Sketches of Some CS1 Programming Assignments*

Paul Jackowitz[1] and Robert McCloskey[2]
Computing Sciences Department
University of Scranton
Scranton, PA 18510
[1]paul.jackowitz@scranton.edu
[2]robert.mccloskey@scranton.edu

Abstract

Among the most important aims of CS 1 is to provide students with a set of programming assignments that allows them to develop fundamental software development skills. Here we give sketches of CS 1 assignments involving the drawing of "ASCII figures" and the printing of cumulative song lyrics, inspired by our use of the textbook by Reges & Stepp (*Building Java Programs — A Back to Basics Approach* [1]). These assignments could be adapted to courses that utilize other textbooks or employ languages other than Java.

1 Introduction

The central goal of CS 1, at least for students majoring in computing, is to begin to develop fundamental software development skills. As the old adage goes, you "learn by doing". Hence, having a good set of programming problems to work on is key in attaining that goal. Here we provide sketches of several programming assignments that we have employed in teaching CS 1 for the past decade, in the hope that others who teach the course will find them, or

adaptations thereof, to be useful. (Complete descriptions are available from the authors.) The assignments reflect our use of *Building Java Programs — A Back to Basics Approach*, by Reges & Stepp [1], which takes the view that objects can wait until after a novice programmer has gained some skill in using the basic procedural programming constructs, including variable declaration and assignment, arithmetic expressions, conditional (i.e., if-else) statements, loops, and method calls/declarations. However, they could be adapted to courses that use other textbooks or other popular languages, including Python, C++, or C#.

2 Procedural Decomposition

Among the difficulties in teaching CS 1 is devising meaningful programming exercises early in the semester, when students have been exposed to only a small subset of a programming language's constructs. Part of the solution lies in making a good choice regarding the order in which those constructs are introduced. In this respect, [1] is rather unconventional.

The first programming principle presented in [1] is procedural decomposition, which suggests that a complex task should be divided into a set of simpler subtasks. In terms of programming in Java, this translates into an appropriate use of methods. In particular, if a subtask is performed multiple times, and in different contexts, that would be a strong indicator that it "deserves" to be implemented in its own method.

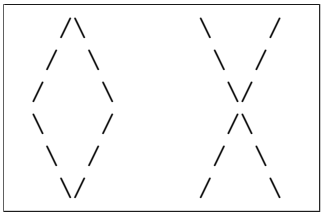

Figure 1

To illustrate this principle, all that is needed are "print" statements and (parameterless) methods, which together can be employed to produce primitive "drawings", such as those in Figure 1 (which are taken from [1]). The idea is that a well-modularized program that draws both figures would reflect the fact that the top half of the diamond shape (a cone) is produced in the same way as the bottom half of the 'X', and vice versa (a V-shape), so that the methods that draw a diamond and an 'X', respectively, would rely upon lower-level methods (containing only statements that print string literals) that draw a cone and a 'V':

```
public static void drawDiamond() { drawCone(); drawV(); }
public static void drawX()       { drawV(); drawCone(); }
```

After presenting this example —and perhaps a few others with slightly more complexity— we typically ask students to make extensive use of procedural

decomposition in developing a program that "draws" one or more figures having repeated parts.

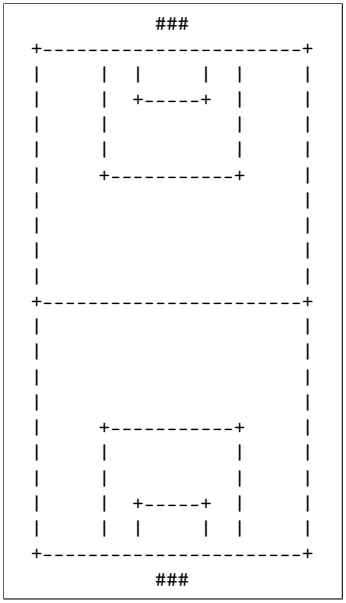

Figure 2

One example is the rendering of the soccer field shown in Figure 2. Notice that there are seven distinct lines of text, each of which appears two or more times in the drawing. Figure 3 shows excerpts from a solution. (There and elsewhere in figures that present source code, we use ellipses to indicate missing parts, omit the prefix "`System.out`" from calls to methods that print strings, and compromise formatting convention by placing multiple statements per line along with other accommodations, so as to save space and provide emphasis.)

