Projektovanje veb aplikacija
- uzorci za podatke -

Principi softverskog inženjerstva, *Elektrotehnički fakultet Univerziteta u Beogradu*

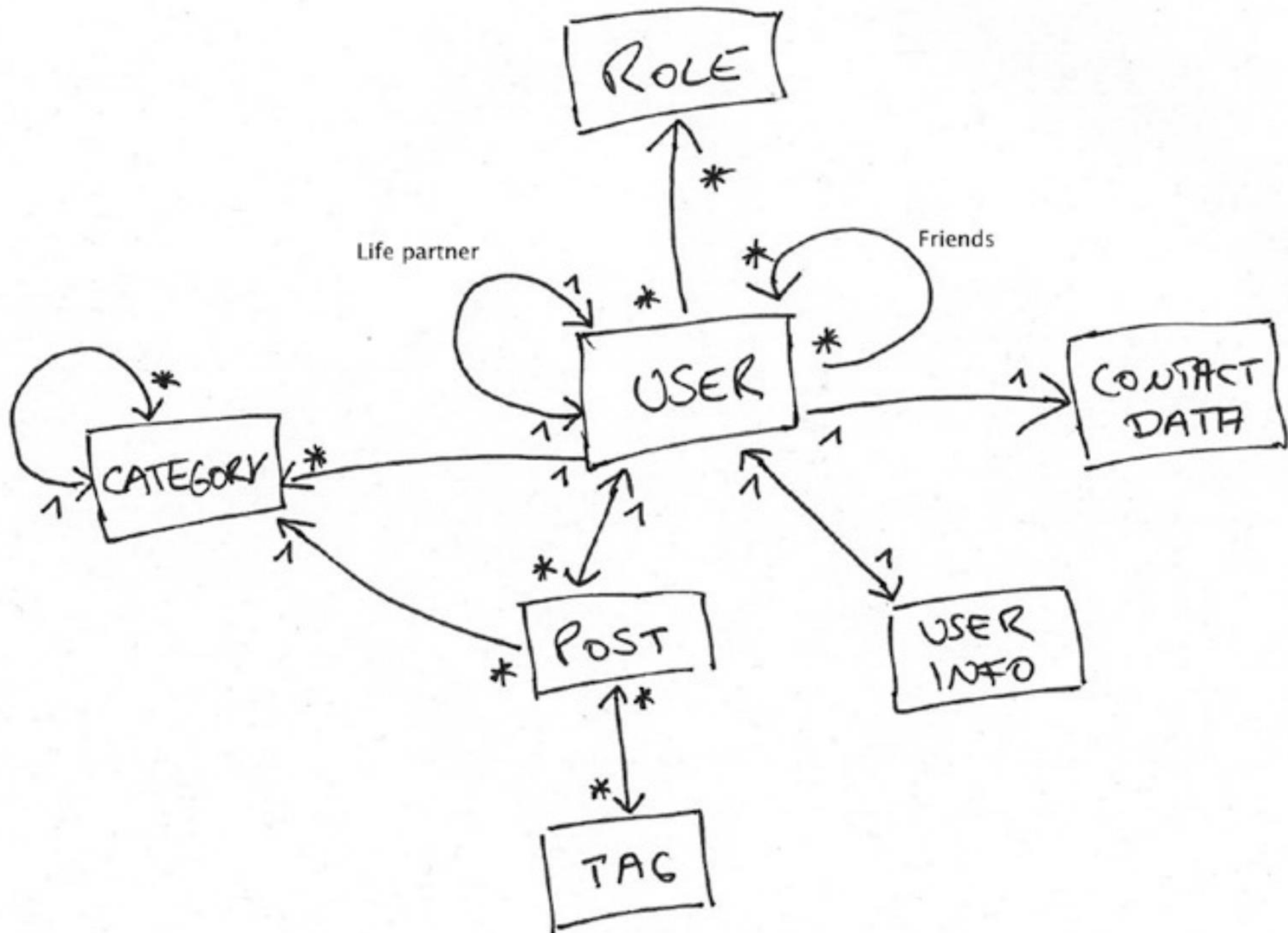
Uvod

- Konceptualni model aplikacije u objektno orijentisanom projektovanju realizije se u vidu takozvanog domenskog modela koji obuhvata i podatke (koji se čuvaju u bazi) i ponašanje (funkcije).
- **Domenski model** aplikacije obuhvata:
- Klase i objekte koji predstavljaju glavne koncepte domena, takozvane **entitete**. U internet prodavnici, glavni elementi bi biti „Kupac“, „Narudžba“, „Proizvod“, „Korpa“ i tako dalje. Domenski specifične klase i objekti se koriste umesto generičkih kolekcija podataka, kao što su PHP nizovi kad god je to moguće.
- Srodni entitetima su takozvani **vrednosni objekti**, ali u poređenju sa entitetima, objekti vrednosti nemaju postojani identitet, u smislu primarnog ključa u bazi podataka (njihovo glavno svojstvo je vrednost, na tome se zasniva poređenje jednakosti dva objekta).
- Međusobne **veze** (**asocijacije**) između domenskih klasa i objekata. U primeru online prodavnice, jedna porudžbina bi imala najmanje jednog Kupca, kao i jednu ili više referenci na naručene proizvode.
- **Poslovnu logiku** implementiranu u okviru domenskih objekata aplikacije kad god je to moguće, a ne na primer, u kontrolerima (kao deo MCV šablona). U internet prodavnici Korpa, na primer, može imati metod izracunajUkupnuCenu(). na osnovu artikala u Korpi i njihove količine.
- Funkcije koje obuhvataju više entiteta obično se realizuju kao tzv. **servisi**, jednostavno zato što se njima ne može jasno dodieliti jedan jedini entitet. U online prodavnici servis „Checkout“ vodi računa o smanjenju zaliha, fakturisanju, ažuriranju istorije narudžbina itd. Usluga se bavi sa više entiteta odjednom.

Prednosti korišćenja domenskog modela

- Odvajanje dela aplikativnog koda koji je specifičan za konkretnu primenu (dakle domenski specifičnog koda) od ostatka koda koji rešava generalne tehničke probleme (kao što je prikaz GUIja, loging, zaštita podataka, slanje pošte itd) olakšava održavanje koda i njegovo menjanje, povećava prenosivost koda (na primer, mogućnost njegvog prebacivanja na novi okvir).
- Promoviše timski rad i između programera i korisnika, menadžera i marketinga. Domenski model prevazilazi razlike u razmišljanju inženjera i ne-inženjera tako što unifikuje terminologiju.

Tekući primer ovog predavanja



Tekući primer – domenski model

- U demo aplikaciji Talking, korisnik može pisati postove. Post uvek ima samo jednog autora (Korisnika). Korisnik (User) može biti u jednoj ili više uloga, ali nije predviđeno da se vrši pretraga korisnika koji imaju određenu ulogu.
- Korisnik može da pristupi zapisu UserInfo koji sadrži datum registracije i datum odjavljivanja, ako su dostupni. Na osnovu UserInfo podataka mogu se pretraživati korisnici.
- Korisnik pristupa zapisu ContactData koji čuva Korisnikov e-mail i telefonski broj. Na osnovu ContactData ne pretražuju se korisnici.
- Korisnik može navesti drugog korisnika kao svog životnog partnera (relacija LifePartner). Korisnikov životni partner je dvosmerna relacija.
- Korisnik može imati neograničen broj prijatelja. Za zadatog korisnika, mogu se pretražiti njegovi prijatelji, ali nije predviđena pretraga u suprotnom smeru.
- Postovi korisnika mogu imati neograničen broj oznaka (Tags). Oznaka se može ponovo koristiti u nekoliko postova. Između Posta i njegovih Oznaka postoji dvosmerno povezivanje.

Tekući primer – domenski model

- Post referiše svoju Kategoriju, međutim, iz Kategorije nije predvieno referisanje njenih postova. Kategorija može imati potkategorije, kao i nadređenu Kategoriju, ako je zadata. Kategorije su korisnički specifične. Korisnik navodi svoje kategorije, ali nema povratnog referisanja, od Kategorije ka Korisniku.
- Ovo je relativno mali model, ali reprezentativan jer poseduje različite vrste relacija među entitetima: 1:1, 1:N, M:N, unidirekcione, bidirekcione i samoreferišuće relacije (sa obe strane relacije je isti entitet).
- Dodatno ćemo u model ubaciti relaciju (jednostrukog) nasleđivanja, kroz razradu da post može biti čisto tekstualni ali može imati i dodatak u vidu ili audio ili video zapisa (ne i jednog i drugog). Dakle Post je nadklasa, a ona se proširuje u dve podklase: AudioPost i VideoPost, gde se na polja osnovne klase, dodaje polje audioUrl, odnosno videoUrl.

