Chapter 3

Pentax 67 / 50mm Lens / Kodak Tri-X Professional

My Backyard

Project Walkthrough

Learning Objectives

- Apply the project-approach strategy to implement a C# programming assignment
- Apply the development cycle to implement a C# programming assignment
- State the actions performed by the development roles of analyst, designer, and programmer
- Translate a project specification into a software design that can be implemented in C#
- State the purpose and use of method stubbing
- State the purpose and use of state transition diagrams
- Explain the concept of data abstraction and the role it plays in the design of user-defined data types
- Use the csc command-line tool to compile C# source files
- Execute C# programs
- State the importance of compiling and testing early in the development process

Introduction

This chapter presents a complete example of the analysis, design, and implementation of a typical classroom programming project. The objective of this chapter is to demonstrate to you how to approach a project and, with the help of the project-approach strategy and the development cycle, formulate and execute a successful project implementation plan.

The approach I take to the problem solution is procedure based. I do this because I find that trying simultaneously to teach problem-solving skills, the C# command-line tools, how to employ the development cycle, and object-oriented design and programming concepts is simply too overwhelming for most students to bear. However, try as I may to defer the discussion of object-oriented concepts, C# forces the issue by requiring that all methods belong to a class. I mitigate this by presenting a solution that results in one user-defined class. As you pursue your studies of C# and progress through the remaining chapters of this book, you will quickly realize that there are many possible solutions to this particular programming project, some of which require advanced knowledge of object-oriented programming theory and techniques.

You may not be familiar with some of the concepts discussed here. Don't panic! I wrote this material with the intention that you revisit it when necessary. As you start writing your own projects examine these pages for clues on how to approach your particular problem. In time, you will begin to make sense of all these confusing concepts. Practice breeds confidence. After a few small victories, you will never have to refer to this chapter again.

The Project-Approach Strategy Summarized

The project-approach strategy presented in Chapter 1 is summarized in Table 3-1. Keep the project-approach strategy in mind as you formulate your solution. Remember, the purpose of the project-approach strategy is to kick-start the creative process and sustain your creative momentum. Feel free to tailor the project-approach strategy to suit your needs.

Strategy Area	Explanation
Application Requirements	Determine and clarify exactly what purpose and features the finished project must have. Clarify your understanding of the requirements with your instructor if the project specification is not clear. *This results in a clear problem definition and a list of required project features.*
Problem Domain	Study the problem until you have a clear understanding of how to solve it. Optionally, express your understanding of the solution by writing a pseudocode algorithm that describes, step-by-step, how you will solve the problem. You may need to do this several times on large, complex projects. *This results in a high-level solution statement that can be translated into an application design.*
Language Features	Make a list of all the language features you must understand and use to draft a competent design and later implement your design. As you study each language feature, check it off your list. Doing so will give you a sense of progress. *This results in a notional understanding of the language features required to effect a good design and solve the problem.*
High-Level Design & Implementation Strategy	Sketch out a rough application design. A design is simply a statement, expressed through words, pictures, or both, of how you plan to implement the problem solution derived in the Problem Domain strategy area. *This results in a plan of attack!*

Table 3-1: Project Approach Strategy

Development Cycle

When you reach the project design phase, you will begin to employ the *development cycle*. It is good to have a broad, macro-level design idea to get you started, but don't make the mistake of trying to design everything up front. Design until you can begin coding and then test some of your design ideas. The development cycle is summarized in Table 3-2.

Step	Explanation
Plan	Design to the point where you can get started on the implementation. Do not attempt to design everything up front. The idea here is to keep your design flexible and open to change.
Code	Implement what you have designed.
Test	Thoroughly test each section or module of source code. The idea here is to try to break it before it has a chance to break your application. Even in small projects you will find yourself writing short test-case programs on the side to test something you have just finished coding.
Integrate/Test	Add the tested piece of the application to the rest of the project and then test the whole project to ensure it didn't break existing functionality.
Refactor	This step applies more to object-oriented programming than to procedural programming. It means to take a comprehensive look at your overall application architecture and migrate general functionality up into base, or even abstract, classes so the functionality can be utilized by more concrete derived classes.

Table 3-2: Development Cycle

Employ the development cycle in a spiral or *iterative* fashion as depicted in Figure 3-1. By iterative, I mean you will begin with the plan step, followed by the code step, followed by the test step, followed by the integrate step, optionally followed by the refactor step. When you have finished a little piece of the project in this fashion, you return to the plan step and repeat the process. Each complete *plan*, *code*, *test*, *integrate*, and *refactor* sequence is referred to as an *iteration*. As you iterate through the cycle, development progresses until you converge on the final solution.

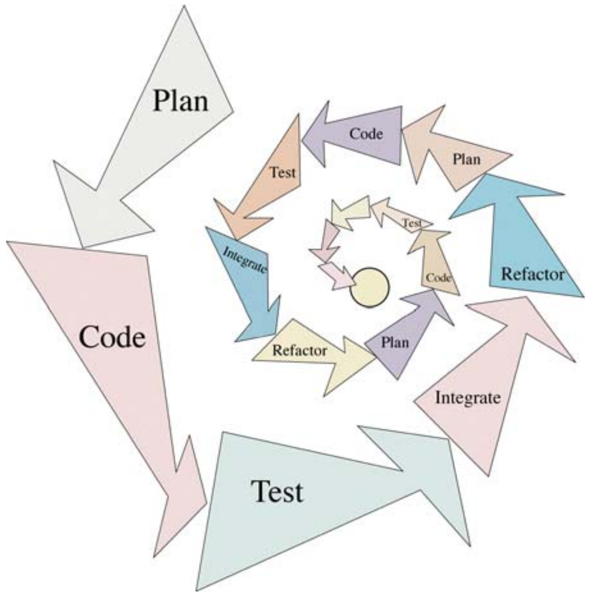


Figure 3-1: Tight-Spiral Development Cycle Deployment

Project Specification

Keeping both the project-approach strategy and development cycle in mind, let's look now at a typical project specification given in Table 3-3.

ITP 136 - Introduction To C# Programming
Project 1
Robot Rat

```
   Objectives:
   Demonstrate your ability to utilize the following language features:
      Value types and reference types
      Two-dimensional arrays
      Program flow-control structures
      Class methods
      Class attributes
      Local method variables
      Constructor methods
      Console input and output

   Task:
   You are in command of a robot rat! Write a C# console application that will
allow you to control the rat's movements around a 20 x 20 grid floor.
   The robot rat is equipped with a pen. The pen has two possible positions, up
or down. When in the up position, the robot rat can move about the floor without
leaving a mark.
   If the pen is down the robot rat leaves a mark as it moves through each grid
position. Moving the robot rat about the floor with the pen up or down at various
locations will result in a pattern written upon the floor.

   Hints:
   - The robot rat can move in four directions: north, south, east, and west.
Implement diagonal movement if you desire.
   - Implement the floor as a two-dimensional array of boolean objects.
   - C# provides the System.Console.ReadLine() and WriteLine() methods that make
it easy to read text from, and and write text to, the console.

   At minimum, provide a text-based command menu with the following or similar
command choices:

                    1. Pen Up
                    2. Pen Down
                    3. Turn Right
                    4. Turn Left
                    5. Move Forward
                    6. Print Floor
                    7. Exit
```

Table 3-3: Project Specification

ITP 136 - Introduction To C# Programming
Project 1
Robot Rat

When menu choice 6 is selected to print the floor, the result might look something like this, assuming you chose '-' to represent a marked area of the floor and '0' to represent an unmarked area. You may use other pattern characters if desired.
.

```
-----000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
```

In this example, the robot rat moved from the upper left-hand corner of the floor five spaces to the east with the pen down.

Table 3-3: Project Specification

Analyzing The Project Specification

Now is a good time to step through the project-approach strategy and analyze the Robot Rat project using each strategy area as a guide, starting with the project's requirements.

Application Requirements Strategy Area

The Robot Rat project seems clear enough, but it omits a few details. It begins with a set of formally stated project objectives. It then states the task you are to execute: namely, to write a program that lets you control a robot rat. But what, exactly, is a robot rat? That's a fair question whose answer requires a bit of abstract thinking. To clarify your understanding of the project's requirements, you decide to ask me a few questions. Your first question is, "Does the robot rat exist?"

If I answered the question by saying, "Well, obviously, the robot rat does not really exist!", I would be insulting you. Why? Because if you are wondering just what a robot rat is, then you are having difficulty abstracting the concept of a robot rat. I would be doing you a better service by saying, "The robot rat exists, but only as a collection of attributes that provide a limited description of the robot rat." I would also add that by writing a program to control the robot rat's movements around the floor, you are actually *modeling* the concept of a robot rat. And since a model of something usually leaves out some level of detail or contains some simplifying assumptions, I will also tell you that the robot rat does not have legs, fur, or a cute little nose.

Another valid requirements question might focus on exactly what is meant by the term *C# console application*. That too is a good question. A C# console application is a program that interacts with the command console using the System.Console.WriteLine() and ReadLine() methods, and optionally can utilize command-line parameters. A C# console program does not usually have a graphical user interface (GUI), although nothing stops you from writing one that does.

What about error checking? Again, good question. In the real world, making sure an application behaves well under extreme user conditions and recovers gracefully in the event of some catastrophe consumes the majority of programming effort. One area in particular that requires extra measures to ensure everything goes well is array processing. As the robot rat is moved around the floor, care must be taken to prevent the program from letting it go beyond the bounds of the floor array.

Something else to consider is how to process menu commands. Since the project only calls for simple console input and output, I recommend treating all input as a text string. If you need to convert a text string into another data type, you can use the methods provided by the System.Convert class. Otherwise, I want you to concentrate on learning how to use the fundamental language features listed in the project's objectives section. So, I promise not to try to break your program when I run it.

You may safely assume that for the purpose of this project the user is perfect. Yet note for the record that this is absolutely not the case in the real world!

To summarize the requirements thus far:

- Write a program that models the concept of a robot rat and its movement upon a floor.
- Think of the robot rat as an abstraction represented by a collection of attributes. (*I discuss these attributes in greater detail in the problem domain section that follows.*)
- Represent the floor in the program as a two-dimensional array of boolean objects.
- Use just enough error checking, focusing on staying within the array boundaries.
- Assume the user is perfect.
- Read user command input as a text string.
- Put all program functionality into one user-defined class. This class will be a C# console application and it will contain a Main() method.

When you are sure you fully understand the project specification, you can proceed to the problem domain strategy area.

Problem-Domain Strategy Area

In this strategy area, your objective is to learn as much as possible about what a robot rat is and how it works in order to gain insight into how to proceed with the project design. A good technique to help jump-start your creativity is to read through the project specification looking for relevant nouns and verbs or verb phrases. A first pass at this activity yields two lists. The list of nouns suggests possible application objects, data types, and object attributes. Nouns also suggest possible names for class and instance fields (*variables and constants*) and method variables. The list of verbs suggests possible object interactions and method names.

