The SIGCSE 2001 Maze Demonstration Program

Richard Rasala, Jeff Raab, Viera K. Proulx
College of Computer Science
Northeastern University
Boston MA 02115
{rasala,jmr,vkp}@ccs.neu.edu

Abstract

This article will describe the SIGCSE 2001 Maze Demo program that may be used as a CS2 laboratory exercise on traversal algorithms. The article will also describe the object-oriented design of the program and the Java Power Tools that were used to enable rapid development of its graphical user interface. Finally, the quality of the program and the speed of its development shows that it is now practical to teach freshmen using full graphical user interfaces rather than interfaces that use the console or a small restricted set of interface widgets.

1. Introduction

At the opening reception for the SIGCSE 2001 conference, the authors were speaking with Michael Goldweber of Xavier University about the Java Power Tools (JPT) toolkit. These tools enable very rapid development of graphical user interfaces in Java. Prof. Goldweber decided to propose a challenge. He suggested that we develop an animated maze traversal program with a full graphical user interface before the end of the SIGCSE conference. He specified that the traversal algorithm should intentionally be simple so that the program could be used as the basis for a student laboratory in which the given algorithm is replaced by a more powerful one designed by the student. The authors accepted the challenge.

The authors met for breakfast the next morning and discussed the design of the classes for about an hour. Two of the authors then worked throughout the morning to build the maze program and by lunchtime a correct working version was finished. The programming time was two and a half hours.

This work was partially supported by NSF grant DUE-9950829.

The next day the program was demonstrated to Prof. Goldweber and he confirmed that the maze program more than fulfilled his requirements and expectations.

In this article, we will describe a refined version of the SIGCSE Maze Demo program. After the conference, we carefully examined and refactored the code [1] to make it as clean as possible. We added a second algorithm so that a student could learn how to select an algorithm in the user interface. Finally, we made it possible to perform several maze animations in parallel.

We believe that the SIGCSE Maze Demo program will be important to CS faculty for several pedagogical reasons:

> As a CS2 laboratory exercise on traversal algorithms, the SIGCSE Maze Demo may be used immediately without modifications.
>
> The SIGCSE Maze Demo shows that with the proper toolkits it is quite feasible to build a laboratory that has a full graphical user interface in a very short period of time and that it is therefore no longer necessary to teach freshmen using more primitive interfaces.
>
> The SIGCSE Maze Demo illustrates in a simple manner how to execute algorithms in separate threads.
>
> The SIGCSE Maze Demo web site comes with an extensive tutorial on the design of the program, the class structure, and the use of the Java Power Tools to create the GUI extremely rapidly. This should be of interest both to those who want to learn more about the Java Power Tools and to those who want a quality case-study in object-oriented design.

2. Overview of the Maze Demo Classes

The Maze Demo program is designed using five classes in addition to the classes provided in the Java Power Tools or JPT. Let us describe these five classes and their responsibilities.

MazeApplication:

> The **MazeApplication** class launches the program, creates the frame for the maze and the controls, defines the control buttons, and starts and stops the traversal algorithms.

Maze:

The **Maze** class creates the maze in a panel using a 2-dimensional array of **MazeCell** objects for the contents.

MazeCell:

The **MazeCell** class creates an individual cell in the maze that can respond to mouse clicks to set its state in coordination with the **Maze** object.

Bug:

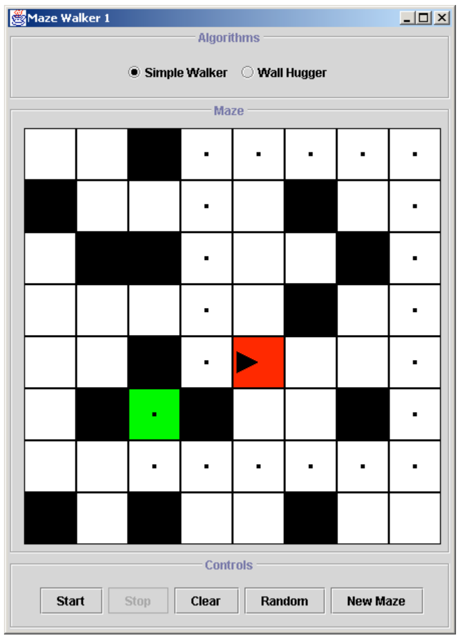
The **Bug** class defines the animated creature or "bug" that traverses the maze.

