O'REILLY®

Fourth Edition

Head First C#

A Learner's Guide to Real-World Programming with C# and .NET Core

Andrew Stellman
& Jennifer Greene

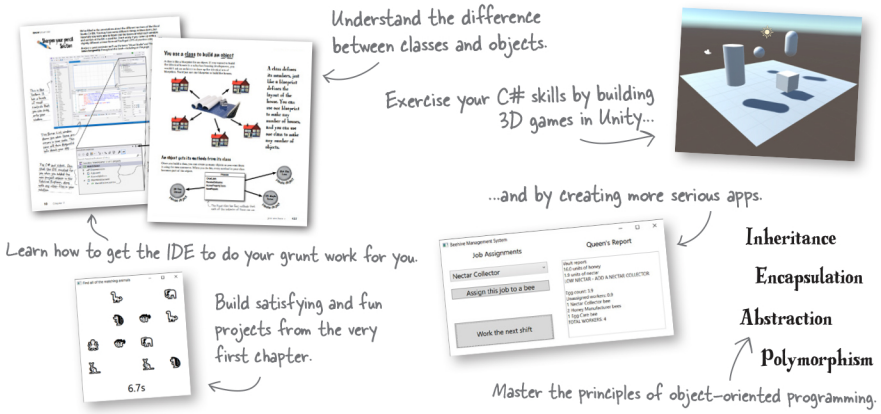
A Brain-Friendly Guide

Head First

C#

What will you learn from this book?

Dive into C# and create apps, user interfaces, games, and more using this fun and highly visual introduction to C#, .NET Core, and Visual Studio. With this completely updated guide, which covers C# 8.0 and Visual Studio 2019, beginning programmers like you will build a fully functional game in the opening chapter. Then you'll learn how to use classes and object-oriented programming, create 3D games in Unity, and query data with LINQ. And you'll do it all by solving puzzles, doing hands-on exercises, and building real-world applications. By the time you're done, you'll be a solid C# programmer—and you'll have a great time along the way!



"Thank you so much! Your books have helped me to launch my career."

—Ryan White
Game Developer

"Andrew and Jennifer have written a concise, authoritative, and most of all, fun introduction to C# development."

—Jon Galloway
Senior Program Manager on the .NET Community Team at Microsoft

"If you want to learn C# in depth and have fun doing it, this is THE book for you."

—Andy Parker
Fledgling C# programmer

What's so special about this book?

Based on the latest research in cognitive science and learning theory, *Head First C#* uses a visually rich format to engage your mind rather than a text-heavy approach that puts you to sleep. Why waste your time struggling with new concepts? This multisensory learning experience is designed for the way your brain really works.

.NET

US $64.99 **CAN $85.99**

ISBN: 978-1-491-97670-8

9 781491 976708 56499

O'REILLY®

Head First C#

Fourth Edition

Andrew Stellman
Jennifer Greene

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First C#

Fourth Edition

by Andrew Stellman and Jennifer Greene

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators:	Kathy Sierra, Bert Bates
Cover Designer:	Ellie Volckhausen
Brain Image on Spine:	Eric Freeman
Editors:	Nicole Taché, Amanda Quinn
Proofreader:	Rachel Head
Indexer:	Potomac Indexing, LLC
Illustrator:	Jose Marzan
Page Viewers:	Greta the miniature bull terrier and Samosa the Pomeranian

Printing History:

November 2007: First Edition.
May 2010: Second Edition.
August 2013: Third Edition.
December 2020: Fourth Edition

No bees, space aliens, or comic book heroes were harmed in the making of this book.

ISBN: 978-1-491-97670-8

[LSI] [2020-11-13]

Downloadable exercise: Hide and Seek

In this next exercise you'll build an app where you explore a house and play a game of Hide and Seek against a computer player. You'll put your collection and interface skills to the test when you lay out the locations. Then you'll turn it into a game, serializing the state of the game to a file so you can save and load it.

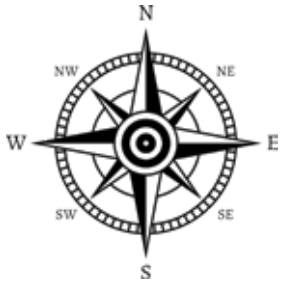
One of the most important ideas we've emphasized throughout this book is that writing C# code is a skill, and the best way to improve that skill is to ***get lots of practice***. We want to give you as many opportunities to get practice as possible!

That's why we created **additional projects** to go with some of the chapters in the rest of this book. We've included these projects at the end of the next few chapters too. We think you should take the time and do this project before moving on to the next chapter, because it will help reinforce some important concepts that will help you with the material in the rest of the book.

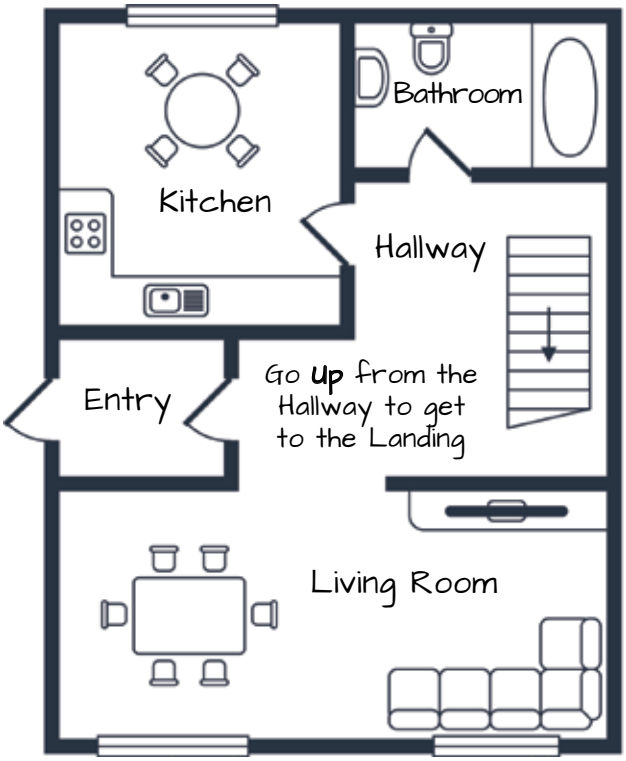
In this project, you'll get practice with the concepts you've learned in the last few chapters. You'll build on what you learned about **test-driven development** in the Chapter 9 project, taking it to the next level by building your own unit tests first, and then implementing the classes so they pass those tests.

One important goal of this project is to give you practice with important development skills that help you design and build your code: unit testing, test-driven development, and even some project planning skills. These skills are valuable for any kind of project, and especially important for anyone who wants to do software development professionally.

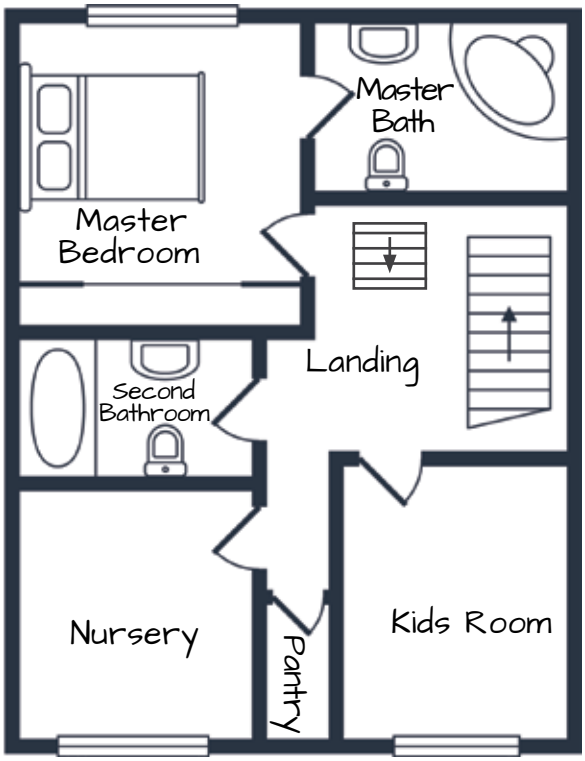
Let's build a house!



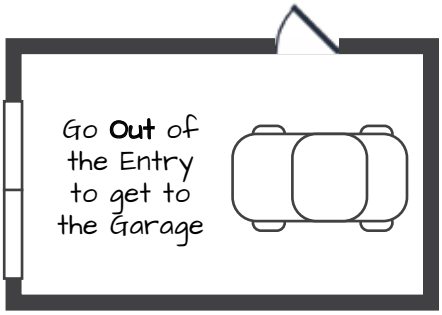
In this project, you'll be creating a game where you play hide-and-seek in a virtual house against computer opponents. Here's the layout of the house. It has two floors, a garage, and an attic. The player will use directions to navigate through the house: they'll go East from the entry to get to the hallway, and from there they'll go Northwest to the kitchen or up to the landing.



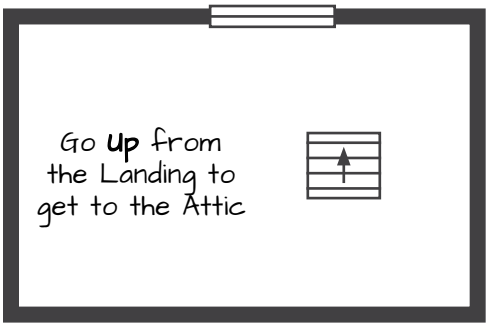
First Floor



Second Floor



Garage



Attic

You'll use test-driven development to build your game in increments

You've been doing larger and larger projects as you've gotten further through the book. To do them, you've been breaking them down into **increments**, or smaller "mini-projects" that you can complete individually, one after the other. You'll take the same incremental approach to this project—***and test-driven development will help***.

We learned about test-driven development in the Chapter 9 *Go Fish!* project. It's called "test-driven" because you write the tests for your classes before you write the code for them. Here's a quick recap of how that worked:

- You **started with a class diagram**, which was drawn if a class that depends on another class is above it in the diagram.
- You created a **skeleton** for the class at the bottom of the diagram. It had all of the class members, but each member was a **stub**—or a temporary piece of code that will eventually be replaced by working code. Each stub throws a NotImplementedException.
- You **added unit tests** for that class. When you first added the tests, they all failed because the stub methods just throw exceptions.
- You **implemented the class**. You knew that your class was done when all of your tests passed.
- You moved to the **next class in the diagram**, creating a skeleton, adding unit tests that failed at first, and implementing the class so the unit tests passed.
- Once once you got to the top of the class diagram, you were done.

