### 1. Make a Playable Mixin

How do you know what should be a mixin? One way to answer that question is to try to identify "-able" behavior in a class. For example, a particular class of objects might be playable, printable, indexable, auditable, taxable, and so on. Once you've identified those capabilities, extract their related behavior into a well-named module that can be mixed in to classes that need to exhibit that behavior.

For example, our Player class currently defines some *playable* behavior: w00t increases the player's health, blam decreases the player's health, and strong? queries the player's health. That's a relatively cohesive set of methods focused on playable behavior.

Our first task then is to extract the playable methods into a Playable module and mix the module back in to the Player class. The functionality of the game won't change at all. Instead, we'll be moving code currently in the class into a module. Our intent in doing this refactoring is to separate concerns so our design is more modular.

We have specs to check our work already (huzzah!), so let's dive right into the solution...

1. In a new file named playable.rb, define a Playable module.

module Playable  
end

1. Next, move the w00t, blam, and strong? methods from the Player class to the Playable module. Note that these are instance methods in the Player class. We want them to also be instance methods in the Playable module (not module methods). So you should be able to cut/paste them directly from Player to Playable.

module Playable  
def w00t  
 @health += 15  
 puts "#{@name} got w00ted!"  
end

def blam  
 @health -= 10  
 puts "#{@name} got blammed!"  
end  
  
def strong?  
 @health > 100  
end  
end

1. Then include the Playable module in the Player class. This makes all the instance methods in the Playable module available as instance methods in the Player class.

require\_relative 'playable'  
class Player  
  
include Playable  
end

1. Run all your specs and they should pass. Also run the game to make sure the players are being w00ted and blammed just as before. Again, because we simply moved code from one place to another, there should be no visible changes to the way the game runs.

Theoretically, at this point we could include the Playable module into any class that needs play-able behavior. We won't actually do that here since Player is the only class that exhibits this type of behavior in our game.

However, that notion segues nicely into our next task, where we will indeed share a mixin between two classes...

### 

### 2. Audit Dice Rollsrequire\_relative 'playable' **class** Player **include** Playable *# existing code* **end**

### 

As luck would have it, the gaming commission just announced that they're requiring us to audit all the dice rolls in our game. (It seems some players have complained that they aren't winning enough.) To be more transparent, whenever a player takes a turn and rolls the die, we need to print out an audit message like this:

Rolled a 5 (Die)

Of course, we could slap the code to do that into the Die class, but auditing is a separate (orthogonal) concern. Added to which, we're thinking we need to slip a loaded die into the game once in a while just to see if anyone is paying attention. Auditing then becomes a chunk of common code (a method) that we can apply to all die classes.

We'll start by making an Auditable module and including it in the existing Die class. Then we'll create a new LoadedDie class and include the auditing behavior there, as well. Finally, we'll run the game with both dice to print an audit trail.

Let's get rolling...

1. In a new file named auditable.rb, create an Auditable module that defines an audit method. This method should rely on the *host class* (the class it's mixed into) having a readable number attribute. When the audit method is called, have it print out the number and the name of the class the module was mixed into, like this:
2. Rolled a 5 (Die)

module Auditable  
 def audit  
 puts "Rolled a #{self.number} (#{self.class})"  
 end  
end

1. As if by design, the Die class already defines a number attribute. (Hey, they don't call us pragmatic for nothin'!) So go ahead and mix the Auditable module into the Die class. When the die is rolled (after the @number is set), call the auditmethod. Then return the @number value as the last expression in the method since the game expects roll to return a number.

require\_relative 'auditable'  
class Die  
 include Auditable  
  
 attr\_reader :number  
  
def roll  
 @number = rand(1..6)  
 audit  
 @number  
end  
end

1. Run all your specs just to double-check that extracting that module didn't break anything. Then run the game and every time a player takes a turn you should see an audit trail of what was rolled.
2. Next let's introduce a loaded die into the game to see if the gaming commission is really paying attention. A loaded die lands with a specific side facing upwards more often than it normally would. We'll have our loaded die always return one of the following random numbers: 1, 1, 2, 5, 6, 6.
3. In a new file named loaded\_die.rb, create a LoadedDie class like this:
4. **class** LoadedDie  
    **attr\_reader** **:number**  
      
