SSE 657

Object-Oriented Project Methods

Project #2

by

Jason Payne

October 27, 2014

TABLE OF CONTENTS

Topics Covered	Topic Examples
Large Scale Designs	• Gathering Features • Use-Case Diagrams • Domain Analysis • Design Patterns
System Architecture	• 3 Q's of Architecture • Risks • Scenarios
Design Principles	• Don't Repeat Yourself Principle • Single Responsibility Principle • Liskov Substitution Principle (delegation, composition, aggregation)

1. 'Game Factory' Application

The following sections will produce a software product from "cradle to grave" in order to demonstrate the concepts of object-oriented analysis & design (OOAD) on a larger scale product. This application was originally created in the previous project, but will be improved to meet the customers' demands. Due to scope limitation, not all features will be fully implemented. However, all features will be used for demonstration purposes in some capacity and this will be noted in the applicable sections.

'Game Factory' Vision Statement:

This project will focus on an application that allows users to play a game chosen from a library of board-based and/or card-based games that are included on the local system. The general concept is that different game libraries could be purchased and grouped together to form larger libraries of games (depending on what is available). In order to attract customers (and game developers), the application should support the following features:

- The application should support a plug-in architecture so that third-party developers can easily extend the application with new or updated games.
- It should include a user interface that is modern and customer-friendly.
- The application will provide the means for optional gameplay and scoreboard messages to be communicated by the game to the user.

NOTE: *Due to scope limitation, the optional gameplay and scoreboard area will not be featured in this project. It is presented for demonstration purposes and will only be referenced when discussing features and requirements.*

2. Large Scale Designs

In software development, the scope of a project can vary from as little as a few lines of code to as large as several million lines of code with complex system designs. However, if the large scale system is put under a proverbial microscope, one would see that a large scale system is nothing more than components and their sub-components integrated together to create a master product. Even with a poorly designed system, applying OOAD concepts reveal that larger scale systems are the product of several small scale sub-systems.

When deciding how or where to start with the software development process, the best way to look at a big problem is to visualize it as individual pieces of functionality. Doing this allows each individual component to be treated as a single individual system (sub-system) which is easier to define requirements for and is easier to design and implement. As each individual sub-system reaches maturity, the sub-systems can then be brought together to create the final product. Concepts that help accomplish this are:

- **Sub-system analysis** – by analyzing the sub-systems and their interactions, it will ensure that the overall system operates correctly in a real-world context.
- **Requirements analysis** – understanding the intended purpose and requirements of the sub-systems will make it easier to understand the intended purpose and requirements of the overall system.
- **Encapsulation** – makes the design more flexible and helps large scale systems to be broken up into smaller segments of functionality.
- **Interfaces** – by coding to an interface (versus an implementation), it helps large scale systems to remain loosely coupled with minimal dependencies.
- **Cohesion** – this ties in with encapsulation and coding to interfaces, but the higher the cohesion of a system, the more independent each sub-system is, and the easier it is to work on those systems independently.

In this section, an application will be presented as a large scale system with inherent risks. Concepts defined by OOAD will be used to help identify and isolate sub-systems that can be more easily designed and implemented. And finally, the complete system will be realized when the sub-systems are brought together as a final product.

NOTE: *Due to scope limitations, this application is presented as a large scale system. However, in reality, this application would not constitute a "large scale system" and is presented for illustration purposes only.*

2.1 System Features / Requirements

By analyzing the vision statement of the application, a list of system features can be created. Because each feature is a high-level feature (or requirement), system analysis reveals lower-level requirements for each feature and is listed below (Table 1):

Table 1: High-Level & Low-Level System Requirements

Feature ID	Feature (from customer / vision statement)	Requirement (for developer to implement)
1	Supports a plug-in architecture	• Game libraries can be updated between sessions. • Games can be easily added/removed (intuitive).

Feature ID	Feature (from customer / vision statement)	Requirement (for developer to implement)
2	Allows any game from the game library to be played	• The application will allow users to play any game from the currently loaded game library. • The rules and conditions of gameplay will be determined by the selected game at run-time.
3	Utilizes a modern user interface	• The application will be implemented as a Windows Presentation Foundation (WPF) application. • The GUI will provide a mechanism for accessing the company website in order to purchase new games. • The GUI will display the list of games that can be played.
4	Provides gameplay and scoreboard communications to the player	• A customizable control on the GUI will be provided for third-party developers to use as the gameplay and scoreboard communications area. • If there is no gameplay and scoreboard communications area for a game, then the customizable control will be hidden.

2.2 Use-Case Diagrams

After establishing features and requirements, it is easy to get stalled while focusing on the lower-level details of a system's capabilities. For large scale systems, after creating a list of features and requirements, the focus still needs to remain at the high-level to ensure that the system will do what it is intended to do. It is easy to forget about an important feature when delving into the low-level details too soon, but use-case diagrams help combat this.

Use-case diagrams are UML diagrams that help keep the focus on the high-level capabilities of a system. They provide the blueprints of a system and serve as a good traceability mechanism for tracing features to system capabilities (as seen in Figure 2-1). Had the diagram below not been created, it could have been very easy to overlook the capabilities required for updating the game library. However, by using OOAD principles, it was common sense to realize that there needed to be a sequence (or use-case) for creating/updating the game library.

