Software Design and Architecture

Object Relational Mapping: A Persistence Mechanism for OO Applications

Data Management Systems

Until recently, the most efficient way to store data was in a relational database

- A relational database can store vast amounts of data in a structured way that allows for efficient storage, access, and search
- More recently, so called **NoSQL** solutions have been gaining production use on truly vast datasets with realtime and concurrent operational constraints
  - Think Facebook and Twitter and their use of Hadoop and Cassandra

Object-Relational Paradigm Mismatch

In the Object Systems, the problem with these persistence mechanisms is that their core abstractions are not objects

They are **tables with rows and columns (RDBMS)**

Or

They are (some variation on) **key-value pairs (NoSQL)**

Object-Relational Paradigm Mismatch

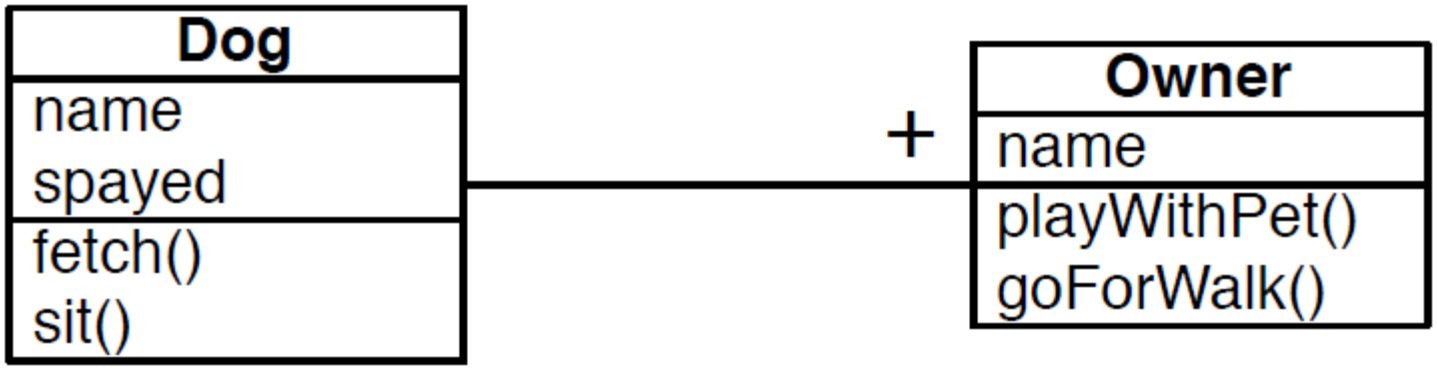
The OO application, on the other hand, has

- Classes, sub-classes, inheritance, associations
- Objects, attributes, methods, polymorphism

These concepts do not easily map into the abstractions of persistence mechanisms

- Even the creation of serialization mechanisms is non-trivial with the work that has to go in to traversing and reconstituting an object graph

An Example



In this object system, you will have Dog objects and Owner objects and some of them will be related to each other

You would represent this UML diagram in the relational database as follows

- a table called dogs to store Dog instances and
- a table called owners to store Owner instances
- columns corresponding to each attribute (plus an implicit id column)
- row corresponding to an instance of the class

How do we handle the relationship between Dog and Owner in this object-relational mapping system?

Based on the diagram

- Each owner has a single dog
- Each dog has at least one owner

This means that two owners can own the same dog

- Owner participates in a “has_one” relationship with Dog
- Dog participates in a “has_many” relationship with Owner

How do we handle the relationship between Dog and Owner in this object-relational mapping system?

The short answer is

- **foreign key relationships and join tables**

The somewhat longer answer is that most **object-relational mapping systems** have ways to specify these relationships

- They then take care of the details automatically
- You might see code like:
  List<Owner> owners = dog.getOwners();
- Behind the scenes, the method will hide the database calls required to find which owners are associated with the given dog

How do we handle the relationship between Dog and Owner in this object-relational mapping system?