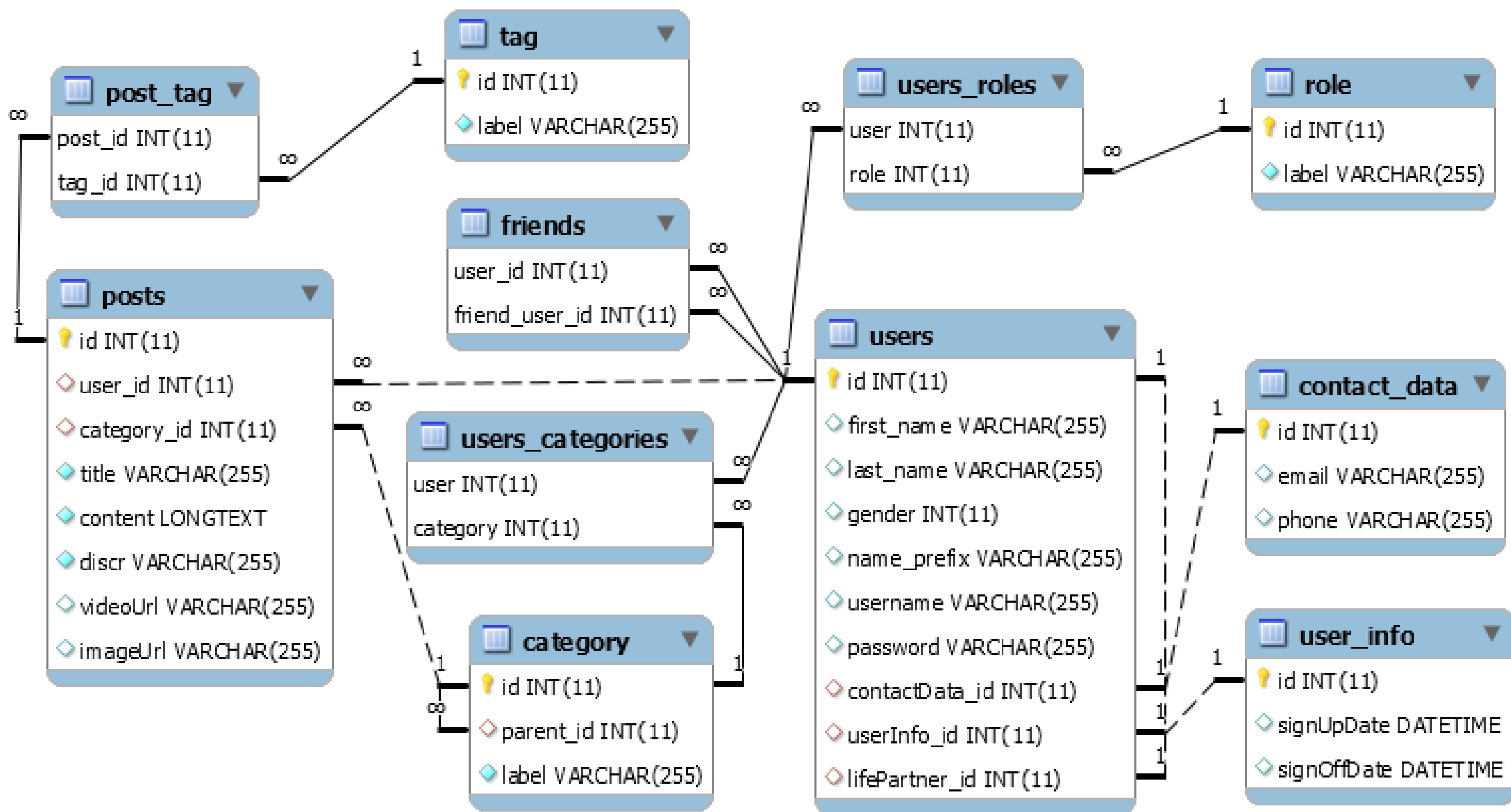
Tekući primer – slučajevi korišćenja

- Pošto je primarni cilj ove aplikacije ilustracija projektovanja sloja podataka, definisani su samo neki osnovni scenariji rada aplikacije:
  - Na osnovnoj strani prikazuje se spisak sažetaka postova sa naslovom, prva dva reda sadržaja, autorom i dugmetom za čitanje punog posta
  - Strana punog posta prikazuje izabrani post naslovom, punim sadržajem, autorom, spiskom tagova ako su definisani (hiperlink koji vodi na stranu sa spiskom postova za dati tag), kategorijom i akcijama nad postom Edit i Delete.
  - Strana za ažuriranje posta prikazuje formu u kojoj se mogu izmeniti naslov i sadržaj postjećeg posta i ima dugme Add Post za čuvanje izmena.
  - U zaglavlju aplikacije postoje linkovi za osnovnu stranu (Talking), za podatke o tekućem korisniku (My posts&Profile) i dodavanje novog posta.
  - Na strani My Posts&Profile prikazuje se ime korisnika, lista naslova njegovih postova, lista prijatelja i ime životnog partnera i hijerarhijska lista kategorija koje je korisnik definisao (ako je definisao). Naslovi postova i životni partner su hiperlinkovi, vode na odgovarajuće strane.
  - N strani za dodavanje novog posta prikazuje se forma u kojoj se mogu definsati naslov i sadržaj i ima dugme Add Post za čuvanje unetog.

Tekući primer - realizacija

- Primer je realizovan u dve varijante:
  - django aplikacija gde su podaci predstavljeni django modelima. Ova realizacija je primer korišćenja projektnog obrasca **Aktivni zapis** (eng. **Active Record**).
  - flask aplikacija, model realizovan sa SqlAlchemy ORM-om. Ova realizacija je primer korišćenja projektnog obrasca **Mapiranje podataka** (eng. **Data Mapper**).
- Ove aplikacije nalaze se na sajtu predmeta kao prilog predavanju.
- ORM je skraćenica od **Object Relational Mapping** tj. objektno-relaciono mapiranje. Ovi sistemi rešavaju problem kako zapamtiti (ostvariti perzistenciju) domenski model koji se realizuje u vidu grafova međusobno povezanih objekata, u relacionoj bazi koja se realizuje u vidu kolekcije međusobno povezanih relacionih tabela.
- Za svaku od gornjih varijanti u nastavku će biti opisani koncepti projektnog obrasca i konkretno korišćenje na primeru *Talking* aplikacije: kreiranje objekata, osnovni CRUD upiti, definisanje i korišćenje relacija i transakciona obrada.

Relaciona šema baze Talking aplikacije



OBRAZAC QUERY BUILDER (GRADITELJ UPITA)

Projektni obrazac Graditelj upita

- Graditelj upita pruža pogodan, "tečan" (engl. fluent) interfejs za kreiranje i pokretanje upita u bazi podataka. Može se koristiti za obavljanje većine operacija sa bazom podataka u aplikaciji kao alternativa pisanja upita na jeziku SQL.
- Tečan interfejs je objektno-orijentisani idiom koji se zasniva na ulančavanju poziva metoda nad istim objektom (što je omogućeno time što svaki metod vraća this). Cilj je povećavanje čitljivosti programa i kreiranje domenski specifičnog jezika.
- Primer:

```
pip install simple-query-builder

from simple_query_builder import *
qb = QueryBuilder(DataBase(), 'my_db.db')
results = qb.select({'b': 'branches'}, ['b.id', 'b.name'])
    .where([['b.id', '>', 1], 'and', ['b.parent_id', 1]])
    .order_by('b.id desc')
    .all()
```

Projektni obrazac Graditelj upita

- Poziv qb.get_sql() vraća string sa ekvivalentnim SQL upitom:

```
SELECT `b`.`id`, `b`.`name` FROM `branches` AS `b` WHERE (`b`.`id` > ?) AND (`b`.`parent_id` = ?) ORDER BY `b`.`id` DESC;
```

- U nekim primenama, kada se poveže sa obrascem mapiranja podataka (data mapper) upit može da se odnosi na objekte i polja a ne na relacione tabele i kolone. Na ovaj način, oni koji pišu upite mogu to činiti nezavisno od šeme baze i promena u šemi može biti lokalizovana na jednom mestu.

- Fowler u svojoj knjizi PoEAA opisuje obrazac Upitni objekat (Query Object) koji je sa stanovišta korišćenja praktično isti kao graditelj upita, jedino što predlaže nešto drugačiju realizaciju u vidu Gof obrasca Interpreter.

