### Objective:

### As the game stands now, you've probably noticed that it feels totally rigged. Every time you play it, the same player wins. And that's because our code always does the exact same thing: each player gets the same number of w00ts or blams. In other words, there is only one path through the game:

### **def** **play** @players.each **do** |player| player.blam player.w00t player.w00t puts player **end** **end**

### Write a Test First

In the previous exercise, we learned how to write unit tests to check code that we already had. Now we're going to turn things around and write the test *first*. In other words, we'll start by expressing what we want our code to do in the form of expectations. Then we'll write the code that fulfills those expectations. Programming this way has several benefits:

* Writing the test first forces you to think through how the code should behave from the *outside* before diving into an implementation. In other words, writing the test first is a design activity. It helps you stay true to good OO principles such as encapsulation.
* When you write a test first, you're starting with a measurable goal. That helps focus you on the problem you're trying to solve. And you know exactly when you're done: when the test passes! In other words, the tests express how the code should behave *when we're finished*.
* If you write the tests first, then you always have safety nets for your code. You don't have to remember to go back in and add unit tests later because they were there from the beginning. And when you always have tests for everything, then you can safely make changes without the fear of breaking something. And trust us, that kind of confidence is essential to being a productive programmer.

We'll start by practicing TDD on a basic comparison: determining whether a player is strong or wimpy. Then once we're comfortable with the rhythm of TDD we'll apply it to the original objective.

So first off, it would be handy to know if a player is strong or wimpy. Let's say that a strong player has a health value greater than 100 and a wimpy player has a health value less than or equal to 100. Who's responsible for making that decision? Well, it's based on a player's health, so it seems logical that the decision should be the responsibility of the Player class. So we'll start by expressing our expectations in the player\_spec.rb file.

In the player\_spec.rb file, create a context that sets up a player with an initial health of 150. By our measure, this is a strong player. So write a code example that expects the result of calling a strong? method on the player to return a truevalue. Remember, methods that end with a question mark (?) generally return a boolean result. Calling them is like asking a yes or no question. Writing this spec seems strange since the Player class doesn't have a strong? instance method yet, but just go with it for now.

context “with a health greater than 100” do  
 before do  
 @player = Player.new(“larry”, 150)  
 end  
 it “is strong” do  
 @player.strong?.should be\_true  
 end  
end

Remember that a context needs to be defined *inside* of the existing describe block. Here's the full spec file at this point if you need to check the nesting.

require\_relative 'player'  
  
describe Player **do**  
  
 before **do**  
 @initial\_health = 150  
 @player = Player.new("larry", @initial\_health)  
 **end**  
  
 it "has a capitalized name" **do**  
 @player.name.should == "Larry"  
 **end**  
  
 it "has an initial health" **do**   
 @player.health.should == 150  
 **end**  
  
 it "has a string representation" **do**   
 @player.to\_s.should == "I'm Larry with a health of 150 and a score of 155."  
 **end**  
  
 it "computes a score as the sum of its health and length of name" **do**   
 @player.score.should == (150 + 5)  
 **end**  
  
 it "increases health by 15 when w00ted" **do**  
 @player.w00t  
  
 @player.health.should == @initial\_health + 15  
 **end**  
  
 it "decreases health by 10 when blammed" **do**  
 @player.blam  
  
 @player.health.should == @initial\_health - 10  
 **end**  
   
 context "with a health greater than 100" **do**  
 before **do**  
 @player = Player.new("larry", 150)  
 **end**  
   
 it "is strong" **do**  
 @player.strong?.should be\_true  
   
 *# or if using Rspec 3.0:*  
 *# @player.strong?.should be\_truthy*  
 **end**  
 **end**  
**end**

Run the spec, and it should fail. No surprise here: it fails because we haven't implemented the strong? method yet. But now we have a goal...

Next, over in the Player class, write just enough code to get the spec to pass.

class Player  
 def strong?  
 true  
 end  
end

Run the spec again, and this time it should pass!

Now that we have a passing test, we can safely refactor. The strong? method couldn't be simpler, so there's nothing to do there. However, there is a way to clean up the spec slightly.