Each instance of dog is assigned a unique id

- 1 | Fido | true
- 2 | Spot | false

Likewise owners

- 1 | Ken
- 2 | Max

A third table is then used to maintain mappings between them

- 1 | 1 ; 1 | 2 ; 2 | 2

This says that Fido is owned by Ken and Max and Spot is owned by Max

How do we handle the relationship between Dog and Owner in this object-relational mapping system?

That third table is known as a join table and has the structure

- dog_fk | owner_fk
- “1 | 1” in a row says that dog 1 is owned by owner 1

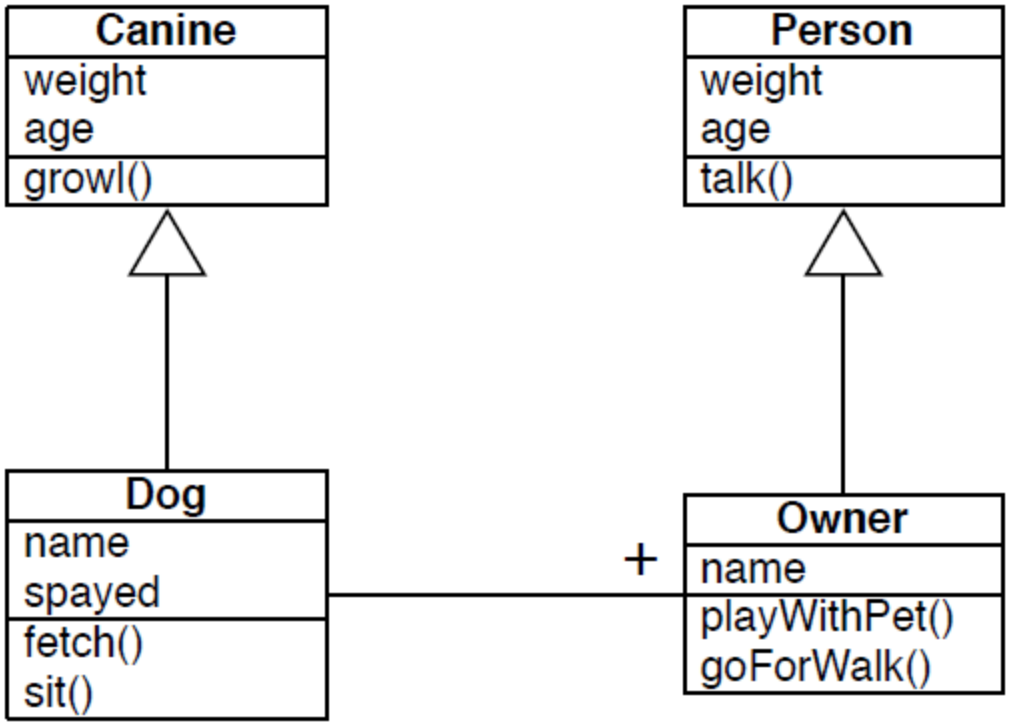
When it is time to implement the code

```
List<Owner> owners = dog.getOwners();
```

Then

- the code gets the id of the current dog asks for all rows in the join table where dog_fk == “id of current dog”
- this provides it with some number of rows; each row provides a corresponding owner_id which is used to lookup the names of the associated owners

A complication



How is inheritance handled?

The answer is “it varies across object-relational mapping systems”

Hibernate, has options to embed the attributes of the superclass into the tables of the subclasses

- Rather than one table per class, no table is generated for the superclass; instead one table per (leaf) subclass is generated
- the subclass table then has columns for each of the superclass atts

How is inheritance handled?

The answer is “it varies across object-relational mapping systems”

ActiveRecord (for Ruby on Rails) has options for creating a single table for the superclass and for each object storing all attributes as key-value pairs in a map

- subclasses are stored in the superclass table and have the option of adding key-value pairs to the map that only they process

How is inheritance handled?

The answer is “it varies across object-relational mapping systems”

There are other options

- including having distinct tables for each superclass and subclass and using foreign-key relationships to track relationships between tables
- an instance of a subclass would get its values from multiple tables

How is inheritance handled?

