Rozwiązania szkieletowe w tworzeniu aplikacji www

Tomasz Łukaszuk

Katedra Oprogramowania
Wydział Informatyki
Politechnika Białostocka

Temat wykładu:

Modele i ORM

Fundusze Europejskie dla Rozwoju Społecznego
Rzeczpospolita Polska
Dofinansowane przez Unię Europejską
NCBR Narodowe Centrum Badań i Rozwoju

Materiały przygotowane w ramach projektu "PB 5.0 – dostosowanie oferty dydaktycznej Politechniki Białostockiej do potrzeb nowoczesnej gospodarki oraz zielonej i cyfrowej transformacji" (nr umowy FERS.01.05-IP.08-0327/23-00) w ramach programu Fundusze Europejskie dla Rozwoju Społecznego 2021-2027 współfinansowanego ze środków Europejskiego Funduszu Społecznego Plus.

Modele i ORM: Agenda

1. MVC: Model
2. Mapowanie obiektowo-relacyjne
3. Django - model i ORM
4. Ruby on Rails - model i ORM
   - Migracje
   - Klasy modeli
   - Interfejs zapytań

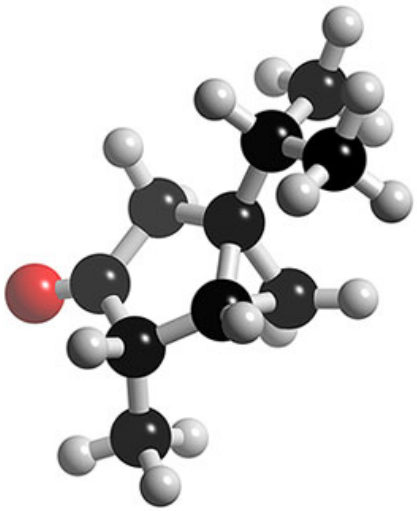
MVC: Model

MVC

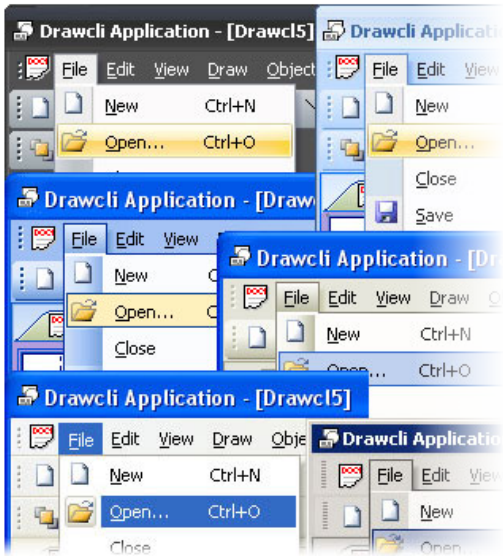
komentarz Trygve Reenskaug odnośnie wzorca MVC

The View is connected to the user's eyes;
the Controller is connected to the user's hands;
and the Model is connected to the user's mind.

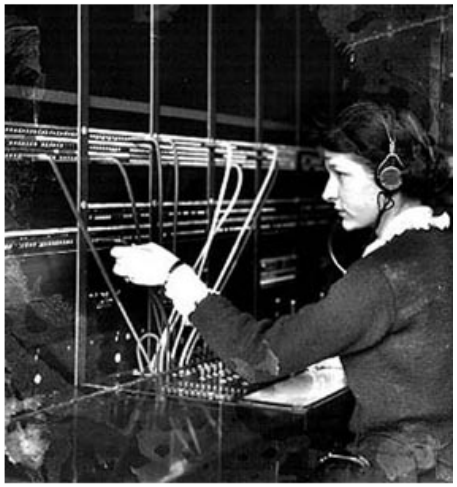
Model:



View:



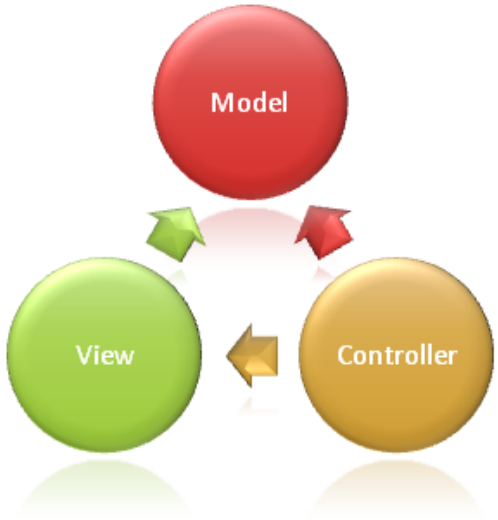
Controller:



