SSE 657

Object-Oriented Project Methods

Project #1

by

Jason Payne

September 29, 2014

TABLE OF CONTENTS

Topics Covered	Topic Examples
Object-Oriented Analysis & Design	• Establishing Requirements • Use Cases (Primary / Optional / Alternate) • UML (Class Diagrams) • Encapsulation & Delegation • Inheritance with Interfaces

1. Object Oriented Analysis & Design (OOAD)

Object Oriented Analysis & Design (OOAD) is a fundamental concept in today's world of software development. It is the by-product of early competing methodologies that were being used for object-oriented software development and modeling. As these methodologies evolved, it started to become increasingly understood that the core concepts at the heart of these methodologies were similar if not the same.

OOAD puts particular emphasis on creating systems that have well-defined requirements with designs that are flexible, developer-friendly, maintainable, and reusable. It promotes modeling techniques such as: **use cases** for bridging the communication gap between customer and developer; **class diagrams** for recognizing potential design flaws; **encapsulation**, **composition**, and **delegation** for maintainability and extensibility; and the **Open-Closed Principle** and the **Single Responsibility Principle** for reusability. Applying OOAD principles to a system yields results that are not only pleasing to the customer, but to the developer as well!

In the following sections, two applications will be presented as illustrations of applied OOAD principles and methodologies. Even though they are presented as two applications, they are essentially the same product that has evolved with the requirements, much like any software application in use today. Initially, the application starts as a stand-alone application that simply does what the customer wants, but by the end has transformed into a more robust application that can easily expand to meet the customer's demands while also remaining developer-friendly.

2. 'Tic-Tac-Toe' Application

In this section, an application is created that allows users to play Tic-Tac-Toe. The application starts out without much consideration for design and simply does what the requirements state. However, by using and applying OOAD concepts, ominous flaws in the requirements and design will be discovered and corrected.

2.1 Requirements & Use Case

Requirements specify the how's, what's and when's of a system. They represent the expectations and desires of the customer. Obviously, it is very important to meet these expectations with an application that does what the customer specifies in the requirements. However, sometimes what the customer specifies does not cover all possible scenarios which can lead to unmet expectations. As will be illustrated in the following sections, applying OOAD concepts can help isolate missing requirements and/or uncovered scenarios.

Listed below are the initial requirements for the Tic-Tac-Toe application. A use case is also presented as a likely scenario path when using the application.

Tic-Tac-Toe, version 1.0 Requirements List	Use Case
1. Two players will be allowed to play the game.	1. The application will prompt the user for the players' names.
2. The players will be able to play a game of Tic-Tac-Toe.	2. The players will play a game of Tic-Tac-Toe.
3. Once a game has been won, the application will ask the players if they wish to play again.	3. Once the game has been won or tied, the application will ask the players if they wish to play again.
4. When the players no longer wish to continue playing, the application will close.	4. When the players no longer wish to continue playing, the application will close.

2.1.1 Implementation

Listed below is the source code for the Tic-Tac-Toe application. Because no design effort was performed, the implementation was put into a single class. While this implementation meets the specified requirements, the following sections will show how this is not the best way to implement the solution.

```
class Program
{
    static void Main()
    {
        Log.Initialize();

        var numOfPlayers = 2;
        var players = new List<string>(numOfPlayers);
        for (var i = 0; i < numOfPlayers; i++)
        {
            Console.WriteLine("Enter player {0}'s name: ", i + 1);
            players.Add(Console.ReadLine());
        }

        while (true)
        {
```

```
            var grid = new bool?[3, 3];
            var isGameWon = false;
            var isPlayerO = false;
            do
            {
                var player = Convert.ToInt16(isPlayerO);

                Console.WriteLine();
                Console.WriteLine("{0}'s turn...", players[player]);
                var input = Console.ReadLine().Split(' ');
                Console.WriteLine();
                var row = Convert.ToInt16(input.First());
                var col = Convert.ToInt16(input.Last());
                grid[row - 1, col - 1] = isPlayerO;

                // Display grid
                for (var r = 0; r < 3; r++)
                {
                    for (var c = 0; c < 3; c++)
                    {
                        var cell = grid[r, c];
                        var icon = cell.HasValue ? ((cell.Value ? "O" : "X")) : "_";
                        Console.Write("{0} ", icon);
                    }
                    Console.WriteLine();
                }
                Console.WriteLine();

                // Check for 3-in-a-row on rows.
                for (var r = 0; (r < 3) && !isGameWon; r++)
                {
                    var numOfX = 0;
                    var numOfO = 0;
                    for (var c = 0; c < 3; c++)
                    {
                        if (!grid[r, c].HasValue) break;

                        if (grid[r, c].Value) numOfO++;
                        else numOfX++;
                        if (numOfX == 3 || numOfO == 3)
                        {
                            isGameWon = true;
                            break;
                        }

                        if (numOfX > 0 && numOfO > 0) break;
                    }
                }

                // Check for 3-in-a-row on columns.
                for (var c = 0; (c < 3) && !isGameWon; c++)
                {
                    var numOfX = 0;
                    var numOfO = 0;
                    for (var r = 0; r < 3; r++)
                    {
                        if (!grid[r, c].HasValue) break;
```

```
                        if (grid[r, c].Value) numOfO++;
                        else numOfX++;
                        if (numOfX == 3 || numOfO == 3)
                        {
                            isGameWon = true;
                            break;
                        }

                        if (numOfX > 0 && numOfO > 0) break;
                    }
                }

                isPlayerO = !isPlayerO;

                var isDraw = true;
                foreach (var cell in grid)
                {
                    if (!cell.HasValue)
                    {
                        isDraw = false;
                        break;
                    }
                }

                if (isDraw) break;
            } while (!isGameWon);

            Console.WriteLine();
            Console.WriteLine("Continue? (y/n)");
            if (Console.ReadLine().ToLower() != "y")
                break;
        }

        Log.Close();
    }
}
```