Nouns & Verbs

A first pass at reviewing the project specification yields the list of nouns and verbs shown in Table 3-4.

Nouns	Verbs
robot rat	move
floor	set pen up
pen	set pen down
pen position (up, down)	mark
mark	turn right
program	turn left
pattern	print floor
direction (north, south, east, west)	exit
menu	

Table 3-4: Robot Rat Nouns and Verbs

This list of nouns and verbs is a good starting point. Now that you have it, what should you do with it? Good question. As I mentioned previously, each noun is a possible candidate for either a variable, a constant, or some other data type, data structure, object, or object attribute within the application. A few of the nouns will not be used. Others have a direct relationship to a particular application element. Some nouns look like they could be very useful, but may not easily convert or map to any application element. Also, the noun list may not be complete. You may discover additional application objects and object interactions as the project's analysis moves forward.

The verb list for this project example derives mostly from the suggested menu. Verbs normally map directly to potential method names. You will need to create these methods as you write your program. Each method you identify will belong to a particular object, and may utilize some or all of the other objects, variables, constants, and data structures identified with the help of the noun list.

The noun list gleaned so far suggests that the Robot Rat project needs further analysis both to expand your understanding of the project's requirements and to reveal additional attribute candidates. How do you proceed? I recommend taking a closer look at several nouns that are currently on the list, starting with *robot rat*. Just what is a robot rat from the attribute perspective? Since pictures are always helpful, I suggest drawing a few. Figure 3-2 has one for your consideration.

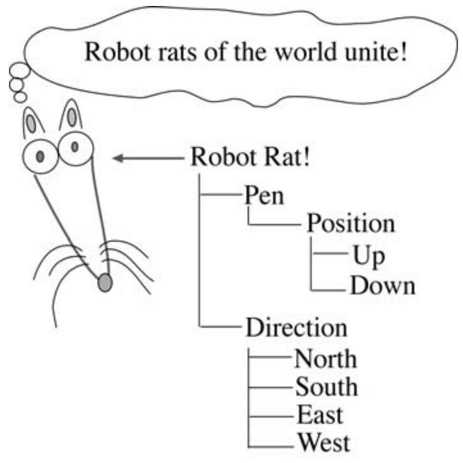


Figure 3-2: Robot Rat Viewed as a Collection of Attributes

Referring to Figure 3-2 — this picture suggests that a robot rat, as defined by the current noun list, consists of a pen that has two possible positions, and the rat's direction. As described in the project specification and illustrated in Figure 3-2, the pen can be either *up* or *down*. Regarding the robot rat's direction, it can face one of four ways: *north*, *south*, *east*, or *west*. Can more attributes be derived? Perhaps another picture will yield more information. I recommend drawing a picture of the *floor* and run through several robot rat movement scenarios as illustrated in Figure 3-3.

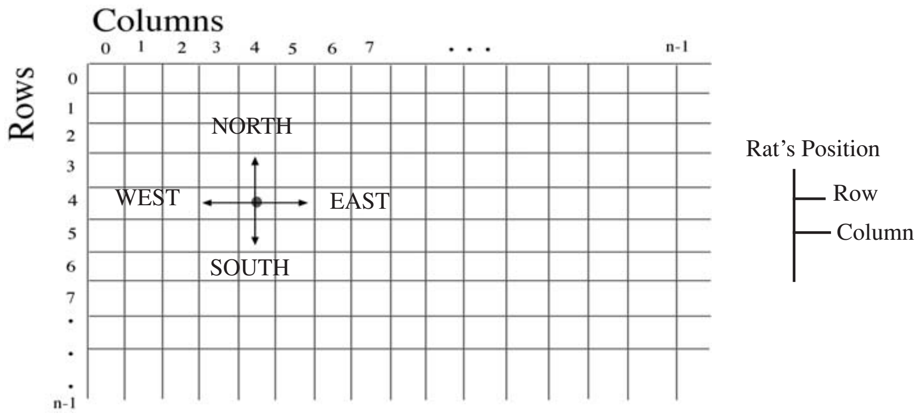


Figure 3-3: Robot Rat Floor Sketch

Figure 3-3 offers a lot of great information about the workings of a robot rat. The floor is represented by a collection of cells arranged by *rows* and *columns*. As the robot rat moves about the floor, its *position* can be determined by keeping track of its current *row* and *column*. These two nouns are good candidates to add to the list of relevant nouns and to the set of attributes that can be used to describe a robot rat. Before the robot rat can move, its current position on the floor must be determined. Upon completion of each robot rat movement, its current position must be updated. Armed with this information, you should now have a better understanding of what attributes are required to represent a robot rat, as Figure 3-4 illustrates.

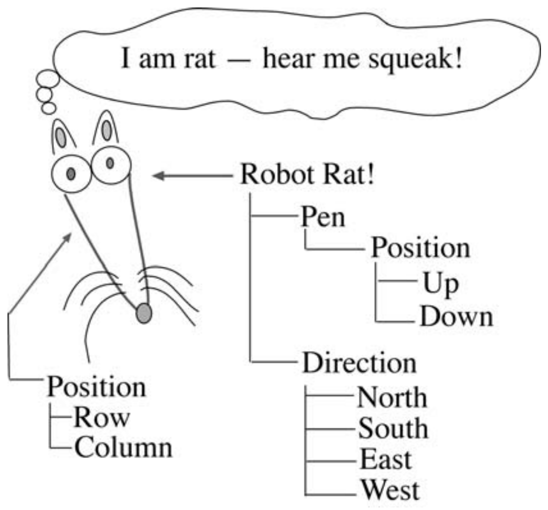


Figure 3-4: Complete Robot Rat Attributes

This seems to be a sufficient analysis of the problem at this point. You can return to this strategy area at any time should further analysis be required. It is now time to take a look at what language features you must understand to implement the solution.

Language-Features Strategy Area

The purpose of the language features strategy area is two-fold: First, to derive a good design to a programming problem you must know what features the programming language supports and how it provides them. Second, you may be forced by a particular programming project to use language features you've never used before. It can be daunting to have lots of requirements thrown at you in one project. The complexities associated with learning the C# language, learning how to create C# projects, learning an integrated development environment (IDE), and learning the process of solving a problem with a computer can induce panic. Use the language-features strategy area to overcome this problem and maintain a sense of forward momentum.

Apply this strategy area by making a list of all the language features you need to study before starting your design. As you study each language feature, mark it off your list. Take notes about each language feature and how it can be applied to your particular problem.

Table 3-5 presents an example check-off list for the language features used in the Robot Rat project.

Check-Off	Feature	Considerations
	C# applications	How do you write a C# application? What's the purpose of the Main() method. What is the meaning of the keywords public, static, and void? What code should go into the Main() method? How do you run a C# application?
	Classes	How do you declare and implement a class? What's the structure of a class. How do you create and compile a C# source file?

Table 3-5: Language Feature Study Check-Off List For Robot Rat Project

Check-Off	Feature	Considerations
	Value data types	What is a value data type? How many are there? What is each one used for? How do you use them in a program? What range of values can each data type contain? How do you declare and use value data type variables or constants in a program?
	Reference data types	What is a reference data type? How do you declare and use reference data types in a program? What's the purpose of the *new* operator? What pre-existing C# classes can be used to help create the program?
	Arrays	What's the purpose and use of an array? How do you declare an array reference and use it in a program? What special functionality do array objects have?
	Two-dimensional arrays	What is the difference between a one-dimensional array and a two-dimensional array? How do you declare and initialize a two-dimensional array? How do you access each element in a two-dimensional array?
	Fields	What is a field? How do you declare class and instance fields? What's the difference between class fields and instance fields? What is the scope of a field?
	Properties	What is a property? How do you implement a property? What's the difference between a read-only property and a read-write property?
	Methods	What is a method? What are they good for? How do you declare and call a method? What's the difference between a static method and a non-static method? What are method parameters? How do you pass arguments to methods? How do you return values from methods?
	Local variables	What is a local variable? How does their use affect class or instance fields? How long does a local variable exist? What is the scope of a local variable?
	Constructor methods	What is the purpose of a constructor method? Can there be more than one constructor method? What makes constructor methods different from ordinary methods?
	Flow-control statements	What is a flow-control statement? How do you use if, if/else, while, do, for, and switch statements in a program? What's the difference between for, while, and do? What's the difference between if, if/else, and switch?
	Console I/O	What is console input and output? How do you print text to the console? How do you read text from the console and use it in your program?

Table 3-5: Language Feature Study Check-Off List For Robot Rat Project

Armed with your list of language features, you can now study each one, marking it off as you go. When you come across a good code example that shows you how to use a particular language feature, copy it down or print it out and save it in a notebook for future reference.

Learning to program is a lot like learning to play a musical instrument. It takes observation and practice. You must put your trust in the masters and mimic their style. You may not at first fully understand why a particular piece of code works the way it does, or why they wrote it the way they did. But copy their style until you start to understand the underlying principles. Doing this builds confidence — slowly but surely. Soon you will have the skills required to set out on your own and write code with no help at all. In time, your programming skills will surpass those of your teachers.

After you have compiled and studied your list of language features, you should have a sense of what you can do with each feature and how to start the design process. More importantly, you will now know where to refer to when you need to study a particular language feature in more depth. However, by no means will you have mastered the use of these features. So don't feel discouraged if, having arrived at this point, you still feel a bit overwhelmed by all that you must know. I must emphasize here that to master the art of programming takes practice, practice, practice!

Once you have studied each required language feature, you are ready to move on to the design strategy area of the project-approach strategy.

Design Strategy Area

You must derive a plan of attack before you can solve the robot rat problem! Your plan will consist of two essential elements: a high-level *software architecture diagram* and an *implementation approach*.

High-Level Software-Architecture Diagram

A high-level software-architecture diagram is a picture of both the software components needed to implement the solution and their relationship to each other. Creating the high-level software-architecture diagram for the Robot Rat project is easy, as the application will contain only one class. On the other hand, complex projects usually require many different classes, and each of these classes may interact with the others in some way. For these types of projects software-architecture diagrams play a key role in helping software engineers understand how the application works.

The Unified Modeling Language (UML) is used industry-wide to model software architectures. The UML class diagram for the RobotRat class at this early stage of your project's design will look similar to Figure 3-5.