źródło: http://c2.com

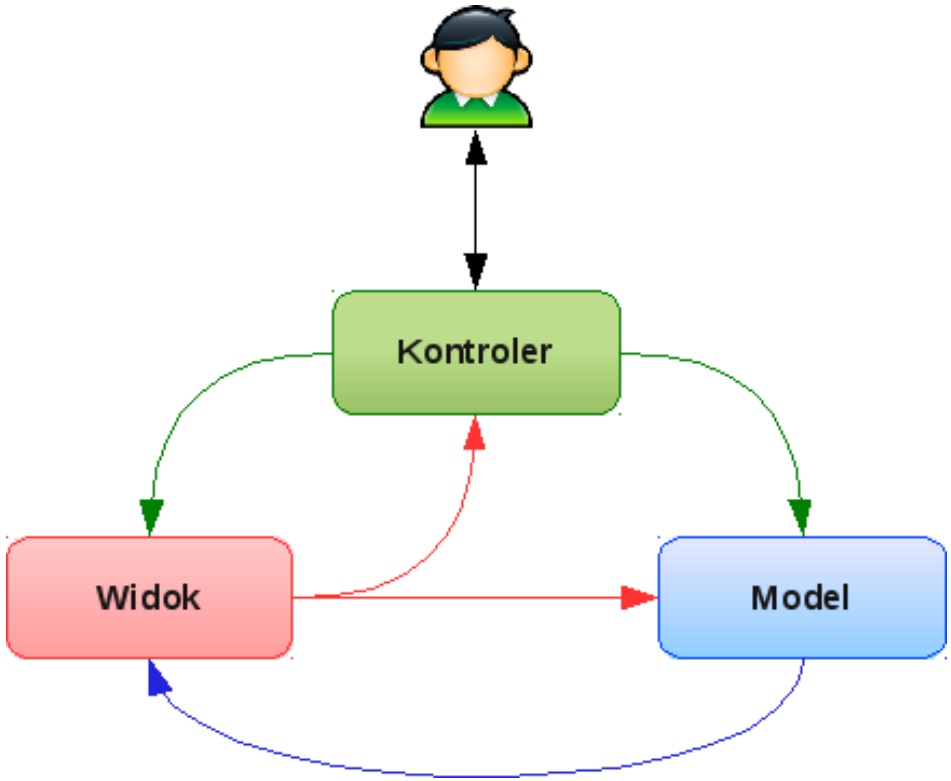
MVC: Model - przypomnienie

- odpowiada za tzw. logikę biznesową
- jedyna część aplikacji, która przechowuje dane w sposób trwały
- różne sposoby przechowywania danych
- szczegóły implementacyjne związane ze sposobem przechowywania danych ukryte przed resztą aplikacji



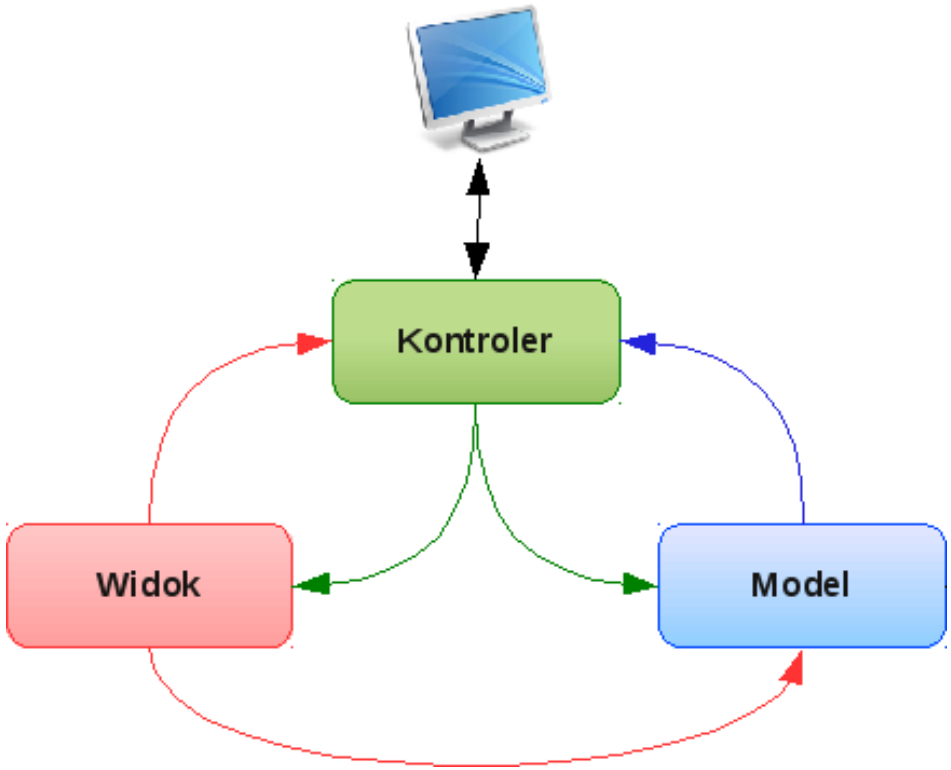
  - model dostarcza innym komponentom spójnego interfejsu do przetwarzania danych
  - jest samodzielny — do poprawnej pracy nie wymaga obecności widoków i kontrolerów
  - dobrze zaprojektowany model powinien być przenośny

MVC Model 1



- klasyczny wzorzec projektowy MVC
- model aktywny

MVC Model 2



- zmodyfikowane ścieżki komunikatów
- model nie komunikuje się w żaden sposób z widokiem
- lepiej nadaje się do aplikacji webowych

Model == Logika biznesowa

Logika biznesowa (logika domenowa, logika dziedzinowa)

- modelowanie rzeczywistych obiektów biznesowych (np. klient, konto, rozkład jazdy)
- określenie interakcji pomiędzy obiektami biznesowymi
- określenie sposobów i metod modyfikowania i aktualizowania obiektów biznesowych

Model != Baza danych

- model jest warstwą pośredniczącą pomiędzy bazą danych (lub innym źródłem danych), a pozostałymi częściami aplikacji, w szczególności kontrolerem
- model ma prawo wybrać dowolny sposób uzyskania żądanych informacji
- model z reguły zawiera więcej informacji (logiki) niż informacje zapisane w bazie danych

Modelowanie danych

- dostosowanie danych do użytku przez kontroler
- zazwyczaj jest to przekazywanie informacji w postaci tablic, wartości logicznych, liczb i ciągów znaków
- w przypadku zamiany źródła danych (np. baza danych na pliki XML) model powinien zwracać dane w takiej samej postaci jak w przypadku pierwotnego źródła danych