MazeAlgorithm:

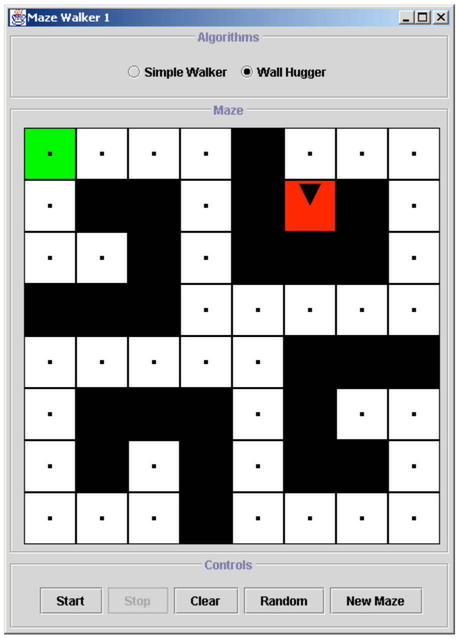
The abstract **MazeAlgorithm** class organizes the algorithmic code for maze traversal. Its **cycle** method must be defined in a derived class to implement a specific algorithm. In this demo program, the algorithms used are so simple that they are defined via static factory methods within the **MazeAlgorithm** class.

3. The Maze Demo User Interface

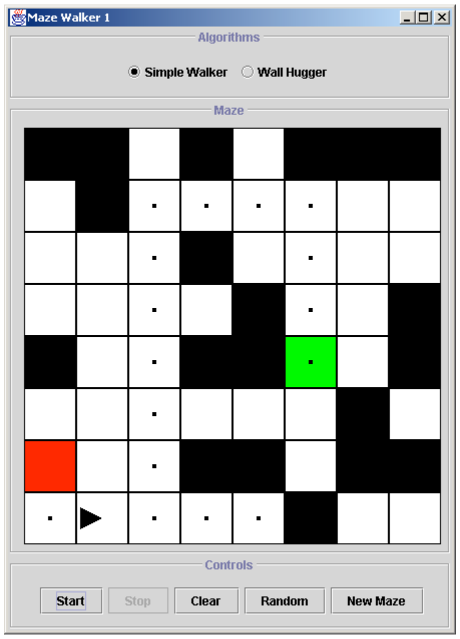
Four screen snapshots of the Maze Demo program are shown below:



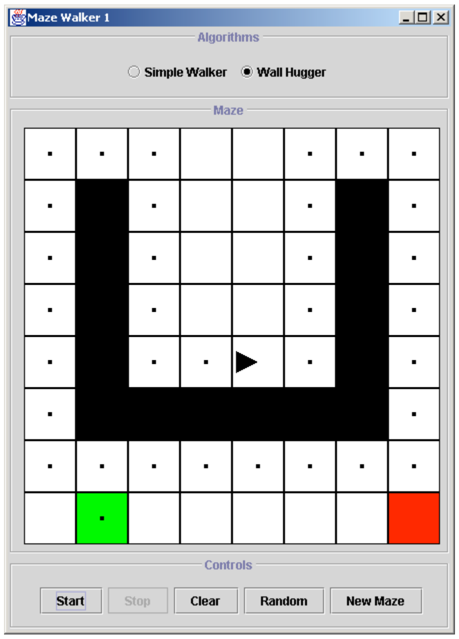
[✓ Simple Walker] Wall Hugger



Simple Walker [✓ Wall Hugger]



[✓ Simple Walker] Wall Hugger



Simple Walker [✓ Wall Hugger]

The screen snapshots illustrate the two *inadequate* algorithms that are provided as the starting point for the student laboratory exercise. The *SimpleWalker* algorithm goes straight ahead until it hits a wall where it turns left. The *WallHugger* algorithm seeks a wall and then follows that wall to the left indefinitely. For each algorithm, an instance is shown in which the algorithm is successful and another instance in which it fails.

