

SSE 659
Design Quality and Maintenance
Project #1

by

Jason Payne

September 17, 2013

TABLE OF CONTENTS

1. Version Control	4
1.1 Version Control Explained	4
1.1.1 Types of Version Control Systems (VCS)	4
<i>Local VCS</i>	4
<i>Centralized VCS</i>	5
<i>Distributed VCS</i>	6
1.2 Git Version Control with GitExtensions	7
1.2.1 GitExtensions	7
2. Testing Tools	10
2.1 NUnit	10
2.2 ReSharper (R#)	10
3. Test Driven Development	12
3.1 Applied Example	12
Appendix A – ‘UNO’ Card Game	19
Requirements	19
Source Code	21
UnoCardGameTests	21
<i>UnoTestSetupBase.cs</i>	22
<i>UnoTests.cs</i>	22
<i>UnoCardTests.cs</i>	26
<i>UnoPlayerTests.cs</i>	27
Uno.Library	28
<i>UnoGameFactory.cs</i>	28
<i>UnoGame.cs</i>	29
<i>UnoCard.cs</i>	31
<i>Player.cs</i>	32
Uno.UI	33
<i>Program.cs</i>	33

Topics Covered	Topic Examples
Version Control	<ul style="list-style-type: none">• Version Control Explained• Git Version Control using GitExtensions
Testing Tools	<ul style="list-style-type: none">• NUnit• ReSharper
Test Driven Development	<ul style="list-style-type: none">• Strategies• Applied Example

1. Version Control

This section will provide some high-level details on version control (a.k.a. revision control or source control) and what paradigms were used to manage the development efforts for this project.

1.1 Version Control Explained

When producing any type of software application, it is necessary – or will become necessary very quickly – to implement a version control system no matter what the scale is of the project. This need for version control stems from the inherent nature of software development in that software ***always*** changes! Software always changes because requirements constantly change. What worked yesterday for customers, may not work as effectively (or at all) today! This idea introduces the concept of versions and there needs to be rules and processes in place for any software development team to track and manage these versions. This is where a version control system can help.

Version control can be described as both a process and toolset that helps system-level managers and development teams' work together to efficiently produce a software product with potentially multiple versions. It is a system that records changes to a file or set of files over time so that developers and managers can recall specific versions later. A version control system (VCS) allows developers to:

- revert files back to a previous state
- revert the entire project back to a previous state
- review changes made over time
- see who last modified something that might be causing a problem
- who introduced an issue and when

Using a VCS also means that if things get messed up or files get lost, it is generally easy to recover.

1.1.1 Types of Version Control Systems (VCS)

Through the years, software development teams have grown the concept of version control based on the needs of their software development teams at the time. However, as technology has advanced and companies have grown, the software industry has blossomed into an integral business element of society that thrives on meeting the demands of customers who desire to consume these latest technological advancements. To meet these demands, companies need all available resources on hand to produce the best products. With this, most software development teams are no longer simply contained in one facility or physical location. They are dispersed across different companies (subcontractors) and locations. So VCSs that worked before on local systems, no longer provide an efficient mechanism for managing development teams. This evolution, however, has birthed different types of VCSs that were built to address limitations present in the other VCSs.

Today, there are three common types of VCSs that are utilized by software development teams: local, centralized, and distributed. The sections below provide brief overviews of each VCS type. Notice how the systems have evolved in order to overcome limitations present in the other VCSs.

Local VCS

One of the first VCSs utilized, this VCS has a simple database that contains all of the changes to files under version control. This VCS works by keeping patch sets (the differences between files) from one

revision to another in a special format on disk; it can then recreate what any file looked like at any point in time by adding up all the patches. This type of VCS works best for single developers as the VCS is contained solely on the developer's system.

Because it is contained on a local system, it offers no collaboration features for teams. For team environments, a local VCS would either need to be installed and maintained on every developer's system or installed and maintained on specially designated development machines. In either scenario, the VCS would have to be maintained and synchronized across multiple systems. Also, if the local system is damaged or corrupted, the VCS on that system is lost!

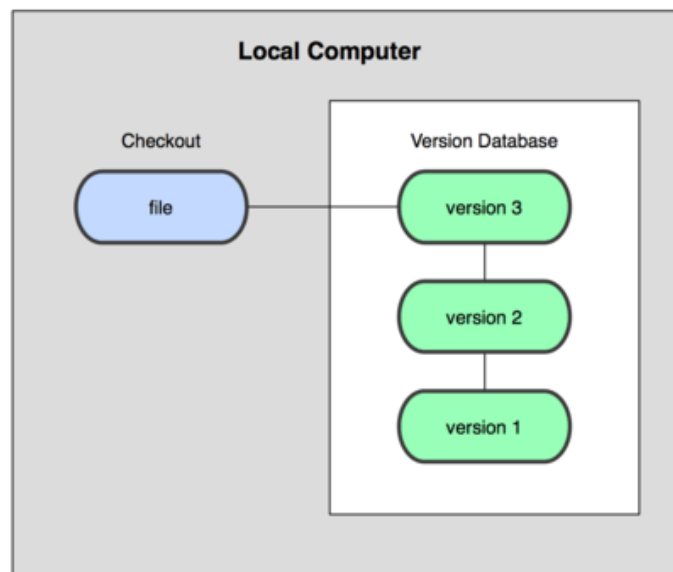


Figure 1-1: Local Version Control System

Centralized VCS

The most glaring limitation with Local VCSs is that they do not promote team collaboration on their systems. Centralized VCSs address this limitation by using a single server as the repository for the versioned files. This allows multiple computers (clients) to access a single server to “check-out” the version controlled files without worrying about synchronization issues and other limitations present with local systems.

With this type of VCS, everyone knows – to a certain degree – what everyone else is working on in the project. Administrators also have fine-grained control over who can do what; and it is far easier to administer a Centralized VCS than it is to deal with local databases on every client. However, the most serious downside to Centralized VCSs is the single point of failure that the centralized server represents. Just as with local systems, if the server is down, then no one can collaborate or save versioned changes to anything they're working on. The most significant risk is obviously the scenario where the hard disk that the central database is housed on becomes corrupted. If proper backups have not been kept, everything is lost except whatever single snapshots people happen to have on their local machines.

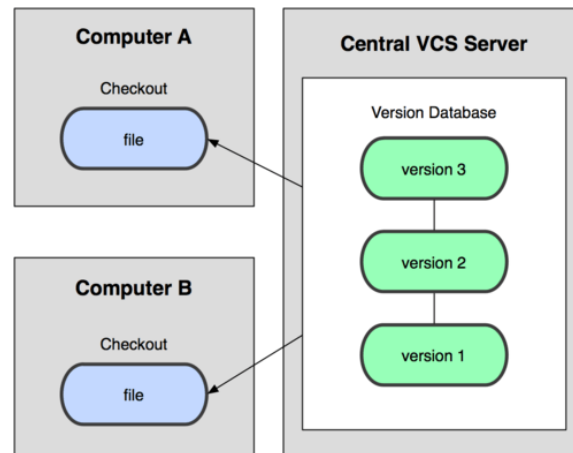


Figure 1-2: Centralized Version Control System

Distributed VCS

In a Distributed VCS, all the features available through a Centralized VCS are available, but where Distributed VCSs differ from and improve upon centralized systems is that they fully mirror the repository. This means that if any server dies and the developers have been collaborating via the server, then any of the client repositories can be copied back up to the server to restore it. Every checkout is essentially a full backup of all the data.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so developers can collaborate with different groups of people in different ways simultaneously within the same project. This allows developers to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

