Table des matières

1 Introduction

Ce document présente le pattern **ActiveRecord** comme définit par Martin Fowler.

L'implémentation utilisée est celle en Ruby.

1.1 DAL: Data Access Layer

Tout logiciel possèdent une couche d'accès aux données. On la nomme `DAL`. Ceci est le terme général.

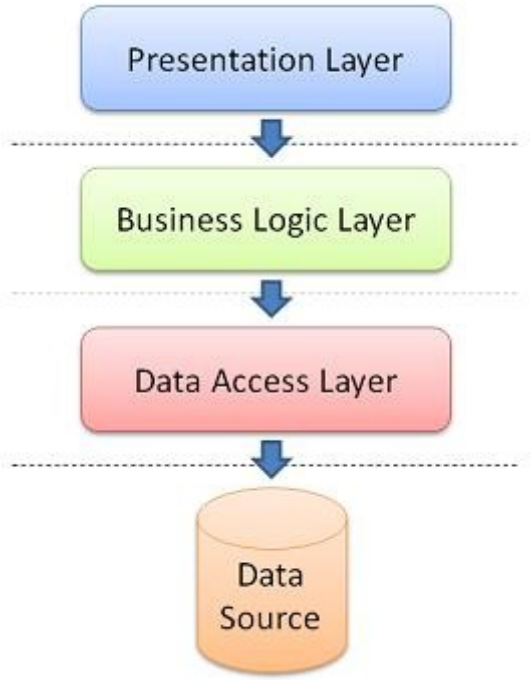


Figure 1: DAL

Il est possible d'implémenter son DAL de façons très différentes.

Voici quelques-unes d'entre-elles

1.2 RecordSet

Exemple dans un langage *curly braces*:

```
1 DbConnection connection = new DbConnection("adapter:mysql;hostname=localhost;
     username=root;password:'';database=poo1_shop");
2 connection.Open();
3
4 DbResult result = connection.Execute("SELECT * FROM products");
5 while (!result.IsEmpty()) {
```

```
6     DbRecordSet record = result.Next();
7
8     print(record['name']);
9     print(record['price']);
10  }
```

- Simple (simpliste ?)
- SQL partout dans le code de l'application

1.3 Table Data Gateway

```
1  class ProductGateway {
2    public DbResult findAll();
3    public DbResult findAllByPriceLowerThan(Money threshold);
4    public DbRecordSet findById(int id);
5    public void update(string name, Money price);
6    public void insert(int id, string name, Money price);
7    public void delete(int id);
8  }
```

- Code SQL localisé dans le Gateway
- Pas de DomainObject

1.4 Row Data Gateway

```
1  class Product {
2    protected int id;
3    protected string name;
4    protected Money price;
5
6    public Product(int id)    { this.id = id; }
7    public int    getId()     { return id; }
8    public string getName()   { return name; }
9    public void   setName(string value) { name = value; }
10   public string getPrice() { return price; }
11   public void   setPrice(Money value) { price = value; }
12
13   public void insert();
14   public void update();
15   public void delete();
16 }
```

```
class ProductFinder {
  public Array<Product> findAll();
  public Product findById(int id);
}
```

- DomainObject pour les produits
- Il faut une classe supplémentaire pour les *finders*

1.5 Data Mapper (ORM)

```
class Product {
  protected int id;
  protected string name;
  protected Money price;

  public Product(int id) { this.id = id; }
  public int getId() { return id; }

  public string getName() { return name; }
  public void   setName(string value) { name = value; }

  public string getPrice() { return price; }
  public void   setPrice(Money value) { price = value; }
}

class ProductMapper {
  public Array<Product> findAll();
  public Product findById(int id);
  public void update(Product product);
  public void insert(Product product);
  public void delete(Product product);
}
```

- DomainObject pour les produits
- DomainObject découplé de la stratégie de persistance
- Utilisation de la classe Mapper pour toutes les opérations de persistance

1.6 ActiveRecord (ORM)

Row Data Gateway + méthodes de *Business Logic* dans la même classe.

