GAME ENGINE ARCHITECTURE
```csharp
public class scriptPlayer : MonoBehaviour
{
    public bool going, spotReached;
    public short direction, movesPlayerMoved; // Added 'short' for direction and movesPlayerMoved
    public Color colorStart, Color colorEnd;

    void Start() // Use this for setting up initial state
    {
        going = false; north = speed = Globals.pSpeed;
    }

    void Update() // If 'Global' is false, do nothing
    {
        if (!Global)
        {
            // Check if still moving
            if (going)
            {
                // Spring to
                if (spotReached)
                {
                    // If north, direction = 1; transform.eulerAngles = new Vector3(0, 0, 90);
                    if (north) direction = 1; transform.eulerAngles = new Vector3(0, 0, 90);
                    // If south, direction = 2; transform.eulerAngles = new Vector3(0, 0, 80);
                    if (south) direction = 2; transform.eulerAngles = new Vector3(0, 0, 80);
                    // If west, direction = 4; transform.eulerAngles = new Vector3(0, 0, 90);
                    if (west) direction = 4; transform.eulerAngles = new Vector3(0, 0, 90);
                }
                spotReached = false; // If steps = 0, add script, move
                // Move in direction
                // Count steps until in next space
                if (Globals.ready) moveF.Move(direction); steps++; if (steps >= (20/speed)) steps = 0; spotReached = true;
                else // Let go of button
                if (spotReached) // If go of button, but still moving
                {
                    if (Globals.ready) moveF.Move(direction); steps++; if (steps >= (20/speed)) steps = 0; spotReached = true;
                }
            }
        }
    }
}
```
WHAT IS A GAME ENGINE?

- Run-time system
  - Low-level architecture
    - 3D system
    - Physics system
    - GUI system
  - Networking system
- High-level architecture
  - Game objects
  - Game mechanics
- Toolsets
  - Level editor
  - Character and animation editor
  - Material creator
- Subsystems
  - Run-time object model
  - Real-time object model updating
  - Messaging and event handling
  - Scripting
  - Level management and streaming
WHAT ARE GAME OBJECTS?

- Anything that has a representation in the game world
  - Characters, props, vehicles, projectiles, cameras, trigger volumes, lights, etc.
- Created/modified by world editor tools
  - Object model presented to designers
- Managed at runtime in the runtime engine
  - Object model efficiently implemented for players
- What is an architecture model that can accomplish all this?
RUN-TIME OBJECT MODEL ARCHITECTURES

- Object-centric
  - Objects implemented as class instances
  - Object’s attributes and behaviors encapsulated within the class(es)

- Property-centric
  - Object attributes are implemented as data tables, one per attribute
  - Game objects are just IDs of some kind
  - Properties of an object are distributed across tables associated with engine systems (keyed by the object’s id)
OBJECT-CENTRIC ARCHITECTURES

- Natural taxonomy of game object types
- Common, generic functionality at root
- Specific game object types at the leaves

Hypothetical PacMan Class Hierarchy

GameObject
  └── MovableObject
      └── DrawableObject
          └── PacMan
          └── Ghost
          └── Pellet
              └── PowerPellet
MONOLITHIC CLASS HIERARCHIES

- Very intuitive for small simple cases
- Tend to grow ever wider and deeper
- Virtually all classes in the game inherit from a common base class

Part of object class hierarchy from Unreal Tournament 2004

Actor
  - Brush
  - Controller
    - AIController
    - PlayerController
  - Info
    - GameInfo
  - Pawn
    - Vehicle
    - UnrealPawn
    - RedeemerWarhead
    - Scout

Light
  - Inventory
    - Ammunition
    - Powerups
    - Weapon
  - HUD
  - Pickup
    - Ammo
    - ArmorPickup
    - WeaponPickup
PROBLEMS WITH MONOLITHIC HIERARCHIES

- Hard to understand, maintain, and modify classes
  - Need to understand a lot of parent classes
- Hard to describe multidimensional taxonomies
  - How to classify objects along more than one axis?
    - e.g. how would you include an amphibious vehicle?
USE MULTIPLE INHERITANCE?