It is important to note that use-case diagrams are not the same thing as use-cases. The diagrams provide a high-level overview of system capabilities while use-cases provide a step-by-step sequence of events necessary to accomplish a task within the system. In the diagram below, the "bubbles" within the use-case diagram typically represent a use-case for that task.

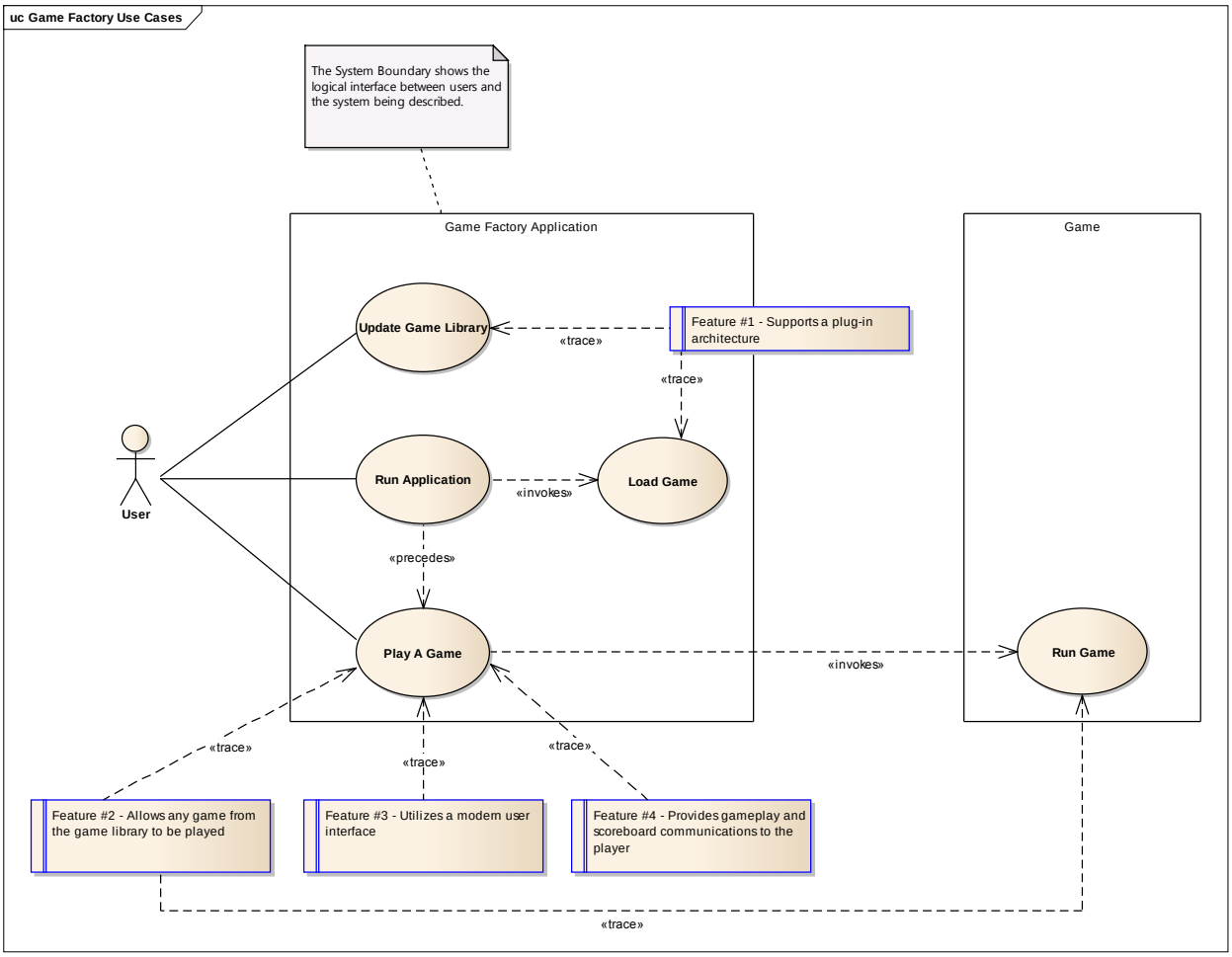


Figure 2-1: 'Game Factory' Use-Case Diagram

2.3 Domain Analysis

The formal definition of domain analysis states that it is "the process of identifying, collecting, organizing, and representing the relevant information of a domain, based upon the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within a domain." In layman's terms, it is the analysis of an entire system and its context that allows the pertinent information to be gathered in such a way that it can be presented to the customers in language they will understand and/or follow.

In Table 2, the results of domain analysis on the 'Game Factory' application are shown. In a real-world scenario, this is what could be presented to stake-holders as a textual demonstration of what the final product will be and what its capabilities will be.

Table 2: Domain Analysis Results

‘Game Factory’ Application Feature List
1. The application will support an intuitive plug-in architecture for updating game libraries. 2. The application will allow any game from the current game library to be played. 3. The application will utilize a modern, user-friendly interface. 4. The application will provide a mechanism for communicating gameplay and scoreboard messages to the user.

2.4 Design

Using the information gathered in the preceding sections, the next step should be to attempt to identify the sub-systems necessary to create the system. Figure 2-2 illustrates the sub-systems identified during analysis of the features and use-case diagrams.

