CS112 Lecture: Extending Classes and Defining Methods

Last revised 1/17/08

Objectives:

1. To introduce the idea of extending existing classes to add new methods
2. To introduce overriding of inherited methods
3. To introduce stepwise refinement

Materials:

1. Projectable of Problem 2.5
2. BlueJ project containing Problem2_5, Problem2_5_Revised, RightTurnerRobot, plus Problem2_5.world
   PlantDiagonal1, DiagonalMoverRobot1, PlantDiagonal2, DiagonalMoverRobot2, plus PlantDiagonal.world
   Problem3_5 plus Problem3_5.world
3. Projectable of MileMover example in book
4. Projectable of Problem 3.5

I. Defining new Instructions for Robots

A. In our first look at our robots, we saw that they are capable of executing just 5 primitive methods: move(), turnLeft(), pickBeeper(), putBeeper(), and turnOff(). All robot programs ultimately are built up out of combinations of these methods.

B. At the same time, though, we saw that working with just these primitives can be a bit awkward. For example, to turn a robot to the right, we had to code a sequence of three turnLeft()s. This is undesirable for two reasons:

   1. It involves extra work for the programmer.

   2. It makes the program obscure to the human reader. A human would find it easier to understand the program if we could use a turnRight() operation instead of three turnLeft()s.

C. Fortunately, there is a built-in mechanism for expanding the vocabulary of our robots.

   1. What we must do is to define a new class of Robot as an *extension* of an existing class.

2. This new class *inherits* all the methods of the class it is derived from.

3. We then define additional methods for this new class of robot, perhaps utilizing the methods it inherited from the class it is derived from

4. Finally, when we create a new robot using the new operation, we create a robot of our newly-defined class, rather than the base class Robot as we have been doing.

   EXAMPLE: The newspaper retrieval program, redone using a new class of robot called RightTurnerRobot that has a turnRight() method.

   a) Project original problem

   b) Show code for original problem - note several lines that have three turnLeft() instructions in a row.

   c) Show definition for class RightTurnerRobot

      (1) extends Robot

      (2) new turnRight() method definition - *NOTE*: within the method definition we don't explicitly name the robot to whom the turnLeft() messages are begin sent - they are being sent to the same robot to whom the turnRight() method was sent. (We could make this explicit by using the Java reserved word *this)*

      (3) RightTurnerRobot constructor that calls constructor for superclass. (To construct a RightTurnerRobot, in effect we first construct an ordinary Robot and then add the new method to it. Note how starting position etc. is passed on to the super constructor.)

   d) Show revised code for task object using turnRight() instruction (Problem2_5_Revised) on a RightTurnerRobot

      (1) In the task object, we now construct a RightTurnerRobot using new (and declare the variable to be of this type)

      (2) Use of turnRight() method where before we had three turnLeft()s

   e) DEMO compilation of task class - note how two class files are created.

   f) DEMO run of revised program

D. Some Naming Conventions

1. The names of newly defined classes follow the class naming convention - the name begins with an *uppercase* letter, and each new word in the name is capitalized.

2. Frequently, the name of an extended class includes the name of the original class - hence RightTurnerRobot. (Of course, this has to be done with caution or we end up with an impossibly-long name!)

3. New methods follow the convention of the name beginning with a *lowercase* letter, and each new word in the name is capitalized - hence turnRight().

4. The name of a new method should clearly indicate to the human reader what the method does. The robot, of course, does not care what name we use - e.g. we could have called our turnRight() method foo() or even turnAround() as far as the robot is concerned (but if the definition called for three turnLeft()s, the robot would still turn right!)

E. Practice: define a new instruction turnAround() that turns the robot to face in the opposite direction from where it is currently facing. (We'll put it in the RightTurnerRobot class for demonstration purposes).

Modify the task to use this (replace two turnLeft's with a turnAround just after // Go back to door).

Modify the Robot class to add instruction

II. Using New Methods to Define Additional Method

Once a new method has been defined, it in effect becomes a new primitive that can then be used in defining other methods.

A. EXAMPLE: Consider the task of having Karel plant a diagonal line of beepers on the corner of 1st St & 1st Ave, 2nd St & 2nd Ave, 3rd St & 3rd Ave .. 10th St & 10th Ave. The program would begin something like this (assuming Karel starts out at 1st & 1st with a bag of 10 beepers, facing east):

```
karel.putBeeper();
karel.move();
karel.turnLeft();
karel.move();
karel.turnRight();
```

```
karel.putBeeper();
karel.move();
karel.turnLeft();
karel.move();
karel.turnRight();
...
```

B. Life would be simpler if we had a class of robots having a capability called moveDiagonally(), perhaps defined as follows:

```
void moveDiagonally()
{
     move();
     turnLeft();
     move();
     turnRight();
}
```

C. Then our main program becomes:

```
karel.putBeeper();
karel.moveDiagonally();
karel.putBeeper();
karel.moveDiagonally();
karel.putBeeper();
karel.moveDiagonally();
karel.putBeeper();
karel.moveDiagonally();
...
```

D. There are actually two different ways of doing this job:

1. We could create a new subclass of RightTurnerRobot, perhaps called DiagonalMoverRobot

   SHOW PlantDiagonal1, DiagonalMoverRobot1
   DEMO operation

2. We could put both turnRight() and moveDiagonally() in the same class

   SHOW, DEMO PlantDiagonal2, DiagonalMoverRobot2

   Note that we have used turnRight() as part of the definition of moveDiagonally() in the same class.