The graphical interface is designed as follows. There are three major panels each with titles: *Algorithms*, *Maze*, and *Controls*. In the Algorithms panel, there are radio buttons that permit the user to select the traversal algorithm. In the Maze panel, there is an array of maze cell objects arranged in a grid layout. In the Controls panel, there are five *action buttons* that permit the user to *Start* or *Stop* the algorithm, *Clear* the maze, create a *Random* maze, or bring up a *New Maze* window with up to 20x20 cells.

The maze cells have four colors that encode their current *state*: *free* (white), *wall* (black), *start* (green), or *goal* (red). The user can edit the cell state using the mouse. A simple click will toggle a free or wall state to its opposite. A control-click will turn a cell into the start state and a shift-click will turn it into the goal state. The program maintains the invariant that there is always precisely one start state and one goal state. It would be possible to modify the program to allow multiple goal states if that possibility is desired.

When the window opens the Stop button is disabled since no algorithm is running. When the Start button is clicked, the algorithm begins, the Stop button is enabled, and the Start, Clear, Random, and algorithm radio buttons are disabled. It is always possible to create a New Maze window since that operation has no effect on a running algorithm. Therefore, the New Maze button is always enabled.

A maze algorithm is always executed in a separate thread since in general it is a very bad idea to execute anything that requires substantial time in the thread that listens to the user interface. The problem with executing a time intensive algorithm in the user interface thread is that the user interface appears to freeze and will not respond to the user. In particular, in this program, the Stop button would not respond so that there would be no way to stop a runaway algorithm short of aborting the program.

Since a maze algorithm is executed in a separate thread, it is possible for the user to edit the maze *as an algorithm is running* by adding or removing walls or even by changing the goal state. In this manner, the user can *help* a weak algorithm by changing the maze in such a way that the algorithm can succeed in finding the goal state. Of course, for a robust algorithm, the maze state should be constant and this option may be specified in the algorithm constructor.

4. Building the Maze Demo User Interface

The key to the rapid development of the Maze Demo user interface is the use of the Java Power Tools that we have created over a period of two years [3,6,7]. The problem with pure Java for building user interfaces is that you are presented with a collection of widgets that require substantial effort to be coaxed into a working interface. It is comparable to being given an expensive automobile with the catch that you must assemble it from thousands of parts. The fact that the same sequences of code occur over and over in Java texts is a sure sign that essential encapsulations have not been made in pure Java.

The fundamental goal of the Java Power Tools is to make the creation of graphical user interfaces extremely rapid. The correspondence between an idea and its expression in code should if possible be 1-to-1: one idea, one line. If several lines of code are required they should be required for conceptual reasons not because some boilerplate code must be added. In the case of interface elements, we see precisely four such conceptual steps:

construct the element

position the element in the interface

send user data from the element to the model

update the element using data sent from the model

The ideal of the Java Power Tools is that these four steps should take four lines of code.

In [6], we stated: "The fundamental design principles of the JPT are that *the elements of a graphical user interface should be able to be combined recursively as nested views* and that *the communication between these views and the internal data models should be as automatic as possible*." We achieve these goals by systematic encapsulation of data, of interface elements, and of the methods that enable communication in the system. Our Java programming style is a subtle combination of object-oriented principles and the functional style of LISP and Scheme that encourages the use of recursive nesting.

Let us now explain how these general ideas play out in the creation of the Maze Demo user interface.

First of all, each of the main panels in the interface is enclosed with titled border. We have a *decorator* class **Display** that can wrap another panel with a title and/or an annotation. Hence we can add titles in the same step that we use to add the panels to the main window.

The Algorithms panel consists of 2 radio buttons. To build this panel, we create a **String** array with the 2 button strings "Simple Walker" and "Wall Hugger". We then pass this array to the constructor of the JPT **OptionsView** class that knows how to build a radio button panel together with the methods needed to extract the current user selection.

