Object Relational Mapping

Kenneth M. Anderson
University of Colorado, Boulder
Lecture 29 — CSCI 4448/5448 — 12/11/12

Credit where Credit is Due

- The slides that cover Hibernate and JPA were developed by Aaron Schram
  - as part of his graduate presentation for this class
- Used with permission (Thanks Aaron!)

Goals of the Lecture

- Introduce the topic of object-relational mapping
- See examples in
  - Ruby on Rails
  - Hibernate

Object-Relational Mapping

- 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

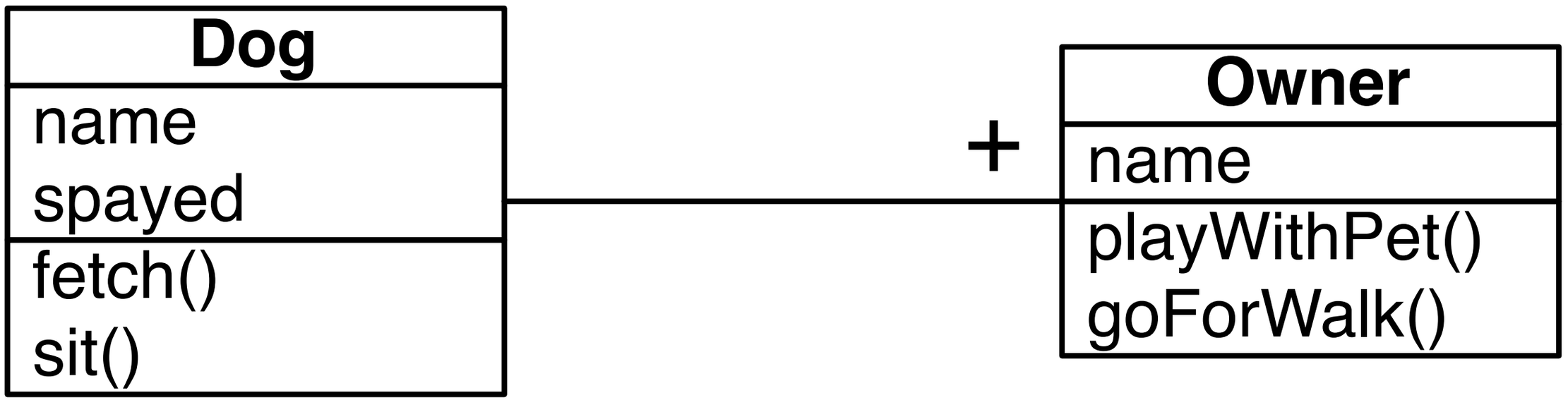
The Trouble with Objects (I)

- From an OO A&D standpoint, 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)

The Trouble with Objects (II)

- The OO world, 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


Dog
name
spayed
fetch()
sit()
+
Owner
name
playWithPet()
goForWalk()

Discussion (I)

- Think about how you would represent the previous UML diagram in a relational database
    - In the system, you will have Dog objects and Owner objects and some of them will be related to each other
- You will at least have
    - a table called dogs to store Dog instances and
    - a table called owners to store Owner instances
- Indeed, this is a convention of many object-relational mapping systems
    - class names are singular; table names are the associated plural form of the word: Person ⇒ People ; Cat ⇒ Cats; etc.

Discussion (II)

- Furthermore, for each table
  - you would have columns that correspond to each attribute (plus an implicit id column)
  - each row would correspond to an instance of the class
    - a spayed dog named Fido might have a row like:
      - 1 | Fido | true

Discussion (III)

- How do we handle the relationship between Dog and Owner?
- 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

Discussion (IV)

- How do we handle the relationship between Dog and Owner?
  - 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