- DomainObject pour les produits
- Méthodes de persistance dans les DomainObjects
- DomainObject couplé à la stratégie de persistance

1.7 Connexion

```ruby
require 'active_record'

ActiveRecord::Base.establish_connection(
    adapter:  'mysql2',
    host:     'localhost',
    username: 'root',
    password: '',
    database: 'poo1_shop'
)
```

1.8 Exemple

```ruby
class Product < ActiveRecord::Base
end
```

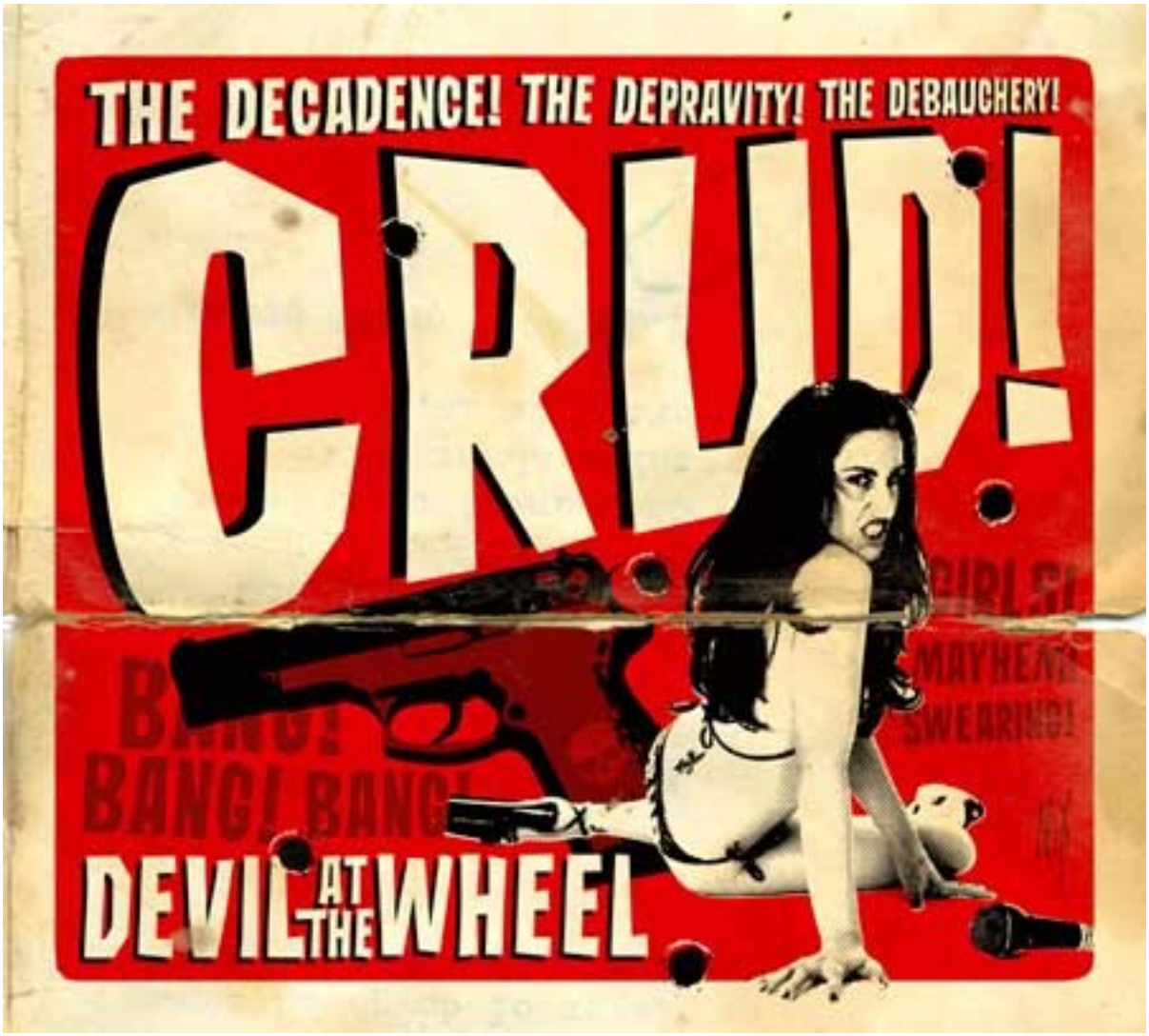
Découvre les champs de la table *automatiquement*. Comment? **Conventions de nommage**