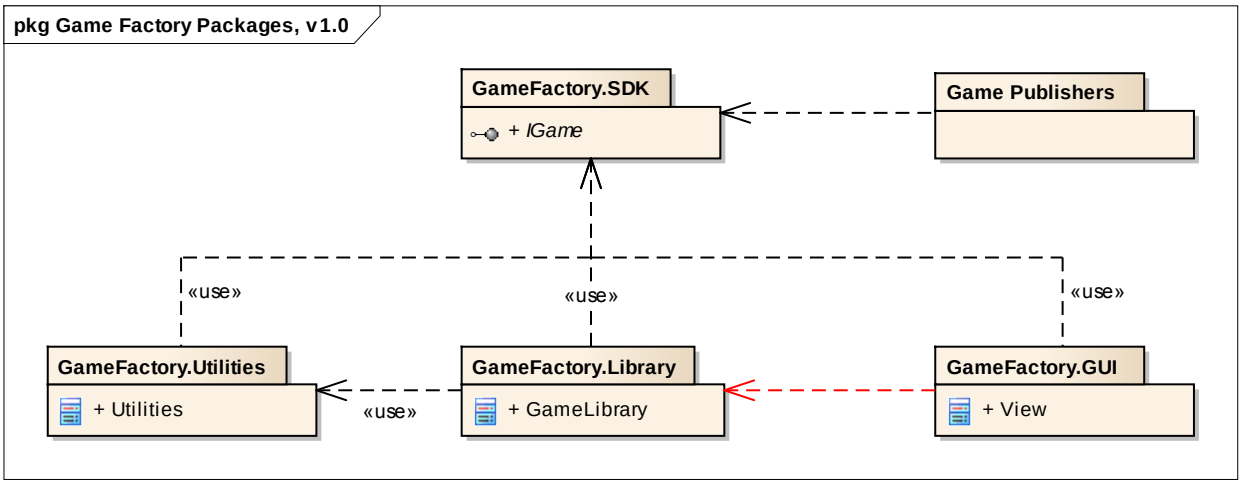


Figure 2-2: 'Game Factory' Sub-systems (Packages)

According to the diagram above, the analysis identified four sub-systems of the application and one (**Game Publishers**) external sub-system that depends on one of the sub-systems of the application (**GameFactory.SDK**). It should also be noted that the package that will be responsible for the GUI portion of the application (**GameFactory.GUI**) has a dependency on the package that will represent the core business objects of the application (**GameFactory.Library**).

Back in Table 1, it was determined that the GUI would be implemented utilizing the Windows Presentation Foundation framework. According to the package diagram above, this would be a rigid and inflexible solution due to the fact that anytime the business objects change, the GUI would have to be updated as well. Fortunately, a good design pattern exists that would help alleviate these issues. So, in order to remove the direct dependence of the GUI on the business objects and to take advantage of useful data-binding features of WPF, the Model-View-ViewModel (MVVM) design pattern should be used for this application.

2.4.1 Model-View-ViewModel (MVVM) Pattern

The MVVM pattern is an architectural design pattern similar to the well-known Model-View-Controller (MVC) pattern. It was created as a direct response to the very useful data-binding feature of WPF which facilitates the separation of view layer development from the rest of the pattern by removing virtually all GUI-specific code from the view layer. This is accomplished by splitting the GUI code into three parts:

- **Model** – represents the domain objects of the application.
- **View** – responsible for providing a visual representation of the domain objects in the way it is seen and interacted with by the user.
- **ViewModel** – wraps the data from the Model and makes it friendly for being presented and modified by the View. It also controls the View's interactions with the rest of the application.

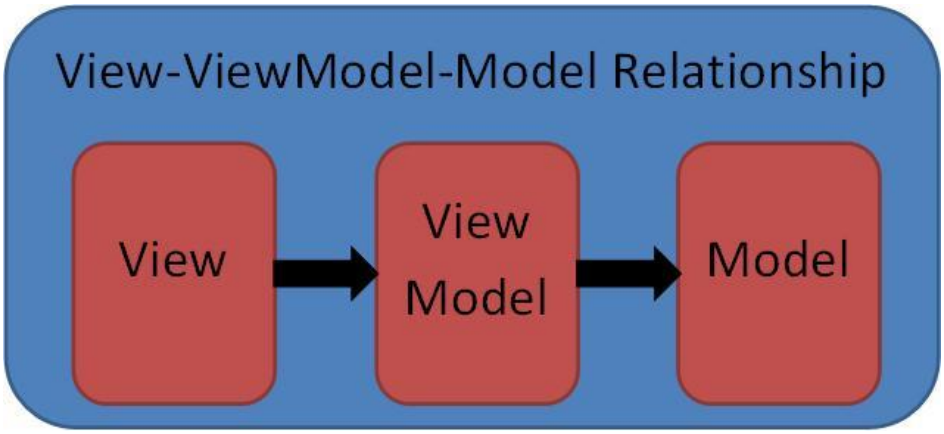


Figure 2-3: MVVM Layer Relationships

Figure 2-3 reveals the architectural relationship between the three layers of the MVVM pattern. The ViewModel should refer to the Model, but the Model should not be aware of the ViewModel. Likewise, the View should be aware of the ViewModel, but the ViewModel should be not aware of the View. In the figure above, the arrows are showing which MVVM part is aware of which.