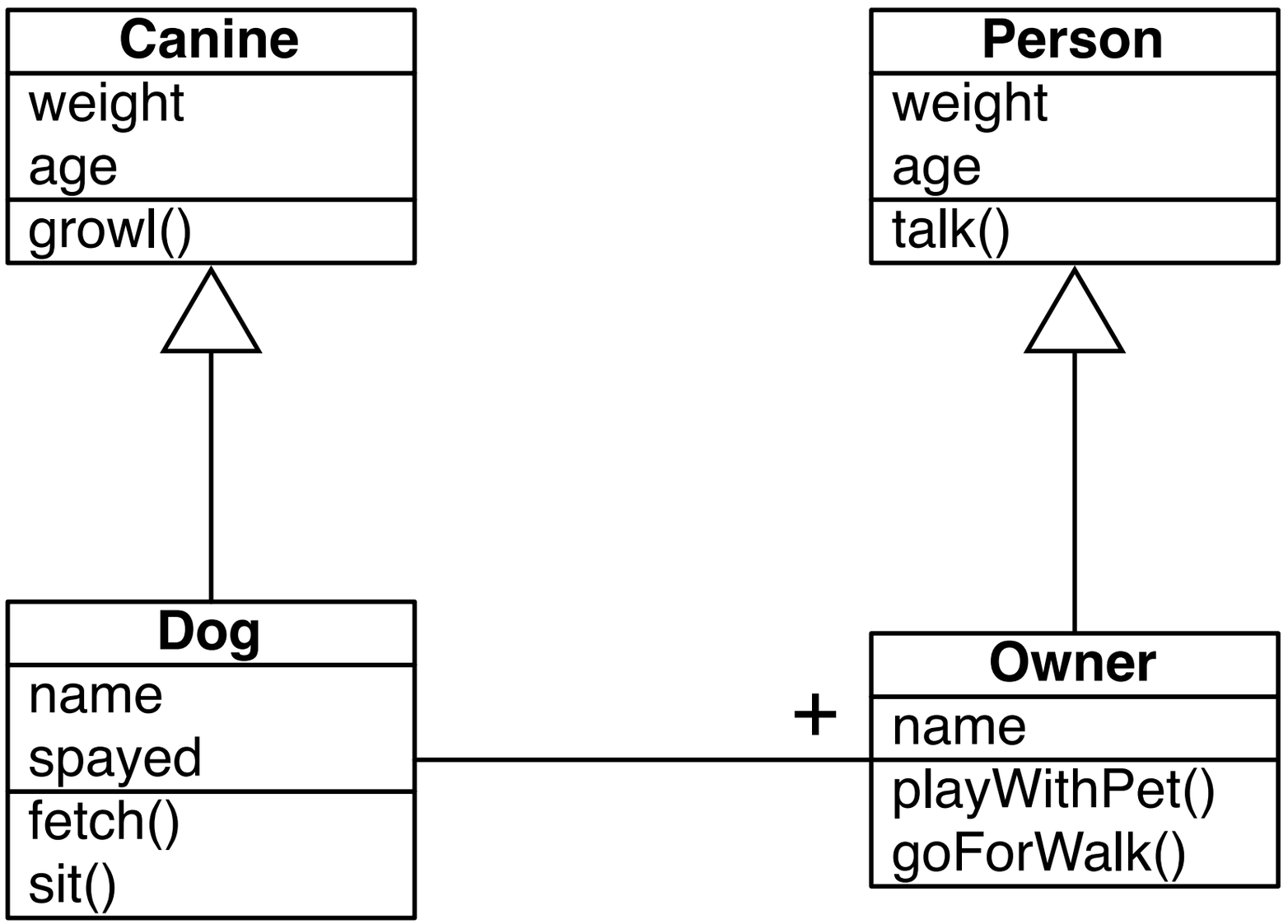
Discussion (V)

- How do we handle the relationship between Dog and Owner?
  - The relationship between Dog and Owner can be handled such that
    - 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

Discussion (VI)

- 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



Now what?

Discussion (I)

- The new version of the example adds parent classes to Dog and Owner
  - In our previous discussion, we said that
    - each class gets a table and each object is represented as a row in that table
    - We also saw that associations between classes get handled via join tables, which are distinct tables in which the rows track information about a specific instance of the association
- How is inheritance handled?

