# Object Relational Mapping

## ORM & SQLAlchemy

*Alexis NEDELEC*

Centre Européen de Réalité Virtuelle
Ecole Nationale d'Ingénieurs de Brest



# Object Relational Mapping

## ORM & SQLAlchemy

1. Introduction (de l'objet au relationnel)
2. Transformation de modèles
3. Présentation de SQLAlchemy
4. Transformation d'Héritage
5. Transformation d' Association
6. Application: Gestion de graphes

# Développement d'applications

## Problématique

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

## Solutions classiques

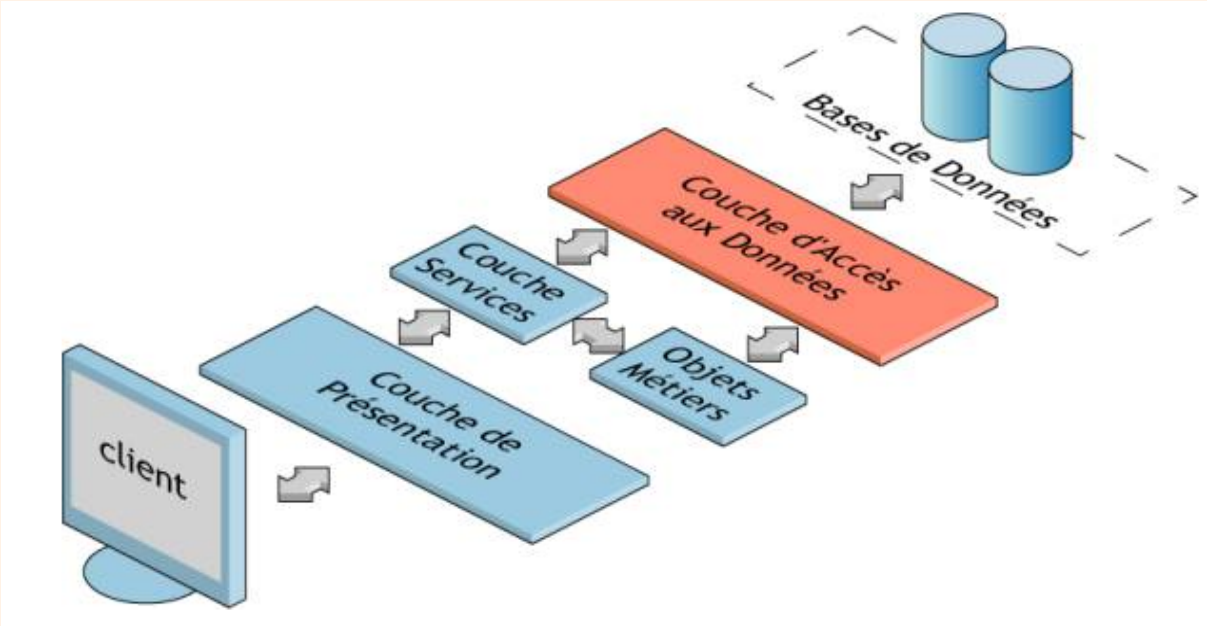
- J2EE/Oracle, .Net/SQL Server
- ASP/Access, PHP/MySQL
- ...

## 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

# Data Access Layer

## DAL : Couche d'accès aux données

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

## DAL : Implémentation

basé sur le modèle "active record pattern " (Martin Fowler)

```
dvd = new DVD()
dvd.title = "Die Hard 6"
dvd.price = 12.50
dvd.save()
# INSERT INTO DVDs (title, price)
#        VALUES    ('Die Hard 6',12.50);
```

# Data Access Layer

## DAL : Implémentation

```
dvd = DVD.find_first("title", "Die Hard 6")
# SELECT *
# FROM DVDs
# WHERE title = 'Die Hard 6' LIMIT 1;
```