The Controls panel consists of 5 *action buttons*. From the conceptual viewpoint, the only important information for a button is its name and its action, that is, what will be done when the button is clicked. All of the usual extras that you see repeated in pure Java code concerning "add" methods and "listeners" is *implementation detail that should not be seen*. Therefore, to create the Controls panel using JPT, we first create 5 action objects that encapsulate both a button name and its action using the JPT class **SimpleAction**. For example, to define the start action, we use the following pattern:

```
protected Action start =
    new SimpleAction("Start") {
        public void perform() { start(); }
    };
```

Notice that the action **start** (which is an object) defers the work of its standard **perform()** method to the method **start()** of the **MazeApplication** class. This idiom is the standard way in Java to implement the design pattern:

Encapsulate action as object.

The idiom converts methods which are not first class in Java to objects which are first class and may be stored and passed around. The idiom also permits the details of the action to be deferred to the methods section of the class.

Once all 5 actions objects have been created, we bundle them into an array of actions. We pass this array to the JPT **ActionsPanel** class that knows how to build a panel with action buttons, button listeners, and all of the implementation detail. In addition to the 5 action definitions, this requires 2 lines of code.

The Maze panel uses the **Maze** constructor to build an nxn grid of **MazeCell** objects where 2 <= n <= 20. Each maze cell must refresh itself as needed, change its state when clicked by the mouse, and maintain its communication with its maze. The first two requirements can be easily obtained by deriving the **MazeCell** class from the JPT **BufferedPanel** class. In a **BufferedPanel**, graphics commands paint to a hidden **BufferedImage** object. This buffer is used to automate the graphics refresh process. In addition, a **BufferedPanel** comes with a mouse listener that uses the JPT **MouseActionAdapter** class. To make the panel responsive to the mouse, it is sufficient to supply the *actions* to perform corresponding to various mouse events. In this case, the **MazeCell** mouse behavior can be defined simply by defining the change-of-state actions that must be performed when the mouse is clicked on a cell.

This completes the definition of the Maze Demo graphical user interface. Hopefully, it is now clear how the entire original program was finished in two and a half hours. The interface was completed and functioning in an hour and the rest of the time was spent on the basic algorithmics. The same development speed would have been impossible using only pure Java.

5. The Maze Demo Algorithmics

The most interesting aspect of algorithmics of the Maze Demo is how the **MazeAlgorithm** abstract class is defined and how the **MazeApplication** class runs an algorithm.

The crucial features of the **MazeAlgorithm** definition are:

```
public abstract class MazeAlgorithm
    implements Runnable, JPTConstants {
    // some details omitted ...
    public void run() {
        initMazeAlgorithm();
        while(isRunning) {
            cycle();
            if (atGoal()) break;
            JPTUtilities.
                pauseThread(PAUSE_TIME);
        }
    }
    public abstract void cycle();
    public void initMazeAlgorithm() {
        isRunning = true;
        maze.setEnabled(enableChange);
    }
    public void stopMazeAlgorithm() {
        isRunning = false;
        maze.setEnabled(true);
    }
    public boolean atGoal() {
        return bug.atGoal();
    }
}
```

The **MazeAlgorithm** class is prepared to run in a separate thread by declaring that it implements the **Runnable** interface and by defining its **run()** method. The **run()** method definition follows the *template method* pattern of [2], that is, it uses three methods **initMazeAlgorithm()**, **cycle()**, and **atGoal()** to define its behavior. The **cycle()** method is abstract and so *must* be defined in a class that extends **MazeAlgorithm**. The other methods may optionally be redefined to provide additional behavior.

For example, the SimpleWalker algorithm has an almost trivial **cycle()** method:

```
public void cycle() {
    if (bug.freeToStep()) bug.step();
    else bug.turn(bug.left());
}
```

The WallHugger is more complex since it must find the wall and then travel along it. In an even more robust algorithm, the method **initMazeAlgorithm()** would presumably perform a graph traversal algorithm to determine the "best" path and then create a sequence of steps and turns that would move the bug from the start to the goal cell one move at a time in the **cycle()** method.

