CollegeBoard
Advanced Placement Program

AP® Computer Science

2007–2008
Professional Development
Workshop Materials

Special Focus:
GridWorld Case Study

The College Board: Connecting Students to College Success

The College Board is a not-for-profit membership association whose mission is to connect students to college success and opportunity. Founded in 1900, the association is composed of more than 5,000 schools, colleges, universities, and other educational organizations. Each year, the College Board serves seven million students and their parents, 23,000 high schools, and 3,500 colleges through major programs and services in college admissions, guidance, assessment, financial aid, enrollment, and teaching and learning. Among its best-known programs are the SAT®, the PSAT/NMSQT®, and the Advanced Placement Program® (AP®). The College Board is committed to the principles of excellence and equity, and that commitment is embodied in all of its programs, services, activities, and concerns.

For further information, visit www.collegeboard.com.

AP® Computer Science
Special Focus Materials for 2007–08
GridWorld Case Study

Table of Contents

Introduction

Debbie Carter, editor
Lancaster Country Day School
Lancaster, Pennsylvania

Starting with the 2007–08 academic year, a new case study based on a grid—called "GridWorld"—will be a required part of the AP® Computer Science A and AB curricula. GridWorld provides a graphical environment in which students can experiment with different types of objects, observing the ways in which programming changes affect the objects' behavior. Questions related to this case study will first appear on the 2008 AP Computer Science Exams. (*AP Computer Science Newsletter*, No. 6, Nov. 15, 2006).

We have commissioned a group of articles and instructional materials to help you prepare to use the GridWorld case study to teach computer science.

Those of us who are familiar with the Marine Biology Simulation (MBS), the previous AP Computer Science case study, will find some familiar features in GridWorld. As Cay Horstmann explains in his article, "The Design of the GridWorld Case Study," GridWorld's design grew from what we learned from our experiences with the MBS in the classroom.

We wanted more flexibility in the types of objects that could populate the world, as well as in the ways that they could be displayed. An object in the world can

- be displayed using an external graphic or via a graphics display class;
- examine a list of other objects in the world and respond to one or more of them;
- behave independently of other objects (or not at all).

In "Integrating GridWorld," Jill Kaminski gives us both an overview and some detailed practical advice about how we might effectively weave the case study throughout our AP Computer Science courses.

Four additional authors, all computer science educators, have contributed hands-on instructional materials, with complete solutions, descriptions of teaching strategies, and suggestions for differentiation for students of varying needs. I would like to thank the contributors for their hard work and cooperation, as well as their continuing commitment to the AP Computer Science community.

We know that you'll find many treasures in the next several pages, ready to be used to excite your AP Computer Science students!

References

College Board. *AP Computer Science Course Description, May 2007, May 2008.*

College Board. *AP Computer Science A and AB Newsletter.* No. 6 Nov. 15, 2006. http://www.collegeboard.com/email/ap_compsci_n8490.html

The Design of the GridWorld Case Study

Cay S. Horstmann
San Jose State University
San Jose, California

Abstract

In this article, I describe the rationale behind decisions that were made in the design of the GridWorld case study. Knowing about these decisions can be useful to address student questions, or simply to satisfy your own curiosity. I also give information about the inner workings of the GUI that is helpful for designing your own worlds.

A Brief History of GridWorld

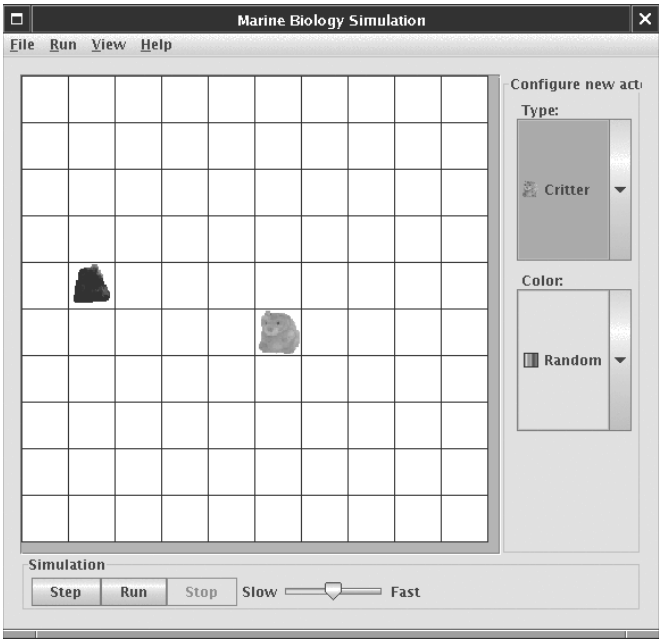
In 2004, the AP Computer Science Development Committee solicited proposals for a new case study. The committee received a number of interesting suggestions, but none of them had the flexibility of the Marine Biology Simulation (MBS) case study for producing exam questions. Instead, we decided to make the MBS code more generic, to easily handle creatures other than fish, and to remove inessential classes. The redesign was governed by four themes:

- Continuity. Teachers who are familiar with MBS should feel right at home in GridWorld.
- Simplicity. Students who see GridWorld for the first time should not be overwhelmed.
- Testability. The framework should give rise to many kinds of exam questions.
- Extensibility. Enthusiasts should be able to design grid-based games, mazes, puzzles, simulations, and so on, without GUI programming.

The first prototype of GridWorld appeared in March 2005. It consisted of the MBS code, with one important change—the ability to add GIF images instead of having to use Java graphics for drawing occupants. It turns out that this ability was already present in the MBS GUI code, but it was well hidden. Figure 1 shows the very first GridWorld screen capture; note the cuddly critter and the frame title.

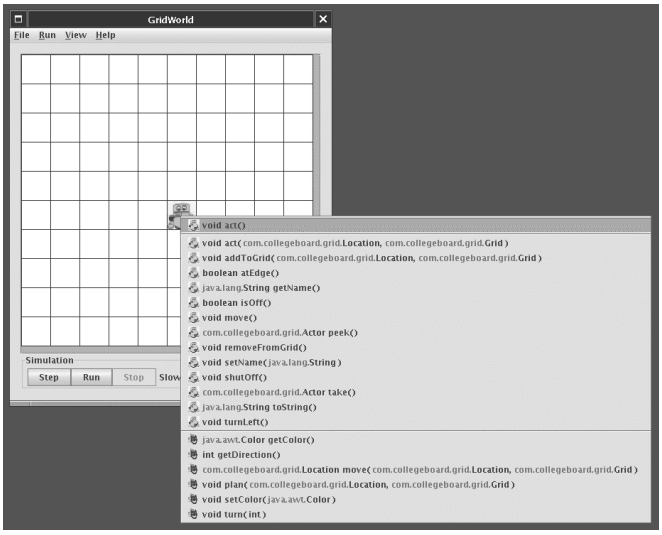
The next months were the "Cambrian explosion" of GridWorld, with numerous exotic life forms appearing rapidly: rock hounds that search for rocks by recursively asking their neighbors, robots that drop flowers, potato-shaped cells, flocks of "boids," and aliens. They became extinct, to be replaced with the bugs and critters that we know today.

Figure 1—First GridWorld screen capture.



The other major innovation was the direct manipulation interface that allows students to invoke constructors and methods. I had always admired this capability in *BlueJ* and experimented with controlling grid occupants from the *BlueJ* workbench. That proved too cumbersome, and I ended up implementing direct manipulation inside GridWorld. Figure 2 shows another historic first—the first screen capture of the method menu. Note the method names and parameters!

Figure 2—First screen capture of the method menu.



Chris Nevison and Barbara Cloud Wells joined in September 2005 to produce the narrative. They invented the crab and chameleon critters that show off the template method pattern. Shortly after the narrative was released, we were delighted to receive a beautiful set of icons from Chris Renard, a student at the School for the Talented and Gifted of the Dallas Independent School District.

The design of GridWorld has now been finalized, and the case study is ready for use in the 2007–08 school year. In this article, I will discuss some of the design decisions behind GridWorld and give tips for advanced users.

Design Decisions

In this section, I discuss some of the design decisions that were made during the development of GridWorld. You may disagree with some decisions; some of your colleagues have done so rather vocally. You may want to discuss the pros and cons of some decisions with your class.

Actors Store Their Location

The MBS case study used an `Environment` that holds `Locatable` objects. This proved to be restrictive when writing exam questions. We decided that we wanted a grid that can hold objects of *any* type. That means that the grid cannot ensure that the grid location and the occupant location are synchronized—the grid doesn't know whether its occupants have locations.

We attempted to solve this problem in an ingenious way, by passing the current location to the `act` method.

```
public void act(Location loc, Grid<E> gr)
```

The actors didn't have to remember their location; the world reminded them. Unfortunately, the location and grid ended up being passed into many other helper methods, leading to tedious code. In the end, we settled for actors that store their location, and supplied the `putSelfInGrid` and `removeSelfFromGrid` methods for synchronizing the actor and grid locations. As your students will undoubtedly find out, the `put` and `remove` methods of the `Grid` interface do not work for actors. The actors will throw exceptions when they find that their locations are not properly set.

The Out-Of-Subset `instanceof` Operator

The `instanceof` operator is not in the AP Java subset, but it is used to sense the type of actors that are being processed. For example, a bug decides that it can move to a location by testing

```
(neighbor == null) || (neighbor instanceof Flower)
```

Going beyond the AP Java subset for the case study has precedent. The MBS case study used the `protected` modifier that is not in the subset. Nevertheless, it is not an ideal situation, and we explored alternatives. Unfortunately, all of the alternatives required students to write code that was more cumbersome than the `instanceof` test. There is no point in making students learn cumbersome custom code when they could instead learn a Java feature.

Note that we use the `instanceof` test in its most benign form, without the dreaded cast. That is, we do not have code of the form

```
if (myActor instanceof Critter)
{
  Critter myCritter = (Critter) myActor; //we don't do in
                                         //GridWorld
  . . .
}
```

Will other GridWorld users exhibit the same good taste and restraint? Only time will tell.

Grid Types are Generic

In the AB course, students look at the `Grid` interface and the `AbstractGrid`, `BoundedGrid`, and `UnboundedGrid` classes. They will see implementations of generic types, even though the implementation of generic types is not currently in the AP Java subset. In the A course, of course, there is no problem. Students use `Grid<E>` much like `ArrayList<E>`. An old-style `Grid` that stores `Object` references requires unsightly casts. We felt that this advantage outweighed the slight complexity of type variables in the AB course.

No `Direction` class

The MBS case study had a `Direction` class for compass directions such as `Direction.NORTH`. The class had few interesting methods, and we eliminated it to minimize the number of classes that students see when they encounter GridWorld for the first time. Now the constants are placed in the `Location` class, which is admittedly not optimal.

Eight neighbors

A location in the grid has eight neighbors. In contrast, the MBS environment could be configured so that a location had four or eight neighbors, and some people even produced triangular or hexagonal grids. This led to cumbersome exam questions since we always had to be explicit about the number of neighbors. We decided that hexagonal grids were cute but not useful enough to pay for the added complexity.

No Random Singleton

Naive users often try to generate random numbers like this:

```
int r = new Random().nextInt(n); // DON'T DO THAT!
```

However, constructing many random number generators will lead to poorly distributed random numbers.

The MBS case study had its own provider for a singleton random generator. That is the right design if you are serious about controlling random number generation. In the context of the MBS narrative, it made sense to have repeatable sequences of pseudo-random numbers. However, the subtleties of random number generation are clearly not a central part of the AP Computer Science curriculum. In GridWorld, we simply generate random numbers with the call

```
int r = (int) (n * Math.random());
```

You Can't Save the World

The MBS case study had a mechanism for saving and reloading simulation data, in files with contents such as

```
bounded 7 5
Fish 3 2 North
```

In GridWorld, that's a lot harder since grid occupants can be arbitrary objects. I solved the problem by using the XML file format for a specialized Java feature called "long-term beans persistence." However, that led to a harder problem. A data file can reference an exotic critter whose implementation is not on the class path. Not to be deterred, I saved a world as a zip file that contains all classes and the data. You could mail your favorite world to someone else, and the recipient's GridWorld program would read the classes directly from that file, using a nifty class loader.

This was, if I may say so, a technical tour de force. Unfortunately, but perhaps not surprisingly in hindsight, users did not send world files to each other. They were just confused that there was one set of classes in their project directory and a separate set in the world files. I ended up removing the feature. My heart still bleeds when I think about it.

Actors Are Concrete

The `act` method of the `Actor` class flips the actor, so that you can easily see if you forgot to override that method. Of course, there would be an even better way of ensuring that the method has been overridden: to declare it abstract.

Making `Actor` into an abstract class would clearly be the better design. However, abstract classes are typically introduced at the end of the A course and reinforced in the AB course. The committee wanted to make it easy to use GridWorld throughout the A course. We therefore decided to use the conceptually simpler concrete actor class. In the AB course, the `AbstractGrid` class gives a good opportunity for studying abstract classes.

Critter Methods Have Restrictive Postconditions

The `act` method of the `Critter` class calls five other `Critter` methods.

```
public void act()
{
  if (getGrid() == null)
    return;
  ArrayList<Actor> actors = getActors();
  processActors(actors);
  ArrayList<Location> moveLocs = getMoveLocations();
  Location loc = selectMoveLocation(moveLocs);
  makeMove(loc);
}
```

Critter subclasses override one or more of these methods, *but not* the `act` method. Ideally, we would have declared `act` as `final`. However, the `final` modifier is not in the AP Computer Science subset, and we wanted to stay within the subset whenever possible.

When formulating an initial set of potential exam questions, we noticed that we needed a tool to enforce the "spirit" of the design. We wanted to ask questions about design alternatives in which one choice is right and the others are wrong. We wanted to signal that certain implementations are reprehensible, such as eating neighbors in the `getActors` method. In order to make clear what is good and bad, we added restrictive postconditions to the five methods that are called in the `act` method.

To ensure that the `getActors` method has no side effects, we added a postcondition "The state of all actors is unchanged." Similarly, we wanted to make sure that the `processActors` method makes use of the list of actors passed as a parameter rather than looking for actors elsewhere. This explains the first postcondition of `processActors`: "The state of all grid occupants other than this critter and the elements of `actors` is unchanged. New occupants may be added to empty locations." The second postcondition "The location of this critter is unchanged" makes sure that the critter movement is carried out by the `getMoveLocations`/ `selectMoveLocation`/ `makeMove` mechanism.

Under the Hood

GridWorld was designed to be extensible, so that you can easily go beyond bugs and critters. In this section, you will see some of the inner workings of GridWorld and learn how to take advantage of them when you produce your own worlds.

How a World Is Displayed

When the `show` method of a `World<T>` is called for the first time, a `WorldFrame<T>` is constructed. (`WorldFrame` is a generic class to minimize unsightly warnings in the code for the constructor and method menus.) In subsequent calls to `show`, the frame is repainted. You will want to call `show` if you update the grid outside the `step` method, so that your changes are displayed.

The frame shows a `GridPanel` that draws the grid (or a portion of the grid if it is too large), the grid occupants, and the selection square that indicates the currently active cell. The selection square can be moved with the arrow keys or the mouse. Occasionally, you will want to hide the selection square. Call

```
System.setProperty("info.gridworld.gui.selection", "hide");
```

Every GridWorld program in the case study constructs a world, populates it, and then calls the `show` method. But you don't have to call `show`. When you write a test program, perhaps for automated grading of your students' work, you can simply call the `step` method and then check whether the grid occupants have acted correctly. For example, to test whether your students' `ZBug` works correctly, place an instance into the grid, call `step` a number of times, and then check the grid for a Z-pattern of flowers.

How Grid Occupants Are Painted

As you know, you can supply GIF images for grid occupants. Simply give the image the same name as the occupant class, such as `MyCritter.gif`. The image is rotated to the occupant's direction and tinted to the occupant's color. The occupant need not be an actor. The `info.gridworld.gui.ImageDisplay` class simply checks whether there are methods `getDirection` and `getColor`.

You can have multiple images for an occupant. Suppose you want hungry critters to look different from normal critters. Supply a method `getImageSuffix` that returns a string `""` or `"_hungry"`, and supply two images `MyCritter.gif` and `MyCritter_hungry.gif`.

Note: (1) One of the images *must* have the same name as the class, even if you never use it. (2) The `getImageSuffix` method must return any separators such as underscores. (3) Images look best if their size is 48 × 48 pixels. Images are always scaled to a size

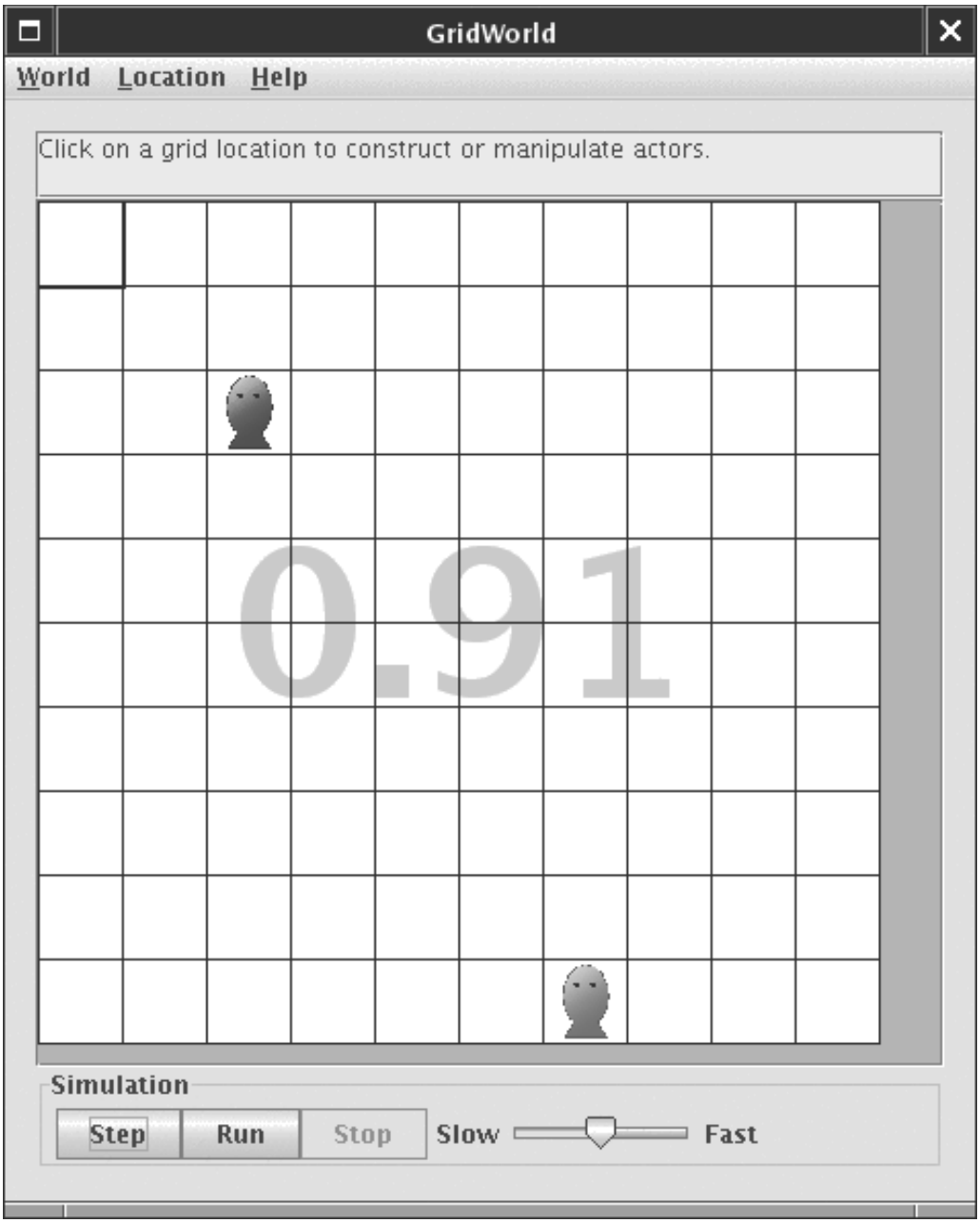
$12 \cdot 2^n \times 12 \cdot 2^n$, where $n \geq 0$. When a grid is first displayed, n is chosen as the largest value for which the grid cells fit in the frame, or 0 if 12×12 cells overflow the frame. When you zoom in or out, n is incremented or decremented. Menu icons are scaled to 16×16. You can supply images of other sizes, but they may not look as good.

