Objective Continued:

Remember back when we split out our class definitions (Game and Player) into multiple source files? Organizing them this way made it easy to reuse individual classes. To use a class, all we have to do is require it. Also, by having each class in its own file, we were able to write unit tests that focused on individual classes. The point is that simply moving code around can make your program better.

Once you've organized things at the file level, the next step is to refactor the code within those files. In particular, look for classes that have taken on too much responsibility. You want a class to have a focused, cohesive set of behaviors. If it starts to get overwhelmed with too many responsibilities, it's time to delegate.

For example, currently the Game class has a play method that looks like this:

**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**

There's a lot going on in that method. It feels like rolling the die and updating the player should be somebody else's responsibility. So the objective is to refactor the code so that this responsibility is delegated to a new GameTurn module. The result will be a slimmed-down play method that reads as follows:

**def** **play**  
 puts "There are #{@players.size} players in #{@title}: "  
   
 @players.each **do** |player|  
 puts player  
 **end**  
   
 @players.each **do** |player|  
 GameTurn.take\_turn(player)  
 puts player  
 **end**  
**end**

Notice that we aren't changing any functionality here. Rather, we're delegating the responsibility of rolling the die and updating the player to a new GameTurn module. In other words, we're simply moving code around. This may seem like a trivial thing to do, but it makes the play method easier to read and creates a cleaner separation of concerns. And, apropos of this exercise, it gives us an opportunity to learn about modules.

Test Code

Whenever you're refactoring code, it's good to take small incremental steps. It's also good to have tests. Without tests, you run the risk of breaking existing functionality. And refactoring is all about *not* changing functionality.

In this case, we don't need any new tests because we aren't adding new functionality. When we're done with this refactoring exercise, all the tests we wrote in the previous exercises should still pass. (If not, we messed something up while refactoring.)

Let's tackle this exercise one small step at a time...

### 

### 

### 1. Create the GameTurn Module

### First we need to create the GameTurn module…

### A *module* is like a class, but a module cannot be instantiated. In other words, you don't create objects from modules. So if you can't instantiate a module, how would you possibly *use* a module?

### It turns out there are many uses for modules. In this exercise we'll use a module to define a *module method*. Think of module methods as just ordinary methods similar to the very first method we wrote at the top level of the program. Module-level methods, however, don't live at the top level; they live inside of a module.

### So on one hand modules are an organizational aid that allow us to group together methods that don't really belong in a class, and also give them a real home. Technically speaking, a module is a *namespace* where methods (and other things) can live without worrying about clashing with other similarly-named things in the program.

1. Define a module called GameTurn in a separate game\_turn.rb file in your studio\_game directory. Module names always start with an uppercase letter, and multi-word module names have each word capitalized.

module GameTurn  
end

1. Inside the module, define a module method called take\_turn that accepts a player object as a parameter.

module GameTurn  
 def self.take\_turn(player)  
end  
end

1. With the method defined, go ahead and **copy** the relevant code from the play method in the Game class and paste it into the take\_turn method in the GameTurn module. You don't want to copy *all* the code, just the parts that roll the die and either w00t, blam, or skip the player.

module GameTurn  
 def self.take\_turn(player)  
 die = Die.new  
 case die.roll  
 when 1..2  
 player.blam  
 when 3..4  
 puts “#{player.name} was skipped.”  
 else  
 player.w00t  
 end  
end  
end

1. The GameTurn module now has dependencies on two classes. At the top of the game\_turn.rb file, require the appropriate files to resolve those dependencies.

require\_relative ‘player’  
require\_relative ‘die’

1. Run all your specs. They should still pass because we haven't changed any of the code they test. Remember, we simply *copied* the code from the Game class. The next step is to remove it...

2. Refactor the Game to Use GameTurn

### With that code neatly tucked away in the GameTurn module, we're ready to refactor the Game class to use it...

1. In the Game class, remove the code that rolls the die and updates the player. Then, in its place, call the module method you defined in the previous section.

def play  
 puts “There are #{@players.size} players in: “  
 @players.each do |player|  
 puts player  
 end  
  
 @player.each do |player|  
 GameTurn.take\_turn(player)  
 puts player  
 end  
end

1. That then creates a dependency on the GameTurn module. You should be getting good at resolving file dependencies by now, so this is the last time we're *required* to remind you. :-)

require\_relative ‘game\_turn”

1. At this point, the game should work exactly as it did at the end of the last exercise. We didn't change any functionality; we simply refactored code that was already working. So, in theory, there should be no visible changes when you run the game. But sometimes refactoring ends up accidentally breaking code. That's why it's always a good idea to write tests and **run them** when you're done refactoring.
2. Re-run the game\_spec.rb file you created in the previous exercise:
3. rspec game\_spec.rb
4. You know the refactoring was successful if all the specs pass!
5. Now, you may be wondering, "Why didn't we create a spec for the take\_turn method in the GameTurn module?" Well, that method gets called by the play method in the Game class. If take\_turn is broken, the specs we already have in game\_spec.rb will fail. In other words, the tests sufficiently cover the code.
6. Finally, as a high-level sanity check (an integration test), run the game to make sure that everything still works.
7. ruby studio\_game.rb

Bonus Round

Fundraising Program

Practice a bit more with modules by refactoring your fundraising program code so that the responsibility for rolling the die and updating the project is delegated to a module, called FundingRound, for example. Got any other ideas for a module? By all means, give 'em a whirl too!

crowdfund.rb

require\_relative 'project'  
require\_relative 'fundrequest'

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

funding\_round.rb  
  
require\_relative 'project'  
require\_relative 'die'

module FundingRound

def self.one\_round(project)  
 die = Die.new  
 number\_rolled = die.roll  
 if number\_rolled.odd?  
 project.remove\_funds  
 else  
 project.add\_funds  
 end  
end  
end

fund\_request.rb

require\_relative 'project'  
require\_relative 'die'  
require\_relative 'funding\_round'

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|  
 FundingRound.one\_round(project)  
 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