It is also of interest to see how a maze algorithm is run in the **MazeApplication** class. This is accomplished by the **start()** method that is associated with the Start button:

```
protected void start() {
   start.setEnabled(false);
   clear.setEnabled(false);
   random.setEnabled(false);
   algorithmOptions.setEnabled(false);

   maze.refresh();

   strategy = createMazeAlgorithm();

   stop.setEnabled(true);

   Thread t = new Thread(strategy) {
      public void run() {
         super.run();
         MazeApplication.this.stop();
      }
   };

   t.setDaemon(true);
   t.start();
}
```

This method disables the buttons that should be inactive during the execution of the algorithm, refreshes the maze to remove the bug tracks from any earlier traversal, calls the method **createMazeAlgorithm()** to create a new instance of the algorithm currently selected by the user, enables the Stop button, and starts a new thread. The thread runs the algorithmic strategy and ensures that when the algorithm is done then the **stop()** method will be called to reset the user interface to its original state. Notice that the thread is made into a *daemon* so that if the application halts then the thread will halt as well.

From a pedagogical viewpoint, this thread code is simple enough that a freshman student can learn the pattern without needing a full discussion of processes such as would occur in an operating systems course. It is good to introduce threads in this simple manner rather than in a fashion that immediately emphasizes complications.

6. Conclusions

We have presented a Maze Demo program with an elegant graphical user interface that is ready to be used as a traversal-algorithms exercise in CS2 courses. One goal of this article is to make this laboratory known to CS faculty.

Our goals in writing this article are much more general however. In [4,5], we have argued that toolkits are fundamental for CS education. Roberts [8] made the same argument. The point is that the pedagogical goals of the CS1 and CS2 courses should be:

> To introduce the fundamental principles of computer science: *information*, *algorithmics*, *encapsulation*, *recursion*, *interaction*, *language*, and *formalism*.
>
> To introduce modern programming methods and practices including graphics and graphical user interfaces whenever possible.

The point of freshman CS education is *not* to teach some particular programming language (C, C++, Java) in its "pure" form so students will be "prepared for industry". The best preparation we can give students is to teach them how to program in an elegant, thoughtful manner that makes them comfortable with both creating and using abstractions and encapsulations. As educators, our task is to educate students about *what can be* and not just about what is commonly done in industry today.

When we made the decision to switch from C++ to Java in the freshman year at our university, we were determined to use graphical user interfaces. It is too hard to do that using pure Java. We have invented the Java Power Tools to solve this problem. We believe that the Maze Demo program shows that we have been successful.

7. Online Materials

The Java Power Tools and related sample files may be found at:

http://www.ccs.neu.edu/teaching/EdGroup/JPT/

The SIGCSE Maze tutorial, source code, and application may be found at:

http://www.ccs.neu.edu/teaching/EdGroup/JPT/Maze/

References

[1] Fowler, Martin, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Reading MA, 1999.

[2] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[3] Raab, Jeff, Rasala, Richard, and Proulx, Viera K., *Pedagogical Power Tools for Teaching Java*, SIGCSE Bulletin, 32(3), 2000, 156-159.

[4] Rasala, Richard, *Design Issues in Computer Science Education*, SIGCSE Bulletin, 29(4), 1997, 4-7.

[5] Rasala, Richard, *Toolkits in First Year Computer Science: A Pedagogical Imperative*, SIGCSE Bulletin, 32(1), 2000, 185-191.

[6] Rasala, Richard, Raab, Jeff, and Proulx, Viera K., *Java Power Tools: Model Software for Teaching Object-Oriented Design*, SIGCSE Bulletin, 33(1), 2001, 297-301.

[7] Rasala, Richard, *Exploring Recursion in Hilbert Curves*, SIGCSE Bulletin, 33(1), 2001, 194.

[8] Roberts, Eric, *Using C in CS1: Evaluating the Stanford Experience*, SIGCSE Bulletin, 25(1), 1993, 117-121.