

Object Relational Mapping

ORM & SQLAlchemy

Alexis NEDELEC

LISYC EA 3883 UBO-ENIB-ENSIETA
Centre Européen de Réalité Virtuelle
Ecole Nationale d'Ingénieurs de Brest

enib ©2008



Développement d'applications

Problématique

- choix de la plateforme de développement
- choix du serveur de base de données

Solutions classiques

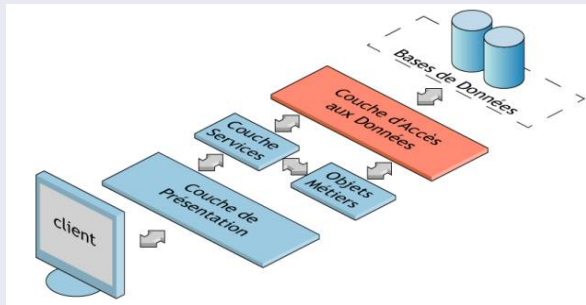
- J2EE/Oracle, .Net/SQL Server
- ASP/Access, PHP/MySQL
- ...

En résumé

- développements orientés objet
- gestion des données dans un SGBD Relationnel

Architecture Multi-niveaux (n-tiers)

Séparation des couches d'applications



- spécifications fonctionnelles indépendantes
- développement en parallèle
- maintenance et évolutivité plus souple

Couche d'accès aux données

DAL : Data Access Layer

- interactions avec la Base de Données
- interface entre l'application et les données
- évite la re-écriture fastidieuse de code

Fonctionnalités minimales du DAL

- lecture des données
- modification des enregistrements
- suppression d'enregistrements
- associer une classe à chaque table de la base

Mapping Objet-Relationnel

DAO : Data Access Object

- modèle de classe équivalente à une table
- doit implémenter une interface de type **CRUD**
 - Create, Read, Update, Delete (gestion des données)

Customer	
CP	<u>ID</u>
	FirstName LastName Age

Customer
+FirstName : string
+LastName : string
+Age : int
+LoadWithID()
+Create()
+Update()
+Delete()

Transformation des entités

Règle de transformation

- chaque entité devient une relation
- chaque entité a un attribut correspondant à une clé primaire
- une instance de classe \leftrightarrow un enregistrement de table

GUID: General Unique Identifier

- identifiant d'entité
- clés uniques, aucune signification vis-à-vis du métier

Transformation des entités : python/SQL

Modèle de classe

```
class User(object) :  
    def __init__(self, name, age, password='toto'):  
        self.name = name  
        self.age = age  
        self.password = password
```

Modèle de table correspondante

```
CREATE TABLE User (  
    user_pk INTEGER NOT NULL PRIMARY KEY,  
    name VARCHAR(20),  
    age INTEGER,  
    password VARCHAR(10) DEFAULT 'toto')
```

Impedance mismatch

Modèle Objet

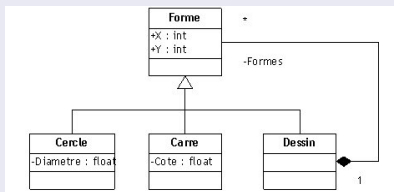


Table correspondante

Forme	
CP	ID
	Type
	X
	Y
	Diametre
	Cote
	FK_Formes

Impedance mismatch

Correspondance Classe-Table

- Entrée : un modèle objet (un graphe)
- Sortie : un modèle relationnel (une table)

Problème de correspondance

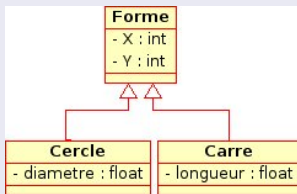
- identification des objets
- traduction des associations, héritages
- navigation dans le graphe d'objet
- dépendance entre les objets

Transformation d'héritage

Trois décompositions possibles

- décomposition par distinction
- décomposition descendante (*push-down*)
- décomposition ascendante (*push-up*)