## Fonctionnalités minimales du DAL

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

# Object Relational Mapping

## **CRUD** : Create, Read, Update, Delete

| Customer | |
|---|---|
| CP | ID |
| | FirstName<br>LastName<br>Age |

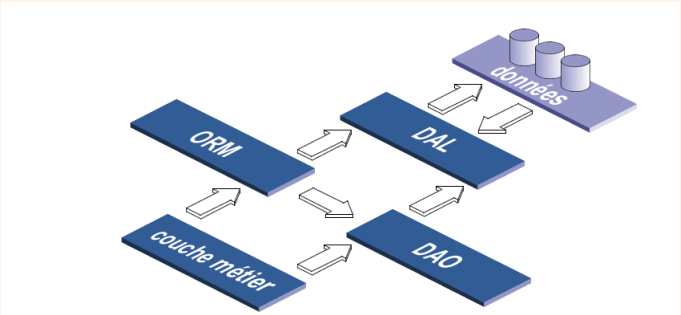
**Customer**

+FirstName : string
+LastName : string
+Age : int

+LoadWithID()
+Create()
+Update()
+Delete()

## **ORM** : associations entre entités (tables et classes)

Transformer associations Tables/Classes et réciproquement



# Transformation de modèle

## 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

# De la classe à la table

## Modèle de classe (python)

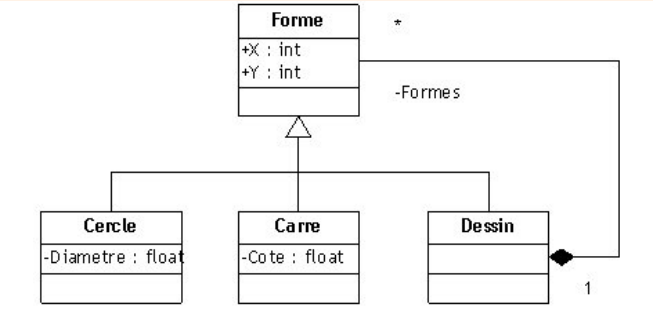
```
class User(object) :
    def __init__(self, name, age, password='toto'):
        self.name = name
        self.age = age
        self.password = password
```

## Modèle de table (SQL)

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

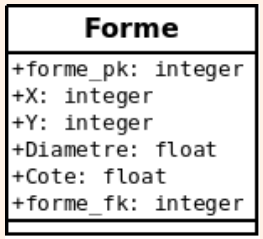
# Impedance mismatch

## Modèle Objet




## Modèle de table

Forme (forme_pk, X,Y,Diametre,Cote,#forme_fk)



# Impedance mismatch

## Correspondance Classe-Table

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

## Problème de correspondance

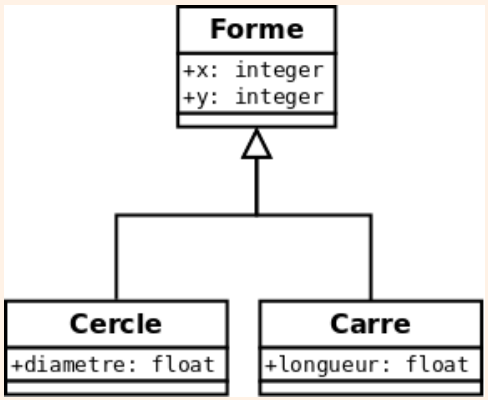
- identification des objets
- traduction des associations, héritages
- navigation dans le graphe d'objet
- dépendance entre les objets

# Transformation d'héritage

## Trois décompositions possibles

- décomposition par distinction (*join*)
- 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

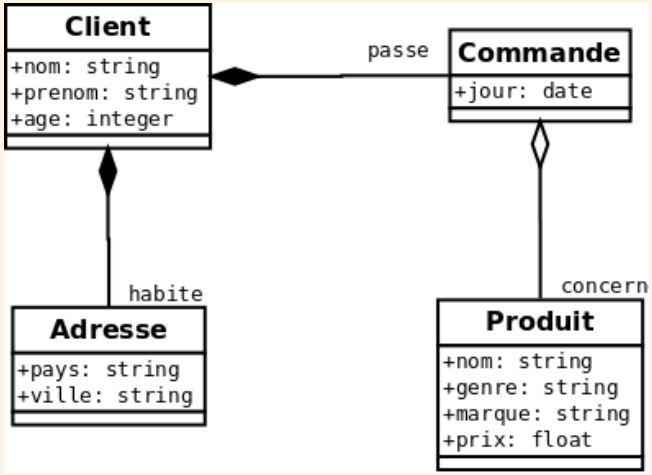
`FORME (forme_pk, type, x, y, diametre, longueur)`

