### 1. Namespace Our Code

Right now all of our classes and modules are defined in the top-level Ruby namespace. This works fine if we only intend to use our code as a standalone program. However, our goal here is to distribute it as a RubyGem. And folks who install our gem may want to use some of our code in their code. In other words, they may want to use our code as a library rather than a standalone program.

For example, suppose another game programmer stumbles on our gem, has a look at our Player class, and decides she'd like to use it in her game program. To do that, she'd require our Player class. But what happens if her program also has a Player class? Well, the two class definitions would end up merging into one frankensteined Player class, which is probably not the kind of player anyone wants in a game.

To avoid these types of naming collisions, we need to wrap our classes (and modules) in a unique namespace. This is the third primary use of modules. We saw how modules are used as buckets of methods and as mixins. Modules are also used to namespace Ruby classes and other modules.

We'll create one top-level module called StudioGame and wrap all our code inside this module. That way, our StudioGame::Player class and another programmer's Player class can peacefully coexist in the same program.

1. Before you get started, run all the specs just to make sure you have a good baseline to work from:
2. rspec .
3. Get in the habit of always running your tests before making any changes, and your life as a programmer will be sweeter.
4. Now go ahead and put all your classes and modules in a StudioGame module. (Free free to use a different name if you fancy something else.) This is a bit tedious as you'll need to go into each file and define the module, but it's well worth it. To help you get started, here's how you'll namespace the Player class in the player.rb file:
5. **module** StudioGame  
    **class** Player  
    ...  
    **end**  
   **end**
6. It's pretty straightforward, perhaps until you get to the code in the treasure\_trove.rb file. That file defines a top-level constant called Treasure and a module called TreasureTrove. Both of these need to be wrapped in the StudioGame module, like so:
7. **module** StudioGame  
    Treasure = Struct.new(**:name**, **:points**)  
     
    **module** TreasureTrove  
    ...  
    **end**  
   **end**
8. Now that you have all the classes namespaced in a module, anytime you want to refer to those classes from *outside* of the module you'll need to use the namespace. For example, over in the studio\_game.rb main program file, you're currently creating a Game object like this:
9. game = Game.new("Knuckleheads")
10. But this won't work now because there is no Game class in the top-level Ruby namespace. Change it to reference the Game class in the StudioGame module.

game = StudioGame::Game.new("Knuckleheads")

1. Run the game, and it should work as expected. So far, so good. But what happens when you run the specs?
2. rspec .
3. Uh oh! They fail, with errors like this:
4. uninitialized constant Player (NameError)
5. What's going on here? Well, if you look in your spec files you'll see that they reference our classes in the top-level Ruby namespace. For example, in the case above it's looking for a Player class. But there is no Player class; it's now the StudioGame::Player class. To fix this, we could change the spec to reference the StudioGame::Player. But there's a better (and easier) way...
6. Put all your specs (the describe blocks) in the StudioGame module, as well. Again, you'll need to go into each file and make the change, but you're getting good at this by now. To start you off on the right foot, here's how you'll change the player\_spec.rb file:
7. **module** StudioGame  
    describe Player **do**  
    ...  
    **end**  
   **end**
8. In the end, you'll have all your game's classes, modules, and specs in the StudioGame namespace. If you then reference the Player class, for example, from anywhere *inside* the module, you don't have to use the namespace prefix. You only have to use the prefix if you reference the class from *outside* the module.
9. Finally, re-run all the specs to make sure they pass:
10. rspec .
11. Get in the habit of always running your tests before *and* after making any changes, and your life as a programmer will be absolute bliss!

### 2. Organize Our Files

Cool, no more class name clashes (say that 10 times fast)! Our next objective is to organize our files into directories according to the RubyGems conventions. That way, folks who use our gem will have a general idea of where to find things. The conventions are fairly simple. We just need to create three subdirectories beneath our top-level studio\_game directory:

* **bin**: contains the main command-line program file and any related files
* **lib**: contains all the code for the application (sometimes called *library files*)
* **spec**: contains all the RSpec files

