Objective Continued:

The overall objective of this exercise is to define the new clumsy and berserk players, then add them in to the game along with the regular cast of players, like so:

knuckleheads = Game.new("Knuckleheads")  
knuckleheads.load\_players  
  
klutz = ClumsyPlayer.new("klutz", 105)  
knuckleheads.add\_player(klutz)  
  
berserker = BerserkPlayer.new("berserker", 50)  
knuckleheads.add\_player(berserker)

That's going to add a whole new dynamic to the game!

1. Define a Clumsy Player

First, let's create the clumsy player. This player has a lot of the same behavior as the other players, but when he finds a treasure it's only worth half of its value.

For example, suppose a clumsy player has found three hammers each worth 50 points and one crowbar worth 400 points. His total hammer points should be 75 and his total crowbar points should be 200, for a grand total of 275 points. (A regular player would have accumulated 550 total points.)

Expressed in code, the goal is to be able to run this:

clumsy = ClumsyPlayer.new("klutz")   
hammer = Treasure.new(**:hammer**, 50)  
clumsy.found\_treasure(hammer)  
clumsy.found\_treasure(hammer)  
clumsy.found\_treasure(hammer)  
crowbar = Treasure.new(**:crowbar**, 400)  
clumsy.found\_treasure(crowbar)  
clumsy.each\_found\_treasure **do** |treasure|  
 puts "#{treasure.points} total #{treasure.name} points"  
**end**  
puts "#{clumsy.points} grand total points"

And then get this output:

Klutz found a hammer worth 25 points.  
Klutz found a hammer worth 25 points.  
Klutz found a hammer worth 25 points.  
Klutz found a crowbar worth 200 points.  
75 total hammer points  
200 total crowbar points  
275 grand total points

Check out the spec below. Then copy the spec into a file called clumsy\_player\_spec.rb that we'll use to check our work.

require\_relative 'clumsy\_player'  
  
describe ClumsyPlayer **do**  
 before **do**  
 @player = ClumsyPlayer.new("klutz")  
 **end**  
   
 it "only gets half the point value for each treasure" **do**  
 @player.points.should == 0  
  
 hammer = Treasure.new(**:hammer**, 50)  
 @player.found\_treasure(hammer)  
 @player.found\_treasure(hammer)  
 @player.found\_treasure(hammer)  
  
 @player.points.should == 75  
  
 crowbar = Treasure.new(**:crowbar**, 400)  
 @player.found\_treasure(crowbar)  
   
 @player.points.should == 275   
   
 yielded = []  
 @player.each\_found\_treasure **do** |treasure|  
 yielded << treasure  
 **end**  
  
 yielded.should == [Treasure.new(**:hammer**, 75), Treasure.new(**:crowbar**, 200)]   
 **end**  
**End**

Let's get right to it...

1. In a new file named clumsy\_player.rb, define a ClumsyPlayer class that inherits from Player. In other words, make ClumsyPlayer a *subclass* of the parent class Player.

require\_relative 'player'  
class ClumsyPlayer < Player  
end

1. Now, here's a case where having some code at the bottom of the clumsy\_player.rb file will really help us. We'll be able to quickly and easily find out how the ClumsyPlayer is behaving. So, add the following example code to the bottom of the clumsy\_player.rb file, after the class definition:

**if** \_\_FILE\_\_ == $0  
 clumsy = ClumsyPlayer.new("klutz")   
   
 hammer = Treasure.new(**:hammer**, 50)  
 clumsy.found\_treasure(hammer)  
 clumsy.found\_treasure(hammer)  
 clumsy.found\_treasure(hammer)  
   
 crowbar = Treasure.new(**:crowbar**, 400)  
 clumsy.found\_treasure(crowbar)  
   
 clumsy.each\_found\_treasure **do** |treasure|  
 puts "#{treasure.points} total #{treasure.name} points"  
 **end**  
 puts "#{clumsy.points} grand total points"  
**end**

1. Run the clumsy\_player.rb program file. What result do you expect?

Klutz found a hammer worth 50 points.  
Klutz found a hammer worth 50 points.  
Klutz found a hammer worth 50 points.  
Klutz found a crowbar worth 400 points.  
150 total hammer points  
400 total crowbar points  
550 grand total points

1. That's not the answer we're looking for, but it tells us that ClumsyPlayer is indeed inheriting all the default behavior (methods) from the Player class.
2. Here's another way to see how a ClumsyPlayer behaves at this point. Run the clumsy\_player\_spec.rb spec file. It should fail with a message like this:

F  
  
Failures:  
  