# Associations entre objets

## Trois types de relations

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

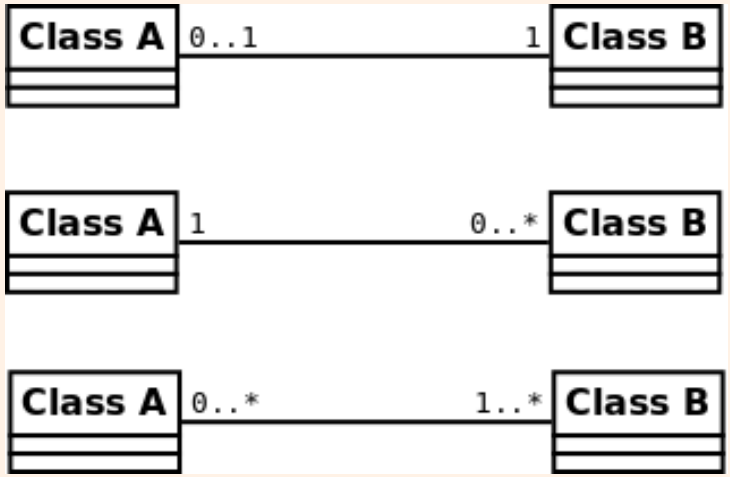
## Modélisation des relations



# Associations entre objets

## Cardinalité de l'association

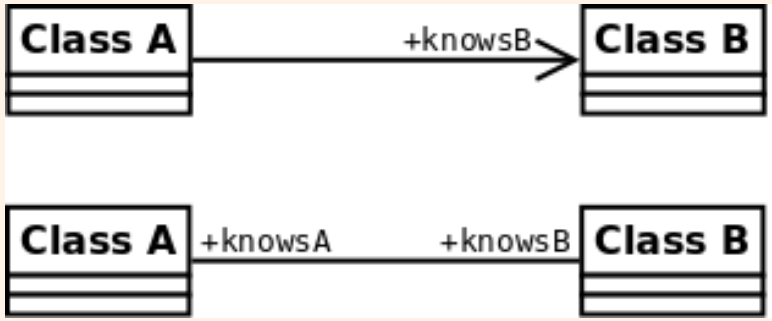
1. **OneToOne** : au plus 1 instance de chaque côté
2. **OneToMany,ManyToOne** : au plus 1 instance d'un côté et au moins 1 de l'autre
3. **ManyToMany** : au moins 1 instance de chaque côté



# Associations entre objets

## Directionnalité de l'assocation

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



# Transformation d'associations

## Modèle Objet

- **OneToOne** : référençe sur l'objet associé
- **OneToMany**,**ManyToOne** : référençe de type collection
- **ManyToMany** : une classe d'association

Référence d'objet naturellement unidirectionnelle

## Modèle relationnel

- **OneToOne,OneToMany,ManyToOne** : une ou plusieurs clés étrangères
- **ManyToMany** : une table de liaison, associative

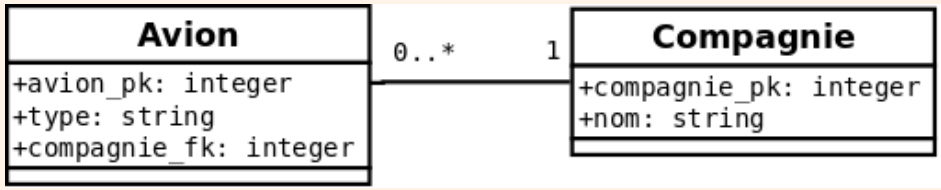
Référence (clé étrangère) naturellement bidirectionnelle (jointure)

# Transformation d'associations

## Association One-to-Many

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

## Exemple d'association One-to-Many



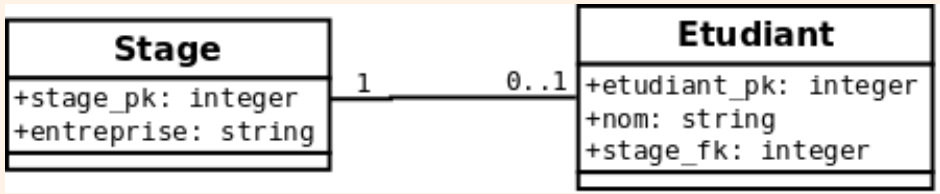

```
COMPAGNIE (compagnie_pk, nom)
AVION (avion_pk, type, #compagnie_fk)
```

