

Using a Maze Case Study to Teach Object-Oriented Programming and Design Patterns

Chris Nevison

Computer Science Dept.

Colgate University

Hamilton, NY 13346

chris@cs.colgate.edu

Barbara Wells

South Fork High School

10205 SW Pratt & Whitney Rd.

Stuart, FL 34997

bcw100@netzero.com

Abstract

In order to teach object-oriented design and programming in introductory computer science it is imperative to teach objects from the very beginning of the course. The use of interacting objects is motivated by examples with an inherent complexity. We describe a case study based on a maze as an example that provides a complex framework but at the same time admits to simple pieces that students can work with early in an introductory course. This case study can be used throughout the first year not only to introduce basic control structures, but also to introduce a number of design ideas and algorithms.

Keywords: Object-oriented programming, design patterns, algorithms.

1 Introduction

Kristen Nygaard (2001, 2002) and others (Nevison and Wells, 2003) have asserted that object-oriented programming should be taught in the context of complex examples. We present a case study based on finding a way through a maze that demonstrates how this can be done. This case study provides material that students can work with from early in a first computer science course to problems appropriate for a second course on data structures and even beyond.

After reviewing other work in the next section, we will first describe the framework for the maze program. We will then describe how some basic programming exercises on control structures appropriate to an introductory class can be developed within this framework.

This will lead to the introduction of some simple design principles and patterns that can be used to demonstrate how even at the introductory level, it makes sense to discuss basic principles of design.

We will then move on to using the case study to provide examples of recursive algorithms that are non-trivial. One advantage of working within the framework that we

provide is that students get visual feedback and reward as well as the satisfaction of completing an assignment. In many cases the visualization shows errors in a program as well and may lead to an understanding of basic principles.

Finally, we will describe how this case study can be used to demonstrate the use of standard data structures, including two-dimensional arrays, the Java `List` and `Set` classes, stacks, queues, and priority queues. These topics are often addressed in the second computer science course. We will also consider how general graphs might be used to generalize the maze problem and point out how the example could be extended to introduce topics found in upper level algorithms courses, such as optimal path algorithms and branch-and-bound algorithms.

2 Other Work

Kristen Nygaard (2001, 2001) has promoted the teaching of introductory programming using the object-oriented paradigm in the context of complexity. This does not mean that we expect students to grasp the many intricacies of programming at once. Rather, they should be shown how they can work on simple parts of a more complex program and see from the start how it is the interaction of many objects that makes a program work. Nygaard used a busy restaurant as an example. We use the objects and concepts involved with a maze as one example that can be used in this way.

Case studies have long been an important component of teaching computer science. Clancy and Lin (1992) have promoted the use of case studies for teaching introductory computer science for several years, working with procedural programming languages such as Pascal. We are adapting this approach to the context of object-oriented programming and design patterns. In this context, we suggest that the use of case studies from the very beginning of the first course can be a successful strategy.

Many authors have also emphasized the importance of teaching "objects early" when teaching an object-oriented programming paradigm in the first computer science course. See, for example, (Alphonse and Ventura, 2002, Barnes and Kolling, 2003, Bruce, Danyluk and Murtaugh, 2001). We agree that "objects early" is quite natural for the object-oriented paradigm. Well-chosen case studies can provide the complexity to motivate object-oriented programming while also providing a context where concepts can be presented in a reasonably simple setting within the more complex environment.

Many of these same authors suggest that graphics is an important motivating tool for teaching introductory computer science and provide helpful programming environments for this approach (Proulx, Raab, and Rasala, 2002). We do not advocate teaching low-level Java graphics in the first programming course. In the context of the maze case study, we provide the graphical user interface, so that students work on programs in the context of using graphics, but do not do any graphics programming proper. An advanced project might be to create an alternative graphics interface in the context of the structure provided by the case study.

Alphonse and Ventura (2002) advocate the use of design patterns in the first computer science course, an approach that fits well with our use of case studies approach in the first courses. Examples of introducing patterns into the first year were presented at the OOPSLA02 "Killer Examples for Design Patterns and Objects First" workshop, (Alphonse, 2002). Astrachan et al. (1998) also consider design patterns an essential part of the early computer science curriculum, as does Proulx (2000).

Using a maze as the basis of explaining ideas in Computer Science or for programming assignments is not original. A notable example of using a maze program to illustrate design ideas is the book by Gamma, Helm, Johnson, and Vlissides on design patterns (1995). They use a different structure for their maze but find it a rich source for illustrating design patterns. We use our own maze to illustrate some design principles and patterns.