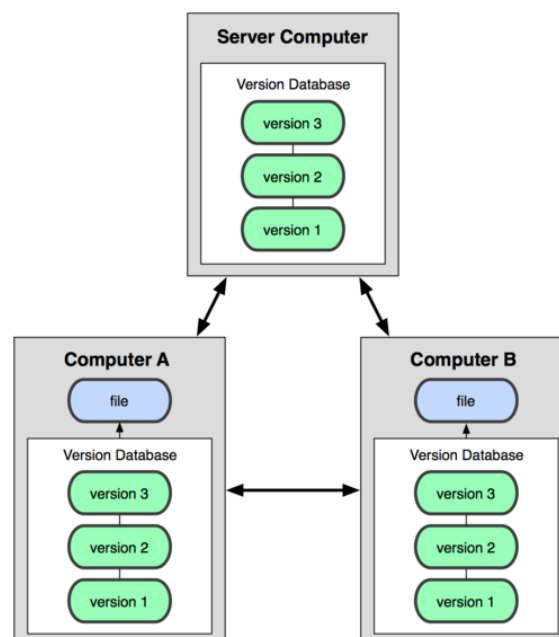


Figure 1-3: Distributed Version Control System

1.2 Git Version Control with GitExtensions

For the purposes of this project, Git will be utilized as the version control system, with GitExtensions serving as the GUI front-end. Git is an open-source distributed version control system that is easy to use. It is efficient with large projects and robust enough to provide a branching system for non-linear development.

Conceptually, most other version control systems store information as a list of file-based changes. These systems think of the information they keep as a set of files and the changes made to each file over time. Git doesn't think of or store its data this way. Instead, Git thinks of its data more like a set of snapshots of a mini file system. For every commit, Git takes a picture of what all of the files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored.

Git has three main states that the files can reside in: *committed*, *modified*, and *staged*. Committed means that the data is safely stored in the local database. Modified means that the file has changed but has not been committed to the database yet. Staged means that a modified file has been marked in its current version to go into the next commit snapshot.

The basic Git workflow is as follows:

1. Modify files in working directory.
2. Stage the files, adding snapshots of them to the staging area.
3. Execute a commit, which takes the files as they are in the staging area and stores that snapshot permanently to the local Git directory.

1.2.1 GitExtensions

Because Git version control is built around a Linux kernel, Git is used strictly through command line statements. However, GitExtensions provides a graphical user interface (Figure 1-4) that allows one to control Git without using the Git bash shell (command line). GitExtensions also provides convenient plug-ins for Visual Studio 2012, which is the IDE used for this project (Figure 1-5).

