

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.S184—Zombies Drink Caffeinated 6.001
 IAP, 2012

Project 4 PART 2

Release date: 2 February, 2012
 Due date: 7 February, 2012, at 11:59pm

The zombies continue banging on the door. Thinking quickly, Alyssa prints out a collection of puzzles from the Mystery Hunt and slides them under the door. The commotion dies down, and the zombies start shuffling about in the hallway asking for brains.

Alyssa shouts, “This won’t last. Back to work!”

Problem 1: Examining your Methodology (OPTIONAL)

One powerful feature Ben’s system has is *introspection*, which is the ability to programmatically examine the structure of classes and instances. This capability can be quite powerful. Every instance of `root-object` has the `GET-CLASS` method. Once you’ve obtained the class as an object, you can invoke any of the methods defined in `standard-metaobject`, including `GET-METHODS`, as you saw earlier.

Having the `ACT` method in `autonomous-person` individually call `AUTO-MOVE`, `AUTO-TAKE`, etc. is cumbersome. Also, if you ever added other `AUTO-` methods, you’d have to manually add their invocations. **Alter `ACT` to invoke all methods that start with `AUTO-` that the object has.** You’ll find `symbol-prefix?` in `oo-eval-part2.scm` handy and will want to add it to `oo-eval`’s list of primitives. What changes do you need to make in the `zombie` class as a result so that their behavior is not changed by your alteration of `autonomous-person`?

Problem 2: Mixin it up

Another powerful property of an object-oriented system is *intercession*, which allows dynamic changing of the properties of classes, objects, methods, slots, and so on.

Generically, a *mixin* is a set of methods that can be applied to augment an existing class or instance. Every computer language out there with mixins seems to define them slightly differently. The design of Ben’s object system allows for the possibility of mixing in additional methods to an individual instance, so let’s implement that.

Specifically, a *mixin* has a `name` and an association list of `methods`. When mixed into an instance, the instance gains the type and the methods of the *mixin*. This is best illustrated with an example:

```
(define name-changer
  (make-mixin 'NAME-CHANGER
    (list
      (list 'NAME-CHANGER
        (lambda (n) (set! :name n))))))

(define book (new named-object 'sicp))
```

```

(book 'NAME)
;=> sicp
((is-a 'NAMED-OBJECT) book)
;=> #t
((is-a 'NAME-CHANGER) book)
;=> #f
(instance-add-mixin! book name-changer)
(book 'CHANGE-NAME! 'way-of-kings)
(book 'NAME)
;=> way-of-kings
((is-a 'NAME-CHANGER) book)
;=> #t

```

Note how the type of the object changes, and it suddenly has an additional method. Furthermore, the method from the mixin has access to the private state of the instance!

First, in `oo-util.scm`, which is run by `oo-eval`, **implement a simple data abstraction for mixins**: `make-mixin`, `mixin?`, `mixin-name`, and `mixin-methods`. A simple tagged list would be a reasonable representation, but it's up to you.

You're going to implement mixins by sneaking a new class into the class hierarchy for the instance. This new class will have the name and methods from the mixin, but its super class will be the original class of the instance. Mixins bring no additional state to the instance.

In `oo-util.scm` again, **implement `instance-add-mixin!` to behave as described**. You will find that using `GET-CLASS` from `root-object` will be needed. You'll also find that you're going to want to be able to mutate the class of an instance, so add a `SET-CLASS!` method to `root-object`.

Hint: `instance-add-mixin!` will need to create a new class. However, the special syntax of `make-class` is going to get in your way. You may want to expose `create-class` or `standard-metaclass` from the underlying evaluator in `oo-eval.scm`. It is, however, also technically possible to solve this problem without any changes to `oo-eval.scm`.

Test out your behaviour with the above example, or others that you come up with.

Problem 3: Re-Zombification!

The current implementation of `create-zombie` is unfortunate because it destroys the victim and re-creates a new instance. Any state associated with the old instance is lost (`health`, for example). Alyssa still holds out hope that the zombification is temporary. Use your mixin implementation to improve zombification.

Create a `zombie-mixin` mixin with copies of the `NAME` and `BITE` methods from the `zombie` class. Change `create-zombie` to use this mixin, instead of instantiating a `zombie` object. The `zombie` class will no longer be needed at all. You'll find that `create-zombie` will now only require a single argument, `victim`.

When a zombie is created, it should announce the transformation and stop trying to take stuff. So when someone becomes a zombie, `create-zombie` should invoke a method, `BECOME-ZOMBIE` to do these things. You'll find that you can prevent the zombie from taking stuff by modifying a certain piece of object state inherited from `autonomous-person`. Include the `BECOME-ZOMBIE` method in your `zombie-mixin`.

When you've implemented zombies, remove the code in `setup` in `oo-world.scm` to create `george` and replace it with code to make a random person the zombie.