# 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



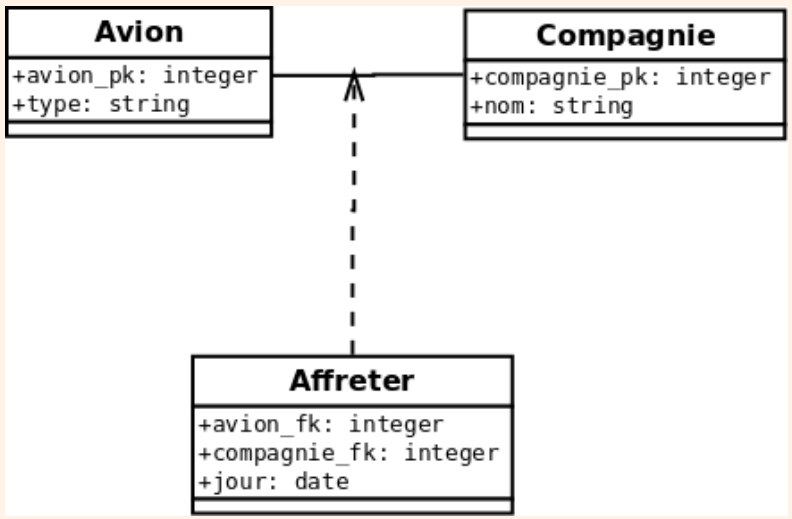

STAGE (stage_pk, entreprise)
ETUDIANT(etudiant_pk,nom,#stage_fk)

# Transformation d'associations

## Association ManyToMany

- clés étrangères : clé primaire d'une table associative

## Exemple d'association ManyToMany




AVION (avion_pk, type)
AFFRETER (#avion_fk, #compagnie_fk, jour)
COMPAGNIE (compagnie_pk, nom)

# QLAlchemy

## 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)

# 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)

# SQLAlchemy

## Composantes principales

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

# Création de tables

## Connexion au SGBDR

```
from sqlalchemy import create_engine, MetaData
engine = create_engine("sqlite:///users.db")
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, 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])
```

# Mapping : Classe-Table

## Création de Classe

```
class User(object):
  def __init__(self, name, age, password):
    self.name=name
    self.age=age
    self.password=password
  def __repr__(self):
    return "<User({},{},{})>".format(self.name,
                                     self.age,
                                     self.password )
```

## Mapping Classe-Table

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

# Mapping : Classe-Table

## Instanciation d'objet

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

## Insertion dans une transaction (Session)

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

# Declarative Base : Classe-Table

## Création 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)
```

# Declarative Base : Classe-Table

## Identification d'objets

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

## Identification des modifications

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

# Declarative Base : Classe-Table

## Gestion de la transaction

```
session.commit()
another_user = User('inconnu',23 'inconnuPWD')
session.add(another_user)
session.rollback()
```

## Modification de la Base de données

```
$ 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
(3 lignes)
```

# Héritage : Exemple

## Classe de base : Employee

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

## Classe héritière : Engineer

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

# Héritage : Exemple

## Classe héritière : Manager

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

## Instanciation d'employées

```
emp = Employee("Dupond")
eng = Engineer("Dupont","developer")
mng = Manager("Durand","business")
```

# Héritage : Transformations

## 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 (join) : 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

# Héritage : Transformations

## SQLAlchemy : propriétes 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

# Joined table mapping (join)

## Classe de base : Employee

```
class Employee(Base):
    __tablename__ = 'employees'
    emp_id = Column(Integer, primary_key=True)
    name = Column(String(50))
    discriminator = Column('type', String(50))
    __mapper_args__ = {
            'polymorphic_on': discriminator,
            'polymorphic_identity': 'employees'
    }
    def __init__(self,name) :
        self.name=name
```

# Joined table mapping (join)

## Héritage : Engineer

