Duality Refactoring – Milestone 3

***Members***

*Kenny Kwan 9591648*

*Liem Ma 6286364*

*Renaud Brunet 6626742*

*Justin Trong Huan Do 6600514*

*Leo Collard 6624480*

# GitHub Link to Project’s Source Code

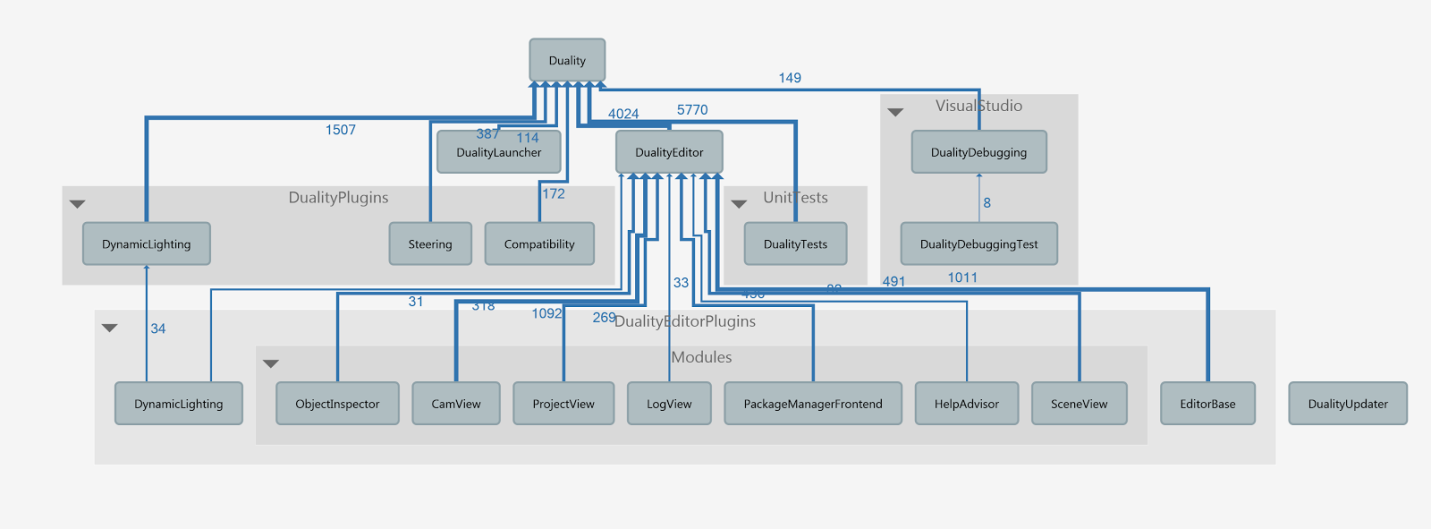
<https://github.com/undue/duality>

# Summary of Project

Duality is a 2D game engine that is currently being developed in C#. It acts as a base framework to assist in game creation and uses a plug-in architecture. It has a graphical user interface that allows for easy and streamlined game creation. It also has an editor that can be used to complete code that is not directly handled by the framework. It is currently developed by Adam Lair from Germany. He has been working in the game development industry for a long time, releasing several game projects since 2006.

This project has been in active development since 2011 and there are no official releases. Nonetheless, it is possible to download and utilize the latest binaries. There are currently 3 contributors, of whom one has developed heavily. Commits occur approximately every two days.