    **def** **roll**  
    numbers = [1, 1, 2, 5, 6, 6]  
    @number = numbers.sample  
    **end**   
   **end**
5. That was a gimme; now it's your turn. Change the LoadedDie class so that every roll is audited.

require\_relative 'auditable'

class LoadedDie  
 include Auditable  
  
 attr\_reader :number  
def roll  
 numbers = [1, 1, 2, 5, 6, 6]  
 @number = numbers.sample  
 audit  
 @number  
end   
end

1. With that "rigged" up (sorry), we need to actually change the game to use the loaded die for every turn taken by a player. So back in the take\_turn method of the GameTurn module, swap out the Die and replace it with a LoadedDie. (Sneaky!)

require\_relative 'loaded\_die'  
module GameTurn  
 def self.take\_turn(player)  
 die = LoadedDie.new  
 end  
end

1. Then run the game again and every time a player takes a turn you should see an audit trail like this:
2. Rolled a 6 (LoadedDie)
3. Run a few rounds and you should see something fishy going on.
4. Finally, make sure to switch the game back to using a Die before someone notices! (The specs will fail, too.)

### Solution

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

### 

### Bonus Round

#### A Better Design

Currently, the Playable module will only work if the class it's mixed into has instance variables named @name and @health. We could probably get away with that. However, it's generally considered a better design if mixins rely on attributes (getter and setter methods) rather than instance variables. That way, if an instance variable changes it doesn't break the mixin.

This change isn't critical, especially for a program of this size, but it makes for a instructive bonus round if you're up for it...

1. Taking stock of the Playable module, it needs to read the name attribute, and read and write the health attribute. The Player class already has a readable name attribute, so the Playable module can use it.
2. Change the Playable module to rely on the name attribute, rather than the @name instance variable.

module Playable  
 def w00t  
 @health += 15  
 puts "#{name} got w00ted!"  
end  
  
def blam  
 @health -= 10  
 puts "#{name} got blammed!"  
end

def strong?  
 @health > 100   
end  
end

1. The Player class also has a health attribute, but it's read-only at this point. So, if we want to avoid tying the module to the @health instance variable, we need to change the health attribute so that it's also writable. This means the health can be changed from outside the class, thereby possibly breaking encapsulation, but it's a tradeoff worth making in this case.
2. In the Player class, change the health attribute to be both readable and writable.

class Player  
 attr\_accessor :health  
end

1. Follow that up by changing the Playable module to rely on the health attribute, rather than the @health instance variable.

module Playable  
 def w00t  
 self.health += 15  
 puts "#{name} got w00ted!"  
end   
  
def blam  
 self.health -= 10  
 puts "#{name} got blammed!"  
end  
  
def strong?  
 health > 100  
end  
end

1. Run all the specs to make sure nothing broke down along the way!

#### Fundraising Program

Let's say that the fundraising program you've been building is so successful that you're considering adding another type of "project" to which people can make contributions: a college saving account. This type of "project" will be similar to the projects you've already created in that people can add funds, remove funds (if your grades start slipping), and find out if the savings account is fully funded. In other words, while there will be differences between a traditional project and a college saving account, both are *fundable*.

Therefore, this is a perfect place to introduce a mixin in your app. Extract the fundable behavior of a project so that later, as your apps continues to evolve, it can be mixed in to all kinds of different project types.

Got any other ideas for a mixin? Give them a go too!

### 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

Now you're equipped with two design techniques for eliminating duplication: inheritance and mixins. They both have their place; the key is knowing when to use one over the other. As a general principle, reserve the use of inheritance for true *is-a*(subclassing) relationships.

Think hard before creating an inheritance hierarchy just to share code. Instead, look for "-able" capabilities and extract them into modules. Then mix them together with other modules to compose classes with a variety of capabilities, each neatly expressed in a single module. Favor mixins and your Ruby programming life will be a lot more enjoyable. Future programmers will thank you, too!

And with that, we've implemented all the features of the game! In the next section, we'll package it up and distribute it to the world (or just your friends, whichever you prefer)...