Zalecenia odnoszące się do modelu w aplikacjach MVC

- Model powinien zawierać logikę biznesową aplikacji, a nie tylko mapować tabele z bazy danych
- błędem jest modyfikowanie Modelu z poziomu Widoku, Model jest dla Widoku dostępny tylko w trybie read-only
- błędem jest wykonywanie logiki biznesowej zmieniającej stan Modelu wewnątrz Kontrolera, Kontroler ma za zadanie przekierowywać wszelkie żądania użytkownika na odpowiednie wywołania Modelu

Mapowanie obiektowo-relacyjne

Mapowanie obiektowo-relacyjne

- ORM = Object-Relational Mapping = Mapowanie obiektowo-relacyjne
- sposób odwzorowania obiektowej architektury systemu informatycznego na bazę danych (lub inny element systemu) o relacyjnym charakterze
- tworzony system oparty jest na podejściu obiektowym, a system bazy danych operuje na relacjach
- zazwyczaj oprogramowanie ORM tworzy "wirtualną obiektową bazę danych"
- rozwiązania komercyjne i darmowe

Tradycyjna technologia dostępu do danych

- nawiązanie połączenia z bazą, poprzez podanie: typu bazy danych, adresu IP (lub domeny), nazwy użytkownika oraz hasła
- wysłanie zapytania SQL, np. `SELECT * FROM Students`
- odebranie wyniku, najczęściej jest to tablica obiektów
- zamknięcie połączenia z bazą danych

Wady tradycyjnej technologii dostępu do danych

- podczas kompilacji programu, poprawność zapytań SQL nie jest sprawdzana
- zmiany w strukturze bazy danych wymuszają zmiany w kodzie aplikacji np. zmiana nazwy tabeli wymusza zmianę wszystkich zapytań operujących na tej tabeli
- w przypadku zmiany dostawcy bazy danych np. z SQL Server na Oracle, część zapytań będzie nieprawidłowa
- powtarzanie tego samego bloku kodu nie jest dobrym zwyczajem w programowaniu obiektowym

Zalety ORM

- wydajność - posiada mechanizmy cache oraz skomplikowane algorytmy optymalizujące
- obiektowość - operacje na bazie danych są przeprowadzane za pomocą programowania obiektowego, możliwość wykorzystania dziedziczenia
- tworzenie bazy danych - tworząc aplikacje można bardziej skupić się nad logiką biznesową, oraz nad tym co powinny reprezentować sobą klasy, i dopiero na tej podstawie narzędziem ORM stworzyć strukturę bazy danych
- brak zapytań SQL - typowe zapytania SQL można zredukować do minimum, lub całkowicie wyeliminować
- efektywność pracy - wszelkie poprawki w logice biznesowej są na bieżąco synchronizowane z bazą danych bez interwencji programisty

Wady ORM

- wprowadza pewien narzut (opóźnienia w transmisji informacji), ponieważ tworzy dodatkową warstwę przez którą muszą być przesłane dane
- pomimo zaawansowanych algorytmów trudno jest otrzymać zaawansowane zapytania SQL, w zoptymalizowanej formie np. wielokrotnie zagnieżdżone, skorelowane zapytania
- nie nadaje się do obróbki dużych ilości danych gdzie wydajność jest podstawowym priorytetem

Django - model i ORM

Baza danych w Django

- Django to "database driven application framework"
- Może używać baz danych MySQL, SQLite, PostgreSQL, Oracle
- Aby uniknąć problemów z połączeniami, programista nie powinien korzystać z własnych połączeń, ale raczej polegać na tych, oferowanych przez framework
- Prezentuje bazę danych jako zbiór obiektów
- Każda kolekcja ma metody, aby pobrać obiekty (wiersze), filtrować je, pobierać w odpowiedniej kolejności, itp.

Modele

- Ogólnie rzecz biorąc, każdy model odpowiada jednej tabeli bazy danych
- Każdy model jest klasą Python'a, dziedziczącą po klasie `django.db.models.Model`
- Każdy atrybut modelu reprezentuje pole bazy danych
- Django daje automatycznie generowane API pozwalające na dostęp do bazy danych

Przykład:

```
1  from django.db import models
2
3  class Person(models.Model):
4      first_name = models.CharField(max_length=30)
5      last_name = models.CharField(max_length=30)
```

Modele Django vs. tabele w bazie danych

Model Django:

```
1  from django.db import models
2
3  class Person(models.Model):
4      first_name = models.CharField(max_length=30)
5      last_name = models.CharField(max_length=30)
```

Kod SQL tabeli utworzonej na podstawie modelu:

```
1  CREATE TABLE myapp_person (
2      "id" serial NOT NULL PRIMARY KEY,
3      "first_name" varchar(30) NOT NULL,
4      "last_name" varchar(30) NOT NULL
5  );
```

Pola

- Najważniejsza część modelu
- ... i jedyny wymagany element modelu
- Lista pól w tabeli bazy danych
- Pola są określane przez atrybuty klasy

```
1  class Musician(models.Model):
2      first_name = models.CharField(max_length=50)
3      last_name = models.CharField(max_length=50)
4      instrument = models.CharField(max_length=100)
5
6  class Album(models.Model):
7      artist = models.ForeignKey(Musician)
8      name = models.CharField(max_length=100)
9      release_date = models.DateField()
10     num_stars = models.IntegerField()
```

Typ pola

- Pole powinno być instancją odpowiedniej klasy `Field`
- Typ klasy Field determinuje: typ kolumny w bazie danych, widget używany w interfejsie administracyjnym Django, wymagania walidacyjne
- Django oferuje kilkanaście typów pól

Typy pól

IntegerField	
AutoField	IntegerField, w którym wartości generowane są automatycznie, używane do kluczy podstawowych
FloatField	
BooleanField	
CharField	
TextField	
EmailField	TextField które sprawdza czy wartośc ma format emaila
URLField	TextField które przechowuje url
XMLField	TextField zawierające XML
PhoneNumberField	
IPAddressField	
DateTimeField	także DateField and TimeField
FilePathField	
FileField	
ImageField	