By applying this design pattern, the package diagram will change such that there is a clear separation of domain object implementation and GUI implementation. The new package diagram is shown below.

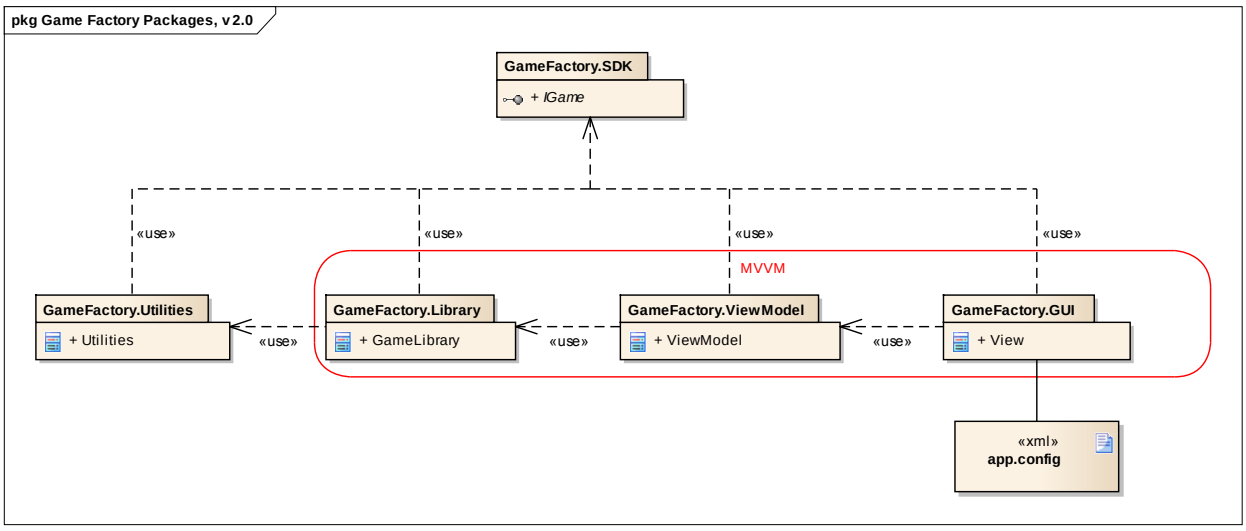


Figure 2-4: 'Game Factory' Sub-systems (Packages) with MVVM

3. System Architecture

When dealing with large scale systems, a common problem is figuring out where to start. Defining requirements and identifying the core parts of the application are critical, but once that has been accomplished, which part should be worked on first? This section will attempt to answer that question by applying OOAD principles to the 'Game Factory' application.

At this point, high-level features and low-level requirements have been established; use-case diagrams have been created providing a good high-level view of system capabilities; and certain design details have been resolved. With all of this information in place, which component should be worked on first? Decomposing a large application into smaller sub-systems is just one step in developing a good starting point to producing a large scale software product. It is also important to understand how the sub-systems work together and which ones may be more important than others. When these things are realized, a system's architecture has been established because the design structure is defined, the most important parts of the system are highlighted, and the relationship between those parts is realized.

3.1 3 Q's of Architecture

To determine the priority of the architectural significance of an application's sub-systems, three questions can be applied to each sub-system:

- **Essence** – is the component a core part of the system? Can the system exist with that component? If not, then that component is a core part of the system.
- **Meaning** – is there a lack of clarity or meaning for the component? Uncertainty of a component could take a good deal of time to gain clarity. Therefore, it is better to spend time on such components earlier rather than later.
- **Unfamiliarity** – how can a component or feature be implemented? If it is a difficult task or an unfamiliar task to get implemented, it is better to spend time on such components earlier rather than later.

The table below illustrates how the "3 Q's of Architecture" have been appliled to the 'Game Factory' application and its feature set.

Table 3: Architectural Characteristics of the 'Game Factory' Feature Set

Feature ID	Feature (from customer / vision statement)	Requirement (for developer to implement)	Architecture Characteristic
1	Supports a plug-in architecture	• Game libraries can be updated between sessions. • Games can be easily added/removed (intuitive).	- Essence - Meaning - Unfamiliar
2	Allows any game from the game library to be played	• The application will allow users to play any game from the currently loaded game library. • The rules and conditions of gameplay will be determined by the selected game at run-time.	- Essence

Feature ID	Feature (from customer / vision statement)	Requirement (for developer to implement)	Architecture Characteristic
3	Utilizes a modern user interface	• The application will be implemented as a Windows Presentation Foundation (WPF) application. • The GUI will provide a mechanism for accessing the company website in order to purchase new games. • The GUI will display the list of games that can be played.	- Essence - Meaning - Unfamiliar
4	Provides gameplay and scoreboard communications to the player	• A customizable control on the GUI will be provided for third-party developers to use as the gameplay and scoreboard communications area. • If there is no gameplay and scoreboard communications area for a game, then the customizable control will be hidden.	- Meaning - Unfamiliar

3.2 Risks

In the previous section, architectural priorities were determined based on the "3 Q's of Architecture." Answering those questions provides insight into the level of risk each feature introduces to the system. The higher the risk of the feature, the more the focus should be on that feature when starting implementation.