So here's a big picture of how we want to organize things for our game gem:

studio\_game  
 ├── LICENSE  
 ├── README  
 ├── bin  
 │ ├── players.csv  
 │ └── studio\_game.rb  
 ├── lib  
 │ └── studio\_game  
 │ ├── auditable.rb  
 │ ├── berserk\_player.rb  
 │ ├── clumsy\_player.rb  
 │ ├── die.rb  
 │ ├── game.rb  
 │ ├── game\_turn.rb  
 │ ├── loaded\_die.rb  
 │ ├── playable.rb  
 │ ├── player.rb  
 │ └── treasure\_trove.rb  
 └── spec  
 └── studio\_game  
 ├── beserk\_player\_spec.rb  
 ├── clumsy\_player\_spec.rb  
 ├── game\_spec.rb  
 ├── player\_spec.rb  
 └── treasure\_trove\_spec.rb

Notice that both lib and spec have a studio\_game subdirectory. This mirrors our namespace. Remember that all our code is in the StudioGamenamespace. By creating a directory structure that mirrors the namespace we make it easier for folks to find our code. It's purely a convention, but by following the convention we make our gem more usable by the community.

We currently have all our files in the top-level studio\_game directory, so we'll need to move the files around. You can probably figure out what to do from the structure shown above, but here's a breakdown of the steps:

1. First, create a subdirectory called bin and move the studio\_game.rb main program file and its players.csv file into this directory.
2. Then create a subdirectory called lib, and a subdirectory within it called studio\_game. Move all the non-spec files into this directory.
3. Next, create a subdirectory called spec, and a subdirectory within it called studio\_game. Move all the spec files into this directory.
4. It's also polite (and convention) for a gem to contain two top-level plain text documentation files: README and LICENSE. Go ahead and create those files now so you don't forget—they're pretty important. The README should contain a short description of what your gem does and how to use it. The LICENSE file should contain, well, the license under which you're distributing your source code. If you're unsure or just need a reasonable, no-lawyers-required example, check out the short terms of the [MIT license](http://en.wikipedia.org/wiki/MIT_License). Choose whatever you like, just don't release your code without some kind of license, otherwise folks won't know if they can use it.

#### Adjust Requires

Moving those files around causes a small wrinkle: our require\_relative statements assume all the files are in the same directory. So we'll need to make a few adjustments:

1. Let's start with the specs. If you try to run the specs, they'll fail. That's because we moved where the class files are relative to the spec files. For example, in the player\_spec.rb file we currently use require\_relative to load the Player class, like this:
2. require\_relative 'player'
3. That tries to load the player.rb file from the same directory as the player\_spec.rb file (it's *relative* to the current directory). This worked when all the files were in the same directory, but the player.rb file is now over in the lib/studio\_game directory. To fix this, we could call require\_relative with the relative directory path where player.rb lives (../../lib/player), but there's a better way.
4. Rather than using require\_relative, we'll use require instead, like this:
5. require 'studio\_game/player'
6. Using require tells Ruby to look for files in something called the *load path*. And when we run RSpec, it automatically adds the lib and specdirectories to the load path (that's one benefit of following the conventions!) So when we call require with the pathstudio\_game/player, it finds the player.rb file beneath the lib subdirectory.
7. Go ahead and change all of your spec files to use require rather than require\_relative. Once you've made those changes, run all the specs by calling the rspec command from the top-level directory like this:
8. rspec
9. Notice that you don't have to tell RSpec where the spec files live (we previously had to run rspec .) because RSpec automatically looks for specs in the spec subdirectory. Yet another reason to follow the conventions!
10. Before moving on, make sure all the specs pass (green across the board!) because we're quickly counting down to distributing our code to the world...
11. But wait, we have a similar problem if we try to run the studio\_game.rb file, which is now in the bin directory. To run it from the top-level directory, type:
12. ruby bin/studio\_game.rb
13. You should get a LoadError, and again that's because the main program assumes all the files are in the same directory. At the top of the studio\_game.rb file we currently require files like this:
14. require\_relative 'game'  
    require\_relative 'clumsy\_player'  
    require\_relative 'berserk\_player'
15. Change those around to point to the correct files.

require\_relative '../lib/studio\_game/game'  
 require\_relative '../lib/studio\_game/clumsy\_player'  
 require\_relative '../lib/studio\_game/berserk\_player'