# Class Diagram of Actual System



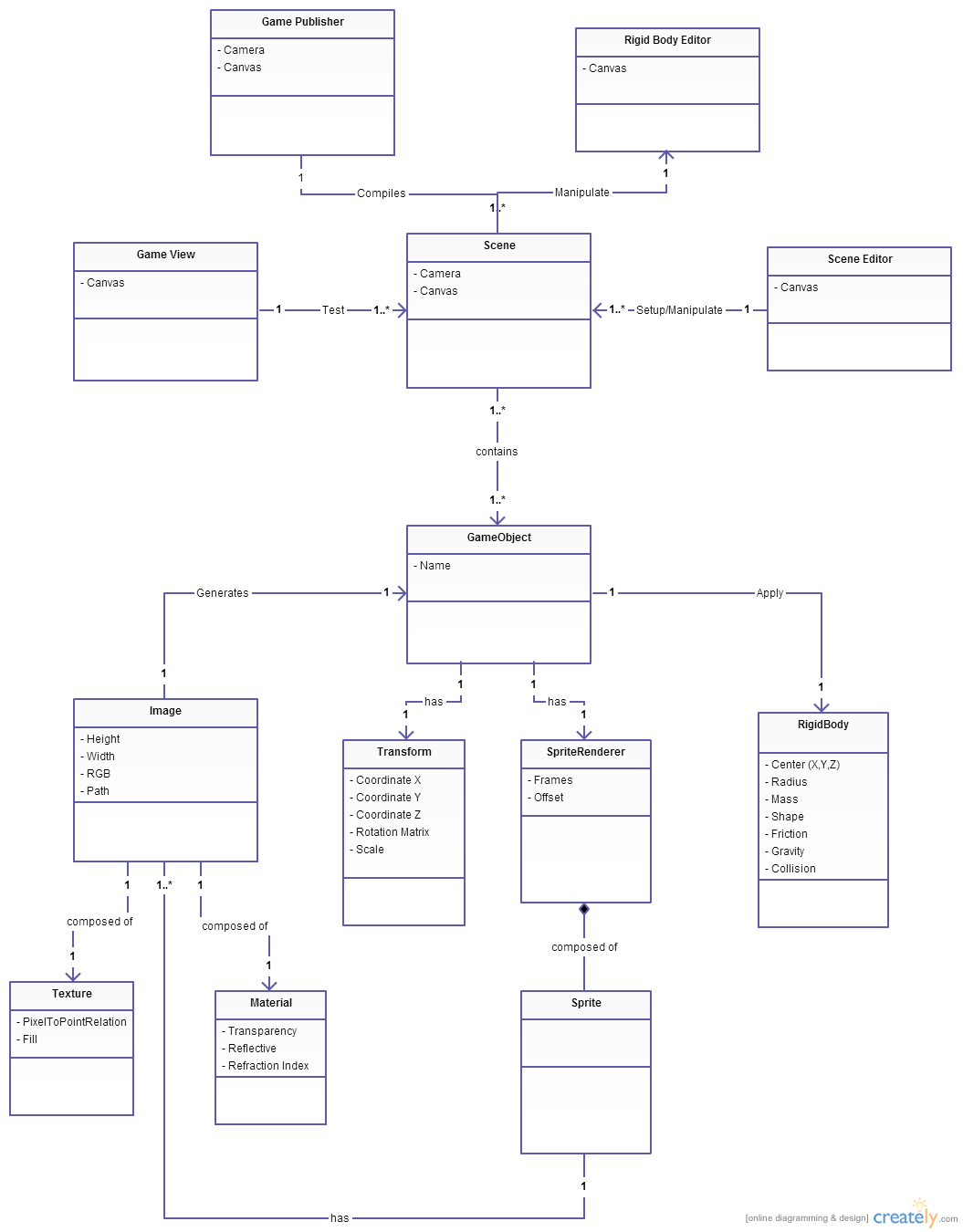
Figure 1- UML Diagram of System Overview and Dependencies

Figure 2 - Conceptual Class Diagram (Milestone 2)