From Table 3 it can be seen that feature 1 and feature 3 have the highest architectural priority, so they naturally become the focus features of the application. Now that the focus features have been determined, the focus features must be scrutinized for further risks to determine the starting point. Using risk analysis it was determined that feature 3 introduced the greater risk based on the following factors:

1) "Dummy" games can be mocked as a stand-in for actual games coming from a plug-in. However, the GUI – by nature – cannot be mocked.
2) Due to the developer's unfamiliarity with WPF, there is another level of risk introduced at the GUI development feature. The same could be said for the plug-in architecture feature, but a well-known design pattern (Inversion of Control) utilized for the plug-in architecture reduces the risk when compared to the risk of WPF unfamiliarity.

3.2.1 Scenarios

A scenario provides all of the benefits of a normal use-case without requiring knowledge of the intimate details of a system. For assessing risks, it helps keep the focus on what the main expectations are of the system which traces to the critical features of the system. In most cases, it is typically a standard list of steps that will get performed while exercising a standard path through the use of an application. In doing this, it helps the developer focus on what needs to be

immediately implemented versus getting stalled by features/requirements that have little to no risk. The main focus of implementation should stay on the absolute minimum required to execute the scenario successfully. The following use-case serves as the scenario that will be used to drive the first stages of development. Implementing the application such that this scenario can succeed clearly defines the starting point of implementation and can also serve – once implemented – as a demonstration version of the product for the stakeholders.

Table 4: Basic Path Scenario Use-Case for the 'Game Factory' Application

Basic Path Scenario For GUI Interaction:
1. Start the application. 2. Select the desired game to play. 3. Click the ‘Play’ button. 4. Gameplay begins.

4. Scenario Implementation

At this point, a starting point for implementation has been determined and a scenario defining a basic use-case path through the application exists. With focus on implementing the bare minimum to get the scenario to succeed, implementation should begin by focusing on feature 3.

NOTE: *This section includes iterative premature versions of the source code. Any references to code files not included in this section indicate that there is no difference in the version at this point in the project and its final version. For such missing references, the code can be found in their final versions in* *Appendix A – 'Game Factory' Source Code**.*

GameFactory.Utilities::MockGame

This class represents mock implementations of games (`IGame`) that will be presented during testing/debugging/development efforts before final product delivery.

```
internal class MockGame : IGame
{
    public string Title { get; private set; }

    private MockGame()
    {
    }

    internal MockGame(string title)
    {
        Title = title;
    }

    public void Play()
    {
        var msg = string.Format("Playing {0}!!!", Title);
        MessageBox.Show(msg, Title, MessageBoxButtons.OK);
    }
}
```

GameFactory.Utilities::Utilities

For initial development efforts, this class will be responsible for creating and loading the mock implementations of games. Future versions will replace this mock implementation with actual context appropriate code that will dynamically load the games from a configuration file.

```
public class Utilities
{
    private static IUnityContainer _container;
    public static ObservableCollection<IGame> LoadGames()
    {
        return new ObservableCollection<IGame>()
        {
            new MockGame("Mock Game #1"),
            new MockGame("Mock Game #2")
        };
    }
}
```

GameFactory.Library::GameLibrary

Refer to GameFactory.GameLibrary::GameLibrary in Appendix A.

GameFactory.ViewModel::ViewModel

Refer to GameFactory.ViewModel::ViewModel in Appendix A.

GameFactory.GUI::View.xaml.cs ("code-behind")

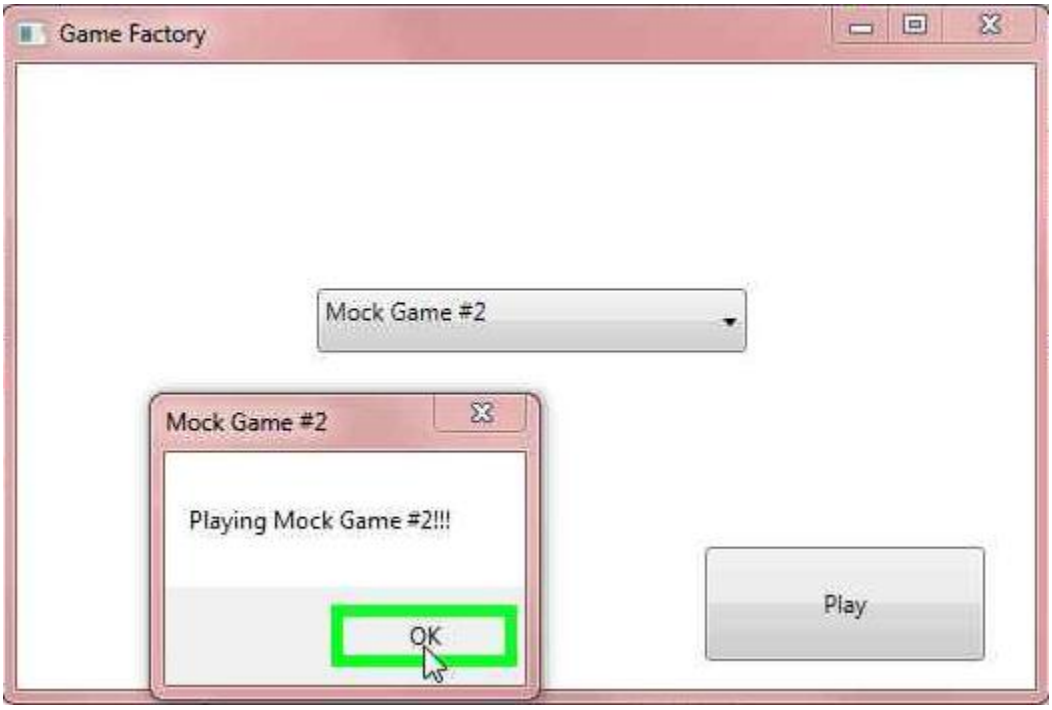
These next two classes represent the actual user-interface that is presented to the user at run-time.

```
public partial class View
{
    public View()
    {
        InitializeComponent();

        DataContext = new ViewModel();
    }
}
```

