## The Problem

Values (data) need to be remembered or *persisted* between invocations of a program, between threads, and between processes.

We want to save object state (attributes) and relationships in a way that can be restored in the future.

## Requirements for a Persistence **Service**

For an O-O style application, the essential functional requirements for persistence are:

* **save** objects to persistent storage
* **retrieve** data and recreate objects from persistent storage
* **update** objects in persistent storage
* **delete** objects from persistent storage
* **preserve relationships** between objects when saving, updating, and retrieving objects
* **maintain object identity** and uniqueness - if we "get" the same object from persistent storage multiple times (perhaps using different kinds of queries), only 1 instance of each object is returned by the persistence service
* **atomic transactions** involving multiple objects (such as saving a collection), meaning the entire operation succeeds or fails. No partial saves or partial updates.

Another requirement which *may* be needed:

* *transactions*, which may be a group of operations, can be **rolled back** (undone).

Larman focuses on persisting objects to a **relational database**, but other kinds of databases may be suitable, depending on the application and structure of persistent data. Other kinds of database are generally referred to as "NoSQL" databases (Not Only SQL) such as object databases ([ObjectDB](https://www.objectdb.com/) for Java), document-oriented (MongoDB), key-value stores (Redis), and graph databases (Neo4j).

## **Object Persistence & Object Identity**

**Object Uniqueness** - In OO languages, each object has its own "identity" and you can distinguish objects even if they have identical values of all attributes. For example:

# Python - create 2 identical Persons

p1 = Person("Harry Hacker", 1409900123456)

p2 = Person("Harry Hacker", 1409900123456)

**p1 is p2**

**False**

On the other hand, if we write:

# Make p1 and p2 refer to the same person

p2 = p1

**p1 is p2**

**True**

Suppose we have a Persistence Service called dao. If we save p1 to persistent storage and then retrieve if again, does it return the same object reference?

# save object to database (persistent storage)

dao.save( p1 )

# retrieve object

p2 = dao.find(name="Harry Hacker")

# same object or a new object with the same attributes?

**p1 is p2**

**True** or **False**?

### **Identity Field**

Persistence services need a way to know what objects have already been added to persistence, and what objects are not yet in persistent storage. The service must also be able to uniquely identify persisted objects so it can update them. The service does not rely on the programming language notion of identity.

Here is why. Again, let dao refer to the persistence service.

# Python - create 2 identical Persons

p1 = Person("Harry Hacker", 1409900123456)

p2 = Person("Harry Hacker", 1409900123456)

# Save to persistent storage

dao.save(p1)

dao.save(p2)

How can the dao distinguish between the above code and this code:

# Save the same object twice

dao.save(p1)

dao.save(p1)

In the first code, p1 and p2 should be saved as separate entities (separate rows in a database table). In the second case, only one object should be saved.

Each entity object is assigned an "id" attribute. Larman calls this the **Object Identifier** (OID). Most persistence frameworks recommend using a numeric primitive (short, int, long) or numeric wrapper object (Short, Integer, Long) rather than String types. The id field usually has no significance in the domain model. For example:

public class Person {

// object id used by persistence service, can be private

protected int id = 0;

// attributes of a Person

private String name;

private long citizenId;

public Person(String name, long citizenId) { ... }

The persistence service interprets the **id** as:

id == 0 or null object is not in persistent storage

id != 0 or null object is in persistent storage. id has the primary key value.

### Django Example (Todo application from ISP)

Use your Django "todo" application from ISP or any Django code on your computer for this example.

In todo/models.py there is a Todo entity class:

from django.db import models

import django.contrib.auth.models

from django.core.validators import MinLengthValidator

class **Todo**(models.Model):

"""Todo is a task with a description and a 'done' flag"""

# Require the description to be at least 3 chars

**description** = models.CharField(

'Description',

max\_length=80,

blank=False,

validators=[MinLengthValidator(3)]

)

**done** = models.BooleanField('Done?', default=False, blank=True)

### Try this in the Django shell

1. create a Todo that is not persisted yet (don't use Todo.objects.create which automatically saves a new object)

2. check the value of the **id** field

3. save it

4. check the **id** field again. Did it change?

5. retrieve another *reference* to the same todo use Todo.objects.get(id=xxx). Is it the same object?

