Jasmine

```
expect {...}.to eq "foo" -->
expect(expr).toEqual("foo").toBeTruthy(),.toBeFalsy()
           .toBeHidden().toHaveClass().toContainText()
<toBeSelected(), toBeChecked(), toBeDisabled(),
toBeVisible(), toBeHidden(), toHaveClass("foo"),
toHaveId("foo"),
toHaveAttr("href", "http://saasbook.info")>
Describe('Clicking Hide button', function() {
 it('hides Movie div', function() {
   $('a#hide').trigger('click');
   expect($('div#movie')).toBeHidden();
 });
});
```

Create "spy" method that replaces real method that is a property of an object.

```
spyOn(MoviePopup,'new').andReturn(value).andCallThrough()
.andCallFake(func)
expect(MoviePopup.new.mostRecentCall.args).toContain("Gra
vity")
expect($.ajax.mostRecentCall.args[0]['url']).toEqual("/mo
vies/1")
```

HTML Fixtures

Provide enough HTML for JS code to do its thing in a familiar environment.

```
<table id="movies"></table>
loadFixtures('movie_row.html') // loads this into
div#jasmine-fixtures
Var htmlResponse = readFixtures('movie_info.html')
spyOn($,'ajax').andCallFake(function(ajaxArgs) {
 ajaxArgs.success(htmlResponse, '200');
});
```

Design Patterns

Design Patterns promote reuse.
Gang of Four (GoF) Patterns: Structural, creational, behavioral.
Pattern != full design. It is more like a blueprint for a design.
Meta-patterns: Separate out the things that change from those that stay the same.
Antipattern: Code that looks like it should probably follow some design pattern, but doesn't.
Symptoms of Antipatterns: Viscosity (Easier to do hack than right thing), inmobility (can't dry out functionality), needless repetition (from inmobility), & needless complexity from generality.

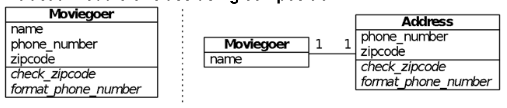
SOLID

Motivation: Minimize cost of change.
Single Responsibility, Open/Closed, Liskov Substitution, Injection of Dependencies, & Demeter.

Single Responsibility Principle

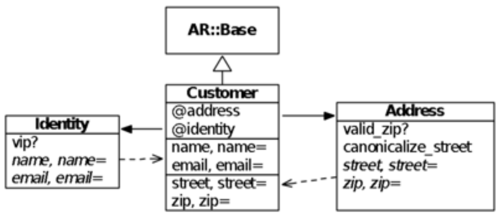
A class should have one and only one reason to change.
What is class's responsibility <25 words?
Models with many sets of behaviors.
Code smell: Lack of Cohesion of methods:
LCOM=1 - sum(MVi)M*V (between 0 & 1)
M = # instance methods.
V = # instance variables.
MVi = # instance methods that access the i'th instance variable.
LCOM-4: # of connected components in graph where related methods are connected by an edge
High LCOM suggests possible SRP violation.
Relationship between ActiveRecord tables & classes needn't be 1:1.

Extract a module or class using composition:



```
class Customer < ActiveRecord::Base
 composed_of :customer_address,
   :mapping => [['adr_street', 'street'], ['adr_city',
'city'], ['adr_zip','zip']]
end
class CustomerAddress
 attr_reader :street, :city, :zip
 def initialize(street,city,zip) ;  @street,@city,@zip =
street,city,zip ; end
```

Extract a module or class using delegation:



```
class Customer < ActiveRecord::Base
def initialize
   @address = Address.new(self)
 end
end
class Address
 def initialize(customer)
   @customer = customer
 end
 attr_reader :customer
 delegate :zip, :zip=, :street, :street=, :to =>
:customer
End
```

Open/Closed Principle

Classes should be open for extension, but closed for source modification.

```
Class Report
 Def output_report
   case@format
   When :html
     HtmlFormatter.new(self).output
   When :pdf
     PdfFormatter.new(self).output
```

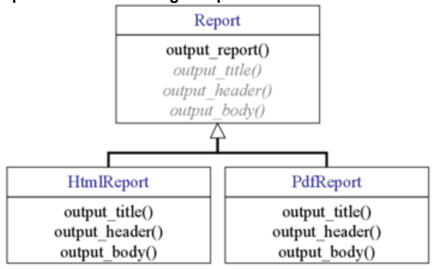
Can't extend without changing report base class.