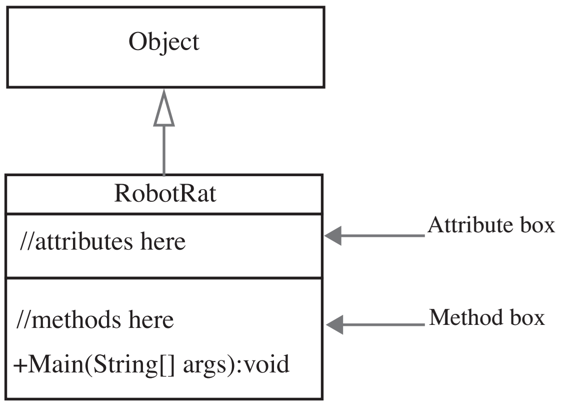


Figure 3-5: RobotRat UML Class Diagram

As Figure 3-5 illustrates, the RobotRat class extends (*inherits*) the functionality provided by the System.Object class. This is indicated by the hollow-pointed arrow pointing from RobotRat to Object. In C#, all user-defined classes implicitly extend Object so you don't have to do anything special to achieve this functionality. The RobotRat class will have attributes (*properties and fields*) and *methods*. Attributes are listed in the attribute box and methods are listed in the method box. The Main() method is shown. The plus sign to the left of the Main() method indicates that it is a *public* method.

Implementation Approach

Before you begin coding, you must have some idea of how you are going to translate the design into a finished project. Essentially, you must answer the following question: "Where do I start?" Getting started is easily 90% percent of the battle!

When formulating an implementation approach, you can proceed *macro-to-micro*, *micro-to-macro*, or a combination of both. I realize this sounds like unorthodox terminology, but bear with me.

If you use the macro-to-micro approach, you build and test a code *framework* to which you incrementally add functionality that ultimately results in a finished project. If you use the micro-to-macro approach, you build and test small pieces of functionality first and then, bit-by-bit, combine them into a finished project.

More often than not, you will use a combination of these approaches. Object-oriented design begs for macro-to-micro as a guiding approach. But both approaches play well with each other, as you will soon see. C# forces the issue somewhat by requiring that all methods and attributes belong to a class.

There will be many unknowns when you start your design. For instance, you could attempt to specify all the methods required for the RobotRat class up front. But as you progress through development, you will surely see the need for a method you didn't initially envision.

The following general steps outline a viable implementation approach to the Robot Rat project:

- Proceed from macro-to-micro by first creating and testing the RobotRat application class, devoid of any real functionality.
- Add and test a menu display capability.
- Add and test a menu-command processing framework by creating several empty methods that will serve as placeholders for future functionality. These methods are known as *method stubs*. (*Method stubbing is a great programming trick!*)
- Once you have tested the menu-command processing framework, you must implement each menu item's functionality. This means that you must implement and test the stub methods created in the previous step. The Robot Rat project is complete when all required functionality has been implemented and successfully tested.
- Develop the project iteratively. This means that you will repeatedly execute the *plan-code-test-integrate* cycle many times on small pieces of the project until the project is complete.

Now that you have an overall implementation strategy, you can proceed to the development cycle. The following sections walk you step-by-step through the iterative application of the development cycle.

Development Cycle: First Iteration

Armed with an understanding of the project requirements, problem domain, language features, and an implementation approach, you are ready to begin development. To complete the project, apply the development cycle iteratively. That is, apply each of the development cycle phases —*plan*, *code*, *test*, and *integrate* —to a small, selected piece of the overall problem. When you've finished that piece of the project, select another piece and repeat the process. The following sections step through the iterative application of the Robot Rat project's development cycle.

Plan (First Iteration)

A good way to start each iteration of the development cycle is to list those pieces of the programming problem you are going to solve this time around. The list should have two columns: one that lists each piece of the program design or feature under consideration, and another that notes you design decisions regarding that feature. Again, the purpose of the list is to help you maintain a sense of forward momentum. You may find, after you make the list, that you need more study in a particular language feature before proceeding to the coding step. That's normal. Even seasoned programmers occasionally need to brush-up on unfamiliar or forgotten language features or application programming interfaces (APIs). (*i.e., .NET Framework classes or third-party APIs*)

The list of the first pieces of the Robot Rat project that should be solved based on the previously discussed implementation approach is shown in Table 3- 6.

Check-Off	Design Consideration	Design Decision
	Program structure	One class will contain all the functionality.
	Creating the C# application class	The class name will be RobotRat. It will contain a "public static void Main(String[] args){ } method.
	Constructor method	Write a constructor that will print a short message to the screen when a RobotRat object is created.

Table 3-6: First Iteration Design Considerations

This is good for now. Although it doesn't look like much, creating the RobotRat application class and writing a constructor that prints a short text message to the console are huge steps. You can now take this list and move on to the code phase.

Code (First Iteration)

Create the Robot Rat project using your development environment. Create a C# source file named RobotRat.cs and in it create the RobotRat class definition. Add to this class the Main() method and the RobotRat constructor method. When complete, your RobotRat.cs source file should look similar to Example 3.1.

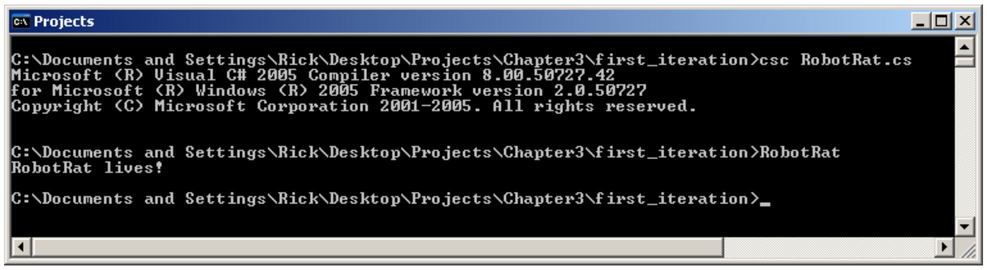
3.1 RobotRat.cs (1st Iteration)

```
using System;

public class RobotRat {
   public RobotRat(){
     Console.WriteLine("RobotRat lives!");
   }

   public static void Main(String[] args){
     RobotRat rr = new RobotRat();
   }
}
```

After you have created the source file, you can move to the test phase.

Test (First Iteration)

The test phase of the first iteration involves compiling the RobotRat.cs file and running the resulting RobotRat.exe file. If the compilation results in errors, return to the code phase, edit the file to make the necessary correction, and then attempt to compile and test again. Repeat the cycle until you are successful. When you have successfully compiled and tested the first iteration of the RobotRat program, move on to the next step of the development cycle. Figure 3-6 shows the results of running Example 3.1.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter3\first_iteration>csc RobotRat.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.


C:\Documents and Settings\Rick\Desktop\Projects\Chapter3\first_iteration>RobotRat
RobotRat lives!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter3\first_iteration>_
```

Figure 3-6: Compiling and Testing RobotRat — First Iteration

Integrate/Test (First Iteration)

There's not a whole lot to integrate at this point, so you are essentially done with the first development cycle iteration. Since this version of the Robot rat project is contained in one class named RobotRat, any changes you make directly to the source file are immediately integrated into the project. However, for larger projects, you will code and test a piece of functionality, usually at the class level, before adding the class to the larger project.

You are now ready to move to the second iteration of the development cycle.

Development Cycle: Second Iteration

With the RobotRat application class structure in place and tested, it is time to add another piece of functionality to the program.

Plan (Second Iteration)

A good piece of functionality to add to the RobotRat class at this time would be the text menu that displays the list of available robot rat control options. Refer to the project specification to see a suggested menu example. Table 3-7 lists the features that will be added to RobotRat during this iteration.

Check-Off	Design Consideration	Design Decision
	Display control menu to the console	Create a RobotRat method named PrintMenu() that prints the menu to the console. Test the PrintMenu() method by calling it from the constructor. The PrintMenu() method will return void and take no arguments. It will use the Console.WriteLine() method to write the menu text to the console.

Table 3-7: Second Iteration Design Considerations

This looks like enough work to do for one iteration. You can now move to the code phase.

Code (Second Iteration)

Edit the RobotRat.cs source file to add a method named PrintMenu(). Example 3.2 shows the RobotRat.cs file with the PrintMenu() method added.

3.2 RobotRat.cs
(2nd Iteration)

```
1    using System;
2
3    public class RobotRat {
4
5       public RobotRat(){
6         PrintMenu();
7       }
8
9       public void PrintMenu(){
10        Console.WriteLine("\n\n");
11        Console.WriteLine("   RobotRat Control Menu");
12        Console.WriteLine();
13        Console.WriteLine("  1. Pen Up");
14        Console.WriteLine("  2. Pen Down");
15        Console.WriteLine("  3. Turn Right");
16        Console.WriteLine("  4. Turn Left");
17        Console.WriteLine("  5. Move Forward");
18        Console.WriteLine("  6. Print Floor");
19        Console.WriteLine("  7. Exit");
20        Console.WriteLine("\n\n");
21      }
22
23   public static void Main(String[] args){
24        RobotRat rr = new RobotRat();
25      }
26   }
```

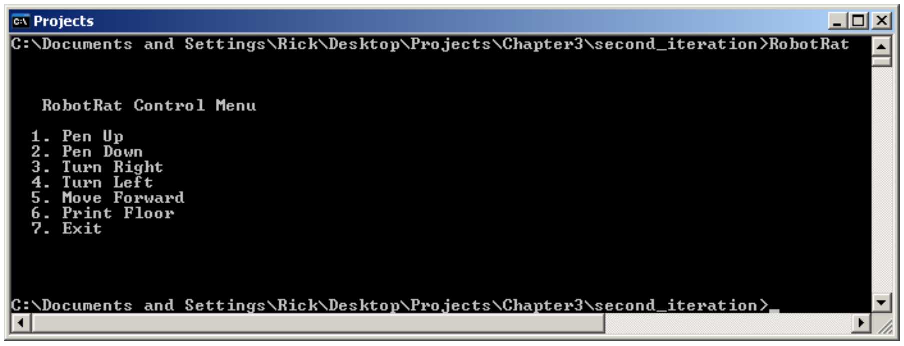
Referring to Example 3.2 — the PrintMenu() method begins on line 9. Notice how it's using the Console.WriteLine() method to write menu text to the console. The "\n" is the escape sequence for the newline character. Several of these are used to add line spacing as an aid to menu readability.

Notice also that the code in the constructor method has changed. I removed the line printing the "RobotRat Lives!" message and replaced it with a call to the PrintMenu() method on line 6.

When you've finished making the edits to RobotRat.cs, you are ready to compile and test.

Test (Second Iteration)

Figure 3-7 shows the results of testing RobotRat at this stage of development. The menu seems to print well enough.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter3\second_iteration>RobotRat


   RobotRat Control Menu

  1. Pen Up
  2. Pen Down
  3. Turn Right
  4. Turn Left
  5. Move Forward
  6. Print Floor
  7. Exit


C:\Documents and Settings\Rick\Desktop\Projects\Chapter3\second_iteration>_
```

Figure 3-7: Compiling & Testing RobotRat - Second Iteration

Integrate/Test (Second Iteration)

Again, there is not much to integrate or test. If you are happy with the way the menu looks on the console, you can move on to the third iteration of the development cycle.