**Actual Class Diagram (Milestone 3):**

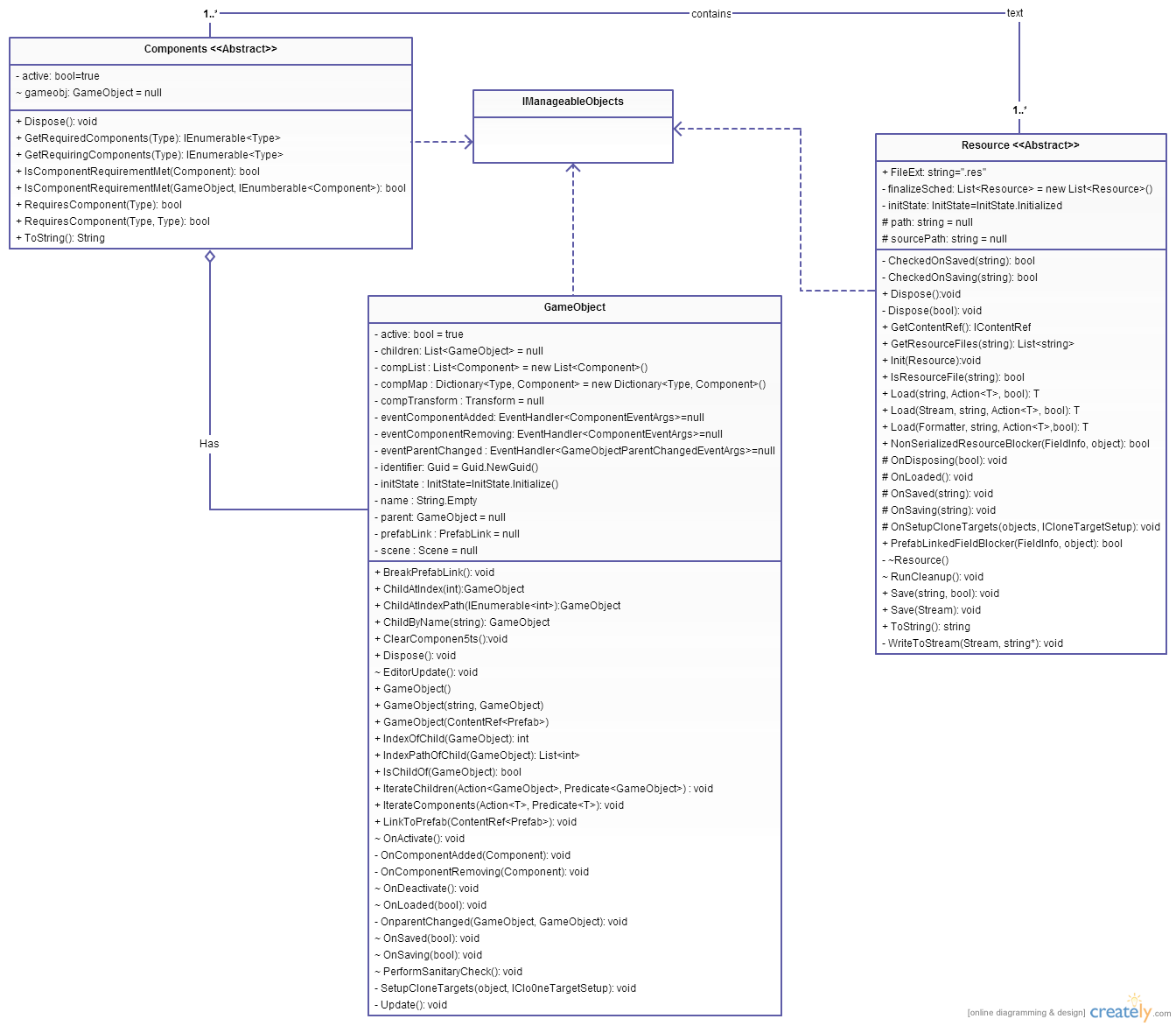
The extracted class diagram is split into three main subsections. These are the gameobject, resource and component diagrams.