Exemple d'héritage simple



Transformation d'héritage

Décomposition par distinction

- chaque sous-classe devient une table
- clé primaire de la sur-classe dupliqué dans les sous-classes
- clés primaires dupliquées : clés primaires et étrangères

Modèles de tables

FORME (forme_pk, x, y)

CERCLE (forme_fk, diametre)

CARRE (forme_fk, longueur)

Transformation d'héritage

Décomposition descendante

- dupliquer les attributs de la sur-classe dans les sous-classes
- sur-classe : vérifier les contraintes de totalité ou de partition

Avec contraintes de totalité ou de partition

CERCLE (cercle_pk, x, y, diametre)

CARRE (carre_pk, x, y, longueur)

Sans contraintes de totalité ou de partition

FORME (forme_pk, x, y)

CERCLE (cercle_pk, x, y, diametre)

CARRE (carre_pk, x, y, longueur)

Transformation d'héritage

Décomposition ascendante

- dupliquer les attributs des sous-classes dans la sur-classe
- supprimer les sous-classes

Modèle de table

FORME (forme_pk, x, y, diametre, longueur)

Un cercle peut être un carré!!!!

Solution : nouvel attribut

- pour représenter les sous-classes (**type**)
- valeurs nulles sur les attributs des autres sous-classes

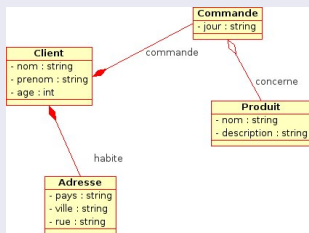
FORME (forme_pk, type, x, y, diametre, longueur)

Relations entre objets

Trois types de relations

- **association** : liens entre instances de classes
- **agrégation** : agrégation indépendante
- **composition** : agrégation composite

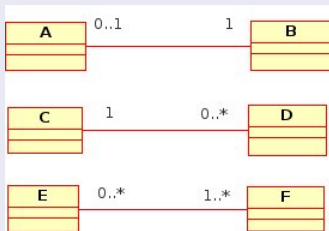
Modélisation des relations



Associations entre objets

Cardinalité de l'association

- **One-to-One** (1) : 1 au maximum entre chaque classe
- **One-to-Many** (1..*), (*..1) : 1 au maximum d'un côté et plus d'1 au maximum de l'autre côté de l'association
- **Many-to-Many** (*) : plus d'1 au maximum de chaque côté de l'association



Associations entre objets

Directionnalité de l'association

- **uni-directional** : une seule classe a connaissance de l'autre
- **bi-directional** : les deux classes se connaissent



Transformation d'associations

Programmation Objet : 3 possibilités

- (1),(*..1) : une variable d'instance référençant l'objet associé
- (1..*),(*) : une variable d'instance de type collection
- (*) : une classe d'association
- référence d'objet naturellement unidirectionnelle

Modèle relationnel

- (1),(*..1),(1..*) : une ou plusieurs clés étrangères
- (*) : une table de liaison, associative
- une association est toujours bidirectionnelle (jointure)

Transformation d'associations

Association One-to-Many

- clé étrangère dans la relation "fils" de l'association

Exemple d'association One-to-Many



COMPAGNIE (compagnie_pk, nom)

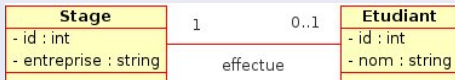
AVION (avion_pk, type, compagnie_fk)

Transformation d'associations

Association OneToOne

- (1-0..1) : clé étrangère dans la relation de cardinalité minimale de zéro
- (0..1-0..1) : choix de clé étrangère arbitraire
- (1-1) : fusionner sans doute les deux relations

Exemple d'association OneToOne



STAGE (stage_pk, entreprise)