DRYing out construction with Abstract Factory Pattern

```
class Report
 def output
   formatter_class =
     begin
       @format.to_s.classify.constantize
     rescue NameError
       # ...handle 'invalid formatter type'
     end
   formatter = formatter_class.send(:new, self)
 end
end
```

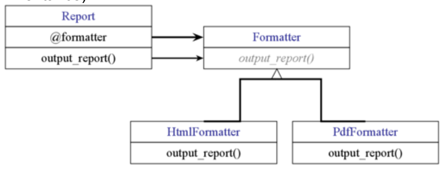
Template method: Set of steps is the same, but implementation of steps different. (Inheritance: subclasses override abstract "step" methods).
Strategy: Task is same, but many ways to do it (composition: component classes implement whole task)

Report Generation using Template Method:



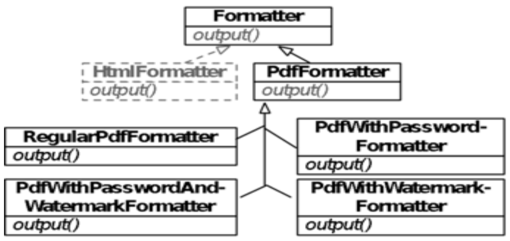
```
Class Report
 Attr_accessor :title, :text
 Def output_report
   Output_title
   Output_header
   Output_body
 End
End
Class HtmlReport < Report
 Def output_title … end
 Def output_header … end
End
Class PdfReport < Report
 Def output_title … end
 Def output_header … end
End
```

Report Generation using Strategy (Prefer composition over inheritance)



```
Class Report
 Attr_accessor :title, :text, :formstter
 Delegate :output_report, :to => :formatter
End
```

DRYing out extension points with Decorator Pattern:



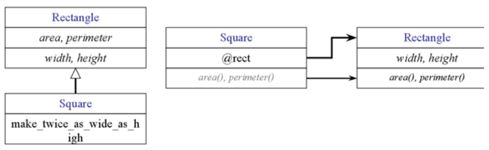
```
class PdfFormatter
 def initialize ; ... ;
end
 def output ; ... ; end
end
class PdfWithPasswordFormatter < PdfFormatter
 def initialize(base) ; @base = base ; end
 def protect_with_password(original_output) ; ... ; end
 def output ; protect_with_password @base.output ; end
end
class PdfWithWatermarkFormatter < PdfFormatter
 def initialize(base) ; @base = base ; end
 def add_watermark(original_output) ; ... ; end
 def output ; add_watermark @base.output ; end
 end
end
# If we just want a plain PDF
formatter = PdfFormatter.new
# If we want a "draft" watermark
formatter =
PdfWithWatermarkFormatter.new(PdfFormatter.new)
# Both password protection and watermark
formatter = PdfWithWatermarkFormatter.new(
 PdfWithPasswordFormatter.new(PdfFormatter.new))
```

Can't close against all types of changes, so have to choose.
Agile methodology can help expose important types of changes early.

Liskov Substitution Principle

A method that works on an instance of type T, should also work on a subtype of T.
If can't express consistent assumptions about "contract" between classe & collaborators, likely LSP violation.
Symptom: Change to subclass requires change to superclass (shotg surgery/refused bequest)

```
class Square
 def initialize(side,top_left_corner)
   @rect = Rectangle.new(side,side,top_left_corner)
 end
 def area ; @rect.area ; end
 def perimeter;@rect.perimeter; end
 def side=(s) ; @rect.width = @rect.height = s ; end
end
```



Demeter Principle

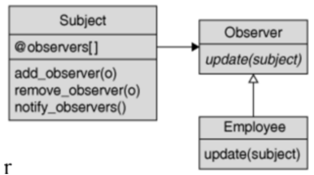
Only talk to your friends...not strangers
You can call methods on yourself and your own instance variables, b not on the results returned by them.
Code smell: Mock Train Wreck
Solutions: Replace method with delegate.

```
class Wallet
 attr_reader :cash # no longer attr_accessor!
 def withdraw(amount)
    raise InsufficientFundsError if amount > cash
    cash -= amount
    amount
 end
end
class Customer
 # behavior delegation
 def pay(amount)
   wallet.withdraw(amount)
 end
end
class Paperboy
 def collect_money(customer, due_amount)
   @collected_amount += customer.pay(due_amount)
```