Sometimes, you may want to achieve more complex drawing effects. Supply a class that extends `info.gridworld.gui.AbstractDisplay` and whose name is the name of your occupant, followed by `Display`, such as `MyCritterDisplay`. Implement the following method:

```
/**
 * Draw the given object. Subclasses should implement this
 * method to draw the occupant facing North in a cell of size
 * (1,1) centered around (0,0) on the drawing surface.
 * (All scaling/rotating has been done beforehand).
 * @param obj the occupant we want to draw
 * @param comp the component on which to draw
 * @param g2 the graphics context
 */
abstract public void draw(Object obj, Component comp,
     Graphics2D g2);
```

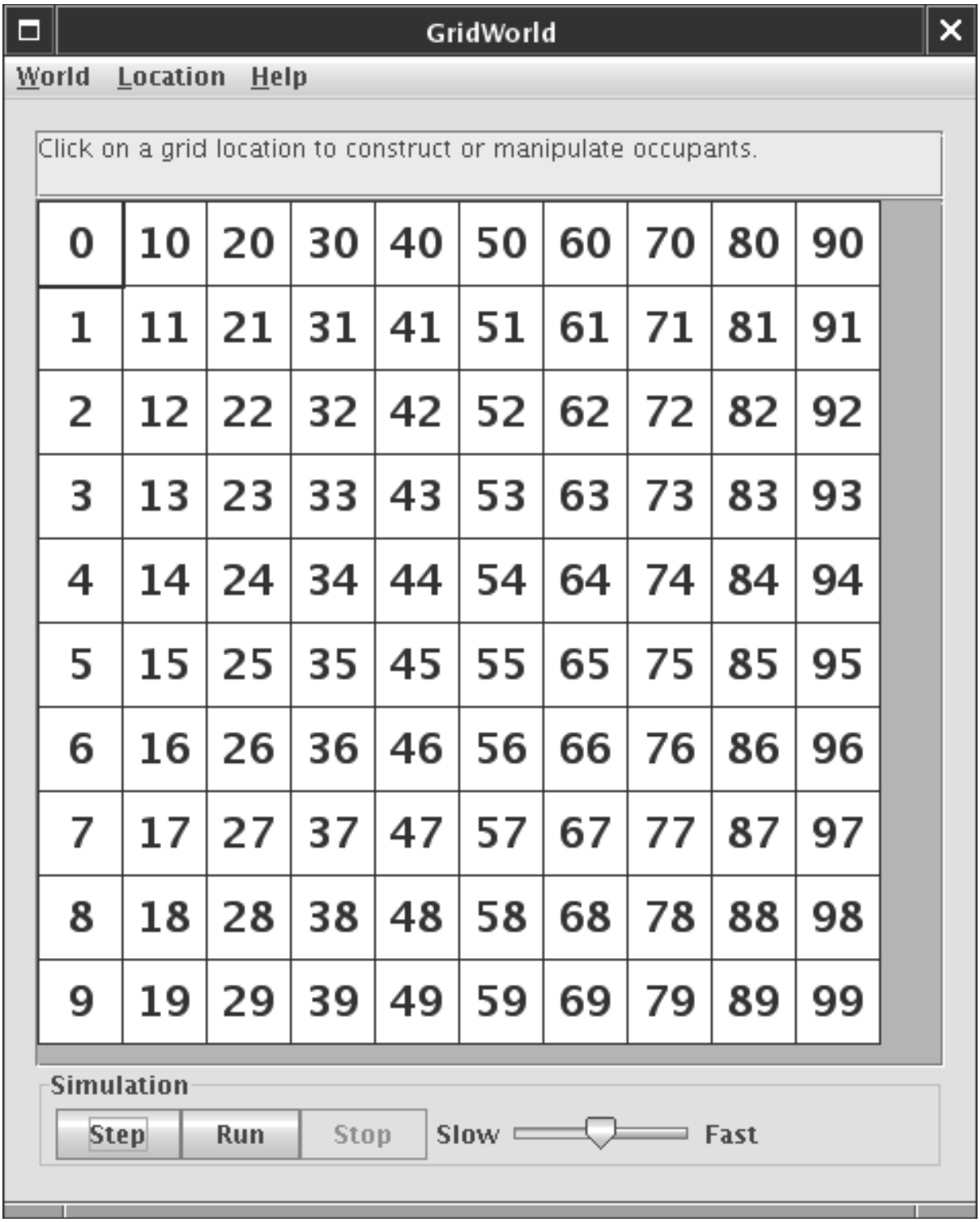
You can use any of the Java AWT drawing calls in the `draw` method.

You may wonder why the `draw` method isn't simply part of the occupant class itself. However, the drawing code is usually quite complex. For example, the class to draw the MBS fish uses the `GeneralPath` and `GradientPaint` classes. By placing the drawing code in a separate class, you can hide that complex code from students.



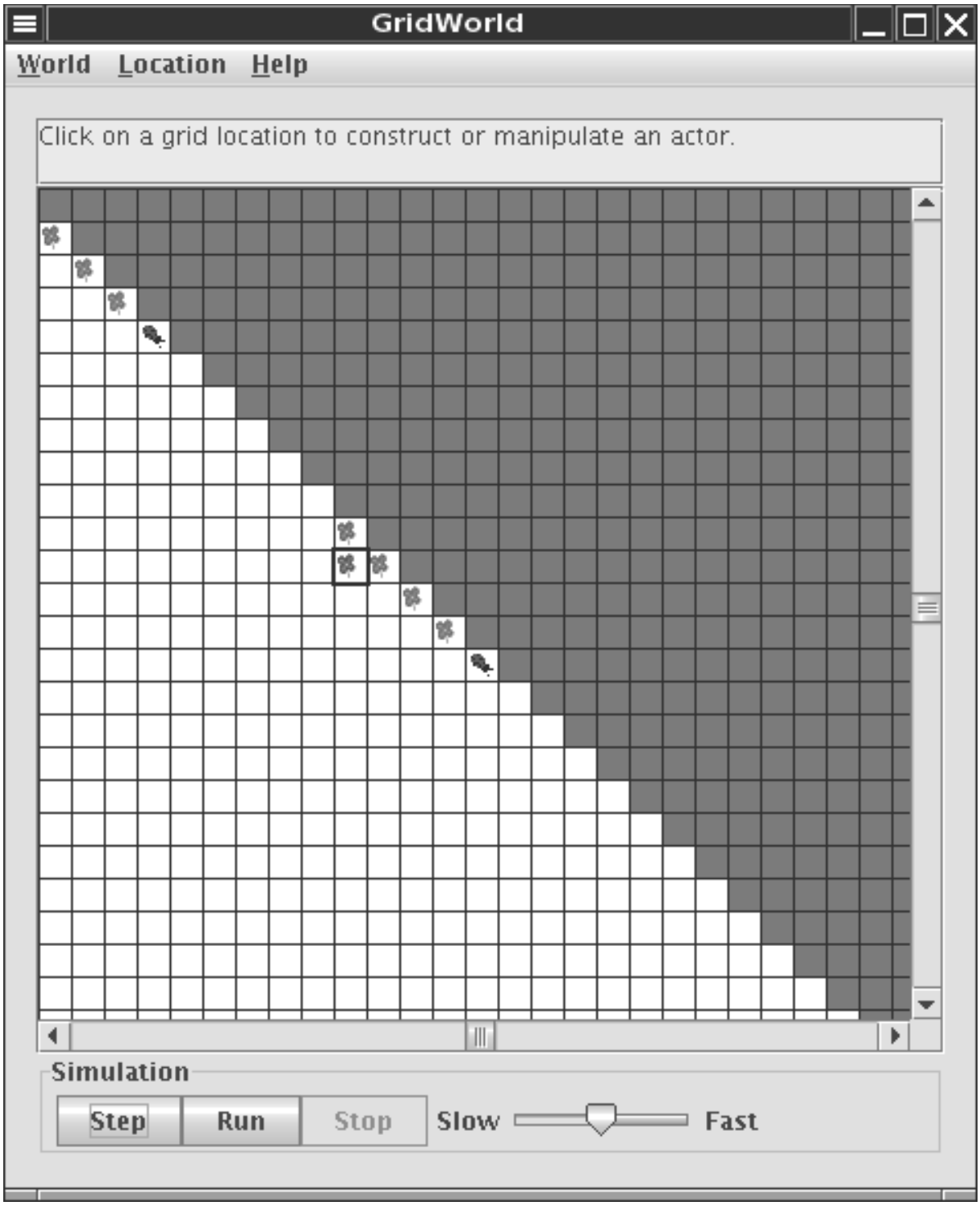
If there isn't a display class or a GIF image that matches an occupant class name, the superclass names are used to find a match. (For that reason, the default drawing of an `Actor` subclass is the actor mask.) If none of the superclasses has a display class or GIF image, then the `info.gridworld.gui.DefaultDisplay` class is used. That class simply fills the cell with a background color and places centered text into the cell. The background color is the value returned by the `getColor` method, or the value of the object itself if it is an instance of the `Color` class. The text is the value of the `getText` method, or, if there is no such method, the `toString` method.

The default display does the right thing for a `World<Color>` or `World<Integer>`.



You can also use the default display for simple game tiles. Define a `Tile` class with `getColor` and `getLabel`.

Note that invalid grid locations are painted gray. Here is an unbounded triangular grid:



How the Buttons Work

When the user clicks the Step button in the GridWorld program, the `step` method of the current world is invoked. When you click on the Run button, the `step` method is invoked repeatedly, until the Stop button is clicked. The frequency depends on the setting of the Slow/Fast slider.

In the `ActorWorld`, the `step` method invokes `act` on each actor.

In your own programs, you can define the `step` method differently. For example, you can easily turn GridWorld into a GUI for Conway's *Game of Life*. Define the `step` method to compute the next generation, then click the `Run` button to see the simulation unfold.

If you leave the `step` method undefined in your `World` subclass, then the buttons won't do anything.

Note that you don't have to click the buttons in your own program. You can write a program that creates a world, invokes `step` multiple times, and calls `show`. The program simply displays the final state of the world. This is useful for speeding up grading—you can visually check whether the student's answer is correct, without having to do any clicking.

How the Message Display Works

The yellow area above the grid is the message display. By default, it is blank for ordinary worlds, and it shows the message "Click on a grid location to construct or manipulate an actor" for `ActorWorld`.

You can set your own message by calling the `setMessage` method. This is useful in many situations. For example, in a game, you can display messages "Player A's turn" or "Game over".

The message can be arbitrarily long; scroll bars will appear when needed.

Each call to `setMessage` replaces the preceding message. If you call `setMessage(null)` in an `ActorWorld`, the default message reappears.

How You Can Intercept Mouse Clicks and Keystrokes

The `World` class has student-friendly mechanisms for intercepting mouse clicks and keystrokes. No knowledge of AWT events is required.

When the user clicks on a grid location, the `locationClicked` method of the `World` is called. By default, that method returns `false`, which tells the framework to initiate the default action, namely to move the selection square and to show the constructor or method menu.

To intercept the mouse click, override the `locationClicked` method. Carry out any desired action and return `true`. The grid location on which the user clicked is passed as a parameter. Typical actions include flipping tiles, populating empty locations, and so on.

Sometimes, you need to ask the user for additional information after the mouse click. The easiest method is to use a `JOptionPane`.

Let's consider a common situation. In a game, a user selects a piece. You want to ask where the user wants to move the piece. You can wait for another mouse click. That means, your `locationClicked` method needs to keep track of the click state (piece selection vs. target selection). Or you can enumerate all legal targets and call `JOptionPane.showOptionDialog`.

When the user hits a key, the `keyPressed` method of the `World` is called. By default, that method returns `false`, which tells the framework to initiate the default key action. If the user hit a cursor key, the selection square is moved. If the user hit the Enter key, the constructor or method menu is shown. All other keys are ignored.

To intercept the keystroke, override the `keyPressed` method. The method receives the current location and the keystroke string, encoded in the same format that is used by the `java.awt.KeyStroke` class. Example keystroke strings are `"INSERT"` or `"alt shift X"`. Your `keyPressed` method should check the keystroke string. If the string matches a keystroke that you want to intercept, carry out any desired action and return `true`. Return `false` for all other keystrokes. It is a good idea to return `false` for the cursor keys and the Enter key. Otherwise, the standard actions are disabled for your world.

Conclusion

You now know the rationale behind some of the design decisions in GridWorld, and you have had a peek under the hood of the GUI. I hope you find this information helpful as you use the GridWorld case study for your Computer Science classes.

Integrating GridWorld

Jill Kaminski
Chaparral High School
Parker, Colorado

Integrating GridWorld

When I first began teaching AP Computer Science, I planned my course around a list of the AP Java Subset and Topic Outline: `if` statements, loops, methods, classes, searching, sorting, case study, etc., etc. I found that using this approach, my students' brains resembled shift registers. They could store content during the current unit of study. But they seemed to "shift" it out as the next unit began, so that they could store the new information. By May, my average student had, shall we say, underwhelming retention. My less-than-average students faired even worse. And this isn't the kind of test you can cram for.

Several years ago, I began to follow the sage advice of former AP Chief Reader Chris Nevison: "Don't ***teach*** the case study! ***Use*** the case study to ***teach computer science***." This advice also appears in the prior case study's teacher's manual: "The case study and the accompanying teacher's manual were designed in such a way that you can use these materials throughout the course. You may, in fact, wish to teach many computer science concepts from the AP CS curriculum through the case study itself." What a concept! I revised my course so that the case study was no longer a unit to be taught, but a tool for teaching a variety of topics.

The good news is: **it worked!** Once my students learned how the case study works, they were able to learn and apply new concepts within that framework. By May, they remembered the concepts as well as the case study. Scores improved! The people rejoiced!

And the best news of all: I think that GridWorld has even more potential to be used throughout a course than the prior case studies (particularly an A course). My students appreciate that this is more of a "real" program, and less like the programs we tend to write in beginning computer science classes (e.g., Student or BankAccount classes). It's well-designed, it's particularly great for teaching inheritance and data structures, and the possibilities for its enhancement by students are really exciting.

I usually begin my year-long AP Computer Science A classes by reviewing the basic Java topics covered in the courses in the pre-AP years. Student activities involve reading, taking notes, doing free-response practice assignments, doing multiple-choice tests, and writing programs. I try to save as much time as possible for the programming, because they enjoy learning by doing, and I find that it's effective. I usually prefer two or so short multiple-choice tests to one long test at the end of each one to three weeks or at the end of each unit.

I typically begin case study work by November. We go through each Part of the Student Manual, and at the conclusion of each Part, we reinforce the concepts with extra labs. I

present new concepts within the GridWorld context, and much of the subset is covered this way. And best of all: it's fun!

Getting Started

You'll have to roll up your sleeves at some point and do some GridWorld programming. You're definitely busy, and probably putting it off, but when you do begin, I think you'll enjoy it. Read Cay Horstmann's article about the design of GridWorld. It will help you understand the behind-the-scenes design and rationale.

If possible, grow to love GridWorld before you present it to your students. They won't love it any more than you do. If this is not possible, then act like you love it. They won't love it any more than you can act like you do. Do the exercises and labs in the Student Manual, and take notes as you think of ways that you can enhance them.

The case study Student Manual is well-written and engaging. Take advantage of this teaching tool! It's a very good textbook for you during these weeks. Throughout your work in the case study, talk through the "Do You Know?" sets with the whole class, and have students do all of the Exercises. These questions and exercises are well designed and valuable.

Part 1

Introduce the case study in a memorable way. As students walk in your room on Day 1 of GridWorld, let them:

- see GridWorld running on a projector.
- see you dressed for the occasion in a festive bug-related t-shirt or tie. Or wear checks or plaid, to represent the grid.
- hear some kind of bug music. Here are a few ideas that are available on iTunes (http://www.apple.com/itunes/):
    - "There Ain't No Bugs On Me" (traditional)
    - "Bugs." Mr. Heath
    - "Bugs." Rosenshontz
    - "Bugs." Renee Austen
    - "Bugs!" Funky Mama
    - "The Bug." (aka "Sometimes You're the Windshield") Mary Chapin Carpenter, and also by Dire Straits
    - "The Time of Your Life." Randy Newman (from *A Bug's Life*—the whole soundtrack is terrific)
    - "Flight of the Bumblebee." Nikolai Rimsky-Korsakov
    - "Antmusic" or "Ant Invasion" Adam and the Ants

You could integrate bug and/or flower decor in your room. Jazz up the computer lab with a bug-related movie poster: *A Bug's Life*, *The Ant Bully* or *Antz* for fun, or *Them!, Eight-Legged Freaks*, *Arachnophobia*, or *The Fly* for horror movie fans. You can find "it" on eBay! Consider an ant farm, or a class pet hermit crab. Perhaps your science department has some creatures that they can loan you. If these ideas don't match your personality or style, then think about other ways to present GridWorld in a positive light.

Explain to your students what a case study is. They probably don't know. Explain to them why the College Board wants them to analyze and modify a complex program.

Explain to them that in the "real world," they'll likely need to modify code more often than write it from scratch. Explain to them that they'll be fired for modifying a class that already works and didn't need to be changed!

Click-ability is a welcome addition! Have students experiment with GridWorld using the table at the end of Part 1, so that they get accustomed to the user interface. This will pay great dividends throughout the course. Take advantage of the *BlueJ*-like drop-down menus to enforce the concepts of state and behavior of objects.

Incorporate ideas from Judy Hromcik's "Early Exercises with GridWorld." Students' interest level skyrockets when they can "do something" to the code, early and often. If your students haven't studied inheritance at this point yet, GridWorld is a terrific tool to present the basic concepts of "Is-A" and "Has-A," even in Part 1.

Part 2

After students are familiar with the basic behaviors of the `Actors`, incorporate role play activities in which students become `Bugs`, `Flowers`, and `Rocks`. Nonacting students can give instructions to the actors. You can use a Twister mat as a fun way to represent the grid, or make a grid on the floor with duct tape.

I'm so grateful to Student Manual authors Chris Nevison and Barbara Cloud Wells for presenting inheritance early! Inheritance allows infinite creativity in the case study.

Don't move on to Part 3 without letting students spend some time extending the `Bug` and/or `BoxBug` classes in various ways. Make sure that students understand the good news about inheritance: you don't have to start from scratch! Most `Bug` behaviors do not have to be rewritten.

As inheritance is explored and new methods are added to subclasses, use the drop-down menus to illustrate that an object of a subclass has its own unique methods, and also the methods in the parent class. A class's methods are displayed in a distinct section of the menu.

This visual cue is a great reminder that an object has methods from its own class, and also methods from its parent classes.

In Part 2, show students that they can create any kind of `Actor` that they'd like. Show them how easy it is to incorporate their own graphics. If you're a Marine Biology Simulation fan, run a demonstration of the Marine Biology Simulation, and have students implement basic MBS Fish movement in GridWorld. Your favorite MBS labs do not have to collect dust after the 2007 AP Exam! Or have students write GridWorld programs using other characters they may know and love, like Karel J Robot, Sonic the Hedgehog, or Pac-Man.

Continue to use Judy Hromcik's ideas as you begin Part 2. Explore Dave Wittry's Game Of Fifteen and WuZiQi projects from his terrific "Board Game Projects" article as you continue in Part 2. When students realize that GridWorld can be modified to implement a wide variety of very different projects, they'll be excited about continuing to work with it, and their understanding of program design will be greatly enhanced.

Part 3

In this important section of the Student Manual, the inner workings of the case study are unwrapped for students. They have likely thought of many creative ideas for GridWorld projects. Now they will learn the information necessary to implement those using solid design practices.

Incorporate some discussion about the GridWorld design into your classes. Review Cay Horstmann's rationale, and share some of this with your students. They will appreciate that the GridWorld designer made the decisions he felt were best, but that he wrestled with some pros and cons along the way. It will really help them solidify their understanding before they begin making major changes to the code.

After completing the Jumper project in the Student Manual, let your students design their own labs! They'll likely need some guidance in the process, but they'll have fun while learning valuable lessons about modifying an existing design. You'll be amazed at what they come up with, and they'll buy in to the assignments.

Part 4

The introduction of the `Critter` class allows the case study to remain fresh. You and your students will think of even more ideas to implement. The sky's the limit! At this point, I think that your problem will no longer be "What will I do?" but rather "We don't have enough time to do all the fun stuff I'd like to!"

After completing the fun activities in the Student Manual, use Robert Glen Martin's Ant Farm. Then, proceed to Reg Hahne's Save My Heart labs. Then try your own ideas, or again, those of your students.

Part 5

It is so much fun to teach data structures using a case study like this! The pros and cons of the various structures come to life as students repeatedly extend the `AbstractGrid` class in various ways.

In Exercise 1, have students use Java's `LinkedList` class first. Then, make a copy of that project, and have them create their own `LinkedList` class using the `ListNode` class. None of the client code from their first `LinkedList` project needs to change after the import of `java.util.LinkedList` is removed, and when there are problems (most often `NullPointerExceptions`!), students have to admit that the problem lies in their `LinkedList` class. Similarly, students can use the case study as the platform for creating a binary search tree to represent the grid.

Download and use Dave Wittry's Generic Data Structures Viewer tool. This terrific application, written by Dave's students, provides students with a visual representation of the data structures that they create. This makes debugging much easier.

And speaking of Dave, definitely consider his AB data structure labs (also from "Board Game Projects.") Students love writing games, and Dave's ideas give very interesting and fun contexts for the study of data structures.

Beyond the Narrative

Continue to use GridWorld as May approaches to reinforce various topics like inheritance, `ArrayLists`, recursion, and design. In the process, students will solidify their knowledge of the topics and the case study itself.

If you didn't get to all of Dave Wittry's more advanced A and AB game projects while studying Parts 3–5, then consider using them after you're finished going through the Student Manual. Dave's ideas provide a launching pad for many other grid-based game ideas. Consider Tic-Tac-Toe, Pac-Man, Minesweeper, Hunt the Wumpus, role-playing games, and games commonly downloaded to programmable calculators. And have fun! The best of these will become project staples for years to come.