- Sledi primer realizacije jednostavnog graditelja SQL upita, preuzet sa https://github.com/co0lc0der/simple-query-builder-python

Jednostavan graditelj upita (izvodi)

```
class MetaSingleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(MetaSingleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]


class DataBase(metaclass=MetaSingleton):
    db_name = 'db.db'
    conn = None
    cursor = None

    def connect(self, db_name=''):
        self.conn = sqlite3.connect(self.db_name)
        self.cursor = self.conn.cursor()
        return self.conn

    def c(self):
        return self.cursor
```

Jednostavan graditelj upita (izvodi)

```
class QueryBuilder:
    _conn = None
    _cur = None
    _query = None
    _sql: str = ''
    _error: bool = False
    _result: Union[tuple, list] = []

    def __init__(self, database: DataBase, db_name='') -> None:
        self._conn = database.connect(db_name)
        self._cur = database.c()

    def query(self, sql: str = '', params=(), fetch=2, column=0):
        try:
            self._query = self._cur.execute(self._sql, self._params)
            if fetch == self._NO_FETCH:
                self._conn.commit()
            elif fetch == self._FETCH_ONE:
                self._result = self._query.fetchone()
            elif fetch == self._FETCH_ALL:
                self._result = self._query.fetchall()
            self.set_error()
        except sqlite3.Error as er:
            self._error = True
            print('SQLite error: %s' % (' '.join(er.args)))
        return self
```

Jednostavan graditelj upita (izvodi)

```
def get_sql(self) -> str:
    return self._sql
def get_error(self) -> bool:
    return self._error

def all(self) -> Union[tuple, list, dict, None]:
    self.query()
    return self._result
def go(self) -> Union[int, None]:
    self.query(self._sql, self._params, self._NO_FETCH)
    return self._cur.lastrowid
def one(self) -> Union[tuple, list, dict, None]:
    self.query(self._sql, self._params, self._FETCH_ONE)
    return self._result

def select(self, table: Union[str, dict], fields: Union[str, list, dict] = '*'):
    if isinstance(table, dict) or isinstance(table, str):
        self._sql += f" FROM {self._prepare_aliases(table)}"
    else:
        self.set_error(f"Incorrect type of table must be String or Dictionary")
    return self
def where(self, where: Union[str, list]'):
    conditions = self._prepare_conditions(where)
    self._sql += f" WHERE {conditions['sql']}"
    return self
```

Jednostavan graditelj upita

- Metaclass mehanizmom pythona obezbeđeno je da ne pravimo više instanci Database klase, to jest realizacija singleton objekta (recikliranje konekcija ka bazi). Konekcija se otvara instanciranjem QueryBuilder objekta, koji može imati više instanci.

- U objektu QueryBuildera se čuva kontekst u vidu stringa _sql koji čine delove SQL upita. Kontekst to jest string se ažurira svakim pozivom nekog od metoda kao što su select(), where() itd. Svaki metod vraća instancu QueryBuilder objekta da bi pozivi mogli da se ulančaju. Na kraju se rezultujući upit dobija konkatenacijom finalnih stringova.

- Metodi kao što su all(), one() služe da završe formiranje sql stringa i pozovu query() koji realizuje upit nad bazom. Metod go() služi u slične svrhe, ali za upite koji ne vraćaju podatke iz baze (insert, delete itd).

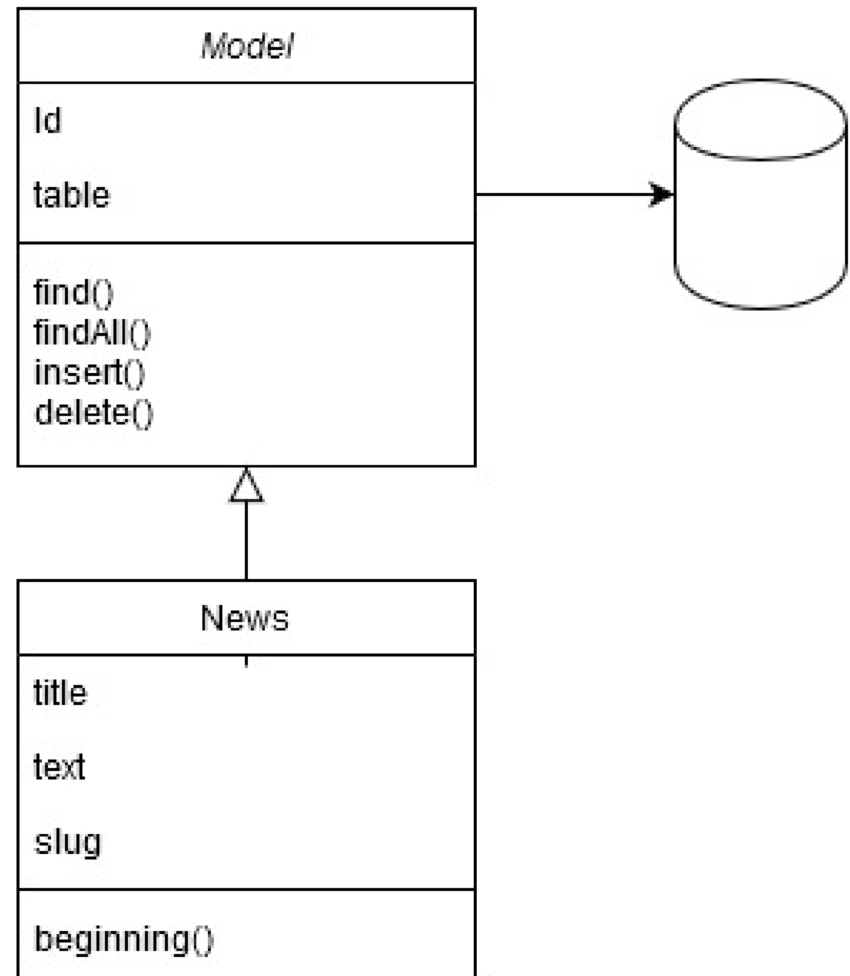
OBRAZAC ACTIVE RECORD (AKTIVNI ZAPIS)

Uzorak aktivni zapis

- Aktivni zapis (eng. Active Record) je arhitektonski projektni obrazac za čuvanje podataka o objektu u memoriji u relacionoj bazi podataka. Ime je smislio Martin Fouler u svojoj knjizi Patterns of Enterprise Application Architecture. Interfejs objekta u skladu s ovim obrascem uključuje funkcije kao što su kreiranje, umetanje, ažuriranje i brisanje objekta u bazi (CRUD), plus svojstva koja manje ili više direktno odgovaraju kolonama u tabeli baze podataka.
- Prema ovom obrascu instanca objekta vezana za jedan red u tabeli. Nakon kreiranja objekta, novi red se dodaje u tabelu. Svaki učitani objekt dobija svoje podatke iz baze podataka. Kada se objekt ažurira, odgovarajući red u tabeli se takođe ažurira. Klasa Model koja “umotava” pristup bazi implementira pristupne metode ili svojstva za svaku kolonu u tabeli ili prikazu.
- Pored ovog osnovnog ponašanja, objekat može dodatno da implementira sopstveno ponašanje (domenski specifično), a u nekim sistemima objektno-relacionog mapiranja, mi ćemo proučiti Eloquent kao primer, dodaju se relacije među objektima, graditelj upita itd.

Jednostavna realizacija aktivnog zapisa

- Model je klasa koja obezbeđuje osnovno ponašanje: mapiranje stanja objekta na red u tabeli baze podataka i CRUD operacije.
- Iz nje izvodimo konkretne modele za naše domenske objekte koji mogu dodati specifično ponašanje.
- U našem primeru, Vest ima osobine naslov, tekst, slug-identifikator za Wordpress sajt. Takođe ima i specifični metod beginning() koji vraća prvih 60 znakova teksta vesti.



Jednostavna realizacija aktivnog zapisa

- Za ilustraciju koncepta može poslužiti https://github.com/xzmeng/tiny-orm (pip install tiny-orm)

Primer korišćenja tiny-orm:

- definiše se model

```
# post.py
from tiny_orm import Model

class Post(Model):

    text = str  # other datatypes: int, float

    def __init__(self, text):
        self.text = text
```

- Stvara se objekat db sa pristupanje bazi:

```
from tiny_orm import Database
db = Database('db.sqlite')  # indicating a database file.
```

- Importovanje Post modela i povezivanje sa bazom:

```
from post import Post
Post.db = db
```

Jednostavna realizacija akt. zapisa (2)

Primer korišćenja tiny-orm:

- Kreira se novi post i smešta u “staging” memoriju (bez komita u bazu).

```
post = Post('Hello World').save()
print(post.id)  # auto generated id
(ispisuje 1)
```

- Update ovog post objekta u staging-u i commit objekta u bazu:

```
post.text = 'Hello Mundo'
post.update()
db.commit()
```

- Za dohvaranje iz baze i CRUD operacije koristi se manager:

```
objects = Post.manager(db)
objects.save(Post('Hello', 'World2')) # novi post
print(objects.get(2))  # get by id from the staging area
#Ispis: {'text': 'World2', 'id': 2, 'title': 'Hello'}
db.close() # ako se prethodnoo ne uradi commit izmene nisu upisane
```

- objects.all() dohvata listu svih postova u bazi.