RSpec adds some syntactic sugar when writing expectations for predicate methods (methods such as strong? that end in a question mark). Go ahead and replace your expectation with this variation:

it "is strong" **do**  
 @player.should be\_strong  
   
 *# or use alternate expectation syntax:*  
 *# expect(@player).to be\_strong*  
**end**

When you call be\_strong, RSpec turns around and calls the strong? method on the @player object. And because we use should, it expects the result of calling the method to return true.

After refactoring, it's important to re-run the specs to make sure we didn't inadvertently break anything. So run the specs again and make sure your code is still green.

OK, that's a good start, but clearly we just faked enough code to get the test to pass. And we know that not all players are strong. So next we need a test for a wimpy player. Doing that will help drive out something more realistic...

In the player\_spec.rb file, create another context that sets up a player with an initial health of 100. Then write a code example that expects the result of calling the strong? method on the player to return a false value.

context “with a health of 100 or less” do  
 before do  
 @player = Player.new(“larry”, 100)  
end  
  
it “is wimpy” do  
 @player.should\_not\_be\_strong  
end  
end

Run the spec, and it should fail. Again, no surprise here: it fails because currently the strong? method *always* returns true.

You know what to do. Back over in the Player class, change the strong? method so that *both* specs pass.

class Player  
 def strong?  
 @health > 100  
 end  
end

Finally, check to see if there's an opportunity to refactor.

That gave us a taste of test-driven development, which is often summarized in three steps: red, green, refactor.

Add Conditional Game Play

Now that you've been through a couple TDD cycles, let's apply it to a slightly more sophisticated conditional. Returning to our original objective, we want to determine what happens to each player based on the roll of a die:

* If a low number (1 or 2) is rolled, we'll blam the player because low rollers aren't very lucky.
* If a medium number (3 or 4) is rolled, we'll skip the player—we won't change their health value. Instead, we'll print out something like "Curly was skipped."
* If a high number (5 or 6) is rolled, we'll w00t the player (high fives for the high rollers!)

Recall that we're currently w00ting and blamming players in the play method of the Game class. The play method encapsulates that behavior nicely. We don't want to change how the play method is called. Rather, we want to change the internal implementation so that the play method rolls a die and then decides what to do with a player. And because we're approaching this test-first, we'll start in the game\_spec.rb file.

1. Create a new spec file called game\_spec.rb in your studio\_game directory. To help you get started, go ahead and copy the following code and paste it into the spec file:
2. require\_relative 'game'  
     
   describe Game **do**  
     
    before **do**  
    @game = Game.new("Knuckleheads")  
     
    @initial\_health = 100  
    @player = Player.new("moe", @initial\_health)  
      
    @game.add\_player(@player)  
    **end**  
      
   **end**
3. This should look familiar. It simply sets up a game with one player. Remember that the before block is called before every example runs, so the instance variables set in before are accessible in the examples.

Now let's start test driving the desired behavior...

#### 

#### 

#### High Number

1. If we call play and a high number is rolled, we expect the player's health to increase by 15. So the game will need to roll a die to get a random number. However, randomness is difficult to test because it's unpredictable by nature. Instead, when we're running the tests we want to pre-define what the die rolls. That way the tests can trigger certain conditions and we can write expectations for each branch of the code.
2. To do that, we'll assume we have a Die class with a roll method that returns a random number between 1 and 6. The play method in the Game class will then create a new Die object and call its roll method. Then, in our tests, we can*stub out* the roll method of any Die object. For example, to make the die roll a 5 we use this incantation:
3. Die.any\_instance.stub(**:roll**).and\_return(5)
4. Don't worry about the stub syntax; it's a finer point of RSpec you can investigate later. For now, just think of it as a way for us to force a specific number to be rolled.
5. **RSpec 3:** If you're using RSpec 3, you can substitute this alternative syntax which does the same thing:
6. allow\_any\_instance\_of(Die).to receive(**:roll**).and\_return(5)
7. Using that stub, write a code example that expresses what should happen when a high number is rolled.

it “w00ts the player if a high number is rolled” do  
 Die.any\_instance.stub(:roll).and\_return(5)  
 allow\_any\_instance\_of(Die).to receive(:roll).and\_return(5)  
 @game.play  
 @player.health.should == @initial\_health + 15  
end