Among the figures that we have used as the basis of assignments are a football field, "boxes" of various shapes stacked upon one another, and others that are rather contrived. Students are expected to recognize the portions of the figures that appear multiple times and, for each one, to formulate and make use of a method that draws it. To be successful, a student must have a clear understanding of how method calls affect a program's flow of control.

The more essential point has to do with motivating students to recognize the potential merit of refactoring a highly redundant sequence of print statements into an equivalent modularized program. We emphasize the potential problems redundant statements may present in software. We ask, what if a statement, used in multiple places, is subsequently determined to be incorrect, or what if such a statement is desired to be modified (for example, to change the width of the soccer field)? How certain might one be that they have found and modified all of the appropriate statements in need of change? Developing an appreciation of such issues, when they can readily see the results in the output, helps to prepare students to apply similar thinking later on when working with much more involved tasks where abstraction will be vitally important. Even recognizing that there may be multiple worthwhile ways to decompose a task has merit, as it may, and should, lead to discussions regarding the relative merits of different approaches.

A quite different kind of programming exercise by which to teach procedural decomposition — again needing only print statements and parameter-less

```
public class SoccerField {

  static void main(String[] args)
  {
    goal(); widthLine();
    northPenaltyBox();
    midField();
    midfieldLine();
    midField();
    southPenaltyBox();
    widthLine(); goal();
  }

  static void northPenaltyBox()
  {
    goalBoxSide();
    goalBoxTop();
    penaltyBoxSide();
    penaltyBoxSide();
    penaltyBoxTop();
  }

  static void midField() {
    midFieldSide();
    midFieldSide();
    midFieldSide();
    midFieldSide();
  }

  static void midfieldLine()
    { widthLine(); }

  static void goal()
    { println(" ### "); }

  static void widthLine() {
    println("+----------------------+");
  }

  static void goalBoxSide() {
    println("|     |  |     |  |     |");
  }

  static void goalBoxTop() {
    println("|     |  +-----+  |     |");
  }

  static void penaltyBoxSide() {
    println("|     |           |     |");
  }

  ...

}
```

Figure 3

methods— is to have students develop a program that prints the lyrics of a cumulative song (i.e., one in which each verse extends the previous one) [2].

The classic example is *The Twelve Days of Christmas*. We have used *Old MacDonald Had a Farm*, *There Was an Old Lady Who Swallowed a Fly*, and the Irish folksong *Rattlin' Bog*. To illustrate, Figure 4 shows excerpts of a Java program that prints the lyrics of one version of *Old MacDonald* and that demonstrates a significant use of procedural decomposition, along with (admittedly) method chaining.

3 Loops

Once students have been introduced to variables, assignment, and arithmetic expressions (which [1] does in Chapter 2), more figure-drawing exercises can be used as a vehicle for exploring iteration and generalization using the for-

```
public class OldMacDonald {

  public static void main(String[] args) {
    preamble(); print("a cow, "); eieio(); verseMoo();
    preamble(); print("a duck, "); eieio(); verseQuack();
    preamble(); print("a pig, "); eieio(); verseOink();
    preamble(); print("a dog, "); eieio(); verseWoof();
  }

  ...

  private static void verseQuack() {
    println("With a quack-quack here and a quack-quack there");
    println("Here a quack, there a quack, everywhere a quack-quack");
    verseMoo();
  }

  private static void verseOink() {
    println("With an oink-oink here and an oink-oink there");
    println("Here an oink, there an oink, everywhere an oink-oink");
    verseQuack();
  }

  ...

  private static void preamble()
    { oldMacHadAFarm(); andOnThisFarm(); }

  private static void oldMacHadAFarm()
    { print("Old MacDonald had a farm, "); eieio(); }

  private static void andOnThisFarm()
    { print("And on this farm he had "); }

  private static void eieio() { println "E-I-E-I-O"; }

}
```

Figure 4

loop construct. As an example, consider once again the diamond and X-shapes discussed earlier. Figure 5 shows most of a program that prints the two shapes. An easy-to-change global constant, `SIZE`, determines the size of each one. Other figure-drawing programs (e.g., for the soccer field) can be refactored in a similar way so that the sizes of the produced figures are determined by global constants.