Development Cycle: Third Iteration

OK. The class structure is in place and the menu is printing to the screen. The next piece of the project to work on should be the processing of menu commands. Here's where you can employ the technique of method stubbing so that you can worry about the details later.

Plan (Third Iteration)

Table 3.8 lists the design considerations for this iteration.

Check-Off	Design Consideration	Design Decision
	Read the user's desired menu command from the console	Use the Console.Readline() method to read a string of characters from the console. Once you get the text string from the console, you will only want to use the first letter of the string. Individual string elements can be accessed using array notation. The first character of a string resides at the 0th element. If you use a variable named "input" to hold the string, the first character can be obtained by using "input[0]".
	Execute the user's selected menu command	This will be accomplished with a user-defined method named ProcessMenuChoice(). This method will return void and take no arguments. The body of the ProcessMenuChoice() method will utilize a `switch` statement that acts on the menu command entered by the user. Each case of the `switch` statement will call a user-defined method to execute the required functionality. These methods will be named as follows: SetPenUp(), SetPenDown(), TurnLeft(), TurnRight(), MoveForward(), and PrintFloor(). To repeatedly print the control menu and process user commands you will need to create another method that calls the PrintMenu() and ProcessMenuChoice() methods in a continuous loop until the user exits the application. The name of this method could be called Run().

Table 3-8: Third Iteration Design Considerations

Check-Off	Design Consideration	Design Decision
	Use method stubbing to test the ProcessMenuChoice() method	This means that the user-defined methods mentioned above will be stubbed out. The only functionality they will provide during this iteration will be to print a short test message to the console.
	Exiting the application.	Use a boolean sentinel variable to exit the application. The name of this variable could be called "keep_going", and initially can be set to true. When the user selects the exit menu item, the program will set the value of keep_going to false.
	Program readability	Use character constants to represent the possible menu values and use them in the body of the switch statement. For example, if the user selects the Pen Up menu option he will enter the character '1'. You could use '1' in the switch statement, but, will you remember that '1' represents Pen Up? Instead of embedding the character '1' in the switch statement, you can declare a character constant named "PEN_UP" and set its value to '1'. In the body of the switch statement you can then use the constant and know exactly what menu choice you're talking about. Using constants in this fashion significantly improves program readability.

Table 3-8: Third Iteration Design Considerations

This list of items will keep you busy for a while. You will actually move between the code and test phase repeatedly in this iteration, almost as if it were a mini development spiral in and of itself. The best place to start is at the top of the list.

Don't try to implement everything on the list completely and then compile. Instead, try to get one or two menu options working with the stubbed methods, then compile and test. When you're happy with the results, add the capability to process another menu option, and so on, and so on, until you've got it licked!

Code (Third Iteration)

Example 3.3 gives the code for the RobotRat.cs file with all the new features implemented. I'll discuss the new code sections following the example.

3.3 RobotRat.cs (3rd Iteration)

```
using System;

public class RobotRat {

   private bool keep_going = true;

   private const char PEN_UP       = '1';
   private const char PEN_DOWN     = '2';
   private const char TURN_RIGHT   = '3';
   private const char TURN_LEFT    = '4';
   private const char MOVE_FORWARD = '5';
   private const char PRINT_FLOOR  = '6';
   private const char EXIT         = '7';

   public RobotRat(){
     Console.WriteLine("RobotRat Lives!");
   }

   public void PrintMenu(){
     Console.WriteLine("\n\n");
     Console.WriteLine("   RobotRat Control Menu");
     Console.WriteLine();
     Console.WriteLine("  1. Pen Up");
     Console.WriteLine("  2. Pen Down");
     Console.WriteLine("  3. Turn Right");
     Console.WriteLine("  4. Turn Left");
     Console.WriteLine("  5. Move Forward");
     Console.WriteLine("  6. Print Floor");
     Console.WriteLine("  7. Exit");
```

```
      Console.WriteLine("\n\n");
    }

    public void ProcessMenuChoice(){
       String input = Console.ReadLine();

       switch(input[0]){
         case PEN_UP :       SetPenUp();
                             break;
         case PEN_DOWN :     SetPenDown();
                             break;
         case TURN_RIGHT :   TurnRight();
                             break;
         case TURN_LEFT :    TurnLeft();
                             break;
         case MOVE_FORWARD : MoveForward();
                             break;
         case PRINT_FLOOR :  PrintFloor();
                             break;
         case EXIT :         keep_going = false;
                             break;
         default :           PrintErrorMessage();
                             break;
       }
    }


    public void SetPenUp(){
      Console.WriteLine("SetPenUp method called.");
    }

    public void SetPenDown(){
       Console.WriteLine("SetPenDown method called.");
    }

    public void TurnRight(){
      Console.WriteLine("TurnRight method called.");
    }

    public void TurnLeft(){
      Console.WriteLine("TurnLeft method called.");
    }

    public void MoveForward(){
      Console.WriteLine("MoveForward method called.");
    }

    public void PrintFloor(){
      Console.WriteLine("PrintFloor method called.");
    }

    public void PrintErrorMessage(){
      Console.WriteLine("Please enter a valid RobotRat control option!");
    }

    public void Run(){
      while(keep_going){
        PrintMenu();
        ProcessMenuChoice();
      }
    }

    public static void Main(String[] args){
      RobotRat rr = new RobotRat();
      rr.Run();
    }

 }
```

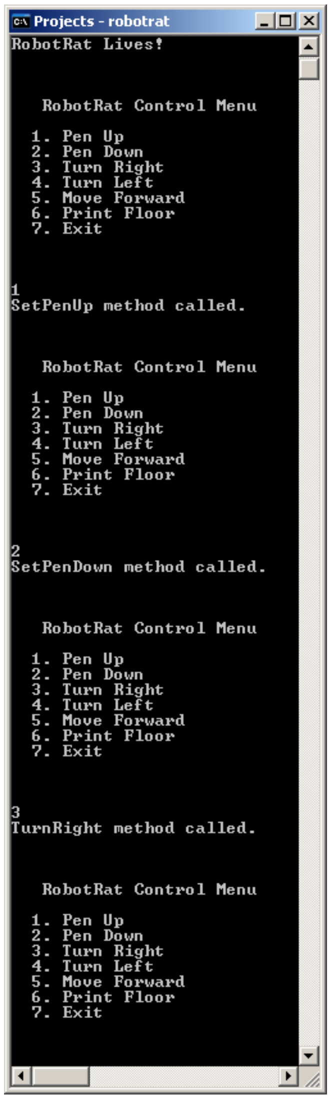
Referring to Example 3.3 — notice that the boolean (bool) variable keep_going has been added to the program along with character (char) constants that represent the range of possible menu choices.

The ProcessMenuChoice() method begins on line 33. The first thing it does is declare a local String variable named "input" and assigns to it the character string read from the console with the help of the Console.ReadLine() method. The first character of the input string (input[0]) is then evaluated by the `switch` statement and compared to the values of each of the character constants. Further processing is passed to the appropriate method. For example, in the case of the user selecting menu option '1' for Pen Up, the `switch` statement compares the '1' character to the constant value contained in the PEN_UP constant. This results in a call to the SetPenUp() method. The `break` keyword exists the body of the `switch` statement.

Notice that the `switch` statement contains a `default` case. If the input character fails to match any of the valid menu options, the `default` case executes. This calls the PrintErrorMessage() method, which writes a short message to the console prompting the user to enter a valid menu option.

Let's now look at how the Run() method works. The Run() method begins on line 85. It contains a `while` loop that is controlled by the value of the keep_going variable. As you can see at the top of the code listing, the value of keep_going is initialized to true. While keep_going is true, the `while` loop will endlessly call the PrintMenu() method followed by the ProcessMenuChoice() method. This will go on until the user enters '7' at the console, which means he wants to exit the program. This causes the value of keep_going to be set to false, at which time the `while` loop in the Run() method exits and the program halts.

Look now at the Main() method on line 92. It creates an instance of RobotRat and calls the Run() method to start the menu display and menu processing cycle.



```
RobotRat Lives!


   RobotRat Control Menu

  1. Pen Up
  2. Pen Down
  3. Turn Right
  4. Turn Left
  5. Move Forward
  6. Print Floor
  7. Exit


1
SetPenUp method called.


   RobotRat Control Menu

  1. Pen Up
  2. Pen Down
  3. Turn Right
  4. Turn Left
  5. Move Forward
  6. Print Floor
  7. Exit


2
SetPenDown method called.


   RobotRat Control Menu

  1. Pen Up
  2. Pen Down
  3. Turn Right
  4. Turn Left
  5. Move Forward
  6. Print Floor
  7. Exit


3
TurnRight method called.


   RobotRat Control Menu

  1. Pen Up
  2. Pen Down
  3. Turn Right
  4. Turn Left
  5. Move Forward
  6. Print Floor
  7. Exit
```

Figure 3-8: Testing Menu Commands

Incremental Testing

Although you could have tried to make all the required modifications to the RobotRat class at one stroke, you most likely would make small changes or additions to the code and then immediately compile and test the results.

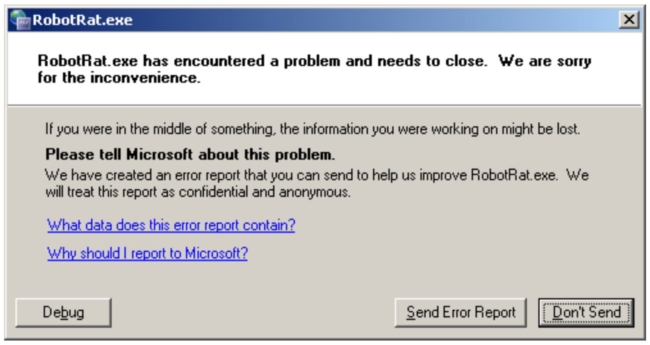
Another area where proceeding in small steps is a good idea would be in coding the body of the ProcessMenuChoice() method. You can write the `switch` statement one case at a time. Then, once you gain confidence that your code works as planned, you can code up the remaining cases in short order.

Integrate/Test (Third Iteration)

Now that you are accepting and processing user menu commands, you can examine the effect this has on the menu display. If you look at Figure 3-8, it appears as if there might be too many spaces between the last menu item and the menu entry prompt. Note that the addition of one feature affects another program feature. You can adjust the space issue in the next development cycle iteration.