1. Finally, we have a potential issue with loading the default players.csv file. In the studio\_game.rb file, we're currently loading players like this:
2. game.load\_players(ARGV.shift || 'players.csv')
3. That assumes that the players.csv file is always in the current working directory. However, if a player file isn't given as a command-line argument, then we want to load the default players.csv file that's in our gem's bin directory.
4. Here's how to do that:
5. default\_player\_file = File.join(File.dirname(\_\_FILE\_\_), 'players.csv')  
   game.load\_players(ARGV.shift || default\_player\_file)
6. Here we create a default\_player\_file variable that contains the absolute directory path to the players.csv file. To get the absolute path, we first get the path to the directory that contains the studio\_game.rb file (remember that the name of the current file is held in the \_\_FILE\_\_ variable). That resolves to the bin directory. Then we tack on the players.csv file name, since it's also in the bindirectory.
7. By doing it this way, we'll always get the absolute path to the default players.csv file no matter where our code gets installed. And that means our gem will automatically adapt to its environment.

### 3. Make a Command-Line Script

As a nice little touch, let's suppose we want users of our gem to be able to run a game from the command line like this:

studio\_game

That makes it feel like a built-in script, similar to how we run rspec. There's no need to explicitly invoke the Ruby interpreter or navigate down into the bin directory. That's kinda convenient.

RubyGems will take care of some of this for us, but we need to do a little prep work first:

1. First, rename the bin/studio\_game.rb file to bin/studio\_game without the .rb extension.
2. Now you can change into the top-level studio\_game directory and run the game this way:
3. ruby bin/studio\_game
4. Next, to run the game without directly invoking the ruby command, add the following "shebang" line as the *first line* of thebin/studio\_game file:
5. #!/usr/bin/env ruby
6. This is a Unix shell convention that tells the shell to execute the rest of the file using the ruby interpreter.
7. On Unix systems (Mac and Linux), you also need to make the file executable with this incantation:
8. chmod +x bin/studio\_game
9. Now, if you're on a Unix system, you can run it as a command:
10. bin/studio\_game
11. And when we package it into a gem, we'll arrange things so that when people install our gem they can just run studio\_game.

### 4. Build a RubyGem

At last, we now have everything in place to package up our game as a RubyGem!

1. First, we need to tell RubyGems a bit about our application. We do that by writing a Gem specification file, or *gemspec*. The gemspec defines what's in the gem, who made it, the version of the gem, and so on. It's a manifest, of sorts.
2. Rather than typing this in from scratch, go ahead and copy the following gemspec and paste it into a new file named studio\_game.gemspecin the top-level application directory (the studio\_game directory):
3. Gem::Specification.new **do** |s|  
    s.name = "INSERT GEM NAME HERE"  
    s.version = "INSERT VERSION HERE"  
    s.author = "INSERT YOUR NAME HERE"  
    s.email = "INSERT YOUR EMAIL HERE"  
    s.homepage = "INSERT HOMEPAGE URL HERE"  
    s.summary = "INSERT SUMMARY HERE"  
    s.description = File.read(File.join(File.dirname(\_\_FILE\_\_), 'README'))  
    s.licenses = ['MIT']  
     
    s.files = Dir["{bin,lib,spec}/\*\*/\*"] + %w(LICENSE README)  
    s.test\_files = Dir["spec/\*\*/\*"]  
    s.executables = [ 'studio\_game' ]  
     