3 The Maze Framework

We provide a framework within which students work. This framework provides several classes that are "black-box" for the students. Some of these classes set up the graphical user interface and need not be seen by the students at all. Others provide functionality used by the students, so that they need to know about the methods that they are expected to use, but need not know about the implementation of the classes providing those methods. Some of these black-box classes may be opened for study later in the course. This is a setup similar to the one used by the Marine Biology Simulation case study developed for the College Board Advanced Placement program in Computer Science (Brady, 2002).

3.1 Walker state

The framework includes two low-level classes for describing the position of a walker in the maze, `Location` and `Direction`. These are borrowed from the College Board *Marine Biology Simulation Case Study* (Brady, 2002).¹ Both these classes are immutable. A third class, `WalkerState`, encapsulates the geometry of the two dimensional grid that forms the maze and the state of a walker in the maze. The `WalkerState` class provides the following constructors and methods:

Constructors

```
WalkerState(Location loc, Direction dir)
    constructs an instance with the given Location
    and Direction
WalkerState(WalkerState state)
    creates a new WalkerState that is a copy of
    state
```

Accessors

```
Location location() returns the location
Direction direction() returns the direction
Direction toRight() returns the direction
                    90 degrees to the right of the
                    state's direction
Direction toLeft() returns the direction 90
                    degrees to the left of the state's
                    direction
Direction toReverse() returns the direction 180
                    degrees from the state's
                    direction
Location neighborTo(Direction dir)
                    returns the location immediately adjacent to the
                    state's location in the given direction
String toString() returns a string representation
                    of this state
```

Modifiers

```
void turnRight() changes direction 90 degrees
                 to the right
void turnLeft() changes direction 90 degrees
                to the left
void turnAround() changes direction 180 degrees
void moveForward() changes the state's location to
                  the adjacent location in the
                  current direction
void moveToward(Direction dir)
                  changes the state's location to the adjacent
                  location in the given direction
```

Since the operations of the class `WalkerState` and the interactions with the other parts of the program can be done completely in terms of `Location` and `Direction` objects, there is no need for students to know about the methods for these two classes. This is a good lesson in working with objects without worrying about the details - taking their interactions as defined by the class given, in this case `WalkerState` and `Maze`.

3.2 The maze

The maze consists of a grid of cells which can either be walls or open. A walker in the maze can only move into open cells, not walls. One open cell is designated as the goal to be reached and an initial state (location and direction) is also part of the maze specification.

We represent the maze with the class `Maze`. It has methods to provide information about the maze.

Accessors

```
int numRows() returns number of rows in maze grid
int numCols() returns number of columns in maze grid
WalkerState start() returns starting state
Location goal() returns location of goal
boolean isWall(Location loc)
                    returns true if location is a wall, otherwise false
```

¹ The MBS case study code is distributed by the College Board under the GNU General Public License, as is the code for this maze case study.

The maze has no modifier methods since it cannot be changed once it has been constructed.

For the first few examples using the framework, the student only needs to know two `Maze` methods: `isWall` and `goal`. Later the methods for getting the dimensions of the maze, `numRows` and `numCols`, and the `start` method can be useful.

3.3 Displaying the maze

The mechanism for displaying the maze is black-box code. The only information that a student or other user needs to know about is the interface `MazeDisplay` that specifies the operations of a display and the interface `MazeDisplayListener` that any maze walker that is interactive must implement.

There are two different maze displays that implement the `MazeDisplay` interface. The first we call the rat's view and it provides a display as if the user were inside the maze. and could see ahead up to three cells and right or left one cell from the current cell or any of those ahead, if the view is not blocked by a wall. Diagram 1 shows two examples of what can be seen, where the bottom center looking up is the initial position, white cells are open, black cells are walls and gray cells are unknown since the view is blocked by a wall.



Diagram 1. Rat's View of Maze

This view is used in the first assignment that asks the student to complete the implementation of an interactive walker that responds to commands input by mouse clicks.

The second display is a bird's eye view, showing the whole maze and indicating the current state of the walker with a cell color (blue) and arrow. This display is introduced when we ask the student to create or modify a random walker, where the program steps through the maze "automatically," rather than in response to mouse clicks, until the goal is reached.