A Bug In The Program

If, while testing, you accidentally pressed the Enter key without typing a menu option, you would have seen the rather disturbing error message shown in Figure 3-9. This message gives you no indication of what went wrong. But if you managed to stay calm and close this window without sending the error report, and looked closely at the command console window in which the Robot Rat program was running, you would see another error message as shown in Figure 3-10. This error message offers a little more information. It says that because you pressed the Enter key with no string to be read in, there was no string from which to access the first character, therefore the attempt to access input[0] resulted in an IndexOutOfRangeException. To fix this problem, you must revisit the ProcessMenuChoice() method and provide some means of either catching the exception and doing something about it, or add some code that prevents the exception from happening in the first place. This latter route is the one I will take in the next example.


RobotRat.exe
What data does this error report contain?
Why should I report to Microsoft?
Debug
Send Error Report
Don't Send

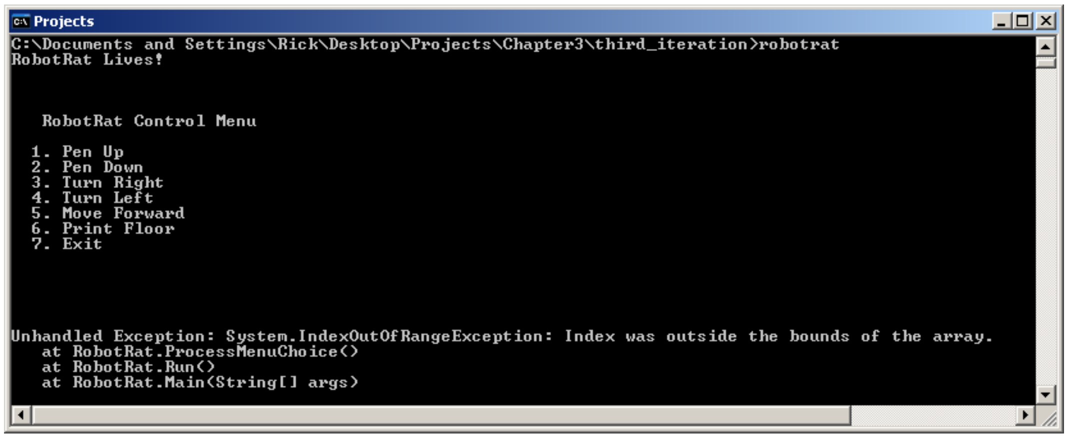
RobotRat.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.

We have created an error report that you can send to help us improve RobotRat.exe. We will treat this report as confidential and anonymous.

Figure 3-9: A Disturbing Error Message



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter3\third_iteration>robotrat
RobotRat Lives!


   RobotRat Control Menu

  1. Pen Up
  2. Pen Down
  3. Turn Right
  4. Turn Left
  5. Move Forward
  6. Print Floor
  7. Exit




Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the array.
   at RobotRat.ProcessMenuChoice()
   at RobotRat.Run()
   at RobotRat.Main(String[] args)
```

Figure 3-10: Unhandled IndexOutOfRangeException Error Message

Example 3.4 shows the slightly modified ProcessMenuChoice() method with some error prevention code added for good measure.

3.4 Modified ProcessMenuChoice() Method

```
1     public void ProcessMenuChoice(){
2           String input = Console.ReadLine();
3
4           if(input == String.Empty){
5             input = "0";
6           }
7
8           switch(input[0]){
9             case PEN_UP :       SetPenUp();
10                                break;
11            case PEN_DOWN :     SetPenDown();
12                                break;
13            case TURN_RIGHT :   TurnRight();
14                                break;
15            case TURN_LEFT :    TurnLeft();
16                                break;
17            case MOVE_FORWARD : MoveForward();
18                                break;
19            case PRINT_FLOOR :  PrintFloor();
20                                break;
21            case EXIT :         keep_going = false;
22                                break;
23            default :           PrintErrorMessage();
24                                break;
25          }
26       }
```

Referring to Example 3.4 — notice on line 4 that if the input string is empty, I give it the value "0". This means that if the user fails to enter a valid input string, the `switch` statement's `default` case will execute and display an error message prompting the user to enter a valid command. Anytime you make a change like this, you must retest your code to make sure everything works fine. This is what is meant by the term *regression testing*.

Development Cycle: Fourth Iteration

The RobotRat code framework is now in place. You can run the program, select menu choices, and see the results of your actions via stub method console messages. It's now time to start adding detailed functionality to the RobotRat class.

Plan (Fourth Iteration)

You now need to both add more attributes to the RobotRat class and begin manipulating those attributes. Two good attributes to start with are the robot rat's direction and pen_position. Another good piece of functionality to implement is the floor. A good goal would be to create, initialize, and print the floor. It would also be nice to load the floor with one or more test patterns.

Table 3-9 lists the design considerations for this development cycle iteration. As you will soon see, more detailed analysis is required to implement the selected Robot Rat program features. This analysis may require the use of design drawings such as flow charts, state transition diagrams, and class diagrams in addition to pseudocode and other design techniques.

Check-Off	Design Consideration	Design Decision
	Implement robot rat's direction	The direction can be an integer (int) variable or a variable of an enumerated type. It will have four possible states or values: NORTH, SOUTH, EAST, and WEST. You can implement these as class constants or as an enumeration. This will make the source code easier to read and maintain. The robot rat's direction will change when either the TurnLeft() or TurnRight() methods are called. The initial direction upon program startup will be EAST.
	Implement robot rat's pen_position	The pen_position will be an integer variable or a variable of some enumerated type. It will have two valid states or values: UP and DOWN. These can be implemented as class constants or enumerations as well. The robot rat's pen_position will change when either the SetPenUp() or the SetPenDown() methods are called. The initial pen_position value upon program startup will be UP.
	floor	The floor will be a two-dimensional array of boolean variables. If an element of the floor array is set to true it will result in the `'-'` character being printed to the console. If an element is false the `'0'` character will be printed to the console. The floor array elements upon program startup will be initialized to false.

Table 3-9: Fourth Iteration Design Considerations

This will keep you busy for a while. You may need to spend more time analyzing the issues regarding the setting of the robot rat's direction and its pen_position. It is often helpful to draw state transition diagrams to graphically illustrate object state changes. Figure 3-11 shows the state transition diagram for pen_position.

As Figure 3-11 illustrates, the pen_position variable is set to the UP state upon program startup. It will remain UP until the SetPenDown() method is called, at which time it will be set to the DOWN state. A similar state transition diagram is shown for the direction variable in Figure 3-12.

As is illustrated in Figure 3-12, the robot rat's direction is initialized to EAST upon program startup. Each call to the TurnLeft() or TurnRight() methods will change the state (value) of the direction variable.

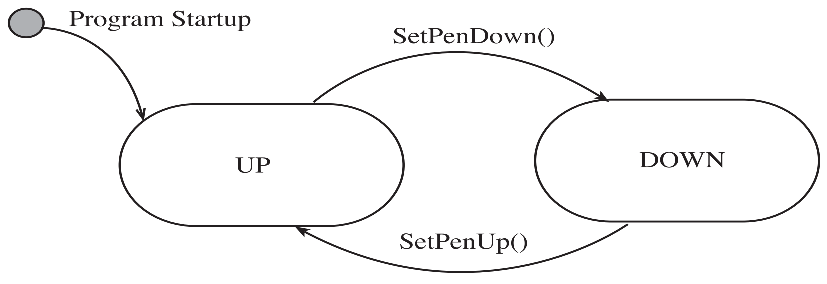


Figure 3-11: pen_position State Transition Diagram

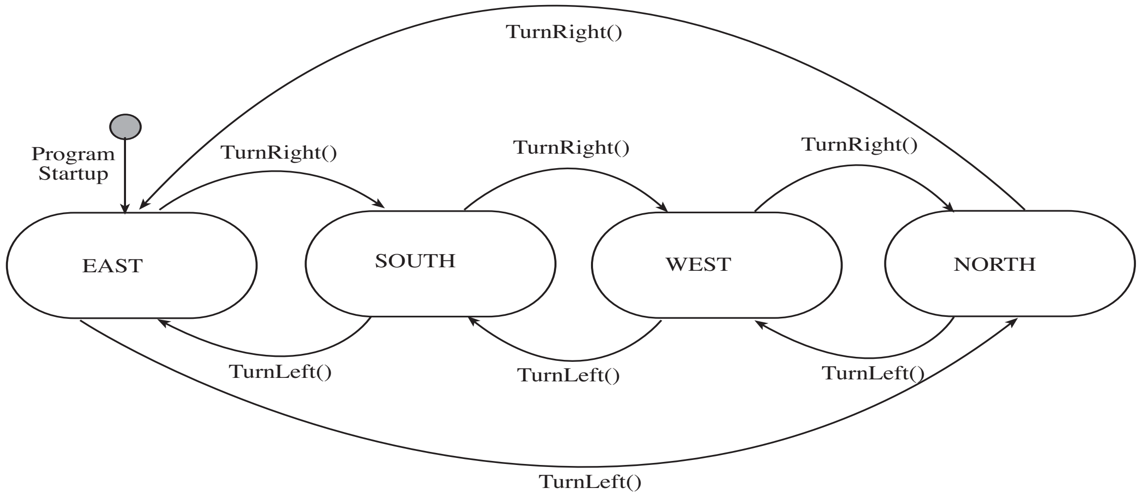


Figure 3-12: State Transition Diagram for the direction Variable

Implementing State Transition Diagrams

State transition diagrams of this nature are easily implemented using a `switch` statement. You could use an `if/else` statement but the `switch` works well in this case. Example 3.5 gives a pseudocode description for the TurnLeft() method:

3.5 Pseudocode for TurnLeft() Method

```
check the value of the direction variable
if direction equals EAST then set the value of direction to NORTH
else if direction equals NORTH then set the value of direction to WEST
else if direction equals WEST then set the value of direction to SOUTH
else if direction equals SOUTH then set the value of direction to EAST
else if direction equals an invalid state set the value of direction to EAST.
```

You could construct a pseudocode description for the TurnRight(), SetPenUp(), and SetPenDown() methods as well using the state transition diagrams as a guide.

Implementing The PrintFloor() Method

Example 3.6 gives the pseudocode for the PrintFloor() method.

3.6 Pseudocode for PrintFloor() Method

```
for each row in the floor do the following
  for each column in each row do the following
    check the value of the floor element
      if the value is true print the '*' character to the console
      else if the value is false print the '0' character to the console
  print a newline character at the end of each row
```

When you feel you have done enough analysis of the current set of robot rat features, you can move on to the code phase of the development cycle.

Code (Fourth Iteration)

The PrintFloor(), TurnLeft(), TurnRight(), SetPenUp(), and SetPenDown() methods already exist as stub methods. In this phase, you will proceed to add the required code to each of these methods, then compile and test the results. Just like the previous iteration, this code phase comprises multiple code, compile, and test cycles.

To proceed, first add the required variable and enum declarations to the top of the class. Any fields declared here could be initialized either by the constructor method or at the point of declaration. Example 3.7 shows a partial code listing for this section of the RobotRat.cs source file, along with a constructor method that initializes the floor.

3.7 RobotRat.cs
(4th Iteration Partial Listing)

