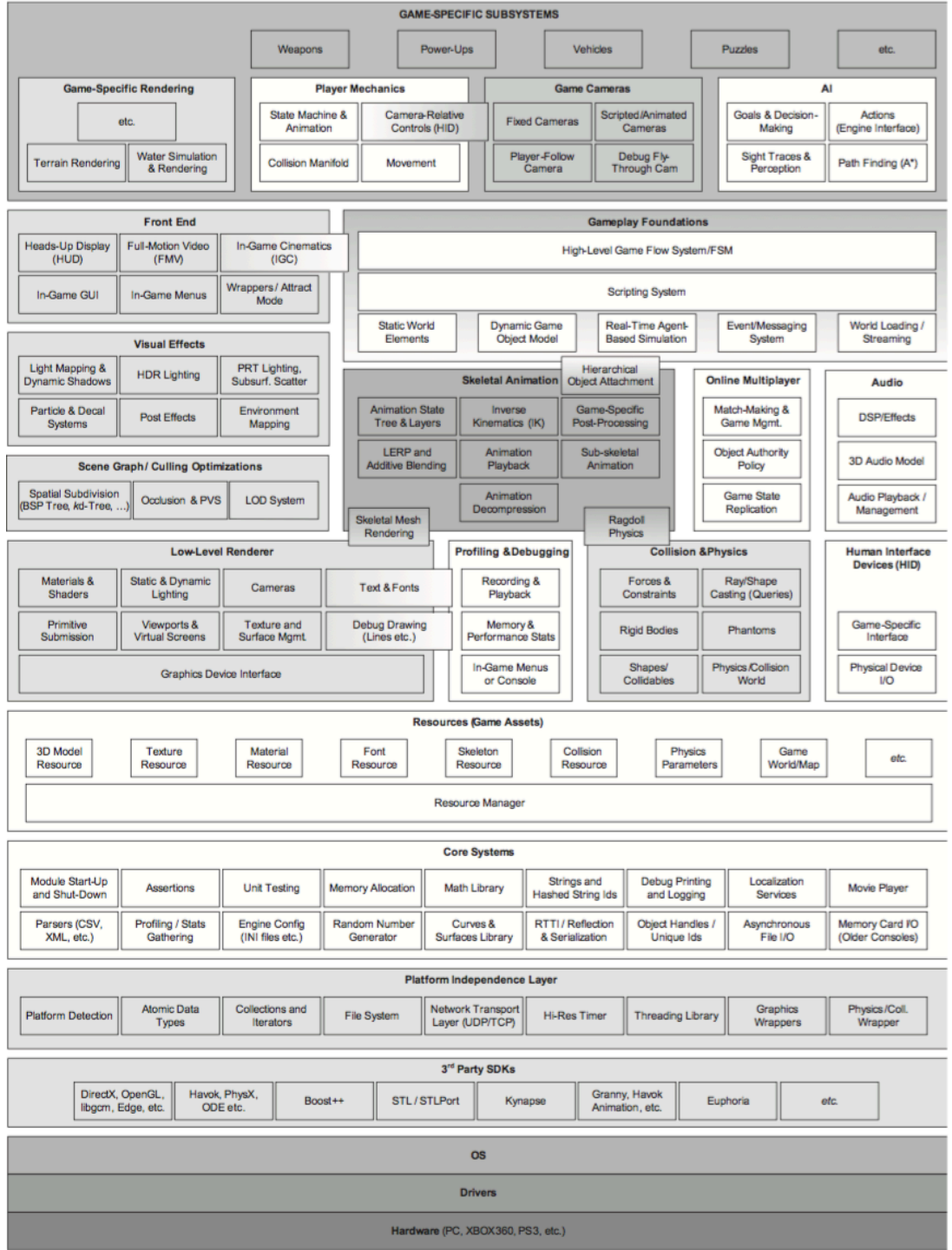
G54GAM Games

Building Games

Software Patterns

Architectural Design Patterns

- A number of individual engine subsystems
- A collection of game objects
  - Forming the “game state”
    - Each has attributes, behaviours
  - E.g. Player avatar
    - Needs to be controlled, moved around
    - Collide with other objects
    - Drawn on the screen
- *Software design patterns* help to organize the engine around predefined concepts
  - Accepted as optimal for their efficiency, elegance and robustness
  - A template for code organisation
  - E.g. Singleton
    - Renderer, ResourceManager, Debug, Input