The important point is that the object-relational mapping system will hide the details from you

- You'll create a new instance and then invoke "**save()**" and the object gets picked apart and its values get stored in the appropriate tables

ORM Systems?

There are many different ORM systems available

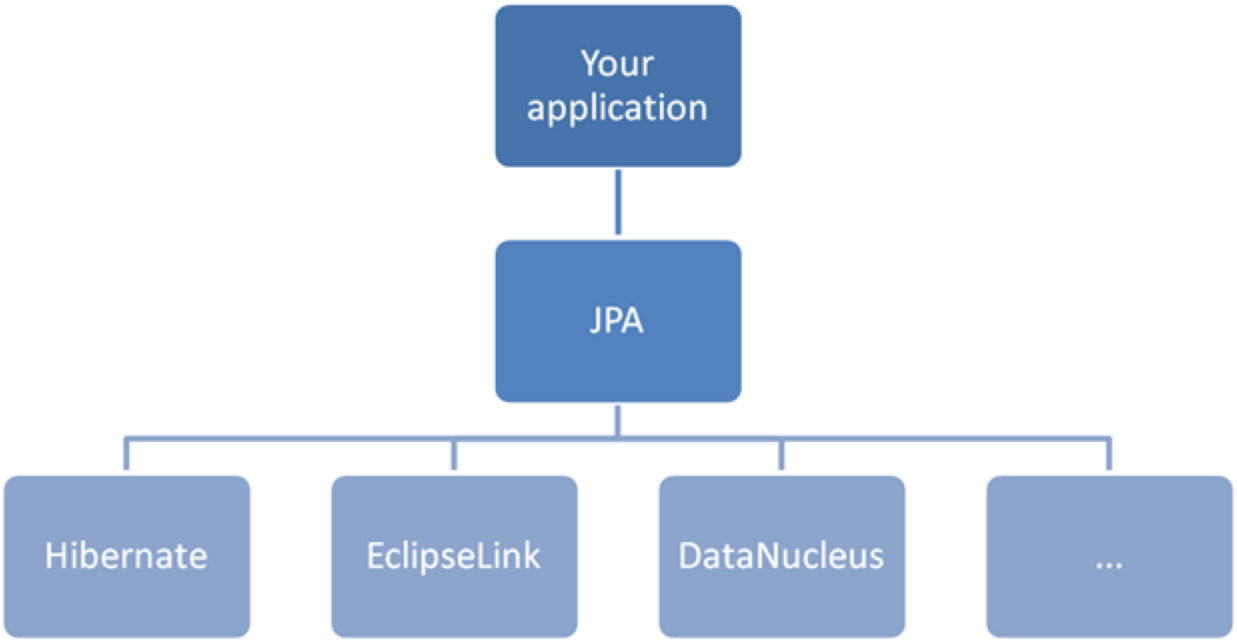
- **CoreData** from Apple
- **Hibernate** from JBoss
- **ActiveRecord** from Ruby on Rails

Hibernate

The most popular JPA vendor is Hibernate (JBoss)

JPA 1.0 was heavily influenced by Gavin King, the creator of Hibernate

- Much of what exists in JPA is adopted directly from the Hibernate project
- Many key concepts such as mapping syntax and central session/entity management exist in both



JPA is just an API (hence Java Persistence API) that requires an implementation to use.

Popular implementations include Hibernate, EclipseLink, OpenJPA and others.

Key Concepts

JPA utilizes annotated Plain Old Java Objects (POJOs)

Define an EntityBean for persistence

@Entity

Create database first

Create program by using database

Define relationships between beans

@OneToOne

@OneToMany

@ManyToOne

@ManyToMany

Key Concepts Cont...

Primitive types and wrappers are mapped by default

- String, Long, Integers, Double, etc.