```
class Engineer(Employee):
    __tablename__ = 'engineers'
    __mapper_args__ = {
            'polymorphic_identity' : 'engineers'
    }
    eng_id = Column( Integer,
                     ForeignKey('employees.emp_id'),
                     primary_key=True )
    eng_info = Column(String(50))
    def __init__(self,name, eng_info) :
        Employee.__init__(self,name)
        self.eng_info=eng_info
```

# Joined table mapping (join)

## Héritage : Manager

```
class Manager(Employee):
    __tablename__ = 'managers'
    __mapper_args__ = {
            'polymorphic_identity' : 'managers'
    }
    mng_id = Column(Integer,
                    ForeignKey('employees.emp_id'),
                    primary_key=True )
    mng_data = Column(String(50))
    def __init__(self,name, mng_data) :
        Employee.__init__(self,name)
        self.mng_data=mng_data
```

# Joined table mapping (join)

## Côté Relationnel : stockage des employés

```
metadata.create_all(engine)
if __name__ == "__main__" :
    from sqlalchemy.orm import sessionmaker
    Session = sessionmaker(bind=engine)
    session = Session()
    session.add(Employee("Dupond"))
    session.add(Engineer("Dupont","developer"))
    session.add(Manager("Durand","business"))
    session.commit()
```

Quelles seront les instances de tables du côté relationnel ?

# Single table mapping (push-up)

## Classe de base : Employee

```
class Employee(Base):
    __tablename__ = 'employees'
    emp_id = Column(Integer, primary_key=True)
    name = Column(String(50))
    discriminator = Column('type', String(50))
    __mapper_args__ = {
            'polymorphic_on': discriminator,
            'polymorphic_identity': 'employees'
    }
    def __init__(self,name) :
        self.name=name
```

# Single table mapping (push-up)

## Héritage : Engineer

```
class Engineer(Employee):
    __mapper_args__ = {
        'polymorphic_identity' : 'engineers'
    }
    eng_info = Column(String(50))
    def __init__(self,name, eng_info) :
        Employee.__init__(self,name)
        self.eng_info=eng_info
```

# Single table mapping (push-up)

## Héritage : Manager

```
class Manager(Employee):
    __mapper_args__ = {
            'polymorphic_identity' : 'managers'
    }
    mng_data = Column(String(50))
    def __init__(self,name, mng_data) :
        Employee.__init__(self,name)
        self.mng_data=mng_data
```

# Single table mapping (push-up)

## Côté Relationnel : stockage des employés

```
# Employee.__table__.drop(checkfirst=True)
metadata.create_all(engine)
if __name__ == "__main__" :
    from sqlalchemy.orm import sessionmaker
    Session = sessionmaker(bind=engine)
    session = Session()
    session.add(Employee("Dupond"))
    session.add(Engineer("Dupont","developer"))
    session.add(Manager("Durand","business"))
    session.commit()
```

Quelles seront les instances de tables du côté relationnel ?

# Concrete table mapping (push-down)

## Classe de base : Employee

```
class Employee(Base):
    __tablename__ = 'employees'
    emp_id = Column(Integer, primary_key=True)
    name = Column(String(50))
    def __init__(self,name) :
        self.name=name
```

# Concrete table mapping (push-down)

## Héritage : Engineer

```
class Engineer(Employee):
    __tablename__ = 'engineers'
    __mapper_args__ = {
          'concrete':True
     }
    eng_id = Column(Integer, primary_key=True)
    name = Column(String(50))
    eng_info = Column(String(50))
    def __init__(self,name, eng_info) :
        self.name=name
        self.eng_info=eng_info
```

# Concrete table mapping (push-down)

## Héritage : Manager

```
class Manager(Employee):
    __tablename__ = 'managers'
    __mapper_args__ = {
          'concrete':True
     }
    mng_id = Column(Integer, primary_key=True)
    name = Column(String(50))
    mng_data = Column(String(50))
    def __init__(self,name, mng_data) :
        self.name=name
        self.mng_data=mng_data
```

# Concrete table mapping (push-down)

## Côté Relationnel : stockage des employés