Output

```
Enter player 1's name:
One
Enter player 2's name:
Two

One's turn...
3 3


_ _ _
_ _ _
_ _ X


Two's turn...
1 1

O _ _
_ _ _
_ _ X
```

```
One's turn...
2 2

O _ _
_ X _
_ _ X


Two's turn...
1 3

O _ O
_ X _
_ _ X


One's turn...
1 2

O X O
_ X _
```

```
_ _ X


Two's turn...
3 2

O X O
_ X _
_ O X


One's turn...
3 1

O X O
_ X _
X O X


Two's turn...
2 1
```

```
O X O
O X _
X O X


One's turn...
2 3

O X O
O X X
X O X


Continue? (y/n)
y

One's turn...
1 1

X _ _
_ _ _
_ _ _


Two's turn...
2 2

X _ _
_ O _
_ _ _


One's turn...
1 2

X X _
_ O _
_ _ _


Two's turn...
3 1

X X _
_ O _
O _ _


One's turn...
1 3

X X X
_ O _
O _ _


Continue? (y/n)
y

One's turn...
2 2

_ _ _
```

```
_ X _
_ _ _


Two's turn...
3 3

_ _ _
_ X _
_ _ O


One's turn...
1 3

_ _ X
_ X _
_ _ O


Two's turn...
3 2

_ _ X
_ X _
_ O O


One's turn...
3 1

_ _ X
_ X _
X O O


Two's turn...
1 1

O _ X
_ X _
X O O


One's turn...
1 2

O X X
_ X _
X O O


Two's turn...
2 3

O X X
_ X O
X O O


One's turn...
2 1

O X X
```

```
X X O
X O O

Continue? (y/n)
n
```

2.1.2 Requirements Analysis

As seen above (in the highlighted portion of the output), a problem exists when a player has a diagonal three-in-a-row winning pattern. This is a significant error as this goes against the fundamental rules of Tic-Tac-Toe. However, as is seen in the initial requirements and use case, nothing is said about what constitutes a winning pattern. This is mostly because it is generally understood what a winning pattern is in Tic-Tac-Toe, but consider a more realistic scenario where a new developer joins a team and is tasked with implementing a basic feature that is obvious to everyone familiar with the system EXCEPT the new developer. In such a scenario, the new developer is not necessarily at fault because there was no requirement or use case scenario that stated the details.

Here, the same thing has happened. There is no stated requirement or use case scenario specifying the details of what constitutes a winning pattern. By analyzing the requirements and use cases, this and other issues can be realized before getting too far in the development lifecycle where it may be too costly or inefficient to implement a correction.

Listed below are some of the problems and issues found by analyzing the requirements and use case:

- The application should allow for a single player against a simulated player with varying degrees of difficulty (easy, medium, hard) – *beyond the scope of this project*.
- The requirements do not have to necessarily state how one plays Tic-Tac-Toe, but the use case should specify more details in that regard.
- Are the players' names being validated (length, alphanumeric v. alpha-only, etc.)?
- Is player input for the row/col being validated?
- In the version 1.0 requirements, requirement 3 specifies that the application will ask the user to continue after a game has been won. The requirement should be updated to state that this happens after a game has been won **or tied**.
- Even though this is currently implemented, there is no specification in how the results (i.e. grid) should be displayed. Should it be a single grid that is updated in real-time or a reoccurring grid that is displayed as the players make their choices (like it is now)? This is actually quiet important due to the potential implications on the currently implemented solution. Because such a thing is platform-dependent (mobile device, website, desktop app, etc.), the current implementation is developed strictly as a console application which limits its use to being strictly a desktop application eliminating potential customers that require a GUI front-end.

2.1.3 Updated Requirements & Use Case

Using the results of the analysis, the requirements list (version 2.0) and use case have been updated to reflect the corrections and changes highlighted by the analysis.

Tic-Tac-Toe, version 2.0 Requirements List
1. Two players will be allowed to play the game.
2. The players will be able to play a game of Tic-Tac-Toe.
3. Once a game has been won **or tied**, the application will ask the players if they wish to play again.
4. When the players no longer wish to continue playing, the application will close.