    s.required\_ruby\_version = '>=1.9'  
    s.add\_development\_dependency 'rspec'  
   **end**
4. Then you'll need to replace the placeholder text in the highlighted lines with your own information. It's fairly self-explanatory, but here's a quick overview:
   * **name**: This field contains the gem name, which should be all lowercase letters with underscores or hyphens separating words. The name isn't just cosmetic—the name you choose here will show up as the name of the gem when folks install it. Also, if you plan to publish your gem to a public server (which we'll do in the next section), the gem must have a unique name. So unless you're the first person to complete this course, odds are there's already a public studio\_game gem. But if you hurry, you could be the proud owner of that gem!
   * **version**: This field contains the gem version, which generally follows the *major.minor.release* numbering notation. For example, "1.0.0" would indicate the first major release of this gem. RubyGems uses this version string to name the gem package and also for dependency management, so it's important to use a consistent numbering strategy.
   * **author**: Your name, because all good artists sign their work. :-)
   * **email**: Your email, because all good developers support the code they write.
   * **homepage**: A URL to the gem's homepage, because all good gems deserve a good home.
   * **summary**: A really short description of the gem, say 5 or 6 words worth.
   * **description**: A longer description of the gem. Here we're exploiting the fact that a gemspec is really just Ruby code. When we build the gem, any dynamic code in the gemspec will be run to create static text. In this case, we use the File class to read the contents of our README file in order to set the value of the description field. Nifty!
   * **licenses**: An array of license abbreviations for the gem. Each license must be a [short name](http://opensource.org/licenses/alphabetical), no more than 64 characters. The full text of the license should be inside of the LICENSE file. [MIT](http://en.wikipedia.org/wiki/MIT_License) is a popular choice for open-source software.
   * **files**: An array of files to include in the gem package. Here's another example of using dynamic code. The Dir[] syntax returns the filenames found by expanding a given pattern (sometimes called a *glob*). In this case, we collect all the files beneath the bin, lib, and spec directories and also add in the README and LICENSE file.
   * **test\_files**: An array of just the test (spec) files. Test files will be loaded as unit tests when the user requests a gem to be unit tested.
   * **executables**: An array of command-line scripts that RubyGems should install when folks install the gem. In this case, it's our studio\_game script which by convention is assumed to be in the bin directory.
   * **required\_ruby\_version**: This is really important. Since we used require\_relative which is only supported in Ruby 1.9 or newer, this ensures that the gem can only be run with that version of Ruby.
   * **add\_development\_dependency**: Since we used RSpec, it's good to list it as a development dependency. Development dependencies are not installed by default, and are not activated when the gem is activated.
5. Whew! That was a whirlwind tour. Most of the time that's all you need, but for more details you can check out the [full GemSpec reference](http://docs.rubygems.org/read/chapter/20).
6. Once you've filled in your information, you're ready to build a gem. To do that, type:
7. gem build studio\_game.gemspec
8. The build command reads and executes the gemspec, then generates a packaged *.gem* file. For example, if you named the gem studio\_game and set the version to 1.0.0, then you'll end up with a file called studio\_game-1.0.0.gem
9. As a sanity check, go ahead and install it locally on your machine to test it out:
10. gem install studio\_game-1.0.0.gem
11. Then, to check if it is indeed installed, type:
12. gem list studio\_game -d
13. The -d option includes details about the gem (author, homepage, summary, etc.) in the listing.
14. Now you should be able to open a new command prompt and run the command-line script like so:
15. studio\_game
16. When you run a game without specifying a player file, you should see all the players from the default players.csv file that's packaged in the gem. That works because of the absolute path change we made earlier. You knew there had to be some upside to doing that...
17. Finally, go ahead and uninstall it from your local machine:
18. gem uninstall studio\_game

### 5. Publish It!

At this point, we could share our package .gem file with our friends simply by e-mailing them the file or putting it on a shared server. Then they would need to install it just as we did previously. But if you don't mind sharing your gem with the rest of the world, there's a better way.

[RubyGems.org](http://rubygems.org/) is the central repository for public gems. This is where Ruby programmers look for gems to use, and it's also where the gem install command looks for remote gems. It's like a clearinghouse of gems. So let's publish our game gem for the world to enjoy:

1. First, you'll need to create an account on [RubyGems.org](http://rubygems.org/). You only need to do this once, and then you can publish as many gems as you can dream up.
2. This is also a good time to double-check that there's not already a public gem with the same name as your gem. Since RubyGems.org is the main repository for almost all open-source Ruby code, it's possible that the name of your gem is already taken. (We warned you that naming is one of the more challenging parts of programming!)
3. Once you're sure that your gem name is available, push your .gem file to the public RubyGems.org server by typing:
4. gem push studio\_game-1.0.0.gem
5. If this is the first time you push a gem, you'll be prompted for your username and password. These credentials are stored for future pushes.
6. In just a few seconds, your gem will be available for installation by anyone in the world. To search for it on the public server, use:
7. gem search -r studio\_game
8. And you should see a response like this:
9. \*\*\* REMOTE GEMS \*\*\*  
     
   studio\_game (1.0.0)
10. To install it from the remote server, use:
11. gem install studio\_game
12. To play a game from the command-line, open a new command prompt and run the command-line script like so:
13. studio\_game
14. Or, if you'd like to use the game as a library, here's an example of how to use it in irb:
15. >> require 'studio\_game/game'  
    => true  
      
    >> game = StudioGame::Game.new("Knuckleheads")  
    => #<StudioGame::Game:0x007fdea10252d8 @title="Knuckleheads", @players=[]>  
      
    >> player = StudioGame::Player.new("Moe", 90)  
    => I'm Moe with health = 90, points = 0, and score = 90.  
      