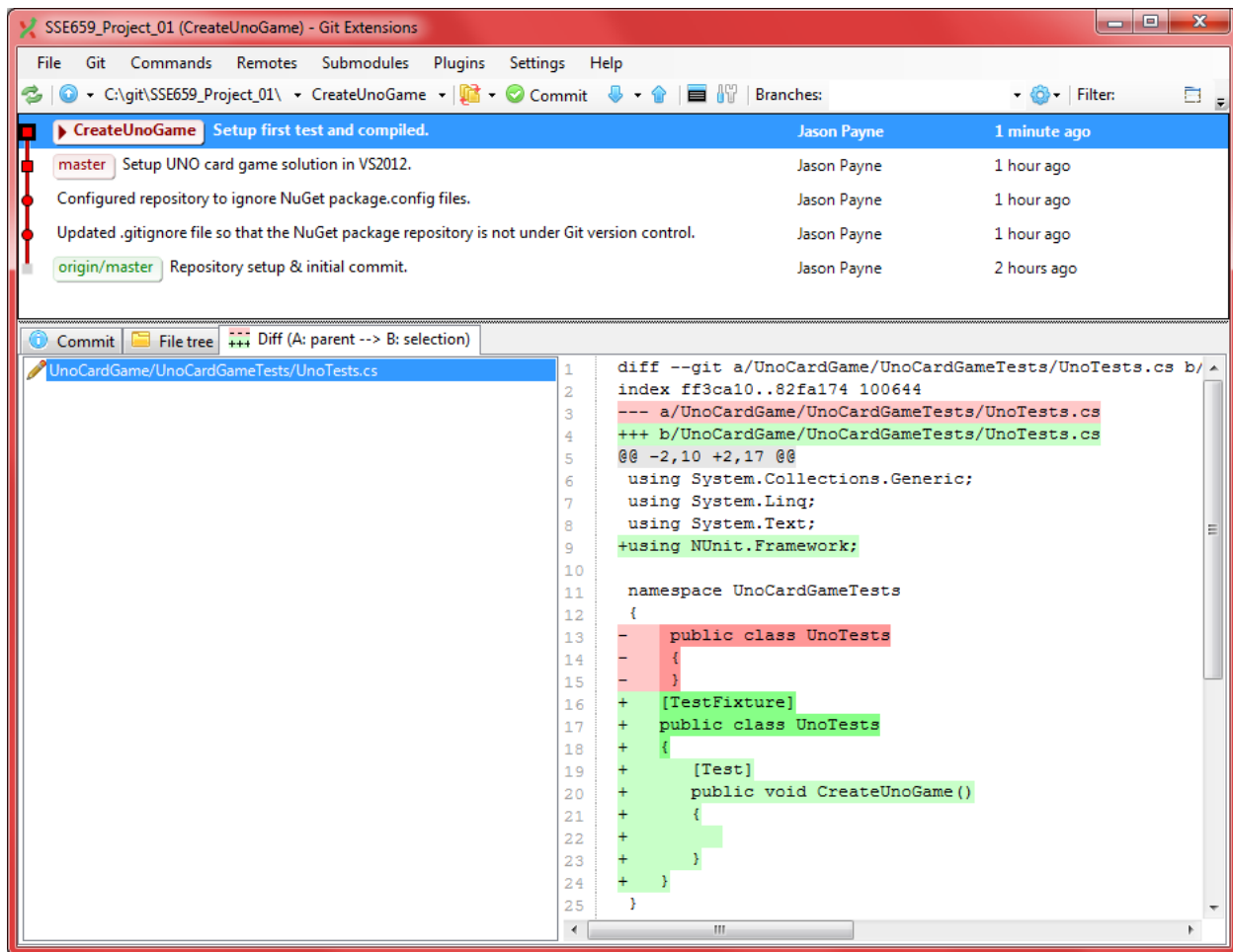


Figure 1-4: GitExtensions main window

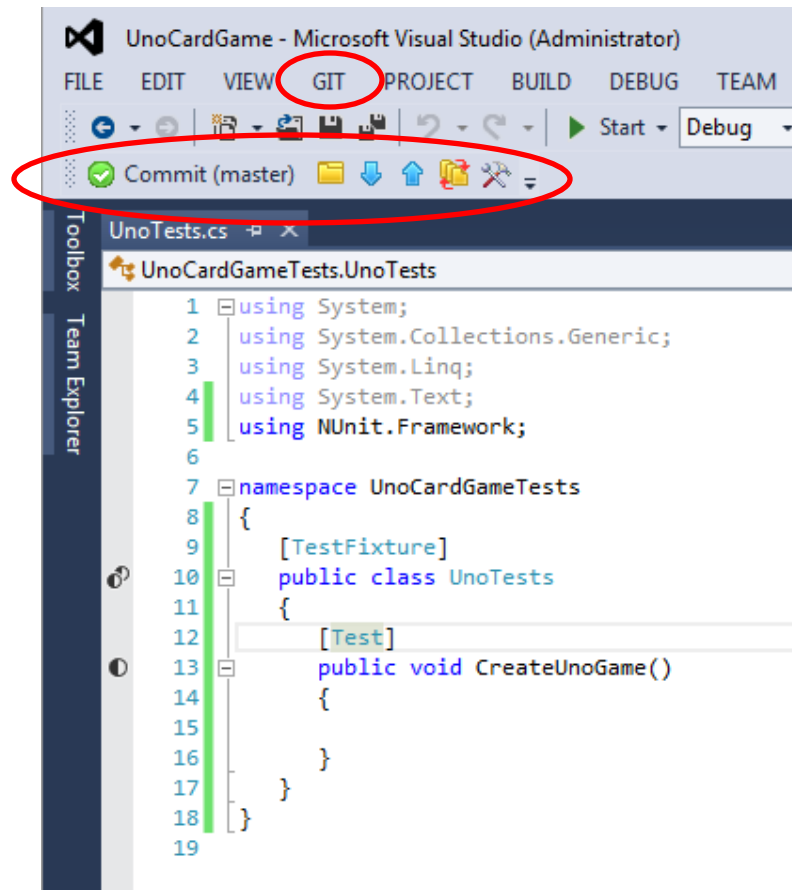


Figure 1-5: GitExtensions plug-in installed in Visual Studio 2012

2. Testing Tools

This section will detail the test tools that were utilized by this project.

2.1 NUnit

NUnit is an open-source unit-testing framework for all .NET languages. It is a widely used tool for unit testing and is currently preferred by many software development teams today. NUnit works by providing a testing framework and a test runner application. The testing framework allows testers to write test cases based on the software under test, or in the case of TDD, allows the tests to drive the implementation efforts – write the test then write the production code that passes the test. NUnit will be the unit-testing framework utilized by this project.

2.2 ReSharper (R#)

ReSharper is a widely-used productivity tool that enhances the capabilities of the Visual Studio 2012 IDE. Some of its more popular features include code inspection (think of FxCop), automated code refactorings, fast navigation tools for browsing code files, and coding assistance. ReSharper integrates with Visual Studio as a plug-in making it seamless and unobtrusive. ReSharper also has integrated unit testing support to make it easy to setup and run unit tests. ReSharper automatically detects NUnit tests simplifying the process to be up and running without the worries of setting up any external test runner GUIs.

ReSharper runs unit tests in the Unit Test Sessions window (Figure 2-1) which is designed to help run any number of unit test sessions, independently of each other, as well as simultaneously. Sessions can be composed of any combination of tests. The unit test tree shows the structure of tests belonging to a session, which can be filtered to show only *passed*, *failed* or *ignored* unit tests. Users can navigate to the code of any unit test by double-clicking the test in the sessions window.

ReSharper will be used as the test runner for this project.

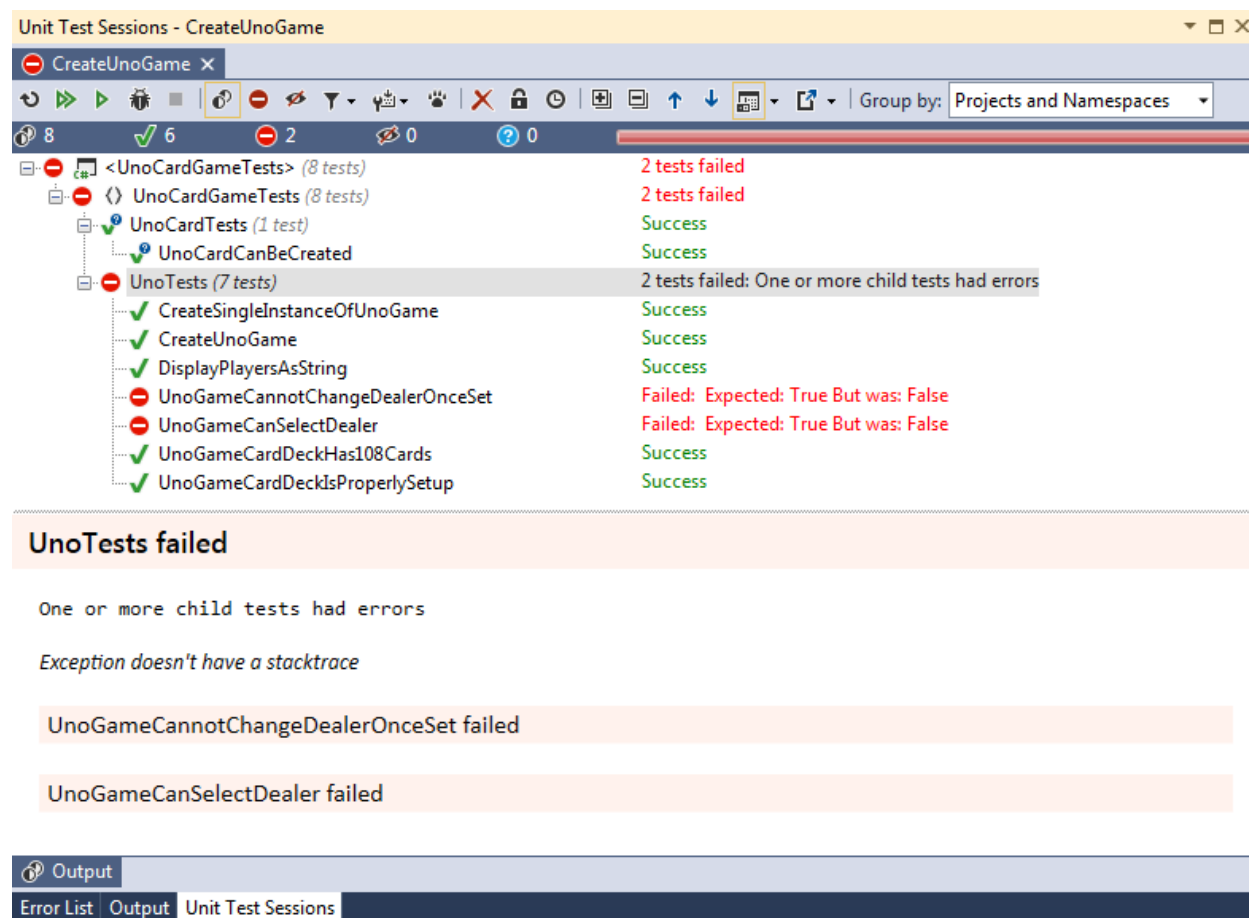


Figure 2-1: ReSharper's Unit Test Sessions window

3. Test Driven Development

Test Driven Development (TDD) is a software development concept that revolves around the idea of letting small automated unit tests drive the implementation of production software. It works off the basis of three simple steps: 1) *write a test that initially fails* (or does not even compile due to the lack of codified objects), 2) *make the test work quickly*, and 3) *refactor to eliminate dependencies and improve design*. The idea is that consistently applying these three simple steps leads to clean code that works which is the ultimate goal of TDD.

For this project, the 'UNO' card game was implemented as a console application. The application was created by using the TDD paradigm of implementation. The rest of this section provides a high-level summary of how TDD was applied during development of the application. For more detailed information including source code, see Appendix A – 'UNO' Card Game.