 1) ClumsyPlayer only gets half the point value for each treasure  
 Failure/Error: clumsy\_player.points.should == 75  
 expected: 75  
 got: 150 (using ==)  
  
Finished in 0.08094 seconds  
1 example, 1 failure

1. We're expecting the points to be cut in half, but clearly that's not happening. Instead, the clumsy player is racking up full point values. Again, that's because as it stands the ClumsyPlayer class inherits the found\_treasure method from its superclass.
2. So, it's pretty clear what we need to do next. We need to specialize the behavior of a ClumsyPlayer so that when he finds a treasure it's only worth half of its value. We also need to print out the name of the found treasure(s) and the associated discounted value(s).
3. To do that, you'll need to *override* the found\_treasure method in the ClumsyPlayer class:
4. **def** **found\_treasure**(treasure)  
    @found\_treasures[treasure.name] += treasure.points  
    puts "#{@name} found a #{treasure.name} worth #{treasure.points} points."  
   **end**
5. The code you'll need to write is very similar to a normal player, and you may end up with some duplication. Don't worry about it. Just get it working for now, and we'll clean up the duplication a bit later.

class ClumsyPlayer < Player  
 def found\_treasure(treasure)  
 points = treasure.points / 2.0  
 @found\_treasures[treasure.name] += points  
 puts "#{@name} found a #{treasure.name} worth #{points} points."  
end  
end

1. Re-run the clumsy\_player.rb program file and the clumsy\_player\_spec.rb spec to check your work.
2. If everything appears to be working, you're ready to add a clumsy player to the game. In the studio\_game.rb file, create a ClumsyPlayer object with the name "klutz" and an initial health of 105. Then add the player to the game.

require\_relative 'clumsy\_player'  
klutz = ClumsyPlayer.new("klutz", 105)  
knuckleheads.add\_player(klutz)

1. Run the game to make sure the new clumsy player is in the mix. Unfortunately, the clumsy player usually comes out on bottom, but at least he keeps trying.

2. A "Super" Refactoring

The solution we came up with in the previous section introduced some unnecessary duplication. "Wait", you say, "I thought we were using inheritance to *avoid* duplication!" That's certainly true. Inheritance is one way to share code in the large. For example, by having ClumsyPlayer inherit from Player we avoided having to duplicate all the code in Player. However, when we overrode the found\_treasure method we ended up introducing a small amount of duplication back in to the subclass.

Let's have a look at both methods to see how they differ. Here's the method in the Player parent class:

**def** **found\_treasure**(treasure)  
 @found\_treasures[treasure.name] += treasure.points  
 puts "#{@name} found a #{treasure.name} worth #{treasure.points} points."  
**end**

And here's the overridden method we wrote in the ClumsyPlayer child class:

**def** **found\_treasure**(treasure)  
 points = treasure.points / 2.0  
 @found\_treasures[treasure.name] += points  
 puts "#{@name} found a #{treasure.name} worth #{points} points."  
**end**

Notice that the only difference between these two methods is how they deal with the treasure's point value. A regular player gets full points for each treasure and a clumsy player gets half the points. That's exactly the behavior we want, but without the duplication.

How can we arrange things to avoid this duplication? Well, the found\_treasure method in the parent Player class (we'll call it the "default" method) does most of what we need. It stores the name-points pair in the hash and prints the message. Ideally, we'd like to be able to just call that default method from the found\_treasure method in the ClumsyPlayer subclass. That's easy enough to do. In the video, you learned that calling super from inside a method calls a method of the same name in the parent class, passing in any parameters.

The bigger question is what do we pass to the default found\_treasure method when we call super? The default method requires a Treasure object: a Struct consisting of a name and points. But we can't just pass along the original Treasureobject that we have in the subclass' found\_treasure method. The points attribute of that Treasure object has the full point value.

Instead, we'll need to create a new Treasure object with the same name as the original treasure, but with points set to half of the original treasure's points. Then we can pass that Treasure object to the default found\_treasure method. Here's how to create the "damaged" treasure:

Treasure.new(treasure.name, treasure.points / 2.0)

Now it's your turn! Use what you learned about super and create a new Treasure object to remove the duplication in the found\_treasure method of the ClumsyPlayer class.

class ClumsyPlayer < Player  
 def found\_treasure(treasure)  
 damaged\_treasure = Treasure.new(treasure.name, treasure.points / 2.0)  
 super(damaged\_treasure)  
 end  
end

Don't forget to re-run the clumsy\_player.rb and clumsy\_player\_spec.rb files to check your work. You haven't changed any functionality, so they should continue to work as before.