```
# Employee.__table__.drop(checkfirst=True)
metadata.create_all(engine)
if __name__ == "__main__" :
    from sqlalchemy.orm import sessionmaker
    Session = sessionmaker(bind=engine)
    session = Session()
    session.add(Employee("Dupond"))
    session.add(Engineer("Dupont","developer"))
    session.add(Manager("Durand","business"))
    session.commit()
```

Quelles seront les instances de tables du côté relationnel ?

# Associations

## Cardinalité de la relation

1. **OneToOne** : maxi de 1 instance de chaque côté
2. **OneToMany,ManyToOne** : maxi de 1 instance d'un côté et de plus d'1 de l'autre
3. **ManyToMany** : plus d'1 instance de chaque côté

## Trois types de mapping

1. **OneToOne** : cas particulier de **One-To-Many**
2. **OneToMany,ManyToOne** : une clé étrangère du côté **Many**
3. **ManyToMany** : une table "secondaire" associative

# Cardinalité : OneToMany

## Association : côté "Many"

```
class Child(Base):
    __tablename__ = 'children'
    id = Column(Integer, primary_key=True)
    parent_id = Column( Integer,
                        ForeignKey('parents.parent_id'))
    name = Column(String, nullable=False)
##    parent = relationship("Parent",
##                          back_populates="children")
    def __init__(self,name) :
        self.name=name
```

- `ForeignKey` : référencer la table `parents`
- `relationship()` : si bidirectionalité
  - référence sur le `parent` (ManyToOne)

# Cardinalité : OneToMany

## Association : côté "One"

```
class Parent(Base):
    __tablename__ = 'parents'
    parent_id = Column(Integer, primary_key=True)
    children = relationship("Child")
##    parent = relationship("Parent", backref="parent")
    name = Column(String, nullable=False)
    def __init__(self,name) :
        self.name=name
```

- `relationship()` : référence sur les `children`
- `backref` : si bidirectionalité non définie sur les children

# Cardinalité : OneToMany

## Côté applicatif : instanciation des objets

```
parent=Parent('Dupont')
parent.children.append(Child('Dupont Junior'))
parent.children.append(Child('Dupont Fils'))
session.add(parent)
```

## Côté Relationnel : stockage des objets

```
SELECT * FROM  parents;
 id |  name
----+--------
  1 | Dupont
SELECT * FROM  children;
 id | parent_id |     name
----+-----------+---------------
  1 |         1 | Dupont Junior
  2 |         1 | Dupont Fils
```

# Cardinalité : OneToMany

## Bidirectionnalité : backref

```
class Parent(Base):
    ...
    children = relationship( 'Child',
                    backref='parent',
                    cascade="save-update, merge, delete"
                )
```

## Bidirectionnalité : mise à jour

```
parent = Parent('Dupont')
child = Child('Dupont Junior')
parent.children.append(child)
...
session.delete(parent) # cascade children destruction
```

# Cardinalité : OneToOne

## OneToOne :cas particulier de OneToMany

- cas particulier de OneToMany ou ManyToOne
- référence simple (`uselist=False`) de l'association.

## OneToOne : association bidirectionnelle

```
class Parent(Base):
    __tablename__ = 'parents'
    parent_id = Column(Integer, primary_key=True)
    child = relationship("Child",
                         uselist=False,
                         back_populates="parent" )
    name = Column(String, nullable=False)
    def __init__(self,name) :
        self.name=name
```

# Cardinalité : OneToOne

## OneToOne : cas particulier de OneToMany

```
class Child(Base):
    __tablename__ = 'children'
    child_id = Column(Integer, primary_key=True)
    parent_id = Column(Integer,
                       ForeignKey('parents.parent_id'))
    name = Column(String, nullable=False)
##    defining bidirectionality from children
    parent = relationship("Parent",
                          back_populates="child")
    def __init__(self,name) :
        self.name=name
```

## Du côté "Child" de la relation

- clé étrangère sur la table `parent`

# Cardinalité : ManyToMany

## ManyToMany : une table associative

- créer une table associative : `Table`
- l'associer aux classes : `relationship()`

## Table Associative : création

```
association_table = Table('association', Base.metadata,
    Column('left_id', Integer, ForeignKey('left.id')),
    Column('right_id', Integer, ForeignKey('right.id'))
)
```

# Cardinalité : ManyToMany

## Table Associative : secondary