3.1 Applied Example

For this project, the focus was mainly on using TDD to implement the foundational components of the UNO game. Those foundational components are considered to be the instance of the game itself, the players, and the cards. Implementing the actual gameplay will not be demonstrated as part of this project, just the creation of the foundational components. However, some features of gameplay were implemented as it was considered helpful when visualizing user stories when creating the tests.

The tables below provide insight into the evolution of the tests for each of the foundational components. This also illustrates how the tests helped form the design of the foundational elements. In all instances, the tests were written first in order to truly exercise the concepts TDD. This was done by first creating a single stand-alone project (within the main Visual Studio solution - `UnoCardGame`) for writing all the tests. This project, named `UnoCardGameTests`, was initially the only project in the solution and tests were written here, before any other projects or classes were added to the solution.

To-Do List	Tests Created(*) / Impacted	New(*) / Impacted C# Objects	Comments
1. Be able to create instances of UNO games.	CreateUnoGame*	UnoGame*	
1. Be able to create instances of UNO games. 2. Create/Add UNO Players to the game.	CreateUnoGame DisplayPlayersAsString*	Player* UnoGame	<p>Tests can now test UnoGame instances for equivalency based on different amount of players being added to games. However, why should the tests be concerned with multiple instances?</p> <p>Should multiple instances of UnoGame objects be allowed? When a game is played, there is only a single instance of the game being played between players.</p>
1. Create/Add UNO Players to the game. 2. Only be able to create a single instance of an UNO game (singleton / static).	CreateSingleInstanceOfUnoGame*	UnoGame	<p>Previously passing tests are failing now because of the static nature of Singletons. The single static instance of the UnoGame is never going out of scope and is thus never reset between the tests.</p> <p>It would be ideal to have the UnoGame be a standard C# class (non-singleton), while providing a way for only a single instance to be created such that it can go in/out of static scope between tests.</p> <p>The Singleton Pattern has already been confirmed not to work, but the Factory Pattern could work while also improving the design.</p>

To-Do List	Tests Created(*) / Impacted	New(*) / Impacted C# Objects	Comments
<ol style="list-style-type: none"> 1. Only be able to create a single instance of an UNO game (singleton / static). 2. Create UNO Game Factory to handle static creation of UnoGame objects. 	CreateUnoGame CreateSingleInstanceOfUnoGame	UnoGameFactory*	UnoGameFactory works in normal conditions, but is still hard to test what amounts to static instances. Using an Inversion of Control (IoC) container could help with this.
<ol style="list-style-type: none"> 1. Only be able to create a single instance of an UNO game (singleton / static). 2. Create UNO Game Factory to handle static creation of UnoGame objects. 3. Create inversion of control container. 	CreateUnoGame CreateSingleInstanceOfUnoGame	UnoGame UnoGameFactory IoC* NOTE: IoC was later refactored and removed so it does not exist in the latest source code.	Before, the tests were not passing b/c they required different startup instances of the UnoGame which was not working b/c of the static lifetime of the instance. After, using the IoC container to store and maintain the static lifetime of the UnoGame instance, the tests started passing b/c the IoC would get destroyed and built back up after each test, essentially creating a new static instance of the UnoGame for each test, which is what was desired. This also helped solidify the design by removing the need for clients (UIs) requiring prior knowledge of the UnoGame constructor and how to construct those objects.



To-Do List	Tests Created(*) / Impacted	New(*) / Impacted C# Objects	Comments
<ol style="list-style-type: none"> 1. Only be able to create a single instance of an UNO game (singleton / static). 2. Create UNO Game Factory to handle static creation of UnoGame objects. 3. Create inversion of control container. 4. Refactor/Delete IoC. 	CreateUnoGame CreateSingleInstanceOfUnoGame	UnoGame UnoGameFactory IoC	Everything the IoC class was doing could be handled by the UnoGameFactory, so the IoC logic was refactored and moved into the UnoGameFactory class allowing for removal of the IoC class and further solidifying encapsulation and design. The next step is to start the process of selecting a dealer for the game.
<ol style="list-style-type: none"> 1. Select the dealer 2. Only be able to create a single instance of an UNO game (singleton / static). 3. Create UNO Game Factory to handle static creation of UnoGame objects. 4. Create inversion of control container. 5. Refactor/Delete IoC. 	UnoGameCanSelectDealer* UnoGameCannotChangeDealerOnceSet* UnoGameThrowsExceptionWhenSelectingDealerWithNoPlayers*	UnoGame	Before the dealer can be selected, the game needs a deck of cards and the ability to shuffle the cards.

To-Do List	Tests Created(*) / Impacted	New(*) / Impacted C# Objects	Comments
1. Select the dealer 2. Initialize/Create deck of cards 3. Create Card object	UnoCardCanBeCreated*	UnoCard*	<p>Now that there is an UnoCard, a deck of UnoCard objects can be created in the UnoGame.</p> <p>Since the UnoGameFactory handles the creation of the UnoGame instance, this class can also handle the initialization of the card deck.</p>
1. Select the dealer 2. Initialize/Create deck of cards 3. Create Card object	UnoGameCardDeckHas108Cards* UnoGameCardDeckIsProperlySetup*	UnoGame UnoGameFactory	<p>The logic used to implement the card deck initialization uses a series of loops, therefore, the deck is very closely grouped when first created.</p> <p>To maintain the integrity of the game, the deck needs to be able to be shuffled before doing anything else to eliminate the groupings of the cards.</p>

To-Do List	Tests Created(*) / Impacted	New(*) / Impacted C# Objects	Comments
1. Select the dealer 2. Shuffle the deck of cards 3. Initialize/Create deck of cards	UnoCardDeckCanBeShuffled* UnoGameCardDeckHas108CardsAfterShuffle* UnoGameCardDeckIsProperlySetupAfterShuffle*	UnoGame	<p>Now that the deck of cards can be shuffled (and tested), the focus can turn back to implementing the feature to select the dealer.</p> <p>Initially, the UnoGameFactory had a static method that would pass an initialized card deck whenever the UnoGame would request it. Unfortunately, that created a dependency on the UnoGameFactory class.</p> <p>However, using the tests as instant feedback, this was easily refactored by simply changing the UnoGame constructor and UnoGameFactory's factory method and without changing any test code (no inter-dependency nor duplication). Now, the UnoGame class is completely encapsulated!</p> <p>Get the tests to pass for selecting the dealer.</p>
1. Select the dealer 2. Shuffle the deck of cards	etc.	etc.	etc.