Test-driven development makes it easier for you to take an incremental approach to your projects, because you can treat each individual class like its own mini-project.

Test-driven development can make your projects easier because it ***helps you follow a plan***: start at the bottom of the class diagram and work your way up, building each class and only moving on to the next one when it's done. That's especially valuable when you're taking an **incremental** approach to your projects—building them in parts—and that's how we'll do this project: we'll create the class diagram for the first part of the project where you explore the house, then we'll update the diagram for the second part, where you play hide-and-seek.

We're giving you more freedom with how you do this project. That means we'll give you fewer instructions—and we're not including solution code.

These projects are getting pretty big! The larger a project is, the more likely it is that your code will look very different from ours. In fact, it may be so different that we don't want to include the solution code in this PDF—not just because it could be very different than yours, but also because it's getting really long, and the solution will end up being many pages of code.

***You can see our solution on GitHub**—and remember, it's **not cheating** to look at our solution if you get stuck: https://github.com/head-first-csharp/fourth-edition/tree/master/Code/Chapter_10*

Part 1: Build an app to explore the house

Here's what it will look like when you run your app:

```
You are in the Entry. You see the following exits:
 - the Hallway is to the East
 - the Garage is Out
Which direction do you want to go: East
Moving East
You are in the Hallway. You see the following exits:
 - the Bathroom is to the North
 - the Living Room is to the South
 - the Entry is to the West
 - the Kitchen is to the Northwest
 - the Landing is Up
Which direction do you want to go: Northwest
Moving Northwest
You are in the Kitchen. You see the following exits:
 - the Hallway is to the Southeast
Which direction do you want to go: Southeast
Moving Southeast
You are in the Hallway. You see the following exits:
 - the Bathroom is to the North
 - the Living Room is to the South
 - the Entry is to the West
 - the Kitchen is to the Northwest
 - the Landing is Up
Which direction do you want to go: Up
Moving Up
You are in the Landing. You see the following exits:
 - the Pantry is to the South
 - the Second Bathroom is to the West
 - the Nursery is to the Southwest
 - the Kids Room is to the Southeast
 - the Master Bedroom is to the Northwest
 - the Attic is Up
 - the Hallway is Down
Which direction do you want to go: Northwest
Moving Northwest
You are in the Master Bedroom. You see the following exits:
 - the Master Bath is to the East
 - the Landing is to the Southeast
Which direction do you want to go: East
Moving East
You are in the Master Bath. You see the following exits:
 - the Master Bedroom is to the West
Which direction do you want to go: Down
There's no exit in that direction
You are in the Master Bath. You see the following exits:
 - the Master Bedroom is to the West
Which direction do you want to go: Baloney
That's not a valid direction
```

When you start the app, you're in the Entry. The floor plan says there are two exits: the Hallway is through a door to the East, and the Garage is outside.

The app prompts you for a direction. You'll enter a direction like Northwest, East, Up, or Out. If there's an exit from your current location in that direction, it will move you to a new location.

The Kitchen is Northwest of the Hallway. To get back to the Hallway from the Kitchen, you have to move in the opposite direction: Southeast

The app keeps track of your current location, lists all of the exits that connect to other locations.

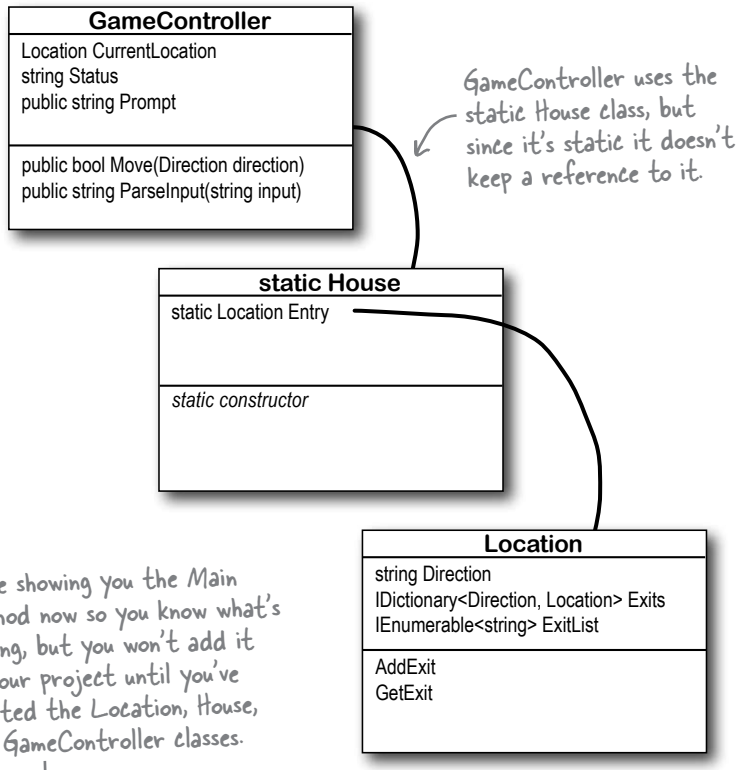
We went up from the Hallway to get to the Landing. Look closely at the floor plan and compare it with the list of exits that the app lists when you're in the Landing. You can use the floor plan and this output to figure out the complete layout of the house.

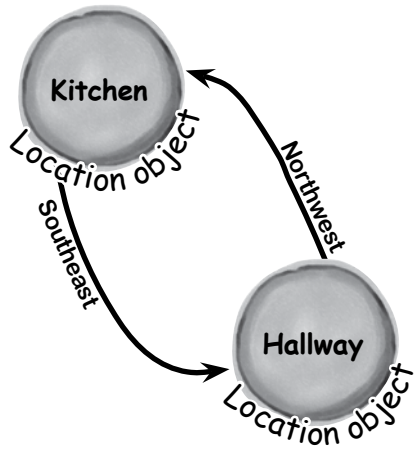
The Master Bath has only one exit, to the West. If you enter Down while in that location, the app will tell you there's no exit in that direction.

The app will let you know if you give it input that isn't a valid direction.

The class diagram for your house explorer

Here's the class diagram for the first part of the project, where you explore the house. It has three classes: Location, House, and GameController.





Each Location uses a Dictionary called Exits to keep track of the other locations that it's connected to. When one Location has an exit to another, that other Location has an exit back to it—for example, the Hallway has a Northwest exit to the Kitchen, which has a Southeast exit back to the hallway. The uses the Direction enum for the Dictionary key, and to figure out which direction the user wants to move.

We're showing you the Main method now so you know what's coming, but you won't add it to your project until you've created the Location, House, and GameController classes.

```
public enum Direction
{
    North = -1,
    South = 1,
    East = -2,
    West = 2,
    Northeast = -3,
    Southwest = 3,
    Southeast = -4,
    Northwest = 4,
    Up = -5,
    Down = 5,
    In = -6,
    Out = 6,
}
```

The Main method calls the GameController

It's just an infinite loop that writes the status, displays a prompt, and parses the input from the user. All of the "smarts" are in the other classes.

```
class Program
{
    static void Main(string[] args)
    {
        GameController gameController = new GameController();
        while (true)
        {
            Console.WriteLine(gameController.Status);
            Console.Write(gameController.Prompt);
            Console.WriteLine(gameController.ParseInput(Console.ReadLine()));
        }
    }
}
```

Long Exercise

Add the Location skeleton, write its unit tests, and then implement it. Start by **creating a new solution** called *HideAndSeek*, then **add an MSTest unit test project** to it, just like you did in Chapter 9. Here's the skeleton for the Location class. It has stub methods that throw exceptions. **Add it to your project.**

Here's the skeleton for the Location class:

Location

string Direction
IDictionary<Direction, Location> Exits
IEnumerable<string> ExitList

AddExit
GetExit

```
public class Location
{
    /// <summary>
    /// The name of this location
    /// </summary>
    public string Name { get; private set; }

    /// <summary>
    /// The exits out of this location
    /// </summary>
    public IDictionary<Direction, Location> Exits { get; private set; }
                                                  = new Dictionary<Direction, Location>();

    /// <summary>
    /// The constructor sets the location name
    /// </summary>
    /// <param name="name">Name of the location</param>
    public Location(string name) => throw new NotImplementedException();

    public override string ToString() => Name;

    /// <summary>
    /// Returns a sequence of descriptions of the exits, sorted by direction
    /// </summary>
    public IEnumerable<string> ExitList => throw new NotImplementedException();

    /// <summary>
    /// Adds an exit to this location
    /// </summary>
    /// <param name="direction">Direction of the connecting location</param>
    /// <param name="connectingLocation">Connecting location to add</param>
    public void AddExit(Direction direction, Location connectingLocation)
    {
        throw new NotImplementedException();
    }

    /// <summary>
    /// Gets the exit location in a direction
    /// </summary>
    /// <param name="direction">Direciton of the exit location</param>
    /// <returns>The exit location, or this if there is no exit in that direction</returns>
    public Location GetExit(Direction direction) => throw new NotImplementedException();
}
```

We gave you the Direction enum we gave you on the previous page. Add it to your project, too.