```
1    using System;
2
3    public class RobotRat {
4
5       private bool keep_going = true;
6
7       private const char PEN_UP       = '1';
8       private const char PEN_DOWN     = '2';
9       private const char TURN_RIGHT   = '3';
10      private const char TURN_LEFT    = '4';
11      private const char MOVE_FORWARD = '5';
12      private const char PRINT_FLOOR  = '6';
13      private const char EXIT         = '7';
14
15      private enum PenPositions { UP, DOWN };
16      private enum Directions { NORTH, SOUTH, EAST, WEST };
17
18      private PenPositions pen_position = PenPositions.UP;
19      private Directions   direction    = Directions.EAST;
20
21      private bool[,] floor;
22
23
24      public RobotRat(int rows, int cols){
25        Console.WriteLine("RobotRat Lives!");
26        floor = new bool[rows,cols];
27      }
```

Referring to Example 3.7 — the enumerations *PenPositions* and *Directions* have been declared on lines 15 and 16. Variables of the enum types named *pen_position* and *direction* have been declared and initialized on lines 18 and 19. The floor array is declared on line 21. It is initialized in the constructor method with the help of two constructor parameters named *rows* and *cols*, which represent the number of rows and columns the floor should have when the RobotRat object is created.

Let's look now at the SetPenUp(), SetPenDown(), TurnLeft(), and TurnRight() methods. Example 3.8 shows the source code for the SetPenUp() method.

3.8 SetPenUp() method

```
public void SetPenUp(){
     pen_position = PenPositions.UP;
     Console.WriteLine("The pen is " + pen_position);
   }
```

As you can see, it's fairly straightforward. The SetPenUp() method just sets the pen_position attribute to the UP state, and then prints a short message to the console showing the user the current state of the pen_position variable. Example 3.9 gives the code for the SetPenDown() method.

3.9 SetPenDown() method

```
public void SetPenDown(){
      pen_position = PenPositions.DOWN;
      Console.WriteLine("The pen is " + pen_position);
   }
```

The SetPenDown() method is similar to the previous method, only it's setting the pen_position to the opposite state. Let's look now at the TurnLeft() method as shown in Example 3.10.

3.10 TurnLeft() method

```
public void TurnLeft(){
     switch(direction){
            case Directions.NORTH : direction = Directions.WEST;
                                    break;
            case Directions.WEST  : direction = Directions.SOUTH;
```

```
                                       break;
               case Directions.SOUTH : direction = Directions.EAST;
                                       break;
               case Directions.EAST  : direction = Directions.NORTH;
                                       break;
             }

        Console.WriteLine("Direction is " + direction);
      }
```

Notice that in the TurnLeft() method the `switch` statement checks the value of the direction field and then executes the appropriate case statement. The TurnRight() method is coded in similar fashion using the state transition diagram as a guide.

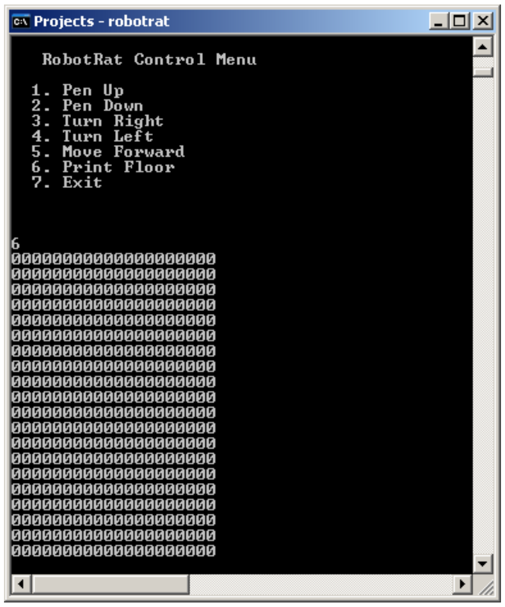
The PrintFloor() method is all that's left for this iteration, and is shown in Example 3.11.

3.11 PrintFloor() method

```
 public void PrintFloor(){
        for(int i = 0; i<floor.GetLength(0); i++){
          for(int j = 0; j<floor.GetLength(1); j++){
            if(floor[i,j]){
              Console.Write('-');
            }else{
              Console.Write('0');
            }
          }
          Console.WriteLine();
        }
     }
```

Test (Fourth Iteration)

There's a lot to test for this iteration. You'll need to test all the methods that were modified. You'll be especially anxious to test the PrintFloor() method since now you'll see the floor pattern print to the console. Figure 3-13 shows the PrintFloor() method being tested. As you can see, it just prints the `'0'` characters to the screen. You might find it helpful to load the floor array with a test pattern to test the `'-'` characters as well.



```
  RobotRat Control Menu

 1. Pen Up
 2. Pen Down
 3. Turn Right
 4. Turn Left
 5. Move Forward
 6. Print Floor
 7. Exit



6
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
```

Figure 3-13: Testing the PrintFloor() Method

Integrate/Test (Fourth Iteration)

Check to see how all the new functionality you added has affected any previously working functionality. It would also be a good idea to see how the floor looks using different floor array dimensions. The changes to the RobotRat constructor method make it easy to create RobotRat objects with different floor sizes. When you are happy with all the work done in this iteration, it's time to move on to the next development cycle iteration.

Development Cycle: Fifth Iteration

All that remains to code at this point is the MoveForward() method. All the other supporting pieces are now in place. You can change the robot rat's direction by turning left or right, and change its pen position to up or down. The MoveForward() method will use all this functionality to model the movement of the robot rat across the floor.

As you get into the planning phase of this iteration, you may discover you need more than just the MoveForward() method to properly move the robot rat. For instance, you'll need some way to get the number of spaces from the user to move the robot rat.

Plan (Fifth Iteration)

Table 3-10 lists the design considerations for this iteration of the development cycle.

Check-Off	Design Consideration	Design Decision
	Coding the MoveForward() method	Write the code for the MoveForward() method. The MoveForward() method will use the direction and pen_position fields to make move decisions. The MoveForward() method will need a way to get the number of spaces to move from the user. You will have to add the current_row and current_col fields to the RobotRat class. These fields will be used to preserve the robot rat's floor position information between moves.
	Getting the number of spaces to move from the user	When the user moves the robot rat they will need to enter the number of spaces so the move can be executed. This can be handled by writing a new method called GetSpacesToMove(). The GetSpacesToMove() method will read a text string from the console and convert it into an integer. The GetSpacesToMove() method will return an integer value and take no parameters.
	Add current_row & current_col fields	The current_row and current_col fields will be declared with the rest of the RobotRat fields and initialized in the constructor or at their point of declaration.

Table 3-10: Fifth Iteration Design Considerations

The first feature from the list to be coded and tested should be the GetSpacesToMove() method. This method will read a string from the console and convert it into an integer value with the help of the System.Convert class.

The MoveForward() method requires you to do some intense study of the mechanics of a robot rat move. A picture like Figure 3.3 is very helpful in this particular case because it enables you to work out the moves of a robot rat upon the floor. You should work out the moves along the NORTH, SOUTH, EAST, and WEST directions and note how to execute a move in terms of the robot rat's current_row, current_col, and direction fields. You also need to manipulate the floor array element values when moving with the pen in the DOWN position, setting each element to true when the robot rat moves through that position on the floor.

It's a good idea to place the robot rat in a starting position. A good starting position to use is current_row = 0, current_col = 0, and direction = EAST. These attributes can either be initialized to the required values or states in the RobotRat constructor method or at their point of declaration.

Once you get the move mechanics worked out you can write a pseudocode description of the MoveForward() method like the one presented in Example 3.12.

3.12 MoveForward() method pseudocode

```
get spaces to move from user
if pen_position is UP do the following
   if direction is NORTH move RobotRat NORTH -- do not mark floor
   if direction is SOUTH move RobotRat SOUTH -- do not mark floor
   if direction is EAST move RobotRat EAST -- do not mark floor
   if direction is WEST move RobotRat WEST -- do not mark floor
if pen_position is DOWN
   if direction is NORTH move RobotRat NORTH -- mark floor
   if direction is SOUTH move RobotRat SOUTH -- mark floor
   if direction is EAST move RobotRat EAST -- mark floor
   if direction is WEST move RobotRat WEST -- mark floor
```

With your feature planning complete, you can now move to the code phase.

Code (Fifth Iteration)

Example 3.13 gives the source code for the GetSpacesToMove() method.

3.13 GetSpacesToMove() method

```
1    public int GetSpacesToMove(){
2       int spaces = 0;
3       String input;
4
5       Console.WriteLine("Please enter number of spaces to move: ");
6       input = Console.ReadLine();
7
8       if(input == String.Empty){
9         spaces = 0;
10      }else{
11         try{
12           spaces = Convert.ToInt32(input);
13
14         }catch(Exception){
15           spaces = 0;
16         }
17      }
18
19      return spaces;
20   }
```

Referring to Example 3.13 — two local variables are declared, one of type int named *spaces*, and the other of type String named *input*. The Console.ReadLine() method on line 6 reads a string from the console after the user has been prompted to enter the number of spaces to move. On line 8, the value of the input variable is compared to String.Empty. If it's empty, spaces is set to 0. If it's not empty, the code tries to convert the string into an integer. Notice that the statement that actually performs the conversion, line 12, is enclosed in a try/catch block. This is necessary because although the input string may not be empty, it may contain a string that does not properly convert into an integer. If an exception is thrown, the spaces variable is set to 0. Finally, the method returns the value of spaces.

The GetSpacesToMove() method can now be used in the MoveForward() method as is shown in Example 3.14.

3.14 MoveForward() method

```
public void MoveForward(){
      int spaces_to_move = GetSpacesToMove();

      switch(pen_position){
        case PenPositions.UP   :
             switch(direction){
                case Directions.NORTH :
                     if((current_row - spaces_to_move) < 0){
                          current_row = 0;
                       }else{
                          current_row = current_row - spaces_to_move;
                        }
                        break;
                 case Directions.SOUTH :
                      if((current_row + spaces_to_move) > (floor.GetLength(1) - 1)){
                current_row = (floor.GetLength(1) - 1);
             }else{
          current_row = current_row + spaces_to_move;
                        }
                        break;
```