    >> game.add\_player(player)  
    => [I'm Moe with health = 90, points = 0, and score = 90.]  
      
    >> game.play(1)

And that's all there is to it. You now have a published Ruby gem! So, pick up your blowhorn, fire up an e-mail, prepare a tweet, or load up FaceBook and announce your accomplishment! And if it isn't too much to ask, e-mail us. We'd love to be one of the first people to download your sparkling new Ruby gem!

### Solution

The full solution for this exercise is in the distribution directory of the [code bundle](https://s3.amazonaws.com/pragmaticstudio/courses/ruby/pragstudio-ruby-code.zip).

### Bonus Round

#### Fundraising Program

If you've been following along and building your own fundraising program, now is the time to publish it as a Ruby gem! And we'd be delighted to hear about it. What ideas did you come up with for virtual attributes, how to randomly assign funds and pledges, outputs, mixins, or inheritance? We'd love to incorporate your super cool ideas back into the bonus round exercises. If it's not too much trouble, e-mail us a link to your published Ruby gem. Fundraisers unite!  
  
funding\_round.rb  
require\_relative 'project'  
require\_relative 'die'  
require\_relative 'pledge\_pool'

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

pledge = PledgePool.random  
project.received\_pledge(pledge)  
end  
end

fund\_request.rb  
require\_relative 'project'  
require\_relative 'funding\_round'  
require\_relative 'pledge\_pool'  
  
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(rounds)  
 puts "There are #{@projects.size} projects that you could fund:"  
 @projects.each do |project|  
 puts project  
end

pledges = PledgePool::PLEDGES  
puts "\nThere are #{pledges.size} possible pledge amounts:"  
pledges.each do |pledge|  
puts "A #{pledge.name} pledge is worth $#{pledge.amount}."  
end

1.upto(rounds) do |round|  
 puts "\nFunding Round #{round}:"  
 @projects.each do |project|  
 FundingRound.one\_round(project)  
  
 puts project  
 end  
end   
end

def print\_name(project)  
 puts "#{project.name}"  
end  
def fully\_funded\_projects  
 @projects.select { |project| project.fully\_funded? }  
end

def under\_funded\_projects  
 @projects.reject { |project| project.fully\_funded? }  
end

def print\_results  
 puts "\n#{fully\_funded\_projects.size} Fully Funded Projects:"  
 fully\_funded\_projects.each do |project|  
 print\_name(project)  
end

puts "\n#{under\_funded\_projects.size} Under Funded Projects:"  
under\_funded\_projects.each do |project|  
  
print\_name(project)  
end

puts "\n#{under\_funded\_projects.size} projects still need your help:"  
 sorted\_under\_funded\_projects.each do |project|  
 formatted\_name = project.name.ljust(20, '.')  
 puts "#{formatted\_name} $#{project.total\_funding\_outstanding} under"  
end  
@projects.each do |project|  
 puts "\n#{project.name}'s pledges:"  
 project.each\_received\_pledge do |pledge|  
 puts "$#{pledge.amount} in #{pledge.name} pledges"  
end  
 puts "$#{project.pledges} in total pledges"  
end  
end

def sorted\_under\_funded\_projects  
 under\_funded\_projects.sort { |a, b| b.total\_funding\_outstanding <=> a.total\_funding\_outstanding}  
end

def load\_projects(from\_file)  
 File.readlines(from\_file).each do |line|  
 name, target\_funding\_amount, funding = line.split(',')  
 project = Project.new(name, Integer(target\_funding\_amount), Integer(funding))  
 add\_project(project)  
 end  
end

