AP® COMPUTER SCIENCE A
2007 SCORING GUIDELINES

Question 2: Pounce Fish (MBS)

Part A: findFish **5 points**

+2 access & check neighbor
- **+1/2** determine current location
- **+1/2** determine current direction
- **+1/2** correctly access any neighbor
- **+1/2** determine if neighbor location is empty

+1 1/2 loop in forward direction
- **+1/2** loop with respect to range
- **+1** access up to `range` consecutive forward locations (as needed)

+ 1 1/2 return value
- **+1/2** return null if reach invalid location in loop
- **+1/2** return object at first non-empty location in loop
- **+1/2** return null if no non-empty location in loop

Special Usage:

-1 missing or incorrect environment access

Part B: act **4 points**

+1/2 call `findFish()`

+1/2 test if `findFish` returned null

+2 not null case
- **+1** `prey.die()` or `environment().remove(prey)`
- **+1** change location to prey's location

+1 null case
- **+1/2** attempt to act (`move()` or `super.move()` OK)
- **+1/2** `super.act()`

Question 2: Pounce Fish (MBS)

PART A:

```
private Fish findFish()
{
    Environment env = environment();
    Location loc = location();
    Direction dir = direction();

    for (int i = 0; i < range; i++) {
        loc = env.getNeighbor(loc, dir);
        if (!env.isEmpty(loc)) {
            return (Fish)env.objectAt(loc);
        }
    }
    return null;
}
```

PART B:

```
public void act()
{
    if (! isInEnv() )
        return;

    Fish prey = findFish();
    if (prey != null) {
        prey.die();             // OR environment().remove(prey);
        changeLocation(prey.location());
    }
    else {
        super.act();
    }
}
```

A2a,

(a) Write the `PounceFish` method `findFish`. If any fish are located within `range` locations in the direction that the `PounceFish` is currently facing, the method returns the nearest of these. Otherwise, the method returns `null`.

Complete method `findFish` below.

```
/** Looks ahead range locations in current direction
  * @return the nearest fish in that direction within range (if any);
  *         null if no such fish is found
  */
private Fish findFish()
{
   Environment env = environment();
   Location frontLoc;

   for (int index = 0; index < range, index ++)
   {
      frontLoc = env.getNeigbir(location(), direction())
      if (env. isEmpty(front Loc) == false)
      {
           return [fish] env.objectAt(frontLoc);
      }

   }
   return null;
}
```

GO ON TO THE NEXT PAGE.

A2a$_2$

(b) Override the `act` method for the `PounceFish` class. A `PounceFish` attempts to find a fish that it can eat. If it finds such a fish, the `PounceFish` eats it (causing it to die) and moves to its location. If the `PounceFish` does not find a fish that it can eat, it acts as an ordinary fish.

In writing `act`, assume that `findFish` works as specified, regardless of what you wrote in part (a).

Complete method `act` below.

```
/** Acts for one step in the simulation
 */
public void act()
{
  if ( ! isInEnv() )
    return;

  // Write your code below.
  if( findfish() == null)
      super.act();
  else
  {
      Fish  foundfish = findFish();
      Location newLoc = foundfish.location();
      foundfish.die();
      changeLocation(newLoc);
  }
}
```

GO ON TO THE NEXT PAGE.

A2b,1

(a) Write the `PounceFish` method `findFish`. If any fish are located within `range` locations in the direction that the `PounceFish` is currently facing, the method returns the nearest of these. Otherwise, the method returns `null`.

Complete method `findFish` below.

```
/** Looks ahead range locations in current direction
 *  @return the nearest fish in that direction within range (if any);
 *          null if no such fish is found
 */
private Fish findFish()
    Location loc = env.getNeighbor(location(), direction());
    for(int i=0; i <=range; i++)
        If (!env.isEmpty(loc))
            return env.objectAt(loc);
        loc = env.getNeighbor(loc, direction());
    return null;
```

GO ON TO THE NEXT PAGE.

A2₂

(b) Override the `act` method for the `PounceFish` class. A `PounceFish` attempts to find a fish that it can eat. If it finds such a fish, the `PounceFish` eats it (causing it to die) and moves to its location. If the `PounceFish` does not find a fish that it can eat, it acts as an ordinary fish.

In writing `act`, assume that `findFish` works as specified, regardless of what you wrote in part (a).

Complete method `act` below.

```
/** Acts for one step in the simulation
 */
public void act()
{
  if ( ! isInEnv() )
    return;

  // Write your code below.

if(findFish() != null)
    Location newLoc = new Location(findFish(). location());
    findFish().die();
    changeLocation(newLoc);
  else
    super.act()
```

GO ON TO THE NEXT PAGE.

A2c,

(a) Write the `PounceFish` method `findFish`. If any fish are located within `range` locations in the direction that the `PounceFish` is currently facing, the method returns the nearest of these. Otherwise, tl method returns `null`.

