Contents

[Week 1.1 Practical Overview 1](#_Toc398201054)

[Game Objects 1](#_Toc398201055)

[Design Options 2](#_Toc398201056)

[Deep Hierarchy 2](#_Toc398201057)

[Visual Example 3](#_Toc398201058)

[Potential Solution 4](#_Toc398201059)

[Class View 4](#_Toc398201060)

[Flat Hierarchy / Composition 5](#_Toc398201061)

[Game Objects and Components 5](#_Toc398201062)

[Game Objects 5](#_Toc398201063)

[Components 6](#_Toc398201064)

[Class Diagram 7](#_Toc398201065)

[Full Signature Class View 7](#_Toc398201066)

# Week 1.1 Practical Overview

## Game Objects

When developing games it is important to have a solid foundation on which you can easily build new games. At the core of each game are the objects which are placed within each level:

* Players
* NPCs
* Static and Dynamic Geometry
* Physics and Non-Physics objects
* Particles and Particle Emitters
* Animated and Non-Animated Models
* Custom Effect Models
* Audio Emitters and Listeners

The list of objects that could potentially be used within a game are nearly endless. How can we as developers provide a solid base from which to build all of these wide ranging objects?

1. Using your knowledge of object orientated programming how would you approach this?
2. What role does inheritance and polymorphism play?
3. What would a class require in order to be the base for every single object in your game?

# Design Options

## Deep Hierarchy

A deep hierarchy design is what you have been taught since you started here. Used throughout the 1990s and early 2000s the deep hierarchy model was the go to design at the time.

Unfortunately the design is not well suited to games development. Typically business applications have well defined and strict requirements that don’t vary much during development, this is nearly the complete opposite of games.

A games design can vary as the project progresses, individuals discover new gameplay mechanics and elements that weren’t part of the original vision. Unlike a business application we as developers will want to reuse as much of our code base between projects as possible.

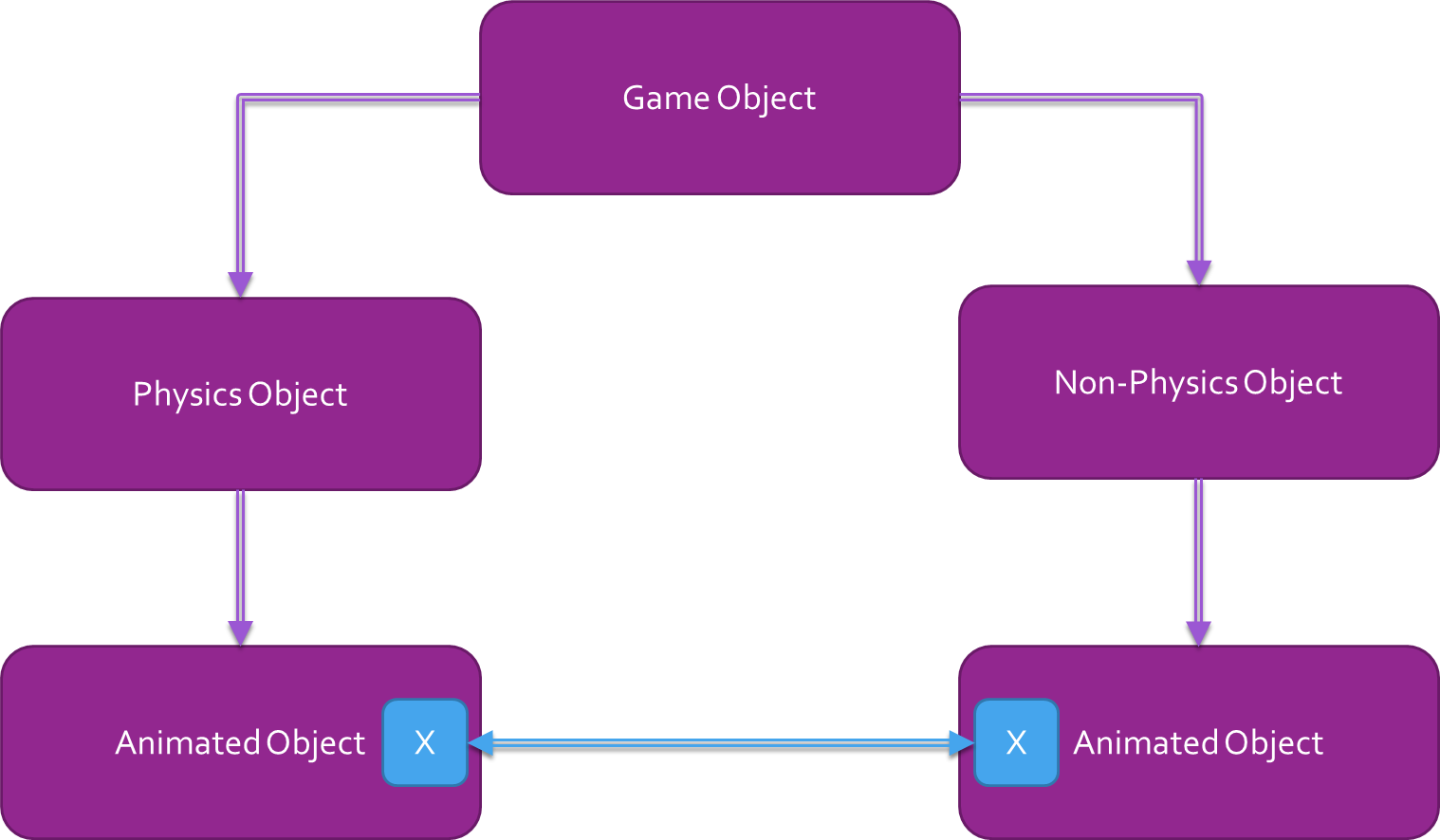
Deep hierarchies are rigid, inflexible, and hard to extend and result in an avalanche of unwanted software design side effects such as:

* Type Explosion
* Duplication of Functionality
* Bloating

The bottom line is that this design is too hard to change and isn’t well suited to software that requires tens of thousands of unique objects.

## Visual Example

In the image below we have a simple example of using a deep hierarchy design. We have a single base called Game Objects which is used for every single type of objects that can be placed in the game.



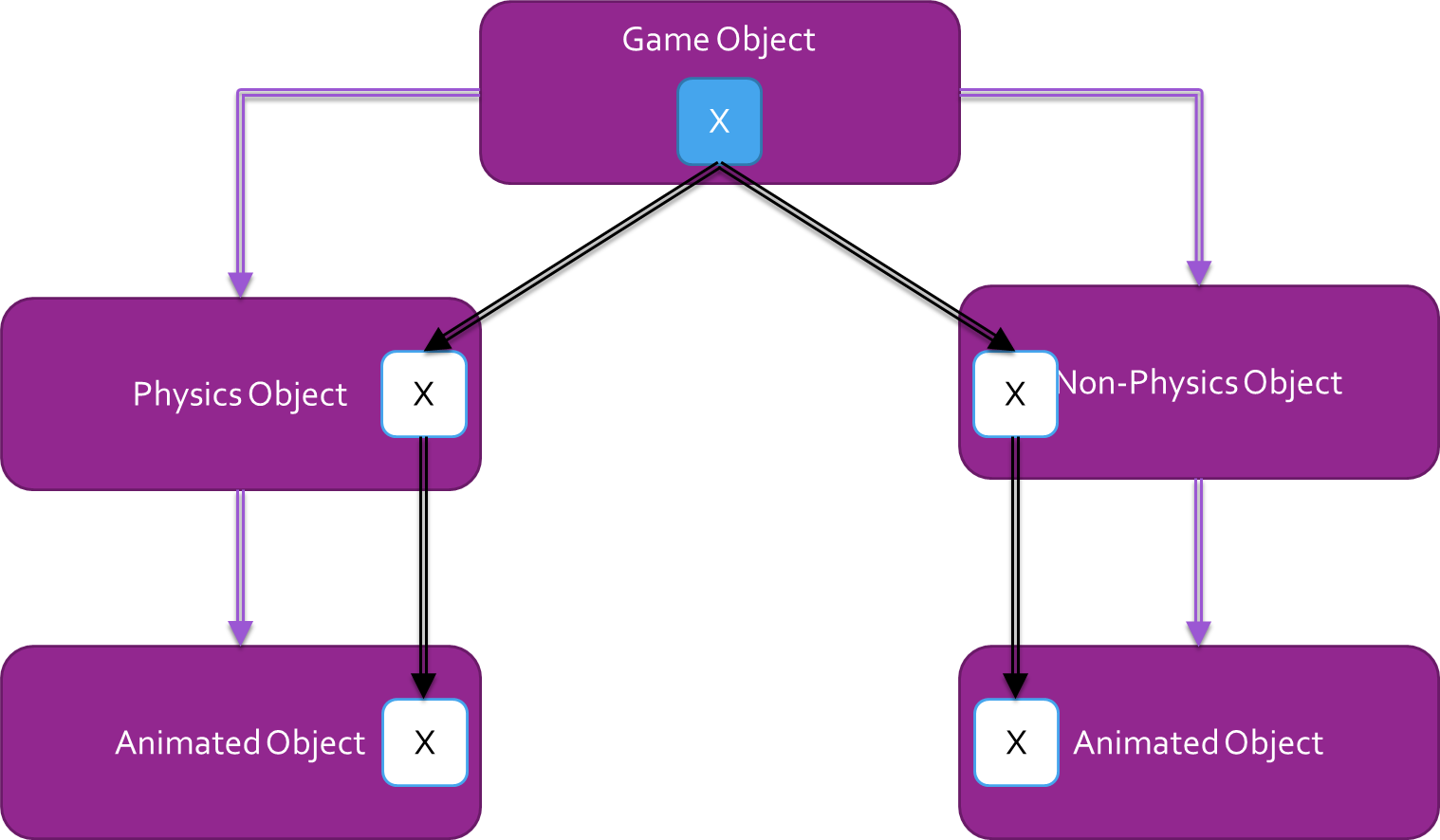
Imagine we have two branches in our inheritance hierarchy:

1. Physics Objects: Those who are part of the physics simulation. These objects will interact both the player and other physics based objects.
2. Non-Physics Objects: Those who are not part of the physics simulation and cannot interact with any Physics objects.
3. Both Physics and Non-Physics objects can be animated.
4. There are now two classes which essentially perform the same task but are in separate branches of the inheritance hierarchy.
5. **How can avoid having this duplicate functionality?**

## Potential Solution

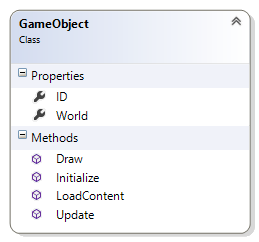
The solution which is typically employed in this situation is push the offending piece of functionality further up the inheritance hierarchy and more than likely all the way to the top, GameObject. Now both Physics and Non-Physics objects have the required animation functionality and there is not duplication, problem solved?

Not really, we have fixed the issue of repeated functionality but we have created an even more dangerous situation known as bloating or drifting of functionality to the base class. The animation code that we moved previously may only be used by 5% of the overall objects in our code but because of the move 100% of the objects have this animation functionality. This immediately increase the size in memory of every single game object derivative created.



It should be clear that if we were to continue with this implementation and every time we had duplicate functionality we would be forced to push the code to further up the hierarchy. At the end of the project your objects would have huge amounts of unnecessary functionality and bringing this code base forward to create a sequel or a new game would be an enormous effort and would more than likely result in starting from scratch and repeating the same mistakes.

## Class View



## Flat Hierarchy / Composition

A flat hierarchy design is an approach that is not taught in software development. The design has been gaining prominence in games development since the early/mid 2000s. Instead of having a deep inheritance hierarchy we will now use **composition** to create game objects.

