshold
```

Key Methods:

```
def check_availability(self, count: int) -> bool: """Check if can
    accommodate count rooms""" return self.available_rooms
    >= count or (
        self.available_rooms + self.overbooking_threshold >= count
    )

def reserve_rooms(self, count: int) -> None: """Reserve
    rooms (decrease availability)""" # Validate capacity
    if not self.check_availability(count):
        raise InsufficientAvailabilityError()

    self.reserved_rooms += count
    self.last_updated = datetime.now() #
    Publish AvailabilityChanged event

def release_rooms(self, count: int) -> None: """Release
    rooms (increase availability)""" # Validate count
    if count > self.reserved_rooms:
        raise ValueError("Cannot release more than reserved")

    self.reserved_rooms -= count
    self.last_updated = datetime.now()
    # Publish AvailabilityChanged event
```

```
    def block_rooms(self, count: int, reason: str) -> None:
        """Block rooms for maintenance/events"""
        # Validate capacity
        self.blocked_rooms += count
        self.last_updated = datetime.now()
        # Publish RoomsBlocked event

    def unblock_rooms(self, count: int) -> None:
        """Unblock rooms after maintenance"""
        if count > self.blocked_rooms:
            raise ValueError("Cannot unblock more than blocked")

        self.blocked_rooms -= count
        self.last_updated = datetime.now()
        # Publish RoomsUnblocked event
```

3.1.3 Waitlist Aggregate

Root Entity: WaitlistEntry

Bounded Components:

- WaitlistEntry (Entity - Root only)

Aggregate Boundary Rationale:

- Independent lifecycle from Reservation
- Can exist without Reservation
- Different business rules dan policies
- Lower priority processing

Consistency Rules (Invariants):

expires_at > created_at
Only ACTIVE entries can be converted
Priority must be valid enum value
guest_count.adults >= 1

Entity Details:

WaitlistEntry Entity (Root)

Identity: waitlist_id (UUID)

Attributes:

```
class WaitlistEntry:
# Identity waitlist_id:
        UUID

        #  Request  Details guest_id:
        UUID room_type_id: str
        requested_dates: DateRange
        guest_count: GuestCount

        # Status & Priority
        priority:  Priority
        status:  WaitlistStatus
```

```
    # Timestamps
    created_at: datetime
    expires_at:  datetime
    notified_at:  Optional[datetime]

    # Conversion
    converted_reservation_id: Optional[UUID]
```

Key Methods:

```
@staticmethod
def add_to_waitlist( guest_id:
    UUID, room_type_id: str,
    requested_dates: DateRange,
    guest_count: GuestCount,
    priority:  Priority

) -> WaitlistEntry:
    """Add new entry to waitlist"""
    # Create entry
    # Set expires_at = created_at + 14 days
    # Publish WaitlistEntryCreated event

def convert_to_reservation(self, reservation_id: UUID) -> None:
    """Convert waitlist entry to actual reservation"""
    # Validate status is ACTIVE
    # Change status to
    CONVERTED
    # Record reservation_id
    #  Publish  WaitlistConverted  event

def expire(self) -> None:
    """Mark entry as expired"""
    if    self.status    ==    WaitlistStatus.ACTIVE:
        self.status = WaitlistStatus.EXPIRED
    # Publish WaitlistExpired event

def extend_expiry(self, additional_days: int) -> None:
    """Extend expiry date"""
    if self.status == WaitlistStatus.ACTIVE: self.expires_at +=
        timedelta(days=additional_days)

def upgrade_priority(self, new_priority: Priority) -> None:
    """Upgrade to higher priority"""
    if new_priority.value > self.priority.value: self.priority =
        new_priority
        # Publish PriorityUpgraded event

def mark_notified(self) -> None:
    """Record notification sent"""
```

```
        self.notified_at = datetime.now()

    def should_notify_again(self) -> bool:
        """Check if time for reminder""" if not
        self.notified_at:
            return True
        days_since_notification = (datetime.now() - self.notified_at).days
        return days_since_notification >= 3             # Remind every 3 days

    def calculate_priority_score(self) -> int:
    """Calculate priority score for ordering"""
    # Base score from priority enum
        score = self.priority.value * 100

        # Earlier request = higher score
        days_waiting = (datetime.now() - self.created_at).days score +=
        days_waiting * 2

        # Closer to expiry = higher urgency
        days_to_expiry = (self.expires_at - datetime.now()).days score += (14
        - days_to_expiry) * 5

        return score
```

3.2 Entities

Entities adalah objects yang memiliki identity yang distinct, berbeda dari attributes-nya.

3.2.1 Reservation Entity

Characteristics:

- Has unique identity (reservation_id)
- Identity remains same even if attributes change
- Mutable attributes can change over time
- Has lifecycle (created → confirmed → checked in → checked out)

3.2.2 SpecialRequest Entity

Characteristics:

- Child entity within Reservation aggregate
- Identity meaningful only within parent context
- Cannot exist without parent Reservation
- Simpler lifecycle than root entity

3.3 Value Objects

Value Objects adalah objects yang tidak memiliki identity, didefinisikan hanya oleh attributes-nya.

3.3.1 DateRange Value Object

Purpose: Encapsulate check-in to check-out period dengan validation

Attributes:

@dataclass(frozen=True)
class DateRange

```
    start:  date        # Check-in date
    end: date           # Check-out date
```

Characteristics:

- Immutable (frozen=True)

- Self-validating

- Equality based on values

- No identity

Validation Rules:

```
def _post_init_(self):
    if self.end <= self.start:
        raise ValueError("Check-out must be after check-in")

    if self.start < date.today():
        raise ValueError("Check-in cannot be in the past")

    if self.nights() > 30:
        raise ValueError("Maximum stay is 30 nights")
```

Methods:

```
def nights(self) -> int:
    """Calculate number of nights"""
    return (self.end - self.start).days

def overlaps(self, other: DateRange) -> bool:
    """Check if overlaps with another range"""
    return not (self.end <= other.start or self.start >= other.end)

def contains(self, date: date) -> bool:
"""Check if date within range""" return self.start
    <= date < self.end

def extend(self, days: int) -> DateRange:
    """Create new range extended by days"""
    return DateRange(self.start, self.end + timedelta(days=days))

def shift(self, days: int) -> DateRange:
"""Create new range shifted by days""" return
    DateRange(
        self.start + timedelta(days=days), self.end
        + timedelta(days=days)
    )

def is_weekend(self) -> bool:
"""Check if includes weekend""" current =
    self.start
    while current < self.end:
        if current.weekday() in [4, 5]:
            return True
        current += timedelta(days=1)
```

```
        return False
```

3.3.2 Money Value Object

Purpose: Represent monetary amounts dengan currency

Attributes:

```
@dataclass(frozen=True)
class  Money:
        amount: Decimal
        currency:  str
```

Validation:

```
def _post_init_(self):
        if self.amount < 0:
                raise ValueError("Amount cannot be negative")

        if  self.currency  not  in  SUPPORTED_CURRENCIES:
                raise ValueError(f"Unsupported currency: {self.currency}")

        # Ensure 2 decimal places
        object._setattr_( self,
                'amount',
                self.amount.quantize(Decimal('0.01'))
        )
```

Methods:

```
def add(self, other: Money) -> Money:

"""Add money (same currency only)"""
        self._assert_same_currency(other)
        return Money(self.amount + other.amount, self.currency)

def subtract(self, other: Money) -> Money:
        """Subtract money"""
        self._assert_same_currency(other)
        return Money(self.amount - other.amount, self.currency)

def multiply(self, factor: Decimal) -> Money:
        """Multiply by scalar"""
        return Money(self.amount * factor, self.currency)

def percentage(self, percent: Decimal) -> Money:
        """Calculate percentage"""
        return Money(
                self.amount * (percent / Decimal('100')), self.currency
        )

def split(self, parts: int) -> List[Money]:
```

```
        """Split into equal parts"""
        amount_per_part = self.amount / parts
        remainder = self.amount - (amount_per_part * parts)

        result = [Money(amount_per_part, self.currency) for _ in range(parts)]

        # Add remainder to first part
        if remainder > 0:
            result[0] = Money(amount_per_part + remainder, self.currency)

        return result

    def _assert_same_currency(self, other: Money) -> None:
        if self.currency != other.currency:
            raise ValueError(
                f"Cannot operate on different currencies: "
                f"{self.currency} and {other.currency}"
            )

    def __str__(self) -> str:
        return f"{self.currency} {self.amount:,.2f}"
```

3.3.3 ReservationStatus Value Object

Purpose: Represent lifecycle state dengan valid transitions

Implementation:

```
class ReservationStatus(Enum):
    PENDING = "pending"
    CONFIRMED = "confirmed"
    CHECKED_IN =
    "checked_in"
    CHECKED_OUT =
    "checked_out" CANCELLED
    = "cancelled" NO_SHOW =
    "no_show"

    def can_transition_to(self, target: ReservationStatus) -> bool: """Check if
        can transition to target status""" valid_transitions = {
            ReservationStatus.PENDING: [
                ReservationStatus.CONFIRME
                D,
                ReservationStatus.CANCELLE
                D
            ],
            ReservationStatus.CONFIRMED: [
                ReservationStatus.CHECKED_IN,
                ReservationStatus.CANCELLED,
                ReservationStatus.NO_SHOW
            ],
            ReservationStatus.CHECKED_IN: [
                ReservationStatus.CHECKED_OUT
```

```
            ],
            # Terminal states - no transitions
            ReservationStatus.CHECKED_OUT: [],
            ReservationStatus.CANCELLED: [],
            ReservationStatus.NO_SHOW: []
        }

        return target in valid_transitions.get(self, [])

    def is_terminal(self) -> bool: """Check if
        terminal state""" return self in [
            ReservationStatus.CHECKED_OUT,
            ReservationStatus.CANCELLED,
            ReservationStatus.NO_SHOW
        ]

    def is_active(self) -> bool: """Check if
        actively in use""" return self in [
            ReservationStatus.CONFIRMED,
            ReservationStatus.CHECKED_IN
        ]
```

3.3.4 CancellationPolicy Value Object

Purpose: Encapsulate refund calculation logic

Attributes:

```
@dataclass(frozen=True)
class CancellationPolicy: policy_type:
    PolicyType
    free_cancellation_hours: int
    penalty_percentage: Decimal

class PolicyType(Enum):
    FLEXIBLE = "flexible"
    MODERATE = "moderate"
    STRICT = "strict"
    NON_REFUNDABLE = "non_refundable"
```

Factory Methods:

```
@staticmethod
def flexible() -> CancellationPolicy:
    """Create flexible policy""" return
    CancellationPolicy(
        policy_type=PolicyType.FLEXIBLE,
        free_cancellation_hours=24,
        penalty_percentage=Decimal('0')
```

```
        )

    @staticmethod
    def moderate() -> CancellationPolicy:
        """Create moderate policy""" return
        CancellationPolicy(
            policy_type=PolicyType.MODERATE,
            free_cancellation_hours=168,            # 7
            days penalty_percentage=Decimal('50')
        )

    @staticmethod
    def strict() -> CancellationPolicy:
        """Create strict policy""" return
        CancellationPolicy(
            policy_type=PolicyType.STRICT,
            free_cancellation_hours=336,            # 14
            days penalty_percentage=Decimal('75')
        )