The `MazeDisplay` interface includes these methods:

Modifiers

```
void displayState(WalkerState state)
    displays the view of the maze appropriate for the
    given state.
void eraseState(WalkerState state)
    erases appropriate parts of state display
void markLocation(Location loc)
    marks a location in the maze display
void unmarkLocation(Location loc)
    removes the mark for a location
void addMazeDisplayListener
    (MazeDisplayListener listener)
    adds the listener to this display
void removeMazeDisplayListener
    (MazeDisplayListener listener)
    removes the listener from this display
```

Initially the student only needs to be concerned with the first two methods that display and erase the state. The `addMazeDisplayListener` method is called in the constructor of the interactive walker class that is initially

provided. This and the fact that any interactive walker class must implement the `MazeDisplayListener` interface is a good opportunity for describing the *Observer* design pattern. The `MazeDisplayListener` interface has one method:

```
void onMazeInput(String mazeInput)
    responds to call by MazeDisplay to which this
    listener has been added; the parameter is set to
    "forward," "right," "left," or "reverse" according
    to the button or location clicked.
```

Diagram 2 shows the relationship between the `MazeDisplay` and the `MazeDisplayListener`.

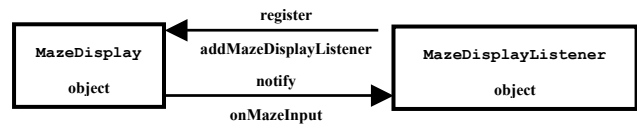


Diagram 2. Listener Relationship

4 Early Assignments

The student would not see the details described above before the first assignment. A programming assignment would be provided with the black-box and other code and a brief explanation of the nature of the listener mechanism. This need not be in any great detail, since the mechanism is already in place.

4.1 An interactive maze walker

The first assignment requires that the student complete the code for one method, `onMazeInput`, for an interactive walker class that is already provided. In fact a working version of the interactive walker called `InteractiveWalkerStart` is provided so that the student can first compile and run the code. This is run from a driver class `MazeDriver` that sets up the rat's view of the maze and constructs the walker with the line `new InteractiveWalkerStart(maze, display);` The student only needs to modify this one line to run a different maze walker. The student assignment is to rename this class `InteractiveWalker` and replace the body of the `onMazeInput` method so that the walker changes state appropriately in response to mouse clicks:

```
public class InteractiveWalkerStart
implements MazeDisplayListener
{
    private WalkerState myState;
    private Maze myMaze;
    private MazeDisplay display;

    public InteractiveWalkerStart(Maze maze,
                                MazeDisplay md)
    {
        myMaze = maze;
        myState = maze.start();
        display = md;
        display.addMazeDisplayListener(this);
        display.displayState(myState);
    }

    public void onMazeInput(String action)
    {
        System.out.println(action);
    }
}
```

- when the user enters "forward" the walker should move forward one location, if that location is not a wall.
- when the user enters "right" the walker should turn to the right without changing location
- when the user enters "left" the walker should turn to the left without changing location
- when the user enters "reverse" the walker should turn around without changing location

In order to complete this assignment, the student must use methods from the `WalkerState` and `Maze` classes.

There are two variations of this assignment that result in slightly different walker behaviors. One might require that on "right" or "left" the walker not only turn but also move into the cell to that side if it is not a wall. A second variation might have the walker move ahead several cells to the next location that has either a wall in front or an open cell to the right or left. Other variations of this sort of walker could also be devised. The second type of walker does not just use conditional statements, it also needs loops, the objective of the next set of programs.

4.2 A random maze walker

The second set of assignments involves loops. The simplest version is a walker that randomly chooses from the actions of the interactive walker described above at each step. It repeats these actions until the goal of the maze is reached. We give this example to the student as a demonstration of the while loop. The code for `RandomWalker` is shown below. This class is not interactive and therefore does not use the listener model. This class calls the methods that display the current state of the walker and the locations that have already been visited. It also includes a time delay, so that the progress of the walker can be viewed.

One can get dizzy running the original `MazeDriver` with this program (although it is fun to let the students do it this way at first). Consequently we also supply `MazeDriver2` that not only includes the bird's eye view of the maze but also allows the user to select the rat's view, bird's eye view, or both.

A first assignment on loops asks the student to modify the `RandomWalker` class so that it does not spin in place so much but instead moves in whatever direction was randomly chosen, if possible. This also gets to the goal of the maze considerably faster than the given version.

Other variations of random walker include walkers that use the `Direction` class constants `NORTH`, `SOUTH`, `EAST`, `WEST` and walkers that keep track of where they have been, using some data structure, and do not repeat locations unless there was no other choice (this involves interesting conditional logic). Another variation would generate a list of possible moves that are not walls, then randomly select the next location from that list.