This example has shown the process taken while using TDD to implement the UNO application. This was shown for example purposes only. More tests were written utilizing these same procedures and the test results are shown below.

```

✓  <UnoCardGameTests> (32 tests) Success
✓  UnoCardGameTests (32 tests) Success
  ✓ UnoCardTests (19 tests) Success
    ✓ TestUnoActionCardWeights (8 tests) Success
      ✓ TestUnoActionCardWeights(Black,Wild,50) Success
      ✓ TestUnoActionCardWeights(Black,WildDraw4,50) Success
      ✓ TestUnoActionCardWeights(Blue,Reverse,20) Success
      ✓ TestUnoActionCardWeights(Green,DrawTwo,20) Success
      ✓ TestUnoActionCardWeights(Green,Skip,20) Success
      ✓ TestUnoActionCardWeights(Red,DrawTwo,20) Success
      ✓ TestUnoActionCardWeights(Red,Skip,20) Success
      ✓ TestUnoActionCardWeights(Yellow,Reverse,20) Success
    ✓ TestUnoFaceCardWeights (10 tests) Success
      ✓ TestUnoFaceCardWeights(Blue,2) Success
      ✓ TestUnoFaceCardWeights(Blue,6) Success
      ✓ TestUnoFaceCardWeights(Green,1) Success
      ✓ TestUnoFaceCardWeights(Green,5) Success
      ✓ TestUnoFaceCardWeights(Green,9) Success
      ✓ TestUnoFaceCardWeights(Red,0) Success
      ✓ TestUnoFaceCardWeights(Red,4) Success
      ✓ TestUnoFaceCardWeights(Red,8) Success
      ✓ TestUnoFaceCardWeights(Yellow,3) Success
      ✓ TestUnoFaceCardWeights(Yellow,7) Success
    ✓ UnoCardCanBeCreated Success
  ✓ UnoPlayerTests (2 tests) Success
    ✓ CanCreateUnoPlayer Success
    ✓ MultiplePlayersWithSameNameStillDifferent Success
  ✓ UnoTests (11 tests) Success
    ✓ CreateSingleInstanceOfUnoGame Success
    ✓ CreateUnoGame Success
    ✓ DisplayPlayersAsString Success
    ✓ UnoCardDeckCanBeShuffled Success
    ✓ UnoGameCannotChangeDealerOnceSet Success
    ✓ UnoGameCanSelectDealer Success
    ✓ UnoGameCardDeckHas108Cards Success
    ✓ UnoGameCardDeckHas108CardsAfterShuffle Success
    ✓ UnoGameCardDeckIsProperlySetup Success
    ✓ UnoGameCardDeckIsProperlySetupAfterShuffle Success
    ✓ UnoGameThrowsExceptionWhenSelectingDealerWithNoPlayers Success

```

Figure 3-1: TDD Test Results

Appendix A – ‘UNO’ Card Game

‘UNO’ is a popular card game that is a variation of the original card game Crazy Eights. A brief history reveals that UNO originated from an Ohio barbershop owner named Merle Robbins in 1971. As popularity of the game grew among family and friends of Mr. Robbins, he had more sets of the card game printed up and sold them from his barbershop and local businesses. As popularity continued to grow for the game, he sold the rights to the game to funeral parlor owner Robert Tezak for \$50,000, plus royalties of ten cents per game. Mr. Tezak then formed International Games to market UNO and enjoyed handsome profits from sells of the UNO card game for years. In 1992, International Games was purchased by the toy company giant, Mattel, who markets and sells the UNO game today.

The rules are simple to learn and the game is easy to play. The game can be played by 2 to 10 players and a winner is declared when a player achieves a score of 500 points. A player wins the hand by being the first one to discard all of the cards in their hand. Points are then awarded to the winner of the hand by combining the points that are assigned to the cards remaining in the opponent’s hand(s).

Requirements

The requirements for this application are simply based on the rules of UNO.

<i>ID</i>	<i>Requirement</i>
UNO_00	The game shall allow 2 to 10 players.
UNO_01	<p>The card deck shall consist of 108 cards defined by the following specifications:</p> <ul style="list-style-type: none"> • Suits differentiated by color (red, green, blue, yellow) • Ranks differentiated by number (0 – 9) • Rank 1 – 9 cards have two copies per suit (i.e. two of Red-1, two of Red-2, etc.) • Rank 0 has one copy per suit (for a total of four 0 ranked cards in the deck) • Two copies of “Draw Two”, “Skip”, and “Reverse” cards per suit • Four copies of “Wild” cards that are <u>not</u> associated with any of the suits • Four copies of “Wild Draw 4” cards that are <u>not</u> associated with any of the suits
UNO_02	<p>For scoring purposes, the following point values shall be associated with the playing cards:</p> <ul style="list-style-type: none"> • Numbered Cards – face value • Draw Two – 20 points • Reverse – 20 points • Skip – 20 points • Wild – 50 points • Wild Draw 4 – 50 points

ID	Requirement
UNO_03	<p>The game dealer shall be decided at the beginning of the game as follows:</p> <ol style="list-style-type: none"> 1) Each player picks a single card from the deck. <ol style="list-style-type: none"> a. If the player picks an action card (Draw Two, Skip, etc.), then that player must pick up another card from the deck until a numbered card is chosen. 2) The player who picks the highest number is the dealer while the remaining players are ordered from left to right in an ascending order based on their card values (i.e. the player with the lowest valued card sits to the immediate left of the dealer and, if more than two players, the player with the highest valued card sits to the immediate right of the dealer).
UNO_04	<p>The player to the left of the dealer shall start play and proceed in clockwise order.</p> <p>NOTE: See requirement for Reverse card in UNO_06 for an exception to this requirement.</p>
UNO_05	<p>Once the dealer has been chosen, gameplay shall proceed as follows:</p> <ol style="list-style-type: none"> 1) Each player is dealt 7 cards with the remaining cards placed face down to form a DRAW pile. 2) The top card of the DRAW pile is turned over to begin a DISCARD pile. 3) The first player has to match the card in the DISCARD pile either by rank, suit or action. For example, if the card is a Red-7, the player must throw down a Red-suited card, a 7-ranked card, or a Wild card. If the player does not have anything to match, they must pick a card from the DRAW pile. If the card from the DRAW pile can be played, then the player can play that card to complete their turn. Otherwise, the current player's turn is over – without discarding – and play moves to the next player. 4) Play continues by repeating step 3) until a player has discarded all but one of their cards. At this point, that player must signal this event by communicating "UNO!" to the rest of the players. NOTE: <u>Failure to do this before the next player starts their turn results in the player who failed to communicate "UNO!" having to pick up two cards from the DRAW pile.</u> <ol style="list-style-type: none"> a. If no player has discarded all of their cards by the time the DRAW pile is depleted, the DISCARD pile will be reshuffled and placed as the new DISCARD pile and play will continue as normal. 5) Once a player has discarded all of their cards, the hand is over. If the last card played in a hand is a Draw Two or Wild Draw 4 card, the next player in sequence must draw the two or four cards. Points are then awarded to the winner based on the accumulated point values associated with the cards remaining in each of the opposing player's hand(s) (see UNO_02 for details on card associated point values). 6) The first player to accumulate 500 points overall, wins the game.

ID	Requirement
UNO_06	<p>The following rules shall apply when an Action card is the first card laid down to the DISCARD pile by the dealer:</p> <ul style="list-style-type: none"> • Draw Two – the player to the left of the dealer must pick up two cards from the deck (ending their turn), and then the turn proceeds to the next player. • Reverse – with two players, the opposing player is skipped and the dealer plays first; with three or more players, the dealer plays first, and then play continues right to left (counter clockwise). • Skip – with two players, the opposing player is skipped and the dealer plays first; with three or more players, the player to the left of the dealer is skipped over and loses their turn. • Wild – the player to the left of the dealer declares the next suit (color) to be matched and then plays. • Wild Draw 4 – the card is returned to the deck, the deck is reshuffled, and the top card is chosen to start the DISCARD pile.
UNO_07	<p>When played, Action (or Word) cards shall execute the following gameplay actions:</p> <ul style="list-style-type: none"> • Draw Two – the next player must draw 2 cards and forfeit their turn. • Reverse – with two players, the opposing player is skipped and the current player gets another turn; with three or more players, the direction of play is reversed such that if play is going left to right (clockwise) it becomes right to left (counter clockwise), and vice versa. • Skip – the next player is skipped over and loses their turn. • Wild – the player declares the next suit (color) to be matched. This card can be played on any card. A Wild card can be played even if the player has another playable card in his hand. • Wild Draw 4 – the player declares the next suit (color) to be matched, and the next player has to pick 4 cards and forfeit their turn. Unlike the Wild card, this card can only be played when the player does not have a card in their hand that matches the suit (color) of the card previously played. <u>NOTE: If played, the next player in sequence (who has to take the four cards) may issue a challenge if they believe that a card in the current suit (color) could have been played. If the challenge is valid, the challenged player must draw the four cards and the challenger gets to keep their turn; if not, the challenger must draw six cards (the four original cards plus two extra).</u>

Source Code

This section lists all of the source code written for this project. It should be noted that Microsoft's Unity library (v2.1.505.2) and CommonServiceLocator library (v1.0) were utilized to implement the inversion of control container for the application.