Different Strokes for Different Folks

All students are created equal. But not all students are able to perform equally in our classes. I usually have a high percentage of AP Computer Science students for whom mine is their only AP course. I encourage their enrollment, and I do my best to help them achieve their best on the exam (even if that means I help them earn a 2). They will still be better off in college for the experience of taking an AP course and exam, according to the 2007 *Advanced Placement Report to the Nation*.

The key is successful differentiation in our classes. Here are a few ideas pertaining to GridWorld in particular:

- Differentiate by various forms of assessments. Multiple-choice quizzes on GridWorld will help prepare students for the AP Exam, and will help improve scores for those who struggle with multiple-choice questions by giving them more practice.
- Differentiate by presenting their assignments in various forms. Students should not only read about a programming project but also see it run (on a projector) prior to attempting it. This will help ESL students and those with poor reading skills.
- Differentiate the amount of work within a programming project. Break down the assignment, and allow credit for the subparts completed. For example, when working on Robert Glen Martin's Ant Farm lab, give credit for implementing the QueenAnt class, and then the WorkerAnt class, and then the Cookie class. This way, even if a student doesn't complete the entire lab, he or she can still gain learning and grades along the way. You can also add additional parts to labs for the advanced students.
- Differentiate the number of programming projects. I usually list multiple assignments within a unit from easiest to hardest. The projects at the bottom are not a punishment for my overachievers, but are designed to provide more fun and challenge to the advanced students. They generally want to get there! This keeps them motivated and gives me time to work on less-challenging labs with students who need more of my help. When grading, I assess whether students did their best to complete the highest number of labs possible, with excellent quality, in the given amount of time.

Closing Thoughts and Ideas (on GridWorld and AP Computer Science in general)

- I've found that my students will work for food! This is an especially effective strategy when I need to give an extended lecture, since they're happy and they often talk less when their mouths are full. From time to time, try one of the following as a treat:
  - Look for fruit snacks in bug shapes.
  - Get a large container of Chex Party Mix. The Chex can represent a grid. Or, add fish objects to the grid, and serve Pepperidge Farm Goldfish crackers. Reuse plastic cups to serve the snacks, and have students write their names on the cups.
  - Get one of the bug-shaped Pez dispensers (Pez Bugz), and reward a kid or two each day with a little Pez candy. They'll be much more likely to answer your questions enthusiastically!
- Read them *On Beyond Bugs! All About Insects* by Tish Rabe.
- Play motivating snippets from movies like *A Bug's Life*, *Cars*, *Finding Nemo*, *Rudy*, and *Invincible*. Look for the inspiring scenes in which our hero is discouraged and a friend picks him up, or when he prevails in the end even though the odds were against him. My favorite is Dory's "Just keep swimming" speech in *Finding Nemo*. I tell my students

to remember this line in the middle of any difficult test, when they might be ready to give up.

- Change gears if things aren't working in your class. Abandon your plans, and meet the kids' needs. Be flexible. Hold them accountable, but be realistic in your expectations.
- Have a donut party for them on the morning of the exam. Let them anxiously ask you those last-minute questions. Give them your sage advice, like "never leave a free-response question blank." Tell them that you're proud of them. Read them some inspirational quotes, stories, or poems. Tell them to "Just keep swimming." After the exam, tell them that if they want to, they can still be software engineers when they grow up, no matter what their test score was! The experience will have been worthwhile, and when they take those first college programming courses, they'll be ahead of the class.
- Smile. Have fun. You shouldn't be teaching unless you love it . . . at least, most of the time!
- Please consider sharing your GridWorld ideas and successes on the AP Computer Science Electronic Discussion group and the CSTA Web Repository.

References

Bergin, Joseph, et al. 2005. *Karel J Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java.* Dream Songs Press.

BlueJ. http://www.bluej.org

College Board. 2007. *Advanced Placement Report to the Nation*.

College Board. *AP Computer Science Electronic Discussion Groups*. http://apcentral.collegeboard.com

Computer Science Teachers Association. *CSTA Source: a Web Repository of K-12 Teaching and Learning Materials for Computer Science.* http://csta.acm.org/Resources/sub/WebRepository.html

Disney/Pixar. 1998. *A Bug's Life*. DVD or VHS.

Disney. 2003. *Finding Nemo.* DVD or VHS. Walt Disney Video.

Disney. 2006. *Invincible*. DVD. Walt Disney Home Video.

Hunt the Wumpus. *Wikipedia*. http://en.wikipedia.org/wiki/Hunt_the_Wumpus

iTunes. http://www.apple.com/itunes/

Levine, David, and Steven Andrianoff. 2003. *Role Playing In an Object-Oriented World*. St. Bonaventure, New York: St. Bonaventure University, Department of Computer Science. http://www.cs.sbu.edu/dlevine/RolePlay/roleplay.html

Minesweeper (computer game). *Wikipedia*. http://en.wikipedia.org/wiki/Minesweeper_%28game%29

Pac-Man. *Wikipedia*. http://en.wikipedia.org/wiki/Pac-Man

Rabe, Tish. 1999. *On Beyond Bugs!: All About Insects*. New York: Random House.

Sonic the Hedgehog. *Wikipedia*. http://en.wikipedia.org/wiki/Sonic_the_Hedgehog

Sony. 1993. *Rudy*. DVD or VHS.

Tic-Tac-Toe. *Wikipedia*. http://en.wikipedia.org/wiki/Tic-tac-toe

Twister. Board game. Hasbro.

Wittry, Dave. "Generic Data Structures Viewer" (GDSV). *AP Computer Science*. Taipei American School. http://www.apcomputerscience.com/gdsv.index.html

Early Exercises with GridWorld

Judith Hromcik
Arlington High School
Arlington, Texas

Chapter 1: Observing and Experimenting with GridWorld

The GridWorld GUI is an interactive GUI. For each actor that has been placed in the grid, students can observe the actor's behavior, test its methods, and learn how it acts by right-clicking on the actor in the grid and running its methods. This unit is designed to let students explore and discover what functionality the actors possess and how the actors behave in the grid.

Modifying `BugRunner.java`

The `BugRunner` class contains a main method that when compiled and executed, runs the GridWorld GUI. The grid is an interactive part of the GUI. Right-clicking on an empty cell in the grid will allow the user to add a new actor in the grid during the simulation. Right-clicking on an occupied cell in the grid will allow the user to run methods for that occupant.

After compiling and running the `BugRunner.java`, make small changes to the file. The original main is shown below:

```
import info.gridworld.actor.ActorWorld;
import info.gridworld.actor.Bug;
import info.gridworld.actor.Rock;

/**
 * This class runs a world that contains a bug and a rock,
 * added at random locations. Click on empty locations to add
 * additional actors. Click on populated locations to invoke
 * methods on their occupants. To build your own worlds,
 * define your own actors and a runner class. See the
 * BoxBugRunner (in the boxBug folder) for an example. This
 * class is not tested on the AP Computer Science A and AB
 * exams.
 */
1. public class BugRunner
2. {
3.   public static void main(String[] args)
4.   {
5.     ActorWorld world = new ActorWorld();
```

```
6.      world.add(new Bug());
7.      world.add(new Rock());
8.      world.show();
9.    }
10. }
```

Suggested changes:

1. Comment out line 7, recompile the file and execute.
   a. Before stepping or running the simulation, right-click on an empty cell in the grid. What types of actors can be added to the grid at this point?
   b. Step or run the simulation. Right-click on an empty cell in the grid. What types of actors can be added to the grid at this point?
2. Uncomment line 7, recompile the file and execute.
   a. Before stepping or running the simulation, right-click on an empty cell in the grid. What types of actors can be added to the grid at this point?
   b. Step or run the simulation. Right-click on an empty cell in the grid. What types of actors can be added to the grid at this point?

After making these changes, students should realize that they can only add the same types of objects that currently exist in the grid.

Questions

1. The original `BugRunner` class adds a `Bug` and a `Rock` to the grid. Before the simulation runs one step, you can add a new `Bug`, `Rock`, or `Actor`. Why can an `Actor` object be added to the grid when the `BugRunner` did not add an `Actor` (i.e. `world.add(new Actor());`)?
2. Right-click on every `Bug` in the grid and run a method that will allow you to move the bugs out of the grid without a runtime exception. What method(s) did you run? Right-click on an empty spot in the grid. Can a `Bug` be added to the grid?

An `Actor` can be added to the grid if any `Actor` subclass has been added in the `BugRunner` class. This is a good time to introduce the idea of inheritance and the idea of IS-A.

A `Bug` IS-A `Actor`
A `Flower` IS-A `Actor`
A `Rock` IS-A `Actor`

When all `Bug` objects are removed from the grid by using the `move` or `removeSelfFromGrid` methods, new `Bug` objects can still be added to the grid. Note: Using the `moveTo` method to remove any `Actor` from the grid will cause a runtime exception to occur.

After the idea of inheritance has been introduced, right-clicking on a `Bug` object in the grid visually shows the students which methods are defined by the `Bug` class (a `Bug` image appears by these methods) and which methods the `Bug` class inherits from the `Actor` class (an `Actor` image appears by these methods).

Exercises

1. Run `BugRunner.java`. Right click on one of the `Bug` objects in the grid and do the following:
   a. list all of the methods that the `Bug` class defines
   b. list all of the inherited methods from the `Actor` class that can be called from a `Bug` object
   c. draw an inheritance hierarchy diagram for the `Bug` class, `Flower` class, and `Rock` class
2. List all the cases that will cause the `canMove` method of a `Bug` object to return `false`. Run the simulation to test your cases.
3. List all the methods that will cause a `Bug` object to turn.
4. Right-click on a bug in the grid and force it to move to a location that is occupied by a flower, a rock, an actor, and another bug. What happens if a bug moves into a location that is occupied by a flower? an actor? a rock? another bug?

Chapter 2: Bug Variations

Chapter 2 of the case study is teaching simple inheritance. The code for the `Bug` class constructors and the `act` method is the only code that the students need to really understand at this point. Students need to only use the `canMove`, `turn` and `move` methods. In such a constrained space, students can focus on creating a subclass of the `Bug` class by modifying the `BoxBug` class.

General notes for creating `Bug` subclasses:
When students extend the `Bug` class

- they should only override the `act` method;
- each call to the `move` method should be guarded by a call to the `canMove` method;
- private instance variables should be added to the subclasses as necessary;
- new methods for the `act` method to call can be added to the subclass;
- constructors for the subclasses should be written to initialize any private instance variables declared in the subclass;
- `super();` or `super(someColor);` can be introduced at this time if students are ready.

Additional Beginning Labs for Chapter 2

JumpingBug:

A `JumpingBug` is a `Bug` that tries to move two spaces ahead each step. If a `JumpingBug` can move, it will move one space ahead and then try to move again. It is possible that it will only move one space if the second move is not possible. If a `JumpingBug` does not move at all, it will turn 90 degrees and change its color to a random color.

Create the `JumpingBug` class. In writing this class, create a new method, `randomColor`, that will return a random color when called.

The key to writing the `act` method for this class is to guard both calls to `move` with calls to `canMove`:

```
public void act()
{
     if (canMove())
     {
        move();
        if (canMove())
          move();
     }
     else
     {
        turn();
        turn();
        setColor(randomColor());
     }
}
```

A common mistake for beginning students is to omit the second call to `canMove`.

This lab emphasizes the need to guard all calls to `move`. It also provides students the opportunity to create additional methods for the `act` method to call.

DiagonalBug

A `DiagonalBug` is a `Bug` that initially faces northeast, northwest, southeast, or southwest. When a `DiagonalBug` cannot move, it turns 180 degrees. Create the `DiagonalBug` class.

In order to solve this problem, students must realize that they will need to write a constructor for the `DiagonalBug` and set the initial direction in the constructor. A random number should be used to determine which one of the four possible directions each `DiagonalBug` will face.

Once a correct constructor is written, the `act` method should be overridden to a make `DiagonalBug` turn 180 degrees when it cannot move instead of 90 degrees.

This lab emphasizes the need to create a constructor for the subclass and set the proper initial conditions.

Solutions and student handouts can be found at http://www.apluscompsci.com/material.htm.

Pedagogical Rationale

Prerequisite knowledge for students:

- compiling and running a Java project
- creating objects
- calling an object's methods
- making small modifications to an existing program

Pedagogical approach: Discovery learning and formative assessment

Discovery learning is "an approach to instruction through which students interact with their environment—by exploring and manipulating objects, wrestling with questions and controversies, or performing experiments" (Ormrod, 1995, p. 442).

The case study offers instructors a natural vehicle to employ this pedagogical approach.

These lessons will allow a teacher to actively engage students in the learning process and ascertain what the students know and understand. These lessons involve classroom discussion, active questioning, and solving small problems. These are all formative assessment strategies.

Reference

Ormrod, J. 1995. *Educational Psychology: Principles and Applications.* Englewood Cliffs, NJ: Prentice-Hall.

Board Game Projects

Dave Wittry
Taipei American School
Taiwan

General Comments and Some Tips for Differentiation

The word "game" here is being used, in some cases, liberally. While three of the "games" are clearly games, two of them are clearly not. I use the term as it will appear to the students. Those games that are not actually "games" are deemed fun by students—so I list them as "games" as well. You will see what I mean if you use them; they're fun!

These projects have more to do with learning about the individual classes within GridWorld (e.g., `Grid`, `World`, `Location`) than with inheritance. These were chosen so that you could see how one might use GridWorld for things other than `Bugs`, `Actors`, `Rocks`, and `Critters`. We all have our favorite labs we've used over the years (e.g., NQueens, Life)—maybe you can start to see how you can still use those ideas but do so within the context of the case study. One side benefit of doing so is that you'll keep your students within a common metaphor (that of the GridWorld) over the course of many different types of projects, thereby helping them focus on the specific computer science topic at hand and not distracting them with a context change each time you want to move in a new curricular direction.

The nice thing about all of these projects is that you can use each one in many different ways depending on what you want to have your students focus on, how much time you have to dedicate to the individual project, and the current knowledge of your students. The labs can be used in AP Computer Science at either the A or AB level; some can even be used twice when either revisiting topics or when demonstrating alternate algorithms/data structures at another point in the course(s). Some tips for how to do so are given and I'm sure you'll have a few more ideas along the way. Each of the projects comes complete with a solution as well as a suggested starting point for students. (See .java files within the student version of each project—see "Materials" section at the end of this document.) You can simply, for example, delete the body of one method, leaving the students to write it. You pick and choose what parts of the project you want them to write. Students who need more help might be given an initial lab setup that has more code filled in, for example, than those who need or desire less assistance. Also, if you have gifted/talented students in your course, the labs are easily extensible to help push them and keep interest. Consider working with such students by giving them some options and asking for their ideas for extension, then let them run with the ball; you'll get much more out of them when they get to choose.

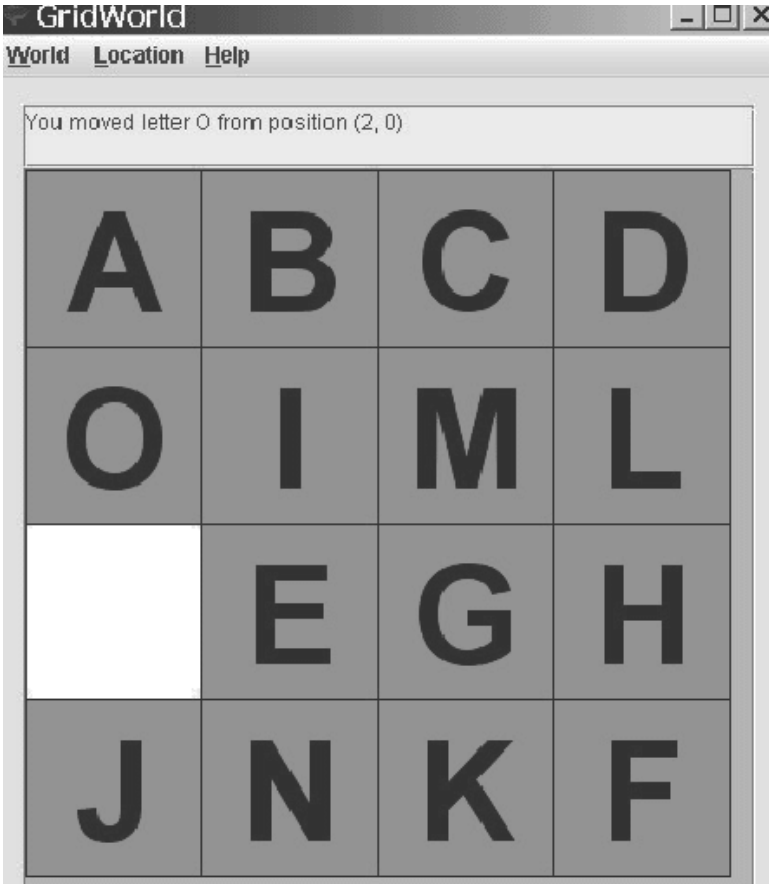
Whether or not we have both A and AB level students in one classroom, we often need to differentiate, not only among levels but within. Some possible extension ideas that would

give the games a more real feel and that apply to all labs below would be to save game state and reinstate at a later time. Or one could keep game statistics. A very advanced extension could be to make the game playable over the network while students sit at different computers. These extensions would teach students, among other things, some of the following:

- to use the `Scanner` class to read/save.
- to serialize an object so as to simplify saving and reinstating the state of an object, as well as sending objects over the network connection.
- to add menu items and buttons, for example, in order to add functionality. This would constitute either inspecting and editing the GridWorld's GUI files and/or creating your own frame with additional controls. The latter option is modeled for you in the Traffic Jam lab below.

The games chosen in this discussion were chosen for several reasons. First and maybe foremost, they were chosen because they are a fun way to learn about computer science. There may be no better way to engage someone than through gaming. Another reason these projects were chosen is that they demonstrate some of the more captivating features of the GridWorld package. What color something is or which custom, student-created graphic is being displayed may not be computer science, but incorporating those simple-to-do features in your projects will sure go a long way toward keeping your students engaged and wanting more. Within the context of these five games you'll see how to accomplish just about everything you need to know about displaying custom colors, graphics, and text. In addition, you'll expose students to the event-driven paradigm with easy-to-handle keyboard/mouse user-driven events. Along the way they will also learn other typical game constructs such as how to take turns, pause play, read/save game data, and more. These will open the door to many other projects. Let the gaming begin!

Game of Fifteen



You may have played this game before. The game of Fifteen is usually played with the numbers 1 to 15. Here, letters of the alphabet are being used. The object of the game is to rearrange the scrambled letters so that they are in alphabetical, row-major order with the hole located at the bottom right. The only pieces you can "slide" in any turn are those located horizontally/vertically from the open slot.

This is a fantastic lab to use early in the case study in the AP Computer Science A course. Your students will get lots of practice with the `Grid`, `Location`, and `World` classes. In addition, they get some good practice with algorithms as they try to figure out how to determine a winner and how to set up a random starting board.

Relevant GridWorld classes and interfaces

```
World, Grid, BoundedGrid, Location
```

Relevant topics

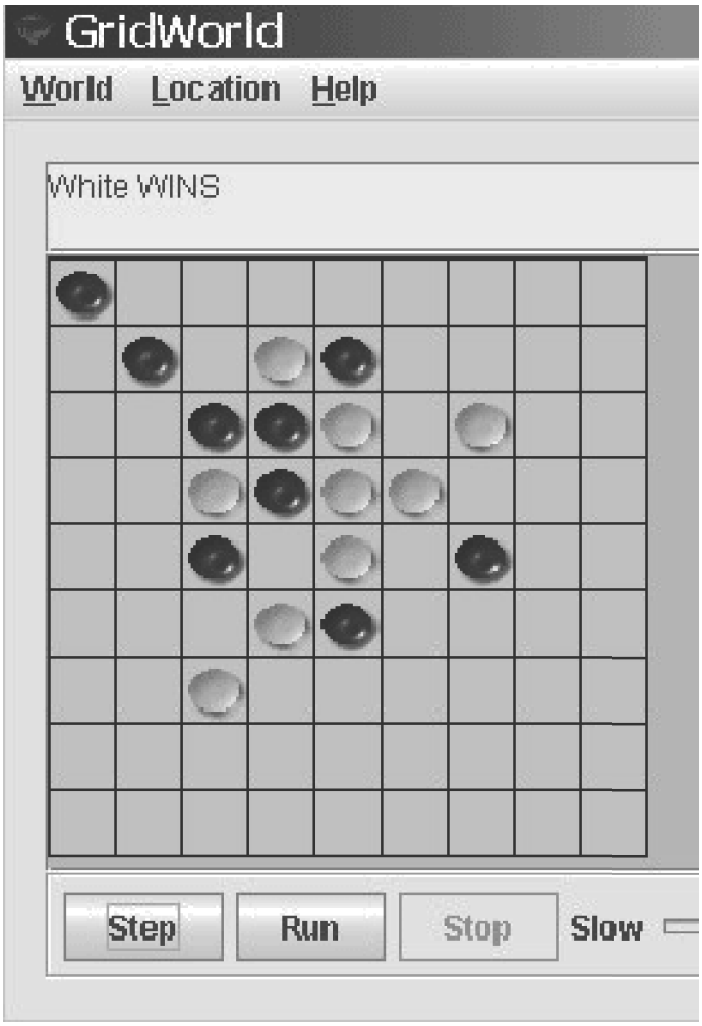
tested on AP Exam:	nested loops, `ArrayList`, "for each" loop, `Math.random()`
not tested on AP Exam:	`Collections` class

Code Example

```
private boolean determineWinner()
{
  Grid<ColorTextPiece> grid = getGrid();
  ArrayList<ColorTextPiece> list = new
  ArrayList<ColorTextPiece>();
  for (int row=0; row<grid.getNumRows(); row++)
     for (int col=0; col<grid.getNumCols(); col++)
       list.add( grid.get(new Location(row, col)) );
  for (int i=0; i<list.size()-2; i++)
  {
    ColorTextPiece first = list.get(i);
    ColorTextPiece second = list.get(i+1);
    if ( first == null || second == null )
       return false; // one of them is the empty cell
    if ( first.getText().compareTo(second.getText()) > 0 )
       return false;
  }
  return true;
}
```

If you're looking for ways to teach arrays within the context of GridWorld, this is the perfect lab. Students will gain practice with creating a sorted array and "shuffling" it in order to create a random starting arrangement. (Here is where you can incorporate `Math.random()` as well.) You can show them `Collections.shuffle()` after they're written their version! Along the way, they'll also get practice with the "for each" loop, handling keyboard events, and using the `Grid` and `Location` classes while they are inserting and removing the game pieces from the grid. The neat thing is, they'll be able to write the whole lab from scratch with just a little bit of guidance. This is a short, fun lab.