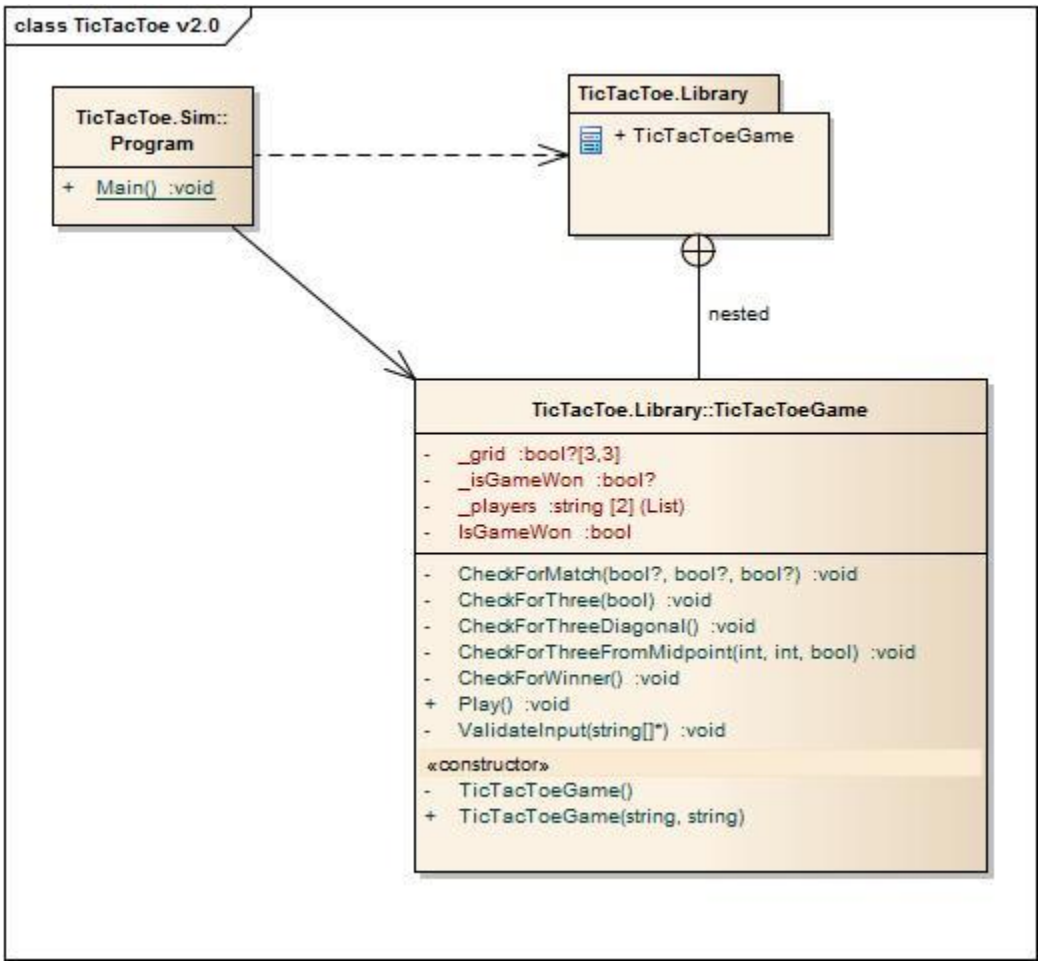
Original Use Case	Updated Use Case
1. The application will prompt the user for the players' names. 2. The players will play a game of Tic-Tac-Toe. 3. Once the game has been won or tied, the application will ask the players if they wish to play again. 4. When the players no longer wish to continue playing, the application will close.	1. The application will prompt the user for the players' names. a. If the name is greater than 25 characters long, the application will automatically truncate the name to the first 25 characters of the name. 2. The players will play a game of Tic-Tac-Toe. a. The players will enter where in the grid they wish to place their symbol by entering two numbers separated by a space. The first number will be for the row and the second number will be for the column. i. If the values entered are invalid formats (i.e. letters) OR if either of the values entered are less than 1 and/or greater than 3 OR if the values are of a cell that already contains a symbol, then the player will be informed of the invalid input and the system will prompt the user to enter in valid values. b. A player wins the game by having their symbol placed 3-in-a-row in a vertical, horizontal, or diagonal direction. i. If there is no winner once the grid is full, the game will be declared a tie. 3. Once the game has been won or tied, the application will ask the players if they wish to play again. 4. When the players no longer wish to continue playing, the application will close.

2.2 Add Design Flexibility

For the initial solution, there is essentially no design for the system. Everything is placed in the same class at the application's run-time entry point (`Program.Main()`). This makes the code very rigid to change because any change will have to take place in the single class. This is in direct contradiction to the Open-Close Principle and the Single Responsibility Principle. Ideally, the application's `Program` class would only need to worry about initializing the game and executing the game itself without concerning itself with the details of the game execution.

With updated requirements and applying some basic object-oriented principles, a more robust solution can be realized. The subsequent sections provide details on a potential solution that provides that realization.

2.2.1 Class Diagram



The class diagram provides a "bird's eye view" of the software system. It does not show details of implementation, but gives a general overview of a proposed solution from a class and package perspective.

2.2.2 Updated Implementation

As seen in the class diagram, the details of the Tic-Tac-Toe gameplay should be encapsulated within another class (`TicTacToeGame`). By encapsulating such details, the `Program` class can now delegate the gameplay execution to the `TicTacToeGame` class without worrying about any details of the gameplay.