La table doit être nommée comme le nom de la classe mais en minuscule et au pluriel => `products`

Pas content? Pas de problème, mais alors il faut être explicite:

```ruby
class Product < ActiveRecord::Base
  self.table_name = 'TBL_LEGACY_PRODUCT'
end
```

2 CRUD



2.1 Create

```
>> titine = Car.new
=> #<Car id: nil, color: nil>
>> titine.color = 'red'
=> "red"
>> titine
=> #<Car id: nil, color: "red">
>> titine.save
=> true
```

```
>> titine = Car.new(color: 'red')
=> #<Car id: nil, color: "red">
>> titine.save
```

```
4  => true
```

```
1  >> titine = Car.create(color: 'red')
2  => #<Car id: 5, color: "red">
```

2.2 Read

```
1  >> Car.find 2
2  => #<Car id: 2, color: "red">
3  >> Car.all
4  => [#<Car id: 1, color: "white">, #<Car id: 2, color: "red">, #<Car id: 3,
       color: nil>, #<Car id: 4, color: "red">]
5  >> Car.first
6  => #<Car id: 1, color: "white">
7  >> Car.last
8  => #<Car id: 4, color: "red">
```

Obtenir un objet par un ou plusieurs de ses attributs:

```
1  >> Car.find_by color: 'white'
2  => #<Car id: 1, color: "white">
3  >> Engine.find_by power: 130
4  => #<Engine id: 1, car_id: 2, power: 130, brand: "Porsche">
5  >> Engine.find_by power: 130, brand: 'GM'
6  => #<Engine id: 3, car_id: 4, power: 130, brand: "GM">
```

2.3 Update

```
1  >> titine = Car.create(color: 'red')
2  => #<Car id: 5, color: "red">
3  >> titine.color = 'yellow'
4  => "yellow"
5  >> titine.save
6  => true
```

```
1  >> titine = Car.first
2  => #<Car id: 1, color: "white">
3  >> titine.color = 'black'
4  => "black"
5  >> titine.save
6  => true
```

```
>> titine = Car.first
=> #<Car id: 1, color: "white">
>> titine.update(color: 'black')
=> true
```

2.4 Delete

```
>> titine = Car.last
=> #<Car id: 5, color: "red">
>> titine.destroy
=> #<Car id: 5, color: "red">
>> Car.find(5)
ActiveRecord::RecordNotFound: Could not find Car with id=5
>> titine.color = 'white'
TypeError: can't modify frozen hash
>> titine
=> #<Car id: 5, color: "red">
```

2.5 Recherche

```
>> Car.where("color = 'red'")
=> [#<Car id: 2, color: "red">, #<Car id: 4, color: "red">]
>> user_form_color = 'red'
=> "red"
>> Car.where("color = '#{user_form_color}'")
=> [#<Car id: 2, color: "red">, #<Car id: 4, color: "red">]
```

```
>> user_form_color = "' OR 1 AND id = 1) -- "
=> "' OR 1 AND id = 1) -- "
>> Car.where("color = '#{user_form_color}'")
=> [#<Car id: 1, color: "white">]
```

```
>> Car.where('color = ?', user_form_color)
=> []
```

```
>> Car.where(color: 'red')
=> [#<Car id: 2, color: "red">, #<Car id: 4, color: "red">]
```

2.6 Requêtage

2.6.1 order

2.6.2 limit & offset

2.6.3 group & having

2.6.4 distinct

2.6.5 pluck

```
1 >> Engine.distinct.pluck(:brand)
2 => ["Porsche", "VW", "GM"]
```

3 Associations

3.1 Principes

On *déclare* l'association dans le corps de la classe du modèle. (Merci ruby, ceci appelle la méthode de classe `has_many`)

```ruby
class Car < ActiveRecord::Base
  has_many :wheels
end
```

