Learning Objectives:

- Review Object-Oriented Programming and the Standard Template Library.
- Understand how Object Factories abstract the construction of an inheritance hierarchy of objects.
- Understand the structure and syntax of XML.
- Understand the theoretical relationship between XML and the Document Object Model (DOM).
- Understand how the DOM may be implemented in C++ (via a 3rd Party library).
- Demonstrate the ability to combine XML parsing, document object modelling (DOM) abstraction, Object Factories, and Assets Libraries into a single cohesive Game-Level initialization.

Task Summary:

The purpose of this assignment is to give you experience working with tools that facilitate flexible game architecture. In this assignment you will:

- construct an extensible, flexible game space using an inheritance hierarchy of objects that exhibit polymorphic behavior and rely extensively on templated storage containers;
- construct a container of object assets using the ObjectFactory pattern; and,
- utilize the TinyXML2 library to load, parse, and configure a game state from XML.

Requirements:

Each group's solution will be slightly different since we are all working towards different final projects.

Setup

Visual Studio Setup

This assignment differs substantially from projects you may have seen in Programming II. The code should be separated into numerous directories:

- ➢ Assets
- ➢ Game
- ➢ Source
- ➢ Third Party
- ➢ Temp

The Asset directory should contain a Config directory containing the *.xml configuration file(s). Eventually you may have Font and Sprite directories here. The Source directory contains all source code (*.cpp & *.h). The Third Party directory is for Third Party source code like TinyXML2 (and eventually SDL2 and Box2D). The Temp directory is the location where temporary object files will be compiled and the Game directory will be the source of the final, compiled, linked, and loaded executable. When you begin this project, you will need to do the following:

1. Create a new, empty Visual Studio project.
2. Create directories, mentioned above, in the project folder
3. move any .cpp and/or .h files to the Source directory (You'll need to remove them and add them back to the project for Visual Studio to recognize the change)

To configure this project, do the following:

1. Set Directories:
    a. In the solution explorer, right-click on the Project and select Properties from the dropdown
    b. Select Configuration Properties and select General
    c. Edit the Output Directory field to be: `$(ProjectDir)Game\`
    d. Edit the Intermediate Directory field to be: `$(ProjectDir)Temp\`
2. Set up for copying the Assets File into the Game directory after compilation.
    a. In the solution explorer, right-click on the Project and select Properties from the dropdown
    b. Select Build Events" then "Post-Build Event"
    c. in the command line field, add the following:
    ```
    xcopy "$(ProjectDir)Assets" "$(ProjectDir)Game\Assets" /E /I /F /Y
    ```

Your project should now compile and execute (and should be portable to other computers).

XML Setup

You are free to structure you XML file as you see fit, however, some considerations:

- Make it easily extensible… Can you easily add new/different object later?
- Make it organized and systematic - a mess of an XML file will be tough to load into your game!

Here is my suggested structure, as it will be easy to adapt in future assignments:

```
<?xml version="1.0" encoding="utf-8"?>
<Level>
      <Object type="ObjectName">
            <Body x=”0.0f” y=”0.0f” angle=”0.0f” />
      </Object>
      <Object type="Object2Name">
            <Body x=”1.0f” y=”1.0f” angle=”90.0f” />
      </Object>
</Level>
```

Object Class:

In general you will need to load at least three objects of different classes from an XML file. These should be stored in a std::vector<std::unique_ptr<Object>>. The parent class should have the following methods and members at minimum.

- **METHODS:**
  - default constructor
  - deconstructor
  - any other constructor’s deemed necessary
  - virtual std::unique_ptr<Object> update() = 0;
  - virtual void draw = 0;
- **MEMBERS:**
  - float x;
  - float y;
  - float angle;
  - or a struct with all these in it.

The update and draw methods will need to be overloaded in the child classes, but do not need to do anything. These will be used in Assignment 2.

Object Factory Class:

The objects should be created via the factory pattern, which, admittedly, will be fairly simple at this point. The object factory hierarchy should mirror the object hierarchy with the base class containing the following:

- **METHODS**
  - virtual std::unique_ptr<Object> create (tinyxml2::XMLElement* objectElement) =0;
- **MEMBERS**
  - NONE

Each child should overload the create method in order to create the specific child object for that factory. As you should be able to discern, the XML parsing for the object should be done within the factory. This keeps the loading method in the Engine class simple and easy to read.

Library Class:

The library class holds a std::map<std::string, std::unique_ptr<ObjectFactory>>. When an object needs to be created based upon a string loaded from the XML, that string is searched for in the map and the appropriate Object Factory's create method is called.

As an example, let's assume we have a pointer to an instance of the Library class called objectLibrary, the library member is called "library" and we want to create an object whose name is stored in a string variable called newObject. Then the following single line will create a fully initialized and ready to use object:

```
objectLibrary -> library.find(newObject)->second->create(objectXML);
```

What makes this super powerful, is we do not even need a switch or if statements to differentiate between what type of object we are creating! If we add a new object, we just have to create a new object factory and modify the library's map! No other code need be changed!

Right now the library class is very simple, but it will grow.

- **METHODS**
  - Constructor that populates the map!
  - Deconstructor that frees up memory
  - any other constructor's deemed necessary
- **MEMBERS**
  - std::map<std::string, std::unique_ptr<ObjectFactory>> library

Engine Class:

The engine class is what ties everything together. It holds the vector of Object* and the library. It will have a load level method that uses the library to populate the vector. It will also have update and draw methods that loop over the objects and call each of their corresponding methods. It looks like this:

- **METHODS**
  - Constructor that calls loadLevel!
  - Deconstructor that frees up memory
  - any other constructor's deemed necessary
  - void loadLevel(std::string levelPath)
  - void update();
  - void draw();
  - bool run(); //this calls update and draw and returns false when game is over. it's used by source.cpp.
- **MEMBERS**
  - std::unique_ptr<Library> objectLibrary;
  - std::vector<std::unique_ptr<Object>> objects;

```
int main()
{
      std::string path = "....";
      std::unique_ptr<Engine> engine {std::make_unique<Engine>(path)};
      while(engine->run()){}
      return 0;
}
```

Suggested Division of Labor

MEMBER 1:

- ➔ Game Objects
- ➔ Game Factories

MEMBER 2:

- ➔ Engine Class
- ➔ Library Class

Write all classes separately, then come together and make them work in harmony.

Side Assignments

TRIFORCE! 10% Save Game - Create a saveLevel method in the Engine class that says the games current state to an XML file. This can override the XML file that is loaded at the beginning. You may want to test the functionality by incrementing the members in your objects before saving them to the file.

JSON (20%) - find a third party library and use it to load from JSON instead of XML.