TicTacToe.Sim::Program

This class represents the application's run-time entry point.

```
class Program
{
    static void Main()
    {
        Log.Initialize();

        var numOfPlayers = 2;
        var players = new List<string>(numOfPlayers);
        for (var i = 0; i < numOfPlayers; i++)
```

```
        {
            Console.WriteLine("Enter player {0}'s name: ", i + 1);
            var name = Console.ReadLine();
            if (name.Length > 25) name = name.Substring(0, 25);
            players.Add(name);
        }

        var game = new TicTacToeGame(players[0], players[1]);

        game.Play();
        Log.Close();
    }
}
```

TicTacToe.Library::TicTacToeGame

This class represents the Tic-Tac-Toe game. All details related to the gameplay of Tic-Tac-Toe are contained in this class.

```
public class TicTacToeGame
{
    private readonly List<string> _players = new List<string>(2);
    private bool?[,] _grid;
    private bool? _isGameWon;

    /// <summary>
    /// Indicates if the game has been won by anyone or not.
    /// </summary>
    private bool IsGameWon
    {
        get { return _isGameWon.HasValue && _isGameWon.Value; }
    }

    protected TicTacToeGame()
    {
    }

    public TicTacToeGame(string playerX, string playerO)
    {
        _players.Add(playerX);
        _players.Add(playerO);
    }

    /// <summary>
    /// This method executes the play of the Tic-Tac-Toe game.
    /// </summary>
    public void Play()
    {
        while (true)
        {
            _grid = new bool?[3, 3];
            var isPlayerO = false;
            _isGameWon = null;
            do
            {
                var player = Convert.ToInt16(isPlayerO);

                // Update the UI.
                Console.WriteLine();
```

```
                Console.WriteLine("{0}'s turn...", _players[player]);

                // Retrieve and validate the input.
                string[] input = { };
                ValidateInput(ref input);

                Console.WriteLine();

                // Place the X or O in the appropriate grid cell.
                var row = Convert.ToInt16(input.First());
                var col = Convert.ToInt16(input.Last());
                _grid[row - 1, col - 1] = isPlayerO;

                // Display the grid.
                for (var r = 0; r < 3; r++)
                {
                    for (var c = 0; c < 3; c++)
                    {
                        var cell = _grid[r, c];
                        var cellValue = cell.HasValue
                            ? ((cell.Value ? "_O_" : "_X_"))
                            : "___";
                        Console.Write("{0}{1}", cellValue, (c < 2) ? "|" : "");
                    }
                    Console.WriteLine();
                }
                Console.WriteLine();

                // Check to see if there is a winner.
                CheckForWinner();

                // Switch to the other player.
                isPlayerO = !isPlayerO;

                // If the game has been won, exit the loop.
                if (IsGameWon) break;

                // Check for a draw game.
                _isGameWon = false;
                foreach (var cell in _grid)
                {
                    if (cell.HasValue) continue;

                    _isGameWon = null;
                    break;
                }
            } while (!_isGameWon.HasValue);

            Console.WriteLine();

            // Prompt the user to continue playing.
            Console.WriteLine("Play again? (y/n)");
            if (!Console.ReadKey().Key.Equals(ConsoleKey.Y))
                break;
            Console.WriteLine();
        }
    }
```

```
    /// <summary>
    /// This method checks the grid for a winning pattern
    /// (i.e three X's or three O's in a row).
    /// </summary>
    private void CheckForWinner()
    {
        // Check for 3-in-a-row on rows.
        CheckForThree();

        // Check for 3-in-a-row on columns.
        CheckForThree(true);

        // Check for 3-in-a-row diagonally.
        CheckForThreeDiagonal();
    }

    /// <summary>
    /// This method checks the grid (vertically or horizontally) for a
    /// winning pattern (i.e three X's or three O's in a row).
    /// </summary>
    /// <param name="checkVertically">If true, checks the grid vertically for the
    /// winning pattern; otherwise it checks the grid horizontally.</param>
    private void CheckForThree(bool checkVertically = false)
    {
        for (var c = 0; (c < 3) && !IsGameWon; c++)
        {
            var row = (checkVertically) ? 1 : c;
            var col = (checkVertically) ? c : 1;

            var midPt = _grid[row, col];
            if (!midPt.HasValue) continue;
            CheckForThreeFromMidpoint(row, col, checkVertically);
        }
    }

    /// <summary>
    /// This method determines if a point is in the middle of three in a row.  If so,
    /// the game is marked as won.
    /// </summary>
    /// <param name="row">The row of the midpoint.</param>
    /// <param name="col">The column of the midpoint.</param>
    /// <param name="checkVertically">Indicates if the check should be performed
    /// vertically or horizontally.  If true, the check should be performed vertically;
    /// otherwise the check is done horizontally.</param>
    private void CheckForThreeFromMidpoint(int row, int col, bool checkVertically)
    {
        var beforeMid = (checkVertically) ? _grid[row - 1, col] : _grid[row, col - 1];
        var afterMid = (checkVertically) ? _grid[row + 1, col] : _grid[row, col + 1];
        CheckForMatch(beforeMid, _grid[row, col], afterMid);
    }

    /// <summary>
    /// This method checks the grid for a winning diagonal pattern
    /// (i.e three X's or three O's in a row).  If a winning patern
    /// is found, the game is marked as won.
    /// </summary>
    private void CheckForThreeDiagonal()
    {
```

```
        var midPt = _grid[1, 1];

        if (!midPt.HasValue) return;

        CheckForMatch(_grid[0, 0], midPt, _grid[2, 2]);

        if (IsGameWon) return;

        CheckForMatch(_grid[2, 0], midPt, _grid[0, 2]);
    }

    /// <summary>
    /// This method determines if three points are the same.
    /// If they are, the game is marked as won.
    /// </summary>
    private void CheckForMatch(bool? left, bool? midPt, bool? right)
    {
        if ((!left.HasValue) || (!right.HasValue)) return;

        if (left == midPt && midPt == right)
        {
            _isGameWon = true;
        }
    }

    /// <summary>
    /// This method validates the input entered by the player.
    /// If invalid input is entered, the user is informed of the
    /// error and prompted to enter another set of input.
    /// </summary>
    /// <param name="input"></param>
    private void ValidateInput(ref string[] input)
    {
        do
        {
            try
            {
                input = Console.ReadLine().Split(' ');

                if (input.Length != 2) throw new FormatException();

                var row = short.Parse(input[0]);
                var col = short.Parse(input[1]);

                if ((row < 1 || row > 3) || (col < 1 || col > 3))
                    throw new FormatException();

                if (_grid[row - 1, col - 1].HasValue) throw new ReadOnlyException();
                break;
            }
            catch (FormatException)
            {
                Console.WriteLine("Invalid input! Try again...");
            }
            catch (ReadOnlyException)
            {
                Console.WriteLine("This cell has a value!  Try again...");
            }
```

```
        } while (true);
    }
}
```