UnoCardGameTests

This is the source code written to implement the tests that drove the main implementation of the application.

UnoTestSetupBase.cs

```

using Uno.Library;
using NUnit.Framework;

namespace UnoCardGameTests
{
    public class UnoTestSetupBase
    {
        protected UnoGame Uno { get; set; }

        [SetUp]
        public void Initialize()
        {
            UnoGameFactory.Initialize();
            Uno = UnoGameFactory.UnoGame;
        }

        /// <summary>
        /// Helper method that creates a certain number of players and adds them to the Uno game
        /// instance.
        /// </summary>
        /// <param name="num">The number of players to create.</param>
        protected void AddPlayers(int numOfPlayers)
        {
            for (int i = 0; i < numOfPlayers; i++)
            {
                Uno.AddPlayer(string.Format("Player_{0}", i + 1));
            }
        }

        /// <summary>
        /// Helper method that creates a certain number of players with the same name and adds
        /// them to the Uno game instance.
        /// </summary>
        /// <param name="num">The number of players to create.</param>
        protected void AddPlayersWithSameName(int numOfPlayers)
        {
            for (int i = 0; i < numOfPlayers; i++)
            {
                Uno.AddPlayer("PlayerWithSameName");
            }
        }
    }
}

```

UnoTests.cs

```

using System;
using System.Linq;
using NUnit.Framework;
using Uno.Library;
using UnoCardColor = Uno.Library.UnoCard.UnoCardColor;
using UnoCardAction = Uno.Library.UnoCard.UnoCardAction;

namespace UnoCardGameTests
{
    [TestFixture]
    public class UnoTests : UnoTestSetupBase
    {
        /// <summary>

```

```

/// Test that when multiple requests for an UNO game are made, that only a single
/// instance of an UNO game is being created.
/// </summary>
[Test]
public void CreateSingleInstanceOfUnoGame()
{
    var numOfPlayers = 2;
    //var unoGame = new UnoGame(numOfPlayers);
    var unoGame = UnoGameFactory.UnoGame;

    Assert.NotNull(unoGame);

    for (int i = 0; i < numOfPlayers; i++)
    {
        unoGame.AddPlayer(string.Format("Player_{0}", i + 1));
    }

    numOfPlayers = 8;
    //var unoGame2 = new UnoGame(numOfPlayers);
    var unoGame2 = UnoGameFactory.UnoGame;

    Assert.NotNull(unoGame2);

    for (int i = 0; i < numOfPlayers; i++)
    {
        unoGame2.AddPlayer(string.Format("Player_{0}", i + 3));
    }

    Assert.AreEqual(unoGame, unoGame2);
    Assert.AreEqual(unoGame.Players.Count, unoGame2.Players.Count);
}

/// <summary>
/// Test that an UnoGame can be successfully created.
/// </summary>
[Test]
public void CreateUnoGame()
{
    var numOfPlayers = 2;

    Assert.NotNull(Unos);

    for (int i = 0; i < numOfPlayers; i++)
    {
        Unos.AddPlayer(string.Format("Player_{0}", i + 1));
    }

    Assert.AreEqual(numOfPlayers, Unos.Players.Count);
}

/// <summary>
/// Test to see if the list of Players can be properly formatted as a string.
/// </summary>
[Test]
public void DisplayPlayersAsString()
{
    AddPlayers(5);

    Unos.ListPlayers();

    var expectedFormat =

```

```

        Uno.Players.Aggregate("",
                                (current, player) =>
                                    current + (player.Name + Environment.NewLine));

    Assert.That(Uno.ListPlayers(), Is.EqualTo(expectedFormat));
}

[Test]
[ExpectedException(typeof(ApplicationException))]
public void UnoGameThrowsExceptionWhenSelectingDealerWithNoPlayers()
{
    Uno.SelectDealer();
}

/// <summary>
/// Test to see that the UnoGame can select the dealer.
/// </summary>
[Test]
public void UnoGameCanSelectDealer()
{
    AddPlayers(3);

    Assert.That(Uno.Dealer, Is.Null);

    Uno.SelectDealer();

    Assert.That(Uno.Dealer, Is.Not.Null);
    Assert.That(Uno.Players.Contains(Uno.Dealer));
}

[Test]
public void UnoGameCannotChangeDealerOnceSet()
{
    AddPlayers(3);

    Assert.That(Uno.Dealer, Is.Null);

    Uno.SelectDealer();

    var initialDealer = Uno.Dealer;

    for (int i = 0; i < 10; i++)
    {
        Uno.SelectDealer();
        Assert.That(initialDealer.Equals(Uno.Dealer), Is.True);
    }
}

/// <summary>
/// Test that the UnoGame deck of cards contains 108 cards.
/// </summary>
[Test]
public void UnoGameCardDeckHas108Cards()
{
    Assert.That(Uno.CardDeck.Count, Is.EqualTo(108));
}

/// <summary>
/// Test that the UnoGame card deck has been properly setup per stated requirements from
/// the card deck description.
/// </summary>
[Test]

```



```

public void UnoGameCardDeckIsProperlySetup()
{
    ValidateUnoGameCardDeck();
}

[Test]
public void UnoGameCardDeckIsProperlySetupAfterShuffle()
{
    Uno.ShuffleDeck();
    ValidateUnoGameCardDeck();
}

[Test]
public void UnoCardDeckCanBeShuffled()
{
    var beforeDeck = Uno.CardDeck;

    Uno.ShuffleDeck();

    Assert.That(beforeDeck, Is.EquivalentTo(Uno.CardDeck));
    Assert.That(beforeDeck, Is.Not.EqualTo(Uno.CardDeck));
}

[Test]
public void UnoGameCardDeckHas108CardsAfterShuffle()
{
    Uno.ShuffleDeck();
    Assert.That(Uno.CardDeck.Count, Is.EqualTo(108));
}

#region Helper Methods

/// <summary>
/// This method provides a set of procedures for validating the card deck.
/// </summary>
private void ValidateUnoGameCardDeck()
{
    CheckUnoCardColor(UnoCardColor.Red);
    CheckUnoCardColor(UnoCardColor.Green);
    CheckUnoCardColor(UnoCardColor.Blue);
    CheckUnoCardColor(UnoCardColor.Yellow);

    var wildCardsCount = Uno.CardDeck.Count(c => c.Color == UnoCardColor.Black);
    Assert.That(wildCardsCount, Is.EqualTo(8));

    // Check that there are exactly 4 of 'Wild' and 'WildDraw4' cards.
    for (var i = 4; i <= 5; i++)
    {
        var coloredWildActionCardsByTwosCount =
            Uno.CardDeck.Count(c => (int)c.Action == i);
        Assert.That(coloredWildActionCardsByTwosCount, Is.EqualTo(4),
            "There are {0} {1}-{2} cards. Was expecting 4 cards.",
            coloredWildActionCardsByTwosCount, UnoCardColor.Black,
            ((UnoCardAction)i));
    }
}

/// <summary>
/// Helper method to help verify the UnoGame card deck based on requirements.
/// </summary>
/// <param name="unoCardColor">The UNO color "suit" to check.</param>
private void CheckUnoCardColor(UnoCardColor unoCardColor)