def save\_under\_funded\_projects(to\_file="needmoremoney.txt")  
 File.open(to\_file, "w") do |file|  
 file.puts "These projects still need your help:"  
 sorted\_under\_funded\_projects.each do |project|  
 formatted\_name = project.name.ljust(20, '.')  
 file.puts "#{formatted\_name} $#{project.total\_funding\_outstanding} under"  
 end  
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(2)  
 @project.funding.should == @initial\_funds + (25 \* 2)  
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(2)  
 @project.funding.should == @initial\_funds - (15 \* 2)  
end

it "assigns a pledge for amount during a project's funding round" do   
 fundrequest = FundRequest.new("VC-Friendly Start-up Projects")  
 project = Project.new("Project ABC", 5000)  
 fundrequest.add\_project(project)  
 fundrequest.request\_funding(1)  
 project.pledges.should\_not be\_zero  
end  
end

grant\_project.rb  
require\_relative 'project'  
class GrantProject < Project  
def remove\_funds  
 @funding -= 0  
 puts "#{@name} has not lost or gained any new funds."  
 end  
end

if \_\_FILE\_\_ == $0  
 grant = GrantProject.new("Project 123", 500, 100)  
 puts "#{grant.name} has $#{grant.total\_funds} in funding towards a goal of $#{grant.target}."  
 grant.remove\_funds  
 puts "#{grant.name} has $#{grant.total\_funds} in funding towards a goal of $#{grant.target}."

end

grant\_project\_specifications.rb  
require\_relative 'grant\_project'  
describe GrantProject do  
 before do  
 @initial\_funds = 1000  
 @project = GrantProject.new("Grant 123", 5000, @initial\_funds)  
end  
  
it "does not ever have funds removed" do  
 @project.remove\_funds  
 @project.funding.should == @initial\_funds  
end  
end

matching\_projects.rb   
require\_relative 'project'

class MatchingProject < Project  
def initialize(name, target\_funding\_amount, funding=0)  
 super(name, target\_funding\_amount, funding)  
 @halfway\_funded = target\_funding\_amount / 2  
end

def halfway\_funded?  
 @halfway\_funded <= funding  
end

def add\_funds  
 if halfway\_funded?  
 @funding += (25\*2)  
 puts "#{@name} has received at least half its funding!" if halfway\_funded?  
 else  
 super  
 end  
end  
end  
  
if \_\_FILE\_\_ == $0  
 matchingproject = MatchingProject.new("Matching 123", 100, 0)  
 3.times { matchingproject.add\_funds }  
 puts matchingproject.funding  
end

matching\_projects\_specifications.rb  
require\_relative 'matching\_project'  
describe MatchingProject do  
 before do  
 @initial\_funds = 0  
 @project = MatchingProject.new("Match 123", 200, @initial\_funds)  
 end  
it "does not match additional funds when the project is not halfway funded" do  
 3.times { @project.add\_funds }  
 @project.halfway\_funded?.should be\_false  
end  
it "is halfway funded when it has received half of its target funding amount" do  
 4.times { @project.add\_funds }  
 @project.halfway\_funded?.should be\_true  
end

it "receives twice as much added funds when it is halfway funded" do  
 7.times { @project.add\_funds }  
 @project.funding.should == @initial\_funds + (4 \* 25) + (3 \* 25 \* 2)  
end   
end

crowdfund.rb  
require\_relative 'project'  
require\_relative 'fundrequest'  
require\_relative 'grant\_project'  
require\_relative 'matching\_project'  
  
project1 = Project.new("Project ABC", 5000, 1000)  
project2 = Project.new("Project LMN", 3000, 500)  
project3 = Project.new("Project XYZ", 75, 25)  
grant1 = GrantProject.new("Grant 123", 500, 100)  
match1 = MatchingProject.new("Match Me", 200, 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.add\_project(grant1)  
projects.add\_project(match1)  
projects.request\_funding(4)  
projects.print\_results