Output

```
Enter player 1's name:
One
Enter player 2's name:
Two

One's turn...
3 3

___|___|___
___|___|___
___|___|_X_


Two's turn...
1 1

_O_|___|___
___|___|___
___|___|_X_


One's turn...
2 2

_O_|___|___
___|_X_|___
___|___|_X_


Two's turn...
1 3

_O_|___|_O_
___|_X_|___
___|___|_X_


One's turn...
1 2

_O_|_X_|_O_
___|_X_|___
___|___|_X_


Two's turn...
3 2

_O_|_X_|_O_
___|_X_|___
___|_O_|_X_


One's turn...
3 1
```

```
_O_|_X_|_O_
___|_X_|___
_X_|_O_|_X_


Two's turn...
2 1

_O_|_X_|_O_
_O_|_X_|___
_X_|_O_|_X_


One's turn...
2 3

_O_|_X_|_O_
_O_|_X_|_X_
_X_|_O_|_X_


Play again? (y/n)
y

One's turn...
1 1

_X_|___|___
___|___|___
___|___|___


Two's turn...
2 2

_X_|___|___
___|_O_|___
___|___|___


One's turn...
1 2

_X_|_X_|___
___|_O_|___
___|___|___


Two's turn...
3 1

_X_|_X_|___
___|_O_|___
_O_|___|___
```

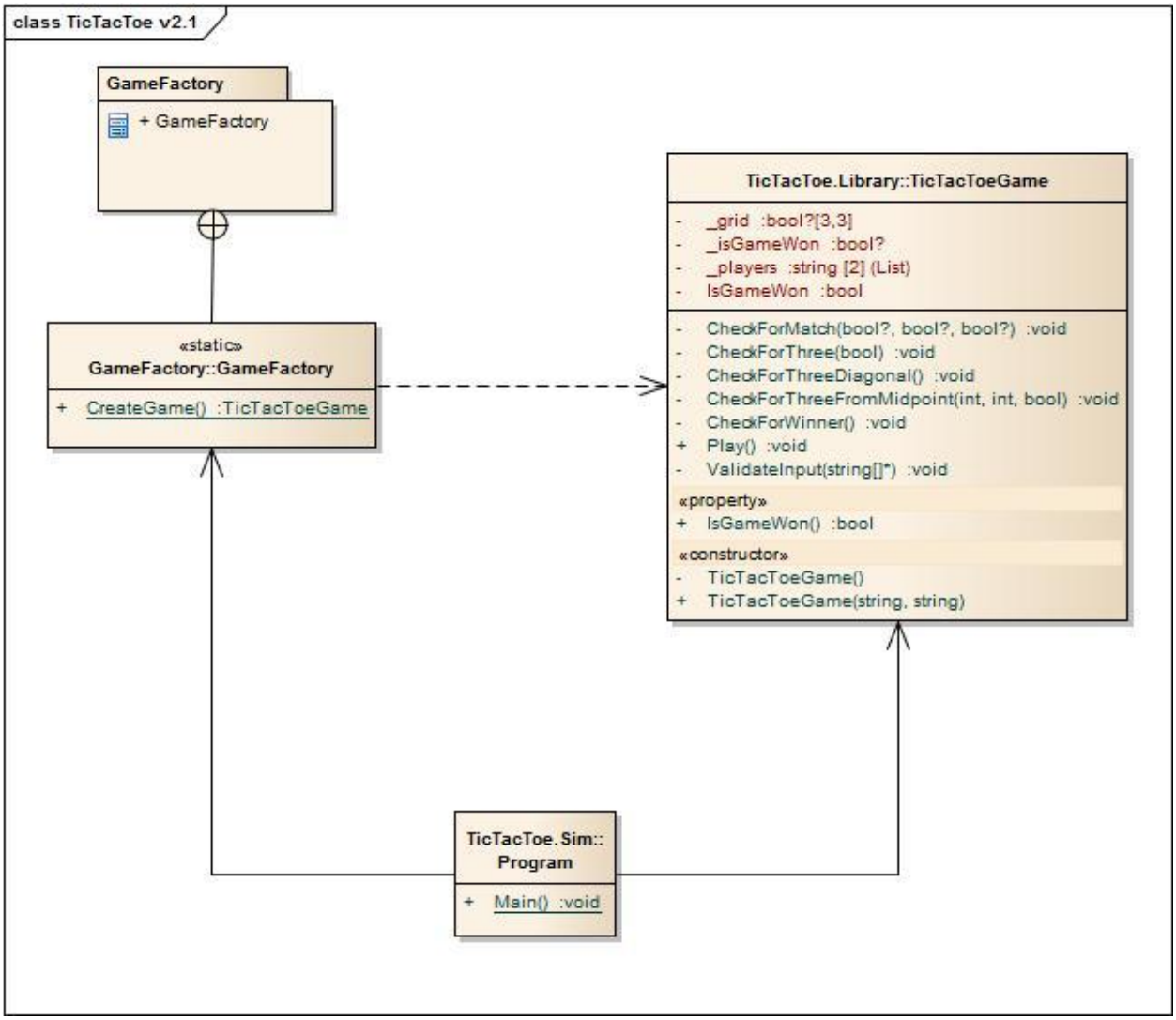
```


One's turn...
1 3

_X_|_X_|_X_
___|_O_|___
_O_|___|___


Play again? (y/n)
y

One's turn...
2 2

___|___|___
___|_X_|___
___|___|___


Two's turn...
3 3

___|___|___
___|_X_|___
___|___|_O_


One's turn...
1 3

___|___|_X_
___|_X_|___
___|___|_O_


Two's turn...
3 2

___|___|_X_
___|_X_|___
___|_O_|_O_


One's turn...
3 1

___|___|_X_
___|_X_|___
_X_|_O_|_O_


Play again? (y/n)
n
```