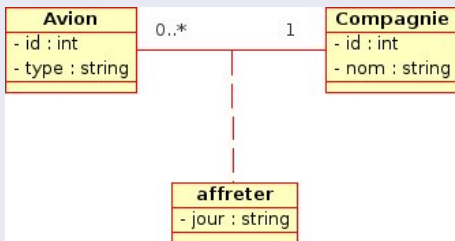
ETUDIANT(etudiant_pk, nom, stage_fk)

Transformation d'associations

Association ManyToMany

- clés étrangères : clé primaire d'une table associative

Exemple d'association ManyToMany



AVION (avion_pk, type, compagnie_fk)

AFFRETER (avion_fk, compagnie_fk, jour)

COMPAGNIE (compagnie_pk, nom)

Objectifs SQLAlchemy

Caractéristiques principales

- Supported databases (DB-API) :
 - SQLite, Postgres, MySQL, Oracle, MS-SQL ...
- Unit Of Work Pattern (Fowler) :
 - gérer des objets dans une transaction
- Function-based query construction :
 - fonction Python pour faire des requêtes
- Database/Class design separation :
 - objets persistants POPO (Plain Old Python Object)

Objectifs SQLAlchemy

Caractéristiques principales

- Eager/Lazy loading :
 - graphe entier d'objets en une/plusieurs requête(s)
- Self-referential tables :
 - gestion en cascade des tables auto-référencées
- Inheritance Mapping :
 - gestion de l'héritage (single, concrete, join)
- Raw SQL statement mapping :
 - récupération des requêtes SQL
- Pre/Post processing of data :
 - gestion des types prédéfinis (Generic, built-in, users)

Architecture SQLAlchemy

Composantes principales

- gestion des pools de connexion (**Engine**)
- information sur les tables (**Metadata**)
- mapping types SQL/ types Python (**TypeEngine**)
- exécution de requêtes (**Dialect**)
- persistance d'objet, ORM (**mapper**, **declarative_base**)

Création de tables

Connexion au SGBDR

```
from sqlalchemy import create_engine, MetaData
engine = create_engine(\
    'postgres://nedelec:toto@localhost/ORMDB')
metadata = MetaData(engine)
```

Définition et création de table

```
from sqlalchemy import Table, Column, Integer, String
users = Table('users', metadata,
    Column('user_pk', Integer, primary_key=True),
    Column('name', String(40)),
    Column('age', Integer),
    Column('password', String(10))
)
users.create()
```

Insertion et Recherche

Insertion d'enregistrements

```
#users = Table('users', metadata, autoload=True)
ins = users.insert()
ins.execute(name='Mary', age=30, password='secret')
```

Recherche d'enregistrements

```
statement = users.select()
result = statement.execute()
row = result.fetchone()
print 'Id:', row[0]
print 'Name:', row['name']
print 'Age:', row.age
print 'Password:', row[users.c.password]
```

Transaction Objet/Relationnel

Gestion de la connexion

```
connection = engine.connect()
trans = connection.begin()
try:
    result = users.select().execute()
    for row in result:
        print 'row', row
    trans.commit()
except:
    trans.rollback()
    raise
```

Transaction Objet/Relationnel

Création de transaction par session

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
# Session = sessionmaker()
# Session.configure(bind=engine)
session = Session()
...
#session.rollback()
session.commit()
```

Mapping Table-Classe

Classe correspondante

```
class User(object):
    def __init__(self, name, age, password):
        self.name = name
        self.age = age
        self.password = password
    def __repr__(self):
        return "<User('%s', '%s', '%s')>" \
            % (self.name, self.age, self.password)
```

Mapping Table-Classe

```
from sqlalchemy.orm import mapper
usermapping = mapper(User, users)
```

Instanciación / Insertion

Instanciación d'objet

```
dt_user = User('Dupont', 20, 'DupontPWD')
print dt_user.name
# primary key : user_pk => attribute : user_pk
print str(dt_user.user_pk)
```

Insertion dans la Base

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
session = Session()
session.add(dt_user)
session.commit()
```