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

pledge\_pool\_specifications.rb  
require\_relative 'pledge\_pool'  
describe Pledge do  
 before do  
 @pledge = Pledge.new(:bronze, 50)  
 end  
it "has a name attribute" do  
 @pledge.name.should == :bronze  
end

it "has an amount attribute" do  
 @pledge.amount.should == 50  
end  
end

describe PledgePool do  
 it "has three pledges" do  
 PledgePool::PLEDGES.size.should == 3  
end

it "has a bronze pledge worth $50" do  
 PledgePool::PLEDGES[0].should == Pledge.new(:bronze, 50)  
end

it "has a silver pledge worth 75 points" do  
 PledgePool::PLEDGES[1].should == Pledge.new(:silver, 75)  
end

it "has a gold pledge worth 100 points" do  
 PledgePool::PLEDGES[2].should == Pledge.new(:gold, 100)  
end

it "returns a random pledge" do  
 pledge = PledgePool.random  
 PledgePool::PLEDGES.should include(pledge)  
end  
end

project.rb  
require\_relative 'pledge\_pool'  
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  
 @received\_pledge = Hash.new(0)  
end

def to\_s  
 "#{@name} has $#{total\_funds} 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 - total\_funds  
end

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

def received\_pledge(pledge)  
 @received\_pledge[pledge.name] += pledge.amount  
 puts "#{@name} received a #{pledge.name} pledge worth $#{pledge.amount}."  
 puts "#{@name}'s pledges: #{@received\_pledge}"  
end   
def pledges  
 @received\_pledge.values.reduce(0, :+)  
end  
  
def total\_funds  
 @funding + pledges  
end

def each\_received\_pledge  
 @received\_pledge.each do |name, amount|  
 yield Pledge.new(name, amount)  
 end  
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  
  
it "computes pledges as the sum of all pledges" do  
 @project.pledges.should == 0  
 @project.received\_pledge(Pledge.new(:silver, 75))  
 @project.pledges.should == 75  
 @project.received\_pledge(Pledge.new(:gold, 100))  
 @project.pledges.should == 175  
 @project.received\_pledge(Pledge.new(:gold, 100))  
 @project.pledges.should == 275  
end  
it "computes total funds as the sum of a projects funding and pledges" do  
 @project.received\_pledge(Pledge.new(:gold, 100))  
 @project.received\_pledge(Pledge.new(:gold, 100))  
 @project.total\_funds.should == 1200  
end  
it "yields each received pledge and its total pledge amount" do  
 @project.received\_pledge(Pledge.new(:bronze, 50))  
 @project.received\_pledge(Pledge.new(:silver, 75))  
 @project.received\_pledge(Pledge.new(:silver, 75))  
 @project.received\_pledge(Pledge.new(:gold, 100))  
 @project.received\_pledge(Pledge.new(:gold, 100))  
 @project.received\_pledge(Pledge.new(:gold, 100))  
  
 yielded = []  
 @project.each\_received\_pledge do |pledge|  
 yielded << pledge  
end  
  
yielded.should == [  
 Pledge.new(:bronze, 50),  
 Pledge.new(:silver, 150),  
 Pledge.new(:gold, 300)  
]  
end  
end

pledge\_pool.rb  
Pledge = Struct.new(:name, :amount)  
module PledgePool  
PLEDGES = [  
 Pledge.new(:bronze, 50),  
 Pledge.new(:silver, 75),  
 Pledge.new(:gold, 100)  
]  
def self.random  
 PLEDGES.sample  
end  
end

fundable.rb  
module Fundable  
  
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 - total\_funds  
end  
  
def fully\_funded?  
 total\_funding\_outstanding <= 0  
end  
end

### Wrap Up

You now have your first published Ruby gem! **Congratulations!**

We had to jump through a few hoops to get here, but if you look back it's fairly straightforward (if not a tad tedious) stuff. You learned how to:

* use modules as namespaces to avoid naming collisions
* organize a project into directories that follow the RubyGems conventions
* write a gemspec file
* generate a packaged .gem file
* publish a gem on RubyGems.org

When you start writing your next Ruby program, you might consider starting with a project structure that already follows the conventions. That way, when it comes time to publish it as a gem you won't have to jockey around the files so much. All you'll need is a gemspec and a couple quick commands, and you'll have another published gem.

And with that, you have your first Ruby program under you belt. That's a major accomplishment, but don't stop here! You need to continue writing Ruby code, otherwise everything you learned will start to leak out of your brain. And trust us, it only takes a week or so before that memory loss starts to happen. So in the next section we'll offer some recommendations of where to go from here.