Separate traversal from computation (Visitor)
Be aware of important events without knowing implementation detail (Observer)

Observer

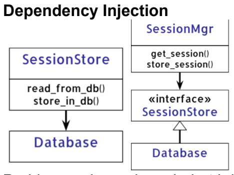
Problem: entity O ("observer") wants to know when certain things happen to entity S ("Subject")



r
Design issues: Acting on events is O's concern -- don't want to pollut S. Also, Any type of object could be an observer or subject -- inheritance is awkward.
Example use cases: Full-text indexer wants to know about new post auditor wants to know whenever sensitive actions are performed by admin.
Example: Maintaining Relational Integrity.
Problem: Delete a customer who "owns" previous transactions (ie foreign keys point to her).
Solution: Merge with "the unknown customer"
AR provides built-in hooks for Observer design pattern.

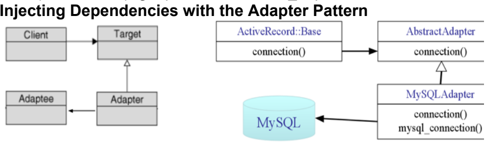
```
# in config/environment.rb
Config.active_record.observers = :customer_observer
Class CustomerObserver < ActiveRecord::Observer
 Def before_destroy … end
End
```

Dependency Injection



Problem: a depends on b, but b interface & implementation can change, even if functionality stable.
Solution: "inject" an abstract interface that a & b depend on. If not ex match, Adapter/Facade Inversion: b (& a) depend on interface vs. a depending on b.
Ruby equivalent: Extract Module to isolate the interface.
Bad (in view): `@vips = User.where('group="VIP"')`
Better: `@vips = User.find_vips`
Best (in controller): `@vips = User.find_vips`

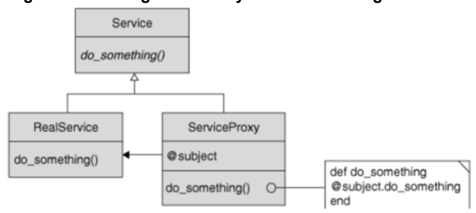
Injecting Dependencies with the Adapter Pattern



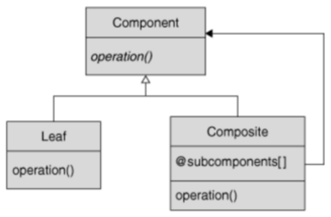
Problem: client wants to use a "service", but the service doesn't do EXACTLY what the app wants, needs slight alterations.
IE: Using either service ConstantContact or MailChimp.
Facade: Adapter is a facade if it may unify distinct underlying APIs in a single, simplified API (like jQuery).

Null Object Facade

Problem: Want invariants to simplify design, but app requirements seem to break this.
Null object: Stand-in on which important methods can be called.
```
@customer = Customer.null_customer
@customer.logged_in? // false
@customer.last_name // "ANONYMOUS"
@customer.is_vip? // false
```
Singleton: Ensuring there's only one of something.

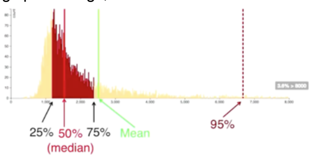


A class that provides only one instance which anyone can access. It is a member of the base class, but immutable & singular.
```
class Customer
 def name ; @name ; end
 def name=(newname) ; @name = newname ; end
 def self.null_customer
   @@instance ||= Customer.new
   # put the singleton in its own class!
   class << @@instance
     def name ; "ANONYMOUS" ; end
     def name=(new) ; raise "Can't change name of null
customer " ; end
     def logged_in? ; false ; end
   end
   @@instance
```
Proxy
Implements same methods as real service object, but intercepts each call.
Do authentication/protect access, defer work (be lazy), like when sending a mail but offline.
Composite



