ActiveRecord Associations:

Models in Rails are subclasses of ActiveRecord::Base. Model attributes are inferred from the columns of the table to which the model is tied. Model attributes are immediately updated whenever the schema of the database changes via a migration.

When we have attributes that are not “primitive”, that is, not a string or a number or a date-time, we know that the model that is “owned” carries the foreign key of the model that owns it.

For example, assume we have a Course model and an Instructor model. The attributes we would like to have for these models is a list of Courses for an Instructor and an Instructor for a Course. Since we can’t have complex attributes in our tables, each record in the Courses table will have a foreign key reference to the Instructor that owns it. Records in the Instructors table will have no references to Courses because the number of attributes a Table can have is fixed and the number of Courses an Instructor can have is variable.

Because ActiveRecord infers model attributes from the associated table, we have an instructor attribute for a course (the attribute is actually instructor_id), but we don’t have a list of Courses for an Instructor. When we declare our associations, we get the list attribute for the Instructor model.

ActiveRecord provides the following associations:

belongs_to
has_many
has_and_belongs_to_many
has_one
has_many :through
has_one :through

The most commonly used associations are the first three in the list. Adding the appropriate association to the Course class, we have:

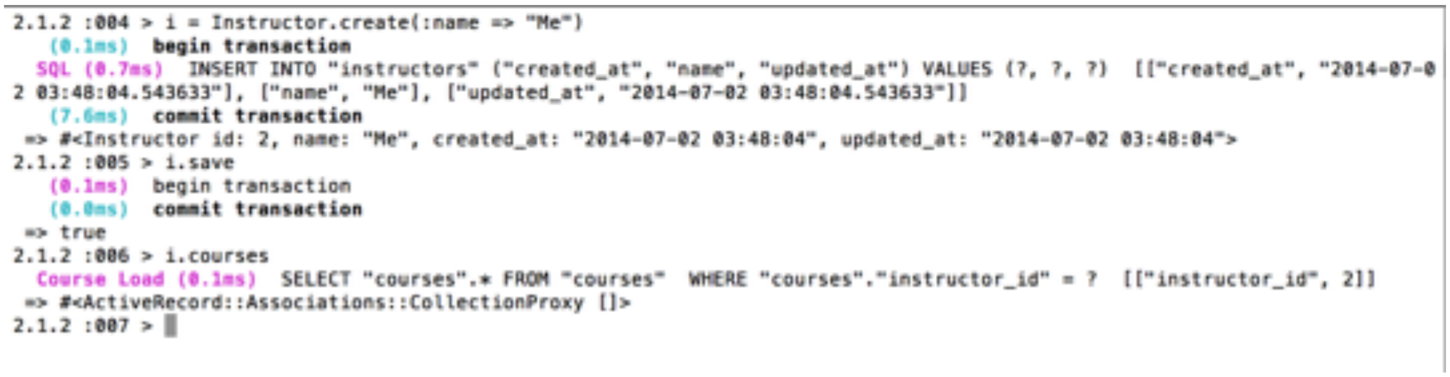
```
# == Schema Information
#
# Table name: courses
#
#  id            :integer          not null, primary key
#  title         :string(255)
#  course_number :string(255)
#  created_at    :datetime
#  updated_at    :datetime
#  instructor_id :integer
#

class Course < ActiveRecord::Base

  belongs_to :instructors

end
```

And the Instructor class looks like:

```
# == Schema Information
#
# Table name: instructors
#
#  id         :integer         not null, primary key
#  name       :string(255)
#  created_at :datetime
#  updated_at :datetime
#

class Instructor < ActiveRecord::Base

  has_many :courses

end
```

We now have a courses attribute for Instructor and a query for it by the same name.

