Awards – new.html.erb, Student has many awards, award belongs to student, drop box to select student

```
<%= f.label :student_id %><br />
<%= f.select :student_id, Student.find(:all , :order => "family_name,
given_name").collect {|s| [(s.given_name + " " + s.family_name), s.id]}
%>
```

Auto delete award when student disappears

```
has_many :awards, :dependent => destroy
```

```
map.resources :students, :has_many => [ :awards ]
…/students/3/awards2
```

AwardsController

```
Before_filter :get_student
Def get_student #private
@student = Student.find(params[:student_id])
End
```

Many-to-many join table, has_and_belongs_to_many :courses & :students

```
class CreateCoursesStudents < ActiveRecord::Migration
  def self.up
    create_table :courses_students, :id => false do |t|
             t.integer :course_id, :null => false
             t.integer :student_id, :null => false
    end
    add_index :courses_students, [:course_id, :student_id], :unique =>
true
  end
  end
  def self.down
    remove_index :courses_students, :column => [:course_id,
:student_id]
    drop_table :courses_students
  end
end
```

```
has_many :potato_scores
has_many :movies, :through => :potato_scores
```

Aspect Oriented Programming
• Cross-cutting concerns: logically centralized requirements that may appear in multiple places in implementation
• Advice: specific piece of code that implements a cross-cutting concern
• Pointcuts: places where we want to "run" advice at runtime
• Advice+Pointcuts make up aspect-oriented programming
• Downside to AOP: not necessarily clear what the execution order is because at pointcuts we alter it
• In statically-compiled languages, compiler must "weave" advice into the code at the pointcuts
– Ruby's dynamic nature means weaving is unnecessary!
– But Ruby doesn't allow arbitrary pointcuts
• In Rails: AOP is used for validations, model lifecycle callbacks (e.g. before_save,
after_update, etc), filters

Design Patterns
• Adapter: translates an API to match a client's expectation
– Goal: isolate client from differences in APIs that it might use
– Example: database "adapters" for MySQL, Oracle, etc.
• Proxy: implements same methods as real API
– Goal: hide remoteness, defer work
– Example: intercept a bunch of bank deposit requests; send the batch of deposits all at once
• Façade: provides a wrapper around an elaborate API, exposing only what's needed
– Goal: simplify client's interaction with API
– Example: wrapper around AWS EC2 API that just lets you start and terminate instances
• Null Object: stand-in for a real object on which "important" methods of real object can be called
• Example: "the logged-in customer"
– Can ask Customer.null_customer whether it's logged in, its name, etc.
– Benefit: only one code path (don't always have to check whether user is logged in)

• Tie to Singleton: you may only want ONE copy of this null customer
– Def: a class that provides only one instance, which anyone can access
– It's still a member of the base class, but immutable and singular
• Though these often go together, know the point of each!

Composite
• Component whose operations make sense on both individuals and aggregates
• Example (from Design Patterns in Ruby): tasks involved in baking a cake
– High-level task: MakeBatter
– Decomposes into AddDryIngredients, AddLiquids, Mix
– Shared functionality:
– Has an estimated duration
– Can be added to process
• Can be recursive: MakeBatter is a sub task of MakeCake

Observer
• Problem: entity O ("observer") wants to know when certain things happen to entity S ("subject")
• Don't want to rewrite S every time a new observer arrives
• Example from HW4: ActiveRecord::CacheSweeper (for both models and controllers)
– EventSweeper is the Observer
– Event is the Subject

Security
– SQL injection: when user input is unfiltered, allowing user to construct arbitrary SQL statement
– Cross-site Request Forgery: unauthorized requests submitted on behalf of a user who is authenticated with a webpage
• Exploits trust a site has for authenticated users
• More attacks:
– Cross-site Scripting: vulnerability where malicious JS injected into a user's trusted webpage, bypassing browser security model
– Buffer Overflow: bug where a program overwrites memory adjacent to a limited-length buffer
– Denial of Service: attempt to make a site unavailable to users, usually by overwhelming it with bogus traffic
– Rails can "sanitize" text input by the user
• h() function escapes HTML
• Good coding practices in SQL queries:
– Project.find(:all, :conditions => ["name = ?", params[:name]]) <- good
– Project.find(:all, :conditions => "name = '#{params[:name]}'") <-bad

JavaScript
• Dynamic, OO language embedded in web browsers
– First class functions/lambdas, "interesting" OO implementation
• Enables access to browser's rendering engine and DOM
– Handlers associate JS functions with Events on DOM elements: onClick, onMouseOver, etc.
• XMLHTTPRequest: allow browser to make asynchronous requests in background, while page is loaded
– DOM: Document Object Model, accessed through window.document
– Usual idiom: m = document.getElementByID("stuff"); then do something to/with m
– Handlers on elements "trigger" JS by calling functions when event occurs