Notice how the winning diagonal pattern is now recognized by the application which fulfills the requirements of the system.

2.3 Create Maintainable & Reusable Design

In section 2.2, a more flexible design was implemented which produced immediate benefits over the legacy implementation. However, there are still things that could be improved upon to make the design a more suitable object-oriented solution. For instance, the `Program` class is still responsible for creating the names of the players. What happens if the number of players ever needs to change (such as the scenario for allowing only a single player to play)? Also, the `Program` class must still be aware of details of the `TicTacToeGame` class (such as the signature of the constructor). Asking the previous question again, if the number of players ever changed, then the signature of the constructor would have to change in two places: in the `TicTacToeGame` class and the `Program` class. Ideally, these types of details would be hidden from classes that used `TicTacToeGame` objects and the design could be updated to be more maintainable and reusable without needing to make changes within consuming classes like the `Program` class.

2.3.1 Class Diagram



2.3.2 Updated Implementation

In the previous design, the `Program` class was still responsible for creating the players' names and for creating the `TicTacToeGame` object. Here, those responsibilities have been moved to a factory class (`GameFactory`) which isolates the `Program` class from worrying about those details. The `Program` class still has a dependency on the `TicTacToeGame` class, but only because it needs access to a public member (`TicTacToeGame.Play()`) exposed by the `TicTacToeGame`'s public interface which is least likely to change.

The good thing about this design is that the `TicTacToeGame` class can now change the signature of its constructor(s) without it impacting the `Program` class. Even better, the `TicTacToeGame` could be extended with new implementations (future sub-classes) and added to the possible games returned by the `GameFactory` class and all of these changes are still isolated from the `Program` class! Also, the `Program` class would function appropriately with future sub-classes of `TicTacToeGame` without requiring any changes! This makes it easier to extend the application and/or change the behavior without changing anything in the `Program` class!

TicTacToe.Sim::Program

As with the previous versions, this is the application's run-time starting point. The biggest difference here is that the details of creating the player names and creating the instance of the `TicTacToeGame` object are now delegated to `GameFactory` class.

```
class Program
{
    static void Main()
    {
        Log.Initialize();

        var game = GameFactory.CreateGame();
        game.Play();

        Log.Close();
    }
}
```

GameFactory::GameFactory

This class is the "factory" for creating `TicTacToeGame` objects. It is also responsible for creating the players' names which are vital for object creation.

```
public static class GameFactory
{
    public static TicTacToeGame CreateGame()
    {
        var numOfPlayers = 2;
        var players = new List<string>(numOfPlayers);
        for (var i = 0; i < numOfPlayers; i++)
        {
            Console.WriteLine("Enter player {0}'s name: ", i + 1);
            var name = Console.ReadLine();
            if (name.Length > 25) name = name.Substring(0, 25);
            players.Add(name);
        }

        return new TicTacToeGame(players[0], players[1]);
    }
}
```

3. 'Game Factory' Application

This section covers the second application created for this project. Even though it is a separate application, it is essentially the same code base used for the Tic-Tac-Toe application, but redesigned to now support different types of games (UNO for this example, but could be Poker/Monopoly/etc.). This will illustrate how OOAD concepts can be utilized to extend a pre-existing code base for an entirely new application.

This application allows ANY game to be played as long as it implements a specific interface. The concept being that this application could be used by third party developers to easily plug-in to the existing application and allow the user to play their game without requiring changes to the Game Factory App's code.

