# Cucumber and RSpec

* 1st step is thinking of “code you wish you had”, assuming methods that if existed would make it a perfect match to the user story.
* Rather than inside – out (start with building blocks and compose to provide desired functionality), go from users outside-in, to reduce wasted coding; e.g., until get to user view, won’t know what you really need. Especially Web apps, since easy to see what the user is doing.
* Both cycles involve taking small steps and listening to the feedback you get from the tools.
* We start with a failing step (red) in Cucumber
  + (the outer cycle). To get that step to pass, we’ll drop down to RSpec
  + (the inner cycle) and drive out the underlying code at a granular level
  + (red/green/refactor).
  + At each green point in the RSpec cycle, we’ll check the Cucumber cycle.
  + If it is still red, the resulting feedback should guide us to the next action in the RSpec cycle.
  + If it is green, we can jump out to Cucumber, refactor
  + if appropriate, and then repeat the cycle by writing a new failing Cucumber step.

# RSpec

* RSpec can be also be used for integration but prefer Cucumber since if facilitates dialogue with the customer and automates acceptance tests
* Opinions vary on whether to test the views
  + Our position: views are user-facing, so use user stories to test => Cucumber

# Example – calling TMDb

* This corresponds to the BDD part of the picture we just saw of how BDD + TDD work together.
  + the step in RED requires new code to be added to the app, so rest of this section of course is how to use TDD to create that code.
* What code do we need?
  + We need a new route, new controller action to receive the Search form, and a new view to render results.

# The code you wish you had

* step 3 is the sad path – we'll do later.
  + for now concentrate on happy path.
* Suppose controller method ALREADY existed.
  + Pastebin example shows the code that would test steps 1 & 2 on slide.
* WALK THRU:
  + line 1 loads helper methods used by all tests; in general should be first line of any spec file
  + line 3 says that this group of tests is about the MoviesController class
  + since that controller has many actions, line 4 says this subgroup of tests is about the functionality of searching tmdb.
  + lines 5-7 are placeholders for the actual test cases, which we'll do next.

# Example: calling TMDb

* This corresponds to the BDD part of the picture we just saw of how BDD + TDD work together.
* The step in RED requires new code to be added to the app, so rest of this section of course is how to use TDD to create that code.
* What code do we need?
  + We need a new route, new controller action to receive the Search form, and a new view to render results.

# TDD for the controller action

* Replacing hardwired method (which was a hack to make sad path of "add movie from TMDb" work in Cucumber) with an empty method,
  + since we will use TDD to drive the creation of the code in that method

# What model method? – Code walkthrough

* start with line 7. RSpec provides a "post" method that simulates posting a form.
  + the first argument is a URI; it will be looked up in routes.rb just like any other URI, as we've seen before.
  + the second argument (a hash) is what will get stuffed into the params[] hash.
  + so the effect of line 7 is just as if someone had filled out a form that had a field named 'search\_terms', entered the word 'hardware' in that field, and clicked Submit.
* \*\* THIS IS AN IMPORTANT CONCEPT: \*\*
  + line 6 sets up an \_expectation\_ of what should happen when line 7 is executed.
  + should\_receive ***replaces*** any existing method called 'find\_in\_tmdb' in the Movie class, with a "stub" method whose sole job is to monitor whether it gets called. (
    - In this case, Movie.find\_in\_tmdb doesn't exist yet, so the "stub" method is the ONLY method, but even if the find\_in\_tmdb method existed, we'd still want to override it here, because we want to isolate the behavior of THIS test from any possible bugs in find\_in\_tmdb.
* the 'with' method further enforces that not only should find\_in\_tmdb get called, but what argument it should receive.
* the net effect is we have a test that checks whether the search\_tmdb controller action, when triggered, tries to call the model method that we will eventually create.

# Seams

* this kind of seam is possible in Ruby because of open classes: you can add or change behaviors of any method at any time, even in another class.
* RSpec takes advantage of this to make seams very easy to create.
* You can create a seam almost anywhere to isolate some code under test from the other methods it needs to collaborate with.

These notes are incomplete

Be sure to define functional tests – those test that exercise a well-defined subset of code