A challenging assignment that exercises a student's ability to use conditional logic is the right-hand (or left-hand) walker. The right-hand walker moves through the maze in a deterministic fashion as if keeping the right hand on the wall at all times. This algorithm will solve many, but not all mazes.

```
import java.util.Random;
public class RandomWalker
{
    private WalkerState myState;
    private Maze myMaze;
    private MazeDisplay myDisplay;

    public RandomWalker(Maze maze,
                        MazeDisplay md)
    { myMaze = maze;
      myState = maze.start();
      myDisplay = md;
      myDisplay.displayState(myState);
    }

    public void run()
    { Random rand = new Random();
      while(!myState.location().equals(
          myMaze.goal()))
      { myDisplay.eraseState(myState);
        myDisplay.markLocation(
            myState.location());
        int choice = rand.nextInt(4);
        if(choice == 0)
        { if(!myMaze.isWall
            (myState.neighborTo
            (myState.direction())))
          { myState.moveForward();
            }
          }
        else if(choice == 1)
          myState.turnRight();
        else if(choice == 2)
          myState.turnLeft();
        else // choice == 3
          myState.turnAround();

        myDisplay.displayState(myState);
        try
        { Thread.sleep(50);
          }
        catch(InterruptedException ex){}
      }
    }
}
```

4.3 Classroom experience

One of the authors, Wells, has used these assignments in an introductory Java programming class with great success. The interactive walker assignments were used shortly after the students learned about if-else statements. They found the assignment challenging, but not overwhelming and they had the motivation to get the walker to find their way through the maze with the graphical user interface. Many students came up with their own variations on the interactive walker.

In the same class, students were given the assignment of writing the random walker -- they were not given the first version of the random walker that we gave above. They found this to be an exciting assignment and immediately demanded a bird's eye view display. These students found the right-hand walker to be a challenging assignment.

5 Design Issues

Sometime after the interactive and the "automatic" (random and right-hand) walkers have been developed in class and assignments, and after inheritance has been introduced, these can be used to demonstrate some ideas about object-oriented design and design patterns. We do not advocate an emphasis on the use of design patterns in the first programming course, but some discussion of

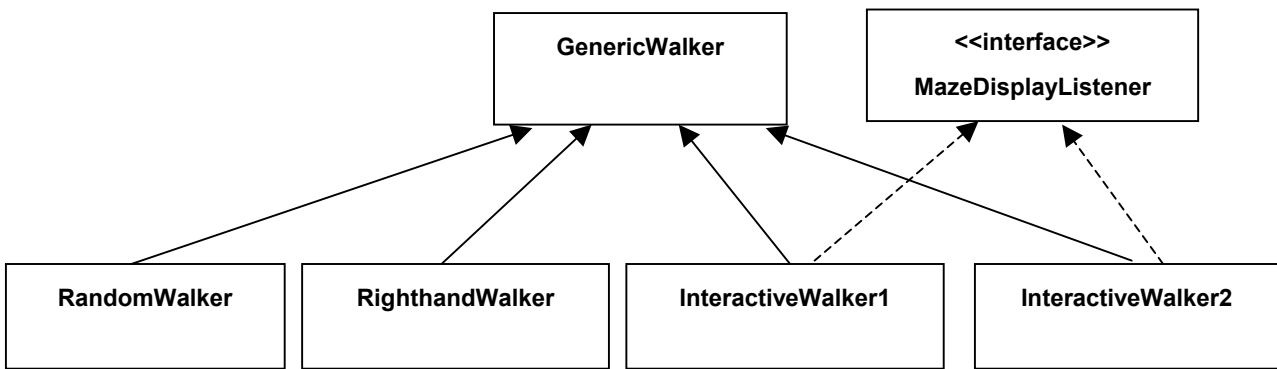


Diagram 3. Inheritance Hierarchy

patterns helps students begin to develop an understanding of design principles.

One of the motivations of object-oriented design and programming is code reuse. An experienced designer will often anticipate the possibilities of code reuse and plan an appropriate inheritance hierarchy. The novice programmer and sometimes the experienced programmer (we speak from experience here) will realize the possibilities after developing several classes without inheritance that have some elements in common. The maze is an excellent example for illustrating this.

This can be developed as a class discussion. The idea is to discuss what the different maze walkers have in common and to "refactor" the code by pulling out the common elements into a base class from which the other maze walker classes can inherit, adding only the additional functionality. This can be effectively done in two stages, the first handling all the walker classes and the second breaking down into groups of similar classes, interactive on the one hand and "automatic" on the other.

5.1 Refactoring: a walker base class

Every maze walker must have the following components in our model:

- a maze
- a maze display
- its state

Each also has a constructor that takes as parameters the maze and the maze display and assigns those to its instance variables as well as setting the initial state using the maze's start method. These can all be wrapped into a base class that we will call `GenericWalker`. In addition, of course, the interactive walkers need to implement the `MazeDisplayListener` interface. This results in an inheritance hierarchy as shown in diagram 3. Here is the code for a generic walker.