Game of WuZiQi (五子棋)



WuZiQi, also known as Gomoku or Gobang, is an ancient, abstract strategy game. An abstract strategy game is one that has no random elements or hidden information. Other examples of abstract strategy games include Chess and Checkers. The object of WuZiQi is to get 5 of your pieces in a row either horizontally, vertically, or diagonally. Black moves first and then play alternates.

Relevant GridWorld classes and interfaces

`World, Grid, BoundedGrid, Location`

Relevant topics

tested on AP Exam: `ArrayList`, 2D arrays, writing `equals()`
not tested on AP Exam: displaying and switching between custom images, taking turns mechanism (`boolean`), event-handling, determining a winner

Graphics

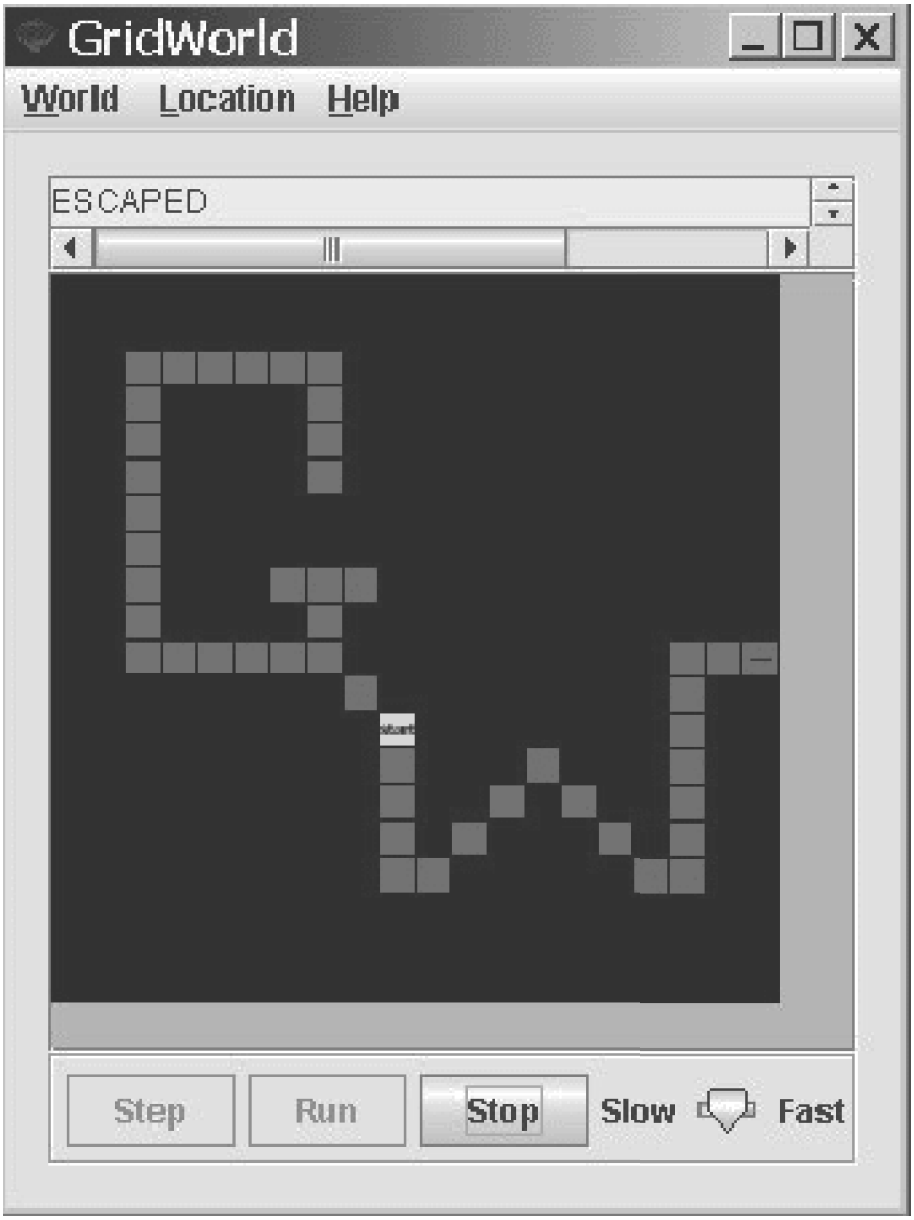
For notes on how to dynamically use/swap custom graphics, colors, and text within GridWorld projects, see the `GamePiece` class. The explanations come with specific code examples and are generalized for any project.

Code Example

```
public boolean determineWinner(Location loc)
{
  // only need to check 5 in a row from the current loc
  // (last move made)
  int dir = Location.AHEAD, consecutive = 0;
  Location nextLoc;
  Grid<GamePiece> gr = getGrid();
  // There are 4 main diagonals to check for 5-in-a-row.
  GamePiece piecePlayed = gr.get(loc);
  for (int numDiagonals=1; numDiagonals<=4; numDiagonals++)
  {
   nextLoc = loc.getAdjacentLocation(dir);
   consecutive = 1; // current piece played counts as 1 so far
   for (int i=1; i<=2; i++)
   {
     while ( gr.isValid(nextLoc)
        && gr.get(nextLoc).equals(piecePlayed) )
     {
      consecutive++;
      nextLoc = nextLoc.getAdjacentLocation(dir);
     }
     dir += Location.HALF_CIRCLE;
     nextLoc = loc.getAdjacentLocation(dir);
   }
   if ( consecutive >= 5 ) return true; // we have a winner
   dir += Location.HALF_CIRCLE; // back to starting diagonal
                                // direction
   dir += Location.HALF_RIGHT;  // turn to the next diagonal
  }
  return false;
}
```

This method alone will give students excellent practice with the `Location` class. It can be written in O(1) in terms of N, where N is the number of pieces played so far. This gives you a nice way to challenge your AB students if you wish, and leads to a nice, yet optional, discussion with the A students depending on the various solutions they code.

Escape from a Maze



This is your typical escape-from-a-maze lab. You click in an empty cell and then the algorithm takes over, finding its way "out" (to the edge of the grid).

In addition, if you're looking to teach I/O, this is a great place to do it. This lab shows you how to create a text file with maze data, read it, and use it to build the initial starting world/grid.

Relevant GridWorld classes and interfaces

```
World, Grid, Location
```

Relevant topics

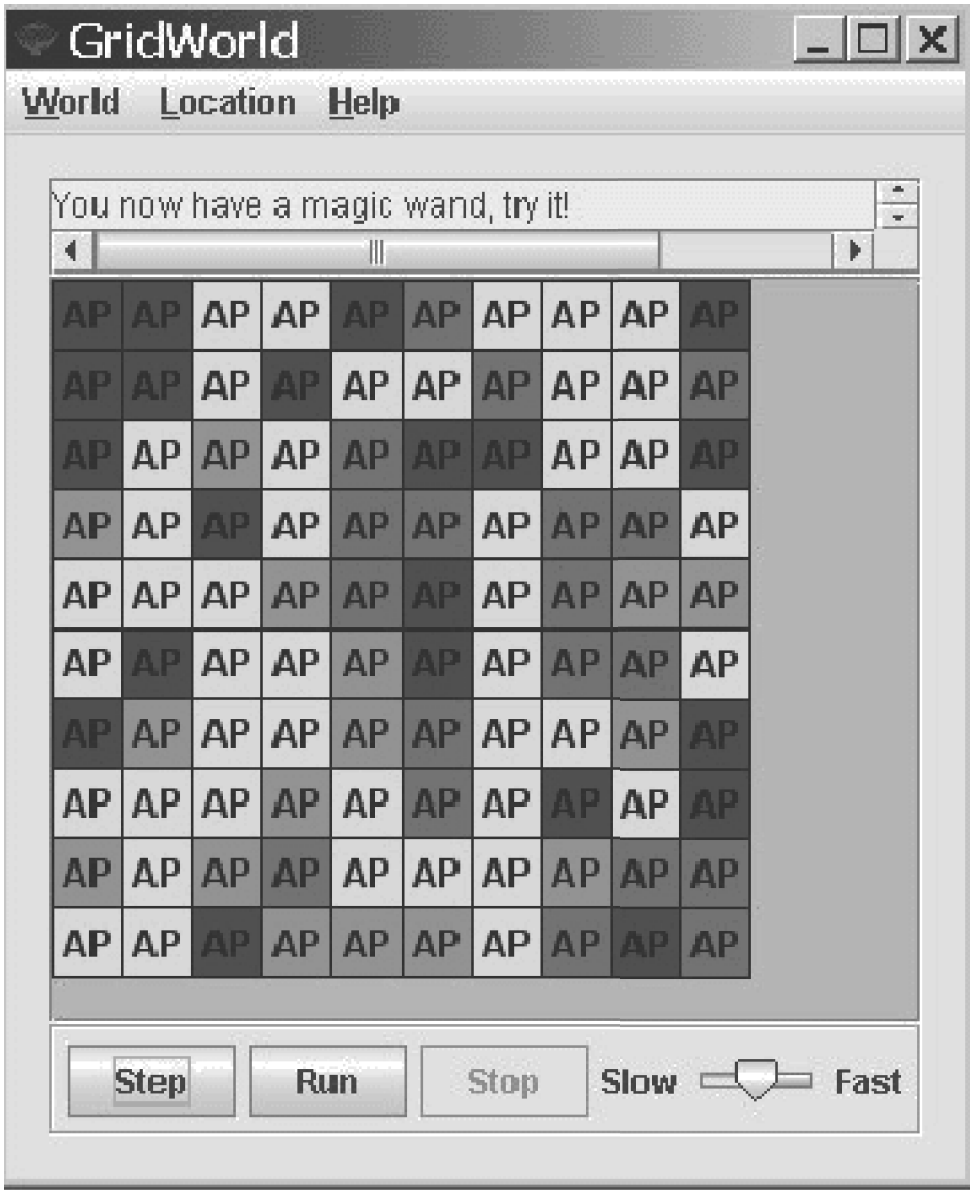
tested on AP Exam:	nested loops, recursion, `Stack`, `ArrayList`, "for each" loop
not tested on AP Exam :	`Scanner` class (file i/o)

Code Example

```
private boolean escape(Location loc)
{
   Grid<Tile> gr = getGrid();
   ArrayList<Location> nbrs = gr.getEmptyAdjacentLocations(loc);
   if ( nbrs.size() == 0 ) return false;
   Stack<Location> stk = new Stack<Location>();
   Location aLoc;
   for (Location tmpLoc: nbrs)
     stk.push(tmpLoc);
   while ( !stk.isEmpty() )
   {
     aLoc = stk.pop();
     if ( onBorder(aLoc, gr.getNumRows(), gr.getNumCols()) )
     {
      add(aLoc, new Tile(Color.RED, "escaped"));
      return true; // found a way out, we're done
     }
     else
     {
      add(aLoc, new Tile(Color.RED));
      nbrs = gr.getEmptyAdjacentLocations(aLoc);
      for (Location tmpLoc: nbrs)
       stk.push(tmpLoc);
     }
     pause("searching...");
   }
   return false;
}
```

While this might more elegantly be written using recursion (sample recursive solution in the project files), this would be a good lab to use early on when teaching stacks. Students will be able to watch the cells being visited (a pause is built in). You can have discussions about how subtle changes in the code will lead to certain pathways being attempted before others. You can have students try to predict what will happen. This lab could be used at the A level when teaching recursion; then you can revisit it in your AB course. Another suggestion might be to have the teacher use this lab as a demo in class and then assign MagicWandWorld.

MagicWandWorld



MagicWandWorld is based on the paint can tool found in many graphic editing programs. You first select a cell (color) with the "eye dropper" and from then on, anything you click will turn that color, along with all adjoining cells of the same color. So, for example, if you touch a light gray cell when you begin and then you go and touch the dark gray cell in the upper-left corner, the dark gray cell and its four blue neighbors will all turn light gray. It's a simple idea—and the kids love it. When things go wrong with their recursive solutions, they immediately know—and often laugh. On their first attempt, for example, students might end up turning the whole grid light gray with one click—that always gets a good laugh from a neighbor.

Relevant GridWorld classes and interfaces

`World, Grid, Location`

Relevant topics

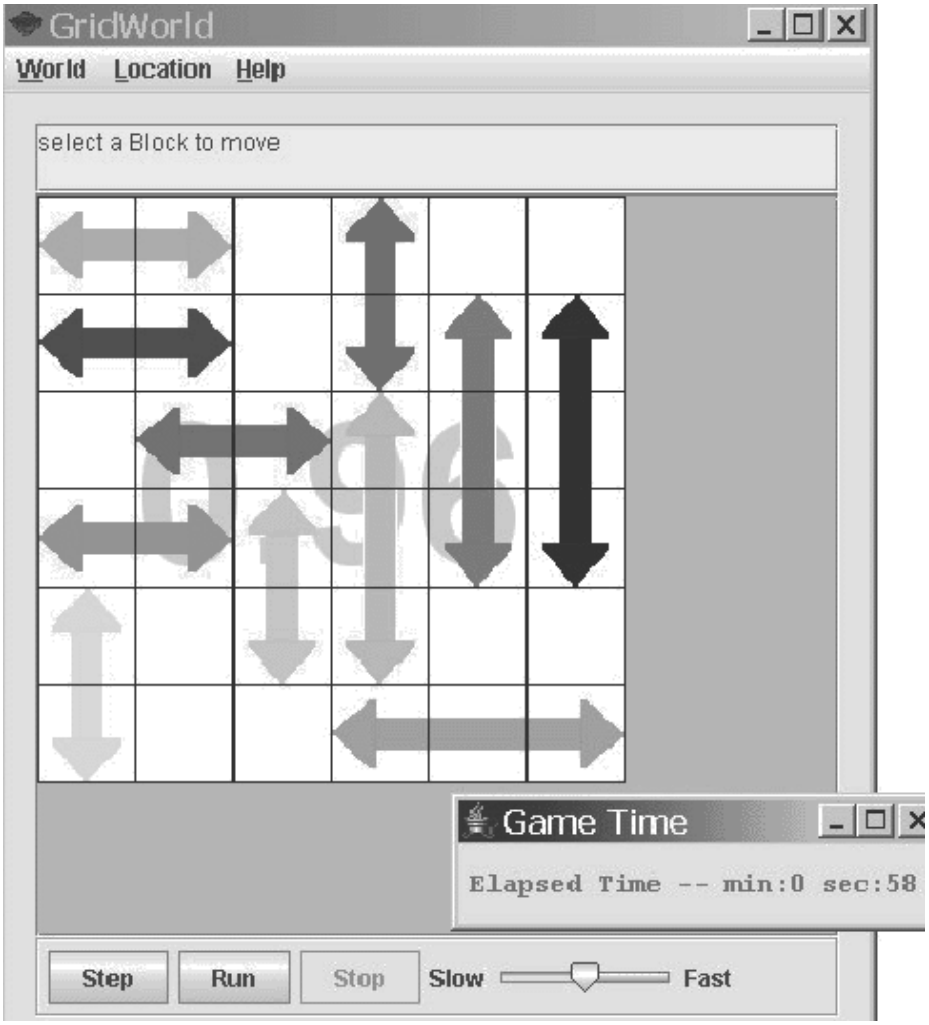
tested on AP Exam: recursion, `Stack`, array, nested loops, "for each" loop, `Math.random()`
not tested on AP Exam: event-handling

Code Example

```
/*
 * This is the recursive method that does the changing of the
   colors.
 */
public void changeColors(Location loc, Color col)
{
   Grid<Tile> gr = getGrid();
   if ( ! gr.get(loc).getColor().equals(col) )
     return;
   gr.put(loc, new Tile(startingColor));
   ArrayList<Location> nbrs =
        gr.getOccupiedAdjacentLocations(loc);
   for (Location aLoc: nbrs)
     changeColors(aLoc, col);
}
```

This is a real good lab to use early on when teaching recursion. The solution is short and manageable, not to mention fun (and it doesn't have anything to do with factorials or Fibonacci numbers! ☺). You might then choose to revisit the lab when you are teaching the AB level, having students, instead, write the solution using a stack. This is a nice way to give application to your discussion of the relationship between recursion and a stack.

Traffic Jam



Traffic Jam is a game based on Rush Hour by Nob Yoshigahara. The goal of the game is simple. Move the arrows around until you can get the red arrow over to the right wall.

This lab is nice for the AB level because one possible way to implement the arrows needing to be stored in the grid is to extend the `BoundedGrid` class. The new class (`MultiCellBoundedGrid`) contains a private `Map` that holds objects of type `Arrow`. Each arrow then knows about itself and how many cells it takes up. When adding objects to `MultiCellBoundedGrid`, the new implementation takes advantage of inheritance, storing all individual pieces of the arrows in the `BoundedGrid`.

This working version provides you with 40 game situations. Students learn to read from a text file to load the games. They use the keyboard to choose between games and then the mouse to start moving arrows. As soon as they start, a `JFrame` pops open and starts timing

them, introducing them to threads, one simple GUI component, and some basic `Graphics` methods. If you don't like to spend a lot of time teaching GUI, this model works as a nice teaser for those students who need differentiation and desire more depth.

Relevant GridWorld classes and interfaces

World, Grid, BoundedGrid, Location

Relevant topics

tested on AP Exam: inheritance (extending `BoundedGrid`), nested loops, `ArrayList`, `Map`, "for each" loop

not tested on AP Exam: `Scanner, StringTokenizer, JFrame, Thread`

Code Example

```
public class MultiCellBoundedGrid extends BoundedGrid<Piece>
{
  //Map holds "arrows" that take up multiple BoundedGrid
  //locations
  private Map<Location, Blockable> blocks;
  public Blockable putBlock(Location loc, Blockable blk)
  {
    Blockable oldBlk = blocks.remove(loc);
    blocks.put(loc, blk);
    // now put individual parts/pieces of Block into
    // BoundedGrid
    for (Piece p: blk.getCells())
     if ( isValid(p.getLocation()) )
      put(p.getLocation(), p);
     else
      throw new IllegalArgumentException(
                "illegal Block location being put");
    return oldBlk;
  }

  public void clearGrid()
  {
    for (Location loc: getOccupiedLocations())
      remove(loc);
    blocks = new HashMap<Location, Blockable>();
  }
  // other methods and constructors not shown
}
```

This project is very involved. It is suggested as a final project. Just as with the other labs, however, you can give students a working version with just a few methods needing to be finished. If you want to have students design their own "Arrows," this lab shows you how the GridWorld GUI works with images, rotating, and coloring.

The code example above gives you a small idea of how the `MultiCellBoundedGrid` works by extending `BoundedGrid`. The `MultiCellBoundedGrid` has a private `Map` which holds the "arrows" (`Blockable`). `BoundedGrid` was extended so that it could hold the individual pieces of each arrow.

Reference

Yoshigahara, Nob. Rush Hour (board game.)
http://www.thinkfun.com/RUSHHOUR.ASPX?PageNo=RUSHHOUR.