Opcje pól

Są przekazywane w nazwanych parametrach do konstruktora obiektu pola.

null	boolean, czy pole może być ustawione na NULL
blank	boolean, czy pole może być puste
choices	kolekcja wartości, które może przyjmować pole
db_column	nazw kolumny w tabeli bazy danych, która ma reprezentować pole
db_index	boolean, czy ma być utworzony indeks dla pola w bazie danych
editable	boolean, czy pole może być edytowane
help_text	opis wyświetlany w module administracyjnym
primary_key	boolean, true jeżeli pole ma być kluczem podstawowym
unique	
verbose_name	

Pola wyboru

```
1  from django.db import models
2
3  class Person(models.Model):
4      GENDER_CHOICES = (
5          (u'M', u'Male'),
6          (u'F', u'Female'),
7      )
8      name = models.CharField(max_length=60)
9      gender = models.CharField(max_length=2, choices=GENDER_CHOICES)
```

Relacje Many-to-one

`django.db.models.ForeignKey`

```
1  class Manufacturer(models.Model):
2      # ...
3
4  class Car(models.Model):
5      manufacturer = models.ForeignKey(Manufacturer)
6      # ...
```

Relacje Many-to-many

django.db.models.ManyToManyField

```
1  class Topping(models.Model):
2      # ...
3
4  class Pizza(models.Model):
5      # ...
6      toppings = models.ManyToManyField(Topping)
```

```
1  class Person(models.Model):
2      ...
3
4  class Group(models.Model):
5      members = models.ManyToManyField(Person, through='Membership')
6
7  class Membership(models.Model):
8      person = models.ForeignKey(Person)
9      group = models.ForeignKey(Group)
10     ...
```

Relacje One-to-one

`django.db.models.OneToOneField`

Są najbardziej przydatne jeżeli chcemy rozszerzyć istniejący model o dodatkowe parametry

```
1  class Place(models.Model):
2      ...
3
4  class Restaurant(models.Model):
5      place = models.OneToOneField(Place, primary_key=True)
6      ...
```

Metadane modelu

Metadane modelu to "wszystko co nie jest polem", czyli np. porządek sortowania (ordering), nazwa tabeli w bazie danych (db_table), przyjazne nazwy modelu w liczbie pojedynczej i mnogiej (verbose_name i verbose_name_plural).

```
1  class Ox(models.Model):
2      horn_length = models.IntegerField()
3
4      class Meta:
5          ordering = ["horn_length"]
6          verbose_name_plural = "oxen"
```

Metody modelu

Metody modelu pozwalają na dodanie do modelu dodatkowych funkcjonalności.
Jest to cenna technika utrzymania logiki biznesowej w jednym miejscu – modelu.

```
1   from django.contrib.localflavor.us.models import USStateField
2
3   class Person(models.Model):
4       first_name = models.CharField(max_length=50)
5       last_name = models.CharField(max_length=50)
6       birth_date = models.DateField()
7       address = models.CharField(max_length=100)
8       city = models.CharField(max_length=50)
9       state = USStateField()
10
11      def is_midwestern(self):
12          "Returns True if this person is from the Midwest."
13          return self.state in ('IL', 'WI', 'MI', 'IN', 'OH', 'IA', 'MO')
14
15      def _get_full_name(self):
16          "Returns the person's full name."
17          return '%s %s' % (self.first_name, self.last_name)
18      full_name = property(_get_full_name)
```

Metody automatycznie tworzone w modelu

- __str__ - Python "magic method" która określa co ma być wyświetlane po wywołaniu `str()` na obiekcie
- __unicode__ - metoda uruchomiana po wywołaniu `unicode()` na obiekcie
- get_absolute_url - metoda mówi Django jak obliczyć URL obiektu

```
1  class Person(models.Model):
2      first_name = models.CharField(max_length=50)
3      last_name = models.CharField(max_length=50)
4
5      def __str__(self):
6          return smart_str('%s %s' % (self.first_name, self.last_name))
7
8      def __unicode__(self):
9          return u'%s %s' % (self.first_name, self.last_name)
10
11     def get_absolute_url(self):
12         return "/people/%i/" % self.id
```

Wykonywanie zapytań (wprowadzenie)

```
1  class Blog(models.Model):
2      name = models.CharField(max_length=100)
3      tagline = models.TextField()
4      def __unicode__(self):
5          return self.name
6
7  class Author(models.Model):
8      name = models.CharField(max_length=50)
9      email = models.EmailField()
10     def __unicode__(self):
11         return self.name
12
13 class Entry(models.Model):
14     blog = models.ForeignKey(Blog)
15     headline = models.CharField(max_length=255)
16     body_text = models.TextField()
17     pub_date = models.DateTimeField()
18     mod_date = models.DateTimeField()
19     authors = models.ManyToManyField(Author)
20     n_comments = models.IntegerField()
21     n_pingbacks = models.IntegerField()
22     rating = models.IntegerField()
23     def __unicode__(self):
24         return self.headline
```

Wykonywanie zapytań I

- Tworzenie obiektów

```
1  b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
2  b.save()
```

- Zapisywanie zmian w obiekcie

```
1  b5.name = 'New name'
2  b5.save()
3
4  entry = Entry.objects.get(pk=1)
5  cheese_blog = Blog.objects.get(name="Cheddar Talk")
6  entry.blog = cheese_blog
7  entry.save()
```

- Pobieranie obiektów

Wykonywanie zapytań II

```
1  all_entries = Entry.objects.all()
2
3  Entry.objects.filter(pub_date__year=2006)
4
5  Entry.objects.exclude(pub_date__gte=datetime.now())
6
7  one_entry = Entry.objects.get(pk=1)
```