Realizacija aktivnog zapisa (izvodi)

```
import sqlite3

#: Dictionary to map Python and SQLite data types
DATA_TYPES = {str: 'TEXT', int: 'INTEGER', float: 'REAL'}

def attrs(obj):
    """ Return attribute values dictionary for an object """
    return dict(i for i in vars(obj).items() if i[0][0] != '_')

def render_column_definitions(model):
    """ Create SQLite column definitions for an entity model """
    model_attrs = attrs(model).items()
    model_attrs = {k: v for k, v in model_attrs if k != 'db'}
    return ['%s %s' % (k, DATA_TYPES[v]) for k, v in model_attrs.items()]


def render_create_table_stmt(model):
    """ Render a SQLite statement to create a table for an entity model """
    sql = 'CREATE TABLE {table_name} (id integer primary key autoincrement, {column_
    column_definitions = ', '.join(render_column_definitions(model))
    params = {'table_name': model.__name__, 'column_def': column_definitions}
    return sql.format(**params)
```

Realizacija aktivnog zapisa (izvodi)

```
class Database(object):
    """ Proxy class to access sqlite3.connect method """

    def connection(self):
        """ Create SQL connection """
        if self.connected:
            return self._connection
        self._connection = sqlite3.connect(*self.args, **self.kwargs)
        self._connection.row_factory = sqlite3.Row
        self.connected = True
        return self._connection

    def commit(self):
        """ Commit SQL changes """
        self.connection.commit()

    def execute(self, sql, *args):
        """ Execute SQL """
        return self.connection.execute(sql, args)
```

Realizacija aktivnog zapisa (izvodi)

```python
class Manager(object):
    """ Data mapper interface (generic repository) for models """
    def __init__(self, db, model, type_check=True):
        self.db = db
        self.model = model
        self.table_name = model.__name__
        self.type_check = type_check
        self.db.executescript(render_create_table_stmt(self.model))

    def all(self):
        """ Get all model objects from database """
        result = self.db.execute('SELECT * FROM %s' % self.table_name)
        return (self.create(**row) for row in result.fetchall())

    def create(self, **kwargs):
        """ Create a model object """
        obj = object.__new__(self.model)
        obj.__dict__ = kwargs
        return obj

    def delete(self, obj):
        """ Delete a model object from database """
        sql = 'DELETE from %s WHERE id = ?'
        self.db.execute(sql % self.table_name, obj.id)
```

Realizacija aktivnog zapisa (izvodi)

```
def get(self, id):
       """ Get a model object from database by its id """
       sql = 'SELECT * FROM %s WHERE id = ?' % self.table_name
       result = self.db.execute(sql, id)
       row = result.fetchone()
       if not row:
         msg = 'Object%s with id does not exist: %s' % (self.model, id)
           raise ValueError(msg)
       return self.create(**row)


   def save(self, obj):
       """ Save a model object """
       clone = copy_attrs(obj, remove=['id'])
       self.type_check and self._isvalid(clone)
       column_names = '%s' % ', '.join(clone.keys())
       column_references = '%s' % ', '
               .join('?' for i in range(len(clone)))
       sql = 'INSERT INTO %s (%s) VALUES (%s)'
       sql = sql % (self.table_name, column_names, column_references)
       result = self.db.execute(sql, *clone.values())
       obj.id = result.lastrowid
       return obj
```

Realizacija aktivnog zapisa (izvodi)

```
class Model(object):
    """ Abstract entity model with an active record interface """
    db = None

    def delete(self, type_check=True):
        """ Delete this model object """
        return self.__class__.manager(type_check=type_check).delete(self)

    def save(self, type_check=True):
        """ Save this model object """
        return self.__class__.manager(type_check=type_check).save(self)

    def update(self, type_check=True):
        """ Update this model object """
        return self.__class__.manager(type_check=type_check).update(self)

    @property
    def public(self):
        """ Return the public model attributes """
        return attrs(self)

    @classmethod
    def manager(cls, db=None, type_check=True):
        """ Create a database managet """
        return Manager(db if db else cls.db, cls, type_check)
```

DJANGO ORM

- primer korišćenja aktivnog zapisa -

Django ORM – kreiranje entiteta

- Svaki model (entitet koji će se čuvati u bazi) predstavljen je klasom koja je podklasa django.db.models.Model. Svaki model ima određeni broj polja klase, od kojih svaka predstavlja polje baze podataka u modelu.

- Svako polje je predstavljeno instancom klase Field(ne navodi se primarni ključ koji je po konvenciji id).

- Model može sadržati i proizvoljne metode domenski specifične logike koji obrađuju podatke modela. U primeru klase User metod assembleDisplayName pravi string reprezentaciju imena korisnika sa prefiksom (gdin ili dja) koji zavisi od pola.

Django ORM – kreiranje entiteta

- Primer klase User iz aplikacije Talking. Nisu prikazane relacije, to ćemo naknadno specifikovati u modelu.

```
# djtalk/models.py
from django.db import models

class User(models.Model):
    FEMALE = 1
    MALE = 2
    UNSPECIFIED = 0
    GENDER_CHOICES = [(UNSPECIFIED, 'Unspecified'), (FEMALE, 'Female'), (MALE, 'Male'), ]
    first_name = models.CharField('First name', max_length=255)
    last_name = models.CharField('Last name', max_length=255)
    gender = models.IntegerField(default=UNSPECIFIED, choices = GENDER_CHOICES)
    name_prefix = models.CharField('Salutation', max_length=255)
    username = models.CharField('User name', max_length=255)
    password = models.CharField(max_length=255)
    . . . .
    def assembleDisplayName(self):
        displayName = ''
        if self.gender == self.MALE:
            displayName += 'Mr.'
        if self.gender == self.FEMALE:
            displayName += 'Mrs.'
        if self.name_prefix != '':
            displayName = self.name_prefix
        displayName += ' ' + self.first_name + ' ' + self.last_name
        return displayName
```

Django ORM – kreiranje entiteta

- Da bi se promene u modelu prenele u bazu (kreirale ili promenile šemu tabela u bazi), potrebno je kreirati migraciju (opis promena šeme) a zatim primeniti migraciju nad bazom. Dakle generalno se ponavljaju sledeća tri koraka:
- Editovanje modela u **models.py**
- Komanda **python manage.py makemigrations**
  pravi odgovarajući python skript, inicijalno je to 0001_initial.py
- Komanda **python manage.py migrate**
  izvršava najsvežiji migracioni skript i pravi promene šeme baze bez brisanja podataka
- Za inicijalne podatke u bazi može se prvo napraviti prazan migracioni skript komandom: **python manage.py makemigrations --empty yourappname**
  a zatim u njemu napisati funkcija def init_data(apps, schema_editor)
- Pogledati u primeru projekta migracioni skript 0002
- Da bi komande radile, potrebno je da u settings.py fajla django sajta postoji linija:

```
INSTALLED_APPS = [
        'djtalk.apps.DjtalkConfig',
```

- a da u djtalks/apps.py postoji klasa DjtalkConfig izvedena iz AppConfig sa poljem name = “djtalk”

Django ORM – kreiranje entiteta

- Da bismo pregledali sql kod nekog migracionog skripta, npr. 0001_initial.py bez izmena baze, zadajemo komandu
- **python manage.py sqlmigrate djtalk 0001**

```
BEGIN;
--
-- Create model User
--
CREATE TABLE "djtalk_user" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
"first_name" varchar(255) NOT NULL, "last_name" varchar(255) NOT NULL,
"gender" integer NOT NULL, "name_prefix" varchar(255) NOT NULL,
"username" varchar(255) NOT NULL, "password" varchar(255) NOT NULL);
COMMIT;
```

Django ORM – CRUD operacije

Kreiranje novog objekta prema model klasi.

```
from .models import Post, User

    post = Post()
    post.title = 'title'
    post.content = 'text'
    post.user = User.objects.get(id=1)
    post.save()
```

- Ovaj deo koda tipično se nalazi u modulu views.py. Django pogledi odgovaraju kontrolerima MVC projektnog uzorka.
- Imena modela koji koristimo moramo importovati iz models.py
- Pravi se novi objekat entiteta, postave polja na odgovarajuće vrednosti i pozivom save() metoda vrši njegov upis u povezanu tabelu baze.
- Frejmvork u pozaidini pristupa bazi (podrazumevano sqlite). Pristupni parametric se podešavaju u **djsite\settings.py**

Django ORM – CRUD operacije

Učitavanje iz baze postojećeg objekta

- Upite je sem u aplikacijii, moguđe zadavati i u Django shell-u
- Učitavanje svih objekata tipa Post (dobija se kolekcija kroz koju je moguće normalno iterirati:

```
C:\djsite>python manage.py shell

(InteractiveConsole)
>>> from djtalk.models import Post
>>> posts = Post.objects.all()
>>> for post in posts:
...   print(post.title)
...   print(post.content)
```

Django ORM – CRUD operacije

Učitavanje iz baze postojećeg objekta

- Nalaženje jednog objekta prema primarnom ključu:

```
>>> from djtalk.models import Post
>>> post = Post.objects.get(id=1)
>>> print (post.title)
```