    @staticmethod
    def non_refundable() -> CancellationPolicy:
        """Create non-refundable policy""" return
        CancellationPolicy(
            policy_type=PolicyType.NON_REFUNDABLE,
            free_cancellation_hours=0,
            penalty_percentage=Decimal('100')
        )
```

Methods:

```
    def calculate_refund( self,
        total_amount: Money,
        cancellation_date: datetime,
        check_in_date: date
    ) -> Money:
        """Calculate refund amount"""
        # Calculate hours until check-in
        check_in_datetime = datetime.combine(
            check_in_date,
            time(14,  0)        # Standard check-in time
        )
        hours_until_checkin = (
            check_in_datetime - cancellation_date
        ).total_seconds() / 3600

        # Free cancellation period
        if hours_until_checkin >= self.free_cancellation_hours:
```

```
            return  total_amount        # 100% refund

        # Calculate penalty
        penalty = total_amount.percentage(self.penalty_percentage) refund =
        total_amount.subtract(penalty)

        # Ensure non-negative
        if refund.amount < 0:
            return Money(Decimal('0'), total_amount.currency)

        return refund

    def describe(self) -> str:
        """Human-readable   description"""
        if  self.policy_type  ==  PolicyType.NON_REFUNDABLE:
            return "Non-refundable - no cancellation allowed"

        days  =  self.free_cancellation_hours  /  24
        if days >= 1:
            return (
                f"Free cancellation up to {int(days)} days before check-in. " f"{self.penalty_percentage}% penalty
                after deadline."
            )
        else:
            return (
                f"Free cancellation up to {self.free_cancellation_hours} hours " f"before check-in.
                {self.penalty_percentage}% penalty after deadline."
            )
```

3.3.5 GuestCount Value Object

Purpose: Encapsulate guest count dengan validation

Attributes:

```
@dataclass(frozen=True)
class GuestCount:
    adults: int
    children: int
```

Validation:

```
def _post_init_(self):
    if self.adults < 1:
        raise ValueError("At least 1 adult required")

    if self.children < 0:
        raise ValueError("Children count cannot be negative")
```

Methods:

```
def total(self) -> int:
```

```
        """Total guest count"""
        return self.adults + self.children

    def requires_extra_bed(self) -> bool:
        """Check if extra bed needed"""
        # Standard room: 1 bed for 2 adults
        # Extra bed needed if > 2 adults or children present
        return self.adults > 2 or self.children > 0

    def fits_in_room_capacity(self, max_capacity: int) -> bool:
        """Check if fits in room"""
        return self.total() <= max_capacity

    def __str__(self) -> str:
        parts = [f"{self.adults} adult(s)"]
        if self.children > 0: parts.append(f"{self.children}
              child(ren)")
        return ", ".join(parts)
```

3.4 Enumerations

3.4.1 ReservationSource Enum

```
class
    ReservationSource(Enu
    m): # Direct Channels
    WEBSITE = "website"
    MOBILE_APP =
    "mobile_app" PHONE =
    "phone"
    WALK_IN = "walk_in"

    # OTAs
    OTA_RESERVATION_COM =
    "ota_reservation_com" OTA_AGODA =
    "ota_agoda" OTA_TRAVELOKA =
    "ota_traveloka" OTA_EXPEDIA =
    "ota_expedia"

    # Other
    CORPORATE = "corporate"
    TRAVEL_AGENT =
    "travel_agent" GDS = "gds"

    def is_direct(self) -> bool:
        """Check if direct reservation channel"""
        return self in [
            ReservationSource.WEBSITE,
            ReservationSource.MOBILE_APP,
            ReservationSource.PHONE,
            ReservationSource.WALK_IN
        ]

    def is_ota(self) -> bool:
        """Check if OTA channel"""
        return self.value.startswith("ota_")

    def commission_rate(self) -> Decimal: """Get
        commission rate for channel"""
        commission_rates = {
            # Direct channels - no commission
            ReservationSource.WEBSITE: Decimal('0'),
            ReservationSource.MOBILE_APP: Decimal('0'),
            ReservationSource.PHONE: Decimal('0'),
            ReservationSource.WALK_IN: Decimal('0'),

            # OTAs - typical commission rates
            ReservationSource.OTA_RESERVATION_COM:
            Decimal('18'), ReservationSource.OTA_AGODA:
            Decimal('15'), ReservationSource.OTA_TRAVELOKA:
            Decimal('20'), ReservationSource.OTA_EXPEDIA:
            Decimal('18'),
```

```
            # Other channels
            ReservationSource.CORPORATE: Decimal('5'),
            ReservationSource.TRAVEL_AGENT: Decimal('10'),
            ReservationSource.GDS: Decimal('12')
        }
        return commission_rates.get(self, Decimal('0'))
```

3.4.2 RequestType Enum

```
class RequestType(Enum):
    # Check-in/out
    EARLY_CHECKIN = "early_checkin"
    LATE_CHECKOUT = "late_checkout"

    # Bed Configuration
    EXTRA_BED =
    "extra_bed" CRIB =
    "crib"
    TWIN_BEDS = "twin_beds"

    # Room Preferences
    HIGH_FLOOR = "high_floor"
    LOW_FLOOR = "low_floor"
    QUIET_ROOM =
    "quiet_room"
    CORNER_ROOM =
    "corner_room"