- Porównywanie obiektów

```
1  some_entry == other_entry
2  some_entry.id == other_entry.id
```

- Usuwanie obiektów

```
1  e.delete()
2
3  Entry.objects.filter(pub_date__year=2005).delete()
```

Wykonywanie zapytań III

- Aktualizacja wielu obiektów na raz

```
1   Entry.objects.filter(pub_date__year=2007).update(headline='the same')
```

Pobieranie obiektów - metody zwracające kolekcje obiektów

all	wszystkie obiekty
filter	obiekty spełniające określone kryteria
exclude	obiekty nie spełniające określone kryteria
distinct	zbiór (nie kolekcja) obiektów; SELECT DISTINCT
order_by	sortuje obiekty według określonych pól
values	zamiast obiektów zwraca zbiór słowników, jeden słownik reprezentuje jeden wiersz z bazy danych
extra	pozwala określić zapytanie w języku SQL

Pobieranie obiektów - metody nie zwracające kolekcji obiektów

get	zwraca dokładnie jeden obiekt, jeśli warunki powodują zwrócenie zera lub więcej niż jeden obiektów, występuje wyjątek
get_or_create	tworzy obiekt, jeśli nie istnieje, a następnie zwraca jeden obiekt, który istniał przed czy też po utworzeniu
create	tworzy obiekt, może być użyta zamiast konstruktora
count	zwraca liczbę obiektów
in_bulk	pobiera zbiór kluczy podstawowych i zwraca słownik z podanych wartości jako klucze i obiektów jako wartości
latest	przyjmuje nazwę kolumny (typu czasowego) i zwraca najnowszych obiektów według danej kolumny

Pobieranie obiektów - Pole wyszukiwania

exact	wartość musi być równa
iexact	porównanie bez uwzględniania wielkości liter
contains	podana wartość jest częścią pola
icontains	“contains” bez uwzględniania wielkości liter
gt, gte, lt, lte	porównania arytmetyczne
in	wartość pochodzi z kolekcji
startswith, istartswith	
endswith, iendswith	
range	pobiera parę i sprawdza, cz wartość jest pomiędzy podanymi wartościami
isnull	boolean, sprawdza czy wartość jest null’em
pk	porównanie do klucza podstawowego

Zarządzanie bazą danych

By using command manage.py with appropriate options:

dbshell	opens command-line shell to database
validate	checks correctness of declarations of models
sqlall	prints SQL to create all objects (tables, indices, custom objects)
sql	prints SQL to create tables
sqlcustom	prints SQL needed to customize tables
sqlclear	prints SQL to drop all objects
sqlflush	prints SQL that is required so database returns to initial state, just after application creation
sqlreset	prints SQL that drops and then creates objects in database
syncdb	checks which database objects exist and creates those that are defined but do not exist yet
cleanup	removes old data from database (old sessions, etc.)
dumpdata	outputs content of the database

Ruby on Rails - model i ORM

Dostęp do bazy danych

- Moduł ActiveRecord odpowiada za mapowanie relacyjno-obiektowe
- http://rubyforge.org/projects/activerecord/
- Klasy dziedziczące po ActiveRecord są odwzorowywane w relacyjnej bazie danych
- Moduł ActiveRecord bazuje na wzorcu projektowym Active Record
- http://en.wikipedia.org/wiki/Active_record_pattern
- Konstruktor ActiveRecord akceptuje hash'e nazwy pól i ich wartości

Rails Database Migrations

- Migracje są wygodnym sposobem, aby zmienić bazę danych w sposób zorganizowany i uporządkowany.
- ActiveRecord śledzi, które migracje zostały już uruchomione.
- W migracjach używa się języka Ruby.
- Migracje są niezależne od bazy danych.
- Migracja jest klasą dziedziczącą po `ActiveRecord::Migration` implementującą dwie metody statyczne: `up` (wykonywana w momencie uruchomienia migracji) i `down` (wycofanie migracji).

Anatomia migracji

```
1  class CreateProducts < ActiveRecord::Migration
2    def self.up
3      create_table :products do |t|
4        t.string :name
5        t.text :description
6
7        t.timestamps
8      end
9    end
10
11   def self.down
12     drop_table :products
13   end
14 end
```

ActiveRecord dostarcza metody, które pozwalają na wykonanie popularnych operacji *data definition* w niezależny od bazy danych sposób.

`create_table`, `change_table`, `drop_table`, `add_column`, `change_column`, `rename_column`, `remove_column`, `add_index`, `remove_index`

Klasa modelu

```
1  class Person < ActiveRecord::Base
2     #validations
3
4     #assosiations
5
6     #user defined methods
7  end
```

Active Record Validations Helpers I

- validates_acceptance_of - pole checkbox zostało zaznaczone
- validates_associated - model jest powiązany z innym modelem i on też powinien być zwalidowany
- validates_confirmation_of - dwa pola tekstowe powinny zawierać to samo
- validates_exclusion_of - wartości atrybutu nie należą do podanego zbioru
- validates_format_of - sprawdza czy wartości pasują do określonego wyrażenia regularnego
- validates_inclusion_of -wartości atrybutu należą do określonego zbioru
- validates_length_of - odpowiednia długość tekstu
- validates_numericality_of - tylko wartości numeryczne

Active Record Validations Helpers II

- validates_presence_of - atrybuty nie są puste
- validates_uniqueness_of - nie zapisano wcześniej obiektu z taką samą wartością atrybutu
- validates_with - przekazuje rekord do walidacji przez inna klasę
- validates_each - walidauje atrybuty z uzyciem bloku