Materials

The annotated code solutions and suggestions for these projects and more related to GridWorld may be found at: www.apcomputerscience.com/gridworld
user: apteacher
password: specialfocus

Ant Farm Project

Robert Glen Martin
School for the Talented and Gifted
Dallas, Texas

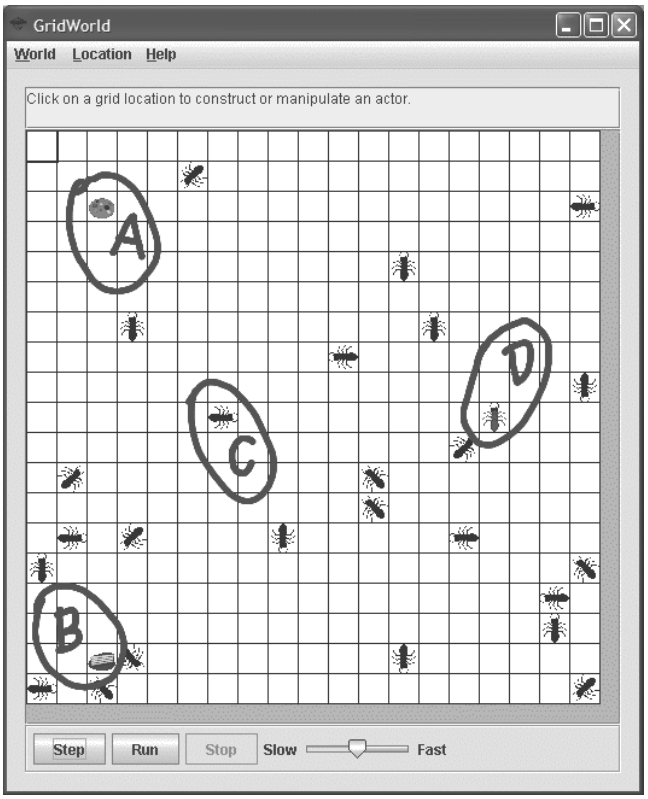
Introduction

This is a detailed overview of Ant Farm, a GridWorld programming project that is appropriate for students taking AP Computer Science A or AB. Teachers can download the student assignment, starter files, and a written follow-up assignment. The completed project and follow-up assignment key are included for teachers. More details about the download are included at the end of this document.

As students complete the Ant Farm project, they write an interface and both concrete and abstract classes. They demonstrate inheritance, encapsulation, and polymorphism. Prior to beginning the Ant Farm project, students must read and understand the first four chapters of the GridWorld narrative.

Overview

Figure 1—Ant Farm (Initial State)—Worker ants hunt for food.



The project utilizes four new types of objects (see **Figure 1**), two kinds of food (**A** - `Cookie` and **B** - `Cake`) and two kinds of ants (**C** - `WorkerAnt` and **D** - `QueenAnt`). Originally, the worker ants walk around in search of food. When they find food, they take a bite. Ants with food turn red. Then the worker ants go in search of a queen ant to give food. Once they give their food to a queen, they turn black and go back to get more food.

Food and queens remain stationary. Worker ants remember the locations of food and the queen. Additionally, they share those locations with other worker ants they meet.

When the Ant Farm program starts, the worker ants are spread around the grid in random locations. Initially, they are disorganized as they search for food. As the worker ants start to find food and the queen, they get more organized (see **Figure 2**). After all the ants learn the locations, they exhibit an emergent behavior that is very organized (see **Figure 3**).

Figure 2—Intermediate State—worker ants start learning locations of food & queen.

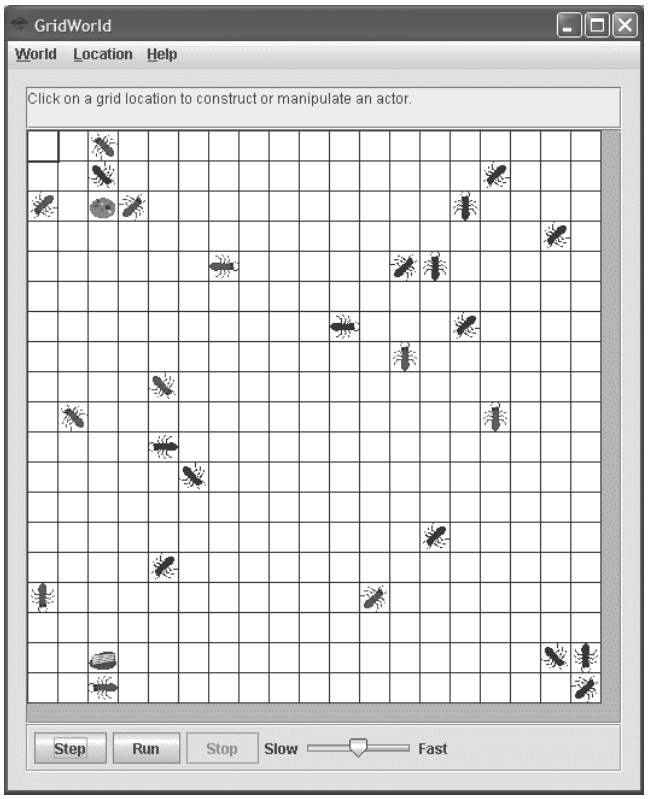
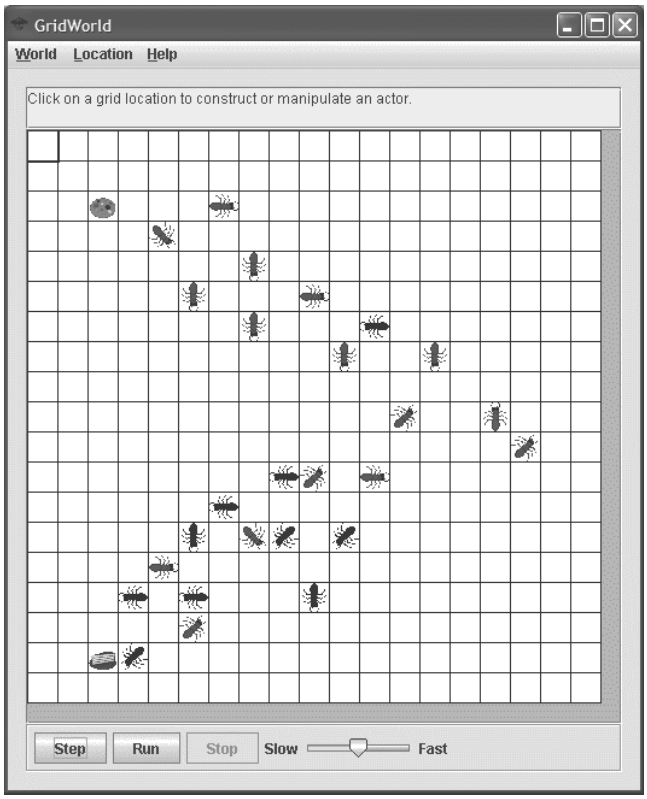


Figure 3—Final State—worker ants know locations of food & queen.



Program Organization

Figure 4—Ant Farm Classes

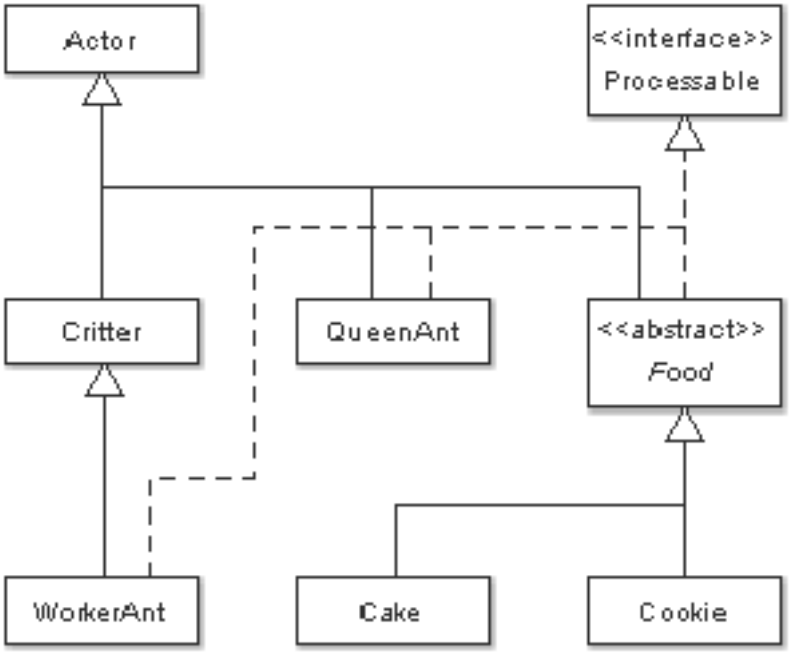


Figure 4 shows the organization of the Ant Farm classes and interface.

GridWorld has a "built-in" `Actor` class for objects that "live" in the grid. Actors that have minimal interaction with other objects in the grid normally inherit from Actor. This is appropriate for `QueenAnt` and `Food`. `Cake`, and `Cookie` also inherit indirectly from `Actor`.

The other "built-in" actor is `Critter`, which inherits from `Actor`. `Critters` have additional methods that are useful for interacting with other actors. `WorkerAnts` need to "communicate" with the `QueenAnt`, `Cake`, `Cookie`, and other `WorkerAnt` objects, so inheriting from `AntFarmCritter` is appropriate for them.

Ant Farm also has a new `Processable` interface. This interface has a single `process` method that is the key to communication between worker ants and the other actors.

Now we'll discuss each of the new classes and interface.

Processable Interface

Figure 5—Processable Interface

```
public interface Processable
{
   void process(WorkerAnt ant);
}
```

Figure 5 shows the complete `Processable` interface. It contains a single public abstract **`process`** method. Note that all interface methods are automatically public and abstract. When implemented in **`QueenAnt`**, `Food`, and `WorkerAnt`, this method processes (communicates with) a single `WorkerAnt` object (the one passed as a parameter). This interface allows worker ants to invoke the other actor's `process` methods polymorphically. The individual `process` methods will do the following:

- `QueenAnt`
  - Get food from the worker ant.
  - Give the queen's location to the worker ant.
- `Food`
  - Give food to the worker ant.
  - Give the food's location to the worker ant.
- `WorkerAnt`
  - Give the saved food location to the other worker ant.
  - Give the save queen location to the other worker ant.

Note that Ant Farm uses the `Processable` interface to implement an interface variant of the Template Design Pattern. The Template Design Pattern normally uses an abstract class to contain the abstract method(s). Then concrete classes (which inherit from the abstract class) implement the method(s) as appropriate. In Ant Farm, we use the `Processable` interface to hold the abstract `process` method. `process` methods are written in the `QueenAnt`, `Food`, and `WorkerAnt` classes, each of which implement `Processable`.

QueenAnt Class

Figure 6—Queen Ant class

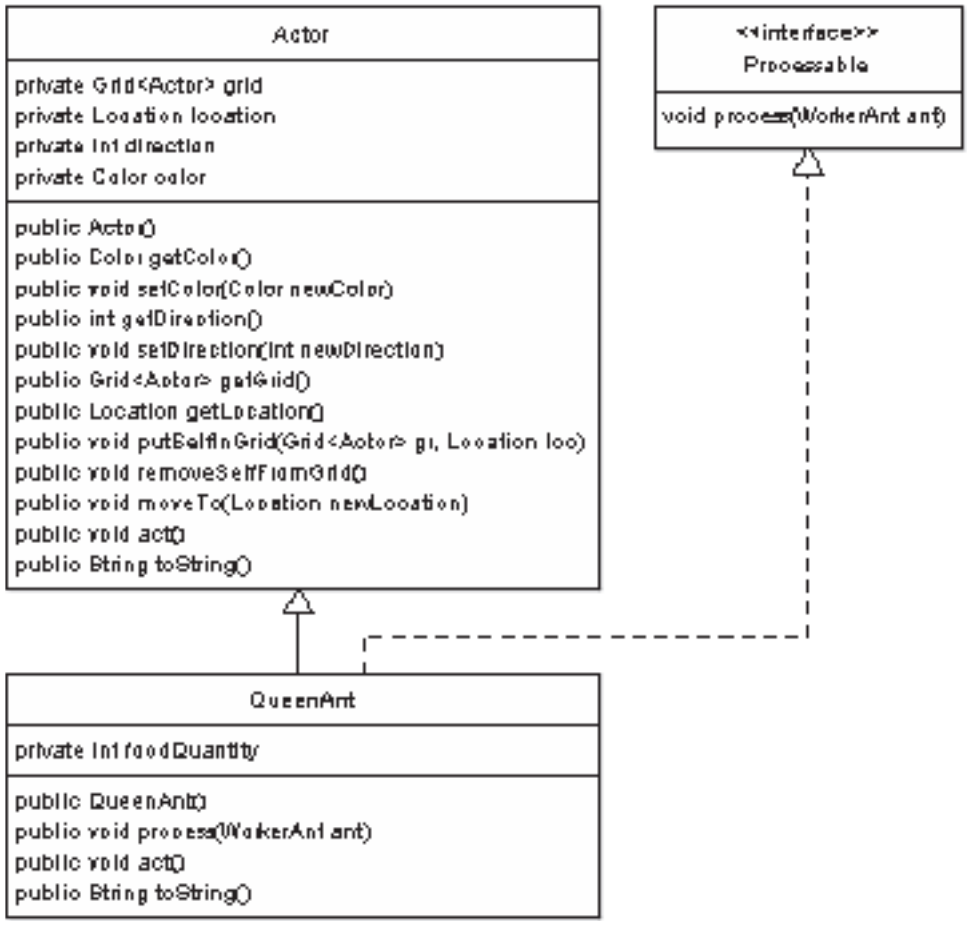


Figure 6 shows the `QueenAnt` class. Queen ants are the simplest of the new Ant Farm objects.

The `QueenAnt` class has one new instance variable (`foodQuantity`) that is used to contain the total amount of food that has been given to the queen by the worker ants. Note that instance variables in Ant Farm, are made `private` to preserve encapsulation.

The `QueenAnt` class has a constructor that initializes `foodQuantity` to `0` and uses the inherited `setColor` method to set the queen's color to `Color.MAGENTA`.

`QueenAnt` implements the `process` method (from `Processable`) to get food from the passed worker ant and to provide it the queen's location. It overrides the `Actor act` method with an empty "do nothing" method (`QueenAnts` don't act).

It also overrides the `Actor` `toString` method to add additional information to the returned string.

This provides a good example of using `super` to call a super class method:

```
return
 super.toString() +
 ", FQty=" + foodQuantity;
```

Food, Cake, and Cookie Classes

Figure 7—Food,Cake, and Cookie Classes

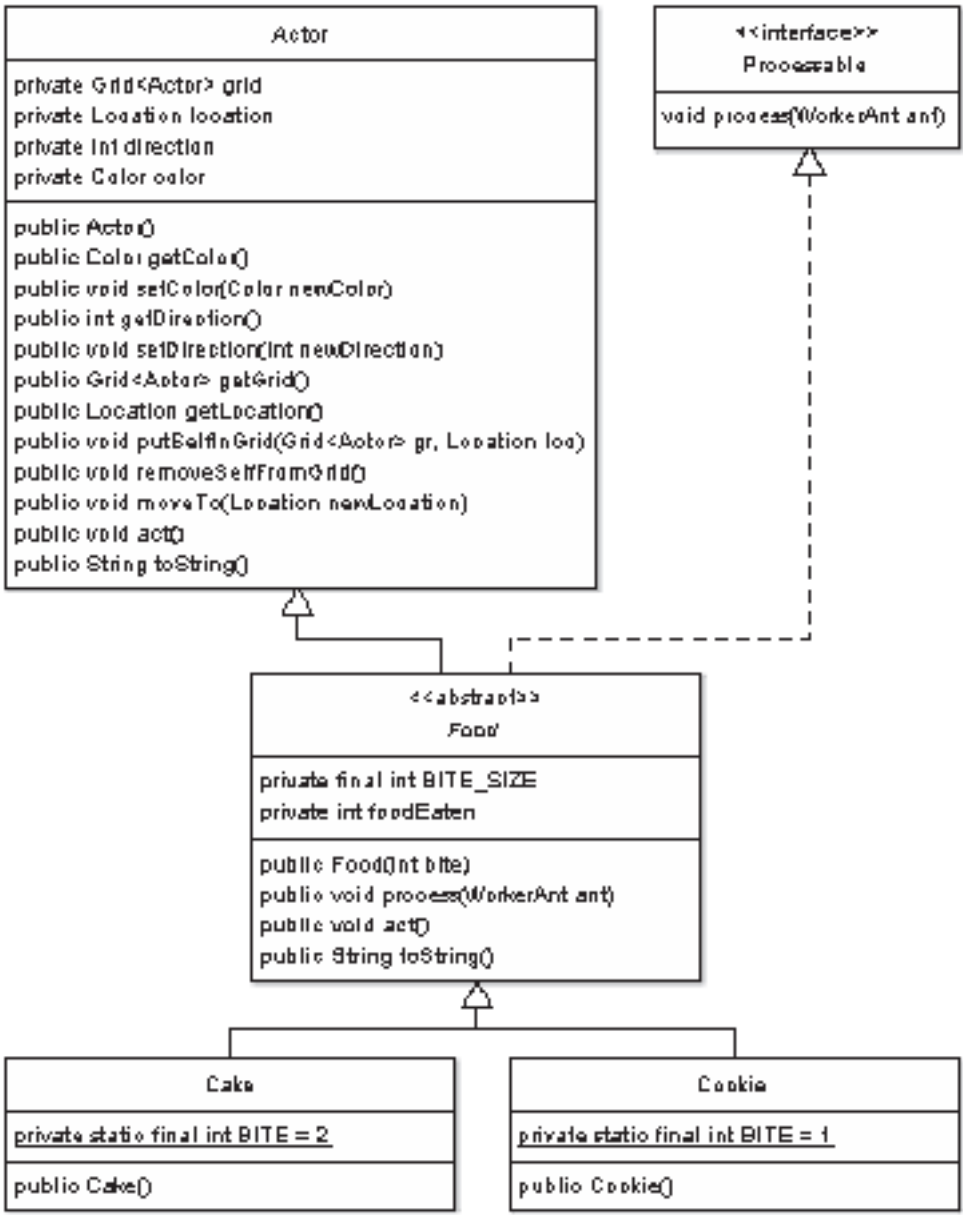


Figure 7 shows the `Food`, `Cake`, and `Cookie` classes. Cake and cookie objects act similarly to queens. However, instead of getting food, they give food.

Different kinds of food are very similar. They differ only by the size of a bite and the displayed image. To take advantage of this similarity, the common instance data and methods are placed in a Food super class. This class contains no abstract methods, but it is declared abstract so that it can not be instantiated. Food contains two instance variables:

- `BITE_SIZE`—a constant that determines how much food is given to a worker ant when it gets food.
- `foodEaten`—keeps track of the total amount of food "given" to worker ants.

The constructor initializes `BITE_SIZE` to the **`bite`** value passed in the parameter, initializes `foodEaten` to 0, and calls `setColor(null)` so that the Cake.gif and Cookie.gif images display with their original coloring.

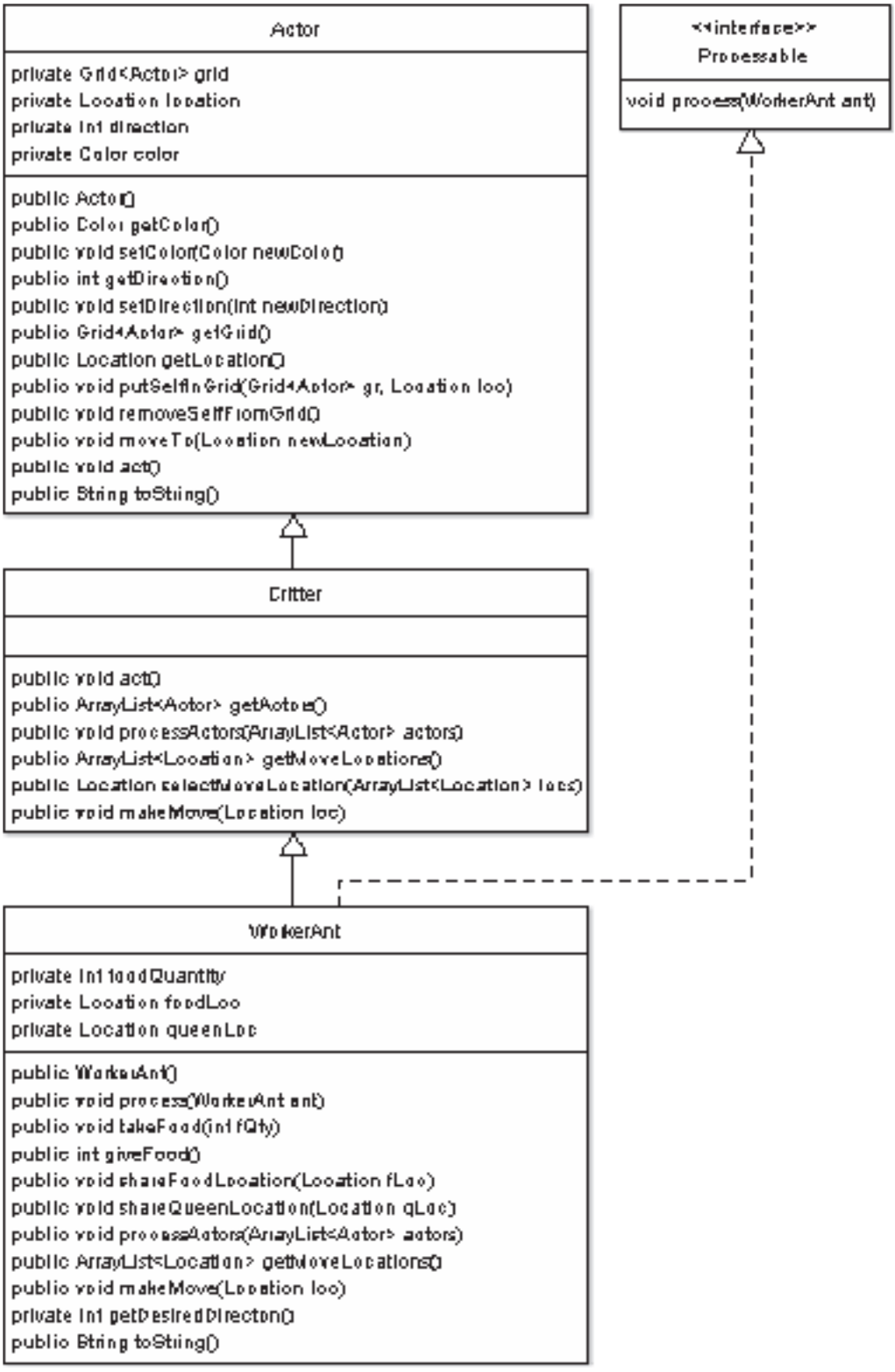
`Food` implements the `process` method (from `Processable`) to give food to the passed worker ant and to provide it the food's location. It overrides the `Actor act` method with an empty "do nothing" method (`Food`s don't act).