    # Special Occasions
    BIRTHDAY = "birthday"
    ANNIVERSARY =
    "anniversary" HONEYMOON
    = "honeymoon"

    #  Accessibility
    WHEELCHAIR_ACCESSIBLE = "wheelchair_accessible"
    HEARING_IMPAIRED = "hearing_impaired"

    # Others
    DIETARY = "dietary"
    LATE_ARRIVAL =
    "late_arrival"

    def has_extra_charge(self) -> bool:
        """Check if request typically has extra charge"""
        charged_requests = [
            RequestType.EARLY_CHECKIN,
            RequestType.LATE_CHECKOUT,
            RequestType.EXTRA_BED
        ]
        return self in charged_requests
```

3.4.3 Priority Enum

```
class Priority(Enum):
    NORMAL = 1
    HIGH = 2
    VIP = 3

    def __lt__(self, other):
        return self.value < other.value

    def __le__(self, other):
        return self.value <= other.value

    def __gt__(self, other):
        return self.value > other.value

    def __ge__(self, other):
        return self.value >= other.value
```

3.4.4 WaitlistStatus Enum

```
class WaitlistStatus(Enum):
    ACTIVE = "active"
    CONVERTED =
    "converted" EXPIRED =
    "expired"
```

3.5 Domain Services

Domain Services berisi business logic yang tidak naturally belong ke entity atau value object tertentu.

3.5.1 AvailabilityService

Purpose: Complex availability queries dan analysis

Methods:

```
class AvailabilityService:
    def __init__( self,
        availability_repository: AvailabilityRepository,
        reservation_repository: ReservationRepository
    ):
        self.availability_repo = availability_repository
        self.reservation_repo = reservation_repository

    def search_availability( self,
        date_range: DateRange,
        room_type_id: str,
        guest_count: GuestCount
    ) -> Dict[date, int]:
```

```
        """Search availability for date range"""
        availability_map = {} current_date
        = date_range.start

        while current_date < date_range.end:
            availability = self.availability_repo.find_by_date_and_room_type( current_date,
                room_type_id
            )

            if availability:
                available = availability.available_rooms availability_map[current_date] =
                available
            else:
                # No availability record = fully available
                availability_map[current_date] = self._get_total_rooms(room_type_id) current_date +=

            timedelta(days=1)

        return availability_map

def is_continuously_available( self,
        date_range: DateRange,
        room_type_id: str,
        required_count: int
) -> bool:
        """Check if continuously available throughout date range"""
        availability_map  =  self.search_availability(
            date_range,
            room_type_id, GuestCount(adults=required_count,
            children=0)
        )

        return all(
            available >= required_count
            for available in availability_map.values()
        )

def calculate_occupancy_rate(
        self,
        date: date, room_type_id:
        str
) -> Decimal:
        """Calculate occupancy rate for specific date and room type"""
        availability = self.availability_repo.find_by_date_and_room_type( date,
            room_type_id
```

```
        )

        if not availability:
            return Decimal('0')

        total = availability.total_rooms reserved =
        availability.reserved_rooms

        if total == 0:
            return Decimal('0')

        return Decimal(reserved) / Decimal(total) * Decimal('100')

def predict_demand(
        self,
        date_range: DateRange,
        room_type_id: str
) -> DemandLevel:
        """Predict demand level based on historical data"""
        # Analyze historical reservations for same period
        historical_occupancy = self._get_historical_occupancy(
            date_range,
            room_type_id
        )

        # Calculate average occupancy
        avg_occupancy = sum(historical_occupancy) / len(historical_occupancy)

        # Determine demand level
        if avg_occupancy >= 85:
            return DemandLevel.HIGH
        elif avg_occupancy >= 65:
            return DemandLevel.MEDIUM
        else:
            return DemandLevel.LOW

def _get_total_rooms(self, room_type_id: str) -> int:
        """Get total rooms for room type from Inventory Context"""
        # This would typically call Inventory Context API
        pass

def _get_historical_occupancy( self,
        date_range: DateRange,
        room_type_id: str
) -> List[Decimal]:
        """Get historical occupancy data"""
        # Query historical data
```

```
        pass

class DemandLevel(Enum):
    LOW = "low"
    MEDIUM =
    "medium" HIGH
    = "high"
```

3.5.2 ReservationValidationService

Purpose: Validate business rules requiring external knowledge

Methods:

```
class ReservationValidationService:
    def _init_( self,
        inventory_client: InventoryClient,
        pricing_client: PricingClient
    ):
        self.inventory_client = inventory_client
        self.pricing_client = pricing_client

    def
        validate_reservation_rul
        es( self,
        reservation: Reservation,
        room_type: RoomType
    ) -> ValidationResult:
        """Validate all reservations rules"""
        errors = []

        # Check minimum stay
        if not self.check_minimum_stay(
            reservation.date_range,
            room_type
        ):
            errors.append("Minimum stay requirement not met")

        # Check blackout dates
        if not self.check_blackout_dates(reservation.date_range): errors.append("Selected dates
            include blackout dates")

        # Check advance reservations limit
        if not self.check_advance_reservations_limit(
            reservation.date_range.start
        ):
            errors.append("Reservations  too far in advance")

        # Validate guest capacity
        if not self.validate_guest_capacity(
            reservation.guest_count,
```

```
            room_type
        ):
            errors.append("Guest count exceeds room capacity")

        return ValidationResult(
            is_valid=len(errors) == 0,
            errors=errors
        )

def
        check_minimum_sta
        y( self,
        date_range: DateRange,
        room_type: RoomType
) -> bool:
        """Check minimum stay requirement"""
        min_stay = self._get_minimum_stay(date_range, room_type)
        return date_range.nights() >= min_stay

def check_blackout_dates(self, date_range: DateRange) -> bool:
        """Check if dates include blackout dates"""
        blackout_dates = self._get_blackout_dates()

        current = date_range.start
        while current < date_range.end:
            if current in blackout_dates:
                return False
            current += timedelta(days=1)

        return True

def check_advance_reservation_limit(self, check_in_date: date) -> bool:
        """Check advance reservation limit"""
        days_in_advance = (check_in_date - date.today()).days
        max_advance_days = 365
        # configurable

        return days_in_advance  <=  max_advance_days

def validate_guest_capacity( self,
        guest_count: GuestCount,
        room_type: RoomType
) -> bool:
        """Validate guest count against room capacity"""
        return guest_count.fits_in_room_capacity(
            room_type.max_capacity
        )

def _get_minimum_stay(
```

```
            self,
            date_range: DateRange,
            room_type: RoomType
        ) -> int:
            """Get minimum stay requirement"""
            # Weekend requirement
            if date_range.is_weekend():
                return 2