Mappings can be defined on instance vars or on accessor methods of the POJO

Supports inheritance and embedding

EntityManger is used to manage the state and life cycle of all entities within a give persistence context

Primary keys are generated and accessed via @Id annotation

An Example

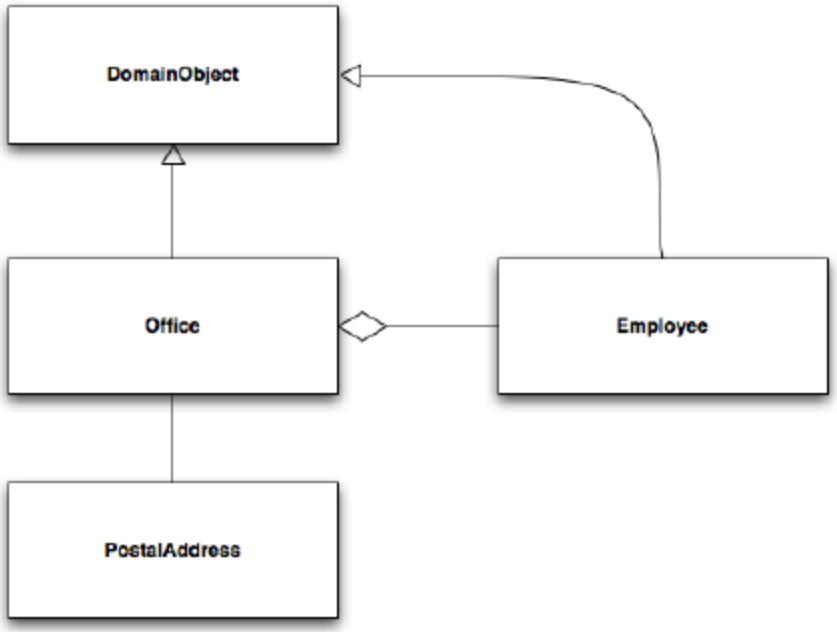
Design an application that allows a customer to view all employees that physically reside in a specific office

Each employee may only reside in one office

Employees must have first name, last name, phone number, id

Each office must have name, postal address, id

The Model



From Model to Code

Our model contains four classes

- Office
- Employee
- DomainObject
- PostalAddress

Office and Employee inherit from DomainObject

DomainObject holds on to best practice attributes such as id, creation date, modified date, version, etc.

From Model to Code Cont...

@Entity must be used to tell JPA which classes are eligible for persistence

@ManyToOne must be used to tell JPA there is an aggregation between Office and Employee

We'll show a use of @Embedded and @Embeddable for the Office-PostalAddress relationship

As well as inheritance using @MappedSuperclass

Domain Object

This class is not to be directly persisted

DB generated Id

For optimistic locking

Store as datetime

Call these methods before creation and modification

```java
@MappedSuperclass
public abstract class DomainObject implements Cloneable
{
    private Long id;
    private int version;
    private Date createDate;
    private Date modifiedDate;

    @Id
    @GeneratedValue
    public Long getId()
    {...}

    private void setId(Long id)
    {...}

    @Version
    public int getVersion()
    {...}

    private void setVersion(int version)
    {...}

    @Temporal(TemporalType.TIMESTAMP)
    public Date getCreateDate()
    {...}

    private void setCreateDate(Date createDate)
    {...}

    @Temporal(TemporalType.TIMESTAMP)
    public Date getModifiedDate()
    {...}

    private void setModifiedDate(Date modifiedDate)
    {...}

    @PrePersist
    private void handleCreateDate()
    {...}

    @PreUpdate
    private void handleModifiedDate()
    {...}

    public Object clone() throws CloneNotSupportedException
    {...}
}
```

Eligible for persistence

Embed PostalAddress in the same table as Office

```java
@Entity
public class Office extends DomainObject
{
    private String name;

    private PostalAddress postalAddress;

    public String getName()
    {...}

    public void setName(String name)
    {...}

    @Embedded
    public PostalAddress getPostalAddress()
    {...}

    public void setPostalAddress(PostalAddress postalAddress)
    {...}
}
```