- Nalaženje upitom sa uslovom (za detalje pogledat Django dokumentaciju Making queries). Sa all() se dohvataju svi objekti, sa filter(uslov) oni koji zadovoljavaju, a sa exclude(uslov) oni koji ne zadovoljavaju uslov, sa get(uslov) jedan određeni objekat. Filteri mogu da se ulančavaju da se specifikuje složeni uslov.
- Osnovni uslov je **field__lookuptype=value**

```
>>> from djtalk.models import Post
>>> post = Post.objects.filter(user__first_name = 'Petar').exclude(title__contains = 'computing')
>>> post
<QuerySet [<Post: Post object (2)>]>
>>> post[0].title
'Jurassic Park'
```

Django ORM – CRUD operacije

Izmena postojećeg entitetskog objekta

- dovoljno je promeniti vrednosti polja i zatim pozvati save() kada treba upisati izmene u bazu. Napomena: ako treba sačuvati i objekat i njegovu M:N relaciju onda se koristi add() umesto save().

```
>>> from djtalk.models import Post
>>> post = Post.objects.get(title="Proba")
>>> post.title = "Novi naslov"
>>> post.save()
```

Brisanje entitetskog objekta

- Kada je entitet prethodno pronađen (učitan), uklanjanje se vrši pozivom metoda delete() :

```
>>> post.delete()
```

Django ORM – relacije

- U domenskom modelu Talking aplikacije definisali smo niz relacija među entitetima. Relacije se karakterišu **kardinalnošću** i **usmerenjem**. Prema kardinalnosti, relacije mogu biti 1:1 (na primer, eniteti user i user_info), 1:N (na primer, entiteti user i post), M:N (entiteti Post i Tag). Prema usmerenju, relacije mogu biti jednosmerne (navigabilne sa jedne strane, npr. od User-a može pristupiti kategorijama koje je definisao, ali obrnuto ne očekujemo) i dvosmerne (navigabilne sa obe strane), na primer, možemo dobiti sve Postove nekog Usera, ali i za određeni Post možemo dobiti njegovog autora.
- Objektima entiteta možemo pristupati i preko više relacija, na primer, za korisnika možemo naći njegovog prijatelja, a zatim naći post tog prijatelja i njihove tagove.
- Django pruža jednostavan način za definisanje različitih relacija 1:1, 1:N, M:N i drugih i u principu sve te relacije su navigabilne sa obe strane.
- Nasleđivanje je takođe jedna od relacija među entitetima. Django ima direktnu podršku za nekoliko vrsta nasleđivanja.

Django ORM – definisanje relacija

1:1 relacija (User ⇔ UserInfo)

- u model UserInfo uvodi se OneToOneField, navodi se klasa u relaciji User i da će polje koje realizuje relaciju a koje će po konvenciji biti nazvano user_id ujedno biti primarni ključ tabele UserInfo.

```
class UserInfo(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE,
            primary_key=True)
    signUpDate = models.DateTimeField('Date published')
    signOffDate = models.DateTimeField('Date published')
```

```
--
-- Create model UserInfo
--
CREATE TABLE "djtalk_userinfo" ("user_id" bigint NOT NULL PRIMARY KEY
REFERENCES "djtalk_user" ("id") DEFERRABLE INITIALLY DEFERRED,
"signUpDate" datetime NOT NULL, "signOffDate" datetime NOT NULL);
```

Django ORM – definisanje relacija

1:N relacija (User ⬄ Post)

- Za definisanje ove relacije uvodimo polje tipa ForeignKey u model Post. Parametar je ime modela User u relaciji.

```
class Post(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
```

- Izaziva generisanje stranog ključa u tabelu post radi podrške relaciji

```
CREATE TABLE "djtalk_post" (...
"user_id" bigint NOT NULL REFERENCES "djtalk_user" ("id")
DEFERRABLE INITIALLY DEFERRED);
```

Django ORM – definisanje relacija

M:N relacija (Tag ⇔ Post)

- Za definisanje ove vrste relacija koristi se ManyToManyField u jednom od modela sa navođenjem modela sa kojim se pravi relacija. Nije važno u koji od modela se stavlja ovo polje.

```
class Post(models.Model):
    tag = models.ManyToManyField(Tag)
```

- Automatski se generiše tabela post_tag radi podrške relaciji

```
CREATE TABLE "djtalk_post_tag" (
"id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
"post_id" bigint NOT NULL REFERENCES "djtalk_post" ("id")
DEFERRABLE INITIALLY DEFERRED,
"tag_id" bigint NOT NULL REFERENCES "djtalk_tag" ("id")
DEFERRABLE INITIALLY DEFERRED);
```

Django ORM – definisanje relacija

Inverzna stana relacije

- Ako je definisana neka relacija, na primer

```
class Post(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
```

- onda se može pristupiti iz objekta klase User svim postovima notacijom **user.post_set.all()**,
  dakle ime povezanog modela uz dodato _set.
- Ako se želi posebno imenovati reverzna strana, u originalnom ForeignKey ili ManyToMany polju dodati
  related_name=‘zeljeno_ime’

Ostale vrste relacija

- Primer Talking aplikacije poseduje još nekoliko zanimljivih vrsta relacija. Postoje samo-referišuće relacije myFriends i lifePartner nad entitetom User. Preporučuje se njihovo samostalno proučavanje u kodu aplikacije.

Django ORM – nasleđivanje

- Postoje tri načina da se (jednostruko) nasleđivanje entiteta realizuje u Doctrine-u:

1. **Multi table inheritance** – u ovoj varijanti u tabeli natklase se pamti sadržaj natklase i sopstvena polja natklase. Svaka potklasa ima dodatnu tabelu čiji je primarni ključ ujedno i strani ključ iz tabele natklase i koja čuva polja specifična za potklasu. Da bi se došlo do svih polja i nasleđenih i sopstvenih, moraju se raditi spajanja nad tabelama.

- **Apstraktna bazna klasa** – u ovoj varijanti natklasa nije entitet nego samo klase koje u python stilu nasleđuju iz nje (stanje tj. polja i informacije o mapiranju stanja u bazu). Drugim rečima, samo za potklase postoje tabele u bazi i njihov sadržaj je takav kao da su nasleđena polja i relacije definisane direktno nad potklasama. U baznu klasu se dodaje deklaracija:

```
class Meta:
    abstract = true
```

- **Proksi klase** – ograničena vrsta nasleđivanja gde se u bazi pamti samo bazna klasa, a proksi klase mogu dodati npr. samo metode

Django ORM – nasleđivanje

- Primer iz aplikacije Talking nasleđivanja sa višestrukim tabelama: model Post i izvedeni modeli VideoPost i ImagePost.
- Bazni model Post se ne menja.
- Model VideoPost izveden je iz modela Post i ima sopstveno polje videoUrl, ostalo nasleđuje. Django automatski ubacuje post_ptr_id strani ključ ujedno i primarni
- Analogno tome se definiše i klasa ImagePost.

```python
class VideoPost(Post):
    videoURL = models.CharField(max_length=255)

class ImagePost(Post):
    imageURL = models.CharField(max_length=255)
```

```sql
CREATE TABLE "djtalk_imagepost" ("post_ptr_id" bigint NOT NULL PRIMARY KEY
REFERENCES "djtalk_post" ("id") DEFERRABLE INITIALLY DEFERRED,
"imageURL" varchar(255) NOT NULL);

CREATE TABLE "djtalk_videopost" ("post_ptr_id" bigint NOT NULL PRIMARY KEY
REFERENCES "djtalk_post" ("id") DEFERRABLE INITIALLY DEFERRED,
"videoURL" varchar(255) NOT NULL);
```

Django ORM – nasleđivanje

```
>>> from djtalk.models import Post, VideoPost, ImagePost, User
>>> post = VideoPost(title='video title', content = 'video desc', videoURL = 'http:',
user=User.objects.get(id=1))
>>> post.save()
>>> post.id
7
>>> post.videopost
<VideoPost: VideoPost object (7)>
>>> videoPost = VideoPost.objects.first()
>>> videoPost.post_ptr
<Post: Post object (7)>
>>> for post in Post.objects.all():
...     print(post.title)
...
Parallel computing
Jurassic Park
China-Made Zhaoxin CPU Hits Retail Market
video title
>>>
```

- Rad sa objektom modela VideoPost. On naravno ima identitet Post objekta i nalazi se u kolekciji tih objekata. Preko reference na Post objekat može se doći do reference na VideoPost (downcast) sa post.videopost, naravno ako se stvarno radi o video postu inače se baca izuzetak
- Video post ima sve svoje i nasleđene atribute, a do reference na osnovni post može se doći preko post_ptr atributa.