Ceci a pour effet d'ajouter plusieurs méthodes (d'instance) afin de pouvoir manipuler la relation comme un objet. Pour ce `has_many` les méthodes sont:

```ruby
  def wheels(force_reload = false)
  end

  def wheels=(objects)
  end
```

3.2 Principes (bis)

Voici comment l'utiliser

```
>> titine = Car.first
=> #<Car id: 1, color: "white">
>> titine.wheels
=> [#<Wheel id: 1, car_id: 1, size: 15>, #<Wheel id: 2, car_id: 1, size: 15>]
```

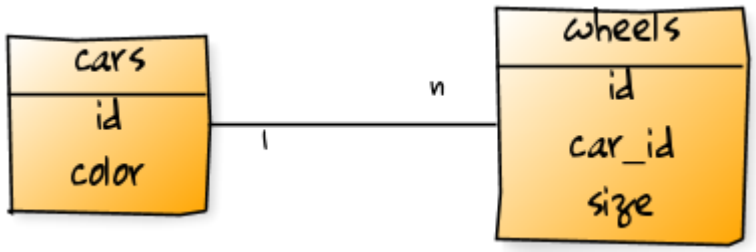
Comment AR arrive-t-il à sortir ces enregistrement avec si peu d'indications? **Conventions de nommage**

`has_many :wheels` permet de *trouver* que les objets associés sont des `Wheel`. La jointure SQL doit connaître la clé étrangère, il se trouve que `has_many` est appelé dans la classe `Car` ce qui permet de générer `car_id`.

```
class Car < ActiveRecord::Base
  has_many :wheels, class_name: 'Wheel', foreign_key: 'car_id'
end
```

La doc de référence de `has_many` sur RoR ou API Dock

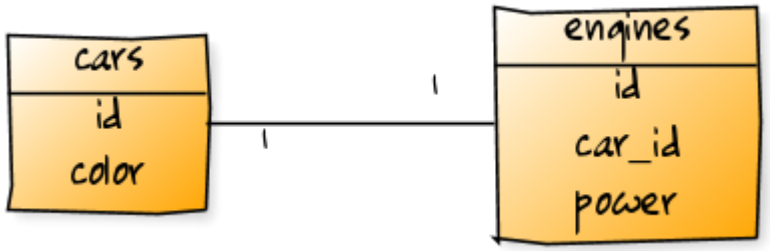
3.3 Association 1 - N




```
class Car < ActiveRecord::Base
  has_many :wheels
end

class Wheel < ActiveRecord::Base
  belongs_to :car
end
```

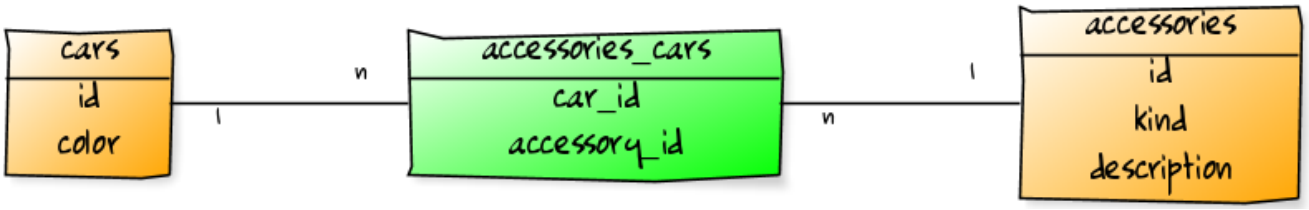
3.4 Association 1 - 1




```ruby
class Car < ActiveRecord::Base
  has_one :engine
end

class Engine < ActiveRecord::Base
  belongs_to :car
end
```