Mapping Classe-Table

Déclaration de classe

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base(engine)
metadata = Base.metadata
class User(Base):
    __tablename__ = 'users'
    user_pk = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)
    password = Column(String)
    ...
#User.__table__.drop(checkfirst=True)
metadata.create_all(engine)
```

cf. : [Elixir \(elixir.ematia.de/trac/wiki\)](http://elixir.ematia.de/trac/wiki)

Identification d'objets

Création et identification d'objets

```
dd_user = User('Dupond', 20 , 'dupondPWD')
session.add(dd_user)
our_user = session.query(User).filter_by(\
    name='Dupond').first()
print our_user
print dd_user is our_user
```

Création et modification d'objets

```
dd_user.password='toto'
print session.dirty
session.add_all([
    User('Dupont', 21, 'dupontPWD'),
    User('Durand', 22, 'durandPWD')])
print session.new
```

Validation dans la base

Validation de la transaction

```
session.commit()
pu_user = User('pondu', 23 , 'ponduPWD')
session.add(pu_user)
session.rollback()
print pu_user
```

Création de table sur le serveur

```
$ psql -h pghostname -U myname ORMDB
```

```
ORMDB=# select * from users;
```

user_pk	name	age	password
1	Dupond	20	toto
2	Dupont	21	dupontPWD
3	Durand	22	durandPWD

Transformations d'héritage

Trois types de décomposition

- 1 ascendante (push-up) : on fait remonter les attributs dans une seule table (la sur-classe)
- 2 descendante (push-down) : on fait descendre les attributs dans les “sous-tables”
- 3 par distinction : on référence la table parente

SQLAlchemy inheritance mapping

- 1 Single table inheritance : transformation ascendante
- 2 Concrete table inheritance : transformation descendante
- 3 Joined table inheritance : transformation par distinction

Exemple d'héritage de classes

Classe de base : Employee

```
class Employee(object):
    def __init__(self, name):
        self.name = name
```

Classes Dérivées : Engineer, Manager

```
class Engineer(Employee):
    def __init__(self, name, eng_info):
        Employee.__init__(self, name)
        self.eng_info = eng_info

class Manager(Employee):
    def __init__(self, name, mng_data):
        Employee.__init__(self, name)
        self.mng_data = mng_data
```

Exemple d'héritage de classes

Instanciation d'employées

```
emps = [ Employee("Dupond"),  
         Employee("Dupont")]  
for e in emps:  
    session.add(e)
```

Instanciation d'employées

```
engs = [ Engineer("Durand", "computer"),  
         Engineer("Durant", "electronics")]
```

Instanciation d'ingénieurs, managers

```
mngs = [ Manager("Pondu", "bissness"),  
         Manager("Randu", "nessbiss")]
```

Transformation ascendante (push-up)

Propriétés de mapping

- hiérarchie de classes : 1 colonne par attribut de la hiérarchie
- `polymorphic_on` : 1 colonne pour distinguer les classes
- `polymorphic_identity` : 1 valeur associée à cette colonne
- `inherits` : pour récupérer les propriétés des classes

Création de la table

```
employees = Table('employees', metadata,
    Column('emp_pk', Integer, primary_key=True),
    Column('name', String(50)),
    Column('mng_data', String(50)),
    Column('eng_info', String(50)),
    Column('type', String(20), nullable=False)
)
```

Transformation ascendante (push-up)

Transformation de classes

```
emps_mapper= mapper(Employee, employees, \  
                    polymorphic_on=employees.c.type, \  
                    polymorphic_identity='employee')  
mapper(Engineer, \  
      inherits=emps_mapper, \  
      polymorphic_identity='engineer')  
mapper(Manager, \  
      inherits=emps_mapper, \  
      polymorphic_identity='manager')
```

Transformation ascendante (push-up)

Table correspondantes dans la Base

emp_pk	name	mng_data	eng_info	type
1	Dupond			employee
2	Dupont			employee
3	Durand		computer	engineer
4	Durant		electronics	engineer
5	Pondu	bissness		manager
6	Randu	nessbiss		manager

Transformation descendante (push-down)

Une table par classe