Complete method `findFish` below.

```
/** Looks ahead range locations in current direction
 *  @return the nearest fish in that direction within range (if any);
 *          null if no such fish is found
 */
private Fish findFish()
{
    if (getDirection.equals("East")|| getDirection.equals("West"))
    {
      for(i=0; i<=range; i++)
      {
        if (!isEmpty (Location(this.row()+1, this.col())))
            return objectAt(Location(this.row()+1, this.col()));
      }
      return null;
    }
    if(getDirection.equals("North")|| getDirection.equals("South"))
    {
     for(i=0; i<=range; i++)
     {
       if(!isEmpty(Location(this.row(), this.col()+1)))
           return objectAt(Location(this.row(), this.col()+1));
     }
     return null;
    }
}
```

GO ON TO THE NEXT PAGE.

A2C2

(b) Override the act method for the PounceFish class. A PounceFish attempts to find a fish that it can eat. If it finds such a fish, the PounceFish eats it (causing it to die) and moves to its location. If the PounceFish does not find a fish that it can eat, it acts as an ordinary fish.

In writing act, assume that findFish works as specified, regardless of what you wrote in part (a).

Complete method act below.

```
/** Acts for one step in the simulation
 */
public void act()
{
  if ( ! isInEnv() )
    return;

  // Write your code below.
  if(findFish())
  {
    Fish f1 = new Fish;
    f1.die;
    this.changeLocation(Location(f1.row(),f1.col()));
  }
  else
    super.act();
}
```

GO ON TO THE NEXT PAGE.

Question 2

Overview

This question was based on the Marine Biology Simulation (MBS) case study and dealt with abstraction and inheritance. Students needed to show their understanding of the case study and its interacting classes by writing member functions for a new `PounceFish` class. In part (a) students were required to implement the private `findFish` method, which searched ahead to find and return the nearest neighboring fish, if there was one within range. In part (b) the students were required to override the Fish `act` method to produce the desired behavior. This involved calling the `findFish` method from part (a) to determine if a fish was within range, and if so, to pounce on that fish (that is, remove the fish from the environment and move to its old location). If no fish existed within range, it behaved as a normal fish by calling the `super.act()` method.

Sample: A2a
Score: 8

In part (a) the student correctly initializes `env`. The student subsequently uses `location()` and `direction()` in a `getNeighbor` call that is syntactically correct and uses that neighbor's location in a correct call to `isEmpty`, so the first 2 points were earned. The `for` loop uses `range` and accesses locations within its body to earn the initial loop ½ point. Since `frontLoc` is not advanced for each iteration of the loop, the student needed to use `frontLoc` instead of `location()` in `getNeighbor` call, so the consecutive access point was not earned. The logic for returning the result of the call is correct in all cases (note: using brackets instead of parentheses is a nonpenalized usage error). Part (a) earned 4 of 5 possible points.

Part (b) earned all 4 points. The `findFish` calls are correct, and the test to determine if `findFish` returned `null` is constructed correctly. The prey fish is correctly killed, and this `PounceFish` is moved to the prey's location. If no prey was present, this `PounceFish` acts as a regular fish.

Sample: A2b
Score: 6

This solution earned 4 of 5 possible points for part (a). In the first line of code, the student correctly determines the fish's current location and direction. If `env` had been declared and initialized, the `getNeighbor` call would have been perfect. Since `env` is used in multiple parts of the solution for this question, there was a 1-point special usage deduction assessed to cover all instances of its use in solving this problem. Therefore the `getNeighbor` call earned the ½ point awarded for its call, as did `isEmpty`. Another ½ point was awarded for the loop, but the full point for accessing consecutive forward locations was not awarded because of an off-by-one error within the loop. The returns are correct in all three cases.

Part (b) is logically correct, and points were earned for calling `findFish` and testing for `null`. The only error is within the first statement of the not-null case. Because there is no copy constructor in `the Location` class, `newLoc` will not reference the prey's location. This has no effect on the prey dying, so that point was earned; however, the point for moving to the prey's location was not awarded. All other points in part (b) were earned.

Question 2 (continued)

Sample: A2c
Score: 3

This solution contains no call to `location()`. The calls to `row()` and `col()` are not used correctly because they are `Location` methods, not `Fish` methods. There is no call to `direction()`. The calls to `getDirection` are not correct because `getDirection` is an `Environment` method, but it is being used as a `Fish` method. Thus the student lost the first two ½ points. The `isEmpty` call is missing the required `Environment` object, as are the `objectAt` calls. A 1-point general usage deduction for missing environment was applied, allowing the rest of the problem to be scored as if `environment()` were included. The parameter used in the `isEmpty` call is now considered a location (this error was penalized because the ½ point allocated for correctly accessing any neighbor was not awarded), so the ½ point for determining if a neighboring location is empty was awarded. There are loops present using `range`, but no credit was given for consecutive access of locations because of the off-by-one error and the hard-coded direction errors. The student uses locations consistently and the logic is correct, so all three ½ points allotted for returns were earned.

In part (b) the `findFish` call is present and that ½ point was awarded, but the student incorrectly constructs the conditional test by treating an object/null as a `boolean true/false`, so the ½ point that was available for the null test was lost. Since `f1` is not the same as the fish returned by `findFish()`, the wrong prey is killed. There is also an incorrect construction of `Location` in the `changeLocation` call, so the student lost the 2 points awarded for these actions. The two ½ points were awarded for the appropriate use of `super.act()` in the case where prey is not present.