# (Optional) Ontology vs Software Design

Bridging Knowledge Representation and Software Design/Architecture (ASE 420/ASE 456)

## The Connection

**You already know ontology concepts!**

We already discussed these topics in ASE420/456:

- **Object-Oriented Programming (OOP)** → Classes, inheritance, relationships
- **Domain-Driven Design (DDD)** → Entities, value objects, domain models
- **Database Design** → Entity-relationship diagrams

**Ontology uses similar ideas but with different goals**

## Today's Roadmap

1. **OOP vs Ontology** → What's similar, what's different
2. **DDD vs Ontology** → Domain modeling approaches
3. **Software Architecture vs Ontology** → Structural patterns
4. **When to use what** → Practical decision making

## Part 1: OOP and Ontology

### The Similarities

**Both use:**

- **Classes** → Define types of things
- **Inheritance** → IS-A relationships (hierarchy)
- **Properties** → Attributes and characteristics
- **Relationships** → How objects connect

## OOP Example: University System

```
// Class hierarchy
class Person {
    String name;
    int age;
}

class Student extends Person {
    String studentId;
    List<Course> enrolledCourses;
}

class Professor extends Person {
    String employeeId;
    List<Course> teachingCourses;
}

class Course {
    String courseCode;
    Professor instructor;
    List<Student> students;
}
```

## Same Domain in Ontology

```
# Class hierarchy
:Person rdf:type owl:Class .
:Student rdf:type owl:Class ;
         rdfs:subClassOf :Person .
:Professor rdf:type owl:Class ;
           rdfs:subClassOf :Person .
:Course rdf:type owl:Class .

# Properties
:hasName rdf:type owl:DatatypeProperty ;
         rdfs:domain :Person ;
         rdfs:range xsd:string .

:enrolledIn rdf:type owl:ObjectProperty ;
            rdfs:domain :Student ;
            rdfs:range :Course .

:teaches rdf:type owl:ObjectProperty ;
         rdfs:domain :Professor ;
         rdfs:range :Course .
```

## Key Similarity: Modeling Structure

**Both answer the same questions:**

1. What types of things exist? → **Classes**
2. How are they related? → **Hierarchy (IS-A)**
3. What properties do they have? → **Attributes**
4. How do they interact? → **Relationships (HAS-A, USES)**

**Both create a "blueprint" for the domain**

## The Critical Differences

| Aspect | OOP | Ontology |
|---|---|---|
| **Goal** | Execute behavior | Represent knowledge |
| **Methods** | Yes (functions) | No |
| **Reasoning** | No | Yes (inference) |
| **Runtime** | Compile/run | Query anytime |
| **Change** | Recompile code | Add triples |
| **Open World** | Closed | Open |

## Difference 1: Behavior vs Knowledge

### OOP: Focuses on what objects DO

```
class Student {
    void enrollInCourse(Course c) {
        if (!hasPrerequisites(c)) {
            throw new Exception("Missing prerequisites");
        }
        courses.add(c);
    }
}
```

### Ontology: Focuses on what objects ARE

```
:Alice rdf:type :Student .
:Alice :enrolledIn :CSC101 .
:CSC101 :hasPrerequisite :CSC100 .
# No behavior, just facts
```

## Difference 2: Closed vs Open World

### OOP: Closed World Assumption

```
Student alice = new Student("Alice");
if (alice.getGPA() == null) {
    // GPA is unknown = assume not set
    return "No GPA";
}
```

### Ontology: Open World Assumption

```
# If Alice's GPA is not stated...
# It's unknown, not necessarily false
# Could be discovered later from another source
```

**OOP: "Not found = false"**

**Ontology: "Not found = unknown"**

## Difference 3: Reasoning

### OOP: No automatic inference

```
class Dog extends Animal {
    boolean warmBlooded = true;
}

Dog rex = new Dog();
// Must explicitly check: rex.warmBlooded
```

### Ontology: Automatic inference

```
:Dog rdfs:subClassOf :Mammal .
:Mammal :hasProperty :warmBlooded .
:Rex rdf:type :Dog .

# Reasoner infers: :Rex :hasProperty :warmBlooded
```

## Real Example: Product Catalog

### OOP Implementation:

```java
class Product {
    String name;
    double price;
    Category category;

    boolean isAffordable() {
        return price < 1000; // Hard-coded logic
    }
}

class Laptop extends Product {
    int ram;
    String processor;
}
```

