Class diagram implementation notes (attributes and class extent)

A few notes before we begin

This assignment will not evaluate your implementation of the following :

- Associations. That includes having objects of a given class in a different class (e.g Games class has list of people that bought the game)
- Inheritance. That includes your work around for multi inheritance and multi aspect

Therefore, if you did them (correctly) Great you just saved yourself a lot of time for the majority of the semester :>. And if not, then that is also fine :> you will deal with it at a different time. (in class implementation part 2 and 3)

The only thing you will be evaluated on is your **attribute and extent persistency implementation**. And their **respective unit test**.

A few notes before we begin

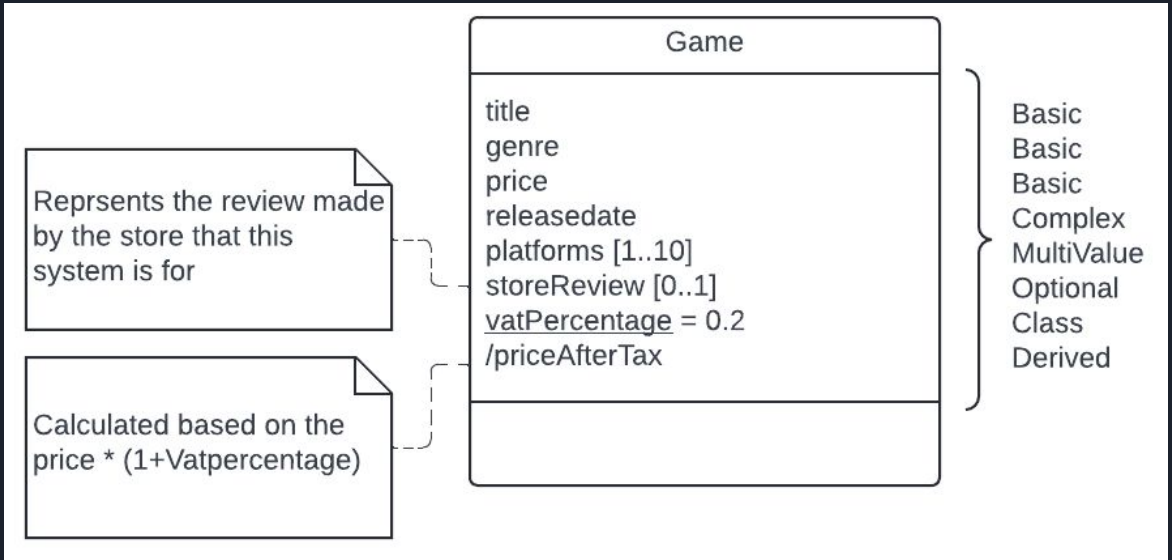
These slides are meant to serve as further assistance to those who might be struggling with certain implementations of their code related to the topics evaluated. Or if there are not sure on what to check for.

Therefore, these slides will **showcase only some** of the techniques of implementing the required points. More importantly, it will **specify exactly** what the requirements of each of the implementations to be considered a valid solution.

If you chose to do it differently and it works with what is required then that is also acceptable :>

Examples showcased

The examples will follow the implementation of one class called game. That stores objects related to video games sold in a store (the system is made for that specific store) and has the following attributes



Class Extent

Class extent : Is a set of all correctly created objects of a given class.

- Each new (Correctly) created object should be automatically placed in the class extent.
  Hint : it can be done using constructor.

```
//adding games to the extent class through the constructor
addGame(this);
```

- The collection for the class extent should be common to all instances of the class (static)
- Modification of the class extent outside of the class should not be possible

```
//Game Container
private static List<Game> games_List = new List<Game>();
```

Class Extent

- The class extent should follow the principles of encapsulation :
  - The collection should be secured and has the correct access modifiers.
  - No public methods should be used when modifying the collection
  - Getters of the collection should only return a copy of the collection or a reference that prevents the modification of the collection.

Example of creating an addGame method

```
//Adding new games to the container
private static void addGame(Game game)
{
    if (game == null)
    {
        throw new ArgumentException("Game cannot be null");
    }
    games_List.Add(game);
}
```

Extent Persistency

The extent collection should allow the possibility of writing to and reading from the permanent storage after each run of the application. As there is no databases used for this assignment, a simple accepted solution to persistency would be serializing the object and writing it to a file. (Don't forget about reading from the file and deserializing it)

Remember to :

- All classes should use the appropriate libraries (E.g System Xml, System.Xml.Serialization)
- All class attributes should be of a type that can be serialized
- Writing and reading methods should be static and should handle exceptions properly
- Since you will be working on a larger scale project than the example provided. It might be a good idea to create one method that reads and writes on all class extents and uses one file!