Figure 3 - Main Diagram

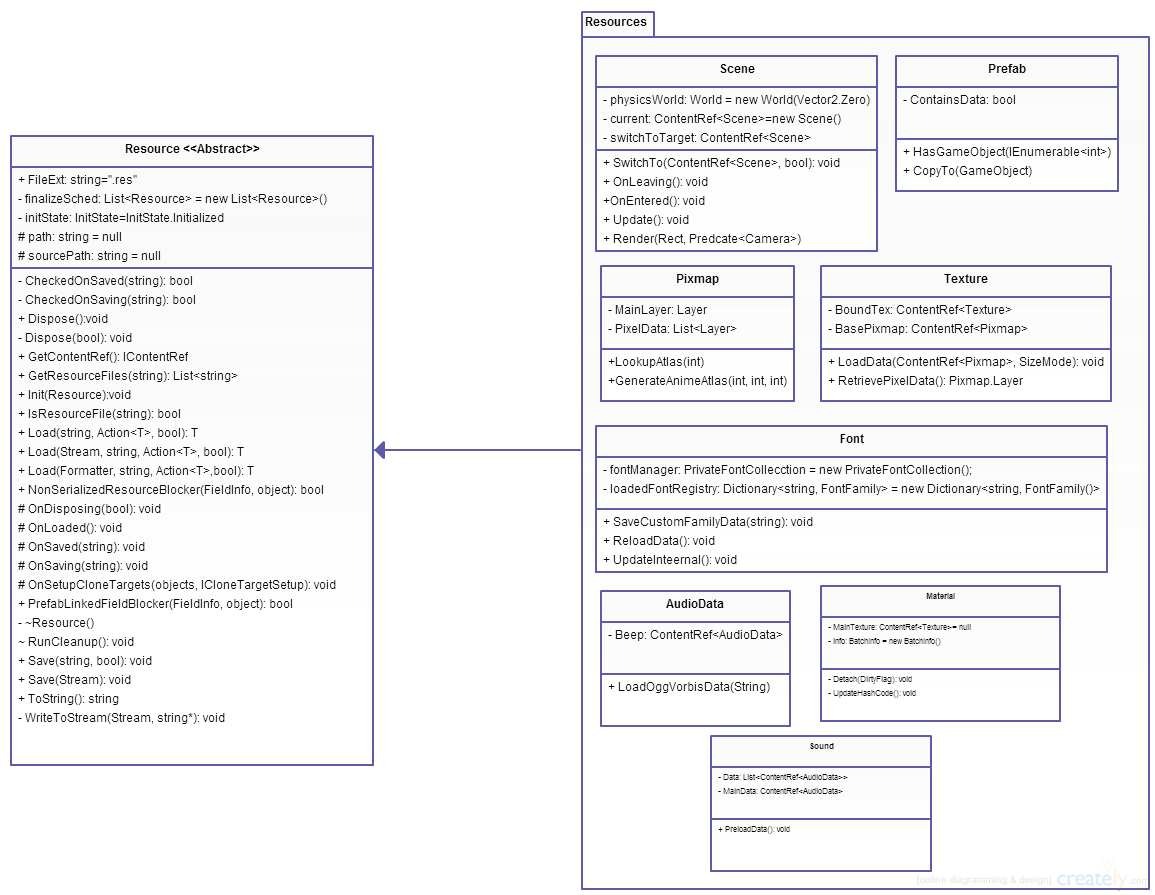


Figure 4 - Resource Diagram

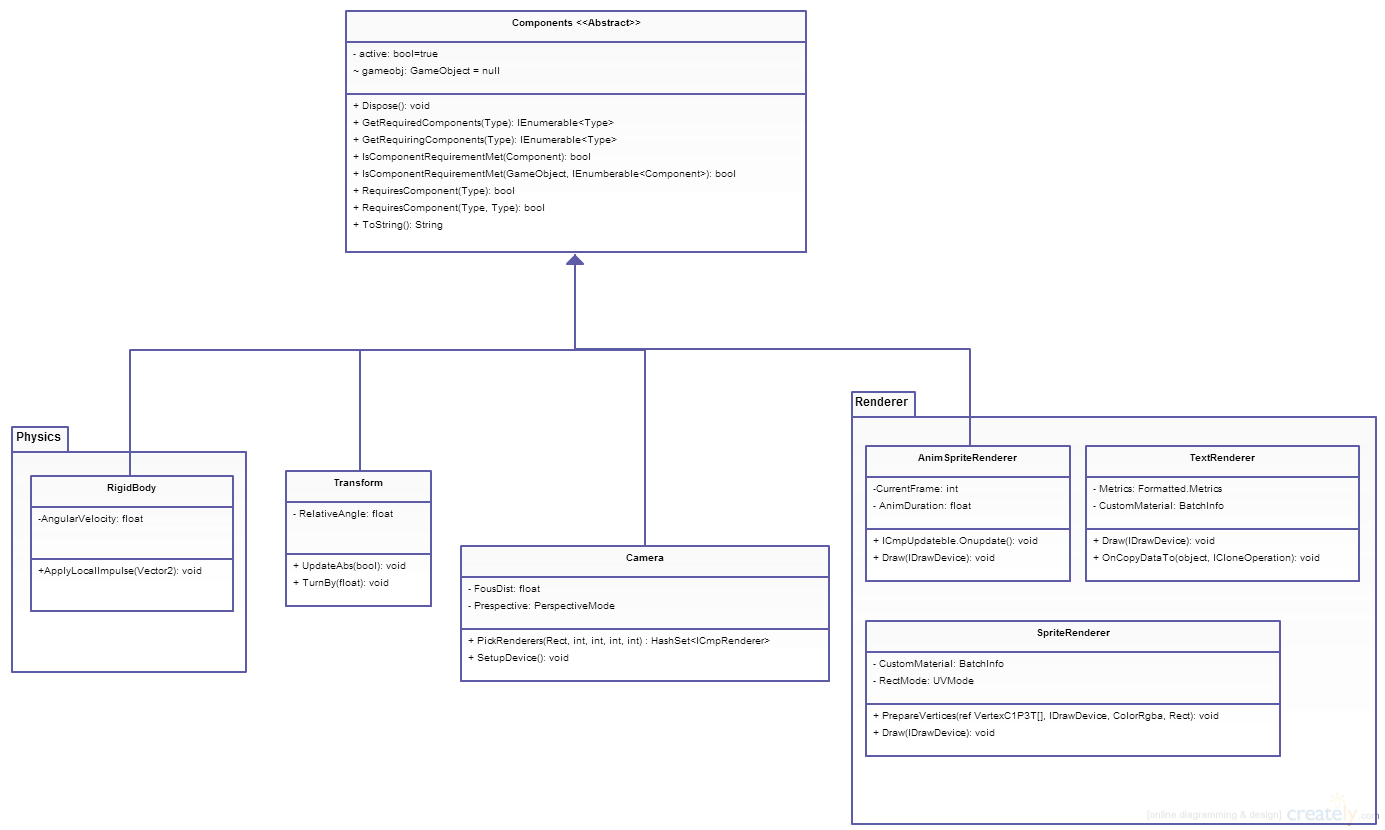


Figure 5 - Component Diagram

**Class Diagram Description:**

The class diagram was extracted from the Duality system using *Enterprise Architecture*. The class diagram is centered around the GameObject and all relevant classes that are associated with it. Since the output of *Enterprise Architecture* was cluttered and much too intricate to be understood, the important classes were extracted in order to create the class diagram of interest.

The GameObject, Component and Resources classes all implement the IManageableObjects interface. Resource is an abstract class that all resource classes must extend. Some of these include the Scene, Font and Sound class. All resources are associated with Components. The GameObject class has multiple components that define the class’s capabilities and responsibilities. Each of these components must extend the base Component abstract class which contains core functionality. Some of these classes are the RigidBody, Camera, Transform and the different types of Renderers.

**Purpose of each class and relationship to other classes:**

IManageableObject <<interface>>

This class serves as a main interface that all almost all objects in Duality will implement. This interface is to manage whether an object is in the editor’s scope or not using the Dispose() method. For the purposes of the area of interest in this milestone, the related classes that implement IManageableObject are GameObject, Resource and Components.

GameObject

Game object serves as a container to group together multiple functionalities provided by components. It acts as a foundation upon which a game entity can be defined. GameObjects can be placed on a Scene Resource and can interact with each other. Due to this, the classes that are related to GameObject are GameObject, the components class and resource class.

Component <<Abstract>>

Each individual class in the component folder is inherited from the Component class. These components are what define each and every individual GameObject. Each Component handles a piece of specific logic. It is possible to compose a GameObject out of any combination of Components at runtime. To avoid ambiguity, a GameObject may at most contain one Component of each type.

Components(RigidBody, Camera, Transform, Renderer)

Each individual class in the components folder serves as a definition of a Game Object. For example, a RigidBody represents a body instance for physical simulation, collision detection and response. The camera is responsible for rendering the current scene. It renders the scene from the objects point of view. A Transform represents a GameObject’s physical location in the world which is relative to its GameObject.Parent. It is the objects position, rotation and scale relative to the scene. The Renderer classes provide an animated sprite to visually represent the Game Object. All of these components inherits from the Component abstract class.