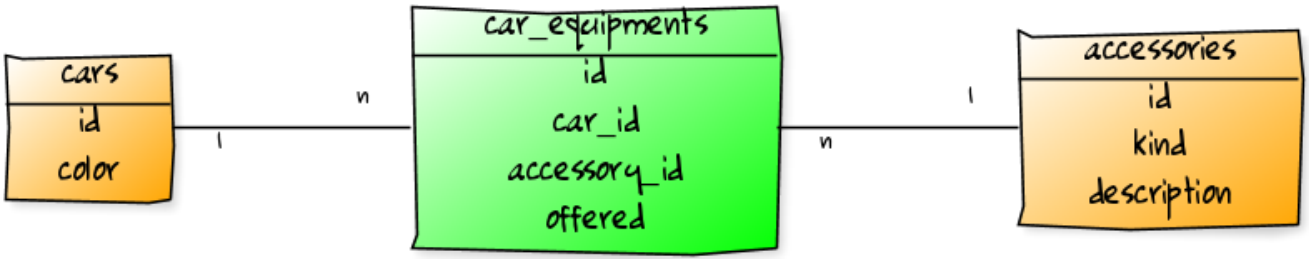
3.5 Association N - N directe




```ruby
class Car < ActiveRecord::Base
  has_and_belongs_to_many :accessories
end

class Accessory < ActiveRecord::Base
  has_and_belongs_to_many :cars
end
```

3.6 Association N - N indirecte




```ruby
class Car < ActiveRecord::Base
  has_many :car_equipments
  has_many :accessories, through: :car_equipments
end

class Accessory < ActiveRecord::Base
  has_many :car_equipments
  has_many :cars, through: :car_equipments
end

class CarEquipment < ActiveRecord::Base
  belongs_to :car
  belongs_to :accessory
end
```

3.7 Exemples

```ruby
car = Car.first

puts car.wheels.count

puts car.accessories
```

```sql
SELECT cars.* FROM cars LIMIT 1

SELECT COUNT(*) FROM wheels WHERE wheels.car_id = 1

SELECT accessories.* FROM accessories
  INNER JOIN accessories_cars
  ON accessories.id = accessories_cars.accessory_id
  WHERE accessories_cars.car_id = 1
```

3.8 Exemples

```
>> car = Car.first
=> #<Car id: 1, color: "white">
>> car.wheels.count
=> 2
>> car.accessories
=> []
>> car.accessories << Accessory.find_by(name: 'autoradio')
=> [#<Accessory id: 1, name: "autoradio", kind: nil, description: "Radio">]
>> car.accessories
=> [#<Accessory id: 1, name: "autoradio", kind: nil, description: "Radio">]
>> car.accessories.delete(Accessory.find_by(name: 'autoradio'))
=> [#<Accessory id: 1, name: "autoradio", kind: nil, description: "Radio">]
>> car.accessories
=> []
```

3.9 Associations lointaines

has_many :through has_one :through belongs_to :through ???

3.10 Gestion des dépendances

dependant => :

Avec :destroy, cascade des dépendances => bien!

3.11 Requêtage

Sur des associations de collections (`has_many` et `has_and_belongs_to_many`), il est possible d'utiliser toutes les méthodes de requêtage déjà vue.

Exemple:

```
Car.first.accessories.where('price < ?', 20).order(:name)
```

4 Scopes & Eager loading

4.1 Pré-chargement (Eager loading)

But: Eviter le problème des requêtes 1+N

```ruby
cars = Car.all

cars.each do |car|
  puts car.wheels.join(',')
end
```

```sql
SELECT  cars.* FROM  cars

SELECT  wheels.* FROM  wheels  WHERE  wheels.car_id  = 1
SELECT  wheels.* FROM  wheels  WHERE  wheels.car_id  = 2
SELECT  wheels.* FROM  wheels  WHERE  wheels.car_id  = 3
SELECT  wheels.* FROM  wheels  WHERE  wheels.car_id  = 4
```

4.2 Pré-chargement (Eager loading)

Solution

```ruby
cars = Car.includes(:wheels).all

cars.each do |car|
  puts car.wheels.join(',')
end
```

```sql
SELECT  cars.* FROM  cars

SELECT  wheels.* FROM  wheels  WHERE  wheels.car_id  IN (1,2,3,4)
```