Django ORM – korišćenje relacija

- Entitet se može učitati preko drugog već dohvaćenog entiteta ako između njih postoji definisana asocijacija (koje se koriste kao “dinamička polja”. Neki primeri iz aplikacije Talking (deo je u kontrolerima, deo u pogledima):

```
>>> users = User.objects.all()
>>> for user in users:
...    for post in user.post_set.all():
...      print(post.title)
```

- Moguće je koristiti i selekciju I filtriranje i na relacijama.
- Podrazumevano, tip dohvatanja za neku relaciju je LAZY, to jest, entitet povezan sa objektom Post učitaće se iz baze pri prvom referisanju preko relacije, posebnim upitom. što potencijalno rezultuje u velikom broju upita kada se npr. iterira kroz sve postove, a onda traže njihovi tagovi. Ako se želi da prilikom učitavanja objekta post odmah budu učitani i sa njim povezani objekti, treba specifikovati vrstu dohvatanja **prefetch_related**.
- Za primer Postova i njihovih Tagova: Bez with bilo bi N+1 SQL upit u bazu za N postova, sa with samo 2 upita: 1 nad tabelom posts, a drugi nad tabelom tags.

```
>>> posts = Post.objects.prefetch_related('tag').all()
>>> for post in posts:
...    for tag in post.tag.all():
...      print(tag.label)
```

Django ORM – ažuriranje relacija

Relacija 1:1 i jedinična strana 1:N

- Primer: User i ContactData. Ako želimo da ažuriramo kontakt podatke korisnika, koristimo save() nad relacijom. Pažnja, ne save() nad objektom korisnika, jer se tako neće zapamtiti izmene u bazi,

```
>>> user = User.objects.get(id=2)
>>> user.contactdata.email
'malina@example.com'
>>>
>>> user.contactdata.email = 'visnja@example.com'
>>> user.save()
>>> user = User.objects.get(id=2)
>>> user.contactdata.email
'malina@example.com'
>>>
>>> user.contactdata.email = 'visnja@example.com'
>>> user.contactdata.save()
>>> user = User.objects.get(id=2)
>>> user.contactdata.email
'visnja@example.com'
```

Django ORM – ažuriranje relacija

Relacija 1:N, strana sa kardinalnošću N i M:N

- Objekat može da se kreira i upiše u bazu sa **create()**
- Za dodavanje objekata na N stranu relacije koristi se **add()**. Objekti treba da već postoje u bazi. Automatski se radi i save relacije.

```
#models.py
class Blog(models.Model):
    pass

class Author(models.Model):
    name = models.CharField(max_length=255)

class Entry(models.Model):
    blog = models.ForeignKey(Blog, on_delete=models.CASCADE, null=True)
    authors = models.ManyToManyField(Author)

>>> from djtalk.models import Entry, Author
>>> entry = Entry.objects.create() # radi i
kreiranje objekta i save u bazu
>>> a1 = Author.objects.create(name="Paja")
>>> a2 = Author.objects.create(name="Zika")
>>> a3 = Author.objects.create(name="Laza")
>>> entry.authors.add(a1, a2, a3)
```

Django ORM – ažuriranje relacija

Uklanjanje relacija među objektima modela

- Za uklanjanje svih objekata iz relacije (ali ne i samih objekata iz baze) koristi se **clear()** nad relacijom.
- Za selektivno uklanjanje jednog ili više objekata iz relacije (bez delete samog objekta iz baze) koristi se **remove(o1, o2,..)** nad relacijom.

```
>>> entry.authors.add(a1, a2, a3)
>>> entry.authors.all()
<QuerySet [<Author: Author object (4)>, <Author: Author object (5)>,
<Author: Author object (6)>]>
>>> entry.authors.remove(a1, a2)
>>> entry.authors.all()
<QuerySet [<Author: Author object (6)>]>
```

Django ORM – transakciona obrada

- Transakcija je skup operacija nad bazom koje se izvršavaju kao jedna nedeljiva celina (atomično). Dakle ili se sve izvrše (comimit), ili se ni jedna ne izvrši (rollback).

- Transakcionom obradom u Djangu upravljaju dva podešavanje u settings.py vezana za bazu DATABASES = {}. To su AUTOCOMMIT, podrazumevano TRUE i ATOMIC_REQUESTS, podrazumevano FALSE.

- Podrazumevani autocommit znači da je svaki pojedinačni upit u bazu (rezultat raznih save() operacija upakovan u sopstvenu transakciju.

- Ako se stavi ATOMIC_REQUESTS na TRUE. svaka Django view funkcija koja obrađuje jedan http zahtev, upakovana je u svoju transakciju. Ovo može imati uticaj na performanse sistema.

Django ORM – transakciona obrada

- Django obezbeđuje eksplicitnu kontrolu tarnsakcija korišćenjem konstrukcije atomic, kao dekoratora, ili kao kontekst menadžera.
- U oba slučaja transakcija se uspešmo završava (commit) ako nije bilo bacanja izuzetaka u obuhvaćenom kodu, inače se radi poništavanje (rollback).

```
from django.db import transaction

@transaction.atomic
def viewfunc1(request):
    # This code executes inside a transaction.
    do_stuff()

def viewfunc2(request):
    # This code executes in autocommit mode (Django's default).
    do_stuff()

    with transaction.atomic():
        # This code executes inside a transaction.
        do_more_stuff()
```

Jedan problem sa django finderima

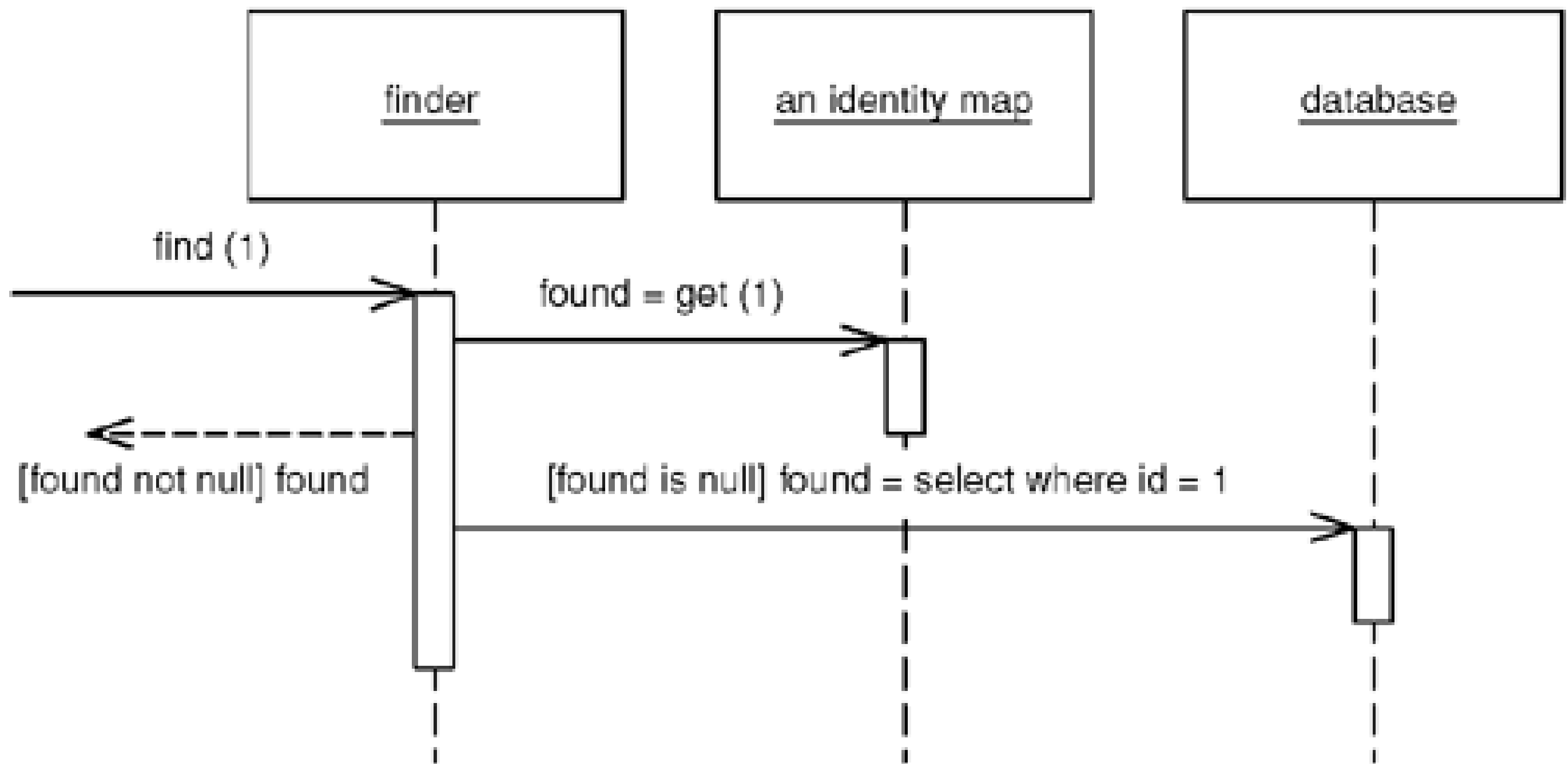
```
#models.py
class Order(models.Model):
    sum = models.IntegerField(default=0)


>>> from djtalk.models import Order
>>> order = Order.objects.create()
>>> order.sum
0
>>> order.sum = 200
>>> order.id
2
>>> order2 = Order.objects.get(id=2)
>>> order2.sum
0
>>> order2.sum = 100
>>> order.save()
>>> order2.save()
>>> order3 = Order.objects.get(id=2)
>>> order3.sum
100
```

- Problemi konzistencije podataka nastaju kada možemo napraviti različita dva memorijska objekta za isti entitet u bazi. Rešenje na nivou obrade kompletnog zahteva jednog korisnika je uzorak **Identity Map**.
- Kada se uzme u obzir konkurentni pristup bazi od strane više korisnika, jedno rešenje je eksplicitno koristiti database transakcije u kodu koji koristi domenske objekte.
- Alternativno, može se transakciona obrada ugraditi u Data Mapper kod putem **Unit of Work** patterna.