Resource <<abstract>>

This abstract class includes common properties that all resources should use. Some attributes include their path the file of interest and methods to read and save the resources as well. Resources are linked to one or more components. Depending on the resources properties it will impact the purpose of the component. For example, a Sound resource that is associated to a specific audio file will allow a component to play it.

Resources (Scene, Font, Material, Pixmap, Prefab, Sound, Texture, AudioData)

Each individual class in the resources folder serves as a runtime representation of the various forms of imported assets (material definitions, textures, pictures, sounds). Resources are associated with and used by the component classes. They provide the GameObject with specific data of interest that will allow it to properly be modeled and rendered.

**Conceptual Classes compared to Actual Classes**

The conceptual class diagram differs from the actual class diagram from the project mainly due to the architecture that was used. The conceptual class was not as detailed as the actual class diagram but it essentially contains all the important classes. The main difference however is that in the actual class diagram, the GameObject is composed of many different components. Each of these components inherits from the component class which is an abstract class. However, in the conceptual class diagram, these components do not inherit from a class but are all associated with the GameObject. It can also be seen that not all the components found in the actual class diagram are present in the conceptual class diagram such as the Camera class and various Renderer classes. The advantage of having all the components inheriting from the component superclass in the actual class diagram is that the GameObject class can now store them all in the same Components List to see what components exists with that Game Object. This could not be done in the conceptual class and if all separate components needed to be called, separate lists would be required. The benefit of the component class and the way that it is being used in the conceptual class is that it will reduce coupling by not requiring separate links to each individual components (RigidBody, SpriteRenderer, Transform etc.). The Resources class in our conceptual diagram were directly associated to the GameObject, however in the actual diagram, they are managed by different components which define a GameObject. There were also many resources that were not thought of such as Font, Sound, and Pixmap. All of these implement a resource interface that outlines similar behaviors of the various resources.

namespace Duality{

public abstract class Component : IManageableObject, IUniqueIdentifyable, ICloneExplicit{

   /// </summary>

   public enum InitContext

   public enum ShutdownContext

   internal GameObject gameobj    = null;

   private  bool active = true;

    public bool Active()

   public bool ActiveSingle()

   public bool Disposed()

   public GameObject GameObj()

   public void Dispose()

   public Component Clone()

public void CopyTo(Component target)

   public bool RequiresComponent(Type requiredType)

   public bool IsComponentRequirementMet(Component evenWhenRemovingThis = null)

public bool IsComponentRequirementMet(GameObject isMetInObj,

IEnumerable<Component> whenAddingThose = null)

   public IEnumerable<Type> GetRequiredComponents()

   public override string ToString()

}

namespace Duality{

public sealed class GameObject : IManageableObject, IUniqueIdentifyable, ICloneExplicit{

   Private Scene scene      = null;

   Private GameObject parent  = null;

   Private List<GameObject> children = null;

   Private List<Component> compList  = new List<Component>();

   Private Dictionary<Type,Component> compMap = new Dictionary<Type,Component>();

   Private string name  = string.Empty;

   Private bool active = true;

   Private InitState initState = InitState.Initialized;

   Private Transform compTransform   = null;

   public GameObject Parent()

   public Scene ParentScene()

   public bool Active()

   public string FullName()

public int HierarchyLevel()

   public IEnumerable<GameObject> Children()

public IEnumerable<GameObject> ChildrenDeep()

   public bool Disposed

   public Transform Transform()

   public Camera Camera

   public ICmpRenderer Renderer()

   public RigidBody RigidBody()

   public GameObject(string name, GameObject parent = null)

   public Component GetComponent(Type t)

   public T AddComponent<T>() where T : Component, new()

   public T RemoveComponent<T>() where T : Component

   public void ClearComponents()

public void Dispose()

}

Figure 6 - Component Class and GameObject Class Relationship

# Code Smells and System Level Refactoring

There are several code smells that can be identified in the Duality system. In this software system there are multiple data classes such as DualityAppData.cs and DualityUserData.cs. Data classes contain only fields with their respective getters and setters. These kinds of classes are not desired and should be modified appropriately. The fields inside these data classes are probably modified outside the class itself but these modifications cannot be captured by only looking at the class. To change a data class, it is needed to search the overall system to see where the get and set methods are used and try to find behavior that could be transferred to the data class. For example in the DualityAppData.cs, the value of the gfxAAQuality is used in the Init(ExecutionEnvironment env = ExecutionEnvironment.Unknown, ExecutionContext context = ExecutionContext.Unknown, string[] args = null) method of the DualityApp.cs and it would be possible to create a method in the DualityAppData.cs with the switch statement that would return the value of the targetAAlevel.