            # Room type requirement
            if room_type.category == "SUITE":
                return 2

            # Default
            return 1

        def _get_blackout_dates(self) -> Set[date]:
            """Get blackout dates from configuration"""
            # This would typically load from configuration
            return set()

    @dataclass
    class ValidationResult:
        is_valid: bool errors:
        List[str]
```

3.5.3 OverbookingService

Purpose: Manage overbooking strategy

Methods:

```
    class OverbookingService:
        def _init_( self,
            reservation_repository: ReservationRepository,
            availability_repository:  AvailabilityRepository
        ):
            self.reservation_repo = reservation_repository
            self.availability_repo  =  availability_repository

        def calculate_overbooking_limit( self,
            date: date, room_type_id:
            str
        ) -> int:
            """Calculate safe overbooking limit"""
            # Get historical data
            no_show_rate = self._get_no_show_rate(room_type_id) late_cancellation_rate  =
            self._get_late_cancellation_rate(room_type_id)
```

```
        # Get total rooms
        total_rooms = self._get_total_rooms(room_type_id)

        # Calculate safe limit
        expected_no_shows = total_rooms * (
            no_show_rate + late_cancellation_rate
        )

        # Cap at 10% for risk management
        max_limit = int(total_rooms * 0.10)

        return min(int(expected_no_shows), max_limit)

    def should_accept_overbooking( self,
        date: date, room_type_id:
        str
    ) -> bool:
        """Decide if should accept overbooking"""
        # Get current availability
        availability = self.availability_repo.find_by_date_and_room_type( date,
            room_type_id
        )

        if not availability:
            return False        # Not even at capacity

        # Check if already overbooked beyond limit current_overbooking = max(0,
        -availability.available_rooms) limit = self.calculate_overbooking_limit(date,
        room_type_id)

        if current_overbooking >= limit:
            return False

        # Check demand level (only overbook for high demand)
        demand = self._assess_demand(date, room_type_id)
        if demand != DemandLevel.HIGH:
            return False

        # Check risk factors
        if self._is_high_risk_period(date):
            return False        # Don't overbook during high-risk periods

        return True

    def manage_overbooking_situation( self,
```

```
        date: date, room_type_id:
        str
    ) -> List[UUID]:
        """Handle actual overbooking situation"""
        # Find reservations that can be relocated
        reservations = self.reservation_repo.find_by_date_and_room_type( date,
            room_type_id,
            status=ReservationStatus.CONFIRMED
        )

        # Score reservations for relocation (lower score = relocate first)
        scored_reservations = [
            (self._calculate_relocation_priority(r), r) for r in
            reservations
        ]
        scored_reservations.sort(key=lambda x: x[0])

        # Return IDs of reservations to relocate
        return [r.reservation_id for score, r in scored_reservations[:2]]

    def calculate_relocation_cost( self,
        reservation: Reservation
    ) -> Money:
        """Calculate cost of relocating a guest"""
        # Cost components:
        # 1. Room rate at comparable hotel
        comparable_rate = reservation.total_amount.multiply(
            Decimal('1.1')
        # Assume 10% higher
        )

        # 2. Transportation cost
        transportation = Money(Decimal('200000'), 'IDR')

        # 3. Compensation/goodwill
        compensation = reservation.total_amount.percentage(
            Decimal('20')
        # 20% discount on future stay
        )

        total_cost = comparable_rate.add(transportation).add(compensation)
        return total_cost

    def _get_no_show_rate(self, room_type_id: str) -> Decimal:
        """Get historical no-show rate"""
        # Query historical data
        return Decimal('0.03')          # 3% typical
```

```
def _get_late_cancellation_rate(self, room_type_id: str) -> Decimal:
        """Get historical late cancellation rate"""
        # Query historical data
        return Decimal('0.02')          # 2% typical

def _get_total_rooms(self, room_type_id: str) -> int:
        """Get total rooms from Inventory Context"""
        pass

def _assess_demand(self, date: date, room_type_id: str) -> DemandLevel:
        """Assess current demand level"""
        pass

def _is_high_risk_period(self, date: date) -> bool:
"""Check if high-risk period for overbooking"""
# Weekend check
        if date.weekday() in [5, 6]:
        # Saturday, Sunday
                return True

        # Check for special events
        # (Would query event calendar)

        return False

def _calculate_relocation_priority( self,
        reservation: Reservation
) -> int:
        """Calculate relocation priority (lower = relocate first)"""
        score = 100

        # OTA reservations easier to relocate
        if reservation.reservation_source.is_ota():
                score -= 30

        # Direct reservations harder to relocate
        if reservation.reservation_source.is_direct():
                score += 30

        # Loyalty members should not be relocated #
        (Would check guest loyalty tier)

        # Multi-night stays harder to relocate
        if reservation.get_nights() > 1: score +=
                20

        return score
```

3.6 Repositories

Repositories provide collection-like interface untuk aggregate persistence.

3.6.1 ReservationRepository Interface