Add a new unit test class to your unit test project called *LocationTests*. Edit its project dependencies to **add a reference to the HideAndSeek project** (again, just like you did in Chapter 9). Your job is to figure write tests for the Location class. They'll fail when you first run them—you'll know you're done with the Location class when they all pass.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace HideAndSeekTests
{
    using HideAndSeek;
    using System.Collections.Generic;
    using System.Linq;

    [TestClass]
    public class LocationTests
    {
        private Location center;

        /// <summary>
        /// Initializes each unit test by setting creating a new the center location
        /// and adding a room in each direction before the test
        /// </summary>
        [TestInitialize]
        public void Initialize()
        {
            // You'll use this to create a bunch of locations before each test
        }

        /// <summary>
        /// Make sure GetExit returns the location in a direction only if it exists
        /// </summary>
        [TestMethod]
        public void TestGetExit()
        {
            // This test will make sure the GetExit method works
        }

        /// <summary>
        /// Validates that the exit lists are working
        /// </summary>
        [TestMethod]
        public void TestExitList()
        {
            // This test will make sure the ExitList property works
        }

        /// <summary>
        /// Validates that each room's name and return exit is created correctly
        /// </summary>
        [TestMethod]
        public void TestReturnExits()
        {
            // This test will test navigating through the center Location
        }

        /// <summary>
        /// Add a hall to one of the rooms and make sure the hall room's names
        /// and return exits are created correctly
        /// </summary>
        [TestMethod]
        public void TestAddHall()
        {
            // This test will add a hallway with two locations and make sure they work
        }
    }
}
```

Long Exercise

Continue building out the Location unit tests and updating the class to make them pass.

1 Add this useful private method.

Look closely at the output from the game—the lists of exits include phrases like `the Hallway is to the East` `the Garage is Out` and `the Landing is Up`. Here's a useful switch expression that will help you create that output.

```
/// <summary>
/// Describes a direction (e.g. "in" vs. "to the North")
/// </summary>
/// <param name="d">Direction to describe</param>
/// <returns>string describing the direction</returns>
private string DescribeDirection(Direction d) => d switch
{
    Direction.Up => "Up",
    Direction.Down => "Down",
    Direction.In => "In",
    Direction.Out => "Out",
    _ => $"to the {d}",
};
```

There's one thing to keep in mind: **your unit tests won't test private methods**. In Chapter 9 we talked about black box testing, where your unit tests only test public behavior. Your private methods will definitely be tested, but only because they're called by public methods that the tests call directly.

2 The test initializer creates a Center location with exits in each direction.

The unit test class we gave you has an Initialize method marked with the `[TestInitialize]` annotation. We saw in Chapter 9 that the annotation tells Visual Studio's test executor to run that method before every test. Start your tests by adding the code for the Initialize method. It should create a new Location instance and assign it to the `center` field in the test class. It should then call its AddExit ten times, adding an exit to a new location in each of the directions in the Directions enum.

We'll start you off with the first few lines of the Initialize method:

```
center = new Location("Center Room");
Assert.AreSame("Center Room", center.ToString());
Assert.AreEqual(0, center.ExitList.Count());

center.AddExit(Direction.North, new Location("North Room"));
```

This line creates a new Location and adds it as a North exit to the center room.

Add those lines to the Initialize method. Then add eleven more lines creating rooms in each direction. Give them names like East Room, Upper Room, and Outside Room.

End the initializer with this line to make sure it added ten exits:

```
Assert.AreEqual(10, center.ExitList.Count());
```

Try running your tests. They'll still fail, of course! They won't pass until you finish your Location class.

3 Add the TestGetExit method and implement Location.GetExit.

Here's the TestGetExit method. It uses GetExit to get the exit to the east of the Center room and uses Assert.AreEqual to check its name to make sure it got the correct exit. Then it uses Assert.AreSame to make sure that a GetExit returns a reference to the location if it's called with a direction where that room doesn't have an exit.

```
/// <summary>
/// Make sure GetExit returns the location in a direction only if it exists
/// </summary>
[TestMethod]
public void TestGetExit()
{
    var eastRoom = center.GetExit(Direction.East);
    Assert.AreEqual("East Room", eastRoom.Name);
    Assert.AreSame(center, eastRoom.GetExit(Direction.West));
    Assert.AreSame(eastRoom, eastRoom.GetExit(Direction.Up));
}
```

Now implement Location.GetExit. You'll know it works when your unit test passes.

4 Add the TextExitList method and implement the ExitList property.

Take another close look at the output that includes the list of exits. That exit list is created by calling string.join to join the list of exits returned by Location.ExitList. Use CollectionAssert.AreEqual to compare a list of strings against `center.ExitList.ToList()`. Pay attention to the order and capitalization of the exit list—use LINQ's OrderBy to make sure they're always in the same order.

5 Add the TestReturnExits method and make AddExit add the return exit.

Here's a useful private method to add a return exit—you'll call it from Location.AddExit:

```
/// <summary>
/// Adds a return exit to a connecting location
/// </summary>
/// <param name="direction">Direction of the connecting location</param>
/// <param name="connectingLocation">Location to add the return exit to</param>
private void AddReturnExit(Direction direction, Location connectingLocation) =>
          Exits.Add((Direction)(-(int) direction), connectingLocation);
```

Can you figure out a good way to test your AddExit method to make sure it generates return exits?

6 Implement TestAddHall and add any other tests you can think of.

We included a test called TestAddHall that adds an East exit to the East Room location, then another exit to the East of that one, and makes sure they both have the right number of exits. Try adding any other tests you can think of to make sure the Location class works.

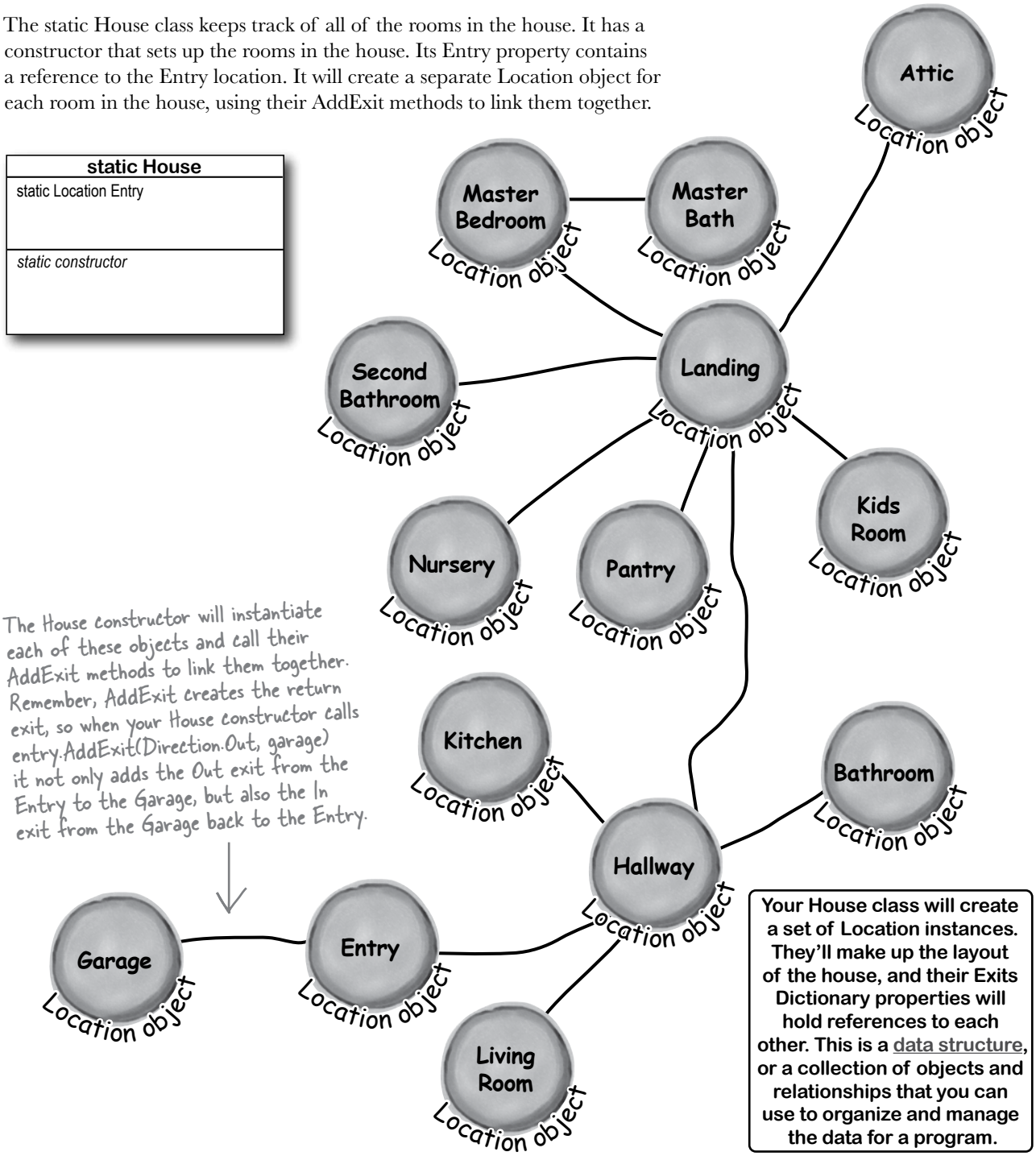
We won't include a solution in this PDF, because our solution will probably be different from ours. You can see our solution on the GitHub page for the book.

If you get stuck, you should definitely look at our solution! Make sure all Location tests pass before you move on.

The House class lays out the floor plan

The static House class keeps track of all of the rooms in the house. It has a constructor that sets up the rooms in the house. Its Entry property contains a reference to the Entry location. It will create a separate Location object for each room in the house, using their AddExit methods to link them together.

static House
static Location Entry
static constructor



The House constructor will instantiate each of these objects and call their AddExit methods to link them together. Remember, AddExit creates the return exit, so when your House constructor calls entry.AddExit(Direction.Out, garage) it not only adds the Out exit from the Entry to the Garage, but also the In exit from the Garage back to the Entry.

Your House class will create a set of Location instances. They'll make up the layout of the house, and their Exits Dictionary properties will hold references to each other. This is a data structure, or a collection of objects and relationships that you can use to organize and manage the data for a program.

Long Exercise

Add this unit test class that tests the House class. We're giving you a unit test class that navigates through the House and verifies the layout. Your job is to **build a House class** that makes this test pass.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace HideAndSeekTests
{
    using HideAndSeek;

    [TestClass]
    public class HouseTests
    {

        [TestMethod]
        public void TestLayout()
        {
            Assert.AreEqual("Entry", House.Entry.Name);

            var garage = House.Entry.GetExit(Direction.Out);
            Assert.AreEqual("Garage", garage.Name);

            var hallway = House.Entry.GetExit(Direction.East);
            Assert.AreEqual("Hallway", hallway.Name);

            var kitchen = hallway.GetExit(Direction.Northwest);
            Assert.AreEqual("Kitchen", kitchen.Name);

            var bathroom = hallway.GetExit(Direction. North);
            Assert.AreEqual("Bathroom", bathroom.Name);

            var livingRoom = hallway.GetExit(Direction.South);
            Assert.AreEqual("Living Room", livingRoom.Name);

            var landing = hallway.GetExit(Direction.Up);
            Assert.AreEqual("Landing", landing.Name);

            var masterBedroom = landing.GetExit(Direction.Northwest);
            Assert.AreEqual("Master Bedroom", masterBedroom.Name);

            var masterBath = masterBedroom.GetExit(Direction.East);
            Assert.AreEqual("Master Bath", masterBath.Name);

            var secondBathroom = landing.GetExit(Direction.West);
            Assert.AreEqual("Second Bathroom", secondBathroom.Name);

            var nursery = landing.GetExit(Direction.Southwest);
            Assert.AreEqual("Nursery", nursery.Name);

            var pantry = landing.GetExit(Direction.South);
            Assert.AreEqual("Pantry", pantry.Name);

            var kidsRoom = landing.GetExit(Direction.Southeast);
            Assert.AreEqual("Kids Room", kidsRoom.Name);

            var attic = landing.GetExit(Direction.Up);
            Assert.AreEqual("Attic", attic.Name);
        }
    }
}
```