1. Run the spec from the command line like so:
2. rspec game\_spec.rb
3. It should fail because we don't have a Die class. That's easy enough to fix...
4. Create a class called Die in a separate die.rb file in your studio\_game directory. Inside the class, define an instance method called roll. There's no need to return anything from this method (yet) because our stub will automatically return a pre-defined number.

class Die  
 def roll  
 end  
end

1. Over in the Game class, require the die.rb file.

require\_relative “die”

1. Then change the play method to make the test pass. Taking the simplest approach, you don't even need to roll the die (yet). For right now you could just fake it by w00ting each player since we only have one spec and that's what it's expecting.

def play  
 puts “There are#{@players.size} players in #{@title}:”  
 @players.each do |player|  
 puts player  
 end  
   
 @players.each do |player|  
 player.w00t  
 puts player  
 end  
end

Run the spec and it should pass! There's nothing obvious to refactor, so let's move on to the next scenario…

Medium Number

1. Next, write a code example that expresses what should happen when a medium number is rolled. Remember that rolling a medium number should not change the player's health.

it “skips the player if a medium number is rolled” do  
 Die.any\_instance.stub(:roll).and\_return(3)  
 @game.play  
  
 @player.health.should == @Initial\_health  
end

1. Run the spec and it should fail because the play method is currently w00ting each player. To get this spec to pass, you could change the play method to skip every player. However, the high number spec would then fail. To get *both* specs to pass, you need to use a conditional...
2. Inside the play method, for each player create a new Die object and call its roll method. Assign the result to a number\_rolled variable.

def play  
 puts “There are #{@players.size} players in #{@title}:”  
 @players.each do |player|  
 puts player  
 end  
  
 @players.each do |player|  
 die = Die.new  
 number\_rolled = die.roll  
   
 player.w00t  
 puts player  
 end  
end

1. Then use an if conditional to skip the player if a number less than 5 is rolled and w00t them otherwise. If the player is skipped, simply print out "(insert player's name) was skipped."

def play  
 puts “There are #{@players.size} players in:”  
 @players.each do |player|  
 puts player  
end  
  
@players.each do |player|  
 die = Die.new  
 number\_rolled = die.roll  
 if number\_rolled < 5  
 puts “#{player.name} was skipped”  
 else  
 player.w00t  
 end  
 puts player  
 end  
end

Run the spec and it should pass! Again, there's nothing obvious to refactor, so let's move on to the third and final scenario…

Low Number

1. Write a code example that expresses what should happen when a low number is rolled—it should decrease the player's health by 10.

it “blams the player if a low number is rolled” do  
 Die.any\_instance.stub(:roll).and\_return(1)  
 @game.play  
 @player.health.should == @initial\_health - 10  
end

1. Run the spec and it should fail because the conditional in the play method doesn't have a branch to handle that situation.
2. Add a branch to the if conditional in the play method to make all the specs pass.

def play  
 puts “There are #{@players.size} players in: “  
 @players.each do |player|  
 puts player  
 end  
  
 @players.each do |player|  
 die = Die.new  
 number\_rolled = die\_roll  
 if number\_rolled < 3  
 player.blam  
 elsif number\_rolled < 5  
 puts ‘#{player.name} was skipped”  
 else  
 player.w00t  
 end  
 puts player  
 end  
end

1. Run the specs and they should pass.
2. Now it's time for a little refactoring. Although the if conditional works just fine, this is a good opportunity to use Ruby's case statement. Change the if conditional to a case statement and specify a range of numbers for each when clause.

def play  
 puts “There are #{@players.size} players in #{title} :”  
 @players.each do |player|  
 puts player  
 end  
  
 @players.each do |player|  
 die = Die.new  
 case die.roll  
 when 1..2  
 player.blam  
 when 3..4  
 puts “#{player.name} was skipped”  
 else   
 player.w00t  
 end  
 puts player  
 end  
end

Don't forget to run the specs after refactoring!

Run All The Specs

You now have two spec files player\_spec.rb and game\_spec.rb. You learned how to run these files individually, but you often want to run all the specs in one fell swoop. To do that, simply call the rspec command and pass in the directory containing your spec files.

For example, if you're in the studio\_game directory where all the spec files live, use a dot (.) which represents the current directory, like so:

rspec .

By default, the rspec command runs all files ending with \_spec.rb in the specified directory and its subdirectories (we don't have any of those yet).