OBRAZAC IDENTITY MAP (MAPA IDENTITETA)

Obrazac Mapa identiteta

- Obezbeđuje da svaki objekat bude učitan samo jednom držeći svaki učitani objekat u mapi. Traži objekte uz pomoć mape kada se referiše na njih.



Dodavanje Identity Map-a u Django

- Učitava instance modela samo jednom u memoriju prvi put kada zatrebaju
- Obezbeđuje deljenje instance sa istim id-jem svuda u kodu do kraja obrade http zahteva

- Instalacija

```
pip install django-idmap
```

Takodje dodati idmap u settings.py u INSTALLED_APPS

- Osnovno korišćenje:
- Klase modela naslediti iz idmap.models.IdMapModel umesto django.db.models

```
from idmap import models
class MyModel (models.IdMapModel):
    . . .
```

OBRAZAC UNIT OF WORK (JEDINICA RADA)

Obrazac Jedinica rada

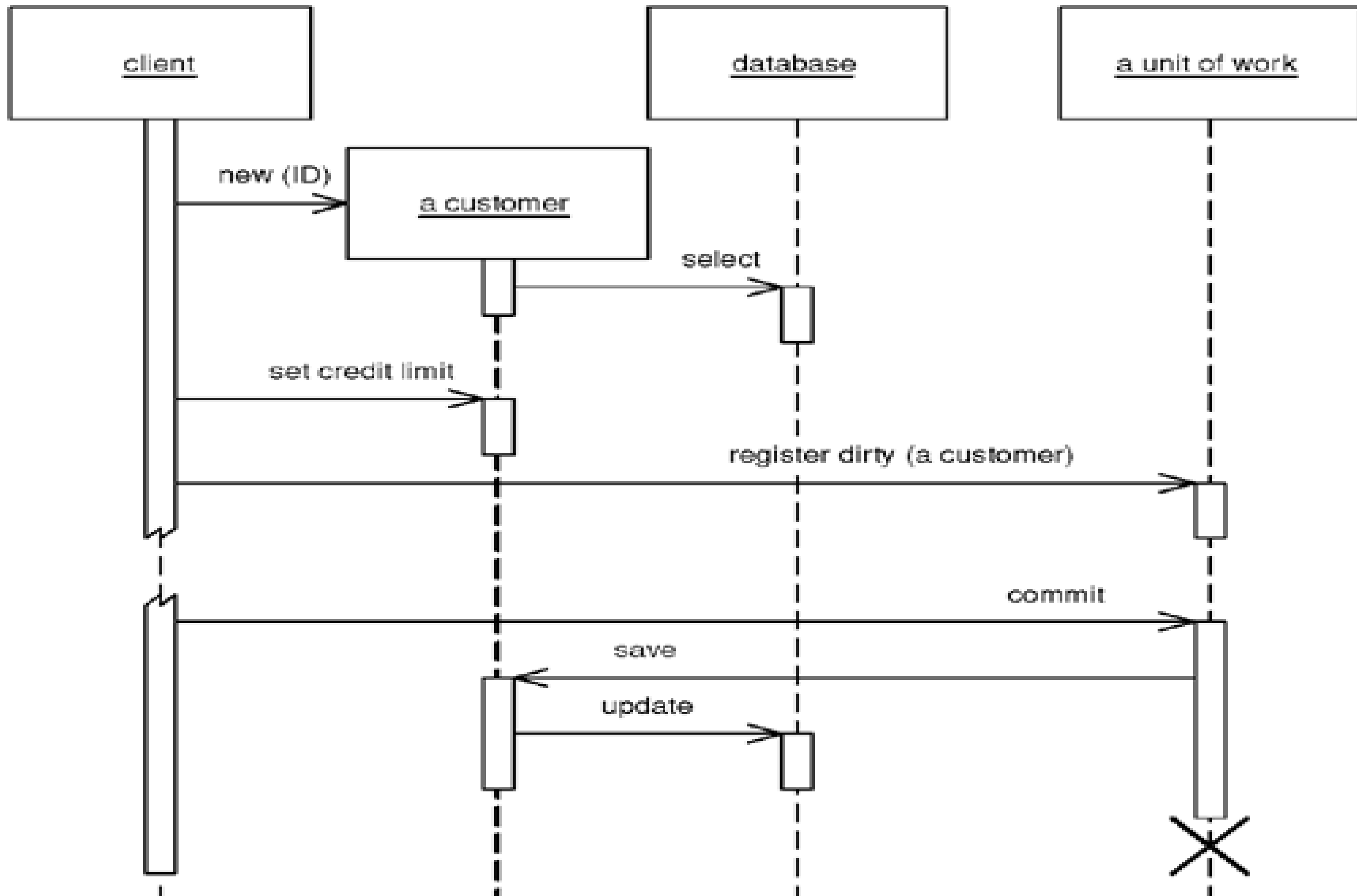
- Održava listu objekata uključenih u poslovne transakcije i koordinira upisivanje promena i rešava probleme konkurencije.
- Kada vučete podatke iz baze podataka, važno je da pratite šta ste menjali; U suprotnom, podaci neće biti zapisani nazad u bazu. Slično morate da ubacite nove objekte koje ste kreirali i uklonite sve objekte koje ste obrisali.
- Možete da menjate u bazi podataka sa svakom promene vašeg objektnog modela, ali to može da dovede do mnogo malih pristupa bazi, koji se sporo izavršavaju. Dodatno je potrebno da imate otvorenu transakciju tokom cele interakcije, što je nepraktično ako imate poslovnu transakciju koja obuhvata više zahteva.
- Situacija je još gora ako treba da pratite objekte koje ste pročitali, da bi izbegli nedoslednosti.
- Jedinica rada vodi evidenciju o svemu što radite tokom poslovne transakcije što može da utiče na bazu podataka. Kada završite, ona preduzme sve što treba da se uradi da promeni bazu podataka kao rezultat vašeg rada.

Kako funkcioniše Jedinica rada

- Svaki put kada se kreira, menja ili briše objekat to se saopšti jedinici rada. Ona takođe zna o objektima koji su učitani iz baze tako da može da proveri nekonzistencije u čitanju tako što proverava da nijedan od objekata nije promenjen u bazi podataka u toku poslovne transakcija.
- Ključna stvar o jedinici rada je da, kada dođe vreme da se izvrši commit, jedinica rada odlučuje šta da radi. Ona otvara transakciju, radi kontrolu konkurentnosti (koristi pesimističko ili optimističko zaključavanje), i zapisuje promene u bazi podataka.
- Programer nikada eksplicitno ne poziva metode ažuriranja baze. Na taj način on ne mora da prati šta se promenilo ili da brine o tome kako referencijalni integritet utiče na poredak u kome treba da se rade stvari.
- Programer (ili sam poslovni objekat) mora jedinici rada da saopšti o kojim objektima treba da vodi računa.