Discussion (II)

- How is inheritance handled?
  - The answer is “it varies across object-relational mapping systems”
  - Some systems, such as hibernate, have 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
- Some systems, such as ActiveRecord (for Ruby on Rails) have 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

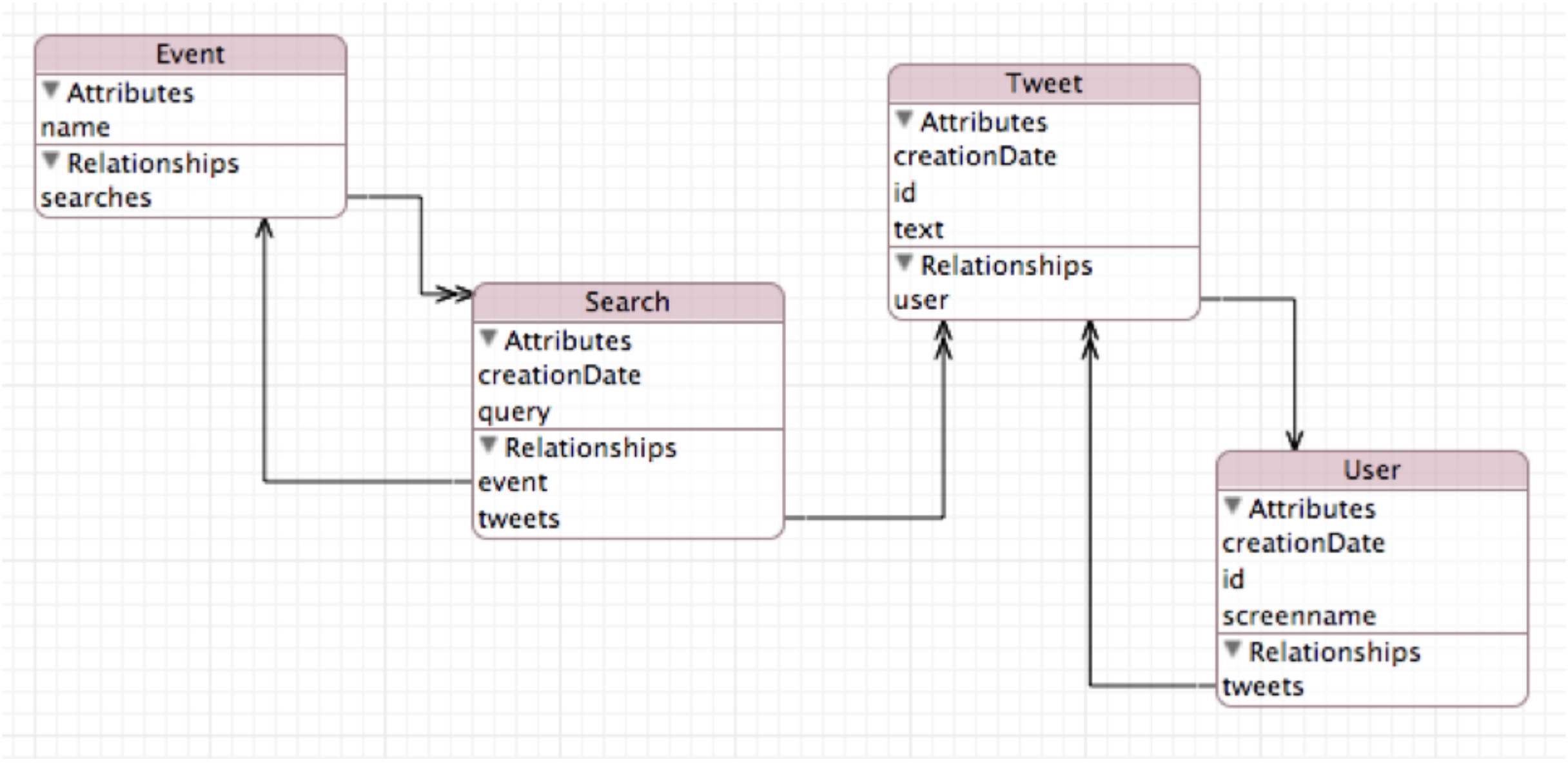
Discussion (II)

- How is inheritance handled?
  - 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
- These variations are just details, however; you might choose one approach over another based on your scalability constraints and your knowledge of how one database performs over another
  - 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
  - Prominent examples
    - CoreData from Apple
    - Hibernate from JBoss
    - ActiveRecord from Ruby on Rails

Apple's CoreData



CoreData has a graphical front-end for specifying the relationships between objects; it generates databases automatically from this spec

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

Key Concepts

- JPA utilizes annotated Plain Old Java Objects (POJOs)
  - Define an EntityBean for persistence
    - @Entity
  - 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

Office-Employees Example

- This was a common interview question at one of my previous employers

Question:

How could you model an employee management system using an ORM?