Component whose operations make sense on both individual & aggregates.
Ex: Regular tickets, VIP tickets, & subscriptions all have a price and can be added to order in common. But, Regular & VIP tickets are for a specific show yet subscription has to track which ticket it "owns".
```
class RegularTicket < Ticket
attr_accessor :price, :show_date
def add_to_order ... end
def refund ... end
end
class MultiTicket < Ticket
def initialize
  @tickets = []
  super
end
attr_reader :tickets
def add_ticket(t)
  @tickets += t
end
def price
  @tickets.sum { |t| t.price }
end
def add_to_order
  @tickets.each { |t| t.add_to_order }
end
```
Single-table Inheritance: Stores objects of diff subclasses (but same parent class) in same table.
Continuous Integration & Deployment
Automation: Consistent deploy process. PaaS sites like Heroku already do this.
Continuous Integration: Integration-testing the app beyond what each developer does.
Continuous Deployment: Push → CI → deploy *several times per day*.
Rational: risk == # engineer-hours invested since last deploy.
Releases are useful as customer-visible milestones.
Availability & Response Time
Gold standard for availability: %99.999 ("five nines") = 5 min per year. ("four nines" = 50 min/year)
Response time is how long before response received. Dominated by latency, not bandwidth.
< 100 ms is "instantaneous" & > 7 sec is abandonment
Graph on right is typical num requests vs response time. Care about most users, not average user.
SLO: Service Level Objective. Time to satisfy user request.
Must specify %ile, target response time, & time window (99% < 1 sec over a 5 min window).
SLA: Service Level Agreement is a SLO to which provider is contractually obligated.
Apdex: Simplified SLO. Given a threshold latency T for user satisfaction:
Satisfactory request take t<=T. Tolerable requests take T <= t <= 4T
Apdex =
# satisfactory + .5*# tolerable# requests ### (0.85-0.93 considered good)
(Can hide systematic outliers if not used carefully -- thus controversial -- can use multiple Apdex)

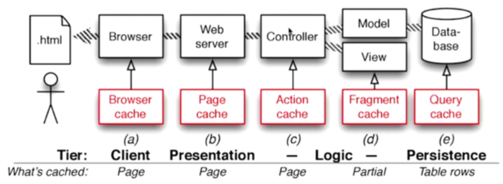
If site slow, just throw more computers at it (If site is large, that might be harder to do)
Tag specific commits with release names: git tag 'happy-hippo' HEAD; git push --tags;



Upgrading code & migrations from n to n+1
Naive update (easy & right way)
Take service offline; apply destructive migration, including data copying; deploy new code; bring service back online. ** This may result in unacceptable downtime.
Incremental upgrades with feature flags
Feature flags:
• Preflight checking: gradual rollout of feature to increasing numbers of users e.g. to scope for performance problems
• A/B testing
• Complex feature whose code spans multiple deploys
• rollout gem covers these cases and more
• Undoing an upgrade: use feature flags instead
—> down-migrations are primarily for development, disasters (not thoroughly testes, not reversible, not sure someone else applied an irreversible migration)

1.Do nondestructive migration
`class SplitName1 < ActiveRecord::Migration`
`  def up`
`    add_column 'moviegoers', 'first_name', :string`
`    add_column 'moviegoers', 'last_name', :string`
`    add_column 'moviegoers', 'migrated', :boolean`
`    add_index 'moviegoers', 'migrated'`
`  end`
`end`
2.Deploy method protected by feature flag
`class Moviegoer < ActiveRecord::Base`
`  # heres version n+1, using Setler gem for feature flag`
`  scope :old_schema, where :migrated => false`
`  scope :new_schema, where :migrated => true`
`  def self.find_matching_names(string) .. end`
`  # auto update records to new schema when they're saved`
`  before_save :update_schema, :unless = lambda { |m| m.migrated? }`
`  def update_schema … end`
3.Flip feature flag on; if disaster, flip it back.
4.Once all records moved, deploy new code without feature flag
5.Apply migration to remove old columns

If something bad happened during migration, don't use down-migrate. Use feature-flags.
Can also use feature-flags for A/B testing, pre-flight-checking, complex features.
Caching



Page caching: Bypasses controller action: caches_page :index
Action caching runs filters first.

Bad cache	Better cache
Caches_page :index Def index If logged_in ? … Else Redirect_to login_path End end	Caches_page :public_index Caches_action :logged_in_index Before_filter :check_logged_in? :only => 'logged_in_index' Def public_index … End Def logged_in_index … end