In the rails console, we can now do the following
We know that there is a many-to-many association between Courses and Students and we



```
2.1.2 :004 > i = Instructor.create(:name => "Me")
   (0.1ms)  begin transaction
  SQL (0.7ms)  INSERT INTO "instructors" ("created_at", "name", "updated_at") VALUES (?, ?, ?)  [["created_at", "2014-07-0
2 03:48:04.543633"], ["name", "Me"], ["updated_at", "2014-07-02 03:48:04.543633"]]
   (7.6ms)  commit transaction
 => #<Instructor id: 2, name: "Me", created_at: "2014-07-02 03:48:04", updated_at: "2014-07-02 03:48:04">
2.1.2 :005 > i.save
   (0.1ms)  begin transaction
   (0.0ms)  commit transaction
 => true
2.1.2 :006 > i.courses
  Course Load (0.1ms)  SELECT "courses".* FROM "courses"  WHERE "courses"."instructor_id" = ?  [["instructor_id", 2]]
 => #<ActiveRecord::Associations::CollectionProxy []>
2.1.2 :007 >
```

have the appropriate join table. Adding the “has_and_belongs_to_many” association in both classes, we get free queries for courses on the Student model and students on the Course model.

Subclassing in Databases:

When we would like to subclass our models, we run into the problem that databases do not support inheritance. We have several cases to consider, the first of which is the simplest.

We have a model for a WorkTask for the semester project. A WorkTask can be a homework assignment, studying for a test, or a project. Using object oriented design, we might have three subclasses of a WorkTask, one for each type. This sort of design doesn’t make sense when working with a database because all three subtypes have the same properties. Rather than having three separate tables for each subtype, only one table should be used with a string attribute for the type. This is called “single table inheritance”.

If we have subclasses that have different attributes, there are two ways to represent the subclasses in the database. We can have a separate table for each subtype, and of course, duplication of columns for the attributes they have in common.

Lets assume we have the following models:

Student < User

Instructor < User

Instructors and Students all have a name and uno_id. Let's assume that the two differ in that a student has a gpa and a type attribute, where type is a string that is "grad" or "undergrad". Let's assume that instructors have a salary attribute. Using the separate table model, we have a schema that looks like:

Student	Instructor
id	id
name	name
uno_id	uno_id
type	salary
gpa	

Effectively, we have eliminated sub-typing altogether.

The other way to deal with this is to have a table for each subtype that contains only the attributes that are unique to that subclass. The common attributes are stored in their own table and the subtype records maintain a foreign key reference to the superclass record they are associated with.

User	Student	Instructor
id	type	salary
name	gpa	user_id
uno_id	user_id	

Here, we trade space savings for complexity. To find a particular student or instructor record, two tables must be inspected. Also, using this model, a user can be both a student and instructor at the same time without duplicating any data.

Polymorphic associations:

In our Moodlenoo application, we would like to have 3 types of Documents based on what models they are associated with. First, we would like to have documents such as slide sets that Instructors can post. We would like to have documents that are assignments that differ from regular Documents in that there can be an associated submission. In all, we have 3 document types, Document, Assignment, and Submission. Document objects in this case can be only 1 of the three types at a time.

Our three associations:

A Document object can be associated with a Course.
A Document object can be associated with an Assignment.
A Document object can be associated with a Submission.

A Document can only be one of the three above. If we were working in Java, the Document class would be subclassed and we would instantiate the appropriate object, but we don't have support for polymorphism in the database. We will have to emulate this behavior by using 'polymorphic associations" and nothing but attributes.

We will use some Rails voodoo to create a pseudo-superclass called Attachable. (See the tutorial notes "Scaffolding the rest of the application"). Assignments and Documents will have a foreign key reference to Attachable and a type field that lets us know if it belongs to Assignments or Courses. We can then declare our associations as follows:

```
class Course < ActiveRecord::Base
  …
  has_many :documents, :as => :attachable
  …
end

class Assignment < ActiveRecord::Base
  …
  has_many :documents, :as => :attachable
  …
end
```

Last-minute functionality change (these kinds of things are a fact of life out in the wild):

If we want to allow students to upload multiple Documents as Submissions, this complicates the matter considerably. Does a Submission have many Documents while at the same time being a Document itself? This doesn't make sense so we will leave Submissions out of the polymorphic association, allowing to to remain a singular model associated with an Assignment.