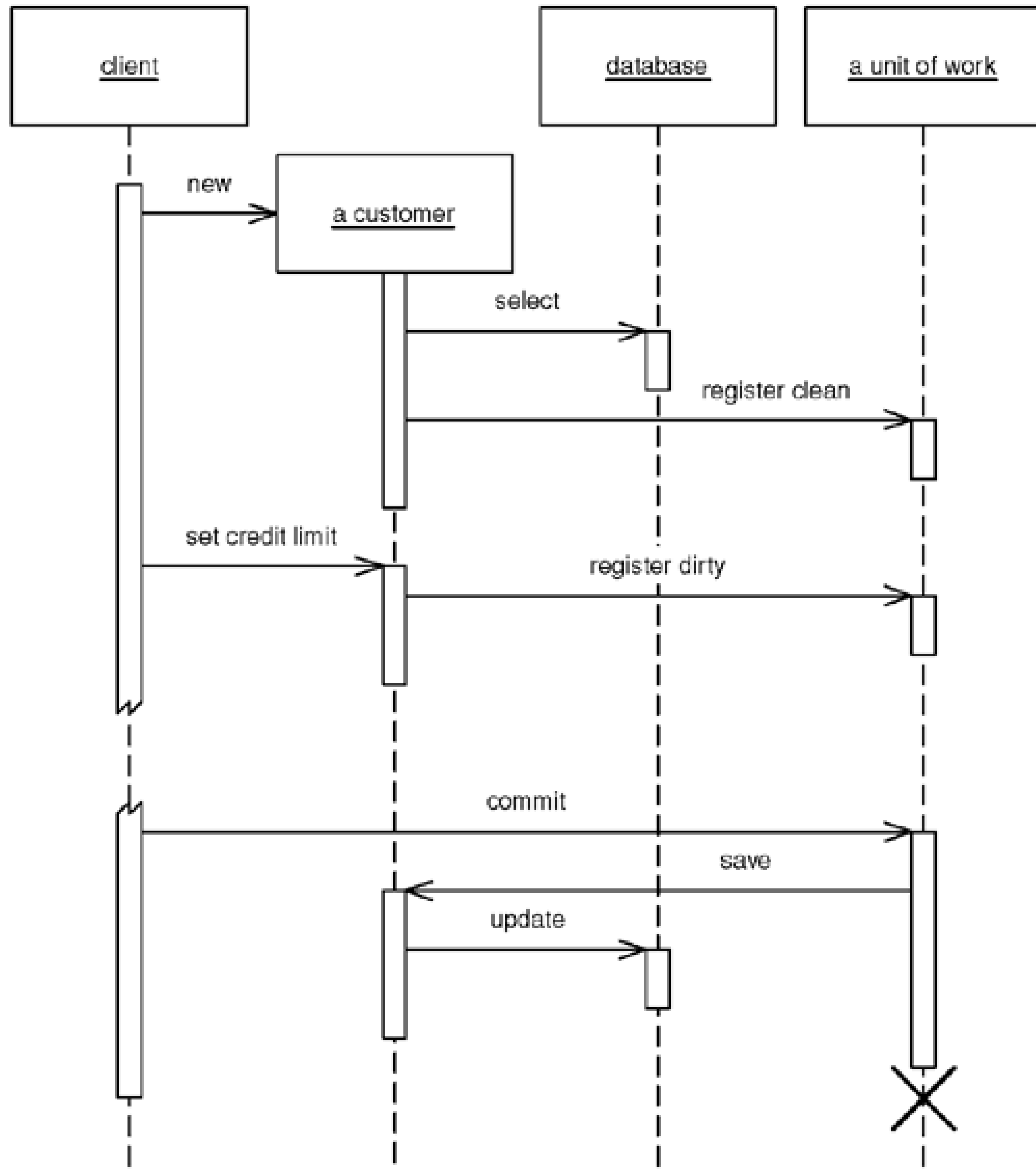
Pozivalac registruje promenjeni objekat

- Sa registracijom od strane pozivaoca, korisnik objekta mora registrovati objekat za promene kod Jedinice rada. Objekat koji nije registrovan neće biti upisan pri commitu.



Objekat sam sebe registruje

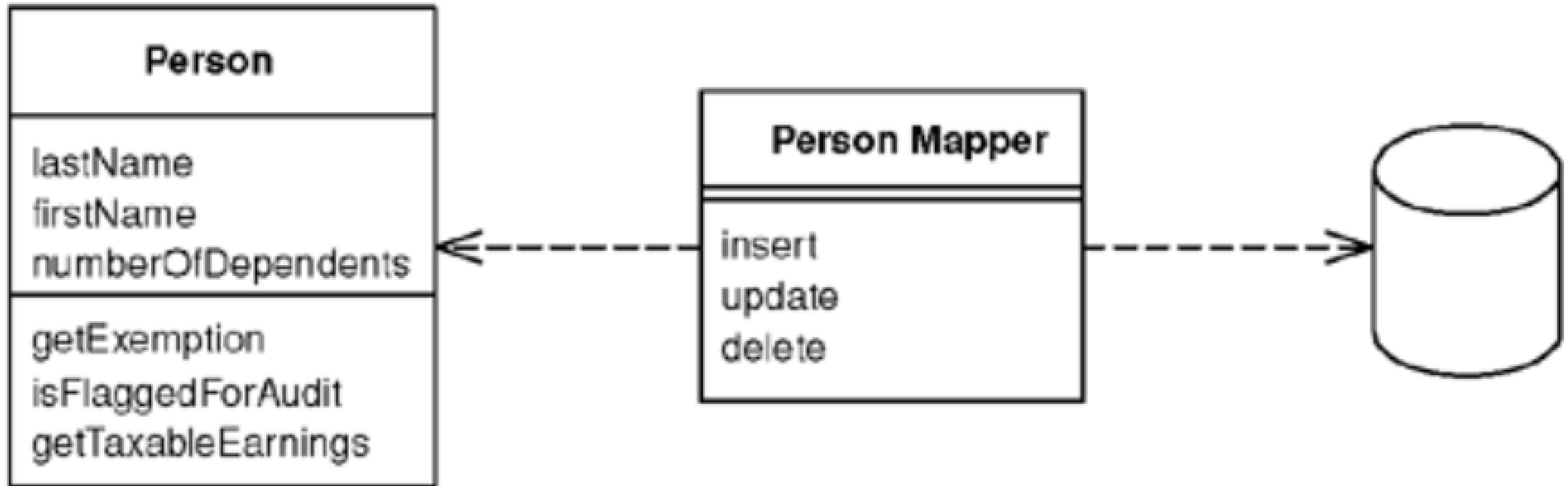
- Registracija se vrši u metodama objekta.
- Učitavanje objekta iz baze registruje objekat kao čist;
- setXY metode registruju objekat kao prljav.
- Jedinica za rad treba da bude prosleđena objektu ili da bude na nekom dobro poznatom mestu.



OBRAZAC DATA MAPPER (MAPIRANJE PODATAKA)

Obrazac Mapiranje podataka

Sloj koji omogućava razmenu podataka između objekata i baze održavajući ih nezavisnim jedne od drugih, kao i od samog Mappera.



Objekti i relacione baze podataka imaju različite mehanizme za strukturiranje podataka. Mnogi delovi objekata, kao što su kolekcije i nasleđivanja, nisu prisutni u relacionim bazama podataka. Kada se izgradi objekatni model sa puno poslovne logike, važno je da se mogu koristiti ovi mehanizmi da se bolje organizuju podaci i ponašanje koje ide sa njima.
Na taj način imamo dve različite šeme, to jest, objekatna šema i relaciona šema se ne podudaraju.

Data Mapper

- Pošto treba razmenjivati podatke između dve šeme, to postaje složen problem. Ako objekti u memoriji poznaju relacionu šemu baze podataka, promene u jednoj šemi imaju tendenciju da se preliju i na drugu.
- Data Mapper je posrednički sloj softvera koji razdvaja objekte u memoriji od baze podataka. Njegova odgovornost je prenos podataka između njih i takođe da ih izoluje jedne od drugih.
- Sa Data Mapperom objekti u memoriji ne treba da znaju ni da postoji baza podataka, ne treba im SQL kod, a svakako ni poznavanje šeme baze podataka. (I baza podataka ne zna ništa o objektima koji je koriste). I sam Data Mapper je čak nepoznat domenskim objektima.
- **SqlAlchemy** ORM za python zasnovan je na Data Mapper I Unit of Work obrascima (nećemo proučavati u ovom kursu).

Kada upotrebljavati ove uzorke?

- Graditelj upita je verovatno najjednostavniji obrazac za korišćenje, ukida potrebu za učenjem sintakse SQLa, sakriva detalje konkretnih implementacija menadžera baza podataka, ali mana je što nema pravog mapiranja na objektni model, ako nije kombinovan sa data mapperom.
- U knjigama (teorijski) se navodi da je Aktivni zapis dobar izbor za domensku logiku koja odgovara direktno tabelama baze podataka, ako su im izomorfne šeme. Ako je, međutim, poslovna logika složena, programer želi mogućnost korišćenja relacija, kolekcija, nasleđivanja itd poslovnih objekata. U takvoj situaciji preporučeni uzorak je Data Mapper.
- Međutim, u praksi npr. Django (Activni z.) ima podršku za gotovo sve stvari kao i SQLAlchemy (Data M.), pa se u obzir za izbor mogu uzeti i pokazatelji kao što su lakoća korišćenja, performanse sistema itd.

Zaključak

- Razumevanje OO dizajna web aplikacija zasnovano je na poznavanju projektnih uzoraka koji se primenjuju u njihovom projektovanju.
- Definisali smo najvažije uzorke i ilustrovali implementacije opšte strukture web aplikacije, kontrolnog, prezentacionog sloja i sloja modelovaja podataka.
- Ne treba samostalno implementirati pomenute šablone, nego iskoristiti gotove biblioteke i aplikativne okvire (frameworks), koji su popularni i provereno ispravno implementirani.

Literatura

- Martin Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley 2002.
- Django documentation, https://docs.djangoproject.com/en/4.1/