### 3. Define a Berserk Player

### Next up is the berserk player. This player also shares much of the same behavior as the normal players. What's unique about a berserk player, however, is he keeps track of how many times he's been w00ted. Five or fewer w00ts, and you couldn't tell him from any other mild-mannered player. But the sixth w00t puts him in a "berserk" state, and from then on every time you callblam on this player he overrules you by calling w00t instead. As a small consolation, he also prints out a warning every time you w00t him when he's gone berserk.

### Expressed in code, the goal this time is to be able to run this code:

berserker = BerserkPlayer.new("berserker", 50)  
6.times { berserker.w00t }  
2.times { berserker.blam }  
puts berserker.health

Running that code should result in the player getting a total of 8 w00ts (6 + 2), each increasing his health by 15, for a total health of 170 (50 + 120). In other words, the two blams count as two w00ts, instead.

If you want to work this in a test-first style or just want a sanity check on your work, copy the code examples below into a file called berserk\_player\_spec.rb.

require\_relative 'berserk\_player'

describe BerserkPlayer do  
 before do

@initial\_health = 50

@player = BerserkPlayer.new("berserker", @initial\_health)

end

it "does not go berserk when w00ted up to 5 times" do

1.upto(5) { @player.w00t }  
 @player.berserk?.should be\_false

end

it "goes berserk when w00ted more than 5 times" do

1.upto(6) { @player.w00t }  
 @player.berserk?.should be\_true

end

it "gets w00ted instead of blammed when it's gone berserk" do

1.upto(6) { @player.w00t }

1.upto(2) { @player.blam }

@player.health.should == @initial\_health + (8 \* 15)

end  
end

This inheritance example is a bit more challenging than the last. The key is to remember that you can use super (with parameters if necessary) to call the next ancestor's method of the same name. You can call super from anywhere within a method—it doesn't have to be the first line in the method, for example.

1. In a new file named berserk\_player.rb, define a BerserkPlayer class that inherits from Player.

require\_relative 'player'  
class BerserkPlayer < Player  
end

1. We want to create a berserk player the same way we create a normal player: initializing them with a name and an initial health value. However, each BerserkPlayer also needs to keep track of how many times he's been w00ted. That's part of a berserk player's state, so the w00t count needs to be stored in an instance variable in the BerserkPlayer.
2. To do that, override the initialize method in the BerserkPlayer class. The trick here is to avoid duplicating the default initialization code. Player should initialize the name and health. BerserkPlayer should only initialize the w00t count.

class BerserkPlayer < Player  
 def initialize(name, health=100)  
 super(name, health)  
 @w00t\_count = 0  
 end  
end

1. Looking ahead, we're going to need to know if a player has gone berserk. It would be handy to be able to ask a berserk player if he's in that crazed state. It's a simple true or false answer, and even though he's crazy, he'll tell us the truth.
2. Define a berserk? method that returns true if the value of @w00t\_count is greater than 5. Otherwise, the method should return false.

def berserk?  
 @w00t\_count > 5  
end

1. OK, now every time w00t is called on a berserk player the @w00t\_count variable needs to be increased by 1. If doing that puts the player in a berserk state, then a warning message such as "Berserker is berserk!" should be printed. However, all the default w00ting behavior defined in the w00t method of the Player class needs to happen as well: the player's health should increase and a message saying he was w00ted should get printed. Again, the trick is to use super to avoid duplication.

def w00t  
 super  
 @w00t\_count +=1  
 puts "#{@name} is berserk!" if berserk?  
end

1. Similar to what we did for the ClumsyPlayer, let's get some visible output on how a BerserkPlayer behaves at this point. Add the following example code to the bottom of the berserk\_player.rb file, after the class definition:
2. **if** \_\_FILE\_\_ == $0  
    berserker = BerserkPlayer.new("berserker", 50)  
    6.times { berserker.w00t }  
   **end**
3. Then run the berserk\_player.rb program file and you should see the player go berserk after the sixth w00t, like so:
4. Berserker got w00ted!  
   Berserker got w00ted!  
   Berserker got w00ted!  
   Berserker got w00ted!  
   Berserker got w00ted!  
   Berserker got w00ted!  
   Berserker is berserk!
5. Now that you can put a berserk player in the berserk state, it's time to act on it. When a player who has gone berserk gets blammed, it has the opposite effect: he winds up getting w00ted instead.
6. So, every time blam is called on a BerserkPlayer object, we'll need to check to see if the player has gone berserk. If so, then we'll w00t the player instead of blamming them. If the player hasn't yet gone berserk, we'll treat them like a normal player (getting blammed will invoke the default blam behavior).