Allow this object to be embedded by other objects

```java
@Embeddable
public class PostalAddress
{
    private String city;
    private String addressOne;
    private String addressTwo;
    private String zipCode;

    private State state;

    public String getCity()
    {...}

    public void setCity(String city)
    {...}

    public String getAddressOne()
    {...}

    public void setAddressOne(String addressOne)
    {...}

    public String getAddressTwo()
    {...}

    public void setAddressTwo(String addressTwo)
    {...}

    public String getZipCode()
    {...}

    public void setZipCode(String zipCode)
    {...}

    @Enumerated(EnumType.STRING)
    public State getState()
    {...}

    public void setState(State state)
    {...}
}
```

State is an Enum that will be treated as a String (varchar)

Eligible for persistence

```
@Entity
public class Employee extends DomainObject
{
    private String firstName;
    private String lastName;
    private String location;
    private String phoneNumber;

    private Office office;

    public String getFirstName()
    {...}

    public void setFirstName(String firstName)
    {...}

    public String getLastName()
    {...}

    public void setLastName(String lastName)
    {...}

    public String getLocation()
    {...}

    public void setLocation(String location)
    {...}

    public String getPhoneNumber()
    {...}

    public void setPhoneNumber(String phoneNumber)
    {...}

    @ManyToOne
    public Office getOffice()
    {...}

    public void setOffice(Office office)
    {...}
}
```

Defines the many to one association with Office

Explanation

@Embeddable and @Embedded

- Allows for the attributes of an embedded class to be stored in the same table as the embedding class

@Enumerated

- Allows for the value of an Enum to be stored in a column in the class's database table

@MappedSuperclass

- Allows for all attributes of the superclass to be utilized by the subclasses
- Duplicates all superclass attributes on subclass tables

The Database

JPA is capable of generating the underlying database for the developer

Most aspects of the generation are available for customization

- The defaults are generally good enough

Any @Entity causes the generation of a database table. Our generated tables are:

- Office table
- Employee table

Office Table

Field	Type
id	bigint(20)
createDate	datetime
modifiedDate	datetime
version	int(11)
name	varchar(255)
addressOne	varchar(255)
addressTwo	varchar(255)
city	varchar(255)
state	varchar(255)
zipCode	varchar(255)

Employee Table

Field	Type
id	bigint(20)
createDate	datetime
modifiedDate	datetime
version	int(11)
firstName	varchar(255)
lastName	varchar(255)
location	varchar(255)
phoneNumber	varchar(255)
office_id	bigint(20)

JPA is a specification that a developer can code to in order to easily leverage ORM technologies

There are a wide variety of vendors that implement the specification

- Coding to the spec allows the developer to be flexible in their choice of vendor implementations with limited ripple throughout the codebase

JPA greatly simplifies persistence of POJOs through a small set of easily utilized annotations

A Player-Team Example with Hibernate

Entities represent a player and a team with a one-to-many relationship. Each team could have many players, whereas a player could only play with a single team at a time.

```
@Entity
public class Player {
   @Id
   @GeneratedValue(strategy=GenerationType.SEQUENCE,
generator = "player_Sequence")
   @SequenceGenerator(name = "player_Sequence",
sequenceName = "PLAYER_SEQ")
   private Long id;
   @Column(name = "name")
   private String name;
   @Column(name = "num")
   private int num;
   @Column(name = "position")
   private String position;
   @ManyToOne(fetch = FetchType.LAZY)
      @JoinColumn(name = "team_id", nullable = false)
      private Team team;
   public Player() {
   }
      // getters/setters
}
```

```
@Entity
public class Team {
   @Id
   @GeneratedValue(strategy=GenerationType.SEQUENCE,
generator = "team_Sequence")
   @SequenceGenerator(name = "team_Sequence",
sequenceName = "TEAM_SEQ")
   private Long id;
   @Column(name = "name")
   private String name;
   @OneToMany(cascade = CascadeType.ALL,
         fetch = FetchType.EAGER,
         mappedBy = "team")
   private List<Player> players;
   public Team() {
   }
      // getters/setters
}
```