Extent Persistency Examples

Important setups

```
using System.Xml;
using System.Xml.Serialization;

namespace MP1
{
    //This class is used to keep track of games in a store
    [Serializable]
    public class Game
                                … (rest of the code)
```

Extent Persistency Examples

Writing method

```
public static void save (string path ="games.xml")
{
    StreamWriter file = File.CreateText(path);
    XmlSerializer xmlSerializer = new XmlSerializer(typeof(List<Game>));
    using (XmlTextWriter writer = new XmlTextWriter(file))
    {
        xmlSerializer.Serialize(writer, games_List);
    }
}
```

Extent Persistency Examples

Reading method snippet 1

```
public static bool load(string path = "games.xml")
{
    StreamReader file;
    try
    {
        file = File.OpenText(path);
    }
    catch (FileNotFoundException)
    {
        games_List.Clear();
        return false;
    }
```

Extent Persistency Examples

Reading method snippet 2

```
XmlSerializer xmlSerializer = new XmlSerializer (typeof(List<Game>));
            using (XmlTextReader reader = new XmlTextReader(file))
            {
                try
                {
                    games_List = (List<Game>)xmlSerializer.Deserialize(reader);
                }
                catch (InvalidCastException)
                {
                    games_List.Clear();
                    return false;
                }
```

Extent Persistency Examples

Reading method snippet 3

```
            catch (Exception)
            {
                games_List.Clear();
                return false;
            }
        }
        return true;
    }
```

Mandatory attributes (General)

Any non-optional attribute is considered **mandatory**

A note about string attribute : Empty strings are not allowed. So make sure to handle such cases.

- When using object types. It is necessary to make an appropriate checks of the values in the setter and throw an exception in the case of setting the incorrect values
- Remember to create a constructor that sets all of the mandatory attributes (aside from derived). In the case that conditions for validating the attributes are not met. The object should not be placed in the class extent.
- Remember to use the principle of encapsulation for the other attributes

Mandatory attributes (General) example

An example of creating a basic mandatory (string) attribute

```
private string _title;
public string Title
{
    get => _title;
    set
    {
        if( String.IsNullOrEmpty(value))
        {
            throw new ArgumentException("Title name can't be empty");
        }
        _title = value;
    }
}
```

Mandatory attributes (Complex)

Complex attribute : an attribute that contains two or more fields to create the value of a whole. Unable to be separated from the object that it is associated with (unlike an association)

The simplest example of showcasing a complex attribute is an attribute related to time with the type DataTime (in C#)

In the case of not using DateTime. A custom class containing the members of a complex attribute is a valid solution. However, remember to :

- Serialize the custom class
- Don’t forget about the principle of encapsulation

In either cases do not forget about the appropriate exception handling

Mandatory attributes (Multi-value)

Multi-Value attribute : Stores more than one value (can be implemented as optional - 0..*)

- Can be done using arrays or collections
- In the case of optional multi-value attributes. There is not need to ensure that there is at least one value added.
- Values added should be evaluated correctly before adding them (e.g Not null, Empty string, out of range, etc..)
- When retrieving the values from the attributes. It should be secured from any modification of the collection (in similar fashion to the class extent)
- Although setters are not mandatory. If used remember to apply the correct validations

Mandatory attributes (Class/static)

Class attribute : A value that is common to all objects of a given class. The value can be modified as long as there are not additional constraints

- Implemented as a static field
- Has a getter and setter

Mandatory attributes (Derived)

Derived attribute : The value of the attribute is calculated based on other data (other attributes, associations, etc)

- Most straightforward implementation is through getters
- Does not have a setter
- When calculating the value. Don’t forget about possible errors (e.g in the case that one or more of the base attributes is optional)

Optional attributes

Optional attribute : the value of this attribute can be null

- Can’t be implemented using simple types (int, double, etc) as they don’t allow null values
- When implementing the setter. Remember that if the value given is not empty. The appropriate constraints should be applied (e.g a non-empty string, a value that is within the specified range, etc)
- Remember about the principle of encapsulation for other attributes
- Remember about implementing other constructors to accommodate for having both required and optional attributes (Although there are methods in which it can be done using one constructor :>)

Unit testing

For unit testing please make sure to have a unit test for the following :

- Check if every attribute in your diagram was implemented correctly :
  - One test to check if gets the correct information back
  - A few tests (depending on the amount of exceptions) to showcase if the the exceptions are being thrown correctly
- A unit test for checking if your class extent is storing the correct classes
- A unit test for checking if the principle of encapsulation is met (if I modify an attribute without using a constructor would it change the values in the class extent)
- A unit test that checks for extent presestiency (are the files stored correctly and are they retrieved correctly after each new run)