It also overrides the `toString` method to include the `BITE_SIZE` and `foodEaten` information.

Because of the `Food` class, the `Cake` and `Cookie` classes are very simple. They contain a single class constant `BITE` which contains the size of a bite. They each have a one statement constructor which passes the value of `BITE` to the `Food` constructor—`super(BITE);`

WorkerAnt Class

Figure 8—WorkerAnt class



`WorkerAnt` (**Figure** 8) is the most complex Ant Farm class. This is to be expected, since it is a`Critter` that interacts with the other objects in the grid.

Worker ants have instance variables to keep track of the amount of food they currently have as well as the locations of the food and the queen.

The constructor initializes these instance variables, makes the ant black, and uses `Math.random` to randomly point the ant in one of the eight valid compass directions.

`WorkerAnt` implements the `process` method to share queen and food locations with other worker ants. `WorkerAnt` has four methods that do the "processing." They are `takeFood`, `giveFood`, `shareFoodLocation`, and `shareQueenLocation`. These methods are called from the `process` methods of the `QueenAnt`, `Food`, and `WorkerAnt` classes.

The `Critter act` method calls the following methods in order:

1. `getActors`—gets a list of actors for interaction.
2. `processActors`—interacts with the list of actors.
3. `getMoveLocations`—gets a list of possible locations for moving.
4. `selectMoveLocation`—chooses one of the possible move locations.
5. `makeMove`—moves.

Worker ants inherit `the Critter act` method, which does the following:

1. Uses the inherited `getActors` to get all the adjacent neighboring actors.
2. `processActors` processes each of the neighboring ant farm actors. This method is a model of elegance and conciseness. The entire method is

```
for (Actor actor : actors)
{
 Processable processor = (Processable)actor;
 processor.process(this);
}
```

An `actor` could be a `QueenAnt`, a `Cake`, a `Cookie`, or a `WorkerAnt`. Without the `Processable` interface, `processActors` would need to determine the type of `actor` and then downcast the `actor` reference before making the call to `process`. But, since each of these classes implements `Processable`, `processActors` only needs to cast the `actor` to `Processable` before the call. This polymorphic processing is allowed because `Processable` contains the `process` method. The Java Run Time Environment (JRE) determines the actual type of object at runtime and calls the appropriate `process` method.

Also note the use of the `this` reference. `this` is a reference to the worker ant executing the `processActors` method, which is exactly the ant that needs to be passed to `process`.

3. `getMoveLocations` does the following:
   a. Calls the private `getDesiredDirecton` method to get the general direction the ant wants to move. This is determined as follows. If
      i. the ant wants food and knows where the food is, then the direction is towards the food.
      ii. the ant has food and knows where the queen is, then the direction is towards the queen.
      iii. neither of the above, then the direction is the ant's current direction.
   b. Creates a list with up to three of the adjacent locations that are in the general direction of the one returned by `getDesiredDirection`**. Locations are included if they meet all of the following criteria. They must be:**
      i. **Adjacent to the current location.**
      ii. **In the desired direction or** 45 degrees to the left and right of the desired direction.
      iii. Valid.
      iv. Empty.
   c. Returns the list of locations.
4. Uses the inherited `selectMoveLocation` to randomly select one of the possible locations. If the list of possible locations is empty, it returns the current location.
5. If the selected move location is different from the current location, `makeMove` moves to the selected location. Otherwise it stays put and changes its direction by randomly choosing between the two directions 45 degrees to the left and right. Then, in either case, it sets its color based on whether it has food (red) or not (black).

`WorkerAnt` also overrides `toString` to include the values of its instance variables.

AntFarmRunner

Like all GridWorld projects, Ant Farm has a "runner" application that is used to set up the initial configuration and show the GUI. `AntFarmRunner` differs from the introductory examples in that it creates its own grid and uses the one parameter `ActorWorld` constructor so that the world will use it.

Modifications

The Ant Farm project is complete. But it can be modified in several ways. Here are a few ideas:

Split Assignment

Since this assignment contains new classes that inherit from both `Actor` and `AntFarmCritter`, it can be split into two assignments. After students read chapter 3 of the GridWorld Narrative, they could be given a revised starter project that includes a working `WorkerAnt.class` file (but no `WorkerAnt.java`). This revised assignment would have students implementing `AntFarmRunner`, `Processable`, `QueenAnt`, `Food`, `Cake`, and `Cookie`.

Later, after completing chapter 4 of the GridWorld Narrative, students could write `WorkerAnt` to complete the project.

A-level Enhancements:

1. Add additional foods. Each of them would extend `Food`.
2. Currently the worker ants and the queen ants don't need the food to survive. The project could be changed so that they use up the food over time and die if they don't receive enough.
3. Currently the food is never totally consumed. Each food could start with an initial number of bites and could "go away" when totally consumed.
4. Additional food objects could be created as the simulation progresses. This would require creating a new world which extends `ActorWorld` and overrides the `step` method.

AB-level Enhancements

1. Currently worker ants vary their directions a little and can navigate around other ants and small obstacles such as rocks. However, they are unable to get around a "wall" of obstacles that would require them taking a longer path.

   Worker ants could be extended to do a breadth-first search for the shortest possible path to the food or queen.

2. Another method of navigating around obstacles would be to use "maps" which would indicate the number of steps required to get to food or a queen. These maps could be created by breadth-first searches initiated in `AntFarmRunner`.

Summary

Ant Farm is a GridWorld project that is appropriate for both A and AB students. As students complete the project, they demonstrate their understanding of the first four chapters of the GridWorld Narrative.

This project utilizes both types of actors. Students inherit from `Actor` for `Food` and `QueenAnt`. Students inherit from `Critter` for `WorkerAnt`, an actor that interacts with the other actors.

As students complete the Ant Farm project, they write an interface, an abstract class, and several concrete classes. They demonstrate inheritance, encapsulation, and polymorphism. The polymorphism is facilitated by the *is-a* relationships created by both class inheritance and interface implementation.

Download

The assignment and starter project can be obtained from http://www.martin.apluscomputerscience.com/gridworld.html. Since this download contains the teacher solution and key, its access is restricted to **teachers only** (Password—abbott). **Please do not share the URL or password with students.** The following files are included:

- Student Files
  - AntFarm Assignment.doc—the assignment
  - AntFarm Student.zip—the starter files (JCreator project)
  - AntFarm Follow-up Questions—questions about the important OOP and GridWorld concepts demonstrated in the Ant Farm project.
- Teacher Files
  - AntFarm Teacher.zip—the completed project
  - AntFarm Follow-up Questions KEY

Reference

As students execute the Ant Farm project, they see an emergent behavior exhibited by the worker ants. Teachers and students may want to explore emergence further at the following URL:

"Emergence." *Wikipedia*. http://en.wikipedia.org/wiki/Emergence

Ant Farm uses an Interface variant of the Template Design Pattern. This can be explored at:

http://en.wikipedia.org/wiki/Template_method_pattern

Special Thanks

I would like to thank

- Cay Horstmann for his helpful design suggestions.
- Chris Renard, one of my students at the School for the Talented and Gifted, for creating the images used in Ant Farm.

Save My Heart

Reg Hahne
Marriotts Ridge High School
Marriottsville, Maryland

This assignment involves the development of a computer simulation that builds from the simple to the complex. As each level of the simulation is implemented, the Actors become more sophisticated in the ways in which they act.

The Actors in this simulation are

Heart—has the same characteristics as a `Rock`. It is a stationary item.
Bacteria—This `Actor` selects the shortest path to the `Heart`.
WhiteBloodCellCritter—interrupts the direct movement of the `Bacteria` to the `Heart`.

Level 0

The simulation at Level 0 contains two Actors: a `Heart` and a `Bacteria`. The `Bacteria`'s goal is to infect the `Heart` by taking the shortest possible path to the `Heart`. The simulation ends when the `Bacteria` lands on a cell that is adjacent to the `Heart`, making the heart turn to the color BLACK. Once the `Heart` turns BLACK, the `act` method has completed its mission and the simulation ends.

Bacteria—Knows the position of the `Heart` and moves in the shortest possible path towards the `Heart`.
Heart—Does not move, but turns BLACK once the `Bacteria` is in an adjacent cell.

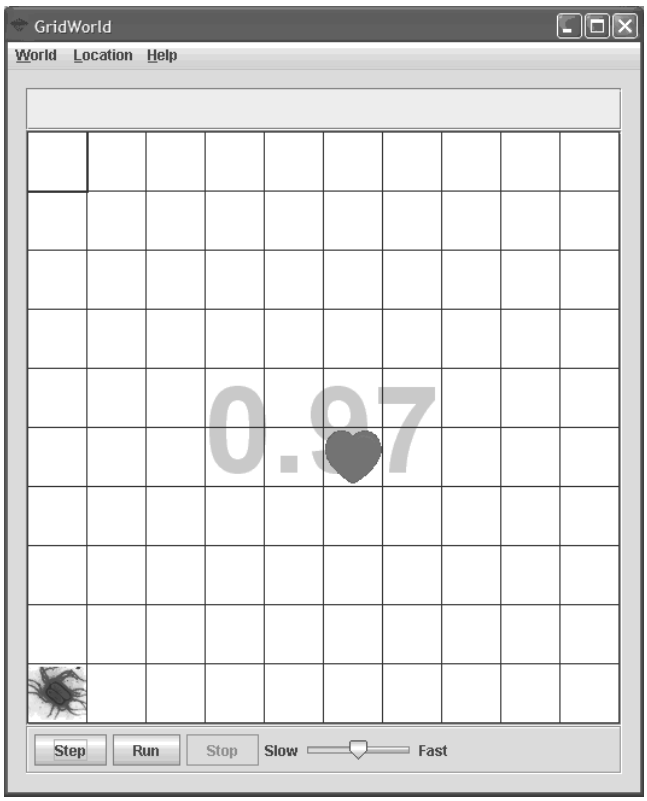
There are two main classes you will need to develop at this level of the simulation; the `Heart` and the `Bacteria`. Let's focus on the `Heart`. The `Heart` extends `Actor` in a similar way that `Rock` extends `Actor`. As the heart is a stationary object, its coding is similar to that of a `Rock`.

Items for consideration

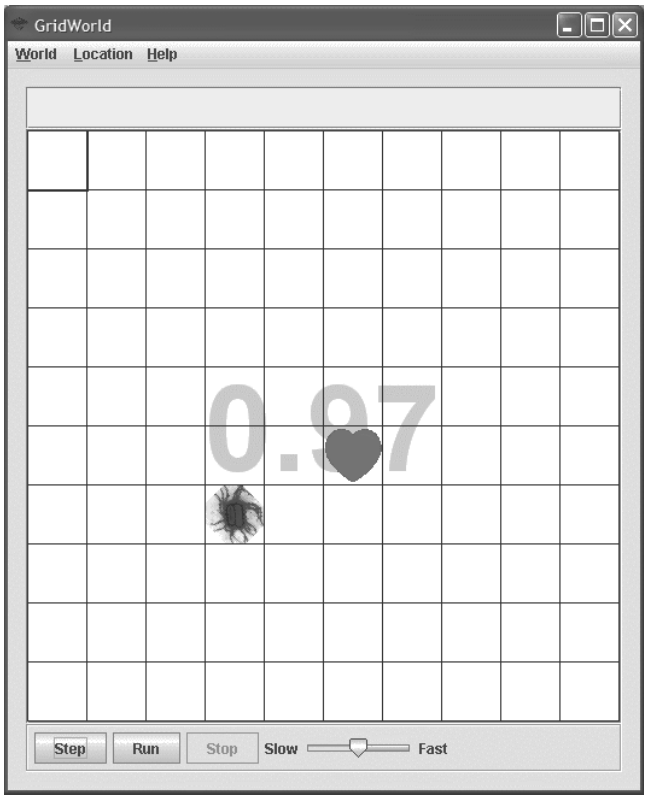
1. Change of color of the `Heart` to BLACK once it has been infected by the `Bacteria`.
2. Use only one constructor (default) as the `Heart`'s color will always be RED at initial runtime.

Graph 0.1 shows initial location of the `Heart` and `Bacteria` prior to run. Graph 0.2 shows the location of the `Bacteria` after several moves towards the `Heart`. Graph 0.3 shows termination.

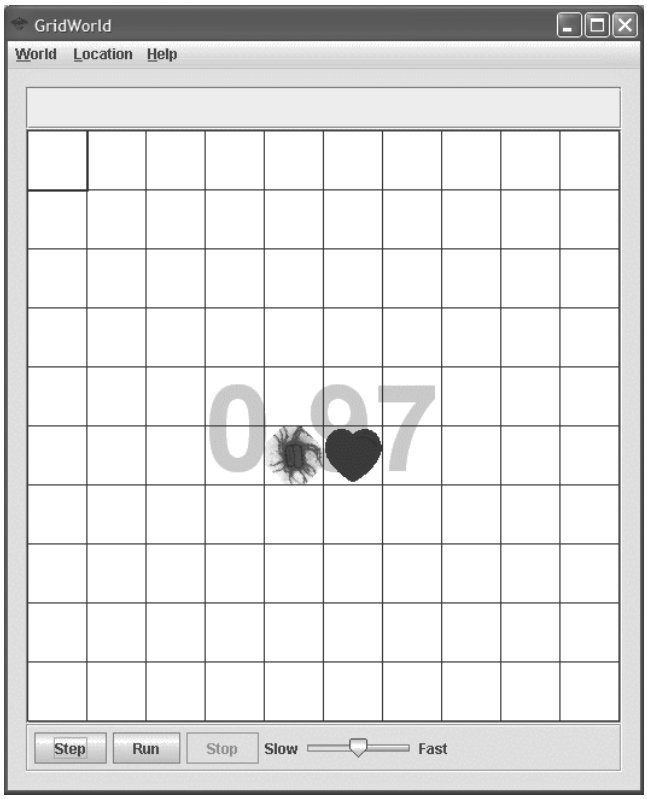
Graph 0.1



Graph 0.2



Graph 0.3



Once the Bacteria has come in contact with the Heart, the Heart changes to BLACK, and the simulation ends.

Bacteria is our next responsibility. All Actors act, and Bacteria is an Actor, therefore the following will be encapsulated within Bacteria's act method. Finding the location of the Heart for the Bacteria to attack and checking if the Bacteria has moved to a location adjacent to the Heart, terminating the simulation and turning the Heart BLACK will all be necessary.

Items for consideration

1. Set the constructor to use a different "bacteria" color.
2. Find the location of the Heart.
3. Obtain locations around the Heart that could terminate the simulation.
4. How to move toward the Heart.
5. Termination.

A suggested algorithm:

```
public void act()
{
      // find Heart location
      //  making sure it's not null
      // get termination locations; those locations
      //  surrounding the Heart
      // if (Heart not found) //not terminating
      //  set direction of the Bacteria
      //  make move toward the Heart
      // else // terminated condition
      //  change Heart to the color BLACK
}
```

The Bacteria constructor should be simple to write. Because Bacteria is an Actor, its constructor can invoke the Actor's setColor method, changing the color of the Bacteria to BLUE.

To find the location of the Heart you will need to create a method getHeartLocation that returns its location.

```
public Location getHeartLocation()
```

To do this you need to scan the entire grid creating a list of objects inhabiting locations. One way to do this is shown below:

```
int col, row = 0;
Location loc = null;
boolean found = false;
while (row < getGrid().getNumRows() && !found)
{
     col = 0;
     while (col < getGrid().getNumCols() && !found)
     {
          loc = new Location (row, col);
          Actor a = getGrid().get(loc);
          if (a != null && a instanceof Heart)
          {
               found = true;
          }
          col++;
     }
     row++;
}
return loc;
```

However, code such as this defeats the purpose of encapsulation and top-down design. You should consider utilizing the `Grid` interface's method `getOccupiedLocations` which allows you to acquire all the objects in the `Grid`. Then, use an enhanced `for` loop to traverse the array of objects, checking for a `Heart`.

```
for (Location loc : inhabitedLocations)
{
     // check if loc is a Heart
}
```

Of course, if the `Heart` is not found then return `null`.

Once the `Heart` has been located, you need to acquire a collection of locations that surround the `Heart` (possibly eight in total). Create a list that will hold all the valid locations that are adjacent to the `Heart`. Once established, check to see if the `Bacteria` is in one of these locations. If it is, the `Heart` is set to BLACK and the `act` method is terminated.

```
public boolean checkForTermination
 (ArrayList<Location> terminationLocations,
 Location BacteriaLocation)
```

If the result of the test is `false` then the `Bacteria` will change direction toward the current location of the `Heart` and move one space forward. This will continue until the `Bacteria` is adjacent to the `Heart`.

Questions to Ponder

1. Why does the simulation cease when a `Bacteria` is in a cell adjacent to the `Heart`?
2. What other classes need to be imported?
3. Why is it a good idea to set the direction of an `Actor` before making it move?
4. Change program cessation from being "in a location next to the `Heart`" to being "in a location the same as the `Heart`," in fact, removing the `Heart`. List the programming changes that need to be made, and then code your solution.

Level 1

The simulation at Level 1 contains three `Actors:` a `Heart`, a `Bacteria` and a `WhiteBloodCellCritter`.

The `Bacteria`'s goal is to infect the `Heart` by taking the shortest possible path to the `Heart`. The simulation ends (`Heart` turns BLACK) when the `Bacteria` lands on a cell that is adjacent to the `Heart`. This is similar to the situation for termination in Level 0. An additional skill that the `Bacteria` possesses is an ability to become aware of a `WhiteBloodCellCritter` and change direction.

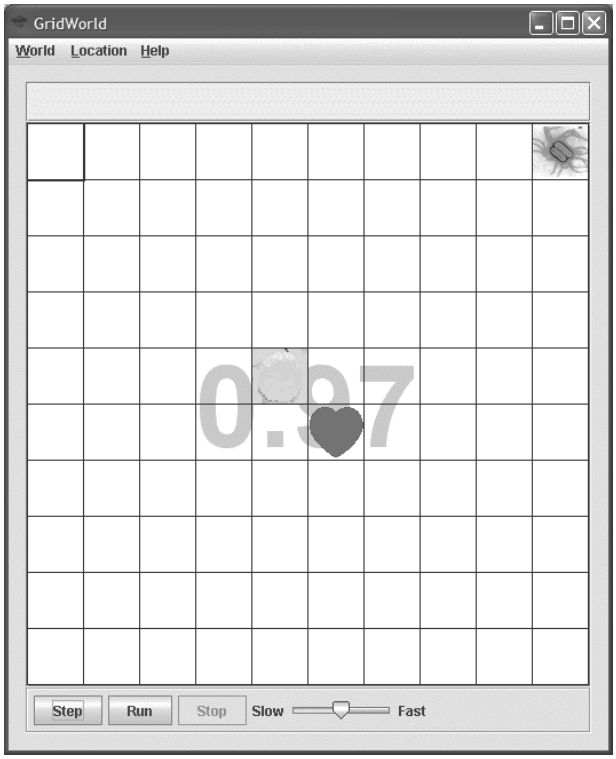
WhiteBloodCellCritter—Moves randomly within a two cell range within a 5 × 5 cell grid with the `Heart` at its center.

Bacteria—Knows the position of the `Heart` and moves in the shortest possible path towards it. In addition, the `Bacteria` can "see" two grid blocks in any direction from its current position. If the `Bacteria` is within two grid blocks of the `WhiteBloodCellCritter`, the `Bacteria` moves one grid block in the opposite direction from the location of the `WhiteBloodCellCritter`.

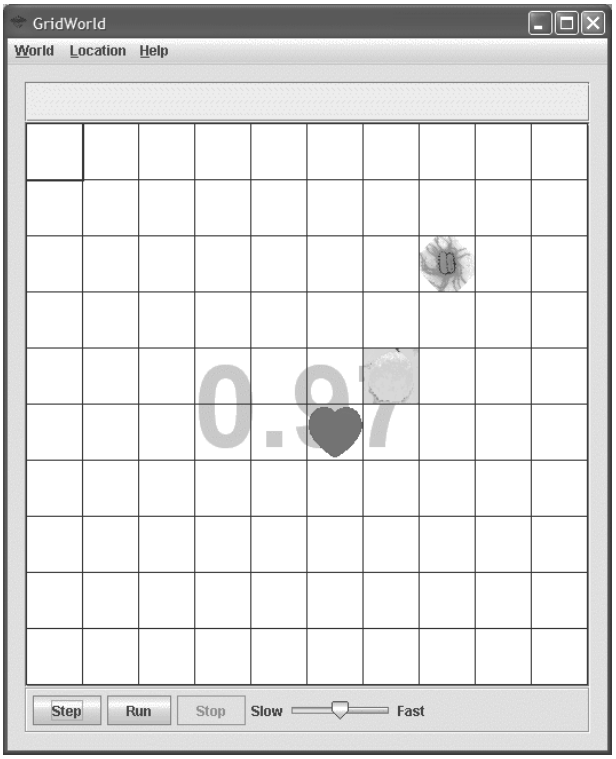
Heart—Does not move, but turns BLACK once the `Bacteria` is in an adjacent cell (as in Level 0).