4.3 Resserrer le périmètre (scopes)

But: Se définir des périmètres métiers

- se définit sur le modèle où l'on désire le resserrement.
- on utilise les mêmes méthodes que pour la recherche.

```ruby
class Car < ActiveRecord::Base
  scope :white, -> { where(color: 'white') }

  scope :small_wheels,
    -> { joins(:wheels).where('wheels.size' => 13).group(:id) }

  scope :wheels_with_size,
    ->(size) {
      joins(:wheels).where('wheels.size' => size).group(:id)
    }
end
```

4.4 Scopes: Utilisation

Utilisation: comme les méthodes pour la recherche

```ruby
puts Car.white

puts Car.wheels_with_size(13)

puts Car.wheels_with_size(13).white
```

```sql
SELECT  cars.* FROM  cars  WHERE  cars.color  = 'white'

```

```
SELECT cars.* FROM cars INNER JOIN wheels
  ON wheels.car_id = cars.id
  WHERE wheels.size = 13 GROUP BY id

SELECT cars.* FROM cars INNER JOIN wheels
  ON wheels.car_id = cars.id
  WHERE wheels.size = 13 AND cars.color = 'white'
  GROUP BY id
```

4.5 Scopes: méthodes?

En réalité la méthode `scope` sur `ActiveRecord::Base` n'est qu'un raccourci syntaxique qui définit une méthode de classe. Ces deux définitions sont identiques:

```
class Car < ActiveRecord::Base
  scope :white, -> { where(color: 'white') }

  def self.white
    where(color: 'white')
  end
end
```

5 Validations

5.1 Pourquoi?

S'assurer de la cohérence des données dans la BD

```ruby
class Car < ActiveRecord::Base
  validates :color, presence: true
end
```

```ruby
>> titine = Car.new
=> #<Car id: nil, color: nil>
>> titine.save
=> false
>> titine.errors.full_messages
=> ["Color can't be blank"]
```

5.2 En standard

```
# Le champs doit être VRAI
acceptance: true

acceptance: {
  # Spécifie le terme considéré comme VRAI
  accept: __string__
}
```

```
# Le contenu du champs doit être le même que le
# champs: nom_du_champs_confirmation
confirmation: true
```

```
# Le champs doit être rempli
presence: true
```

5.3 En standard

```
# Le champs doit respecter un format défini par une Regexp
format: { with: __regexp__ }
```

```
# Le contenu du champs doit être exclu d'un ensemble
exclusion: {
  # Spécifié le contenu refusé
  in: __array__
  # ou  within
}
```

```
# Le contenu du champs doit faire partie d'un ensemble
inclusion: {
  # Spécifié le contenu accepté
  in: __array__
  # ou  within
}
```

5.4 En standard

```
# Le champs doit être d'une certaine longueur (texte)
length: {
  # La longueur doit être au minimum de la valeur spécifiée
  minimum: value
  # La longueur doit être au maximum de la valeur spécifiée
  maximum: value
  # La longueur doit être comprise dans l'intervalle spécifié
  in: __range__ # ou  within
  # La longueur doit être la valeur spécifiée
  is: value
}
```

```
# Le champs doit être unique dans la BD
uniqueness: true

uniqueness: {
  # Permet de tester l'unicité restreinte par rapport à un champs
  scope: 'parent_id'
}
```

5.5 En standard

```
# Le champs doit être numérique
numericality: true

numericality: {
  # Contrôle que le champs soit un entier
  only_integer: true
  # Contrôle que le champs soit plus grand que la valeur spécifiée
  greater_than: value
  # Contrôle que le champs soit plus grand ou égale à la valeur spécifiée
  greater_than_or_equal_to: value
  # Contrôle que le champs soit égal à la valeur spécifiée
  equal_to: value
  # Contrôle que le champs soit plus petit que la valeur spécifiée
  less_than: value
  # Contrôle que le champs soit plus petit ou égal à la valeur spécifiée
  less_than_or_equal_to: value
  # Contrôle que le champs soit impair
  odd: true
  # Contrôle que le champs soit pair
  even: true
```

```
21 }
```

5.6 Exemple

```ruby
class Car < ActiveRecord::Base
  validates :color,
    presence: true,
    length: { minimum: 3, maximum: 31 }

  validates :price,
    numericality: {
      greater_than: 2000,
      less_than: 1000000,
      even: true
    }

  validates :name, uniqueness: true
end
```