The point here is for students to learn how to generalize their programs so that straightforward differences in output can be effected without having to make wholesale modifications to the source code. Of course, once students learn how to assign values to variables through input at runtime, those inputs can be used in place of constants, thereby yielding programs that are sensitive to user (or file) input.

```
public class drawDiamondAndX {
  private static final int SIZE = 6;

  public static void main(String[] args)
    { drawDiamond(); drawX(); }

  public static void drawDiamond()
    { drawCone(); drawV(); }

  public static void drawX()
    { drawV(); drawCone(); }

  public static void drawCone() {
    for (int i=1; i <= SIZE; i++) {
      for (int j=0; j != SIZE-i; j++) {
        { print(" "); }
      print("/");
      for (int j=0; j != 2*(i-1); j++)
        { print(" "); }
      println("\\");
    }
  }
  ...
}
```

Figure 5

4 Parameterization

Parameter passing is introduced in Chapter 3 of [1], providing the opportunity to further improve the figure-drawing (and lyric-producing) programs by making use of parameters —in place of global constants or variables— to dictate the behavior of methods.

This is our next step in teaching students when, how, and why to use methods. By utilizing parameters, methods become more self-contained, insulating them from being coupled with external data and thereby increasing their potential for reuse.

The task of having a program draw a checkerboard-like pattern, as in Figure 6, is one that we have employed at this point in the course. This pattern can be described by multiple parameters, including ones indicating the height and width of each cell and the number of rows and columns on the board. Additionally, the characters employed in printing the alternating "light" and "dark" cells can be parameters.

```
public class CheckerBoard {
  static final char DARK_CHAR = '#';
  static final char LIGHT_CHAR = '-';

  static void main(String[] args) {
    int boardSize = 3;
    int cellSize = 4;
    drawBoard(boardSize,cellSize);
  }

  static char otherOf(char c) {
    if(c == DARK_CHAR)
      { return LIGHT_CHAR; }
    else
      { return DARK_CHAR; }
  }

  static void drawBoard(int boardN,
                        int cellN) {
    char start = DARK_CHAR;
    for(int i=0; i != boardN; i++) {
      drawRow(start,boardN,cellN);
      start = otherOf(start);
    }
  }

  static void drawRow(char start,
                      int columns,
                      int cellSize) {
    for(int i=0; i != cellSize; i++) {
      drawLine(start,columns,cellSize);
      println();
    }
  }

  static void drawLine(char start,
                       int columns,
                       int cellWidth) {
    for(int i=0; i != columns; i++) {
      print(start,cellWidth);
      start = otherOf(start);
    }
  }

  static void print(char c, int n) {
    for(int i=0; i != n; i++)
      { print(c); }
  }
}
```

Figure 7

Figure 7 presents the source code for a program that prints such a checkerboard-like pattern. The character variables are global and the board and cells are square, so as to limit the number of parameters to two, but the essential idea is illustrated. Indeed, one can provide this program to students and ask them to generalize it so that the `drawBoard()` method receives all six potential parameters rather than only two.

```
####----####
####----####
####----####
####----####
----####----
----####----
----####----
----####----
####----####
####----####
####----####
####----####
```