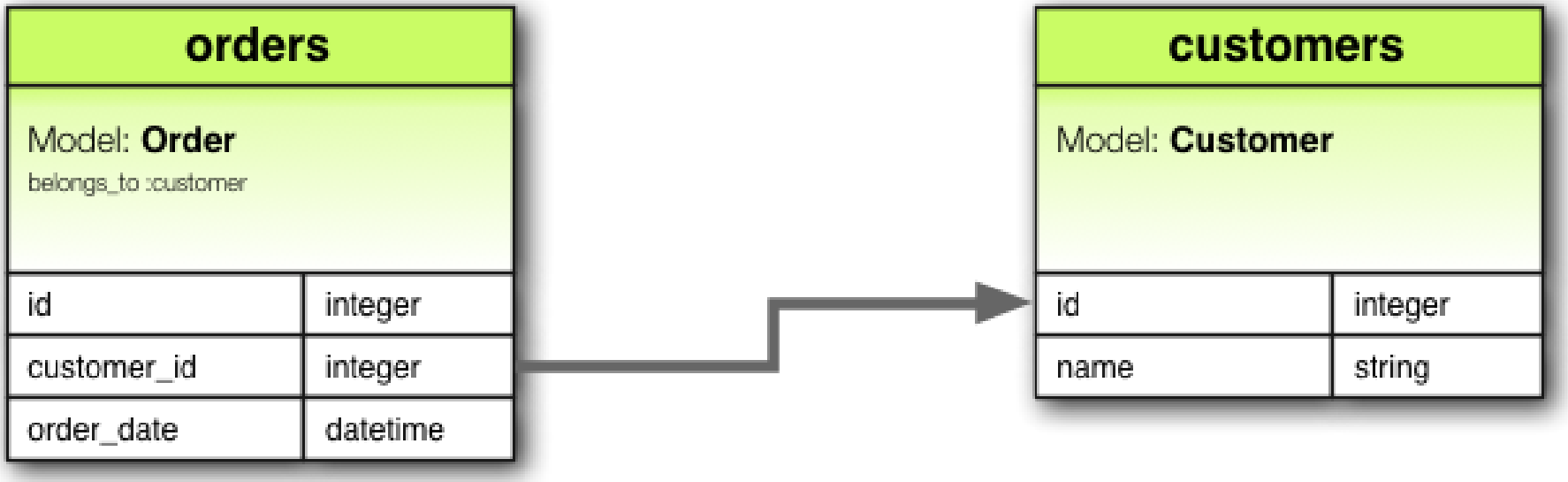
```
1  class Person < ActiveRecord::Base
2    validates_acceptance_of :terms_of_service
3    validates_confirmation_of :email
4    validates_presence_of :email_confirmation
5    validates_length_of :name, :minimum => 2
6    validates_format_of :legacy_code, :with => /\A[a-zA-Z]+\z/,
7      :message => "Only letters allowed"
8  end
```

Active Record Associations

Typy asocjacji:

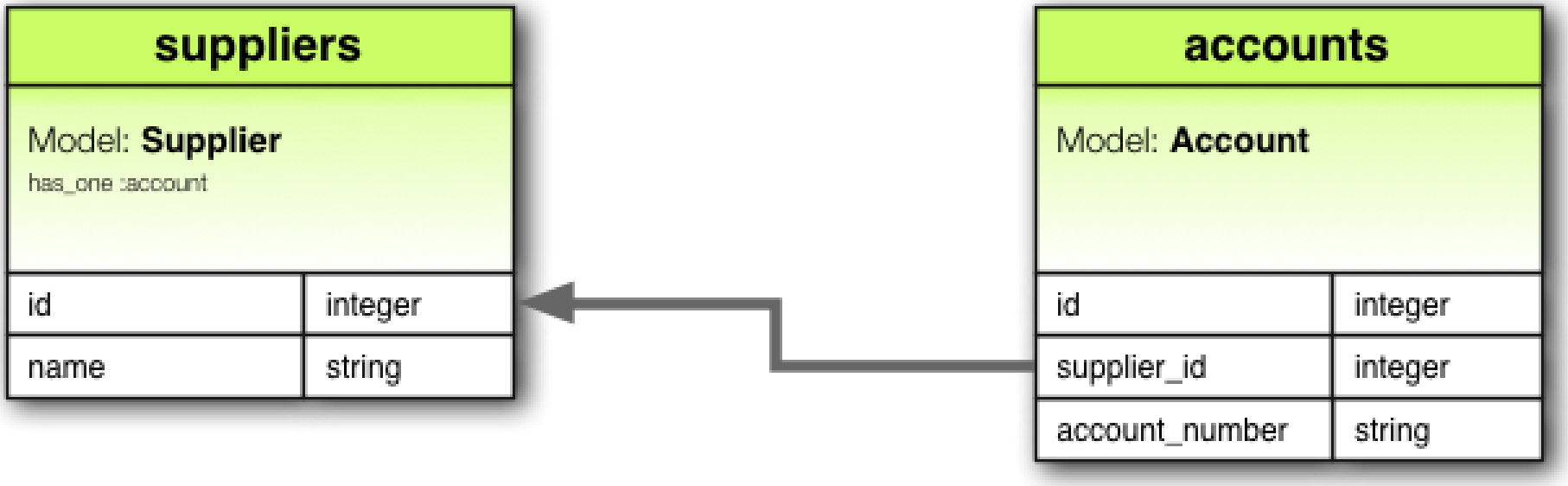
- belongs_to
- has_one
- has_many
- has_many :through
- has_one :through
- has_and_belongs_to_many