$ **python manage.py shell**

django>>> from todo.models import \*

django>>> **todo1 = Todo(description="Just do it")**

django>>> **todo1.id**

django>>> **todo1.save()**

django>>> **todo1.id**

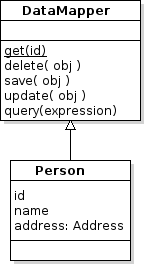
8

django>>> **todo2 = Todo.objects.get(id=8)**

## IS IT THE SAME OBJECT? ##

django>>> **todo1 is todo2**

## Two Designs For a Persistence Component

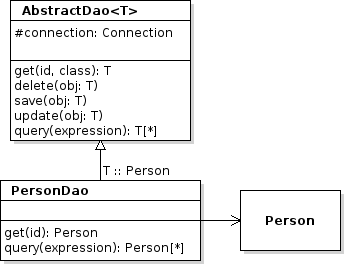
**1. Direct Mapping**

The classes provide their own persistence methods, typically through a superclass (to eliminate duplicate code and allow all objects to share a reference to the database connection).

This makes use of *Information Expert* but is an inflexible design, since entity classes (Person) must extend the Data Mapper class.

This design often uses the *Active Object Pattern*, to guarantee consistency between threads.

### 2. Data Access Objects or Database Mapp**er**

In this design, a separate class provides persistence services. This class is called a *Data Access Object* (DAO) and the design pattern is also called the *Data Access Object Pattern,* but Larman calls it the *Database Mapper* pattern.

DAO classes usually contain a lot of common code, so it's typical to create an abstract superclass for common DAO code, and create a subclass for each concrete DAO.

This design applies the *Single Responsibility Principle,* and can make use of different concrete implementations of the DAOs using a single interface.

### Common Elements in Both Designs

All the DAO (or DataMappers in the case of direct mapping design) usually share a single connection to the database. Both designs may use a superclass to factor-out common code.

**Database Constraints & Domain Rules**

"*There should be only one object (person) for a given Citizen ID."* is a **domain rule**.

Many Persistence Frameworks (including Django) let you specify uniqueness constraints **in code**.

At a lower level, you can also add constraints to the database schema.

It is more transparent to add constraints in code (or declaratively in a configuration file) rather than adding constraints to the SQL schema.

Django Example: the citizen\_id must be unique, not null, and at most 13 digits.

class Person(models.Model):

citizen\_id = models.CharField(**unique=True**,

max\_length=13,

**null=False**)

**Lazy Instantiation versus Eager Instantiation**

This usually applies to attributes of a *persisted* object that refer to another object. When you retrieve an object from persistent storage, should related objects be created *immediately* or wait until the code attempts to access the related object?

Suppose a Person object has an Address (a separate object):

class **Person** {

// object id for persistence

protected int id = 0;

// person's attributes

private String name;

private **Address** address;

Person p = new Person("Harry Hacker");

Address address =

new Address("50 Pahonyotin Rd", "Lat Yao", "Bangkok", 10900);

p.setAddress(address);

**dao.save(address);**

**dao.save(p);**

Now both Person p and address are in persistent storage.

If we retrieve "Harry Hacker" from storage using a query like this:

Person q = dao.find(name="Harry Hacker");

Should the dao also recreate (*materialize*) the address for q?

**Eager Instantiation** - materialize the Address attribute immediately

**Lazy Instantiation** - wait until the code tries to access the address (maybe it never will!) before materializing the Address.

Larman calls this "Lazy Materialization", but it means the same thing.

**Virtual Proxies**

There are 2 ways to implement *lazy instantiation*.

Virtual Proxies - supply a placeholder object that does not contain any data values. The first time any attribute or method is accessed, replace the proxy with a real object

Code Rewriting - In a language like Java, the persistence service can rewrite the bytecode for your classes to add code for lazy instantiation. For example, it might add code like this (in *italics*):

public class Person {

private Address address;

public Address getAddress() {

*if (this.address == null) {*

*this.address = addressAao.find("person\_id = "+this.id);*

*}*

return this.address;

}

### Cascading

When you save an object to the database, should related objects be saved also?

This is called *cascading*. Persistence frameworks let you specify which operations should "cascade", typically save and delete.

## **References**

Introduction to NoSQL Databases https://www.guru99.com/nosql-tutorial.html

https://www.dre.vanderbilt.edu/~schmidt/PDF/Active-Objects.pdf, describes the Active Objects design pattern but doesn't refer to persistence.