Figure 6

Figure 8 presents the source code for a program that is a refactoring of the Old MacDonald program to make use of parameters, in this case strings that represent the animal names and utterances. This program also presents an opportunity for if-else statements to be used for the purpose of choosing the correct article (i.e., "a" versus "an") to precede those strings.

```
public class OldMacDonald {

  public static void main(String[] args) {
    preamble("cow"); verseMoo();
    preamble("duck"); verseQuack();
    preamble("pig"); verseOink();
    preamble("dog"); verseWoof();
  }

  static void preamble(String animal)
    { oldMacHadAFarm(); andOnThisFarm(animal); eieio(); }

  static void verseQuack()
    { withA("quack"); verseMoo(); }

  static void verseOink()
    { withA("oink"); verseQuack(); }

  static void oldMacHadAFarm()
    { print("Old MacDonald had a farm, "); eieio(); }

  static void withA(String noise) {
    println("With a " + noise + "-" + noise + " here and a " +
            noise + "-" + noise + " there");
    println("Here a " + noise + ", there a " + noise +
            ", everywhere a " + noise + "-" + noise);
  }

  static void andOnThisFarm(String animal) {
    print("And on this farm he had a " + animal + ", ");
  }
}
```

Figure 8

5 Arrays

Later in the course, when arrays have been introduced, it is possible to employ them in programs that produce cumulative songs, resulting in code that is very easy to modify so as to produce alternative lyrics. This continues the process of teaching students to strive for generality and flexibility in the programs they develop. Figure 9 illustrates this with excerpts from yet another version of the Old MacDonald program. Here, a pair of parallel arrays is used to specify the various animal species' names, and their corresponding utterances, making it easy to add or modify verses.

```
public class OldMacDonald {

  static String[] animals = { "cow", "duck", "pig", "dog" };
  static String[] utterances = { "moo", "quack", "oink", "woof" };

  public static void main(String[] args) {
    for (int i=0; i != animals.length; i++) { verse(i); }
  }

  static void verse(int k) {
    oldMacHadAFarm();
    andOnThisFarm(animals[k]);
    restOfVerse(k);
  }

  static void restOfVerse(int k) {
    for (int j = k; j != -1; j--) {
      withA(utterances[j]);
    }
    oldMacHadAFarm(); println();
  }

  static void andOnThisFarm(String animal) {
    print("And on this farm he had a " + animal + ", "); eieio();
  }
   ...
}
```

Figure 9

6 Objects

Figure 10 presents a further refactoring of the Old MacDonald program that uses a single array of Animal objects, rather than a pair of parallel arrays of strings. The Animal class is straightforward and is thus omitted here. This mature version makes even better use of abstraction and language features. Note the ease and reliability with which verses may be added and modified. It also decouples the `verse()` and `restOfVerse()` methods from any global variables.

7 Conclusion

When teaching CS 1, one must decide in what order to introduce various programming language constructs, and once that order is decided upon a corresponding set of constraints is imposed. Reges & Stepp [1], like every textbook,

```
public class OldMacDonald {

  public static void main(String[] args) {
    Animal[] animals = { new Animal("cow", "moo"),
                         new Animal("duck", "quack"),
                         new Animal("pig", "oink" ),
                         new Animal("dog", "woof" )
                       };
    for (int i=0; i != animals.length; i++)
      { verse(animals,i); }
  }

  static void verse(Animal[] animals, int k) {
    oldMacHadAFarm(); andOnThisFarm(animals[k].getName());
    restOfVerse(animals,k);
  }

  static void restOfVerse(Animal[] animals, int k) {
    for (int j = k; j != -1; j = j-1) {
      withA(animals[j].getUtterance());
    }
    oldMacHadAFarm(); println();
  }
  ...
}
```

Figure 10

follows one such ordering. Debates about the relative merits of one versus another (e.g., objects-early vs. objects-late) have gone on for decades and will undoubtedly continue. The purpose of this brief paper has not been to necessarily advocate for one particular ordering, but rather to share sketches of assignment ideas we have employed that may be used under some of these constraints. If they inspire related ideas and sketches usable under other sets of constraints, then all the better.

References

[1] Stuart Reges and Marty Stepp. *Building Java Programs: A Back To Basics Approach*. Pearson, Hoboken, NJ, 2019.

[2] Wikipedia. Cumulative_song. `https://en.wikipedia.org/wiki/Cumulative_song`.