Smalltalk Programming
Lesson 19

In the last lesson you added the ability for enemies to move left and right. You also enabled stepping for your enemy morph so that it is now moving on its own. In this lesson you will make changes to check for contact with an enemy. The code to check for contact with an enemy is important because it lets the shot know when it hits an enemy. An enemy cannot be destroyed if your shot cannot find it or know that it hit the enemy.

1. There are 4 steps the code needs in order to shoot and destroy an enemy. What might those 4 steps be?

2. The first would be the ability to find the enemy. The second would be to know if the shot hits an enemy. The third would be to remove the shot when it hits an enemy. The fourth would be to remove the destroyed enemy from the game screen.

3. First, you will need create a method to find enemies that exist on the game screen. Which existing class might work best for keeping track of enemies on the game screen?

4. The *ShooterGame* class would work best. It instantiates (creates) the enemy instances, so it works well. Create the method *ShooterGame>>enemies* using the code below.

enemies

```
^self submorphs select: [:morph | morph isMemberOf: Enemy]
```

5. Look at the line of code. Can you figure out what it is doing?

6. Second will be to determine if the shot hits an enemy. How will your shot know when it hits an enemy? Which class might work best for having a method to check this?

7. The *Shot* class works best for this. The *Shot>>move* method moves the shot. Each time the shot moves, a check should be made to see if the shot contacts an enemy.

8. Look at your *Shot>>move* method. Notice that in addition to moving the shot, the method also checks whether the shot moves off the game screen. This check would be better placed in its own method, since the *move* method should only be concerned with moving the shot, not checking if it contacts anything. The new method could also check if the shot comes into contact with an enemy.

9. To make these code changes, you will learn another powerful tool in Squeak/Smalltalk – refactoring. Refactoring means changing the way a program is written to make it cleaner and easier to understand, without changing how it works or what it does.

10. Start by selecting the highlighted code below in the *Shot>>move* method. Right click your mouse and you will see a menu similar to Figure 1. Select “refactor source” and then “extract method”. You will then receive a popup box, like Figure 2 below, asking for the method name to create. Enter *checkContact* for the name and then press enter, or select “Accept”.

move

```
self position: (self position) - (0@5).
self top < owner top ifTrue: [self delete]
```

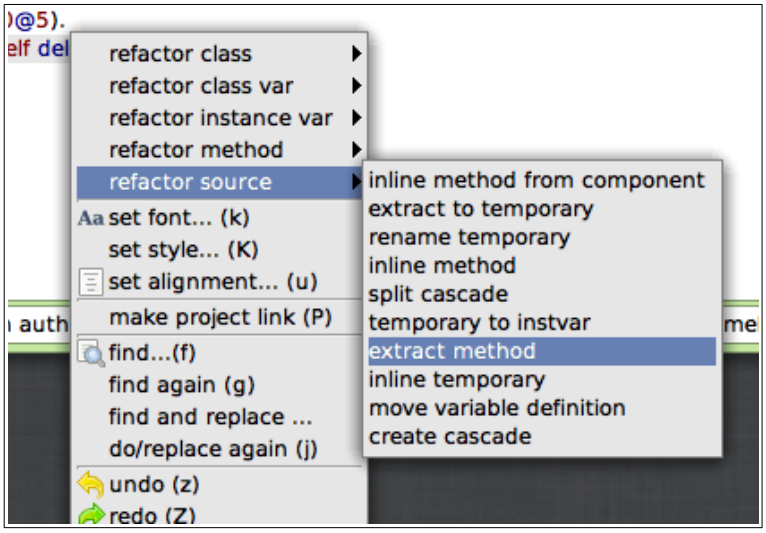

Figure 1: Selecting refactor source -> extract method

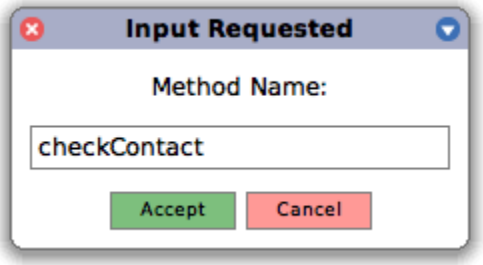

Figure 2: Entering name for new method

11. Look at the code changes that the refactoring made to your *Shot>>move* and *Shot>>checkContact* methods. This was a simple change, however refactoring can handle complex changes, making them seem very easy.

12. The code changes in *Shot>>checkContact* below will provide the second and third steps needed in order to destroy an enemy with a shot. Type and save the changes below.

```
checkContact
    self top < owner top
        ifTrue: [ self delete ].
    owner enemies
        do: [:enemy | (self bounds intersects: enemy bounds)
            ifTrue: [self delete]]
```

13. Look at each line of code. Can you figure out what each line of the new code is doing?

14. Test the code changes. What do you notice from the changes that you made?

15. You probably noticed a couple of things. First is that the enemy is still invincible. The second is that one or more debuggers were triggered.

16. The enemy is invincible because the code to remove it has not been added yet. This code will be added in the next lesson.

17. You will not use the debugger to determine the reason for the error this time. The debugger was triggered because the *Shot* instance was deleted as a submorph from your *ShooterGame* instance with "self delete". Once the *Shot* instance was deleted it was no longer a submorph of the *ShooterGame* instance (meaning it was no longer part of the game). Because it was deleted, the shot no longer had an "owner". The code triggers the debugger when evaluating "owner enemies" because the shot does not have an owner anymore. You can close the debugger(s).

18. The purpose of raising the error was to teach a couple of things. First, it teaches that objects can change during execution within a method. It is necessary to keep track of changes like this. Second, it teaches about the ^ character, which is called a *method return* in Smalltalk. The ^ character is used to return a value. When ^ is used, a value is purposely returned so that it can be used (although it does not always have to be used). Another effect of using the ^ is that it stops further execution in a method since the

method return immediately returns its value. The ^ allows the method to exit without having to run the remaining code.

19. Make use of this method return feature in *Shot>>checkContact*. Make the following change and then test it.

```
checkContact
   self top < owner top
      ifTrue: [ ^self delete ].
   owner enemies
      do: [:enemy | (self bounds intersects: enemy bounds)
         ifTrue: [self delete]]
```

20. The enemy is still invincible and it is still taunting you for not being able to defeat it! The changes that you will make in the next lesson will fix this.

21. Save and Quit your Smalltalk image.