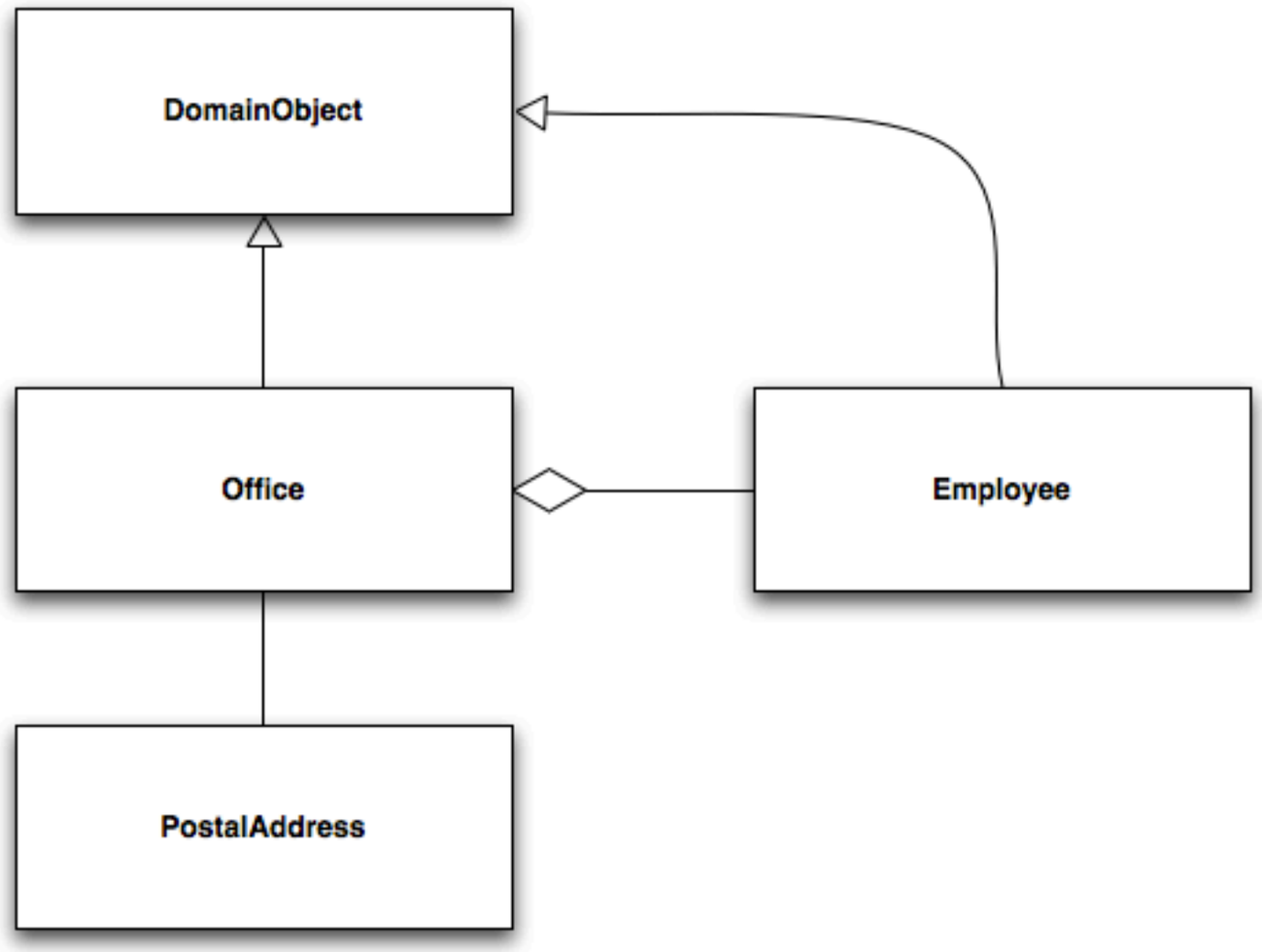
Question Details

In the interview we would build the whole application

- 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
- Any ORM will do, we'll use JPA...