This code follows the recommended pattern of keeping all data fields private and providing methods for accessing them in subclasses. Specifying that these accessor methods are protected indicates the intent that they should only be accessed within the inheritance hierarchy.

The code for any other walker is then simplified because it does not need to redo the construction given here and can focus on the code that makes the walker operate,

```

public class GenericWalker
{
    private Maze myMaze;
    private MazeDisplay myDisplay;
    private WalkerState myState;

    public GenericWalker(Maze maze,
                        MazeDisplay display)
    { myMaze = maze;
      myDisplay = display;
      myState = myMaze.start();
      myDisplay.displayState(myState);
    }

    protected Maze maze()
    { return myMaze;
    }

    protected MazeDisplay display()
    { return myDisplay;
    }

    protected WalkerState state()
    { return myState;
    }
}
  
```

```

public abstract class
InteractiveWalkerStart2
    extends GenericWalker
    implements
MazeDisplayListener
{
    public InteractiveWalker2(Maze maze,
                            MazeDisplay md)
    { super(maze, md);
      display().addListener(this);
    }

    public void onMazeInput(String mazeInput)
    { System.out.println(mazeInput);
    }
}
  
```

using the accessor methods to access the instance variables inherited from the `GenericWalker`. Here is the `InteractiveWalkerStart` rewritten as a subclass of `GenericWalker`. Any of the interactive walkers would look just like this, differing only by the code within the `onMazeInput` method. The random walkers or right hand rule walkers would not have the `onMazeInput` method but would instead have the `run` method.

5.2 Refinement: abstract interactive walker

Once students get the idea that inheritance can lead to much simpler code in the subclasses, then we can look

further into this example. A good guide for this example is the *Template* design pattern that suggests a base class implement the aspects of an algorithm that are common to the prospective subclasses, and encapsulates the details of the algorithm in abstract methods that are filled in by the subclasses. This abstract class fits nicely into the inheritance hierarchy.

Looking at the family of interactive walkers that were developed in the exercises, every one has a common control structure in the `onMazeInput` method - they all check the value of the parameter `mazeInput` and take a different action based on its value. They all also display the new state after taking an action. These aspects can be drawn out in discussion with students and result in the abstract class, `InteractiveWalker`, a subclass of `GenericWalker`, which the new versions of concrete interactive walkers will extend, a shown in diagram 4.

The code for the abstract `InteractiveWalker` class is shown below the diagram, followed by the code that implements `InteractiveWalkerStart3`, functionally the same as `InteractiveWalkerStart`, as a subclass. Each concrete interactive walker need only fill in the four abstract methods that specify what the walker is to do in response to each command. Students can reimplement one or more of the other interactive walker variations.

The use of the template design pattern here demonstrates the power of inheritance.

5.3 Refinement: abstract automatic walker

The automatic maze walkers, random and right-hand rule walkers, can be refactored in a similar manner. Creating the class hierarchy is a good assignment after working through the interactive walker example. It could also be a fruitful class discussion, followed by students working up their own versions followed by presentations and additional discussion. The resulting inheritance hierarchy should look much like the one given for the abstract interactive walker and its subclasses, except there is no `MazeDisplayListener` interface. Of course, both of these are parts of the same hierarchy with the `GenericWalker` as the base class.

Some of the points that come up when applying the template design pattern to the automatic walkers are the following:

- they all loop until the goal is found
- within the loop they all apply the same calls to the display methods and also incorporate a timing delay
- some, but not all, use a data structure to keep track of where they have been; this data structure must be initialized before the loop
- some students may have had variations that did something after the loop, like printing a message

These points suggest the outline of the algorithm:

initialize -- specific to each version (not needed by all)

while loop --

 display and timing

 take a step -- specific to each version

wrapup after loop -- specific to each version (not needed by all)