- NOOOO!!!!!!
- There’s a reason languages like Java don’t have it
- Derived classes often end up with multiple copies of base class members
- Compiler cannot resolve ambiguities
MIX-IN CLASSES

Mix-in classes (stand alone classes with no base class) can solve deadly diamond problem

Similar to an interface with implemented methods (traits) except mix-ins have state (e.g. can store properties)

Supported natively in many languages

Can be implemented in C++ using templates
Mix-in classes (stand alone classes with no base class) can solve deadly diamond problem.

Another approach is to use composition or aggregation in addition to inheritance.
OBSERVATIONS ABOUT INHERITANCE

- Not every set of relationships can be described in a directed acyclic graph
- Class hierarchies are hard to change
- Functionality drifts upwards
- Specializations pay the memory cost of the functionality in siblings and cousins
OTHER ISSUES WITH INHERITANCE

- Consider a simple generic GameObject specialized to add properties for full blown physical simulation

- What if you want to use physical simulation on objects that don’t use skeletal animation?
SHOULD YOU EVER USE INHERITANCE?

- Very hotly debated topic – particularly in game engine development
- Yes, it is possible to throw it out entirely, but you trade one set of issues for another, so not always the right answer
- Inheritance works well for logically reasoning about game objects and taxonomies
- Shallow inheritance with composition can be both performant and easier to maintain
One “hub” object contains pointers to instances of various service class instances as needed (e.g. composition).
COMPONENT-BASED EXAMPLE (1/2)

class GameObject {

protected:

    // My transform (position, rotation, scale)
    Transform m_transform;

    // Standard components
    MeshInstance* m_pMeshInst;
    AnimationController* m_pAnimController;
    RigidBody* m_pRigidBody

public:

    GameObject() {
        ...
    ...
}
GameObject() {

    // Assume no components. Derived classes will override

    m_pMeshInst = nullptr;

    m_pAnimController = nullptr;

    m_pRigidBody = nullptr;

}

~GameObject() {

    // Automatically delete any components

    delete m_pMeshInst;

    delete m_pAnimController;

    delete m_pRigidBody;

};
class Vehicle : public GameObject {

protected:

    // Add some more components specific to vehicles
    Chassis* m_pChassis;

    Engine* m_pEngine;

    // …

public:

    Vehicle() { 
        // Construct standard GameObject components
        m_pMeshInst = new MeshInstance();

        m_pRigidBody = new RigidBody();

        m_pAnimController = new AnimationController(*m_pMeshInst);
Vehicle() {
    // ...
    // Construct vehicle-specific components
    m_pChassis = new Chassis(*this, *m_pAnimController);
    m_pEngine = new Engine(*this);
}

~Vehicle() {
    // Only need to destroy vehicle-specific components
    delete m_pChassis;
    delete m_pEngine;
}
**USING COMPOSITION**

- "Hub" class owns its components and manages their lifetimes (i.e. creates and destroys them)

- Naive component creation:
  - The GameObject class has pointers to all possible components, initialized to nullptr
  - Only creates needed components for a given derived class
  - Destructor cleans up all possible components for convenience
  - All optional add-on features for derived classes are in component classes
MORE FLEXIBLE (AND COMPLEX) ALTERNATIVE

- Root GameObject contains a list of generic components
- Derive specific components from the component base class
- Allows arbitrary number of instances and types of components
EXAMPLE: UE4 AND UACTORCOMPONENTS
AGGREGATION VERSUS COMPOSITION

- In composition, child life cycle is managed by parent object
  - Child cannot exist outside of parent object
- In aggregation, child life cycle exists outside of parent object
  - Child exists outside of parent object but can be associated with parent object
EXAMPLE: GODOT AND SCENES

https://github.com/razcore-art/godot-open-rpg
DO WE NEED A GAME OBJECT?

- We can reduce GameObject to an id and a list of its components
  - Id is unique to each game object (entity)
  - Components of a game object describe properties of that game object
  - Systems run on all their associated components to update game object
- Basis of Entity-Component-System style architecture
PROPERTY-CENTRIC ARCHITECTURES

- Think in terms of properties (attributes) of objects rather than in terms of objects.
- For each property, build a table containing that property’s values keyed by object ID.
- Now you get something like a relational database.
  - Each property is like a column in a database table whose primary key is the object ID.
- Object’s behavior defined by its property types and scripts.
  - Scripts have a script ID in object’s properties and can be the target of messages.