The House class has two members: the Entry property that returns the starting location for the player, and the constructor that sets up the data structure. Remember, House is a static class, so use the static access modifier when you declare the members.

The GameController class manages the game

Go back a few pages in this PDF and have a look at the Main method. It's really short; all it does is use the GameController properties and methods to write the status to the console, write a prompt, and get input from the user. The GameController has the code to manage the game: it **parses** the input that the user types in, **moves the player** through the house, and **provides status** that the app can show the player.

Here's the skeleton for the GameController class.

```
public class GameController
{
    /// <summary>
    /// The player's current location in the house
    /// </summary>
    public Location CurrentLocation { get; private set; }

    /// <summary>
    /// Returns the the current status to show to the player
    /// </summary>
    public string Status => throw new NotImplementedException();

    /// <summary>
    /// A prompt to display to the player
    /// </summary>
    public string Prompt => "Which direction do you want to go: ";

    public GameController()
    {
        CurrentLocation = House.Entry;
    }

    /// <summary>
    /// Move to the location in a direction
    /// </summary>
    /// <param name="direction">The direction to move</param>
    /// <returns>True if the player can move in that direction, false oterwise</returns>
    public bool Move(Direction direction)
    {
        throw new NotImplementedException();
    }

    /// <summary>
    /// Parses input from the player and updates the status
    /// </summary>
    /// <param name="input">Input to parse</param>
    /// <returns>The results of parsing the input</returns>
    public string ParseInput(string input)
    {
        throw new NotImplementedException();
    }
}
```

GameController

Location CurrentLocation
string Status
public string Prompt
private House house

public bool Move(Direction direction)
public string ParseInput(string input)

We could have made the Move method private, since no other classes use it. We made it public so you could write separate tests for it. Did we make the right choice?

The ParseInput method parses a string that the user typed in. Parsing means analyzing text. You'll use the Enum.TryParse method, just like you did with card values in Chapter 9.

Long Exercise

Add this unit test class that tests the GameController class. We want the output to look a specific way, so we added tests to make sure it does. Add more tests, then **build a GameController class** that passes.

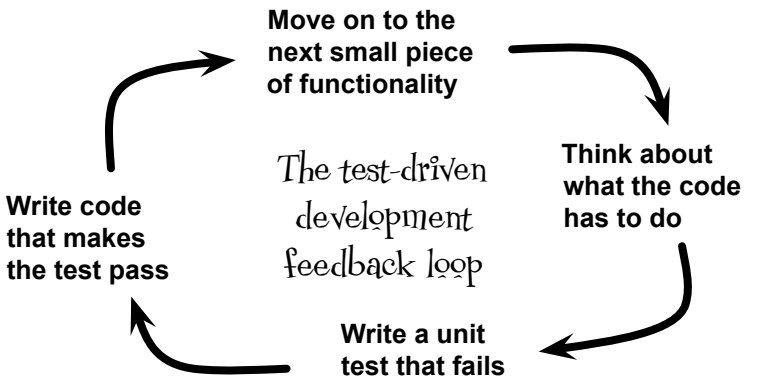
```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using HideAndSeek;
using System;

[TestClass]
public class GameControllerTests
{
    GameController gameController;

    [TestInitialize]
    public void Initialize()
    {
        gameController = new GameController();
    }

    [TestMethod]
    public void TestMovement()
    {
        Assert.AreEqual("Entry", gameController.CurrentLocation.Name);

        Assert.IsFalse(gameController.Move(Direction.Up));
        Assert.AreEqual("Entry", gameController.CurrentLocation.Name);

        Assert.IsTrue(gameController.Move(Direction.East));
        Assert.AreEqual("Hallway", gameController.CurrentLocation.Name);

        Assert.IsTrue(gameController.Move(Direction.Up));
        Assert.AreEqual("Landing", gameController.CurrentLocation.Name);

        // Add more movement tests to the TestMovement test method
    }

    [TestMethod]
    public void TestParseInput()
    {
        var initialStatus = gameController.Status;

        Assert.AreEqual("That's not a valid direction", gameController.ParseInput("X"));
        Assert.AreEqual(initialStatus, gameController.Status);

        Assert.AreEqual("There's no exit in that direction",
                        gameController.ParseInput("Up"));
        Assert.AreEqual(initialStatus, gameController.Status);

        Assert.AreEqual("Moving East", gameController.ParseInput("East"));
        Assert.AreEqual("You are in the Hallway. You see the following exits:" +
                        Environment.NewLine + " - the Bathroom is to the North" +
                        Environment.NewLine + " - the Living Room is to the South" +
                        Environment.NewLine + " - the Entry is to the West" +
                        Environment.NewLine + " - the Kitchen is to the Northwest" +
                        Environment.NewLine + " - the Landing is Up", gameController.Status);

        Assert.AreEqual("Moving South", gameController.ParseInput("South"));
        Assert.AreEqual("You are in the Living Room. You see the following exits:" +
            Environment.NewLine + " - the Hallway is to the North", gameController.Status);

        // Can you add more input parsing tests to the TestParseInput test method?
    }
}
```

We're using Environment.NewLine in TestParseInput instead of @ verbatim strings (like in Chapter 9). You can use verbatim strings instead, just be careful when you mix line breaks in verbatim strings with Environment.NewLine, because you may end up with a test that will pass on Windows but fail on macOS (or vice versa) with confusing test failure messages that show you two strings that appear identical but claim that they're different. The problem is that it's comparing stings that have macOS/Unix line endings \n with strings that have Windows line endings \r\n. We stuck with Environment.NewLine to keep our code cross-platform.

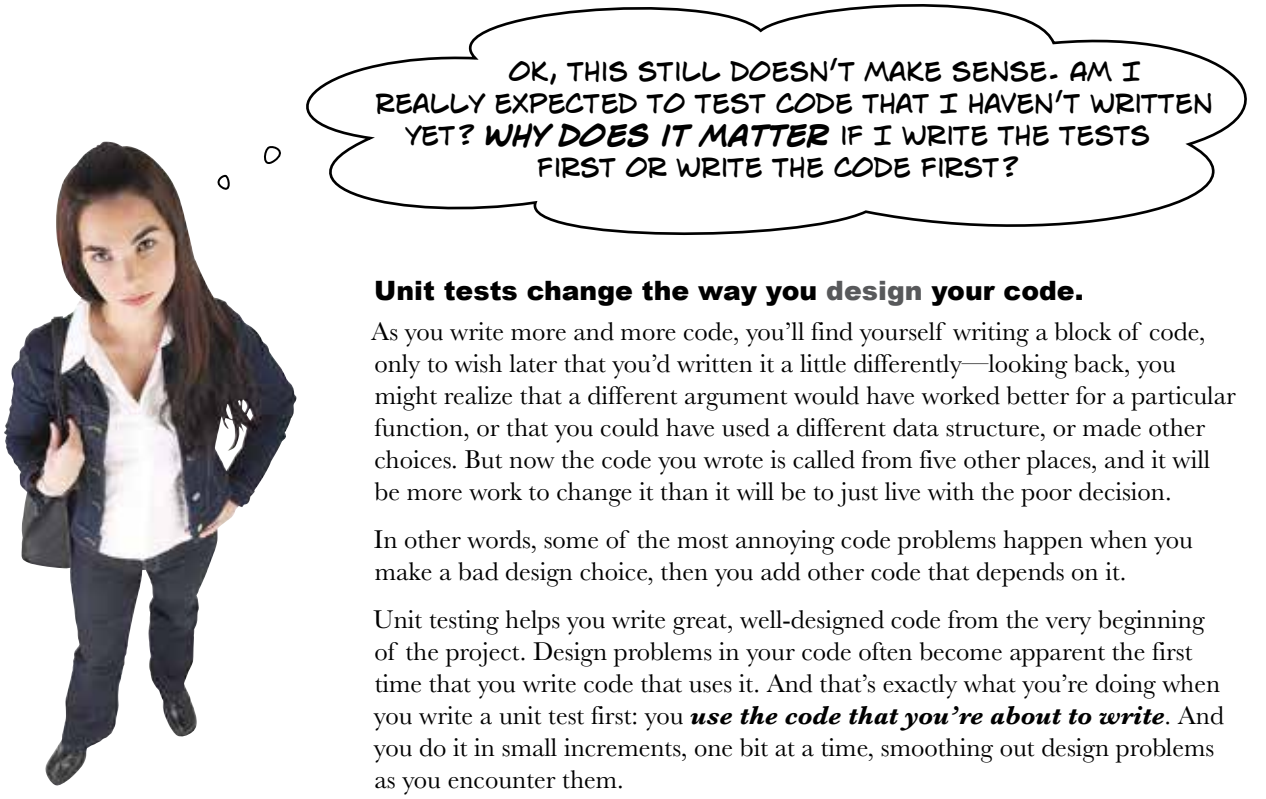
That's the end of part 1 of the project. Once all three classes pass their tests, add the Main method that we gave you earlier and you run it. Does the output match the sample output we gave you? If not, *add a failing test* that reproduces the problem, then update your code to fix it. That's how test-driven development helps you fix bugs.