Below are several phases of a graphical simulation demonstrating how the `Bacteria` and `WhiteBloodCellCritter` interact.

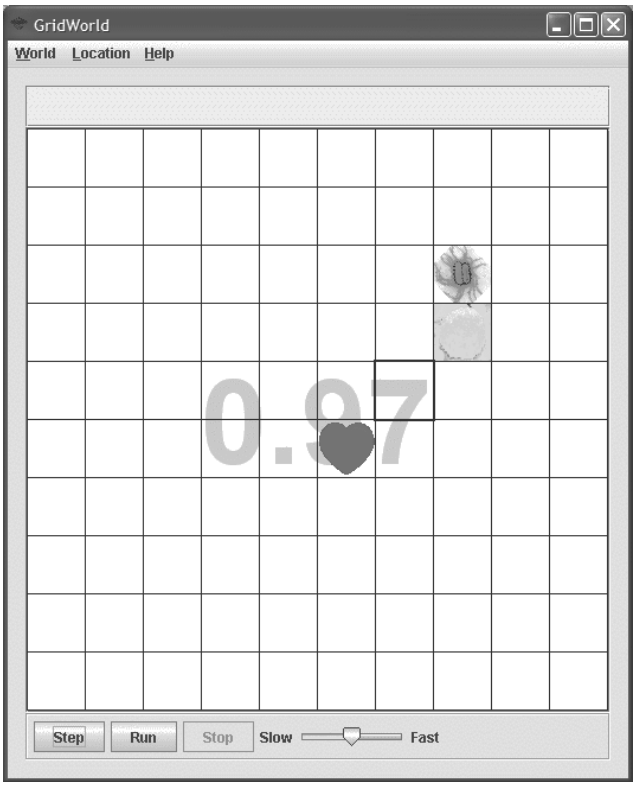
Graph 1.1



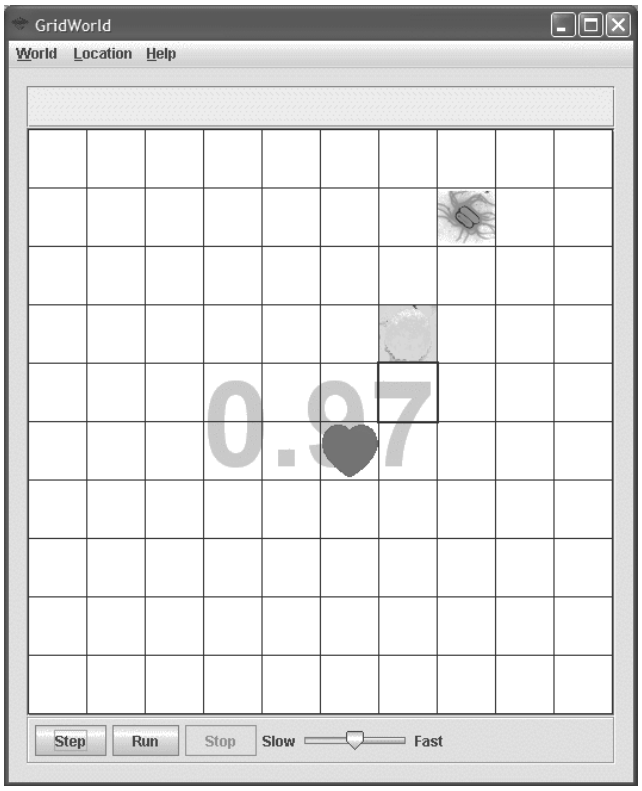
Graph 1.2



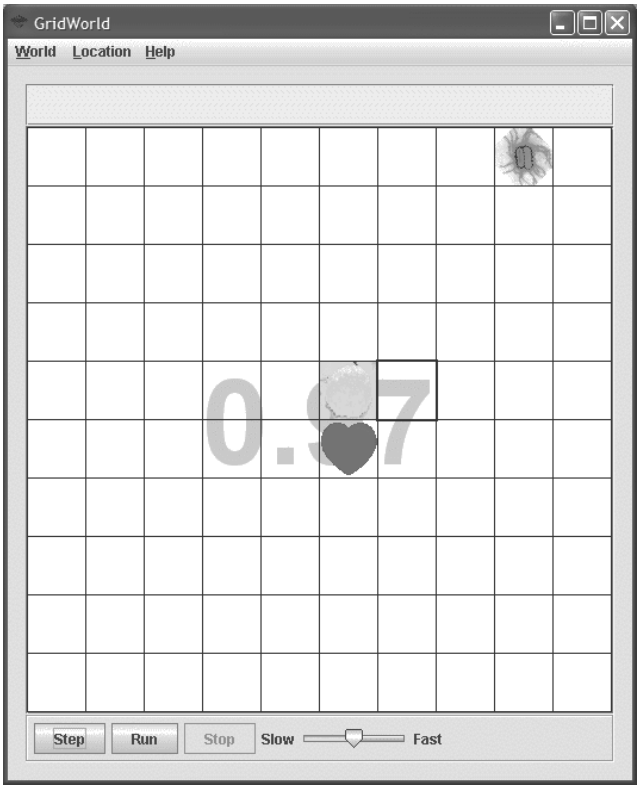
Graph 1.3



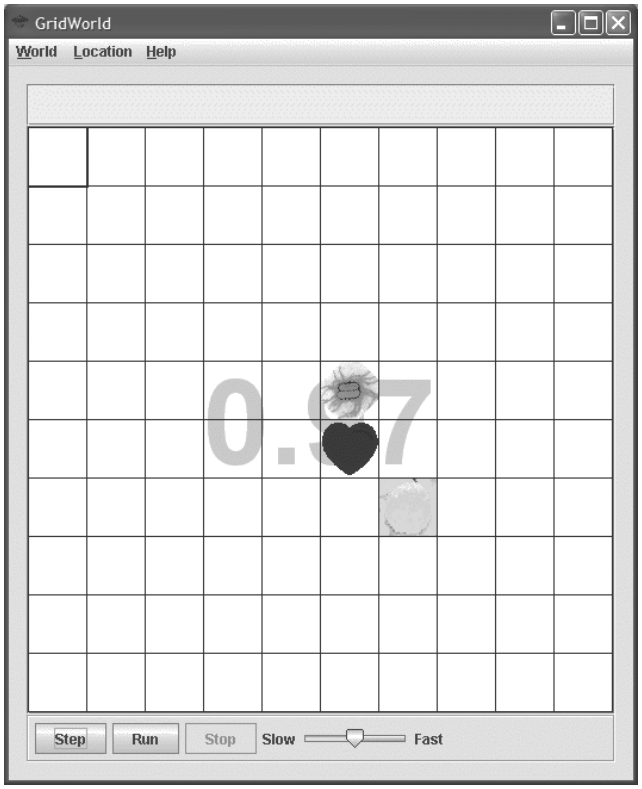
Graph 1.4



Graph 1.5



Graph 1.6



Graph 1.1 shows initial location of the `Heart, Bacteria,` and `WhiteBloodCellCritter` prior to run. Graph 1.2 shows the location of the `Heart, Bacteria` and `WhiteBloodCellCritter` after two steps of the simulation. After many steps of the simulation Graph 1.3 shows the location of the `Heart, Bacteria` and `WhiteBloodCellCritter`. The resolution of this encounter is shown in Graph 1.4 and then Graph 1.5. Graph 1.6 shows termination.

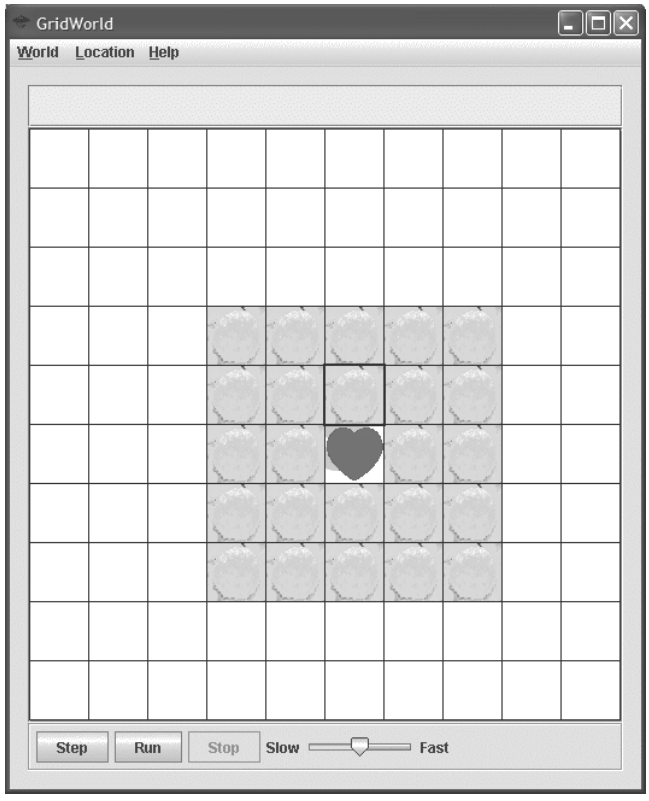
Items for consideration

There are three main classes to consider at this level of the simulation:

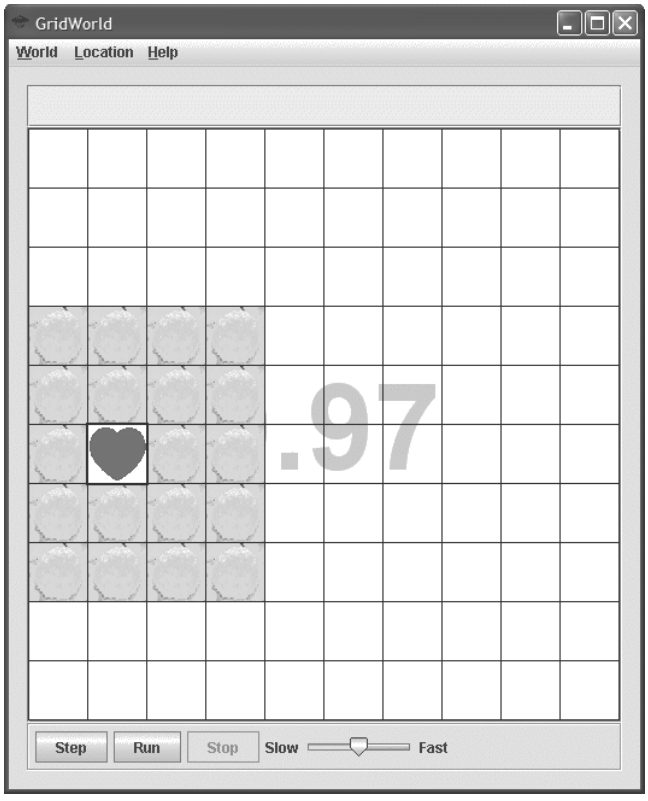
1. The `Heart` will not change location.
2. A new class, `WhiteBloodCellCritter`, needs to be created.
3. The `Bacteria` will need to be modified to consider the position of the `WhiteBloodCellCritter`.

Let's look at the creation of the `WhiteBloodCellCritter` class. A `WhiteBloodCellCritter` will be able to move within a 5 × 5 cell grid around the `Heart`. Graph 1.7 and Graph 1.8 show possible `WhiteBloodCellCritter`'s realm of movement.

Graph 1.7



Graph 1.8



Because a WhiteBloodCellCritter is a Critter, its movement should be implemented by overriding the methods that are called by Critter's act method. Remember the act method of the Critter class calls five other methods.

```
public void act()
{
     if (getGrid() == null)
          return;
     ArrayList<Actor> actors = getActors();
     processActors(actors);
     ArrayList<Location> moveLocs = getMoveLocations();
     Location loc = selectMoveLocation(moveLocs);
     makeMove(loc);
}
```

The WhiteBloodCellCritter constructor should be simple. Because WhiteBloodCellCritter is a Critter and Critter is an Actor, the constructor can invoke the Actor's setColor method, setting the color of a WhiteBloodCellCritter to YELLOW.

The getActors and processActors methods are implemented to do nothing.

The most challenging method to write is getMoveLocations. This method returns an ArrayList of possible locations that the WhiteBloodCellCritter can legally move. Remember the restrictions placed upon the WhiteBloodCellCritter. Below is the method header and pseudo-code outlining a possible process.

```
public ArrayList<Location> getMoveLocations()
{
     // get the empty adjacent cells
     // remove locations that are outside its realm
     // return the list
}
```

To obtain the empty locations, create an ArrayList of possible moves using the getEmptyAdjacentLocations method found in AbstractGrid. Once obtained, you will need to remove all the invalid locations. Method removeInvalidLocations outlines an implementation plan.

```
public ArrayList<Location> removeInvalidLocations
     (ArrayList<Location> locations)
{
     // make a new ArrayList
     // place only valid items in this list
     //   need to check if locations are valid
}
```

This plan needs to remove locations that are outside of the realm of the WhiteBloodCellCritter's movement. To do this requires the return of an ArrayList of valid locations. These valid locations depend on the location of the Heart. By using the getHeartLocation method from the Bacteria class, we will be able to process the current ArrayList, removing additional invalid locations and returning a list of possible valid locations to which the WhiteBloodCellCritter can move. To accomplish this task a method withinHeartRange needs to be developed.

```
public boolean withinHeartRange(Location loc,
     Location heartLocation)
{
     // returns true if loc is within a valid distance of
     // the location of the Heart
}
```

Once getMoveLocations has returned the ArrayList of possible valid moves, the Critter's selectMoveLocation and makeMove methods will be called.

Now you need to address the Bacteria and the new actions that it takes in relation to the location of the WhiteBloodCellCritter. There are several methods that the current Bacteria have, and some additional methods to write. It would be prudent to inherit all the current Bacteria methods and create a new class Bacteria1 that extends Bacteria.

```
public class Bacteria1 extends Bacteria
```

Items for consideration

1. Set the constructor to use a different "bacteria1" color.
2. Find the location of the Heart and WhiteBloodCellCritter.
3. Check to see if close to a WhiteBloodCellCritter.
4. Set reverse direction.
5. Move like a Bacteria.

The Bacteria1 constructor should be simple. Because Bacteria1 is a Bacteria you can invoke the Bacteria constructor by using super to call the parent class's constructor. Make sure you pass the color GREEN as a parameter in this call.

As before, the act method needs to be modified to accommodate these new requirements. This modification requires you to redesign Level 0 of the act method. Use the following steps in the redesign of your new act method.

```
public void act()
{
     // find Heart and WhiteBloodCellCritter locations
     //   make sure neither is null
```

```
            // check to see if close to a white blood cell
        //   if so next move is retreat
        //        move in the reverse direction
        //   else next move is attack
        //        move in the direction of the Heart
        //             but check for termination
    }
```

Create the `getWhiteBloodCellLocation` method by mimicking the `getHeartLocation` method found in the `Bacteria` class.

As you know, when a `Bacteria1` discovers the existence of a `WhiteBloodCellCritter`, it moves in the opposite direction from the `WhiteBloodCellCritter`. To facilitate the discovery of the `WhiteBloodCellCritter`, the method `closeToWhiteBloodCell` needs to be created. Use the following header for this method:

```
public boolean closeToWhiteBloodCell()
{
     // get the white blood cell location
     // return true if detected a white blood cell
     // else return false
}
```

To help detect a `WhiteBloodCellCritter` you will need to write the method `getValidLocations` that will return an `ArrayList` of valid locations that the `Bacteria1` can search.

```
public ArrayList<Location> getValidLocations(Location bacteria)
```

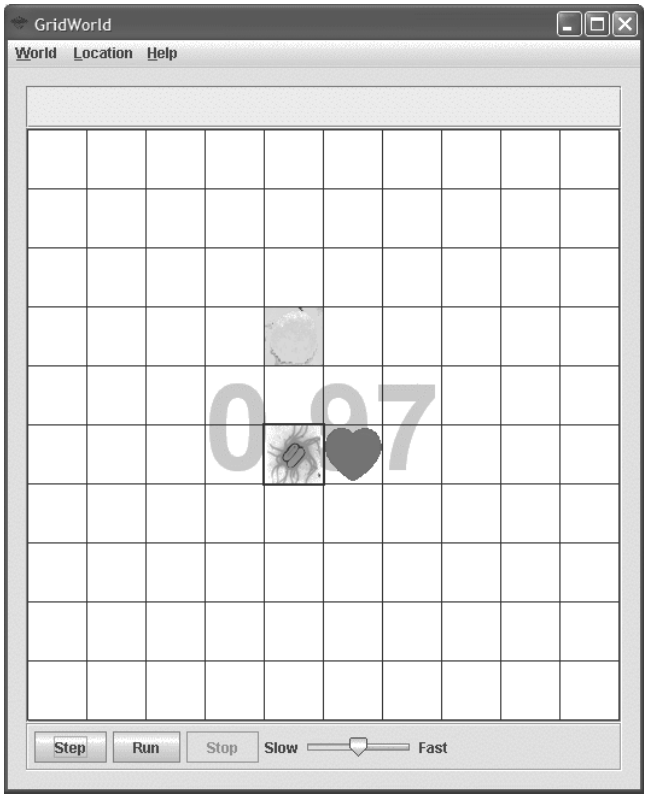
Remember this area is a 5 × 5 cell grid surrounding the `Bacteria1`. Those valid locations need to be compared to the location of the `WhiteBloodCellCritter` and if within range move accordingly. This movement is in the reverse direction away from the `WhiteBloodCellCritter`. If no contact with a `WhiteBloodCellCritter` is discovered to be true, then the `Bacteria1` moves like a regular `Bacteria`. This code can be found in Level 0 of the simulation.

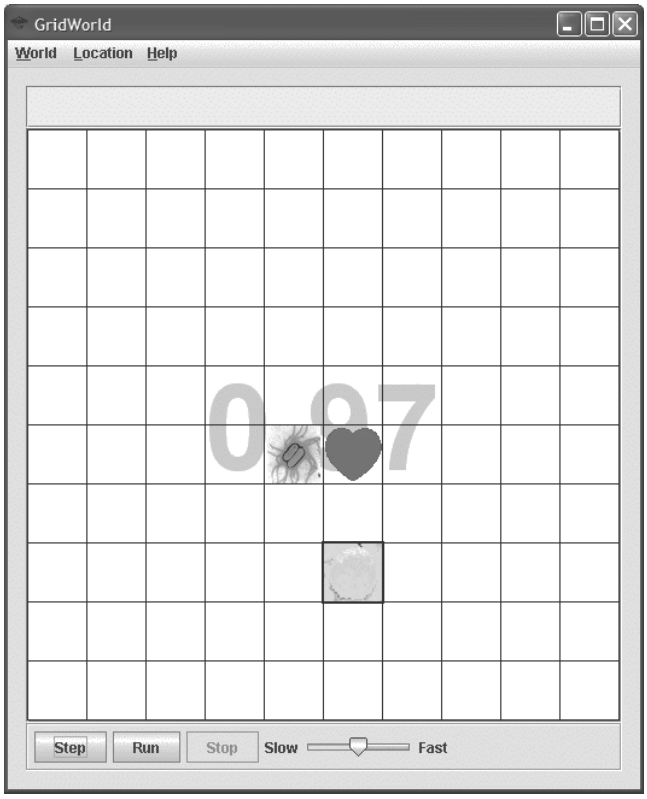
Once the `getMoveLocations` has returned the `ArrayList` of possible valid moves, the `Critter`'s `selectMoveLocation` and `makeMove` methods will be called.