```
emps = Table('emps', metadata,
             Column('emp_pk', Integer, primary_key=True),
             Column('name', String(50))
)

engs = Table('engs', metadata,
             Column('eng_pk', Integer, primary_key=True),
             Column('name', String(50)),
             Column('eng_info', String(50)),
)

mngs = Table('mngs', metadata,
             Column('mng_pk', Integer, primary_key=True),
             Column('name', String(50)),
             Column('mng_data', String(50)),
)
```

Transformation descendante (push-down)

Solution “brutale”

```
mapper(Employee, emps)
mapper(Engineer, engs)
mapper(Manager, mngs)
```

Problème

Rechercher “Dupont” \Rightarrow consulter toutes les tables

Solution SQLAlchemy

- dictionnaire : `polymorphic_union()`
- clé du dictionnaire : identité polymorphique
- valeur : table correspondante dans la hiérarchie de classes

Transformation descendante (push-down)

```
polymorphic_union()
```

```
pjoin = polymorphic_union({'employee': emps,  
                           'manager': mngs,  
                           'engineer': engs},  
                           'type'  
)  
emps_mapper = mapper(Employee, emps,  
                      select_table=pjoin,  
                      polymorphic_on=pjoin.c.type,  
                      polymorphic_identity='employee'  
)
```

Transformation descendante (push-down)

```
polymorphic_union()
```

```
mapper(Manager, mngs,  
        inherits=emps_mapper,  
        concrete=True,  
        polymorphic_identity='manager'  
)  
mapper(Engineer, engs,  
        inherits=emps_mapper,  
        concrete=True,  
        polymorphic_identity='engineer'  
)
```

Transformation descendante (push-down)

Table emps, engs, mngs dans la Base

emp_pk	name
--------	------

-----+-----

1	Dupond
---	--------

2	Dupont
---	--------

eng_pk	name	eng_info
--------	------	----------

-----+-----+-----

1	Durand	computer
---	--------	----------

2	Durant	electronics
---	--------	-------------

mng_pk	name	mng_data
--------	------	----------

-----+-----+-----

1	Pondu	bissness
---	-------	----------

2	Randu	nessbiss
---	-------	----------

Transformation par distinction (**join**)

Join table inheritance

- 1 table par classe : contient les nouvelles informations
- références sur la table mère

Une table par classe

```
emps=Table('emps', metadata,  
           Column('emp_pk', Integer, primary_key=True),  
           Column('name', String(50)),  
           Column('type', String(30), nullable=False)  
)
```

Transformation par distinction (join)

Une table par classe

```
engs = Table('engs', metadata,
             Column('emp_fk', Integer,
                   ForeignKey('emps.emp_pk'),
                   primary_key=True),
             Column('eng_info', String(50))
)

mngs = Table('mngs', metadata,
             Column('emp_fk', Integer,
                   ForeignKey('emps.emp_pk'),
                   primary_key=True),
             Column('mng_data', String(50))
)
```

Transformation par distinction (`join`)

Une table par classe

```
mapper(Employee, emps,  
        polymorphic_on=emps.c.type,  
        polymorphic_identity='employee')  
mapper(Engineer, engs,  
        inherits=Employee,  
        polymorphic_identity='engineer')  
mapper(Manager, mngs,  
        inherits=Employee,  
        polymorphic_identity='manager')
```

Transformation par distinction (`join`)

Table emps dans la Base

```
ORMDB=# select * from emps;
```

```
emp_pk | name | type
```

```
-----+-----+-----  
      1 | Dupond | employee  
      2 | Dupont | employee  
      3 | Durand | engineer  
      4 | Durant | engineer  
      5 | Ponde  | manager  
      6 | Randu  | manager
```

(6 lignes)

Transformation par distinction (`join`)

Table engs, mngs dans la Base