```
                   case Directions.EAST :
                        if((current_col + spaces_to_move) > (floor.GetLength(0) - 1)){
                            current_col = (floor.GetLength(0) - 1);
               }else{
            current_col = current_col + spaces_to_move;
                          }
                          break;
                   case Directions.WEST :
                        if((current_col - spaces_to_move) < 0){
                            current_col = 0;
                         }else{
                           current_col = current_col - spaces_to_move;
                         }
                        break;
                   }
                   break;
         case PenPositions.DOWN :
              switch(direction){
                 case Directions.NORTH :
                      while((current_row > 0) && (spaces_to_move-- > 0)){
                         floor[current_row--, current_col] = true;
                       }
                      break;
                  case Directions.SOUTH :
                       while((current_row < floor.GetLength(0) - 1) && (spaces_to_move-- > 0)){
                floor[current_row++, current_col] = true;
                        }
                        break;
                  case Directions.EAST :
                       while((current_col < floor.GetLength(1) - 1) && (spaces_to_move-- > 0)){
           floor[current_row, current_col++] = true;
                        }
                        break;
                  case Directions.WEST :
                       while((current_col > 0) && (spaces_to_move-- > 0)){
         floor[current_row, current_col--] = true;
                        }
                        break;
              }
              break;
       }
     }
```

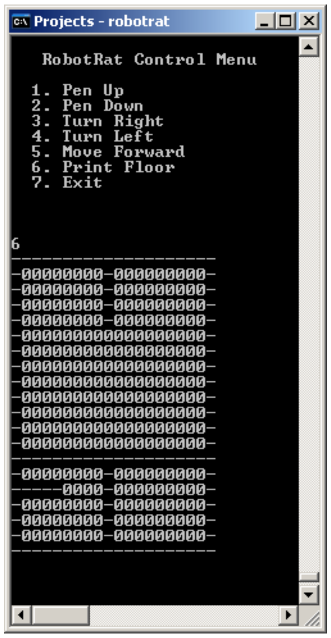
Referring to Example 3.14 — the MoveForward() method contains nested `switch` statements. The outer `switch` statement evaluates the value of the robot rat's pen_position field. The inner `switch` statements evaluate the value of the robot rat's direction field. Thus, the code performs the appropriate form of movement processing by controlling the value of these two variables by changing the position of the pen (UP or DOWN) and the robot rat's direction (NORTH, SOUTH, EAST, or WEST).

Test (Fifth Iteration)

This will be the most extensive test session of the RobotRat project yet. You must test the MoveForward() method in all directions and ensure you are properly handling move requests that attempt to go beyond the bounds of the floor array. Figure 3-14 shows a screen shot of the floor after completing a series of moves.

Integrate/Test (Fifth Iteration)

At this point in the development cycle, you will want to test the entire integrated RobotRat project. Move the robot rat with the pen up and pen down. Move in all directions and try making and printing different floor patterns. The emphasis in this phase is to test all RobotRat functionality, noting the effects of the latest features on existing functionality.



```
  RobotRat Control Menu

 1. Pen Up
 2. Pen Down
 3. Turn Right
 4. Turn Left
 5. Move Forward
 6. Print Floor
 7. Exit



6
--------------------
-00000000-000000000-
-00000000-000000000-
-00000000-000000000-
-00000000-000000000-
-000000000000000000-
-000000000000000000-
-000000000000000000-
-000000000000000000-
-000000000000000000-
-000000000000000000-
-000000000000000000-
-000000000000000000-
--------------------
-00000000-000000000-
-----0000-000000000-
-00000000-000000000-
-00000000-000000000-
-00000000-000000000-
--------------------
```

Figure 3-14: Testing Robot Rat Movement in All Directions

Final Considerations

You have now completed the core development efforts, but I would hesitate calling the Robot Rat project finished just yet. After you have finished the last integration and testing cycle, you will want to revisit and edit the code for neatness and clarity. Perhaps you can do a better job of formatting the code so it's easier to read. Maybe you thought of a few cool features to add to the project to get extra credit.

When you have reached the point where you feel you are done with the project, you will want to give it one last review. Table 3-11 lists a few things to review before submitting your project to your instructor.

Check-Off	Review	What To Check For
	Source code formatting	Ensure it is neat, consistently aligned, and indented to aid readability.
	Comments	Make sure they're used to explain critical parts of your code and that they are consistently formatted.
	File comment header	Add a file comment header at the top of all project source files. Make sure it has your name, class, instructor, and the name of the project. The file comment header format may be dictated by your instructor or by coding guidelines established at work.
	Printed copies of source code files	Make sure that it fits on a page in a way that preserves formatting and readability. Adjust the font size or paper orientation if required to ensure your project looks professional.
	Class files on floppy disk, CD-ROM, or USB memory stick (i.e., removable media)	Ensure all the required source and executable files are present. Try running your project from the removable medium to make sure you have included all the required files.

Table 3-11: Final Project Review Checklist

Complete RobotRat.cs Source Code Listing

3.15 RobotRat.cs (Complete Listing)

```
using System;

/// <summary>
/// RobotRat class lets a user control the movements
/// of a robot rat around a floor represented by
/// a 2-dimensional array.
/// </summary>
public class RobotRat
{

    private bool keep_going = true;
    private const char PEN_UP = '1';
    private const char PEN_DOWN = '2';
    private const char TURN_RIGHT = '3';
    private const char TURN_LEFT = '4';
    private const char MOVE_FORWARD = '5';
    private const char PRINT_FLOOR = '6';
    private const char EXIT = '7';
    private enum PenPositions { UP, DOWN };
    private enum Directions { NORTH, SOUTH, EAST, WEST };
    private PenPositions pen_position = PenPositions.UP;
    private Directions direction = Directions.EAST;
    private bool[,] floor;
    private int current_row = 0;
    private int current_col = 0;

    /// <summary>
    /// Constructor method.
    /// </summary>
    /// <param name="rows">Integer value representing number of floor rows</param>
    /// <param name="cols">Integer value representing number of floor columns</param>
    public RobotRat(int rows, int cols)
    {
        Console.WriteLine("RobotRat Lives!");
        floor = new bool[rows, cols];
    }

    /// <summary>
    /// Prints the menu to the screen.
    /// </summary>
    public void PrintMenu()
    {
        Console.WriteLine("\n\n");
        Console.WriteLine("   RobotRat Control Menu");
        Console.WriteLine();
        Console.WriteLine("  1. Pen Up");
        Console.WriteLine("  2. Pen Down");
        Console.WriteLine("  3. Turn Right");
        Console.WriteLine("  4. Turn Left");
        Console.WriteLine("  5. Move Forward");
        Console.WriteLine("  6. Print Floor");
        Console.WriteLine("  7. Exit");
        Console.WriteLine("\n\n");
    }

    /// <summary>
    /// Processes user's menu selection.
    /// </summary>
    public void ProcessMenuChoice()
    {
        String input = Console.ReadLine();

        if (input == String.Empty)
        {
            input = "0";
        }

        switch (input[0])
        {
            case PEN_UP: SetPenUp();
                break;
            case PEN_DOWN: SetPenDown();
                break;
            case TURN_RIGHT: TurnRight();
                break;
            case TURN_LEFT: TurnLeft();
                break;
            case MOVE_FORWARD: MoveForward();
                break;
            case PRINT_FLOOR: PrintFloor();
                break;
            case EXIT: keep_going = false;
                break;
            default: PrintErrorMessage();
                break;
        }
    }

    /// <summary>
    /// Sets the pen to the UP state.
    /// </summary>
    public void SetPenUp()
    {
        pen_position = PenPositions.UP;
        Console.WriteLine("The pen is " + pen_position);
    }

    /// <summary>
    /// Sets the pen to the DOWN state.
    /// </summary>
    public void SetPenDown()
    {
        pen_position = PenPositions.DOWN;
        Console.WriteLine("The pen is " + pen_position);
    }


    /// <summary>
    /// Turns the robot rat right.
    /// </summary>
    public void TurnRight()
    {
        switch (direction)
        {
            case Directions.NORTH: direction = Directions.EAST;
                break;
            case Directions.EAST: direction = Directions.SOUTH;
                break;
            case Directions.SOUTH: direction = Directions.WEST;
                break;
            case Directions.WEST: direction = Directions.NORTH;
                break;
        }

        Console.WriteLine("Direction is " + direction);
    }



    /// <summary>
    /// Turns the robot rat left.
    /// </summary>
    public void TurnLeft()
    {
        switch (direction)
        {
            case Directions.NORTH: direction = Directions.WEST;
                break;
            case Directions.WEST: direction = Directions.SOUTH;
                break;
            case Directions.SOUTH: direction = Directions.EAST;
                break;
            case Directions.EAST: direction = Directions.NORTH;
                break;
        }

        Console.WriteLine("Direction is " + direction);
    }


    /// <summary>
    /// Prints the floor pattern to the console.
    /// </summary>
    public void PrintFloor()
```

```
      {
          for (int i = 0; i < floor.GetLength(0); i++)
          {
              for (int j = 0; j < floor.GetLength(1); j++)
              {
                  if (floor[i, j])
                  {
                      Console.Write('-');
                  }
                  else
                  {
                      Console.Write('0');
                  }
              }
              Console.WriteLine();
          }
      }

      /// <summary>
      /// Prints an error message. Called if a user enters an invalid
      /// menu choice.
      /// </summary>
      public void PrintErrorMessage()
      {
          Console.WriteLine("Please enter a valid RobotRat control option!");
      }


      /// <summary>
      /// This method continuously displays the menu
      /// and processes user menu choices.
      /// </summary>
      public void Run()
      {
          while (keep_going)
          {
              PrintMenu();
              ProcessMenuChoice();
          }
      }

      /// <summary>
      /// Called to move the robot rat forward.
      /// </summary>
      public void MoveForward()
      {
          int spaces_to_move = GetSpacesToMove();

          switch (pen_position)
          {
              case PenPositions.UP: switch (direction)
                  {
                      case Directions.NORTH:
                          if ((current_row - spaces_to_move) < 0)
                          {
                              current_row = 0;
                          }
                          else
                          {
                              current_row = current_row - spaces_to_move;
                          }
                          break;
                      case Directions.SOUTH:
                          if ((current_row + spaces_to_move) > (floor.GetLength(0) - 1))
                          {
                              current_row = (floor.GetLength(1) - 1);
                          }
                          else
                          {
                              current_row = current_row + spaces_to_move;
                          }
                          break;
                      case Directions.EAST:
                          if ((current_col + spaces_to_move) > (floor.GetLength(1) - 1))
                          {
                              current_col = (floor.GetLength(0) - 1);
                          }
                          else
                          {
                              current_col = current_col + spaces_to_move;
                          }
```