To colorize output for all the specs, tack on the --color option:

rspec . --color

Now, every time you add or change code, you can easily run all the specs to make sure everything still works. Do that as part of your development process, and all your worries melt away.

**Windows Users:** If you see funny ANSI characters instead of colors, a quick fix is to install ANSICON.

### 4. Random Die Rolls?

Wait a minute! What about that Die class? We left the roll method empty. We got by with that only because in the tests we stubbed out the roll method. But when we run the actual game, that clearly won't work. During the game the roll method needs to return a random number.

1. Change the roll method to return a random number between 1 and 6.

class Die  
 def roll  
 rand(1..6)  
 end  
end

1. We won't write a unit test for this method because, well, all it does is produce a random number and unpredictable behavior like this is kinda hard to test (how would we know which number to expect?).
2. Instead, just as a quick and dirty test, jump out to your command prompt and open an irb session in the directory that contains your die.rb file. Then load the file using the load helper and call the roll method a few times. You basically just want to make sure it's acting like a real die, not a loaded die that always rolls the same number. Here's an example:

>> load 'die.rb'  
=> true  
>> die = Die.new  
=> #<Die:0x007fabc402df88>  
>> die.roll  
=> 2  
>> die.roll  
=> 5  
>> die.roll  
=> 3  
>> die.roll  
=> 1

1. Once you're confident it's giving you random numbers, change the roll method so that it stores (or assigns) the number it rolls in an instance variable called @number. While you're at it, create a readable number attribute so that the number that was rolled can be accessed from outside the class. That way, if you forget the outcome of the roll you can ask the die to remind you. We'll use the number attribute to audit die rolls in a future exercise.

class Die  
 attr\_reader :number  
  
 def roll  
 @number = rand(1..6)  
 end  
end

1. Do we need to write an initialize method to set up the @number instance variable? Actually, no, we don't need an initialize method because you'll recall that instance variables spring into existence when you assign to them.
2. On the other hand, we may want our die to always have a valid number between 1 and 6 even if it hasn't been rolled yet. Think of it as the die rolling itself when it's created. To do that, write an initialize method that calls roll, which in turns sets @number.

class Die   
 attr\_reader :number  
 def initialize  
 roll  
 end  
  
 def roll  
 @number = rand(1..6)  
 end  
end

As an aside, in the video we didn't create a Die class. We said we'd leave that to you. And you might be wondering if you really need a full-blown class for a die. Well, it's a design decision, and you could go either way on this one. But think about a die in the real world: it has behavior (you roll it) and state (the number that's currently showing). So it makes good sense to model it as a class. Having a class lets you encapsulate die-related detail and create die objects. Plus it gave you more practice creating another class with state and behavior. :-)

With the die properly implemented, you can now run the game and you should get random game play:

1. ruby studio\_game.rb
2. Think of running the game as an integration test of sorts. Unit tests tend to focus on individual classes and methods. They give us some degree of confidence, but we also need to run the program as a whole.
3. In fact, run it *multiple* times for the full random effect. Takes bets around the office as to which player will win. And as always, feel free to experiment until you've sufficiently rolled all this around in your head. (We never promised all our jokes would be funny.)

### Bonus Round

#### Fundraising Program

It stands to reason that not all of the projects in the fundraising program will receive the same level of funding. Some projects might appeal more to your contributors and thus receive more funds, and some projects might fall behind schedule and have funds taken away. How do you want your app to figure this out? That's entirely up to you. Let your creative juices flow!

One possible idea would be to use a random die roll. If an even number is rolled, add funds to a project. If an odd number is rolled, remove funds from a project. (Why not try implementing this feature with the test-first approach for practice?)