```
from abc import ABC, abstractmethod
from typing import List, Optional from
uuid import UUID
from datetime import date

class ReservationRepository(ABC):
    """Repository interface for Reservation aggregate"""

    @abstractmethod
    def save(self, reservation: Reservation) -> None:
        """Persist  reservation"""
        pass

    @abstractmethod
    def find_by_id(self, reservation_id: UUID) -> Optional[Reservation]:
        """Find reservation by ID"""
        pass

    @abstractmethod
    def find_by_confirmation_code( self,
        confirmation_code: str
    ) -> Optional[Reservation]:
        """Find reservation by confirmation code"""
        pass

    @abstractmethod
    def find_by_guest(
        self,
        guest_id:  UUID,
        status:  Optional[ReservationStatus]  =  None
    ) -> List[Reservation]:
        """Find all reservations for a guest"""
        pass

    @abstractmethod
    def find_by_date_and_room_type(
        self,
        date: date, room_type_id:
        str,
        status:  Optional[ReservationStatus]  =  None
    ) -> List[Reservation]:
        """Find reservations for specific date and room type"""
```

```
        pass

    @abstractmethod
    def find_arriving_today(self) -> List[Reservation]:
        """Find reservations checking in today"""
        pass

    @abstractmethod
    def find_departing_today(self) -> List[Reservation]:
        """Find reservations checking out today"""
        pass

    @abstractmethod
    def delete(self, reservation_id: UUID) -> None:
        """Delete reservation (soft delete)"""
        pass

    @abstractmethod
    def exists_by_confirmation_code(
        self,
        confirmation_code: str
    ) -> bool:
        """Check if confirmation code exists"""
        pass
```

3.6.2 AvailabilityRepository Interface

```
class AvailabilityRepository(ABC):
    """Repository interface for Availability aggregate"""

    @abstractmethod
    def save(self, availability: Availability) -> None:
        """Persist availability"""
        pass

    @abstractmethod
    def find_by_date_and_room_type(
        self,
        date: date, room_type_id:
        str
    ) -> Optional[Availability]:
        """Find availability for specific date and room type"""
        pass

    @abstractmethod
    def find_by_date_range(
        self,
        date_range: DateRange,
```

```
        room_type_id: str
    ) -> List[Availability]:
        """Find availability for date range"""
        pass

    @abstractmethod
    def find_low_availability( self,
        threshold: int = 3
    ) -> List[Availability]:
        """Find dates with low availability"""
        pass
```

3.6.3 WaitlistRepository Interface

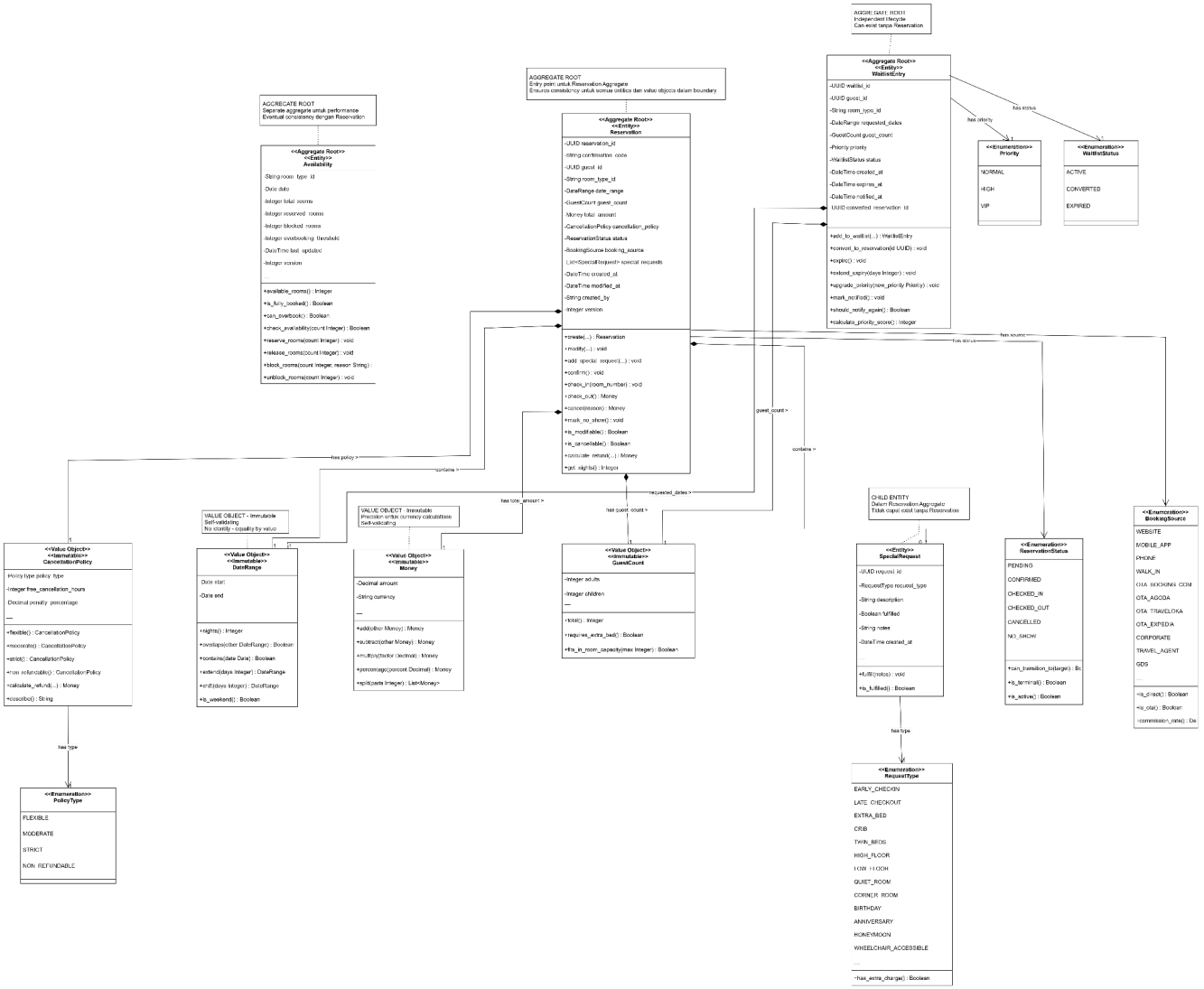
```
class WaitlistRepository(ABC):
    """Repository interface for Waitlist aggregate"""