## Same in Ontology

```
:Product rdf:type owl:Class .
:Laptop rdfs:subClassOf :Product .

:hasPrice rdf:type owl:DatatypeProperty ;
          rdfs:domain :Product ;
          rdfs:range xsd:decimal .

:hasRAM rdf:type owl:DatatypeProperty ;
        rdfs:domain :Laptop ;
        rdfs:range xsd:integer .

# Rule for affordability (not hard-coded)
:AffordableProduct owl:equivalentClass [
    rdf:type owl:Restriction ;
    owl:onProperty :hasPrice ;
    owl:hasValue ?price .
    FILTER(?price < 1000)
] .
```

## When OOP Shines

**Use OOP when:**

✓ Need to execute behavior and algorithms

✓ Performance-critical operations

✓ Encapsulation of business logic

✓ Building interactive applications

✓ Well-defined, stable requirements

**Example:** Game engine, web server, mobile app

## When Ontology Shines

**Use Ontology when:**

✓ Need to integrate multiple data sources

✓ Reasoning and inference are important

✓ Domain knowledge changes frequently

✓ Need to explain decisions (traceability)

✓ Semantic search and discovery

**Example:** Knowledge base, recommendation system, medical diagnosis

## Part 2: DDD and Ontology

### Domain-Driven Design Concepts

**DDD focuses on:**

- **Ubiquitous Language** → Shared terminology
- **Bounded Contexts** → Domain boundaries
- **Entities** → Objects with identity
- **Value Objects** → Immutable descriptors
- **Aggregates** → Consistency boundaries

## DDD Example: E-commerce

```
// Entity: Has identity
class Order {
    OrderId id;  // Identity
    CustomerId customerId;
    List<OrderLine> lines;
    OrderStatus status;

    void addItem(Product product, int quantity) {
        lines.add(new OrderLine(product, quantity));
    }
}

// Value Object: No identity
class Money {
    BigDecimal amount;
    Currency currency;
}

// Aggregate: Order is aggregate root
```

## Same Domain in Ontology

```
# Entity concept
:Order rdf:type owl:Class .
:OrderLine rdf:type owl:Class .

# Identity
:hasOrderId rdf:type owl:DatatypeProperty ;
            rdfs:domain :Order .

# Relationships (similar to aggregates)
:hasOrderLine rdf:type owl:ObjectProperty ;
              rdfs:domain :Order ;
              rdfs:range :OrderLine .

# Value properties
:hasAmount rdf:type owl:DatatypeProperty ;
           rdfs:domain :OrderLine .

:hasStatus rdf:type owl:DatatypeProperty ;
           rdfs:domain :Order ;
           rdfs:range :OrderStatus .
```

## DDD and Ontology: Strong Similarities

1. **Domain Focus**

- Both model real-world domains
- Both use domain expert language
- Both capture domain rules

2. **Entity Modeling**

- DDD Entities ≈ Ontology Classes
- DDD Value Objects ≈ Ontology Datatypes
- DDD Aggregates ≈ Ontology Class hierarchies

3. **Relationships**

- DDD Associations ≈ Ontology Object Properties
- Both model domain interacti

## The Key Differences

| Aspect | DDD | Ontology |
|---|---|---|
| **Purpose** | Guide software design | Represent knowledge |
| **Behavior** | Included | Not included |
| **Boundaries** | Explicit (contexts) | Flexible (namespaces) |
| **Evolution** | Code changes | Triple additions |
| **Validation** | Unit tests | Reasoning engines |
| **Integration** | APIs/services | Query languages |

## Difference 1: Bounded Contexts

### DDD: Strict boundaries

```
// Sales Context
class Customer {
    CustomerId id;
    String name;
    CreditLimit limit;
}

// Shipping Context
class Customer {
    CustomerId id;
    Address shippingAddress;
    // Different properties!
}

// Same entity, different contexts
```

## Ontology: Unified Model

```
# Single unified model
:Customer rdf:type owl:Class .

# All properties in one namespace
:hasName rdfs:domain :Customer .
:hasCreditLimit rdfs:domain :Customer .
:hasShippingAddress rdfs:domain :Customer .

# Or use different namespaces
sales:Customer owl:sameAs shipping:Customer .
```

**Ontology prefers integration over separation**

## Difference 2: Ubiquitous Language

### DDD: Language shapes code

```
// Domain expert says: "Customer places Order"
class Customer {
    void placeOrder(Order order) {
        // Business logic matches language
    }
}
```