GameFactory.GUI::View.xaml (View)

```
<Window x:Class="GameFactory.GUI.View"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sdk="clr-namespace:GameFactory.SDK;assembly=GameFactory.SDK"
        Title="Game Factory" Height="350" Width="525">
    <Window.Resources>
        <DataTemplate x:Key="GameItemTemplate" DataType="sdk:IGame">
            <TextBlock Text="{Binding Path=Title}"
                       VerticalAlignment="Bottom" HorizontalAlignment="Center"/>
        </DataTemplate>
    </Window.Resources>
    <Grid>
        <Button x:Name="buttonPlay"
                Content="Play" Height="57" Width="140" Margin="344,241,0,0"
                VerticalAlignment="Top" HorizontalAlignment="Left" IsEnabled="True"
                Command="{Binding PlayGameCommand}"/>

        <ComboBox x:Name="comboBoxGames" ItemsSource="{Binding Games}"
                  ItemTemplate="{StaticResource GameItemTemplate}"
                  VerticalAlignment="Top" HorizontalAlignment="Left"
                  Height="32" Width="215" Margin="150,112,0,0"
                  SelectedItem="{Binding SelectedGame}">
        </ComboBox>
    </Grid>
</Window>
```

Output after implementing for feature 3 (Utilize a modern user interface):



Note how the mock games are presented in the drop-down box and the 'Play' button is enabled even when a game is not selected. These issues would need to be fixed prior to final delivery, but this accomplishes the task at hand which was to implement enough of the application in order to successfully traverse the scenario path.

Appendix A – *'Game Factory' Source Code*

This section includes the final version of the source code created for the 'Game Factory' application used in this project.

GameFactory.SDK::IGame

This is the interface provided as part of the public API to game publishers. Any game that wishes to be playable from the 'Game Factory' application should implement this interface. This interface also serves as the base type for all games that are consumed by the application (**Liskov Substitution Principle**). See the GameFactory.GUI::View.xaml section for an example of the IGame interface being utilized to satisfy the LSP.

```
public interface IGame
{
    string Title { get; }

    void Play();
}
```

GameFactory.Utilities::Utilities

This class serves as a utility class that can be consumed by objects when they need something utilitarian in nature to be performed. For this project, it serves as the game loader when the application starts up. This class helps satisfy the **Single Responsibility Principle** by isolating the responsibility of loading games to a single class. If a customized game loader was ever used to replace the current process used to load the games, the change would be isolated to this class only. It also satisfies the **Don't Repeat Yourself Principle** by working in conjunction with the MVVM pattern by being the single point of access when the games need to be loaded dynamically.

```
public class Utilities
{
    private static IUnityContainer _container;

    public static ObservableCollection<IGame> LoadGames()
    {
        // Use Inversion of Control to dynamically load games from configuration file.
        _container = _container ?? new UnityContainer().LoadConfiguration();
        var games = _container.ResolveAll<IGame>();
        return new ObservableCollection<IGame>(games);
    }
}
```

GameFactory.GameLibrary::GameLibrary (Model)

This class serves as the Model in the **Model**-View-ViewModel pattern and represents the domain object of the 'Game Factory' application. It also satisfies the **Single Responsibility Principle** by being the class solely responsible for representing the data model of the application.

```
public class GameLibrary
{
    public ObservableCollection<IGame> Games { get; private set; }

    public GameLibrary()
    {
        Games = Utilities.LoadGames();
    }

    public void PlayGame(string selectedGame)
```

```
    {
        var gameToPlay = Games.FirstOrDefault(game => selectedGame == game.Title);

        if (gameToPlay == null) return;
        gameToPlay.Play();
    }
}
```

GameFactory.ViewModel::DelegateCommand

This class allows custom WPF commands to be created in view models for data-binding purposes, thus satisfying the **Single Responsibility Principle**.

```
public class DelegateCommand : ICommand
{
    private Action _executeMethod;
    private Predicate<object> _canExecuteMethod;

    #region ICommand Members

    public DelegateCommand(Action executeMethod,
                           Predicate<object> canExecuteMethod = null)
    {
        _executeMethod = executeMethod;
        _canExecuteMethod = canExecuteMethod;
    }

    public bool CanExecute(object parameter)
    {
        if (_canExecuteMethod == null)
            return true;

        return _canExecuteMethod(parameter);
    }

    public event EventHandler CanExecuteChanged;

    public void Execute(object parameter)
    {
        _executeMethod();
    }

    #endregion
}
```