This might also be a good point at which to determine if a project is fully funded. You could calculate this multiple ways, such as (a) the amount of funding equals or exceeds the amount of the target funding amount or (b) the total funding outstanding is less than or equal to zero. (How about writing a test for this calculation?)

crowdfunding.rb  
  
require\_relative 'project'  
require\_relative 'fund\_request'

project1 = Project.new("Project ABC", 5000, 1000)  
project2 = Project.new("Project LMN", 3000, 500)  
project3 = Project.new("Project XYZ", 75, 25)

projects = FundRequest.new("VC-Friendly Start-up Projects")

puts projects.title  
projects.add\_project(project1)  
projects.add\_project(project2)  
projects.add\_project(project3)

projects.request\_funding

fund\_request.rb

require\_relative 'project'  
require\_relative 'die'  
class FundRequest

attr\_reader :title  
def initialize(title)  
 @title = title  
 @projects = []  
end

def add\_project(a\_project)  
 @projects.push(a\_project)  
end

def request\_funding  
 puts "There are #{@projects.size} projects that you could fund:"  
 @projects.each do |project|  
 puts project  
end

@projects.each do |project|  
 die = Die.new  
 number\_rolled = die.roll  
 if number\_rolled.odd?  
 project.remove\_funds  
 else  
 project.add\_funds  
 end  
 puts project  
 end  
end  
end

fund\_request\_specifications.rb  
require\_relative 'fundrequest'  
describe FundRequest do

before do  
 @fundrequest = FundRequest.new("VC-Friendly Start-up Projects")  
 @initial\_funds = 1000  
 @project = Project.new("Project ABC", 5000, @initial\_funds)

@fundrequest.add\_project(@project)  
end

it "adds funds to a project if an even number is rolled" do  
 Die.any\_instance.stub(:roll).and\_return(4)  
 @fundrequest.request\_funding  
 @project.funding.should == @initial\_funds + 25  
end

it "removes funds to a project if an odd number is rolled" do  
 Die.any\_instance.stub(:roll).and\_return(3)  
 @fundrequest.request\_funding  
 @project.funding.should == @initial\_funds - 15  
end  
end  
  
project.rb  
class Project  
  
attr\_accessor :name  
attr\_reader :funding, :target  
  
def initialize(name, target\_funding\_amount, funding=0)  
 @name = name  
 @target = target\_funding\_amount  
 @funding = funding  
end  
  
def to\_s  
 "#{@name} has $#{@funding} in funding towards a goal of $#{@target}."  
end

def remove\_funds  
 @funding -= 15  
 puts "#{@name} lost some funds!"  
end

def add\_funds  
 @funding += 25  
 puts "#{@name} got more funds!"  
end

def total\_funding\_outstanding  
 @target - @funding  
end

def fully\_funded?  
 total\_funding\_outstanding <= 0  
end  
end

if \_\_FILE\_\_ == $0  
 project = Project.new("Project ABC", 5000, 1000)  
 puts project.name  
 puts project.funding  
 project.remove\_funds  
 puts project.funding  
 project.add\_funds  
 puts project.funding  
end

project\_specifications.rb  
require\_relative 'project'  
describe Project do

before do  
 @initial\_funds = 1000  
 @project = Project.new("Project ABC", 5000, @initial\_funds)  
 $stdout = StringIO.new  
end   
  
it "has an initial target funding amount" do  
 @project.target.should == 5000  
end

it "computes the total funds outstanding as the target funding amount minus the funding amount" do   
 @project.total\_funding\_outstanding.should == (5000 - 1000)  
end  
  
it "increases funds by 25 when funds are added" do  
 @project.add\_funds  
 @project.funding.should == @initial\_funds + 25  
end

it "decreases funds by 15 when funds are removed" do  
 @project.remove\_funds  
 @project.funding.should == @initial\_funds - 15  
end  
  
context "created without a funding amount" do  
 before do  
 @project = Project.new("Project ABC", 5000)  
 end

it "has a default funding amount of 0" do  
 @project.funding.should == 0  
 end  
end

context "when total funding outstanding is less than or equal to 0" do  
 before do  
 @project = Project.new("Project ABC", 5000, 5000)  
 end

it "is fully-funded" do  
 @project.should be\_fully\_funded  
 end  
end  
  
context "when total funding outstanding is greater than 0" do  
 before do  
 @project = Project.new("Project ABC", 5000, 1000)  
 end

it "is under-funded" do  
 @project.should\_not be\_fully\_funded  
 end  
end  
end

die.rb  
class Die  
  
attr\_reader :number  
def initialize  
 roll  
end  
  
def roll  
 @number = rand(1..6)  
end  
end

if \_\_FILE\_\_ == $0  
 die = Die.new  
 puts die.roll  
 puts die.roll  
 puts die.roll  
end