3.1 Requirements

Game Factory App, version 1.0 Requirements List	'Game Factory App' Use Case
1. The player can choose to play Tic-Tac-Toe or the UNO card game. 2. The number of players will be determined by the gameplay of the chose game (two for Tic-Tac-Toe, two to ten for UNO, etc.). 3. Gameplay will be determined at run-time based upon the chosen game. 4. When the players no longer wish to continue playing, the application will close.	1. The application will present the player with a list of games that can be played. a. If the user enters an invalid value, the user will be informed of the error and allowed to try again. 2. Once a game has been chosen, gameplay will begin. a. Refer to each game's use case for more details specific to each game. 3. When the gameplay has completed, the application will close.

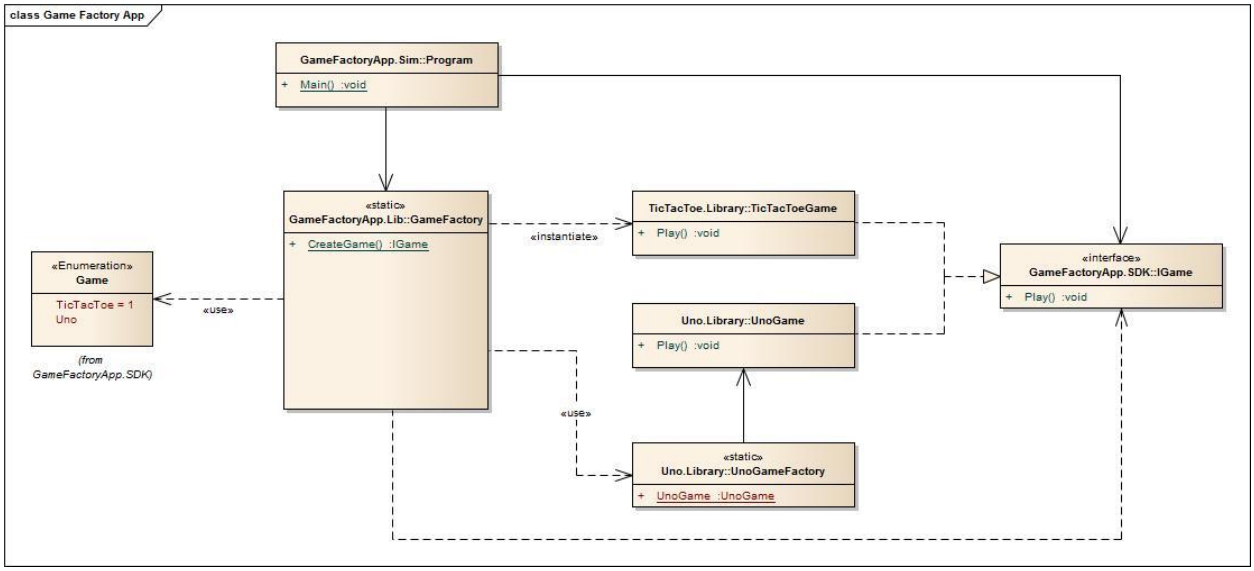
3.1.1 Alternate Path Use Cases

Tic-Tac-Toe Use Case (same as above)
1. The game will prompt the user for the players' names. 2. The players will play a game of Tic-Tac-Toe. a. The players will enter where in the grid they wish to place their symbol by entering two numbers separated by a space. The first number will be for the row and the second number will be for the column. i. If the values entered are invalid formats (i.e. letters) OR if either of the values entered are less than 1 and/or greater than 3 OR if the values are of a cell that already contains a symbol, then the player will be informed of the invalid input and the game will prompt the user to enter in valid values. 3. Once the game has been won or tied, the game will ask the players if they wish to play again. 4. When the players no longer wish to continue playing, the game will exit.

UNO Use Case

1. The game will prompt the user for the number of players.
2. The game will allow each user to enter their name.
3. The game will automatically determine the dealer (based upon the rules of UNO).
4. Each player is dealt 7 cards with the remaining cards placed face down to form a DRAW pile.
5. The top card of the DRAW pile is turned over to begin a DISCARD pile.
6. The active player has to match the card in the DISCARD pile either by rank, suit or action.
    a. If the player does not have anything to match, they must pick a card from the DRAW pile.
        i. If the card from the DRAW pile can be played, then the player can play that card to complete their turn. Otherwise, the current players' turn is over – without discarding – and play moves to the next player.
7. Play continues by repeating step 6 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.
    a. 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.
    b. 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.
8. Once a player has discarded all of their cards, the hand is over.
    a. 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.
9. Points are then awarded to the winner based on the accumulated point values associated with the cards remaining in each of the opposing players' hand.
10. The first player to accumulate 500 points overall, wins the game.
11. Once the game has been won, the game will exit.

3.2 Design



3.3 Implementation

As seen from the class diagram above, the design uses some of the same classes (and code), but the biggest difference here is that the `Program` class now simply expects an `IGame` object to be returned from the `GameFactory` class. This allows ANY game to be loaded at run-time (based upon the choice by the user) and played.

Classes for the UNO game can be referenced in Appendix A – *UNO Card Game*.