```
                        break;
                    case Directions.WEST:
                        if ((current_col - spaces_to_move) < 0)
                        {
                            current_col = 0;
                        }
                        else
                        {
                            current_col = current_col - spaces_to_move;
                        }
                        break;
                }
                break;
            case PenPositions.DOWN: switch (direction)
                {
                    case Directions.NORTH:
                        while ((current_row > 0) && (spaces_to_move-- > 0))
                        {
                            floor[current_row--, current_col] = true;
                        }
                        break;
                    case Directions.SOUTH:
                        while ((current_row < floor.GetLength(0) - 1) && (spaces_to_move-- > 0))
                        {
                            floor[current_row++, current_col] = true;
                        }
                        break;
                    case Directions.EAST:
                        while ((current_col < floor.GetLength(1) - 1) && (spaces_to_move-- > 0))
                        {
                            floor[current_row, current_col++] = true;
                        }
                        break;
                    case Directions.WEST:
                        while ((current_col > 0) && (spaces_to_move-- > 0))
                        {
                            floor[current_row, current_col--] = true;
                        }
                        break;
                }
                break;


        }
    }

    /// <summary>
    ///  Gets the number of spaces to move from the user.
    /// </summary>
    /// <returns></returns>
    public int GetSpacesToMove()
    {
        int spaces = 0;
        String input;

        Console.WriteLine("Please enter number of spaces to move: ");
        input = Console.ReadLine();

        if (input == String.Empty)
        {
            spaces = 0;
        }
        else
        {
            try
            {
                spaces = Convert.ToInt32(input);

            }
            catch (Exception)
            {
                spaces = 0;
            }
        }

        return spaces;
    }


    /// <summary>
    /// The RobotRat's Main method.
    /// </summary>
    /// <param name="args"></param>
    public static void Main(String[] args)
    {
        RobotRat rr = new RobotRat(20, 20);
        rr.Run();
    }

}
```

This listing was automatically formatted with Microsoft C# Express Edition. Comments were added to describe the function of each method. You can automatically generate Extensible Markup Language (XML) documentation for this class file by compiling the RobotRat.cs file with the /doc:[documentation output filename] option. For example, to generate an XML documentation file named robotrat_docs.xml compile the robotrat.cs file by entering the following at the command line:

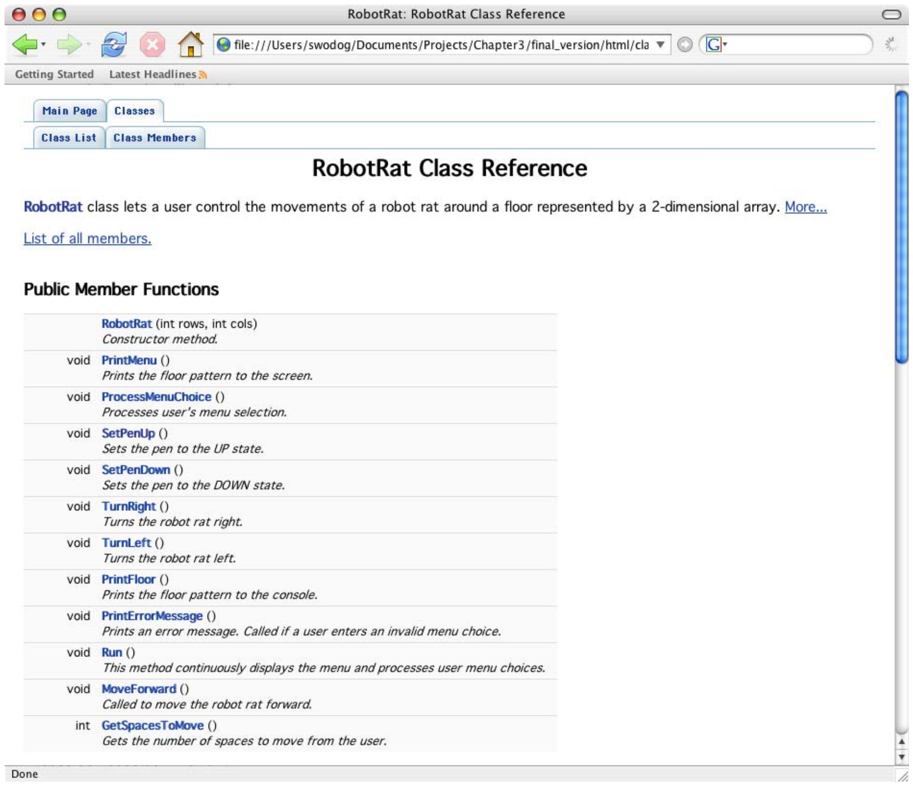
```
csc robotrat.cs /doc:robotrat_docs.xml
```

Example 3.16 shows the results.

3.16 robotrat_docs.xml

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>RobotRat</name>
    </assembly>
    <members>
        <member name="T:RobotRat">
            <summary>
            RobotRat class lets a user control the movements
            of a robot rat around a floor represented by
            a 2-dimensional array.
            </summary>
        </member>
        <member name="M:RobotRat.#ctor(System.Int32,System.Int32)">
            <summary>
            Constructor method.
            </summary>
            <param name="rows">Integer value representing number of floor rows</param>
            <param name="cols">Integer value representing number of floor columns</param>
        </member>
        <member name="M:RobotRat.PrintMenu">
            <summary>
            Prints the floor pattern to the screen.
            </summary>
        </member>
        <member name="M:RobotRat.ProcessMenuChoice">
            <summary>
            Processes user's menu selection.
            </summary>
        </member>
        <member name="M:RobotRat.SetPenUp">
            <summary>
            Sets the pen to the UP state.
            </summary>
        </member>
        <member name="M:RobotRat.SetPenDown">
            <summary>
            Sets the pen to the DOWN state.
            </summary>
        </member>
        <member name="M:RobotRat.TurnRight">
            <summary>
            Turns the robot rat right.
            </summary>
        </member>
        <member name="M:RobotRat.TurnLeft">
            <summary>
            Turns the robot rat left.
            </summary>
        </member>
        <member name="M:RobotRat.PrintFloor">
            <summary>
            Prints the floor pattern to the console.
            </summary>
        </member>
        <member name="M:RobotRat.PrintErrorMessage">
            <summary>
            Prints an error message. Called if a user enters an invalid
            menu choice.
            </summary>
        </member>
        <member name="M:RobotRat.Run">
            <summary>
            This method continuously displays the menu
            and processes user menu choices.
            </summary>
        </member>
        <member name="M:RobotRat.MoveForward">
            <summary>
            Called to move the robot rat forward.
            </summary>
        </member>
        <member name="M:RobotRat.GetSpacesToMove">
            <summary>
             Gets the number of spaces to move from the user.
            </summary>
            <returns></returns>
        </member>
        <member name="M:RobotRat.Main(System.String[])">
            <summary>
            The RobotRat's Main method.
            </summary>
            <param name="args"></param>
        </member>
    </members>
</doc>
```

Now, XML isn't pretty to look at and is not at all easily readable by humans. If you want to create professional grade documentation in HTML or another format, you can use an open-source documentation generator like NDoc [http://ndoc.sourceforge.net] or Doxygen [http://www.stack.nl/~dimitri/doxygen/]. **Note:** At the time of this writing NDoc does not work with .NET Framework version 2.0 or greater. These tools generate documentation from commented source files. Figure 3-15 shows a partial screen capture from the documentation generated from the final version of the RobotRat.cs file using Doxygen on a Macintosh running OSX 10.3.


RobotRat: RobotRat Class Reference
file:///Users/swodog/Documents/Projects/Chapter3/final_version/html/cla
Getting Started
Latest Headlines
Main Page
Classes
Class List
Class Members
Done

RobotRat Class Reference

RobotRat class lets a user control the movements of a robot rat around a floor represented by a 2-dimensional array. More...

List of all members.

Public Member Functions

	RobotRat (int rows, int cols) *Constructor method.*
void	PrintMenu () *Prints the floor pattern to the screen.*
void	ProcessMenuChoice () *Processes user's menu selection.*
void	SetPenUp () *Sets the pen to the UP state.*
void	SetPenDown () *Sets the pen to the DOWN state.*
void	TurnRight () *Turns the robot rat right.*
void	TurnLeft () *Turns the robot rat left.*
void	PrintFloor () *Prints the floor pattern to the console.*
void	PrintErrorMessage () *Prints an error message. Called if a user enters an invalid menu choice.*
void	Run () *This method continuously displays the menu and processes user menu choices.*
void	MoveForward () *Called to move the robot rat forward.*
int	GetSpacesToMove () *Gets the number of spaces to move from the user.*

Figure 3-15: Robot Rat HTML Documentation Generated with Doxygen

Summary

Use the project-approach strategy to systematically produce a solution to a programming problem. The purpose of the project-approach strategy is to help novice programmers maintain a sense of forward momentum during their project-development activities. The project-approach strategy comprises four strategy areas: application requirements, problem domain, language features, and high-level design and implementation strategy. The project approach strategy can be tailored to suit your needs.

Use the development cycle to methodically implement your projects. Apply the plan, code, test, and integrate steps iteratively in a tight-spiral fashion.

When you've completed your project, review it to ensure it meets all submission requirements related to comment headers, formatting, neatness, and the way it appears when printed. If you submit your project on a floppy disk or another type of removable medium, double-check to ensure you've included all the files necessary to run the project.

Skill-Building Exercises

1. **Project-Approach Strategy:** Review the four project-approach strategy areas. Write a brief explanation of the purpose of each strategy area.

2. **Project-Approach Strategy:** Tailor the project-approach strategy to better suit your needs. What strategy areas would you add, delete, or modify? Explain your rationale.

3. **Development Cycle:** Review the phases of the development cycle. Write a brief description of each phase.

4. **UML Tool:** Obtain a UML tool. Explore it capabilities. Use it to create class and state transition diagrams.

5. **Documentation Generator:** Download and install one of the documentation generators mentioned in this chapter. Try generating HTML documentation from the final version of the RobotRat.cs file given in Example 3.15.

6. **Microsoft's Documentation Generator:** Microsoft removed documentation generating capabilities from Visual Studio 2005. However, they made their internal documentation generator, Sandcastle, available for public download from their website. Download Sandcastle and try to use it to generate RobotRat class documentation.

Suggested Projects

1. **Project-Approach Strategy and Development Cycle:** Apply the project-approach strategy and development cycle to your programming projects.

Self-Test Questions

1. List and describe the four project-approach strategy areas.

2. What is the purpose of the project-approach strategy?

3. List and describe the four phases of the development cycle demonstrated in this chapter.

4. How should the development cycle be applied to a programming project?

5. What is method stubbing?

6. What is the purpose of method stubbing?

7. When should you first compile and test your programming project?

8. List at least three aspects of your project you should double-check before your turn it in to your instructor.

10. What is the purpose of a state transition diagram?

References

Microsoft Developer Network (MSDN) [www.msdn.com]

Doxygen website [http://www.stack.nl/~dimitri/doxygen/]

NDoc SourceForge project website: [http://ndoc.sourceforge.net]

Notes