The belongs_to Association



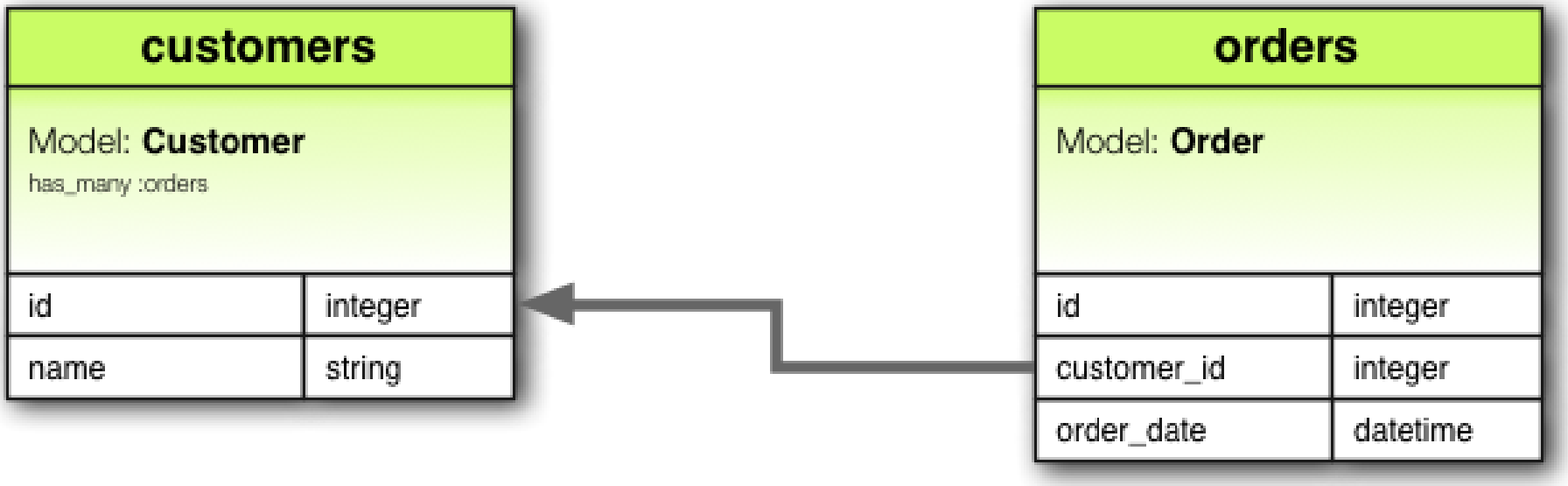

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

The has_one Association



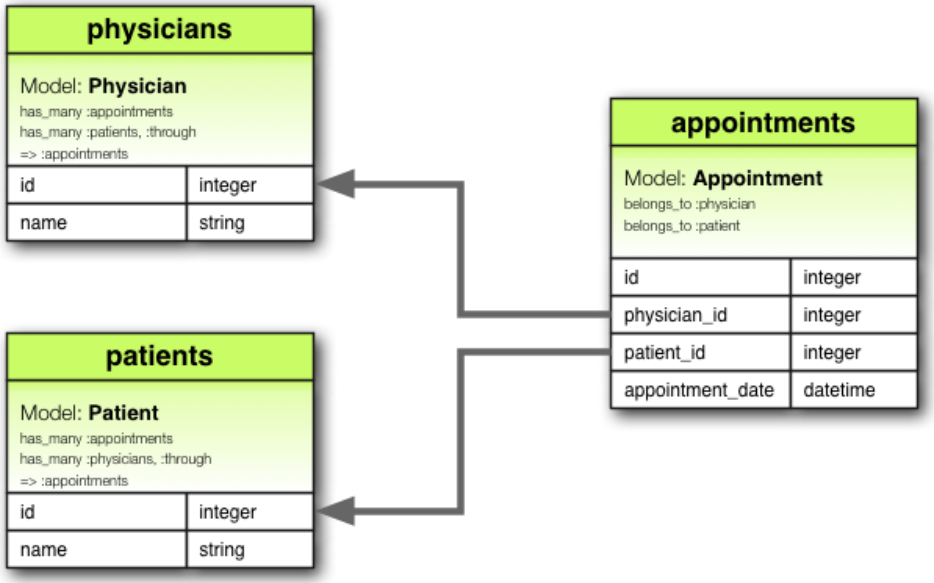

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

The has_many Association




```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

The has_many :through Association

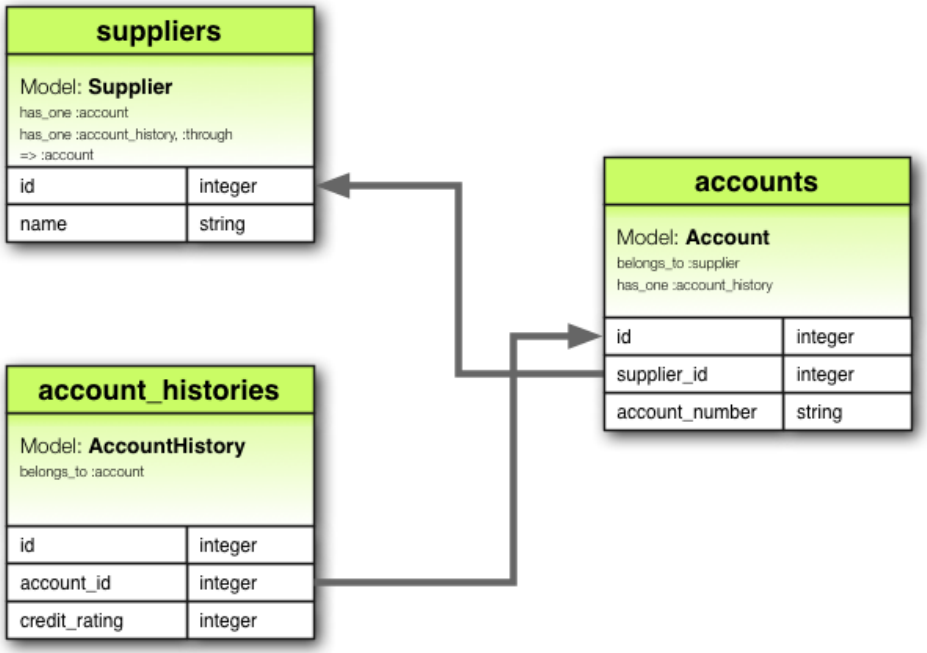