All resource classes include a static file extension at the beginning of them. These static classes can be regrouped together in order to allow them to be quickly accessed and edited when needed. This Static class will group all the static variables for the resources. This increases the coupling by one but will provide a library class which will contain information about the various extensions. (Refer to Figure 9)

In Scene.cs there is a GameObjectManager which acts as a middleman class. This is due to the fact that its purpose is to interact with a list of GameObjects. However, GameObjects should be coupled with the Scene class instead to increase cohesion. This is also inconsistent with other areas of code which do not use a middleman. For example, the GameObject class does not have a ComponentManager class present. To correct this, the GameObject list is copied into Scene.cs and all method declarations from the GameObjectManager are copied into to the interface IGameObjectManager. Those method implementations are then copied into the Scene.cs class which will implement IGameObjectManager. All references are then changed from the GameObjectManager to the now internal GameObject list and methods regarding the list. Every method transfer is copied and tested. Once all tests succceed, GameObjectManager can be removed from the Scene class. As an example, the GameObject List allObj can be copied into IGameObjectManager. The method implementation of AllObjects can then be copied into Scene.cs. The method OnSaving in Scene.cs that uses AllObjects can then be changed from “foreach (GameObject obj in this.objectManager.AllObjects)” to “foreach (GameObject obj in this.AllObjects)”. All tests are then run again and repeat the process for all other methods that will be used. This increases the cohesion of the Scene.cs class since it is where GameObjects are placed and modified. (Refer to Figure 8)

There is also a good amount of code cleanup that is possible to do. There are many libraries being imported but not used by the class. Although these libraries could potentially be used later on with unimplemented features, it creates unnecessary dependencies in the current code and should be removed. For example, Component.cs is using System.Reflection; using OpenTK; but these are never used in the code and can be safely removed.

# Specific Refactorings that you will implement in Milestone 4

After inspecting all the code smells from the solution, one of the refactorings that will be implemented in milestone 4 is to remove the middleman. Currently, the utility class GameObjectManager is used in the Scene class in order to keep track of the registered objects. However, there is a better approach. Instead, the scene class will directly implement the interface IGameObjectManager which will contain the method declarations for managing the registered object. As a final result, there will be no middleman between the Scene class and the GameObject class. The scene will be talk directly to the GameObject class, which will reduce coupling.

namespace Duality

{

   public class GameObjectManager

   {

          Private List<GameObject> allObj = new List<GameObject>();

          public int Count()

          public IEnumerable<GameObject> AllObjects

          public IEnumerable<GameObject> ActiveObjects

public IEnumerable<GameObject> RootObjects

          public IEnumerable<GameObject> ActiveRootObjects

          public event EventHandler<GameObjectEventArgs>  GameObjectAdded;

          public event EventHandler<GameObjectEventArgs>  GameObjectRemoved;

          public event EventHandler<GameObjectParentChangedEventArgs>   ParentChanged;

      public event EventHandler<ComponentEventArgs> ComponentAdded;

      public event EventHandler<ComponentEventArgs> ComponentRemoving;

      public bool AddObject(GameObject obj)

      public void AddObject(IEnumerable<GameObject> objEnum)

      public bool RemoveObject(GameObject obj)

      public void RemoveObject(IEnumerable<GameObject> objEnum)

      public void Clear()

      public void Flush()

      private bool AddObjectDeep(GameObject obj)

      private bool RemoveObjectDeep(GameObject obj)

      private void RegisterEvents(GameObject obj)

      private void UnregisterEvents(GameObject obj)

private void OnObjectAdded(GameObject obj)

private void OnObjectRemoved(GameObject obj)

          private void OnParentChanged(object sender, GameObjectParentChangedEventArgs e)

private void OnComponentAdded(object sender, ComponentEventArgs e)

private void OnComponentRemoving(object sender, ComponentEventArgs e)