```

```

{
    // Check for exactly 25 colored cards.
    var colorCards = Uno.CardDeck.Where(c => c.Color == unoCardColor);
    Assert.That(colorCards.Count(), Is.EqualTo(25));

    // Check for exactly 19 numbered colored cards.
    var numberedColorCards = colorCards.Where(c => c.Rank >= 0);
    Assert.That(numberedColorCards.Count(), Is.EqualTo(19));

    // Check that there are exactly 2 of the '1' through '9' cards.
    for (var i = 1; i < 10; i++)
    {
        var coloredFaceCardsByTwos = numberedColorCards.Where(c => c.Rank == i);
        Assert.That(coloredFaceCardsByTwos.Count(), Is.EqualTo(2),
            "There are {0} {1}-{2} cards. Was expecting 2 cards.",
            coloredFaceCardsByTwos.Count(), unoCardColor, i);
    }

    // Check that there is only a single colored zero card.
    var coloredZeroCardCount = numberedColorCards.Count(c => c.Rank == 0);
    Assert.That(coloredZeroCardCount, Is.EqualTo(1));

    // Check that there are exactly 6 action cards.
    var coloredActionCards = colorCards.Where(c => c.Action != UnoCardAction.None);
    Assert.That(coloredActionCards.Count(), Is.EqualTo(6));

    // Check that there are exactly 2 of 'Skip', 'Reverse', and 'DrawTwo' cards.
    for (var i = 1; i <= 3; i++)
    {
        var coloredActionCardsByTwos = coloredActionCards.Where(c => (int) c.Action == i);
        Assert.That(coloredActionCardsByTwos.Count(), Is.EqualTo(2),
            "There are {0} {1}-{2} cards. Was expecting 2 cards.",
            coloredActionCardsByTwos.Count(), unoCardColor, ((UnoCardAction) i));
    }
}

#endregion // Helper Methods
}
}

```

UnoCardTests.cs

```

using NUnit.Framework;
using Uno.Library;
using UnoCardColor = Uno.Library.UnoCard.UnoCardColor;
using UnoCardAction = Uno.Library.UnoCard.UnoCardAction;

namespace UnoCardGameTests
{
    [TestFixture]
    public class UnoCardTests : UnoTestSetupBase
    {
        [Test]
        public void UnoCardCanBeCreated()
        {
            var unoCard = new UnoCard(UnoCardColor.Red, 0);
            Assert.That(unoCard, Is.Not.Null);
            Assert.That(unoCard.Action, Is.EqualTo(UnoCardAction.None));
            Assert.That(unoCard.Rank, Is.EqualTo(0));
            Assert.That(unoCard.Color, Is.EqualTo(UnoCardColor.Red));
            Assert.That(unoCard.Weight, Is.EqualTo(0));
        }
    }
}

```

```

        var actionUnoCard = new UnoCard(UnoCardColor.Black, UnoCardAction.WildDraw4);
        Assert.That(actionUnoCard, Is.Not.Null);
        Assert.That(actionUnoCard.Action, Is.EqualTo(UnoCardAction.WildDraw4));
        Assert.That(actionUnoCard.Rank, Is.EqualTo(-1));
        Assert.That(actionUnoCard.Color, Is.EqualTo(UnoCardColor.Black));
        Assert.That(actionUnoCard.Weight, Is.EqualTo(50));
    }

    [TestCase(UnoCardColor.Red, 0)]
    [TestCase(UnoCardColor.Green, 1)]
    [TestCase(UnoCardColor.Blue, 2)]
    [TestCase(UnoCardColor.Yellow, 3)]
    [TestCase(UnoCardColor.Red, 4)]
    [TestCase(UnoCardColor.Green, 5)]
    [TestCase(UnoCardColor.Blue, 6)]
    [TestCase(UnoCardColor.Yellow, 7)]
    [TestCase(UnoCardColor.Red, 8)]
    [TestCase(UnoCardColor.Green, 9)]
    public void TestUnoFaceCardWeights(UnoCardColor color, int rank)
    {
        var card = new UnoCard(color, rank);
        Assert.That(card.Weight, Is.EqualTo(rank));
    }

    [TestCase(UnoCardColor.Red, UnoCardAction.DrawTwo, 20)]
    [TestCase(UnoCardColor.Green, UnoCardAction.DrawTwo, 20)]
    [TestCase(UnoCardColor.Blue, UnoCardAction.Reverse, 20)]
    [TestCase(UnoCardColor.Yellow, UnoCardAction.Reverse, 20)]
    [TestCase(UnoCardColor.Red, UnoCardAction.Skip, 20)]
    [TestCase(UnoCardColor.Green, UnoCardAction.Skip, 20)]
    [TestCase(UnoCardColor.Black, UnoCardAction.Wild, 50)]
    [TestCase(UnoCardColor.Black, UnoCardAction.Wild, 50)]
    [TestCase(UnoCardColor.Black, UnoCardAction.WildDraw4, 50)]
    [TestCase(UnoCardColor.Black, UnoCardAction.WildDraw4, 50)]
    public void TestUnoActionCardWeights(UnoCardColor color, UnoCardAction action,
                                          int expected)
    {
        var card = new UnoCard(color, action);
        Assert.That(card.Weight, Is.EqualTo(expected));
    }
}

```

UnoPlayerTests.cs

```

using NUnit.Framework;
using Uno.Library;

namespace UnoCardGameTests
{
    [TestFixture]
    class UnoPlayerTests : UnoTestSetupBase
    {
        [Test]
        public void CanCreateUnoPlayer()
        {
            var expectedName = "Test";
            var player = new Player(expectedName);
            Assert.That(player != null);
            Assert.That(player.Name == expectedName);
        }
    }
}

```

```

[Test]
public void MultiplePlayersWithSameNameStillDifferent()
{
    var name = "Player_1";
    var player1 = new Player(name);
    var player2 = new Player(name);

    Assert.That(!Equals(player1, player2));
}
}
}

```

Uno.Library

This is the core piece of the UNO application. This serves as a library so that other GUI clients could potentially use this to drive an UNO application with an advanced GUI.