3. Which approach is better?

   a) Probably more opportunities for reuse by making RightTurnerRobot its own class - it can be reused in many situations.

   b) However, making each extension its own class can lead to a proliferation of classes, so we don't want to carry this too far, creating an inheritance hierarchy that is excessively deep.

   c) In this case, the turnRight() capability is so simple that it makes little sense to create a separate class just for this!

III. Inheriting Methods; Over-Riding Inherited Methods

A. Note that, when we extend a class to define new methods, we ordinarily *inherit* all the methods of the base class we are extending.

   1. *Example:* RightTurnerRobot inherits the methods move(), turnLeft(), etc. from Robot.

   2. *Example:* In our first solution to the plant diagonal problem, DiagonalMoverRobot() inherits turnRight() from RightTurnerRobot plus all the methods that RightTurnerRobot inherited from Robot.

B. On occasion, instead of simply inheriting a method, a subclass may need to redefine the method to either add additional behavior or perhaps change its behavior totally. We call this *overriding* the method.

   1. Really good examples of this are a bit hard to develop in the robot world. The book mentioned one:

      ASK CLASS

      Creating a robot class called Mile_Mover that moves a mile (8 blocks) every time it is told to do a move.

      This is done by overriding move to call the original, inherited move 8 times:

      PROJECT Mile_Mover example from book

      Actually, this is a bit questionable, since a method whose effect is well-known now has surprising behavior. It would be easy to inadvertently write a program that sends a robot into a wall by forgetting that move() now means go 8 blocks.

   2. We will see better examples of overriding later in the course.

IV. Stepwise Refinement

A. We introduced the idea of defining new instructions as a mechanism for simplifying the task of writing robot programs, and for making the resultant programs more readable. It also turns out that this idea is related to a powerful concept for designing complicated programs called **STEPWISE REFINEMENT**.

B. Suppose we are given a fairly large and complex task for Karel to accomplish. It may not be immediately obvious how we are to go about writing a robot program to accomplish the task. However, we often can get a start on the job by breaking it up into smaller parts. For example, consider problem 3.5 in the book

PROJECT - Problem 3.5

1. Since the assignment allows us to choose Karel's starting position, let's choose to put him at the bottom left corner of the "H", facing north. Suppose we had available to us procedures for drawing each of the letters of the alphabet, which start with the robot at the lower left corner of the letter and leave it in position to start the next letter, again facing north. Then, our basic task would be fairly simple:

```
karel.drawH();
karel.drawE();
karel.drawL();
karel.drawL();
karel.drawO();
```

2. Now, what we have done is to convert our original problem into four new problems - defining instructions for drawing each of the letters. Let's consider drawH() first:

   a) The H obviously consists of three parts: two vertical bars of 5 beepers each, plus a horizontal bar of 2 beepers.

   b) If we had available to us methods for drawing these bars, then we could define drawH() fairly easily. We will call these bar5() and bar2(). It will be convenient to assume that each method can expect the robot to be on the right square facing the right way at the outset, and will leave it at the other end of the bar facing in the same direction upon completion.

c) It will also be helpful to define a method move2() that moves two squares - analogous to bar2(), but without actually putting down beepers

d) Finally, we will make use of turnRight() and turnAround() defined previously.

e) We can now define drawH as follows:

```
void drawH()
{
  bar5();
  turnAround();
  move2();
  turnLeft();
  move();
  bar2();
  move();
  turnLeft();
  move2();
  turnAround();
  bar5();
  turnLeft();
  move2();
  turnLeft();
}
```

f) Defining the three methods bar2(), bar5() and move2() is fairly simple:

ASK CLASS

3. The remaining methods can be defined similarly:

   ASK CLASS

4. At this point, we are ready to define a new class called HelloDisplayerRobot. We will give it a method called displayHello which the main program will call.

5. *SHOW*: Complete program: Problem3_5, HelloDisplayerRobot
   DEMO

C. The idea behind stepwise refinement is this:

1. Break the original problem into a logical series of steps, assuming, for the moment, that you have code available to perform each of them.

2. Define each of these steps as a new method, and refine each into still smaller steps.

3. Continue the process until all steps are primitives. Where possible, try to develop general-purpose methods that can be used multiple times (e.g. turnRight(), bar5(), etc.)

D. Note that often the difference between a good solution to a problem and a bad one (in terms of programming effort, elegance) is a good initial refinement. Be willing to spend some time on this.