}

Figure 7 - GameObjectManager Class

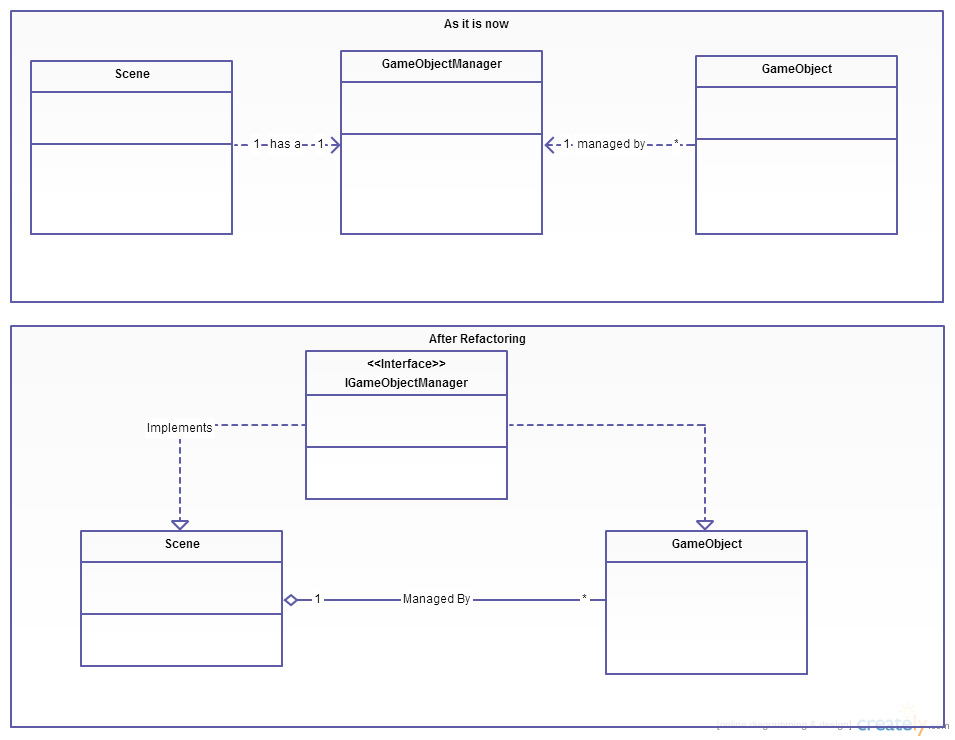


Figure 8 - Refactoring for GameObjectManager to Remove Middleman

Another possible refactoring would be the grouping of static file extension constants for the classes inheriting from the Resource interface into a separate static class. These values can be considered as global variables, which means that they can be accessed from anywhere in the solution simply by calling the static class that has all the file extensions available for the resources. Since those values are CONSTANTS, they cannot be modified except inside each class that is using them. By grouping them into a single class, it makes it easier to modify the hardcoded value of all extension at once from the same class.

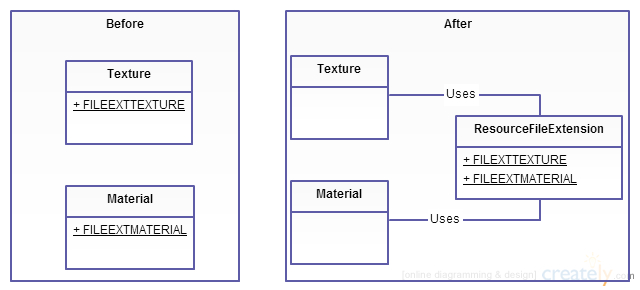


Figure 9 - Refactoring for Grouping Static File Extension Constants

In the GameObject class there is a very complex hierarchy of if statements. It is located in the setter of the Parent attribute, which is also a GameObject. There are more than 10 if statements to check different behaviors before actually executing code. It is possible to extract some methods out of it and also decompose the conditions into simpler ones so that it is more comprehensive.