Here, we'll just build out the model tier

The Model


DomainObject
Office
Employee
PostalAddress

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

DomainObject

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
    {...}
}
```

Office

Eligible for persistence

Embed PostalAddress in the same table as Office

```
@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)
    {...}
}
```

PostalAddress

Allow this object to be embedded by other objects

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

```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)
    {...}
}
```

Employee

Eligible for persistence

Defines the many to one association with Office

```java
@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)
    {...}
}
```

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

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)

FK to Office

Take Aways

Take Aways

- 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

ActiveRecord

- ActiveRecord is the Object-Relational Mapping system that is used by the Ruby on Rails web application framework
    - It takes advantage of “convention over configuration” to provide reasonable defaults that will meet most developers needs
        - For instance, if you create a table in your database called dogs and add a Ruby class called Dog to your Rails app, ActiveRecord can figure out that the two are connected
    - It will then provide methods for searching the table…
        - … and returning instances of the Dog class for manipulation and display by other parts of Ruby on Rails
    - It also autogenerates ids for each instance and will even generate attributes that will track, for instance, when a row was last updated

ActiveRecord Features (I)

- The code in a Ruby class that makes use of ActiveRecord is often quite simple; for instance, many of them look like this
  - class Order < ActiveRecord::Base
  - end
- A name and a subclass relationship and that's it
  - Note: ActiveRecord::Base is ActiveRecords key class and it (by default) indicates when a class will be associated with a table in a database
- Class Order will have an associated table called orders
  - The attributes associated with Order are then inferred by ActiveRecord at run-time; it adds attributes, getters, and setters to an Order object dynamically based on the information it finds in the associated table

ActiveRecord Features (II)

- ActiveRecord supports three types of relationships
  - One-to-One: declared via has_one and belongs_to
  - One-to-Many: declared via has_many and belongs_to
  - Many-to-Many: declared via has_and_belongs_to_many
- These declarations go in the class definition and reference the other class that participates in the relationship via a Ruby symbol

ActiveRecord Features (III)

- class Order < ActiveRecord::Base
  - has_many :line_items
- end
- class LineItem < ActiveRecord::Base
  - belongs_to: order
- end
  - belongs_to indicates the presence of a foreign key; in this example, line_items will contain an auto-generated foreign key to the orders table referencing the particular order that contains the line item
    - the full set of line_items associated with an order is found by scanning the line_items table

Support for CRUD (I)

- Creating a new instance of an object is as simple as
  - my_order = Order.new
  - order.name = “Ken Anderson”
  - order.email = “kena@cs.colorado.edu”
    - order.save
- Note: no need to set “order.id”; it is auto-generated
- Finding instances can be located via methods **find** (takes an id or a set of ids and returns object instances) or **where** (locates objects based on att values)
  - can autogenerate search routines via the find_by_<attname>
    - find_by_name and find_by_name_and_phonenumber

Support for CRUD (II)

- Support for update is as simple as finding an object, changing its attribute value, and invoking save
  - my_order = Order.find(5)
  - my_order.name = “Max Anderson”
  - my_order.save
- For deleting objects, two methods can be used: delete/delete_all and destroy/destroy_all
  - The former of each pair takes an id or a set of ids; the latter of each pair takes a query that first finds matching objects and then invokes either delete or destroy
    - destroy ensures that constraints are followed during deletion; delete bypasses those constraints

Support for Transactions

- ActiveRecord has support for transactions (as long as the underlying database supports transactions!)
  - This allows you to ensure that changes to model objects that need to be atomic are handled successfully, otherwise partial changes are rolled back and an exception is thrown
- The transaction is handled by a class method on a model object
  - account1 = Account.find(1);
  - account2 = Account.find(2)
  - Account.transaction do
    - account1.withdraw(100); account2.deposit(100);
  - end

This transaction will either transfer the money successfully or leave both objects unchanged

Simple Example (I)

- Let's take a look at the basic workflow of ORM in Ruby on Rails using the legendary "depot" example that has been featured in four editions of the following book
  - Agile Web Development with Rails by Sam Ruby (and others)
  - <http://www.pragprog.com/titles/rails4/agile-web-development-with-rails>
- I won't show the entire example (which eventually shows all the ins and outs of using ActiveRecord, migrations, rake, etc. in Ruby on Rails
  - In this example, we'll create the foundation for an e-commerce site in Rails centered around the model object called "Product"
- Note: I'm using Rails 3.1.3 and the an old version of Ruby 1.9.2 to run these examples

Simple Example (II)

- Create a Rails application
  - rails new depot
- This command creates a new Rails 3.0 application called depot; now type:
  - cd depot; rails generate scaffold Product title:string description:text image_url:string price:decimal
- This tells rails to generate the classes needed to have a model object called Product with attributes title, description, image_url and price
  - It creates a file called <date+id>_create_products.rb in the db/migrate directory; this file is known in Rails as a “migration” as it contains instructions to create this model object in an sqlite3 database and can be used to apply or rollback changes to the database structure

Simple Example (III)

- That file looks (kind of) like this; (below is the file generated by Rails 3.0.7)