### Ontology: Language becomes triples

```
# Same language, but descriptive not procedural
:Customer :places :Order .
:alice :places :order123 .

# Focus on WHAT is true, not HOW to do it
```

## Part 3: Software Architecture and Ontology

### Architecture = Structure + Constraints

**Both define:**

- **Entities** → What exists
- **Relationships** → How things connect
- **Constraints** → What rules must hold

## Architecture Example: Microservices

```
┌──────────────┐    ┌──────────────┐
│   Order      │───▶️│  Payment     │
│  Service     │    │  Service     │
└──────────────┘    └──────────────┘
       │                    │
       ▼                    ▼
┌──────────────┐    ┌──────────────┐
│  Inventory   │    │  Shipping    │
│  Service     │    │  Service     │
└──────────────┘    └──────────────┘

Constraints:
- Order depends on Payment
- Payment must succeed before Shipping
- Inventory must be checked
```

## Same Architecture in Ontology

```
# Services as classes
:OrderService rdf:type :Service .
:PaymentService rdf:type :Service .
:InventoryService rdf:type :Service .
:ShippingService rdf:type :Service .

# Dependencies
:OrderService :dependsOn :PaymentService .
:OrderService :dependsOn :InventoryService .
:ShippingService :dependsOn :PaymentService .

# Constraints
:PaymentService :mustPrecedeInWorkflow :ShippingService .
```

## Architecture Pattern: Layered Architecture

```
┌─────────────────────────┐
│   Presentation Layer    │
├─────────────────────────┤
│   Application Layer     │
├─────────────────────────┤
│   Domain Layer          │
├─────────────────────────┤
│   Infrastructure Layer  │
└─────────────────────────┘

Rules:
- Each layer only depends on layer below
- No circular dependencies
```

## Layered Architecture in Ontology

```
:Layer rdf:type owl:Class .
:PresentationLayer rdfs:subClassOf :Layer .
:ApplicationLayer rdfs:subClassOf :Layer .
:DomainLayer rdfs:subClassOf :Layer .
:InfrastructureLayer rdfs:subClassOf :Layer .

# Dependency rules
:dependsOnLayer rdf:type owl:ObjectProperty .

# Constraints (using SHACL or SWRL)
IF ?layer1 :dependsOnLayer ?layer2
   AND ?layer2 :dependsOnLayer ?layer1
THEN violation  # No circular dependency
```

## Similarities: Domain Knowledge

### Both capture domain knowledge:

### Software Architecture (DDD):

```
// Domain knowledge in code structure
class LoanApplication {
    // Business rule
    boolean approve() {
        return creditScore > 700
            && debtRatio < 0.4
            && employmentYears >= 2;
    }
}
```

### Ontology:

```
# Same domain knowledge in rules
:ApprovableLoan owl:equivalentClass [
```

## Key Architectural Differences

| Aspect | Architecture | Ontology |
|---|---|---|
| **Execution** | Runtime system | Query system |
| **Components** | Services/modules | Classes/instances |
| **Communication** | APIs/messages | SPARQL queries |
| **State** | Mutable | Immutable facts |
| **Validation** | Tests | Reasoning |
| **Documentation** | Diagrams | Self-describing |

## When Architecture Is Better

**Use traditional software architecture when:**

✓ Building executable systems

✓ Need performance and scalability

✓ Complex workflows and processes

✓ Real-time interactions

✓ Transaction management

**Example:** Banking system, social media platform, IoT system

## When Ontology Is Better

**Use ontology when:**

✓ Knowledge integration from multiple sources

✓ Need semantic interoperability

✓ Flexible domain model (frequent changes)

✓ Reasoning and validation are critical

✓ Need to explain system decisions

**Example:** Healthcare records integration, regulatory compliance, research data

## The Hybrid Approach: Best of Both Worlds

### Modern systems use BOTH:

```
┌─────────────────────────────────────┐
│     Application Layer (OOP)         │
│  - Business logic                   │
│  - User interactions                │
│  - Performance-critical operations  │
└──────────────┬──────────────────────┘
               │
               ▼
┌─────────────────────────────────────┐
│    Ontology Layer                   │
│  - Domain knowledge                 │
│  - Rules and constraints            │
│  - Integration and reasoning        │
└─────────────────────────────────────┘
```