    @abstractmethod
    def save(self, waitlist_entry: WaitlistEntry) -> None:
        """Persist waitlist entry"""
        pass

    @abstractmethod
    def find_by_id(self, waitlist_id: UUID) -> Optional[WaitlistEntry]:
        """Find waitlist entry by ID"""
        pass

    @abstractmethod
    def find_active_for_dates(
        self,
        date_range: DateRange,
        room_type_id: str
    ) -> List[WaitlistEntry]:
        """Find active waitlist entries for date range"""
        pass

    @abstractmethod
    def find_expired(self) -> List[WaitlistEntry]:
        """Find expired waitlist entries"""
        pass

    @abstractmethod
    def find_by_guest(self, guest_id: UUID) -> List[WaitlistEntry]:
        """Find all waitlist entries for a guest"""
        pass
```

---

4. CLASS DIAGRAM

4.1 Complete Class Diagram - Reservation Context

Berikut adalah Class Diagram lengkap yang menggambarkan struktur dan relasi antar komponen dalam Reservation Context:



Gambar 4 Class Diagram

Link Draw Io: https://drive.google.com/file/d/12_8QrZnkMqKC32e0TAnG89VT35rmag9s/view?usp=sharing

4.2 Diagram Components Overview

4.2.1 Reservation Aggregate Structure

<<Aggregate Root>>
Reservation

- reservation_id: UUID
- confirmation_code: String
- guest_id: UUID
- room_type_id: String

- date_range: DateRange
- guest_count: GuestCount
- total_amount: Money
- cancellation_policy: CancellationPoli
- status: ReservationStatus
- reservation_source: ReservationSource
- special_requests: List<SpecialRequest>
- created_at: DateTime
- modified_at: DateTime
- version: Integer

+ create(...)
+ modify(...)
+ confirm()
+ check_in()
+ check_out()
+ cancel()
+ add_special_request(...)
+ is_modifiable(): Boolean
+ is_cancellable(): Boolean
+ calculate_refund(...): Money

contains

0..*

<<Entity>>
SpecialRequest

- request_id: UUID
- request_type: RequestType
- description: String
- fulfilled: Boolean
- notes: String

4.2.2 Value Objects

<<Value Object>>
DateRange

- start: Date
- end: Date

+ nights(): Integer
+ overlaps(...): Boolean
+ contains(...): Boolean
+ extend(...): DateRange
+ shift(...): DateRange
+ is_weekend(): Boolean

<<Value Object>>
Money

- amount: Decimal
- currency: String

+ add(...): Money
+ subtract(...): Money
+ multiply(...): Money
+ percentage(...): Money
+ split(...): List<Money>

<<Value Object>>
GuestCount

- adults: Integer
- children: Integer

+ total(): Integer
+ requires_extra_bed()
+ fits_in_room_capacity()

4.2.3 Availability Aggregate

<<Aggregate Root>>
Availability

- room_type_id: String (PK)
- date: Date (PK)
- total_rooms: Integer
- reserved_rooms: Integer
- blocked_rooms: Integer
- overbooking_threshold: Integer
- last_updated: DateTime
- version: Integer

+ available_rooms: Integer (computed)
+ is_fully_booked: Boolean (computed)
+ can_overbook: Boolean (computed)

+ check_availability(...): Boolean
+ reserve_rooms(...)
+ release_rooms(...)
+ block_rooms(...)
+ unblock_rooms(...)

4.2.4 Domain Services

<<Domain Service>>
AvailabilityService

+ search_availability(...)
+ is_continuously_available(...)
+ calculate_occupancy_rate(...)
+ predict_demand(...)

<<Domain Service>>
ReservationValidationService

+ validate_reservation_rules(...)
+ check_minimum_stay(...)
+ check_blackout_dates(...)
+ validate_guest_capacity(...)

<<Domain Service>>
OverbookingService

+ calculate_overbooking_limit(...)
+ should_accept_overbooking(...)
+ manage_overbooking_situation(...)
+ calculate_relocation_cost(...)

4.3 Relationship Details Key

Relationships:

1. **Reservation → DateRange** (Composition)
   - Multiplicity: 1 to 1
   - DateRange is part of Reservation lifecycle
2. **Reservation → SpecialRequest** (Composition)
   - Multiplicity: 1 to 0..*
   - SpecialRequest cannot exist without Reservation
3. **Reservation → Money** (Composition)
   - Multiplicity: 1 to 1
   - Represents total_amount
4. **Reservation → GuestCount** (Composition)
   - Multiplicity: 1 to 1
5. **Reservation → CancellationPolicy** (Composition)
   - Multiplicity: 1 to 1
6. **Services → Repositories** (Dependency)
   - Services depend on repositories untuk data access

5. KESIMPULAN

5.1 Ringkasan Desain Taktis

Dokumen ini telah merancang model taktis lengkap untuk **Reservation Context** sebagai Core Domain dalam Sistem Reservasi Hotel. Desain mencakup:

Glosarium Ubiquitous Language:

- 60+ term bisnis yang didefinisikan dengan jelas
- Konsistensi bahasa antara domain experts dan developers
- Eliminasi ambiguitas dalam komunikasi tim

Model Domain yang Komprehensif:

- 3 Aggregates: Reservation, Availability, Waitlist
- 2 Entities: Reservation (root), SpecialRequest (child)

- 5 Value Objects: DateRange, Money, Reserva- tionStatus, CancellationPolicy, GuestCount
- 4 Enumerations: ReservationSource, RequestType, Priority, WaitlistStatus
- 3 Domain Services: AvailabilityService, ReservationValida- tionService, OverbookingService

Design Principles yang Diterapkan:

- Aggregate boundaries yang jelas untuk maintain consistency