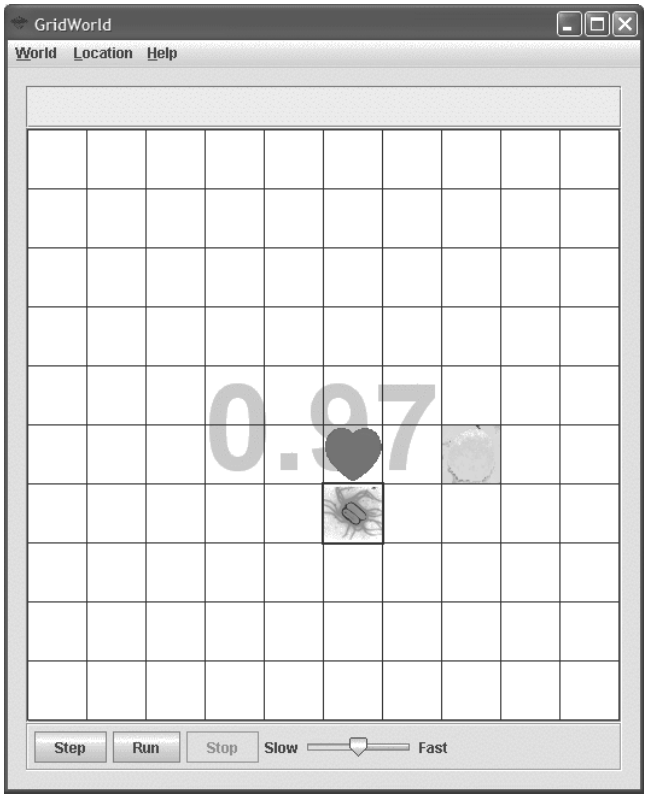
Questions to Ponder

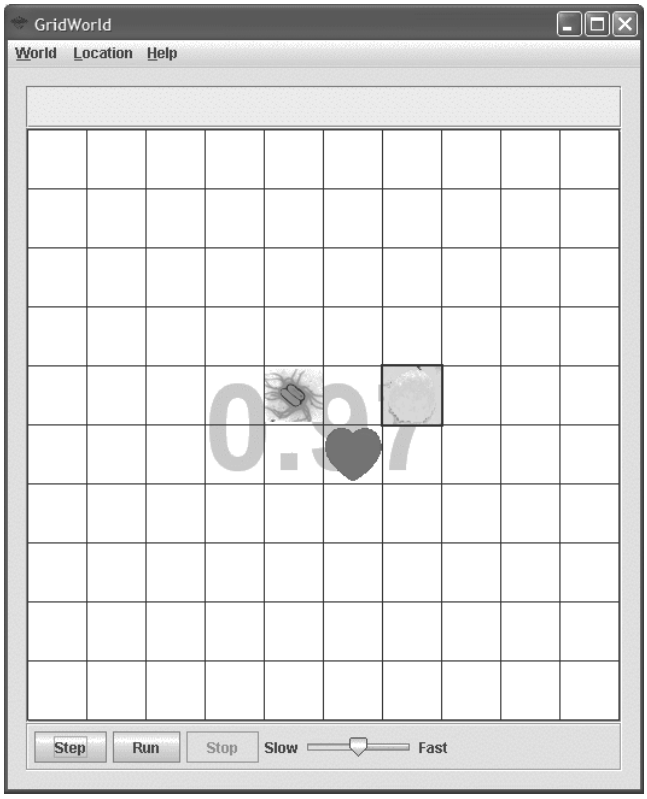
1. Why does the simulation appear not to work correctly for some situations? For example, looking at the simulations below one would think that termination should take effect on the next step, however it does not. What possible reasons could make your `act`

method work in such a way? The following four simulation situations do not terminate correctly. How can this be corrected? List the steps needed to support your reason(s).









2 Why does the following code not work?

```
for (Location loc : locations)
{
      if (validLocation (loc, getHeartLocation()))
      {
           tempHold.add(temp);
      }
}
```

3. Why create a new array in `removeInvalidLocations` in the `WhiteBloodCellCritter` method?
4. Once the `Heart` turns BLACK the simulation should end. Why does the simulation continue?
5. Why is it a good idea to enlarge the grid size to 12 x 12? How can this be accomplished?
6. (AB) Another way to implement a method to obtain valid locations is to modify the method `getValidAdjacentLocations` in `AbstractGrid` allowing the creation of a list of locations in the area within which the `WhiteBloodCellCritter` can move.
7. (A or AB) See if you can develop another way to implement the way in which valid locations for the `WhiteBloodCellCritter` can be found.

Level 2

The simulation at Level 2 contains three `Actors`: a `Heart`, a `Bacteria` and a `WhiteBloodCellCritter`.

The `Bacteria`'s goal is to infect the `Heart` by taking the shortest possible path to the `Heart`. The simulation ends (heart turns BLACK) when the `Bacteria` lands on a cell that is adjacent to the `Heart`. This is similar to the situation to termination in Level 0. The `Bacteria` possesses an ability to become aware of a `WhiteBloodCellCritter`, and if so, changes direction. This situation existed in Level 1 of the simulation. In addition, the `WhiteBloodCellCritter` will attack (move toward) a `Bacteria` if the `Bacteria` is within a two-cell distance of a `WhiteBloodCellCritter`. This attack is represented by the `WhiteBloodCellCritter` moving in the direction of the `Bacteria`, but is unable to travel further than the two-cell distance limitation placed upon it in Level 1 of the simulation.

WhiteBloodCellCritter—Moves randomly within a two-cell range within a 5 × 5 cell grid with the `Heart` at its center. The `WhiteBloodCellCritter` can "see" two grid blocks in any direction from its current position. If the `Bacteria` is within two grid blocks of the `WhiteBloodCellCritter`, the `WhiteBloodCellCritter` attacks the `Bacteria` by moving one grid block in the direction of the `Bacteria`.

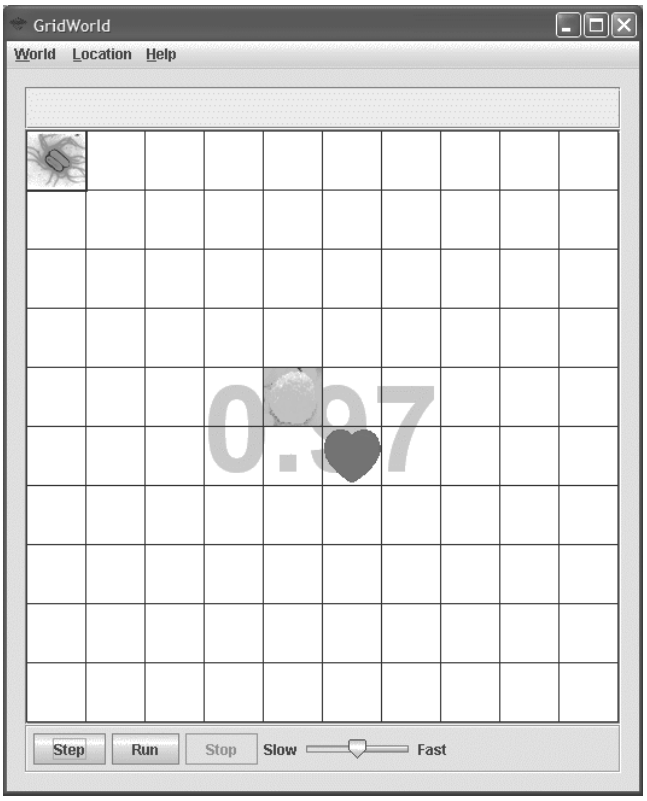
Bacteria—Knows the position of the `Heart` and moves in the shortest possible path toward it. In addition, the `Bacteria` can "see" two grid blocks in any direction from its current position. If the `Bacteria` is within two grid blocks of the `WhiteBloodCellCritter` the `Bacteria` moves one grid block in the opposite direction away from the location of the `WhiteBloodCellCritter`.

Heart—Does not move, but turns BLACK once the `Bacteria` is in an adjacent cell.

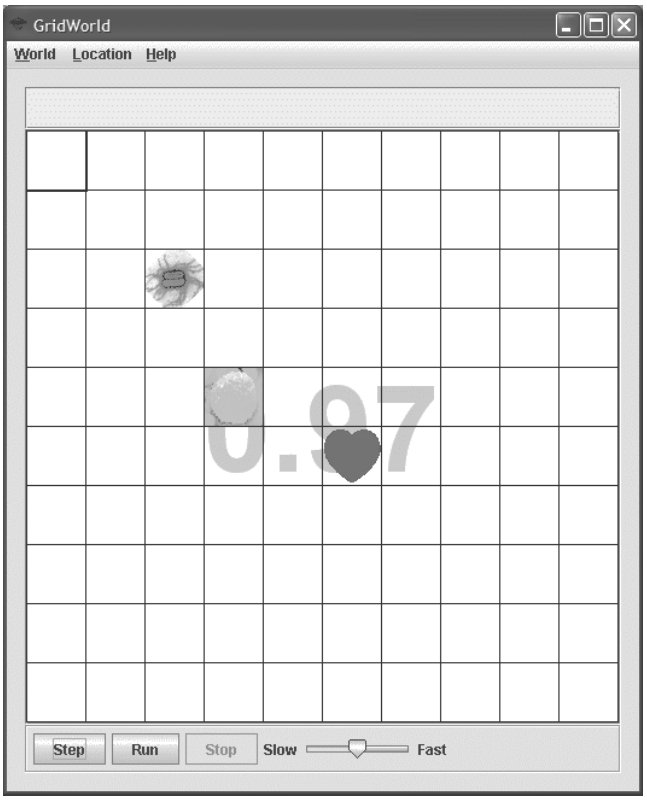
Below are several phases of a graphical simulation demonstrating how the `Heart`, `Bacteria` and `WhiteBloodCellCritter` interact.

Graph 1.9 shows initial location of the `Heart`, `Bacteria1` and `WhiteBloodCellCritter` prior to run. Graph 1.10 shows the location of the `Heart`, `Bacteria1` and `WhiteBloodCellCritter` after two steps of the simulation. The resolution of this encounter is shown in Graph 1.11 and Graph 1.12.

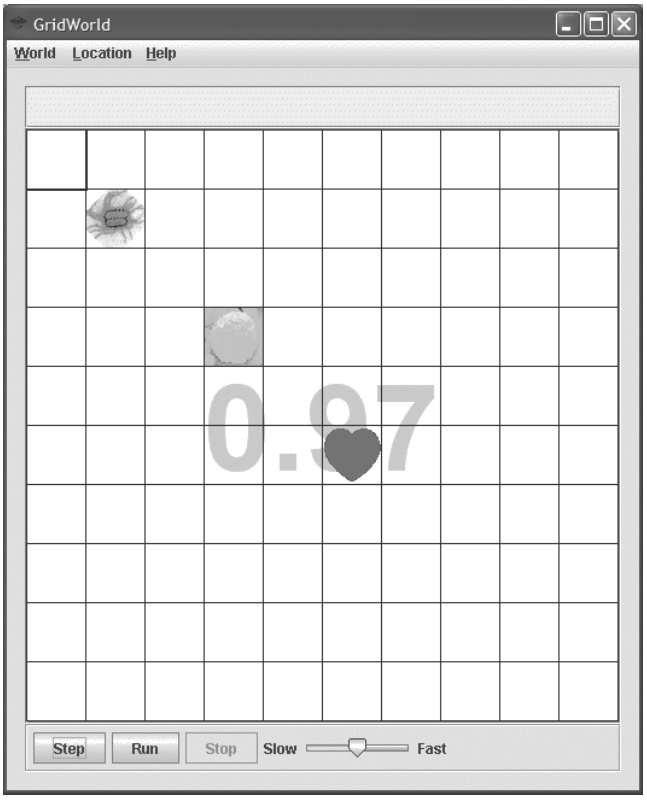
Graph 1.9



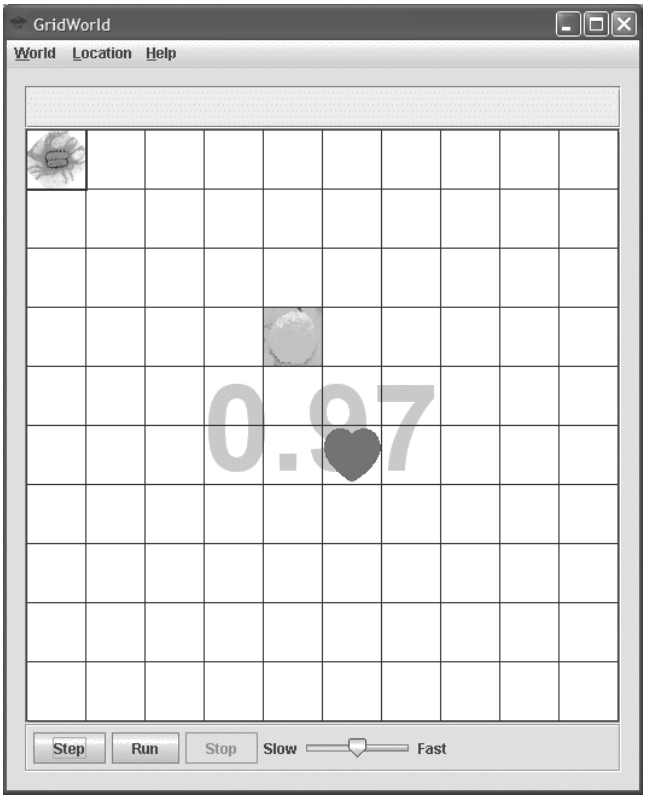
Graph 1.10



Graph 1.11



Graph 1.12



Addressing the new actions of the `WhiteBloodCellCritter` reveals many similar traits from its Level 1 implementation. It would be prudent to utilize existing methods and write new ones for the additional responsibilities that the `WhiteBloodCellCritter` has been given. To do this, inherit all the current `WhiteBloodCellCritter` methods and create a new class `WhiteBloodCellCritter2` that extends `WhiteBloodCellCritter`.

```
public class WhiteBloodCellCritter2 extends
     WhiteBloodCellCritter
```

Items for consideration

1. Set the constructor to use a gentle "whiteBloodCellCritter2" color.
2. Find the location of the `Heart` and `WhiteBloodCellCritter2`.
3. Check to see if close to a `Bacteria1`.
4. Move in `Bacteria1`'s direction or move like a `WhiteBloodCellCritter` as in Level 1.

Because a `WhiteBloodCellCritter2` is a `Critter`, its movement should be implemented by overriding the methods that are called by `Critter`'s `act` method. Remember the `act` method of the `Critter` class calls five other `Critter` methods.

```
public void act()
{
     if (getGrid() == null)
          return;
     ArrayList<Actor> actors = getActors();
     processActors(actors);
     ArrayList<Location> moveLocs = getMoveLocations();
     Location loc = selectMoveLocation(moveLocs);
     makeMove(loc);
}
```

The `WhiteBloodCellCritter2` constructor should be simple. Because `WhiteBloodCellCritter2` is a `WhiteBloodCellCritter` and `WhiteBloodCellCritter` is an `Actor`, its constructor can invoke the `Actor`'s `setColor` method, setting the color of a `WhiteBloodCellCritter2` to ORANGE.

`WhiteBloodCellCritter`'s `getActors` and `processActors` methods were implemented to do nothing, so we don't need to override them here.

The most challenging method to write is `getMoveLocations`. This method returns an `ArrayList` of possible locations that the `WhiteBloodCellCritter2` can legally move. There are now two scenarios that the `WhiteBloodCellCritter2` must take

into account before movement takes place: when within range of a `Bacteria1`, or when not within range of a `Bacteria1`. When not within range of a `Bacteria1` is the easier code to write and is the same as that used for the `WhiteBloodCellCritter2` in Level 1 of the simulation. When within range of a `Bacteria1` there are more things to consider. Below is the method header and pseudo-code outlining a possible process.

```
public ArrayList<Location> getMoveLocations()
{
     // if close to a Bacteria1
     //  find the direction toward the Bacteria1
     //  create an array of valid locations
     // else
     //  get the empty adjacent cells
     //  remove locations that are outside its realm
     //  create an array of valid locations
     // return the array
}
```

Let's first look at what the `WhiteBloodCellCritter2` will do if it moves within range of a `Bacteria1`, or if a `Bacteria1` moves within range of a `WhiteBloodCellCritter2`. Either situation should produce similar results, i.e., the `WhiteBloodCellCritter2`'s next move being toward the `Bacteria1`.

If the `WhiteBloodCellCritter2` is within range of a `Bacteria1` the `WhiteBloodCellCritter2` will find the direction from its location toward the `Bacteria1`. Writing a `getBacteriaLocation` method that is very similar to `getHeartLocation` is needed.

```
public Location getBacteriaLocation()
```

Once you have obtained its location the next step is to ascertain if the `WhiteBloodCellCritter2` is within range of the `Bacteria1`. The method `bacteriaDetected` detects if the `WhiteBloodCellCritter2` is within range of the `Bacteria1`.

```
public boolean bacteriaDetected(Location whiteBloodCell,
                                Location bacteria)
```

The next task is to find the valid locations the `WhiteBloodCellCritter2` can move to, returning true if the location parameter is a valid location that then gets added to the `moveLocs` array. This can be overridden from Level 1.

```
public boolean validLocation(Location loc)
```

If the `WhiteBloodCellCritter2` is not within range of a `Bacteria1` then it moves the same as it did in Level 1 of the simulation.

Once the `getMoveLocations` has returned the `ArrayList` of possible valid moves the `Critter`'s `selectMoveLocation` and `makeMove` methods will be called.

Questions to Ponder

1. Think about other ways the `Bacteria1` could move once it "sees" a `WhiteBloodCellCritter2.`
2. How could you improve the chances of the `Bacteria1` gaining quicker access to the `Heart`, or the `WhiteBloodCellCritter2` defending the `Heart`?

Level 3

The simulation at Level 3 contains three `Actors`: a `Heart`, a `Bacteria` and a `WhiteBloodCellCritter`.

The `Bacteria`'s goal is to infect the `Heart` by taking the shortest possible path to the `Heart`. The simulation ends (heart turns BLACK) when the `Bacteria` lands on a cell that is adjacent to the `Heart`. This is similar to the situation for termination in Level 0. The `Bacteria` possesses an ability to become aware of a `WhiteBloodCellCritter2`, and if so, changes direction by a set pattern and deploys a decoy of itself, hoping that the `WhiteBloodCellCritter2` will chase this new entity. This existed in Level 1 of the simulation. The `WhiteBloodCellCritter2` will act as it did in Level 2 of the simulation.

WhiteBloodCellCritter—Moves randomly within a two-cell range; a 5 × 5 cell grid with the `Heart` at its center. The `WhiteBloodCellCritter2` can "see" two grid blocks in any direction from its current position. If the `Bacteria` is within two grid blocks of the `WhiteBloodCellCritter2` the `WhiteBloodCellCritter2` attacks the `Bacteria` by moving one grid block in the direction of the `Bacteria` (as in Level 2).

Bacteria—Knows the position of the `Heart` and moves in the shortest possible path toward it. In addition, the `Bacteria` can "see" two grid blocks in any direction from its current position. If the `Bacteria` is within two grid blocks of the `WhiteBloodCellCritter2`, the `Bacteria` moves in a predefined pattern in relationship to the `WhiteBloodCellCritter2` (as in Level 1) while deploying a decoy of itself.

Heart—Does not move, but turns BLACK once the `Bacteria` is in an adjacent cell (as in previous levels).

This assignment and accompanying files may be found at
http://www.apcomputerscience.com/gridworld

- student assignment handouts
- source code for complete solution (all levels)
- graphics
- a detailed narrative for two additional levels (AB extensions):
  - Level 4: Similar to Level 3. The data structures used to simulate the game could be a PriorityQueue, Map or Set. In fact it would be interesting to analyze which is the best to use.
  - Level 5: Implements many bacteria and white blood cell critters—a real challenge.

About the Authors

Chief Editor

Debbie Carter teaches AP Computer Science and assists faculty with technology integration at Lancaster Country Day School in Lancaster, Pennsylvania. She has served as both a Reader and Question Leader for the AP Computer Science Exams and has been a College Board consultant since 1996. She currently sits on the board of the Computer Science Teachers Association.

Authors

Instructional Unit: Save My Heart
Reg Hahne is Instructional Team Leader of Career Technology Education (CTE) at Marriotts Ridge High School in Maryland. Reg was an AP Computer Science Exam Reader for many years and has served on the AP Computer Science Development Committee.

Article: The Design of the GridWorld Case Study
Cay Horstmann is a professor in the Department of Computer Science at San Jose State University, California, and author of many popular computer science text books. Cay is presently a member of the AP Computer Science Development Committee.

Instructional Unit: Early Exercises for GridWorld
Judith Hromcik is an AP Computer Science teacher at Arlington High School in Arlington, Texas. Judith previously served on the AP Computer Science Development Committee and has been a Reader and a Question Leader at the AP Computer Science Exam Reading. She has been a College Board consultant since 1997 and has conducted many AP Computer Science Summer Institutes. Judith wrote the solutions for the new GridWorld case study and has pilot-tested GridWorld in her classes. She has a strong interest in developing curriculum for computer science and technology applications.

Article: Integrating GridWorld
Jill Kaminski has been privileged to be a computer science teacher since 1999. Prior to that, she was a software engineer for over 12 years, working primarily on firmware for various spacecraft projects. She teaches in both traditional and online environments, and has been an AP Computer Science Exam Reader. She lives in Colorado with her husband.

Instructional Unit: Ant Farm
Robert Glen Martin has been an AP Computer Science teacher since 2000. He has been teaching at the School for the Talented and Gifted (*Newsweek*'s 2006 Best Public High School) since 2002. He is an AP Computer Science Exam Reader and a College Board consultant. Glen's computer science program has been recognized every year since 2005 by the College Board as the small school leader in helping the widest segment of total school

population attain college-level mastery of AP Computer Science A and AB. He was also named the 2005–06 Siemens Advanced Placement Teacher of the Year for Texas.

Instructional Unit: Board Games
Dave Wittry is an AP Computer Science teacher at the Taipei American School in Taiwan. Dave has been an AP Computer Science Exam Reader for many years and a College Board consultant since 2005. He has led many AP Computer Science Summer Institutes over the past 5 years. Dave is an exam contributor for *Be Prepared for the AP Computer Science Exam in Java* by Maria Litvin. He moved to Taiwan in 2005 with his wife, Jody, and two young children, Kaylin and Darren.


F07CSSF125