```
class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    children = relationship("Child",
                    secondary=association_table,
                    backref="parents")
    def __init__(self,name) :
        self.name=name
```

# Cardinalité : ManyToMany

## Table Associative : secondary

```
class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    def __init__(self,name) :
        self.name=name
```

# Cardinalité : ManyToMany

## Côté applicatif : validation de session

```
parent = Parent('Dupont')
child = Child('Dupont Junior')
parent.children.append(child)
child = Child('Dupont Fils')
parent.children.append(child)
session.add(parent)
parent = Parent('Durand')
child.parents.append(parent)
session.add(child)
```

# Cardinalité : ManyToMany

## Côté Relationnel : stockage des parents,enfants

```
 SELECT * FROM left;     SELECT * FROM right;
parent_id |  name        child_id |  name
----------+---------     ---------+----------------
  1       | Dupont        1       | Dupont Junior
  2       | Durand        2       | Dupont Fils

SELECT * FROM associations;
left_id   | right_id
----------+-----------
1         | 2
2         | 2
1         | 1
```

# Application aux graphes

## Objectifs

- créer un graphe orienté simple (association Nœud-Arête)
- en faire graphe pondéré (Héritage d'Arête)
- en faire graphe coloré (Héritage de Nœud)
- le sauvegarder, le charger etle visualiser dans une application

## Graphe simple orienté

- une arête est constitué de deux noeuds (above,below)
- d'un noeud peut partir plusieurs arêtes (following edges)
- sur un noeud peut arriver plusieurs arêtes (previous edges)

# Application aux graphes

## classe Node

```
class Node(Base) :
  __tablename__="nodes"
  node_id=Column(Integer,primary_key=True)
  name=Column(String(50))
  def __init__(self, name="n"):
    self.name = name
  def get_name(self) :
    return self.name
```

# Application aux graphes

## classe Node

```
def __repr__(self):
  return "<Node({})>".format(self.name)

def above_neighbors(self):
  return [x.node_above for x in self.previous_edges]
def below_neighbors(self):
  return [x.node_below for x in self.following_edges]
```

Références `previous_edges`,`following_edges`

- définies sur la classe `Edge` (`relationship()`)

# Application aux graphes

## classe Edge

```
class Edge(Base) :
  __tablename__="edges"
  edge_id=Column(Integer,
                 primary_key=True )
  above_id=Column(Integer,
                  ForeignKey("nodes.node_id") )
  below_id=Column(Integer,
                  ForeignKey("nodes.node_id") )
```

# Application aux graphes

## classe Edge

```
node_above=relationship(
  Node,
  primaryjoin=above_id==Node.node_id,
  backref="following_edges"
)
node_below=relationship(
  Node,
  primaryjoin=below_id==Node.node_id,
  backref="previous_edges"
)
```

Tests `primaryjoin`

- pour distinguer les deux références sur la classe `Node`

# Application aux graphes

## Côté applicatif : instanciation des objets

```
engine = create_engine("sqlite:///graph.db")
Base.metadata.create_all(engine)
session = sessionmaker(engine)()
n1 = Node("n1")
n2 = Node("n2")
e1=Edge(n1,n2)
n3 = Node("n3")
e2=Edge(n1,n3)
session.add_all([n1,n2,n3])
##session.add_all([e1,e2])
session.commit()
```

# Cardinalité : OneToMany

## Côté Relationnel : stockage des objets

```
SELECT * FROM edges;
edge_id     above_id    below_id
----------  ----------  ----------
1           1           2
2           1           3
SELECT * FROM nodes;
node_id     name
----------  ----------
1           n1
2           n2
3           n3
```

# Bibliographie

## Livres


- **J. Myers, R. Copeland** : “Essential SQLAlchemy 2e” Éditions O’Reilly (2015)
- **C.Soutou** : “UML 2 pour les bases de données” Éditions Eyrolles(2007)


## Adresses “au Net”

- `www.sqlalchemy.org` : le site officiel
- `www.agiledata.org/essays/mappingObjects.html` : ORM in details
- `www.hibernate.org`
- `www.oracle.com` : TopLink JPA (Java Persistence API)