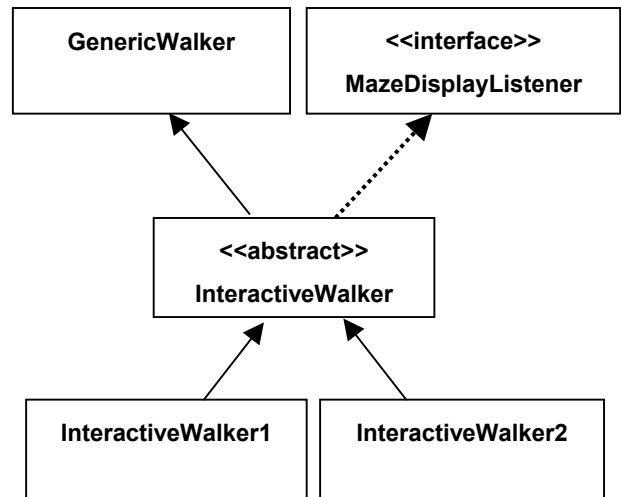


Diagram 4. Refinement

```

public abstract class InteractiveWalker
    extends GenericWalker
    implements MazeDisplayListener
{
    public InteractiveWalker(Maze maze,
        MazeDisplay md)
    { super(maze, md);
      display.addMazeListener(this);
    }

    public void onMazeInput(String mazeInput)
    {
        display().eraseState(state());

        if(mazeInput.equals("forward"))
            forward();
        else if(mazeInput.equals("right"))
            right();
        else if(mazeInput.equals("left"))
            left();
        else if(mazeInput.equals("reverse"))
            reverse();
        else
            throw new IllegalArgumentException();

        display().displayState(state());
    }

    protected abstract forward();
    protected abstract right();
    protected abstract left();
    protected abstract reverse();
}
  
```

```

public class InteractiveWalkerStart3
    extends InteractiveWalker
{
    public InteractiveWalker1(Maze maze,
        MazeDisplay md)
    { super(maze, md);
    }
    protected void forward()
    { System.out.println("forward");
    }
    protected void right()
    { System.out.println("right");
    }
    protected void left()
    { System.out.println("left");
    }
    protected void reverse()
    { System.out.println("reverse");
    }
}
  
```

This outline is the basis for our version of the code for the abstract `AutoWalker`.

In this code we have used a variation that can be useful in application of the template pattern. Since not every `AutoWalker` needs to do an `initialize` method or a `wrapup` method, we have made these empty methods rather than abstract methods. In that way, those versions of an automatic walker that do not need them can simply leave them unimplemented -- the empty version in the template will be called and will do nothing. On the other hand, the `takeStep` method must be implemented by any concrete automatic walker for it to make sense, so it is an abstract class.

When we look at recursive walkers we will see that we can use a similar approach to organizing our classes. We will look at advanced iterative walkers that do depth-first and breadth-first search as classes that extend `AutoWalker`.

```
public abstract class AutoWalker
    extends GenericWalker
{
    public AutoWalker(Maze maze,
                     MazeDisplay md)
    { super(maze, md);
    }

    public void run()
    { initialize();

      while(!state().location().equals(
          maze().goal()))
    { display().eraseState(state());
      display().markLocation(
          state().location());

      takeStep();

      display().displayState(state());
      try
      { Thread.sleep(100);
      }
      catch(InterruptedException ex){}
    }

    wrapup();
}

protected void initialize()
{}

protected abstract void takeStep();
protected wrapup()
{}
}
```

6 Recursion

Finding a path through a maze is a classic problem for recursion, so we will not dwell on it here, other than to make some observations as to how it fits into the framework that we have developed.

6.1 A recursive walker as an assignment

There are several ways to present the problem of recursively solving a maze as an assignment. One is to simply ask the students to develop a recursive maze walker class. In the context of what we have done up to this point, this should be a subclass of the `GenericWalker` class. One could also give students a partial solution to the problem. One way to do this is to

give a solution that has the correct recursive algorithm in terms of recursive calls and checking for walls, but leave out any data structure that keeps track of where you have been, thereby causing an infinite recursion. Give the students the problem of fixing this version. This puts the focus on the issue of the second rule of recursion: you must make *progress* toward the base case (the first rule is that you must have a base case). Depending on what data structures have been covered at this point in your course, the following are reasonable solutions to this problem:

- an `ArrayList` of locations visited
- a Boolean two-dimensional array where true indicates the corresponding position has been visited
- a Java `List` or `Set` of locations that have been visited.

6.2 Variations on recursion

Students can do different variations of recursive solutions as additional exercises. For example, they can do the recursion based on going forward, right, left, reverse from the current state or they can use the constants from the `Direction` class to drive the recursion. These give somewhat different behavior, although there is no reason to prefer one over the other. A "guided" recursion would first move to locations closer to the goal.

6.3 Refinement of the recursion classes

If more than one version of a recursive solution is done by the students or used for demonstration, then one might also look into refactoring the recursive walker classes along the lines of the interactive walker and automatic walker class hierarchy. Factor out the common elements of a recursive solution into an abstract class and each specific recursive walker would be a concrete subclass of that abstract class.

7 The Maze Implementation

An independent line of work within this framework is the implementation of the `Maze` class. For the initial assignments that emphasize control structures, the `Maze` is left as a black-box class. But it can easily be opened and discussed as an example for different data structures.