- Immutable value objects untuk data integrity
- Rich domain model dengan business logic di entities
- Repository pattern untuk clean architecture
- Domain events untuk loose coupling

Walaupun desain Domain Services seperti **AvailabilityService**, **ReservationValidationService**, dan **OverbookingService** telah dijabarkan secara lengkap dalam Milestone 3 untuk memberikan gambaran utuh mengenai proses bisnis di Reservation Context, sebagian besar logika detailnya belum akan diimplementasikan secara penuh pada Milestone 4.

Pada fase implementasi (M04), service-service ini akan tersedia dalam bentuk **interface dasar atau stub**, dan hanya fungsi yang diperlukan langsung oleh API pada Reservation Aggregate yang direalisasikan. Integrasi lintas context (misalnya ke Pricing Context dan Inventory Context) masih akan menggunakan **simulasi data** dan **placeholder logic**. Langkah ini diambil agar implementasi M04 tetap fokus pada stabilitas core lifecycle reservasi tanpa menimbulkan ketidakkonsistenan atau ruang lingkup berlebih di luar konteks inti.

5.2 Key Design Decisions

1. Aggregate Granularity:

- Reservation sebagai aggregate yang relatif kecil namun complete
- Availability dipisah untuk optimize concurrency
- Waitlist independent untuk lifecycle yang berbeda

Rationale: Balance antara consistency requirements dan performance

2. Value Objects Usage:

- DateRange untuk encapsulate period logic
- Money untuk prevent calculation errors
- CancellationPolicy untuk encapsulate business rules

Rationale: Immutability, self-validation, dan business logic encapsulation

3. Status Management:

- Enum based status dengan state machine validation
- Explicit state transition methods
- Terminal states untuk audit trail

Rationale: Prevent invalid state transitions dan maintain data integrity

4. Domain Services:

- Services untuk logic yang span multiple aggregates

- Clear separation dari application services

- Stateless operations

Rationale: Single Responsibility Principle dan domain logic centralization

5.3 Implementation Guidelines Best

Practices untuk Implementasi:

1. **Always Validate in Domain Layer**
   - Entities and value objects self-validate
   - Fail fast dengan clear error messages
   - Prevent invalid state dari entering system
2. **Use Repositories Correctly**
   - Only persist aggregate roots
   - Use specifications untuk complex queries
   - Keep repository interface in domain layer
3. **Publish Domain Events**
   - Every significant state change publishes event
   - Use outbox pattern untuk reliability
   - Event consumers should be idempotent
4. **Maintain Invariants**
   - Aggregate ensures invariants at all times
   - No partial updates that break invariants
   - Use transactions properly
5. **Testing Strategy**
   - Unit test entities and value objects in isolation

- Integration test aggregates dengan repositories
- Contract test untuk anti-corruption layers

Sebagai persiapan menuju Milestone 4, perlu ditegaskan bahwa tidak semua elemen desain taktis pada Milestone 3 akan langsung direalisasikan pada tahap implementasi awal. Lingkup implementasi M04 difokuskan pada aggregate inti yang membentuk lifecycle reservasi yaitu **Reservation**, **Availability**, dan **Waitlist**, beserta repository dasar untuk ketiganya (PostgreSQL + Outbox Table).

Sementara itu, sejumlah domain services masih berada pada level desain konseptual dan akan diimplementasikan secara bertahap. Misalnya, *AvailabilityService* hanya mengaktifkan fungsi dasar yang dipakai Reservation API, sementara analitik occupancy masih berupa *stub*. *ReservationValidationService* mengimplementasikan aturan utama terlebih dahulu, sedangkan aturan lanjutan tetap sebagai desain. *OverbookingService* juga belum diwujudkan penuh, dan logika overbooking masih berupa kontrak konseptual. Dengan penegasan ini, kesinambungan lingkup dari M02 → M03 → M04 menjadi jelas: **Milestone 4 hanya mengimplementasikan Reservation Context**, sedangkan interaksi dengan Pricing, Inventory, Guest, Operations, dan Payment masih sebatas kontrak dan event, belum domain logic sesungguhnya.

---

6. DAFTAR PUSTAKA

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley.

Vernon, V. (2016). *Domain-Driven Design Distilled*. Addison-Wesley.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Millett, S., & Tune, N. (2015). *Patterns, Principles, and Practices of Domain-Driven Design*. Wrox.

Richardson, C. (2018). *Microservices Patterns: With Examples in Java*. Manning.

Newman, S. (2021). *Building Microservices* (2nd ed.). O'Reilly.

Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly.

Slide Kuliah II3160 – Teknologi Sistem Terintegrasi, STEI ITB: - Domain-Driven Design (DDD). (2022). - Architectural Styles, Architectural Patterns, and Design Patterns. (2022). - System Modeling. (2022).

Dokumentasi Online: - Domain-Driven Design Reference. (2015). Eric Evans. https://www.domainlanguage.co - Microsoft .NET Architecture Guides. https://docs.microsoft.com/en-us/dotnet/architecture/