5.7 Options supplémentaires

Permettre les champs non renseigné

```ruby
class Car < ActiveRecord::Base
  validates :color, length: , allow_blank: true
  validates :price, numericality: true, allow_nil: true
end
```

Choisir le moment de la validation, par défaut: `save`

```ruby
class Car < ActiveRecord::Base
  validates :color, length: {minimum: 3}, on: :create
  validates :price, numericality: true, on: :update
end
```

5.8 Validations conditionnelles

Chaque validation peut être executée ou non selon l'état de l'object

```ruby
class Wheel < ActiveRecord::Base
  validates :size, numericality: true, if: :mounting?

  protected

  def mounting?
    # return true or false
  end
end
```

```ruby
class Car < ActiveRecord::Base
  validates :color, length: {minimum: 3},
    unless: ->(car) { car.engine.nil? }
end
```

5.9 Associations

```ruby
class Car < ActiveRecord::Base
  validates_associated :wheels
end
```

Attention, en consultant `#errors` on aura simplement un `invalid` sur le champs `wheels`, pour avoir le détail il faudra consulter les objets de la relations.

5.10 Personnalisées

Il y a plusieurs possibilités pour ajouter des validations personnalisées, voici la façon *direct* (sans réutilisation)

```ruby
class Car < ActiveRecord::Base
  validate :quality_test, on: :create

  protected

  def quality_test
    if something_gone_wrong
      errors.add(:base, "Bad quality!")
    end
  end
end
```

5.11 Externes

6 Callbacks & Observers

6.1 Callbacks, callwhat?

Afin d'assurer l'état des objets ActiveRecord, il est possible de *s'accrocher* (hook) à plusieurs moments du cycle de vie d'un objet.

Ceci est fait en déclarant des **callbacks**

```
1  class Teacher < ActiveRecord::Base
2    before_validation :set_acronym
3
4    protected
5
6    def set_acroynm
```

```
7     self.acronym = "#{firstname.first}#{lastname.first}#{lastname.last}".
          upcase \
8       unless acronym
9   end
10 end
```

6.2 Callbacks à la modification

Les callbacks sont executés dans cet ordre selon les trois scénarios

```
1  A la création:        A la mise à jour:     A la destruction:
2
3  before_validation     before_validation     before_destroy
4  after_validation      after_validation      around_destroy
5  before_save           before_save           after_destroy
6  around_save           around_save
7  before_create         before_update
8  around_create         around_update
9  after_create          after_update
10 after_save            after_save
```

6.3 Callbacks à la lecture

Il existe également les callbacks

`after_initialize` et `after_find`

qui permettent l'execution de code lors de la création d'un objet et lors du rapatriement d'un objet depuis la BD.

Référez vous au guide pour les détails

6.4 Observers

- Similaire au *callbacks*
- Lorsque le code du callback n'est pas directement lié au modèle
- Le code du modèle **n'est pas touché**!

Exemple: envoyer un mail de notification lorsqu'un nouveau commentaire est posté.

Depuis la version 4.0 de ActiveRecord, ce comportement a été déporté dans une **gem** externe: `rails-observers`

6.5 Observers : exemple

```
1 require 'rails/observers/activerecord/active_record'
```

Définition d'un observeur

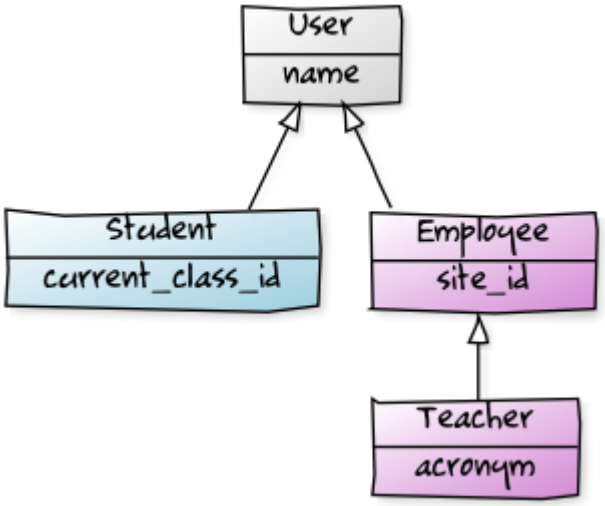
```
1 class CommentObserver < ActiveRecord::Observer
2   observe :comment
3
4   def after_create(model)
5     # envoi du mail de notification
6   end
7 end
```