***NOTE**: The `GameFactory` class has dependencies on the concrete classes of `IGame` along with the `Game` enumerated type. In a more realistic scenario, a better solution would be to load these classes into an Inversion of Control (IoC) container from a configuration file. By doing this, it would remove the dependencies on the concrete classes of `IGame` and simply rely on retrieving the concrete instances at run-time. However, that goes beyond the scope of this project and the current implementation is shown for illustration purposes only.*

GameFactoryApp.SDK::IGame

This interface is the main interface that games loaded by the Game Factory App should implement in order to be playable.

```
public interface IGame
{
    void Play();
}
```

GameFactoryApp.SDK::Game (enum)

The enumerated type that represents the selection of games that can be selected by the user. This is shown for illustration purposes only as a stand-in for loadable plug-ins.

```
public enum Game
{
    TicTacToe = 1,
    Uno,
}
```

TicTacToe.Library::TicTacToeGame

Same as above.

GameFactoryApp.Lib::GameFactory

This class represents the "factory" that produces the games to be played by the user. The `CreateGame` method is the factory method that returns an instance of an `IGame` type, from which, the `IGame.Play()` method can be executed to begin gameplay. One thing to note is that even though the two IGame object are instantiated in different ways, those details are hidden from the `Program` class.

```
public static class GameFactory
{
    public static IGame CreateGame()
    {
        IGame game;

        var selection = ChooseGame();

        switch (selection)
        {
            case Game.TicTacToe:
                game = new TicTacToeGame();
                break;
            case Game.Uno:
                UnoGameFactory.Initialize();
                game = UnoGameFactory.UnoGame;
                break;
            default:
                throw new ArgumentOutOfRangeException();
        }

        return game;
    }

    private static Game ChooseGame()
    {
        Console.WriteLine("******** Welcome to the Game Factory ********");
        Console.WriteLine("  Please choose from the following games:");
        Console.WriteLine("  1 - TicTacToe");
        Console.WriteLine("  2 - UNO");
        Console.WriteLine("*****************************************");

        do
        {
            var input = Console.ReadKey();
            Console.WriteLine();
            switch (input.KeyChar)
            {
                case '1':
                case '2':
                    return (Game)Enum.Parse(typeof(Game), input.KeyChar.ToString());
            }
            Console.WriteLine("Not a valid choice.  Try again...");
        } while (true);
    }
}
```

GameFactoryApp.Sim::Program

This class is the Game Factory App's run-time starting point. This is essentially the same as the previous version except that now this class expects the game created from the factory to be an `IGame` type (versus a `TicTacToeGame` type), so that the `IGame.Play()` method can be called.

```
internal class Program
{
    private static void Main()
    {
        Log.Initialize();

        var game = GameFactory.CreateGame();
        game.Play();

        Log.Close();
    }
}
```

Output 1:

```
******** Welcome to the Game Factory ********
  Please choose from the following games:
  1 - TicTacToe
  2 - UNO
*********************************************
1
Enter player 1's name:
One
Enter player 2's name:
Two

One's turn...
...
...
...
Play again? (y/n)
n
```

Output 2:

```
******** Welcome to the Game Factory ********
  Please choose from the following games:
  1 - TicTacToe
  2 - UNO
*********************************************
2
Welcome to the card game UNO!

How many players will be playing (2-10)?
3
Enter player 1's name:
One
Enter player 2's name:
Two
Enter player 3's name:
Three

Let's get started!  Selecting a dealer now...
One is the dealer!
...
...
...
```

Appendix A – *UNO Card Game*

This appendix lists the source code utilized in the main sections of the project. It represents the card game, UNO. This is listed for illustration purposes as full implementation of the UNO card game was beyond the scope of this project.

Uno.Library::Player

```
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.Library::UnoCard

```
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);
    }
}
```

Uno.Library::UnoGame

```
public sealed class UnoGame : IGame
{
    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;
    }

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

        // TODO: Implement UNO gameplay.
        Console.WriteLine("...Playing UNO...");

        Console.ReadKey();
    }
```

```
}
```

Uno.Library::UnoGameFactory

```
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 readonly IUnityContainer _container = new UnityContainer();

    public static void Initialize()
    {
        _container.RegisterInstance(new UnoGame(CreateUnoCardDeck()),
            new ContainerControlledLifetimeManager());
        var uno = UnoGame;
        var min = UnoGame.MinNumOfPlayers;
        var max = UnoGame.MaxNumOfPlayers;

        Console.WriteLine("Welcome to the card game UNO!\n\n" +
                          "How 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();
    }

    public static UnoGame UnoGame
    {
        get { return _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;
    }
}
```

Non-Direct Activity Report

Date	Duration (minutes)	Specific Task / Activity
19-Aug-2014	85	Project #1 setup & research
25-Aug-2014	38	Work on Project #1
31-Aug-2014	125	Work on Project #1
9-Sep-2014	51	Work on Project #1
14-Sep-2014	144	Work on Project #1
19-Sep-2014	206	Work on Project #1
21-Sep-2014	130	Work on Project #1
22-Sep-2014	105	Work on Project #1
23-Sep-2014	156	Work on Project #1
24-Sep-2014	207	Work on Project #1
25-Sep-2014	90	Work on Project #1
27-Sep-2014	276	Work on Project #1
28-Sep-2014	420	Work on Project #1
29-Sep-2014	480	Work on Project #1
Sum for Report	**2513**	**/ 1500 (5 weeks @ 300/wk)**
Sum for Class	**2513**	**/ 4500 (15 weeks @ 300/wk)**