Test-driven development makes it easier to write code

When you do **test-driven development** (or **TDD**), it means that you write unit tests *first*, and then you write the code that the tests validate. The test fails when you first write it—after you write the test, you write the code to make it pass. When you're in the habit of writing unit tests first, it helps you think about what it means for your code to work correctly, and a lot of developers find that it makes their projects come out better.



TDD forces you to really think through your code's behavior before you dive into writing it.



Unit tests change the way you design your code.

As you write more and more code, you'll find yourself writing a block of code, only to wish later that you'd written it a little differently—looking back, you might realize that a different argument would have worked better for a particular function, or that you could have used a different data structure, or made other choices. But now the code you wrote is called from five other places, and it will be more work to change it than it will be to just live with the poor decision.

In other words, some of the most annoying code problems happen when you make a bad design choice, then you add other code that depends on it.

Unit testing helps you write great, well-designed code from the very beginning of the project. Design problems in your code often become apparent the first time that you write code that uses it. And that's exactly what you're doing when you write a unit test first: you ***use the code that you're about to write***. And you do it in small increments, one bit at a time, smoothing out design problems as you encounter them.

```
public class House {
```

```
public class Location {
```

```
public class GameController {
```

Code is always divided into discrete units

In C#, those units are typically classes (but they could be individual methods or entire namespaces). You've written a lot of code going through this book. Take a few minutes and think back to some of your projects. How was the code divided up? Did you think about your code as being divided into discrete units when you were writing it?

Each unit gets its own unit tests

The name "unit testing" is pretty self-explanatory: you write tests for the units of code. In C#, unit testing is typically done on a class-by-class basis. Those tests are written in the same language as the rest of the code, and are stored in the same repository. The tests access whatever part of the unit is visible to the rest of the code. For your classes, that means the unit tests use the public methods and fields to make sure the class works.

```
public class HouseTests {
```

```
public class LocationTests {
```

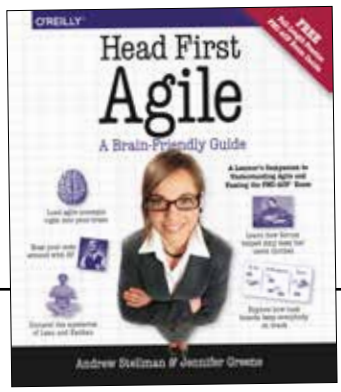
```
public class GameControllerTests {
```

Test-driven development is more than a technique. It's a mindset that helps you think differently about programming, design, and code. That's why TDD is a great habit to get into, because it helps you write code that's easier to go back and work with later.

Writing the unit tests first forces the developer to think about how the code is going to be used

Every unit of code is used by at least one other unit somewhere in the system—that's how code works. But when you're writing code, there's a paradox: in a lot of cases, you don't really know exactly how the unit you're working on will be used until you actually use it.

Test-driven development helps you catch problems in your code early, when they're much easier to fix. It's surprisingly easy to design a unit that's difficult to use later, and just as easy to "seal" in that poor design by writing additional units that depend on it. But if you write a small unit test every time you make a change to a unit, a lot of those design decisions become obvious.



Test-driven development is an important part of **agile development**, which teams all over the world use to work together and build great software. If you're curious about agile, check out our book, **Head First Agile**.

Part 2: Make it a game of hide-and-seek

You've set up a data structure that lays out the plans for a virtual house. Now you'll turn it into a game:

- You'll add hiding places to some of the locations.
- You'll add five opponents, who will scatter throughout the house and hide in those hiding places.
- You'll give the player a way to check hiding places. The game is over when all opponents are found!

Here's a sample run of the game. This player found all five opponents in 19 moves:

```
1: Which direction do you want to go: East
Moving East
You are in the Hallway. You see the following exits:
 - the Bathroom is to the North
 - the Living Room is to the South
 - the Entry is to the West
 - the Kitchen is to the Northwest
 - the Landing is Up
You have not found any opponents
2: Which direction do you want to go (or type 'check'): Northwest
Moving Northwest
You are in the Kitchen. You see the following exits:
 - the Hallway is to the Southeast
Someone could hide next to the stove
You have not found any opponents
3: Which direction do you want to go (or type 'check'): Check
You found 1 opponent hiding next to the stove
You are in the Kitchen. You see the following exits:
 - the Hallway is to the Southeast
Someone could hide next to the stove
You have found 1 of 5 opponents: Owen
4: Which direction do you want to go (or type 'check'): Southeast
Moving Southeast
You are in the Hallway. You see the following exits:
 - the Bathroom is to the North
 - the Living Room is to the South
 - the Entry is to the West
 - the Kitchen is to the Northwest
 - the Landing is Up
You have found 1 of 5 opponents: Owen
5: Which direction do you want to go (or type 'check'): Up
Moving Up
You are in the Landing. You see the following exits:
 - the Pantry is to the South
 - the Second Bathroom is to the West
 - the Nursery is to the Southwest
 - the Kids Room is to the Southeast
 - the Master Bedroom is to the Northwest
 - the Attic is Up
 - the Hallway is Down
You have found 1 of 5 opponents: Owen
```

Some of the locations have hiding places where an opponent can hide. For example, someone could hide next to the stove in the kitchen.

You'll update the GameController's parser so it checks the hiding place in the current location.

The game keeps track of which opponents the player has found so far.

The game keeps track of how many moves the player has made so far. Checking a hiding place counts as a move.

```
6: Which direction do you want to go (or type 'check'): West
Moving West
You are in the Second Bathroom. You see the following exits:
 - the Landing is to the East
Someone could hide in the shower
You have found 1 of 5 opponents: Owen
7: Which direction do you want to go (or type 'check'): Check
Nobody was hiding in the shower
You are in the Second Bathroom. You see the following exits:
 - the Landing is to the East
Someone could hide in the shower
You have found 1 of 5 opponents: Owen
8: Which direction do you want to go (or type 'check'): East
Moving East
You are in the Landing. You see the following exits:
 - the Pantry is to the South
 - the Second Bathroom is to the West
 - the Nursery is to the Southwest
 - the Kids Room is to the Southeast
 - the Master Bedroom is to the Northwest
 - the Attic is Up
 - the Hallway is Down
You have found 1 of 5 opponents: Owen
9: Which direction do you want to go (or type 'check'):
Northwest
Moving Northwest
You are in the Master Bedroom. You see the following exits:
 - the Master Bath is to the East
 - the Landing is to the Southeast
Someone could hide in the closet
You have found 1 of 5 opponents: Owen
10: Which direction do you want to go (or type 'check'): Check
You found 2 opponents hiding in the closet
You are in the Master Bedroom. You see the following exits:
 - the Master Bath is to the East
 - the Landing is to the Southeast
Someone could hide in the closet
You have found 3 of 5 opponents: Owen, Joe, Bob
11: Which direction do you want to go (or type 'check'):

...

You have found 4 of 5 opponents: Owen, Joe, Bob, Ana
17: Which direction do you want to go (or type 'check'): South
Moving South
You are in the Pantry. You see the following exits:
 - the Landing is to the North
Someone could hide inside a cabinet
You have found 4 of 5 opponents: Owen, Joe, Bob, Ana
18: Which direction do you want to go (or type 'check'): Check
You found 1 opponent hiding inside a cabinet
You won the game in 19 moves!
Press P to play again, any other key to quit.
```

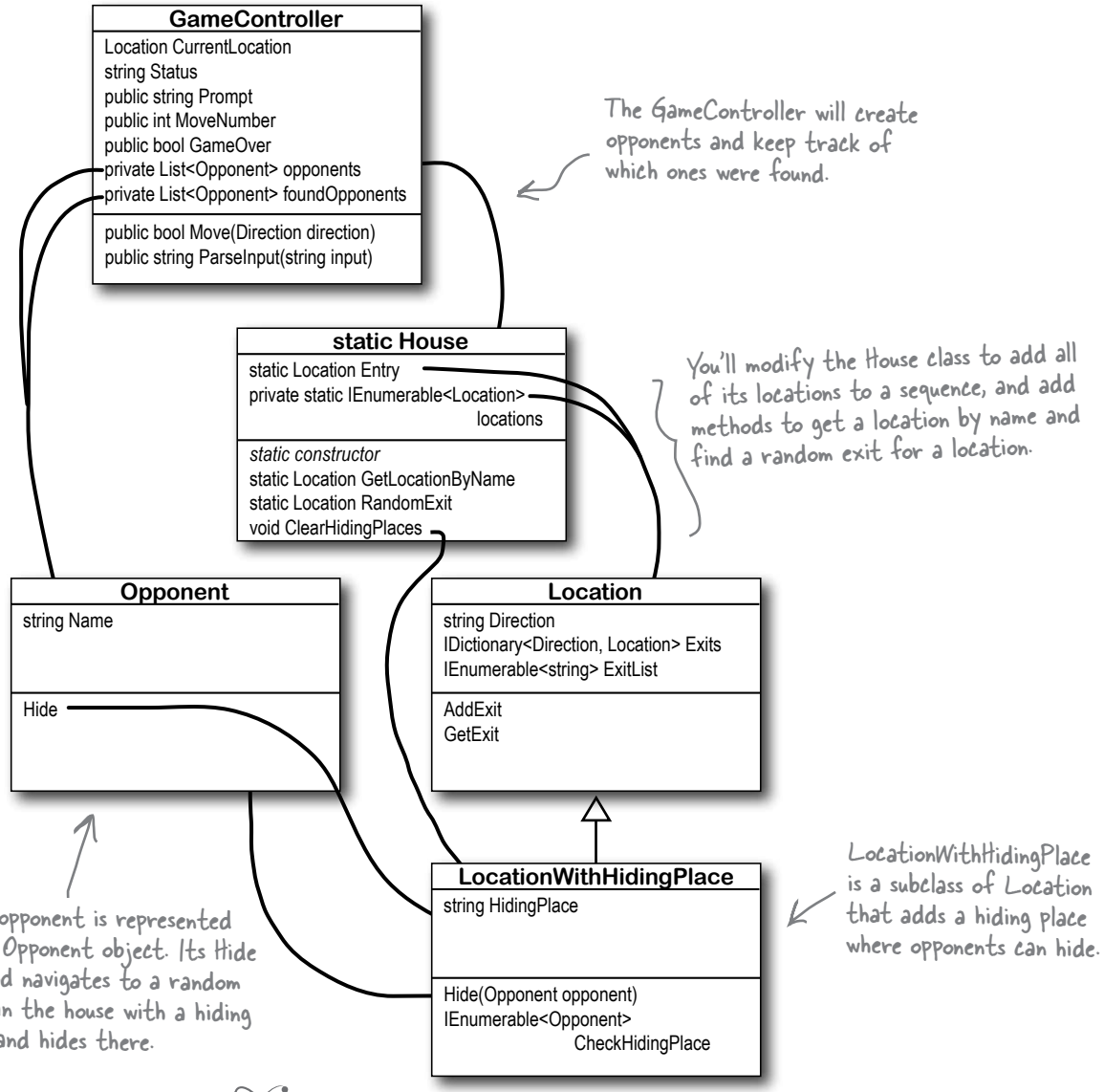
You'll build on the code you wrote in Part 1. Your game still needs to list the exits and navigate through the house—and your unit tests will help you make sure you don't accidentally reak that code.