```
class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      t.string :title
      t.text :description
      t.string :image_url
      t.decimal :price

      t.timestamps
    end
  end

  def self.down
    drop_table :products
  end
end
```

In code, this says “If we are applying this migration, then create the table products; if we are rolling back this migration, then drop (delete) the products table

Simple Example (IV)

- On the line that deals with defining the price in the migration, change it to read:
  - t.decimal :price, :precision => 8, :scale => 2
- Now, we ask Rails to apply this migration using a tool called rake
  - rake will discover that we have no database and will, as a result,
    - create one, and
    - apply the migration (which will, in turn, create the products table)
- Type: “rake db:migrate” in the depot directory and rake will create the database
  - This creates the file “development.sqlite3” in depot/db

Simple Example (V)

- How did Rails (rake) know to create this file?
    - Rails is designed around a concept called “convention over configuration”
        - when we created the depot application, Rails configured the app with a bunch of defaults; relevant to our situation here, there are defaults that say:
            - use sqlite3 as a database if not told otherwise
            - start in “development” mode (rather than “production” or “test”)
            - store the database in the db directory
            - etc.
- sqlite3 is a flat file database; you can use a hex editor to confirm that the newly created file contains a products table as specified by our migration

Simple Example (VI)

- And, that's it. We are ready to test our web app
  - Execute the command: "rails server" and use a web browser to visit the page: http://localhost:3000/products
- You will be presented with a web page that allows you to create, view, edit, and delete instances of the Product class!
- Now, if you've never used Rails before, you might be saying
  - "Where did all this functionality come from?"
- Well, when we created the database migration a few slides ago, we used the command
  - "rails generate scaffold Product…"
- The keyword here is "scaffold"; this tells Rails to auto-generate controllers and views that can create, read, update and delete the Product class, all for "free"

Simple Example (VII)

- If you check the sqlite3 file with a hex editor, you can again confirm that the database is being populated with instances of the products that you specify via the web interface
- Take a look at
  - depot/app/controllers/products_controller.rb
  - depot/app/views/products/index.html.erb
- to get a feel for how Rails is doing all this
  - In that auto-generated code, you will see references to
    - Product.new, Product.find, Product.save
  - all examples of ActiveRecord in action!
- You can see more advanced uses of ActiveRecord by buying the book!

Wrap Up

- Object-Relational Mapping Systems allow OO systems to take advantage of the scalability and efficiency benefits provided by modern persistence mechanisms
    - They provide services for “breaking apart” objects and storing them inside of tables or key-value stores and for “hydrating” objects stored in a persistence mechanism
        - bringing them back to object form, allowing getters and methods to be invoked, polymorphism to occur, setters to be written, etc.
            - all the while ensuring that proper database code is generated and invoked automatically to ensure that the current state of the object graph is always maintained
- We saw examples of CoreData, Hibernate and ActiveRecord