```
ORMDB=# select * from engs;
```

```
eng_fk | eng_info
```

```
-----+-----
```

```
3 | computer
```

```
4 | electronics
```

(2 lignes)

```
ORMDB=# select * from mngs;
```

```
mng_fk | mng_data
```

```
-----+-----
```

```
5 | bissness
```

```
6 | nessbiss
```

(2 lignes)

Transformation de relations

Cardinalité de la relations

- 1 **One-To-One** (1) : maxi de 1 instance de chaque côté
- 2 **One-To-Many** (1..*),(*..1) : maxi de 1 instance d'un côté et de plus d'1 de l'autre
- 3 **Many-To-Many** (*) : plus d'1 instance de chaque côté

Trois types de mapping

- 1 **One-To-One** : cas particulier de **One-To-Many**
- 2 **One-To-Many** : une clé étrangère du côté **Many**
- 3 **Many-To-Many** : une table “secondaire” associative

Transformation de relations : OneToMany

Création des classes

```
class User(Base):
    __tablename__ = 'users'
    user_pk = Column(Integer, primary_key=True)
    ...

class Address(Base):
    __tablename__ = 'addresses'
    address_pk = Column(Integer, primary_key=True)
    email = Column(String, nullable=False)
    user_fk = Column(Integer,
                     ForeignKey('users.user_pk'))
    user_ref = relation(User,
                        backref=backref('addr_refs',
                                         order_by=address_pk))

metadata.create_all(engine)
```

Transformation de relations : OneToMany

Relation SGBDR : ForeignKey

```
class Address(Base):  
    ...  
    user_fk=Column(Integer,ForeignKey('users.user_pk'))
```

Relation Objet : relation(), backref()

```
class Address(Base):  
    ...  
    user_ref=relation(User,  
                       backref=backref('addr_refs',  
                                         order_by=address_pk))
```

Transformation de relations : OneToMany

Creation d'objets en relation

```
user_dt = User('Dupont', 30)
print user_dt.addr_refs
user_dt.addr_refs = [Address(email='dupont@bidon.com'),
                    Address(email='dupont@bondi.com')]
print user_dt.addr_refs
```

Navigation entre objets en relation

```
print user_dt.addr_refs[1]
print user_dt.addr_refs[1].user_ref
```

insertion dans la base

```
session.add(user_dt)
session.commit()
```

Transformation de relations : OneToMany

Etat de la base de données

```
$ psql -h pghostname -U myname ORMDB
```

```
ORMDB=> select * from users;
```

user_pk	name	age	password
1	Dupont	30	

(1 ligne)

```
ORMDB=> select * from addresses;
```

addr_pk	email	user_fk
1	dupont@bidon.com	1
2	dupont@bondi.com	1

(2 lignes)

Manipulation d'objets en relation

Chargement à la demande : **Lazy loading relation**

```
# pas de requête sur la base
user_dt = session.query(User) \
                .filter_by(name='Dupont') \
                .one()

print user_dt

# une requête SQL sur la base pour charger les adresses
print user_dt.addr_refs
```

Chargement en une seule fois : **Eager loading relation**

```
from sqlalchemy.orm import eagerload
user_dt = session.query(User) \
                .options(eagerload('addr_refs')) \
                .filter_by(name='Dupont').one()

print user_dt.addr_refs
```

Transformation de relations : OneToMany

Jointures internes : join()

```
join(User, Address)
join(User, Address, User.user_pk==Address.user_fk)
join(User, Address, User.addr_refs)
join(User, Address, 'addr_refs')
```

Jointures internes : query() + join()

```
session.query(User).select_from(join(User,Address)).\
    filter(Address.email=='dupont@bidon.com').all()
session.query(User).join(User.addr_refs).\
    filter(Address.email=='dupont@bidon.com').all()
session.query(User).join(User.addr_refs).all()
```

Transformation de relations : ManyToMany

Création de la table associative

```
mails= Table('users_mails', metadata,
             Column('address_refs', Integer,
                    ForeignKey('addresses.address_pk')),
             Column('user_refs', Integer,
                    ForeignKey('users.user_pk')))
)
```

Référencement : relation(...,secondary=...,...)