You'll use the code you wrote in Part 1 to lay out and navigate the house as a starting point. You'll add classes to manage the opponents and hiding places, and update your GameController and other classes to turn it in to a game.

The game ends when the player finds the last opponent.

Here's the updated class diagram

You'll add two classes, Opponent and LocationWithHidingPlace, and make changes to the House and GameController classes and the Main method.



The Opponent.Hide method calls the LocationWithHidingPlace class's Hide method, but its CheckHidingPlace returns a sequence of Opponent references. Which class do you build first? Can you build them in parts?

Long Exercise

Add these unit tests to the House class. Your updated House class has two additional methods. The GetLocationByName method takes the name of a location and returns a reference to the Location with that name (or the entry, if the name isn't found). The RandomExit method takes a location and returns a random exit. The Opponent class will use that method to navigate to a random location to hide in.

After you add this test method, your code won't compile until you add a stub GetLocationByName method to Location that returns a Location reference.

```
[TestMethod]
public void TestGetLocationByName()
{
    Assert.AreEqual("Entry", House.GetLocationByName("Entry").Name);
    Assert.AreEqual("Attic", House.GetLocationByName("Attic").Name);
    Assert.AreEqual("Garage", House.GetLocationByName("Garage").Name);
    Assert.AreEqual("Master Bedroom", House.GetLocationByName("Master Bedroom").Name);
    Assert.AreEqual("Entry", House.GetLocationByName("Secret Library").Name);
}

[TestMethod]
public void TestRandomExit()
{
    var landing = House.GetLocationByName("Landing");

    House.Random = new MockRandom() { ValueToReturn = 0 };
    Assert.AreEqual("Attic", House.RandomExit(landing).Name);

    House.Random = new MockRandom() { ValueToReturn = 1 };
    Assert.AreEqual("Hallway", House.RandomExit(landing).Name);

    House.Random = new MockRandom() { ValueToReturn = 2 };
    Assert.AreEqual("Kids Room", House.RandomExit(landing).Name);

    House.Random = new MockRandom() { ValueToReturn = 3 };
    Assert.AreEqual("Master Bedroom", House.RandomExit(landing).Name);

    House.Random = new MockRandom() { ValueToReturn = 4 };
    Assert.AreEqual("Nursery", House.RandomExit(landing).Name);

    House.Random = new MockRandom() { ValueToReturn = 5 };
    Assert.AreEqual("Pantry", House.RandomExit(landing).Name);

    House.Random = new MockRandom() { ValueToReturn = 6 };
    Assert.AreEqual("Second Bathroom", House.RandomExit(landing).Name);

    var kitchen = House.GetLocationByName("Kitchen");
    House.Random = new MockRandom() { ValueToReturn = 0 };
    Assert.AreEqual("Hallway", House.RandomExit(kitchen).Name);
}
```

You'll need to add a collection of Location references to the House class and add each location to it. There's more than one way to do this! You could use a List<Location>, a Map<string, Location>, or something else entirely. You'll also need to add a static Random field called Random that the RandomExit method will use to choose a random exit from a location.

TestRandomExit uses a mock random number generator to check the exits that RandomExit returns when Random.Next returns specific values. Your Landing location will need to add its exits in a specific order to pass this test.

Add this MockRandom class to your project. The tests reuse the MockRandom class that you created in the *Go Fish!* project from Chapter 9, so you'll need to add it to your unit test project as well.

```
/// <summary>
/// Mock Random for testing that always returns a specific value
/// </summary>
public class MockRandom : System.Random
{
    public int ValueToReturn { get; set; } = 0;
    public override int Next() => ValueToReturn;
    public override int Next(int maxValue) => ValueToReturn;
    public override int Next(int minValue, int maxValue) => ValueToReturn;
}
```

You can see our solution in the HideAndSeek_part_2 project in our Chapter 10 code folder on GitHub.
https://github.com/head-first-csharp/

HOLD ON! DIDN'T YOU SAY THAT WE SHOULD START AT THE **BOTTOM** OF THE CLASS DIAGRAM? THE **House** CLASS IS IN THE MIDDLE OF THE DIAGRAM. WHY DID WE START PART 2 THERE?

Starting at the bottom of the class diagram is a general guideline to help you find classes without dependencies.

We asked you to start Part 1 with unit tests for the Location class, then move on to create unit tests for the House class, and finish up with unit tests for the GameController class. This order made sense because GameController ***depends on*** House—meaning that it has members that use members of the House class—and the House class in turn, ***depends on*** Location. So it made sense to start with Location first, because having a working Location class made it easier to build and test your House class. Those dependencies are ***reflected in the class diagram***: we generally draw those diagrams so that a class that depends on another class is above it in the diagram—and we drew lines on that diagram to show those dependencies.

In this case, we decided to start with the House class—even though it's in the middle of the class diagram—because the new code that we added doesn't depend on any of the other code we're adding in Part 2.

There's another reason we wanted to start Part 2 with the changes to the House class. When you refactor or modify your code, your unit tests can help you do it safely and make sure that you didn't accidentally break your code.

You'll see an example of unit tests helping you safely modify your code in the next section. The next class you'll add is LocationWithHidingPlace, the subclass of Location that adds a place for an opponent to hide. After you add it, you're going to modify your House class to replace some of the lines that create a new Location instance with ones that instantiate LocationWithHidingPlace. When you do, you'll be able to use your tests to make sure **you don't accidentally cause any bugs** when you modify your House class.

When you refactor or modify your code, your unit tests can help you do it safely and make sure that you didn't accidentally break your code.

Long Exercise

Use TDD to add the LocationWithHidingPlace class. Look closely at the bottom of the updated class diagram. Did you notice two lines that show how Opponent depends on LocationWithHidingPlace, which depends right back on Opponent? When you have two classes that depend on each other, you can still test one of them first—you just need to create a skeleton of the other that has stub methods. Start by **adding an Opponent skeleton** with a stub Hide method that throws an exception.

```
public class Opponent
{
    public readonly string Name;
    public Opponent(string name) => Name = name;
    public override string ToString() => Name;

    public void Hide()
    {
        throw new NotImplementedException();
    }
}
```

There are other ways to design this app so these two classes don't both depend on each other. We designed our app this way to give an example of how to to TDD in this situation.

Add the LocationWithHidingPlaceTests unit test class to test LocationWithHidingPlace. It

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace HideAndSeekTests
{
    using HideAndSeek;
    using System.Collections.Generic;
    using System.Linq;

    [TestClass]
    public class LocationWithHidingPlaceTests
    {
        [TestMethod]
        public void TestHiding()
        {
            // The constructor sets the Name and HidingPlace properties
            var hidingLocation = new LocationWithHidingPlace("Room", "under the bed");
            Assert.AreEqual("Room", hidingLocation.Name);
            Assert.AreEqual("Room", hidingLocation.ToString());
            Assert.AreEqual("under the bed", hidingLocation.HidingPlace);

            // Hide two opponents in the room, then check the hiding place
            var opponent1 = new Opponent("Opponent1");
            var opponent2 = new Opponent("Opponent2");
            hidingLocation.Hide(opponent1);
            hidingLocation.Hide(opponent2);
            CollectionAssert.AreEqual(new List<Opponent>() { opponent1, opponent2 },
                hidingLocation.CheckHidingPlace().ToList());

            // The hiding place should now be empty
            CollectionAssert.AreEqual(new List<Opponent>(),
                hidingLocation.CheckHidingPlace().ToList());
        }
    }
}
```

Your LocationWithHidingPlace class will extend the Location class. Its constructor will take two parameters, `name` and `hidingPlace`, and call the base constructor with `name`. The class will have a private Opponent collection to keep track of the opponents currently hiding in the hiding place. Once the hiding place is checked, the opponents are found, so it clears the collection.

Use type assertions to add hiding places to the house

Now that you've added your LocationWithHidingPlace and Opponent classes, you can update your House class to add hiding places—and you'll use Opponent objects to test it. You'll also use **IsInstanceOfType**, an assertion that validates that an object is a specific type.

```
Assert.IsInstanceOfType(reference, typeof(type));
```

Here's one of the assertions that you'll add to your House tests:

```
Assert.IsInstanceOfType(House.GetLocationByName("Garage"), typeof(LocationWithHidingPlace));
```

This assertion will pass if House.GetLocationByName("Garage") is an instance of LocationWithHidingPlace, but it will fail if it's still an instance of Location.

Long Exercise

Use TDD to finish the House class. Add these tests to validate that locations with rooms places were instantiated with the right type and that the ClearHidingPlaces method is working.