The `Maze` class is basically a container class that carries the information about the maze. The numbers of rows and columns and the starting state and goal location are trivial instance variables. The interesting data structure is how to store the information about walls for use by the `isWall` method. In the original implementation, we use an `ArrayList` of locations that are walls. The implementation of the `isWall` method is then just a sequential search through the `ArrayList`. We do not use the `ArrayList` method `contains` because we wanted to be able to use this example in the context of using an `ArrayList` with a limited number of methods, accessing elements by index. Of course, it would be easy to substitute a call to the `contains` method, if desired.

In fact, this brings up the issue that the appropriate abstraction is a set. We are using an `ArrayList` to represent a set; why not just use a Java `Set` class? This is,

of course the correct thing to do, as soon as the `Set` hierarchy has been introduced into your course.

Another implementation of the `Maze` would use a Boolean two-dimensional array to represent the walls. This is another exercise that can be used when two-dimensional arrays are covered in your course.

Thus the `Maze` class provides an interesting context for demonstrating or making assignments on some of the standard data structures that we cover in the first year of Computer Science: expandable arrays (`ArrayList`), two-dimensional arrays, sets.

8 Data Structures

The maze framework provides some interesting applications of standard data structures: stacks, queues, and priority queues. By controlling an iterative search using these data structures we develop depth-first, breadth-first, and best-first (heuristic) searches respectively. The problem of recording the moves made during the search and reconstructing the path provides an interesting application of stacks.

Since each of these searches is iterative, the classes implementing them can be developed as subclasses of the `AutoWalker` class discussed above. Then the focus can be on the steps that need to be done, without regard to the graphics display that is taken care of in the `AutoWalker` class.

8.1 Depth-first search

We propose giving a partial solution for this class, leaving the students two parts to complete. First, we give a correct iterative depth-first search algorithm using a stack for the intermediate storage structure, but without any means for preventing the search from repeating itself. The first problem is for the student to select a data structure and modify the methods so that the search does not loop endlessly, but eventually moves to the goal.

The second part of this assignment that the student needs to complete is the recovery of the path from start to goal that is found, with no side paths. We have the main search put every state that is checked onto the stack `visitStack`. The problem is then to write a helper method `extractPath`, that extracts the path itself as a stack, with goal at the bottom and start at the top.

Here is the initial code for the `DepthWalker` class that the student is given. It is a subclass of `AutoWalker`. Recall that any such subclass needs to implement the `takeStep` method and, optionally, the `initialize` and `wrapup` methods. We use an interface for `Stack`, with implementing class `StackAL`. These are taken from materials developed for the College Board AP CS program, available at <http://apcentral.collegeboard.com>

8.2 Breadth-first search

We can modify the depth-first search, once completed, to become a breadth-first search simply by replacing the `searchStack` with a queue, changing calls to `push` and

```
public class DepthWalker0 extends AutoWalker
{
    Stack searchStack;
    Stack visitStack;

    public DepthWalker0(Maze maze,
                        MazeDisplay md)
    { super(maze, md);
    }
    protected void initialize()
    { searchStack = new StackAL();
      visitStack = new StackAL();
      searchStack.push(new
                        WalkerState(state()));
    }
    protected void takeStep()
    { if(searchStack.isEmpty())
      { throw new IllegalStateException(
                "no solution");
      }
      setState((WalkerState)searchStack.pop());
      visitStack.push(state());
      if(!state().location().
          equals(maze().goal()))
      { Location loc =
        state().neighborTo(Direction.NORTH);
        if(!maze().isWall(loc))
        { searchStack.push(new
          WalkerState(loc, Direction.NORTH));
        }
        loc =
          state().neighborTo(Direction.EAST);
        if(!maze().isWall(loc))
        { searchStack.push(new
          WalkerState(loc, Direction.EAST));
        }
        loc =
          state().neighborTo(Direction.SOUTH);
        if(!maze().isWall(loc))
        { searchStack.push(new
          WalkerState(loc, Direction.SOUTH));
        }
        loc =
          state().neighborTo(Direction.WEST);
        if(!maze().isWall(loc))
        { searchStack.push(new
          WalkerState(loc, Direction.WEST));
        }
      }
    }

    protected void wrapup()
    { displayPath(extractPath());
    }

    private Stack extractPath()
    { Stack path = new StackAL();
      // code to extract path
      // from visitStack here
      return path;
    }

    private void displayPath(Stack path)
    { while(!path.isEmpty())
      { WalkerState current =
        (WalkerState)path.pop();
        display().displayState(current);
      }
    }
}
```

`pop` to enqueue and dequeue. (We provide a `Queue` interface and implementation from the AP CS program.) Note that the `visitStack` stack should not be changed. These two walkers demonstrate two standard search strategies used for finding a goal and even for an optimal solution to a problem, such as a shortest path.