JavaScript & Rails
• Rails provides libraries to smooth out browser differences (e.g. Prototype library)
• Also provides helpers that generate JS
– "Raw" JS requires many steps to do XHR
– Rails tries to simplify using generators & controller methods
– Usual thing: use methods in controller to respond to XHR requests & render partials, use link_to_remote or similar in view
• Concentrate on knowing when to use JS, not details on Rails helpers for using JS

When to use JavaScript?
• Some things can be done using just CSS
– e.g. changing link color when hovering above a link
– e.g. having different styling for print vs. screen
• Interactive UIs
– Many things can be done w/JS without XHR

• e.g. disabling form fields depending on what other fields are set as ("Shipping Address same as Billing Address?")
– Is the UI adding value or just annoying?
• Sometimes minimalism provides a better user experience
• XHR
– Provide rich experiences for users
• Gmail, Google Docs
– Beware: server load, "What's the URL" problem

Service Level Objective (SLO)
• Time to satisfy user request ("latency" or "response time")
• Rather than average or worst case, specify target response time for X% of users
• Includes %ile, target response time, time window
• You should know what each of these colored lines means (eg, 99% of users have response time <= 64.55 ms)
• Don't forget that the time window over which the requests are observed is important!

Scaling
• How to respond to large user demand?
• Big idea: Replicate stateless components ("horizontal scalability")
• Goal: interchangeability (send any request to any server)
o Requirement: app servers must be RESTful
• Very difficult to scale the database => it will become the bottleneck
• Can use EC2 servers => don't have to manage your own infrastructure; easy to scale out by requesting more instances

Caching
• Types of caching:
– Browser caching
– Rails caching
• Good resource: http://guides.rubyonrails.org/caching_with_rails.html
• Types:
– Page caching (HW4)
– Action caching
– Fragment caching
– Query caching
• Main benefit: speed
• Main drawback: inconsistency => need to manage cache expiration
• You should know when to use each type of caching

REST + Caching
• Cacheability is an aspect of RESTfulness:
• From REST Wikipedia article: "Clients are able to cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance."
• Main idea: server tells client (ie, browser) whether the item is cacheable and how long it is expected to be valid

Page Caching vs. Action Caching
• Both apply if you want to cache a whole page
• Both are specified in controller
• Page caching (caches_page): request can be fulfilled at web server without touching Rails stack (ie, doesn't even hit controller)
– Benefit: Very fast!
– Drawback: Can't be used when you need authentication
• Action caching (caches_action): incoming request does go to Rails stack => can run before filters
– Benefit: Allows authentication; still serves request from cached copy

Fragment Caching
• Lets you mix real-time and cached content
• Important because dynamic sites build pages from a variety of content (eg, news articles, ads, logo banner, etc.)
– Some changes infrequently => good candidate for caching
– Some is very dynamic/requires authentication => bad candidate for caching
• Happens in view

Query Caching
• Main Idea:
– Cache query results at database
– When a new query comes in, see if it can be served from cache (ie, see if it's been executed before).

– Reduces database load

Validations
validates_length_of :secret, :in => 6..24
validates_format_of :secret, :with => /[0-9]/,:message => "must contain at least one number"
validates_format_of :secret, :with => /[a-z]/,:message => "must contain at least one lower case character"
validates_format_of :email,
:with => /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\Z/i,:message => "doesn't look like a proper email address"

Mocks/Stubs
my_mock = mock(name) #name = class
MyClass.stub!(:msg).and_return(value)
A.stub!(:msg).and_return(:default_value)
A.should_receive(:msg).with(:arg).and_return(:special_value)
Can substitute MyClass or A with my_mock

SOFA (short, do one thing, few arguments, consistent lvl of abstraction)

Blocks (anonymous λ)
(map '(lambda (x) (+ x 2)) mylist)
mylist.map { |x| x+2 }
(filter '(lambda (x) (even? x)) mylist)
mylist.select do |x| ; x.even? ; end
(map
'(lambda (x) (+ x 2))
(filter '(lambda (x) (even? x)) mylist))
mylist.select {|x| x.even?}.map {|x| x+2 }
Internal Iterators as a special case
me = "Armando"
each_state do |state|
puts "#{me} would hate to live in #{state}"
end
def each_state
states = ['AK','AL','AR', ..etc.. ,'UT','WY']
states.each do |state|
yield(state)
end
end

Modules
A module can have no instances.
A module can have no subclasses.
module Trig
  PI = 3.141592654
  def Trig.sin(x)
  end
  def Trig.cos(x)
  end
end

Yield
Iterator fib(n) Usage: fib(5){ |x| puts x} "0, 1,2,3,4,5"
Def fib(n)
X=0 Y=1
n.times do
yield (x)
x,y=y,x+y
end end

def test
  puts "You are in the method"
  yield
  puts "You are again back to the method"
  yield
end
test {puts "You are in the block"}
This will produce following result:
You are in the method
You are in the block
You are again back to the method
You are in the block