```
[TestMethod]
public void TestHidingPlaces()
{
    Assert.IsInstanceOfType(House.GetLocationByName("Garage"), typeof(LocationWithHidingPlace));
    Assert.IsInstanceOfType(House.GetLocationByName("Kitchen"), typeof(LocationWithHidingPlace));
    Assert.IsInstanceOfType(House.GetLocationByName("Living Room"), typeof(LocationWithHidingPlace));
    Assert.IsInstanceOfType(House.GetLocationByName("Bathroom"), typeof(LocationWithHidingPlace));
    Assert.IsInstanceOfType(House.GetLocationByName("Master Bedroom"),
                                                        typeof(LocationWithHidingPlace));
    Assert.IsInstanceOfType(House.GetLocationByName("Master Bath"), typeof(LocationWithHidingPlace));
    Assert.IsInstanceOfType(House.GetLocationByName("Second Bathroom"),
                                                        typeof(LocationWithHidingPlace));
    Assert.IsInstanceOfType(House.GetLocationByName("Kids Room"), typeof(LocationWithHidingPlace));
    Assert.IsInstanceOfType(House.GetLocationByName("Nursery"), typeof(LocationWithHidingPlace));
    Assert.IsInstanceOfType(House.GetLocationByName("Pantry"), typeof(LocationWithHidingPlace));
    Assert.IsInstanceOfType(House.GetLocationByName("Attic"), typeof(LocationWithHidingPlace));
}

[TestMethod]
public void TestClearHidingPlaces()
{
    var garage = House.GetLocationByName("Garage") as LocationWithHidingPlace;
    garage.Hide(new Opponent("Opponent1"));

    var attic = House.GetLocationByName("Garage") as LocationWithHidingPlace;
    attic.Hide(new Opponent("Opponent2"));
    attic.Hide(new Opponent("Opponent3"));
    attic.Hide(new Opponent("Opponent4"));


    House.ClearHidingPlaces();
    Assert.AreEqual(0, garage.CheckHidingPlace().Count());
    Assert.AreEqual(0, attic.CheckHidingPlace().Count());
}
```

This test creates four Opponent objects and hides them in two locations in the house, then clears the hiding places and checks them to make sure they're empty.

You'll need to add "using System.Linq" to your test class to use the Count() method.

Your existing House unit tests will help to make sure that you don't accidentally break any of the code that's already there.

Opponent.Hide requires a random location

The next thing you'll do is implement the Opponent.Hide method. You'll replace the method in the skeleton that we just gave you with one that passes a test that we give you.

```
public void Hide()
{
    throw new NotImplementedException();
}
```

When you call Opponent.Hide, it will start at the entry, then move through a random number—between 10 and 50— of locations, calling House.RandomExit at each location to find the next place to go. If the opponent ends up in a location that doesn't have a hiding place, they'll keep calling House.RandomExit and going to that location until they get to a location with a hiding place, and hide in that location.

...but MockRandom won't work for this test!

We'll need to use a mock random number generator to make our test reproducible. We *could* use this code this to test the Opponent.Hide method, but there's a problem:

```
var opponent1 = new Opponent("opponent1");
Assert.AreEqual("opponent1", opponent1.Name);
```

Create an opponent and make sure its name is set correctly.

```
House.Random = new MockRandom() { ValueToReturn = 0 };
opponent1.Hide();
```

Use MockRandom like you did with the house tests, then call the opponent's Hide method.

```
var bathroom = House.GetLocationByName("Bathroom") as LocationWithHidingPlace;
CollectionAssert.AreEqual(new[] { opponent1 }, bathroom.CheckHidingPlace().ToList());
```

Make sure the opponent hid in the expected location.

Let's figure out what happens when the opponent tries to hide:

1. The opponent will move a random number of steps—but in this case, that number is zero, so they end up in the Entry.
2. The Entry doesn't have a hiding place, so the opponent calls House.RandomExit, which calls the MockRandom.Next method, which returns 0. It returns the first exit in its exit list, Hallway.
3. Hallway doesn't have a hiding place, so the opponent calls House.RandomExit, which calls the MockRandom.Next method, which returns 0. It returns the first exit in its exit list, Entry.
4. We're back at step 2. Uh-oh! Now we're stuck in an infinite loop.

Take a few minutes and really understand what's going on there.

The same thing happens if you set MockRandom.ValueToReturn to 1 (or any other number)—the test will still end up in an infinite loop.

When you implement the Opponent.Hide method, add this statement to the end of the method:

```
System.Diagnostics.Debug.WriteLine( $"{Name} is hiding " +
$"{(currentLocation as LocationWithHidingPlace).HidingPlace} in the {currentLocation.Name}");
```

That will write a line to the application output telling you exactly where each opponent is hiding, which can be very useful when you're debugging your code so you can go straight to the opponent locations.

Long Exercise

Use TDD to finish the Opponent class. You'll need a slightly different mock random number generator. Add this MockRandomWithValueList class to your unit test project:

```
using System.Collections.Generic;

/// <summary>
/// Mock Random for testing that uses a list to return values
/// </summary>
public class MockRandomWithValueList : System.Random
{
    private Queue<int> valuesToReturn;
    public MockRandomWithValueList(IEnumerable<int> values) =>
        valuesToReturn = new Queue<int>(values);
    public int NextValue()
    {
        var nextValue = valuesToReturn.Dequeue();
        valuesToReturn.Enqueue(nextValue);
        return nextValue;
    }
    public override int Next() => NextValue();
    public override int Next(int maxValue) => Next(0, maxValue);
    public override int Next(int minValue, int maxValue)
    {
        var next = NextValue();
        return next >= minValue && next < maxValue ? next : minValue;
    }
}
```

Take a few minutes and really understand how we're using MockRandomWithValueList. Step through the code (in the debugger, or even better, on paper!) and figure out when it calls Random.Next. Then determine what value the mock random number generator returns and how it's used in the rest of the code.

Add the OpponentTests unit test class to test Opponent and implement the Opponent.Hide method. It uses MockRandomWithValueList to get the Hide method to navigate to a specific room in the house.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace HideAndSeekTests
{
   using HideAndSeek;
   using System.Linq;

   [TestClass]
   public class OpponentTests
   {
      [TestMethod]
      public void TestOpponentHiding()
      {
         var opponent1 = new Opponent("opponent1");
         Assert.AreEqual("opponent1", opponent1.Name);

         House.Random = new MockRandomWithValueList(new int[] { 0, 1 });
         opponent1.Hide();
         var bathroom = House.GetLocationByName("Bathroom") as LocationWithHidingPlace;
         CollectionAssert.AreEqual(new[] { opponent1 }, bathroom.CheckHidingPlace().ToList());

         var opponent2 = new Opponent("opponent2");
         Assert.AreEqual("opponent2", opponent2.Name);

         House.Random = new MockRandomWithValueList(new int[] { 0, 1, 2, 3, 4 });
         opponent2.Hide();
         var kitchen = House.GetLocationByName("Kitchen") as LocationWithHidingPlace;
         CollectionAssert.AreEqual(new[] { opponent2 }, kitchen.CheckHidingPlace().ToList());
      }
   }
}
```

If you modified your house layout to add rooms or change exits, your opponents may end up in a different location. In that case, you may need to modify the test to change the expected location names so they match the locations where your oppnents end up.

Add opponents to your GameController

Now that you've got your Opponent class and a LocationWithHidingPlace for opponents to hide in, you can **modify GameController to add opponents**. You'll keep track of the opponents in a private List<Opponent> called `opponents`, and the opponents that the player has found in another private List<Opponent> called `foundOpponents`. You'll also have properties to keep track of the number of moves the player has made, and

Add these fields and properties to your GameController, ***replacing*** the existing Prompt property:

Do this!

```
/// <summary>
/// The number of moves the player has made
/// </summary>
public int MoveNumber { get; private set; } = 1;

/// <summary>
/// Private list of opponents the player needs to find
/// </summary>
public readonly IEnumerable<Opponent> Opponents = new List<Opponent>()
{
    new Opponent("Joe"),
    new Opponent("Bob"),
    new Opponent("Ana"),
    new Opponent("Owen"),
    new Opponent("Jimmy"),
};

/// <summary>
/// Private list of opponents the player has found so far
/// </summary>
private readonly List<Opponent> foundOpponents = new List<Opponent>();

/// <summary>
/// Returns true if the game is over
/// </summary>
public bool GameOver => Opponents.Count() == foundOpponents.Count();

/// <summary>
/// A prompt to display to the player
/// </summary>
public string Prompt => $"{MoveNumber}: Which direction do you want to go (or type 'check'): ";
```

You'll need to add "using System.Linq" to use the Count method.

Then, modify the GameController constructor to clear the hiding places and tell each opponent to hide:

```
public GameController()
{
    House.ClearHidingPlaces();
    foreach (var opponent in Opponents)
        opponent.Hide();

    CurrentLocation = House.Entry;
}
```

Long Exercise

Modify GameControllerTests to add the TestParseCheck method. This method sets up a specific game by clearing the house, hiding opponents in specific rooms, then uses ParseInput to simulate a complete game that includes checking various locations. Make sure you add `using System.Linq;` to the top.