GameFactory.ViewModel::ViewModel (ViewModel)

This class serves as the ViewModel in the Model-View-**ViewModel** pattern. Its sole purpose is to serve as the mediator between the View and the Model, thus satisfying the **Single Responsibility Principle**.

```
public class ViewModel : INotifyPropertyChanged
{
    public ObservableCollection<IGame> Games { get; private set; }

    public GameLibrary Model { get; private set; }

    public DelegateCommand PlayGameCommand { get; private set; }

    public ViewModel()
```

```
    {
        Model = new GameLibrary();
        Games = Model.Games;
        SelectedGame = null;
        PlayGameCommand = new DelegateCommand(OnPlayGame);
    }

    private void OnPlayGame()
    {
        Model.PlayGame(SelectedGame.Title);
    }

    private IGame _selectedGame;

    public IGame SelectedGame
    {
        get { return _selectedGame; }
        set { _selectedGame = value; }
    }


    public event PropertyChangedEventHandler PropertyChanged;

    [NotifyPropertyChangedInvocator]
    protected virtual void
        OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        var handler = PropertyChanged;
        if (handler != null)
            handler(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

GameFactory.GUI::View.xaml.cs (View)

This class serves as the View in the Model-**View**-ViewModel pattern. Its sole purpose is to serve as the visual representation of the data model, thus satisfying the **Single Responsibility Principle**.

```
public partial class View
{
    public View()
    {
        InitializeComponent();

        DataContext = new ViewModel();
    }

    private void GameSelector_OnSelectionChanged(object sender,
                                                 SelectionChangedEventArgs e)
    {
        var selectedGame = comboBoxGames.SelectedItem.ToString();

        buttonPlay.IsEnabled = !string.IsNullOrWhiteSpace(selectedGame);
    }
}
```

GameFactory.GUI::View.xaml (View)

This object serves as the GUI-specific code needed to create windows and forms seen by the user during the application's run-time execution. It utilizes WPF's Extensible Application Markup Language (XAML) processor to create the user experience.

```
<Window x:Class="GameFactory.GUI.View"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sdk="clr-namespace:GameFactory.SDK;assembly=GameFactory.SDK"
        Title="Game Factory" Height="350" Width="525">
    <Window.Resources>
        <DataTemplate x:Key="GameItemTemplate" DataType="sdk:IGame">
            <TextBlock Text="{Binding Path=Title}"
                       VerticalAlignment="Bottom" HorizontalAlignment="Center"/>
        </DataTemplate>
    </Window.Resources>
    <Grid>
        <Button x:Name="buttonPlay"
                Content="Play" Height="57" Width="140" Margin="344,241,0,0"
                VerticalAlignment="Top" HorizontalAlignment="Left" IsEnabled="True"
                Command="{Binding PlayGameCommand}"/>

        <Button Content="Exit" HorizontalAlignment="Left" Height="57" Margin="36,241,0,0"
                VerticalAlignment="Top" Width="140" IsEnabled="True"/>

        <ComboBox x:Name="comboBoxGames" ItemsSource="{Binding Games}"
                  ItemTemplate="{StaticResource GameItemTemplate}"
                  VerticalAlignment="Top" HorizontalAlignment="Left"
                  Height="32" Width="215" Margin="150,112,0,0"
                  SelectionChanged="GameSelector_OnSelectionChanged"
                  SelectedItem="{Binding SelectedGame}">
        </ComboBox>
    </Grid>
</Window>
```

App.config

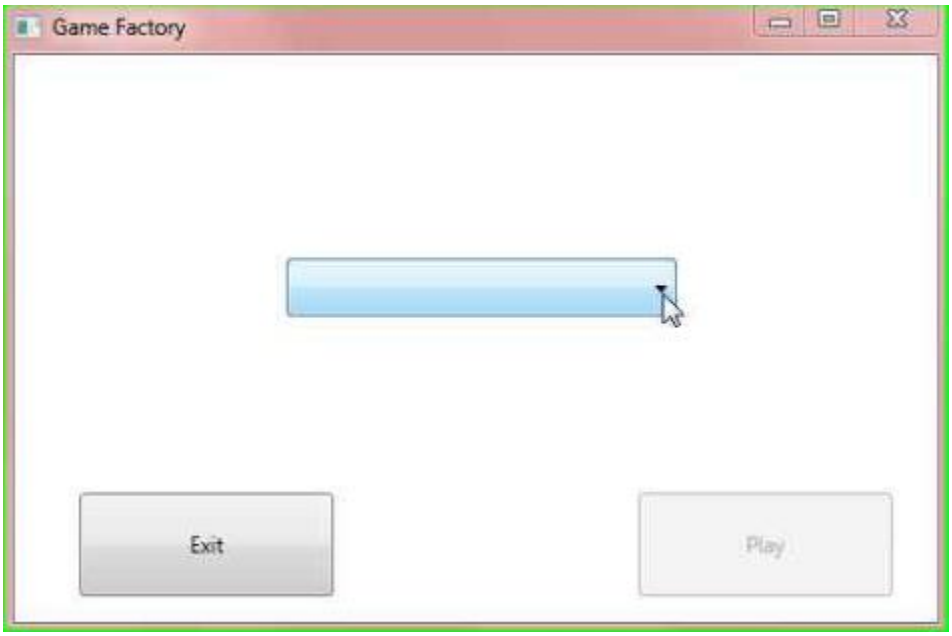
This file is the configuration file for the application. It is where the games are defined for the IoC container to build at run-time. This allows for easy addition/removal of games without affecting the code (i.e. plug-in architecture).

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

    <configSections>
        <section name="unity"
                 type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
                 Microsoft.Practices.Unity.Configuration"/>
    </configSections>

    <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
        <alias alias="IGame" type="GameFactory.SDK.IGame, GameFactory.SDK" />
        <alias alias="TicTacToe" type="GamePublisher.TicTacToe, GamePublisher" />
        <alias alias="Uno" type="GamePublisher.Uno, GamePublisher" />
        <alias alias="BlackJack" type="GamePublisher.BlackJack, GamePublisher" />
        <alias alias="Monopoly" type="GamePublisher.Monopoly, GamePublisher" />
        <alias alias="Checkers" type="GamePublisher.Checkers, GamePublisher" />
        <alias alias="Battleship" type="GamePublisher.Battleship, GamePublisher" />
        <container>
```