```
class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, :through => :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, :through => :appointments
end
```

The has_one :through Association

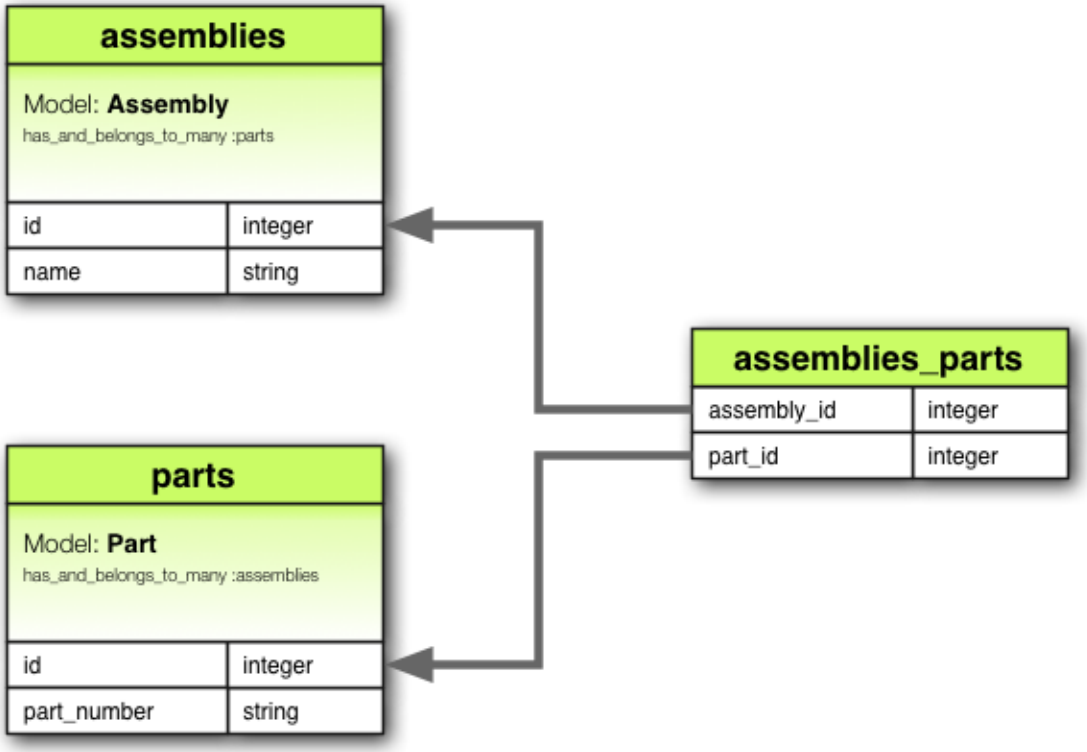



```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

The has_and_belongs_to_many Association




```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

Active Record - interfejs zapytań

```
1  class Client < ActiveRecord::Base
2    has_one :address
3    has_many :orders
4    has_and_belongs_to_many :roles
5  end
6
7  class Address < ActiveRecord::Base
8    belongs_to :client
9  end
10
11 class Order < ActiveRecord::Base
12   belongs_to :client, :counter_cache => true
13 end
14
15 class Role < ActiveRecord::Base
16   has_and_belongs_to_many :clients
17 end
```

Pobieranie obiektów

Pobieranie pojedynczych obiektów

```
1  client = Client.find(10)
2  client = Client.first
3  client = Client.last
```

Pobieranie wielu obiektów

```
1  client = Client.find(1, 10)
2
3  User.find_each do |user|
4    NewsLetter.weekly_deliver(user)
5  end
```

Zapisywanie obiektów do bazy danych

- create, create!
- save, save!
- update
- update_attributes, update_attributes!

Active Record - metody do wyszukiwania obiektów

- where
- select
- group
- order
- limit
- offset
- joins
- includes
- lock
- readonly
- from
- having

Active Record - metody do wyszukiwania obiektów

```
1  Client.where("orders_count = ? AND locked = ?", 2, false)
2
3  Client.order("created_at ASC")
4
5  Client.select("viewable_by, locked")
6  Client.select("DISTINCT(name)")
7
8  Client.limit(5).offset(30)
9
10 Order.group("date(created_at)").order("created_at")
11
12 Order.group("date(created_at)").having("created_at > ?", 1.month.ago)
```

Dynamiczne metody do wyszukiwania obiektów

Dla każdego pola (atrybutu) zdefiniowanego w tabeli Active Record tworzy metody *finder method*.
Załóżmy, że w modelu `Client` mamy pole `first_name`

metody *finder method*:

- find_by_first_name
- find_all_by_first_name
- find_last_by_first_name
- find_or_create_by_first_name
- find_or_initialize_by_first_name

```
1  Client.find_by_name("Ryan")
2  Client.find_by_first_name_and_locked("Ryan", true)
3  Client.find_or_create_by_first_name("Ryan")
```

Inne metody dostępu do danych

```
1  #sprawdzanie istnienia obiektu
2  Client.exists?(1)
3  Client.exists?(1,2,3)
4  Client.where(:first_name => 'Ryan').exists?
5
6  #obliczenia
7  Client.count
8  Client.where(:first_name => 'Ryan').count
9  Client.average("orders_count")
10 Client.minimum("age")
11 Client.maximum("age")
12 Client.sum("orders_count")
```