8.3 Best-first search

A "best-first," "guided," or "heuristic" search is a search based on a criterion for determining that some steps in the search are more promising than others. In the case of the maze problem, we take the location with the shortest distance to the goal as the most likely to lead to the goal. In order to carry out a best-first search we replace the stack or queue used previously with a priority queue with the distance to the goal from a given state as the priority. For the priority queue we use the `PriorityQueue` interface used for the AP CS program and provide a min-heap implementation. This requires a class that contains an item (state) and its priority (distance to maze goal). Otherwise we simply replace the `searchStack` with a priority queue, using the methods `add` and `removeMin` in place of `push` and `pop`.

This sequence of assignments gives examples of the use of these three standard data structures within an interesting context. It also is the beginning of topics that would be covered in a more advanced course on algorithms, such as finding the shortest path or developing a branch-and-bound algorithm.

8.4 Classroom experience

One of the authors, Nevison, has used both the recursion assignment and the depth-first, breadth-first, and best-first search assignments in a second computer science course, with the focus on data structures. These assignments worked quite well. The students liked working with a problem with a visual component, and the visualization provided feedback on the how their program was working. For example, seeing a recursive or iterative solution that continuously moves between two or a few states demonstrates the problem of not keeping track of where one has been previously to force progress toward the base case (the maze goal).

Another example of the effectiveness of visual feedback is the assignment to implement the `extractPath` method for the depth-first and other iterative searches. Students will often get a "solution" that compiles and runs, but when run marks a "path" from start to goal with many side branches. This visual feedback shows that the program has problems and may provide clues as to what the problem is.

Students find these assignments to be interesting and get engaged with finding good solutions. Some students are ready to take the problem a step further with an extra credit shortest-path solution for the maze.

9 Summary

We have presented an example demonstrating how one can use case studies to teach introductory computer science from the early in the first course through the material typically covered in the first year of Computer Science. We have had success with this approach in our own classes.

The advantages we see to using case studies include:

- Complexity in a controlled situation
- Introduction of objects early
- Incremental introduction of topics
- Provide a familiar context for students throughout a course
- Provide a realistic context for the application of concepts
- Provide a context for introducing design patterns

10 References

Alphonse, Carl, and Phil Ventura, 2002. "Object-Orientation in CS1-CS2 by Design," Proceedings of the 7th Annual Conference on Innovation and Technology in computer Science Education (ITiCSE 2002), 70-74.

Astrachan, Owen, Geoffrey Berry, Landon Cox, Garrett Mitchener, 1998. "Design Patterns: An Essential Component of CS Curricula," ACM SIGCSE Bulletin v 30 n 1, 153-160.

Barnes, David, and Michael Kolling, 2003. *Objects First with Java*, Prentice-Hall: New York.

Brady, Alyce, 2002. *The Marine Biology Simulation Case Study*, The College Board: New York. Available from <http://apcentral.collegeboard.com>. Accessed Nov., 2003.

Bruce, Kim, Andrea Danyluk, Thomas Murtagh, 2001. "A Library to Support a Graphics-Based Objects-First Approach to CS 1," ACM SIGCSE Bulletin v33 n1, 6-10.

Christensen, Henrik, and Michael Caspersen, 2002. "Frameworks in CS1 - a Different Way of Introducing Event-driven Programming," Proceedings of the 7th Annual Conference on Innovation and Technology in computer Science Education (ITiCSE 2002), 75-79.

Clancy, Michael, and Marcia Lin, 1992. "Case Studies in the Classroom," ACM SIGCSE Bulletin v 24 n 1, 220-224.

Gamma, Helm, Johnson, and Vlissides, 1994. *Design Patterns*, Addison-Wesley: Boston.

Nevison, Christopher, and Barbara Wells, 2003. "Teaching Objects Early and Design Patterns in Java Using Case Studies," Proceedings of the 8th Annual Conference on Innovation and Technology in computer Science Education (ITiCSE 2003).

Nygaard, Kristen, 2001. Invited Talk OOPSLA'01, Educators' Symposium

Nygaard, Kristen, 2002. "COOL (Comprehensive Object-Oriented Learning," (keynote address) Proceedings of the 7th Annual Conference on Innovation and Technology in computer Science Education (ITiCSE 2002), 218.

Proulx, Viera, 2000. "Programing Patterns and Design Patterns in the Introductory Computer Science Course," ACM SIGCSE Bulletin v 32 n 1, 80-84.

Proulx, Viera, Jeff Raab, Richard Rasala, 2002. "Objects from the Beginning," Proceedings of the 7th Annual Conference on Innovation and Technology in computer Science Education (ITiCSE 2002), 70-74.