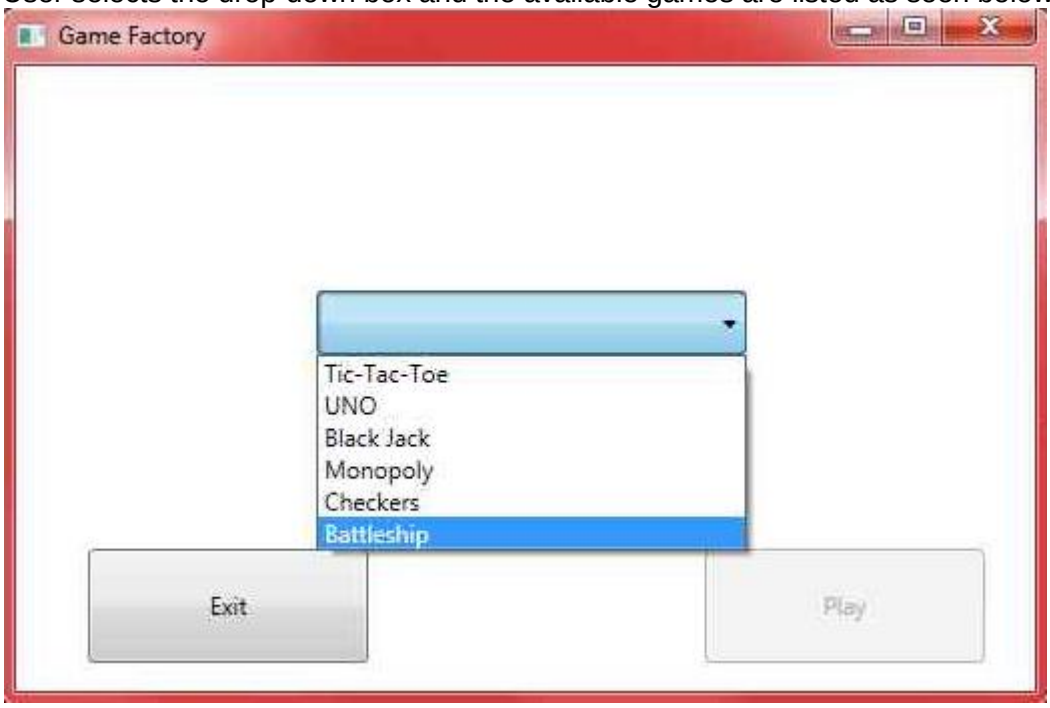
```
            <register type="IGame" mapTo="TicTacToe" name="TicTacToe"/>
            <register type="IGame" mapTo="Uno" name="Uno"/>
            <register type="IGame" mapTo="BlackJack" name="BlackJack"/>
            <register type="IGame" mapTo="Monopoly" name="Monopoly"/>
            <register type="IGame" mapTo="Checkers" name="Checkers"/>
            <register type="IGame" mapTo="Battleship" name="Battleship"/>
        </container>
    </unity>

    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
</configuration>
```

Final Version Output:

Step 1: User runs the application and the following window is displayed. Note that the 'Play' button is disabled.



Step 2: User selects the drop-down box and the available games are listed as seen below.



Step 3: Once a game has been selected, the 'Play' button is enabled and gameplay can begin.

Non-Direct Activity Report

Date	Duration (minutes)	Specific Task / Activity
8-Oct-2014	90	Work on Project #2
9-Oct-2014	27	Work on Project #2
11-Oct-2014	50	Work on Project #2
12-Oct-2014	168	Work on Project #2
19-Oct-2014	250	Work on Project #2
20-Oct-2014	230	Work on Project #2
22-Oct-2014	122	Work on Project #2
23-Oct-2014	274	Work on Project #2
26-Oct-2014	187	Work on Project #2
27-Oct-2014	540	Work on Project #2
Sum for Report	**1938**	**/ 1500 (5 weeks @ 300/wk)**
Sum for Class	**4451**	**/ 4500 (15 weeks @ 300/wk)**