This test calls the GameController's ParseInput method to simulate playing the game.

```
[TestMethod]
public void TestParseCheck()
{
    Assert.IsFalse(gameController.GameOver);

    // Clear the hiding places and hide the opponents in specific rooms
    House.ClearHidingPlaces();
    var joe = gameController.Opponents.ToList()[0];
    (House.GetLocationByName("Garage") as LocationWithHidingPlace).Hide(joe);
    var bob = gameController.Opponents.ToList()[1];
    (House.GetLocationByName("Kitchen") as LocationWithHidingPlace).Hide(bob);
    var ana = gameController.Opponents.ToList()[2];
    (House.GetLocationByName("Attic") as LocationWithHidingPlace).Hide(ana);
    var owen = gameController.Opponents.ToList()[3];
    (House.GetLocationByName("Attic") as LocationWithHidingPlace).Hide(owen);
    var jimmy = gameController.Opponents.ToList()[4];
    (House.GetLocationByName("Kitchen") as LocationWithHidingPlace).Hide(jimmy);

   // Check the Entry -- there are no players hiding there
    Assert.AreEqual(1, gameController.MoveNumber);
    Assert.AreEqual("There is no hiding place in the Entry",
                    gameController.ParseInput("Check"));
    Assert.AreEqual(2, gameController.MoveNumber);

   // Move to the Garage
    gameController.ParseInput("Out");
    Assert.AreEqual(3, gameController.MoveNumber);

    // We hid Joe in the Garage, so validate ParseInput's return value and the properties
    Assert.AreEqual("You found 1 opponent hiding behind the car",
                    gameController.ParseInput("check"));
    Assert.AreEqual("You are in the Garage. You see the following exits:" +
                       Environment.NewLine + " - the Entry is In" +
                       Environment.NewLine + "Someone could hide behind the car" +
                       Environment.NewLine + "You have found 1 of 5 opponents: Joe",
                    gameController.Status);
    Assert.AreEqual("4: Which direction do you want to go (or type 'check'): ",
                    gameController.Prompt);
    Assert.AreEqual(4, gameController.MoveNumber);

   // Move to the bathroom, where nobody is hiding
    gameController.ParseInput("In");
    gameController.ParseInput("East");
    gameController.ParseInput("North");
   // Check the Bathroom to make sure nobody is hiding there
    Assert.AreEqual("Nobody was hiding behind the door", gameController.ParseInput("check"));
    Assert.AreEqual(8, gameController.MoveNumber);
```

Notice how we used an uppercase C in one assertion and a lowercase c in another? This is testing that you're doing a case-insensitive check when you parse the word "Check" in the input.

```
    // Check the Bathroom to make sure nobody is hiding there
     gameController.ParseInput("South");
     gameController.ParseInput("Northwest");
     Assert.AreEqual("You found 2 opponents hiding next to the stove",
                     gameController.ParseInput("check"));
     Assert.AreEqual("You are in the Kitchen. You see the following exits:" +
                      Environment.NewLine + " - the Hallway is to the Southeast" +
                      Environment.NewLine + "Someone could hide next to the stove" +
                      Environment.NewLine + "You have found 3 of 5 opponents: Joe, Bob, Jimmy",
                     gameController.Status);
     Assert.AreEqual("11: Which direction do you want to go (or type 'check'): ",
                     gameController.Prompt);
     Assert.AreEqual(11, gameController.MoveNumber);

     Assert.IsFalse(gameController.GameOver);

     // Head up to the Landing, then check the Pantry (nobody's hiding there)
     gameController.ParseInput("Southeast");
     gameController.ParseInput("Up");
     Assert.AreEqual(13, gameController.MoveNumber);

     gameController.ParseInput("South");
     Assert.AreEqual("Nobody was hiding inside a cabinet",
                     gameController.ParseInput("check"));
     Assert.AreEqual(15, gameController.MoveNumber);

     // Check the Attic to find the last two opponents, make sure the game is over
     gameController.ParseInput("North");
     gameController.ParseInput("Up");
     Assert.AreEqual(17, gameController.MoveNumber);

     Assert.AreEqual("You found 2 opponents hiding in a trunk",
                     gameController.ParseInput("check"));
     Assert.AreEqual("You are in the Attic. You see the following exits:" +
                        Environment.NewLine + " - the Landing is Down" +
                        Environment.NewLine + "Someone could hide in a trunk" +
                        Environment.NewLine +
                        "You have found 5 of 5 opponents: Joe, Bob, Jimmy, Ana, Owen",
                     gameController.Status);
     Assert.AreEqual("18: Which direction do you want to go (or type 'check'): ",
                     gameController.Prompt);
     Assert.AreEqual(18, gameController.MoveNumber);

     Assert.IsTrue(gameController.GameOver);
}
```

You'll also need to **modify two assertions** at the end of the TestParseInput method. Replace this line:

```
        Environment.NewLine + " - the Landing is Up", gameController.Status);
```

With this:

```
        Environment.NewLine + " - the Landing is Up" +
        Environment.NewLine + "You have not found any opponents", gameController.Status);
```

And replace this line:

```
        Environment.NewLine + " - the Hallway is to the North", gameController.Status);
```

With this

```
        Environment.NewLine + " - the Hallway is to the North" +
        Environment.NewLine + "Someone could hide behind the sofa" +
        Environment.NewLine + "You have not found any opponents", gameController.Status);
```

Update the Main method

If all of the tests pass, then your game should work! Update the Main method to create a new GameController for each new game, and use its GameOver property to check if the game is over.

```
using System;

namespace HideAndSeek
{
    class Program
    {
        static void Main(string[] args)
        {
            while (true)
            {
                var gameController = new GameController();
                while (!gameController.GameOver)
                {
                    Console.WriteLine(gameController.Status);
                    Console.Write(gameController.Prompt);
                    Console.WriteLine(gameController.ParseInput(Console.ReadLine()));
                }

                Console.WriteLine($"You won the game in {gameController.MoveNumber} moves!");
                Console.WriteLine("Press P to play again, any other key to quit.");
                if (Console.ReadKey(true).KeyChar.ToString().ToUpper() != "P") return;
            }
        }
    }
}
```

That's right. Test-driven development gives you the freedom to start the classes that have the fewest dependencies and move on from there.

Not only that, but they also help you catch bugs so you can make sure your code works. In fact, this is a great opportunity to practice testing—and maybe catch some bugs in your code. Before you go on to the last part of the project, take some time and ***add more unit tests to your test classes***. What happens when you give them weird data? Find some edge cases to check? Adding unit tests will help you understand your code better.

Part 3: Make your game load and save

In Part 3 you'll add load and save commands to load and save your game. Here's what it will look like when you save a game to a file, then restart the game and load that file:

```
You are in the Entry. You see the following exits:
 - the Hallway is to the East
 - the Garage is Out
You have not found any opponents
1: Which direction do you want to go (or type 'check'): Out
Moving Out
You are in the Garage. You see the following exits:
 - the Entry is In
Someone could hide behind the car
You have not found any opponents
2: Which direction do you want to go (or type 'check'): check
You found 1 opponent hiding behind the car
You are in the Garage. You see the following exits:
 - the Entry is In
Someone could hide behind the car
You have found 1 of 5 opponents: Bob
3: Which direction do you want to go (or type 'check'): In
Moving In
You are in the Entry. You see the following exits:
 - the Hallway is to the East
 - the Garage is Out
You have found 1 of 5 opponents: Bob
4: Which direction do you want to go (or type 'check'): East
Moving East
You are in the Hallway. You see the following exits:
 - the Bathroom is to the North
 - the Living Room is to the South
 - the Entry is to the West
 - the Kitchen is to the Northwest
 - the Landing is Up
You have found 1 of 5 opponents: Bob
5: Which direction do you want to go (or type 'check'): Northwest
Moving Northwest
You are in the Kitchen. You see the following exits:
 - the Hallway is to the Southeast
Someone could hide next to the stove
You have found 1 of 5 opponents: Bob
6: Which direction do you want to go (or type 'check'): Check
You found 2 opponents hiding next to the stove
You are in the Kitchen. You see the following exits:
 - the Hallway is to the Southeast
Someone could hide next to the stove
You have found 3 of 5 opponents: Bob, Owen, Jimmy
7: Which direction do you want to go (or type 'check'): save my_saved_game
Saved current game to my_saved_game
```

We started out playing the game as usual. We found three of the opponents, and now we're in the Kitchen.

We used the 'save' command to save the current game to a file called my_saved_game.json. (in the same folder as the binaries for the game).

We hit ^C and then restarted the game

```
You are in the Entry. You see the following exits:
 - the Hallway is to the East
 - the Garage is Out
You have not found any opponents
1: Which direction do you want to go (or type 'check'): load my_saved_game
Loaded game from my_saved_game
You are in the Kitchen. You see the following exits:
 - the Hallway is to the Southeast
Someone could hide next to the stove
You have found 3 of 5 opponents: Bob, Owen, Jimmy
7: Which direction do you want to go (or type 'check'):
```

When we load the game from my_saved_game.json it restores the found opponents, move number, and current location.

The training wheels come off!

For this last part of the project, we're not including tests or code. Your job is to figure out how to do it! There are many different ways to solve this problem. Here's what we did to create our solution:

1. We created a class called SavedGame with four properties: a string to store the name of the player's location, a Dictionary with opponent names as the key and their hiding place location names as the value, a List of found opponent names, and move number.
2. We add a unit test that hid opponents in specific locations in the house, called ParseInput to play the game and find some of them, and then called ParseInput to save the game to a temporary file. Then it creates a new GameController and calls ParseInput to load the game from the temporary file, and checks various values to make sure the game was loaded. Then we delete the temporary file.
3. We added a unit test to make sure the game does not allow filenames that included slashes or spaces (to prevent accidentally overwriting important system files).
4. We updated GameController to parse the load and save commands and added the code that loads and saves the game, using JsonSerializer to serialize and deserialize the SavedGame class to a file in the current execution folder.

You can do this!

You know enough C# to create the skeleton methods, write the test, and implement the code. If you get stuck, it's ***not* cheating** to check our solution on GitHub to see how we solved this problem. In fact, that's a great way to get these ideas to stick in your brain.

Get creative!

Did you finish the project? Don't stop there—keep going! There are lots of ways you can improve the game. Here are a few ideas:

- ★ Create a XAML or ASP.NET Blazor version of the game. You can build a user interface that reuses the classes as the Console App version.
- ★ Add rooms to your house with more hiding places.
- ★ Add more opponents.
- ★ Add a score.
- ★ **Add a timer.**
- ★ Try adding **interactive fiction** components to your game. Add an inventory and items to pick up (like keys that unlock rooms or open hiding places). You'll need to update your parser to add more commands like *take*, *unlock*, and *inventory*.

Have you ever played an interactive fiction game (sometimes called a text adventure)? If not, try playing one online! We recommend starting with the award-winning *Spider and Web* by Andrew Plotkin: https://eblong.com/zarf/zweb/tangle/