Fragment caching for views: Caches HTML resulting from rendering part of a page:
```
- Cache "movies_with_ratings" do
 = render :collection => @movies
```
How do we detect when cached versions no longer match database?
```
Class MovieSweeper < ActionController::Caching::Sweeper
 Observe Movie
 # if a movie is created or deleted, movie list becomes
invalid & rendered partials become invalid.
 Def after_save(movie) ; invalidate ; end
 Def after_destroy(movie) ; invalidate ; end
 Private
 Def invalidate
   Expire_action :action => ['index', 'show']
   Expire_fragment 'movies_with_ratings'
```
N+1 queries problem
You are doing n+1 queries to traverse an association, rather than 1 query
```
@fans = Moviegoer.where("zip = ?", code)
@fans.each.do |fan|
 @fans.movies.each do |movie|
   // BAD: each time thru this loop causes a new database
query!
```
Fix with: `@fans = Moviegoer.where("zip = ?", code).includes(:movies)`
Indices make DB calls faster.
SSL protects communication of data from eavesdroppers. No protection in db. (Diffie Hellman)
Protect against **CSRF** by placing csrf_meta_tags in application.html.haml & protect_from_forgery in ApplicationController.
```
//assume jQuery has been loaded.
//e.val() gets value of jQuery -wrapped DOM elemente;
//e.val(newVal) sets new value
MyStore.compute_total_purchase = function(){
    var total = 0.0;
    var taxRate = parseFloat($('#tax').val())/100.0;
    var subtotal = $('.price').each(function(elt){
        //each 'price' field is a float number
        total += parseFloat(elt.val());
    }
    total += (total*taxRate);
    $('#total').val(total);
}
```
// Jasmine test for the compute_total_purchase function
Create an HTML fixture (snippet of HTML) to use with Jasmine that includ one or more .pricetextfields, a #taxtextfield, and a #totaltextfield. Do variou tests where you first populate .priceand #tax, then call compute_total_puchase,and verify the result in #totalis correct.
```
$(document).ready(function(){
    $('#tax').change(MyStore.compute_total_purchase);
});

//new and improved to match #tax, #total, .price:
$(document).ready(function(){
$('body#checkout#tax').change(MyStore.compute_total_pur
ase) //...others such as$'(#checkout#tax').change(...)
});
```
AJAX app:
+ Likely faster page load time, since user's browser can open various JavaScript connections in parallel
+ Much less load on server since it won't spend time waiting for responses from sources
– Higher load on sources even though they're returning same info over and over during repeated visits over short timescales
– Testing client code requires stubbing/Webmocking multiple sources
Conventional Rails app:
+ Can take advantage of caching to suppress refresh of sources at the expen of having sometimes-stale information
+ Easy to test by stubbing out all sources at server
– Hard to get concurrency across requests to sources
//a simple fix that repairs the vulnerability.
```
@movies = Movie.where(["titleLIKE?",'%'+title_words+'%'

class Movie < ActiveRecord::Base
  has_many:reviews
  def average_review_score
    self.reviews.average('potatoes')
  end
end

class Review < ActiveRecord::Base
  belongs_to: movie
  after_save: update_average_score_for_movie
after_destroy:update_average_score_for_movie
  def update_average_score_for_movie
    movie.update_attributes(:average_score=>movie
average_review_score)
  end
end
```
```
•(alias for jQuery())
•Select elements:  $('p  .myclass')
•Give elements secret jQuery powers:
          this --> $(this)
          document.window --> $(document.window)
•Create elements:var elt = $("<span>Hola, mundo</span>")
•Run a function when document ready:$(RP.setupFunc)
•Select elements with $() (or wrap to give them secret
jQuery powers)
•Inspect them…
          text() or html()
          is(:checked), is(:selected), etc.
          attr('src')
•Add/remove CSS classes, hide/show
•Create setup function that binds handler(s) on element
 - common ones: onclick, onsubmit, onchange
```
AJAX: Asynchronous Javascript And Xml
```
•JSAPI call XmlHttpRequest (a/k/a xhr) contacts server
asynchronously (in background) and without redrawing pa
- Normal HTTP request, w/special header:
          X-Requested-With: XmlHttpRequest
•Controller action receives request via route
•What should it render in response?
          render :layout => false
          render :partial => 'movies/show'
          render :json => @movies  (calls to_json)
          render :xml => @movies  (calls to_xml)
          render :text => @movie.title
          render :nothing => true

$.ajax({type: 'GET',
        url: URL,
        timeout: milliseconds,
        success: function,
        error: function
            // many other options possible
      });

e.g. Server side:
class MoviesController < ApplicationController
  def show
    @movie = Movie.find(params[:id])
    render :partial=>'movie', :object=>@movie if
      request.xhr?
    # else render default (show.html.haml)
  end
end
```