def blam  
 if berserk?  
 w00t  
 else  
 super  
 end  
end

1. Here's what the complete BerserkPlayer class should now look like:

require\_relative 'player'

class BerserkPlayer < Player  
 def initialize(name, health=100)  
 super(name, health)  
 @w00t\_count = 0  
end  
  
def berserk?  
 @w00t\_count > 5  
end

def w00t  
 super  
 @w00t\_count +=1  
 puts "#{@name} is berserk!" if berserk?  
end

def blam  
 if berserk?  
 w00t  
 else  
 super  
end  
end  
end

1. As a quick visual check, change the example code at the bottom of your berserk\_player.rb file to look like this:
2. **if** \_\_FILE\_\_ == $0  
    berserker = BerserkPlayer.new("berserker", 50)  
    6.times { berserker.w00t }  
    2.times { berserker.blam }  
    puts berserker.health  
   **end**
3. Then run that file and you should see that the player gets w00ted a total of 8 times and ends up with a total health value of 170 (50 + 120):

Berserker got w00ted!  
Berserker got w00ted!  
Berserker got w00ted!  
Berserker got w00ted!  
Berserker got w00ted!  
Berserker got w00ted!  
Berserker is berserk!  
Berserker got w00ted!  
Berserker is berserk!  
Berserker got w00ted!  
Berserker is berserk!  
170

1. Visual checks are good, but we also have a spec that will automatically check the results. Run theberserk\_player\_spec.rb file to make sure all the code examples pass.
2. Finally, in the studio\_game.rb file, create a BerserkPlayer object with the name "berserker" and an initial health of 50. Then add the player to the game.

require\_relative 'berserk\_player'  
berserker = BerserkPlayer.new("berserker", 50)  
knuckleheads.add\_player(berserker)

1. After all your hard work, have a little fun! Run the game with a small number of iterations, and the berserk player should generally lose. Run the game with a large number of iterations (greater than, say, 500) to make the berserk player go berserk and he should always win!

### 

### Bonus Round

#### Give Your Clumsy Player a Boost

Ready to boost your inheritance skills? Here's a little challenge...

Specialize the behavior of the ClumsyPlayer so that he has a counter-balance to his clumsy ways. As an upside to only getting half points for each treasure, give him an extra boost (or ten) in health every time he's w00ted. To do that, initialize a clumsy player with a number representing his boost factor. Every time he's w00ted, give him that number of w00ts.

class ClumsyPlayer < Player  
 attr\_reader :boost\_factor

def initialize(name, health=100, boost\_factor=1)  
 super(name, health)  
 @boost\_factor = boost\_factor  
 end

def w00t  
 @boost\_factor.times { super }  
end  
end

clumsy = ClumsyPlayer.new("klutz", 105, 3)

Add code examples to the clumsy\_player\_spec.rb file to specify this behavior.

context "with a boost factor" do  
 before do  
 @initial\_health = 100  
 @boost\_factor = 5   
 @player = ClumsyPlayer.new("klutz", @initial\_health, @boost\_factor)  
end  
  
it "has a boost factor" do  
 @player.boost\_factor.should == 5  
end

it "gets boost factor number of w00ts when w00ted" do  
 @player.w00t  
 @player.health.should == @initial\_health + (15 \* @boost\_factor)  
end  
end

Experiment with creating more types of players with unique behaviors. Make sure to remove any unnecessary duplication that may crop up along the way.

Fundraising Program

Can you think of a different kind of project that is like a project we already have, but with slightly different behaviors? Use class-level inheritance in your fundraising program to share common characteristics across the projects and then create a subclass for the specialized behavior.

Need an idea to get the juices flowing? Add a grant project. It is a similar kind of project to the other projects you have but it assumes that funds cannot be removed from it.

Now try something more complex like a project that receives matching funds after it gets half of its funding secured. The sky's the limit when it comes to different kinds of projects, so be creative and have fun experimenting!

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

Wrap Up

Inheritance is a powerful design technique. From the conceptual standpoint, inheritance lets you model *is-a* relationships between parent classes and their children. For example, in our game a ClumsyPlayer is a specialized type of Player. That's a fairly tightly-bonded relationship, in the same way that inheritance in the real world is a lifelong relationship (whether you like it or not). From the coding standpoint, inheritance offers a convenient way to share code. Child classes inherit methods from their parents, and can override those methods or add new methods.

Be careful when you use inheritance in your programs, though. Use it only for true *is-a* relationships. When inheritance is shoe-horned into programs simply to share code, it becomes a real conceptual mess. Moreover, deep inheritance hierarchies are a sure sign of trouble to come.