UnoGameFactory.cs

```

using System.Collections.Generic;
using Microsoft.Practices.Unity;
using UnoCardColor = Uno.Library.UnoCard.UnoCardColor;
using UnoCardAction = Uno.Library.UnoCard.UnoCardAction;

namespace Uno.Library
{
    public static class UnoGameFactory
    {
        private const int UnoCardColors = 4;
        private const int UnoCardRanks = 9;
        private const int UnoCardActions = 3;
        private const int UnoWildCards = 4;
        private const int UnoWildCardActions = 2;

        private static IUnityContainer m_container = new UnityContainer();

        public static void Initialize()
        {
            m_container.RegisterInstance(new UnoGame(CreateUnoCardDeck()),
                                         new ContainerControlledLifetimeManager());
        }

        public static UnoGame UnoGame
        {
            get { return m_container.Resolve<UnoGame>(); }
        }

        private static List<UnoCard> CreateUnoCardDeck()
        {
            var newDeck = new List<UnoCard>();

            // Create the color cards.
            for (int color = 0; color < UnoCardColors; color++)
            {
                // Create the single zero card.
                newDeck.Add(new UnoCard((UnoCardColor) color, 0));

                // Create two instances of the 1-9 cards.
                for (int rank = 1; rank <= UnoCardRanks; rank++)
                {
                    newDeck.Add(new UnoCard((UnoCardColor) color, rank));
                    newDeck.Add(new UnoCard((UnoCardColor) color, rank));
                }
            }
        }
    }
}

```

```

    }

    // Create two instances of the action cards.
    for (int action = 1; action <= UnoCardActions; action++)
    {
        newDeck.Add(new UnoCard((UnoCardColor) color, (UnoCardAction) action));
        newDeck.Add(new UnoCard((UnoCardColor) color, (UnoCardAction) action));
    }
}

// Create the Wild cards.
for (int wildCard = 0; wildCard < UnoWildCards; wildCard++)
{
    for (int wildCardAction = 4;
        wildCardAction < UnoWildCardActions + 3;
        wildCardAction++)
    {
        newDeck.Add(new UnoCard(UnoCardColor.Black, (UnoCardAction) wildCardAction));
        newDeck.Add(new UnoCard(UnoCardColor.Black,
            (UnoCardAction) wildCardAction + 1));
    }
}

return newDeck;
}
}
}

```

UnoGame.cs

```

using System;
using System.Collections.Generic;
using System.Linq;

namespace Uno.Library
{
    public sealed class UnoGame
    {
        public const int MinNumOfPlayers = 2;
        public const int MaxNumOfPlayers = 10;

        private List<Player> m_players;
        private List<UnoCard> m_deck;

        private UnoGame(){}

        internal UnoGame(List<UnoCard> deck)
        {
            m_players = new List<Player>();
            m_deck = deck;
        }

        public IList<UnoCard> CardDeck
        {
            get { return m_deck.AsReadOnly(); }
        }

        public IList<Player> Players
        {
            get { return m_players.AsReadOnly(); }
        }
    }
}

```

```

public Player Dealer { get; private set; }

public void AddPlayer(string name)
{
    m_players.Add(new Player(name));
}

public string ListPlayers()
{
    return Players.Aggregate("",
                              (current, player) =>
                              current + (player.Name + Environment.NewLine));
}

public void SelectDealer()
{
    if (m_players == null || !m_players.Any())
        throw new ApplicationException(
            "There are no players in the game yet! Please add some players before "+
            "attempting to select the dealer.");

    // If the dealer has already been set, then do not allow the dealer to be reset.
    if (Dealer != null) return;

    ShuffleDeck();

    var faceCards = m_deck.Where(ac => ac.Action == UnoCard.UnoCardAction.None);

    IDictionary<Player, UnoCard> players = new Dictionary<Player, UnoCard>();
    for (int i = 0; i < m_players.Count; i++)
    {
        var p = m_players[i];
        var c = faceCards.ElementAt(i);
        players.Add(p, c);
    }

    var playerRanks =
        players.OrderBy(kvp => kvp.Value.Rank)
                .ThenBy(kvp => kvp.Value.Color)
                .ThenBy(kvp => kvp.Key.UniqueId);

    m_players = playerRanks.Select(kvp => kvp.Key).ToList();

    Dealer = m_players.Last();
}

public void ShuffleDeck()
{
    var tempDeck = new List<UnoCard>(m_deck.Count);

    var random = new Random();

    for (int i = 0; i < m_deck.Count; i++)
    {
        do
        {
            var card = m_deck[random.Next(m_deck.Count)];
            if (tempDeck.Contains(card)) continue;
            tempDeck.Add(card);
            break;
        } while (true);
    }
}

```

```

        m_deck = tempDeck;
    }
}

```

UnoCard.cs

```

using System.ComponentModel.DataAnnotations;

namespace Uno.Library
{
    public class UnoCard
    {
        public UnoCardColor Color { get; internal set; }

        [Range(-1, 9, ErrorMessage = "Value for {0} must be between {1} and {2}.")]
        public int Rank { get; internal set; }

        public UnoCardAction Action { get; internal set; }

        public int Weight { get; private set; }

        public enum UnoCardColor
        {
            Red,
            Green,
            Blue,
            Yellow,
            Black
        }

        public enum UnoCardAction
        {
            None,
            Skip,
            Reverse,
            DrawTwo,
            Wild,
            WildDraw4
        }

        private UnoCard(){}

        private UnoCard(UnoCardColor color)
        {
            Color = color;
        }

        public UnoCard(UnoCardColor color, int rank)
            : this(color)
        {
            Rank = rank;

            if (rank > -1) Weight = rank;
        }

        public UnoCard(UnoCardColor color, UnoCardAction action)
            : this(color, -1)
        {
            Action = action;
        }
    }
}

```

```

        switch (action)
        {
            case UnoCardAction.DrawTwo:
            case UnoCardAction.Reverse:
            case UnoCardAction.Skip:
                Weight = 20;
                break;
            case UnoCardAction.Wild:
            case UnoCardAction.WildDraw4:
                Weight = 50;
                break;
        }
    }

    public override string ToString()
    {
        return (Action == UnoCardAction.None)
            ? string.Format("{0}-{1}", Color, Rank)
            : string.Format("{0}-{1}", Color, Action);
    }
}

```

Player.cs

```

namespace Uno.Library
{
    public class Player
    {
        private static int m_id;

        private int m_uniqueId = -1;

        public int UniqueId
        {
            get { return m_uniqueId; }
            private set { m_uniqueId = value; }
        }

        public string Name { get; private set; }

        public Player(string name)
        {
            UniqueId = m_id++;
            Name = name;
        }

        public override bool Equals(object obj)
        {
            if ((obj == null) || !(obj is Player)) return false;
            return Equals(obj as Player);
        }

        public override int GetHashCode()
        {
            return (Name != null ? Name.GetHashCode() : 0);
        }

        protected bool Equals(Player other)
        {
            return (string.Equals(Name, other.Name)) && (UniqueId == other.UniqueId);
        }
    }
}

```



```

        public override string ToString()
        {
            return string.Format("{0}, Id: {1}", Name, UniqueId);
        }
    }
}

```

Uno.UI

This is the source code written for the console user interface.

Program.cs

```

using System;
using Uno.Library;

namespace Uno.UI
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            UnoGameFactory.Initialize();
            var uno = UnoGameFactory.UnoGame;
            var min = UnoGame.MinNumOfPlayers;
            var max = UnoGame.MaxNumOfPlayers;

            Console.WriteLine(
                "Welcome to the card game UNO!\n\nHow many players will be playing ({0}-{1})?",
                min, max);

            int numOfPlayers;
            while ((!int.TryParse(Console.ReadLine(), out numOfPlayers)) ||
                (numOfPlayers < min || numOfPlayers > max))
            {
                Console.WriteLine("Please enter a valid number of players ({0}-{1}):", min, max);
            }

            for (int i = 0; i < numOfPlayers; i++)
            {
                Console.WriteLine("Enter player {0}'s name: ", i + 1);
                uno.AddPlayer(Console.ReadLine());
            }

            Console.WriteLine();

            Console.WriteLine("Let's get started! Selecting a dealer now...");
            uno.SelectDealer();
            Console.WriteLine("{0} is the dealer!", uno.Dealer.Name);

            // TODO: Implement UNO gameplay.

            Console.ReadKey();
        }
    }
}

```