Enregistrement de l'observer, puis instantiation (à faire une seule fois dans l'application)

```
1 ActiveRecord::Base.observers += [:comment_observer]
2 ActiveRecord::Base.instantiate_observers
```

7 Structures avancées

7.1 Héritage en BD ?




```ruby
class User < ActiveRecord::Base
end

class Student < User
  belongs_to :current_class
end
```

```ruby
class Employee < User
  belongs_to :site
```

```ruby
end

class Teacher < Employee
end
```

Comment implémenter ceci dans la base de donnée?

7.2 STI: Single Table Inheritance

id	type	name	current_class_id	site_id	acronym
1	Employee	Joe	**NULL**	42	**NULL**
2	Employee	Bob	**NULL**	34	**NULL**
3	Teacher	Suzanne	**NULL**	34	SE
4	Teacher	Jack	**NULL**	12	JK
5	Teacher	Nancy	**NULL**	34	NY
6	Student	Nina	123	**NULL**	**NULL**
7	Student	Joe	101	**NULL**	**NULL**

7.3 STI: Intégré à AR

```ruby
>> Student.create(name: 'Alan', current_class: CurrentClass.first)
=> #<Student id: 8, name: "Alan", current_class_id: 101, site_id: nil, acronym
    : nil>
```

```sql
INSERT INTO users
SET type, name, current_class_id, site_id, acronym
VALUES ('Student', 'Alan', 101, NULL, NULL)
```

```ruby
>> Employee.all
=> [#<Employee id: 1, name: "Joe", ...>,
    #<Employee id: 2, name: "Bob", ...>,
    #<Teacher id: 3, name: "Suzanne", ...>, ...]
```

```sql
SELECT users.* FROM users WHERE type IN ('Employee', 'Teacher')
```

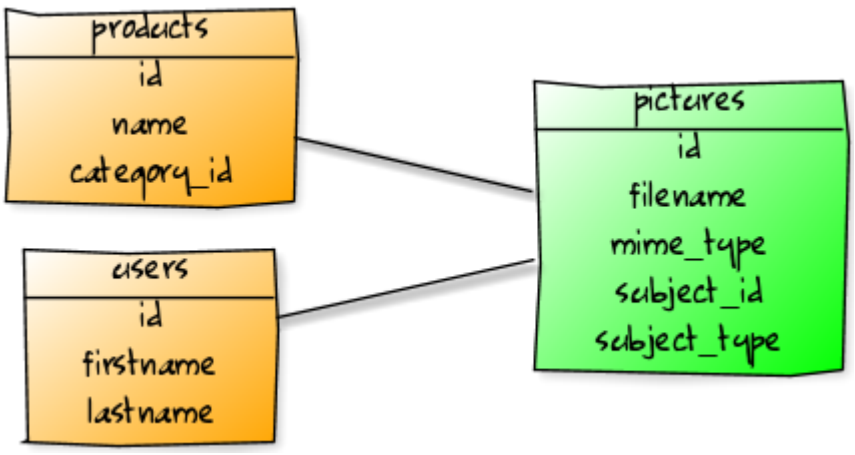
7.4 Associations polymorphiques

Permet d'associer des éléments de types différents
sur une seule association.

Exemple:

- Ajouter une *photo* à un utilisateur
- Ajouter une *photo* à un produit

7.5 Assoc. polymorphiques




```ruby
class Picture < ActiveRecord::Base
  belongs_to :subject, polymorphic: true
end

class User < ActiveRecord::Base
  has_many :pictures, as: :subject
end

class Product < ActiveRecord::Base
  has_many :pictures, as: :subject
end
```