```
class Address(Base):
    __tablename__ = 'addresses'
    ...
    usersmails = relation('User',
                          secondary=mails,
                          backref='addresses')
```

Transformation de relations : ManyToMany

Instantiation d'objets en relation

```

user_dt = User('Dupont', 30)
session.add(user_dt)
addr_dt = Address(email='dupont@bidon.com')
session.add(addr_dt )
addr_dt.usersmails.append(user_dt)
addr_dt.usersmails.append(User('Dupond', 20, 'dupondPWD'))
session.commit()

```

Etat de la base de données

```
ORMDB=> select * from users;
```

user_pk	name	age	password
1	Dupont	30	
2	Dupond	20	dupondPWD

Transformation de relations : ManyToMany

Etat de la base de données

```
ORMDB=> select * from addresses;
```

```
address_pk | email
```

```
-----+-----
```

```
1 | dupont@bidon.com
```

(1 ligne)

```
ORMDB=> select * from users_mails;
```

```
address_refs | user_refs
```

```
-----+-----
```

```
1 | 1
```

```
1 | 2
```

(2 lignes)

Transformation de relations : ManyToMany

Bidirectionnalité de la relation

```
class User(Base):
    __tablename__ = 'users'
    ...
    addrmails= relation('Address',
                        secondary=mails,backref='users')
    ...
    ...
addr_dd = Address(email='durand@donbi.com')
session.add(addr_dd)
user_dd=User('Durand', 25 , 'durandPWD')
session.add(user_dd)
user_dd.addrmails.append(addr_dd)
session.commit()
```

Transformation de relations : ManyToMany

Etat de la base de données

```
ORMDB=> select * from users;
```

user_pk	name	age	password
1	Dupont	30	
2	Dupond	20	dupondPWD
3	Durand	25	durandPWD

(3 lignes)

Transformation de relations : ManyToMany

Etat de la base de données

```
ORMDB=> select * from addresses;
```

address_pk	email
1	dupont@bidon.com
2	durand@donbi.com

(2 lignes)

```
ORMDB=> select * from users_mails;
```

address_ref	user_ref
2	3
1	1
1	2

(3 lignes)

Declarative mapping

Définition de classe/Table

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class SomeClass(Base):
    __tablename__ = 'someTable'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
```

Association Classe/table

- `__table__` :
- `__mapper__` :

Declarative mapping

Définition d'attributs

```
class SomeClass(Base):  
    __tablename__ = 'some_table'  
    id = Column("some_table_id", Integer, primary_key=True)  
    name = Column("name", String(50))
```

Définition d'attributs

```
class SomeClass(Base):  
    __tablename__ = 'some_table'  
    id = Column(Integer, primary_key=True)  
    name = Column(String(50))
```

Bibliographie

Livres

- **R. Copeland** : “Essential SQLAlchemy”
Éditions O'Reilly (2008)
- **C.Soutou** : “UML 2 pour les bases de données”
Éditions Eyrolles(2007)

Bibliographie

Adresses “au Net”

- www.sqlalchemy.org : le site officiel
- www.dotnetguru.org/articles/Persistence : suivre le lien [livreblanc/ormmapping.htm](http://livreblanc.ormmapping.htm) (Sébastien Ross)
- www.agiledata.org/essays/mappingObjects.html : ORM in details
- www.limsi.fr/Individu/pointal/python.html : les liens python de Laurent Pointal
- www.hibernate.org
- www.oracle.com : TopLink JPA (Java Persistence API)
- www.ezpdo.net : ORM et PHP