Problem 4: The obvious solution

Alyssa looks up from Ben’s code and yawns. “This is so frustrating! Nothing else in this pile of parentheses makes any sense! This pile of Cs, Ns, Hs, and Os probably means something important—a chemical formula?— but it’s getting harder and harder to concentrate....uhh..”

You offer her some of your soda.

“Thanks! That certainly should keep me fully awake a little longer.”

At the same time you both exclaim, “CAFFEINE!?!?”

Alyssa says, “Suddenly this rambling comment in Ben’s code about ‘weaponized RedBull’ makes a lot more sense. Maybe extreme doses of caffeine can reverse the transformation! Quickly, model the possibility of using caffeinated weaponry! I’ll see what I can scrounge up in the lab here...”

Add a new class to `oo-types.scm` called `weaponized-caffeine` which subclasses `weapon`. This class should override the `HIT` method such that when hitting something which is a zombie, it transforms them back into a person. To do that, write a procedure called `pop-class!` which takes an instance and undoes the effect of `instance-add-mixin!`. Note that this will require that you add a `SET-PARENT-CLASS!` method to `standard-metaclass` in `oo-eval.scm` because your system does not have the ability to add methods to classes at runtime. The weapon should `DESTROY` itself after being used.

Modify setup in `oo-world.scm` to add several instances of this class to the world.

Add a method to the `autonomous-person` class named `AUTO-CURE` which will look to see if there is a `weaponized-caffeine` object in its inventory and a `zombie` at the current location. If so, it will invoke the weapon’s `HIT` method on the zombie. If you did not do problem 1, then there will be an extra step to perform to cause autonomous people to try this.

Run the simulation and see if this stems the zombie menace. Try differing quantities of people and caffeine to see what sort of ratio would work best.

It’s all in the delivery

You finish up your simulation and tell Alyssa, “I think it can work, if we can find a reasonable means of delivery!” when the zombies return in force. In a matter of moments they’ve bashed an opening in the door and are trying to climb through and over the barricade.

You try to e-mail and Zephyr your results to everyone you can think of only to find that there’s a problem with the lab’s network connection. You say, “Well, Alyssa, if you’ve got any bright ideas, now would be the time...”

“How’s your aim?!” she yells, and throws a liquid-filled balloon at the zombies. You see that she found some party balloons somewhere and had been filling them up with soda! After several volleys, the zombies start to relent and then finally collapse.

A few minutes later, Ben Bitdiddle wakes up. “Ow, my aching head. What happened? Why am I lying in a heap of students next to a ... barricade?”

You start to try to explain, when Ben interrupts, “Hey! I just had the greatest idea! Since I have an evaluator which supports object-oriented programming, I should be able to rewrite `oo-eval` to use objects to store all of its state, including the environment, and then I could run `oo-eval` inside `oo-eval!`”

Alyssa just hits him with an inflatable lambda.

Problem 5: Your turn (OPTIONAL)

Now that you've had an opportunity to play with both an implementation of an object system and our "world" of characters, places, and things, we want you to extend the object system and/or the world in some substantial way. The last part of this project gives you an opportunity to do this. As you haven't had much freedom to design your own code up until now, this exercise gives you the opportunity to demonstrate your knowledge of design principles.

You could dive into the object system further. Or you could just have fun with the game! You should add at least one new OOP behavior or class to the system. You don't have to stick with the zombie theme or the MIT locale if you don't want to. Here are a few ideas that we've come up with for extensions, but you certainly aren't limited to these:

Sleep-deprivation Some hypothesize that sleep deprivation caused the zombie plague. Make students go to sleep every once in a while. If they don't sleep for long enough, they turn into a zombie!

Scared-masses Make people who run from zombies.

Game Actually implement some sort of game with a win-condition. For instance, turn in your last `pset` to a `professor` so that you can go to Killian Court and get your `diploma`.

REPL Add a simple Read-Eval-Print-Loop so that you can play the game with simpler commands like `(GO NORTH)` and `(GET LAMP)`.

Multiple inheritance Add multiple inheritance to the system. How many different pieces of code would be touched by this change?

MOP Improve the `Metaobject Protocol`. Add the ability to add/change methods and slots dynamically at runtime. Add the ability to subclass the `standard-metaclass` and use the result when creating new classes. Move the inheritance mechanism into the metaclasses, so the type of behavior you get depends on which metaclass you select when you create a class.

Garbage collection Instances are added to the global table of instances by `make-instance`, but they are never removed. When a new instance is requested and the table has reached a certain size, go through it and discard items that are no longer reachable from any other objects. What's the root set from which to start your search?

Crazier stuff Racket has lots of libraries for things like graphics, networking, a web server, etc. Go nuts.

Submission

Your submission should be one (or more) text files containing your solutions to the project—code modified, added, etc. Please make sure it is easy for the course staff to spot what you've done for each problem. Be sure to include test cases that show your code works.

We encourage you to work with others on projects as long as you acknowledge it. If you cooperated with other people, please indicate your consultants' names and how they collaborated.