Game Engine which are built using composition are commonly known as:

* Game Object Component Model
* Game Object Component Architecture

Commercial game engines such as Unreal Engine 4, Unity and CryEngine are all built using this design. If you want to be able to develop multiple games from a single code base without having to restart each time then this architecture is the clear winner at the moment. Some notable games that have been developed using the GOCM design include:

* Call of Duty Series
* Halo Series
* Crysis Series
* BioShock Series
* XCOM: Enemy Unknown
* Mass Effect Series
* Destiny
* Prototype 1 & 2

## Game Objects and Components

So what the hell is **composition** and how would I implement the **Game Object Component Model?**

## Game Objects

In a game engine which utilizes composition we still have the Game Object class but we no longer use it to hold any functionality. Instead the Game Object simply acts as a container for other objects known as components.

Every Game Object will only have two key properties:

1. ID : a unique string that will be used to identify individual game objects
2. World: a matrix holding the current transformations for the object

Beyond this the game object will provide a collection that can hold as many components as needed. This collection will responsible for the initialization, updating and drawing of the components.

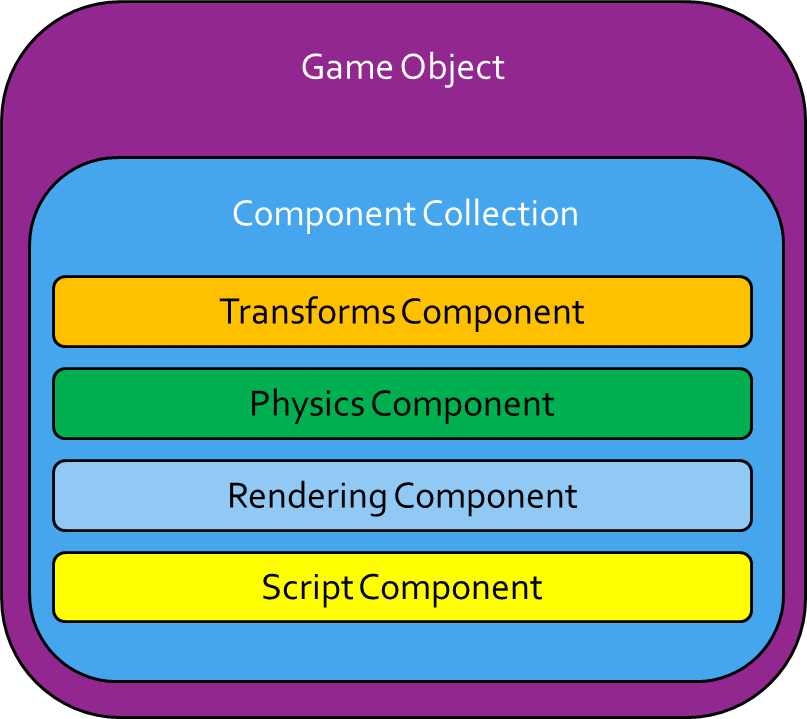


The easiest way to think of this is:

1. Game Objects are containers for components and contain with no game functionality
2. Components implement single pieces of functionality that can be reused anywhere
3. Game Objects gain their functionality through their mix of components

## Components

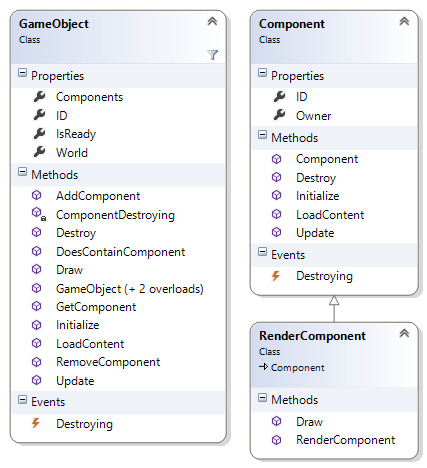
Components are single self-contained units that perform a single specialised task. They should not be tied to any other components or tied to a specific type of game object, this way we can reuse all of our components among all of our game objects.



Over the semester we will be developing a series of components that perform specific tasks:

* ModelComponent
* CustomEffectComponent
* PhysicsComponent
* Controllers
* ScriptComponent

# Class Diagram



# Full Signature Class View