## Real-World Example: Healthcare System

### Application Code (Java/Spring):

```java
@Service
class PatientService {
    void admitPatient(Patient patient) {
        // Workflow and UI logic
        validateInsurance(patient);
        assignRoom(patient);
        notifyStaff(patient);
    }
}
```

### Ontology Layer:

```
# Medical knowledge
:Patient :hasDiagnosis :Diabetes .
:Diabetes :contraindicatedWith :Aspirin .
:Patient :prescribedMedication :Aspirin .
# Reasoner detects: CONFLICT!
```

## Comparison Summary Table

| Feature | OOP | DDD | Architecture | Ontology |
|---|---|---|---|---|
| **Classes** | ✓ | ✓ | ✓ | ✓ |
| **Inheritance** | ✓ | ✓ | Limited | ✓ |
| **Behavior** | ✓ | ✓ | ✓ | ✕ |
| **Reasoning** | ✕ | ✕ | ✕ | ✓ |
| **Flexibility** | Low | Medium | Low | High |
| **Integration** | Hard | Medium | Hard | Easy |
| **Validation** | Tests | Tests | Tests | Reasoning |

## Decision Guide: When to Use What

**Use OOP/Traditional Design:**

- Building applications
- Need behavior and algorithms
- Performance is critical
- Stable requirements

**Add DDD:**

- Complex domain
- Multiple teams
- Need ubiquitous language
- Long-term maintenance

## Decision Guide (continued)

**Add Ontology:**

- Multiple data sources
- Frequent domain changes
- Need reasoning/inference
- Regulatory compliance
- Semantic search required

**Use All Three:**

- Large enterprise systems
- Healthcare, finance, government
- Long-term evolution
- Multiple stakeholder integration

## Practical Example: Library System

### OOP for behavior:

```
class Library {
    void checkoutBook(Member m, Book b) {
        if (m.canBorrow() && b.isAvailable()) {
            createLoan(m, b);
        }
    }
}
```

### DDD for domain:

```
// Ubiquitous language from librarians
class Loan extends AggregateRoot {
    LoanId id;
    Member borrower;
    Book item;
}
```

## Library System (continued)

### Ontology for knowledge:

```
# Facts and rules
:Member123 :borrowed :Book456 .
:Book456 rdf:type :RareBook .
:RareBook :maxLoanDays 7 .

# Reasoner can check:
# Is loan overdue?
# Can member borrow more books?
# What books are related?
```

# Migration Path: From Design to Ontology

## Step 1: Start with your domain model

```
class Product {
    String name;
    Category category;
}
```

## Step 2: Extract to ontology

```
:Product rdf:type owl:Class .
:belongsToCategory rdfs:domain :Product .
```

## Step 3: Add reasoning rules

```
:PopularProduct owl:equivalentClass [
    :hasSalesCount ?count
    FILTER(?count > 1000)
]
```

## Tools for Hybrid Development

**Design Tools:**

- UML diagrams (PlantUML, draw.io)
- IDE (IntelliJ, VS Code)

**Ontology Tools:**

- Protégé (ontology editor)
- GraphDB (triple store)

**Integration:**

- Apache Jena (Java)
- OWL API (Java)
- rdflib (Python)

## Key Insights

**OOP and Ontology are complementary:**

- OOP = HOW things work
- Ontology = WHAT things are

**DDD and Ontology share goals:**

- Both model domains
- Both capture expert knowledge
- Different execution strategies

**Architecture and Ontology:**

- Architecture = system structure
- Ontology = knowledge structure
- Both [illegible]

## Best Practices

1. **Use the right tool for the job**
    - Don't use ontology for simple CRUD
    - Don't use OOP for knowledge representation
2. **Start simple**
    - Begin with familiar OOP/DDD
    - Add ontology layer when needed
3. **Keep them separate**
    - Application code (behavior)
    - Ontology (knowledge)
    - Clear interface between them

## Key Takeaways

✓ OOP and Ontology solve different problems
✓ DDD domain models translate well to ontologies
✓ Architecture patterns apply to knowledge too
✓ Modern systems benefit from hybrid approach
✓ Start with what you know, add ontology incrementally

You already understand ontology concepts through OOP and DDD!

## Next Steps

**In upcoming lectures:**

1. Hands-on: Building ontologies in Protégé
2. Connecting ontology to applications
3. Querying with SPARQL
4. Real project: Hybrid system design

**Practice: Take your current project's domain model and sketch it as an ontology**