GAME-SPECIFIC SUBSYSTEMS
Weapons
Power-Ups
Vehicles
Puzzles
etc.
Game-Specific Rendering
etc.
Terrain Rendering
Water Simulation & Rendering
Player Mechanics
State Machine & Animation
Camera-Relative Controls (HID)
Collision Manifold
Movement
Game Cameras
Fixed Cameras
Scripted/Animated Cameras
Player-Follow Camera
Debug Fly-Through Cam
AI
Goals & Decision-Making
Actions (Engine Interface)
Sight Traces & Perception
Path Finding (A*)
Front End
Heads-Up Display (HUD)
Full-Motion Video (FMV)
In-Game Cinematics (IGC)
In-Game GUI
In-Game Menus
Wrappers / Attract Mode
Gameplay Foundations
High-Level Game Flow System/FSM
Scripting System
Static World Elements
Dynamic Game Object Model
Real-Time Agent-Based Simulation
Event/Messaging System
World Loading / Streaming
Visual Effects
Light Mapping & Dynamic Shadows
HDR Lighting
PRT Lighting, Subsurf. Scatter
Particle & Decal Systems
Post Effects
Environment Mapping
Skeletal Animation
Hierarchical Object Attachment
Animation State Tree & Layers
Inverse Kinematics (IK)
Game-Specific Post-Processing
LERP and Additive Blending
Animation Playback
Sub-skeletal Animation
Animation Decompression
Skeletal Mesh Rendering
Ragdoll Physics
Online Multiplayer
Match-Making & Game Mgmt.
Object Authority Policy
Game State Replication
Audio
DSP/Effects
3D Audio Model
Audio Playback / Management
Scene Graph / Culling Optimizations
Spatial Subdivision (BSP Tree, kd-Tree, ...)
Occlusion & PVS
LOD System
Low-Level Renderer
Materials & Shaders
Static & Dynamic Lighting
Cameras
Text & Fonts
Primitive Submission
Viewports & Virtual Screens
Texture and Surface Mgmt.
Debug Drawing (Lines etc.)
Graphics Device Interface
Profiling & Debugging
Recording & Playback
Memory & Performance Stats
In-Game Menus or Console
Collision & Physics
Forces & Constraints
Ray/Shape Casting (Queries)
Rigid Bodies
Phantoms
Shapes/ Collidables
Physics/Collision World
Human Interface Devices (HID)
Game-Specific Interface
Physical Device I/O
Resources (Game Assets)
3D Model Resource
Texture Resource
Material Resource
Font Resource
Skeleton Resource
Collision Resource
Physics Parameters
Game World/Map
etc.
Resource Manager
Core Systems
Module Start-Up and Shut-Down
Assertions
Unit Testing
Memory Allocation
Math Library
Strings and Hashed String Ids
Debug Printing and Logging
Localization Services
Movie Player
Parsers (CSV, XML, etc.)
Profiling / Stats Gathering
Engine Config (INI files etc.)
Random Number Generator
Curves & Surfaces Library
RTTI / Reflection & Serialization
Object Handles / Unique Ids
Asynchronous File I/O
Memory Card I/O (Older Consoles)
Platform Independence Layer
Platform Detection
Atomic Data Types
Collections and Iterators
File System
Network Transport Layer (UDP/TCP)
Hi-Res Timer
Threading Library
Graphics Wrappers
Physics/Coll. Wrapper
3rd Party SDKs
DirectX, OpenGL, libgcm, Edge, etc.
Havok, PhysX, ODE etc.
Boost++
STL / STLPort
Kynapse
Granny, Havok Animation, etc.
Euphoria
etc.
OS
Drivers
Hardware (PC, XBOX360, PS3, etc.)

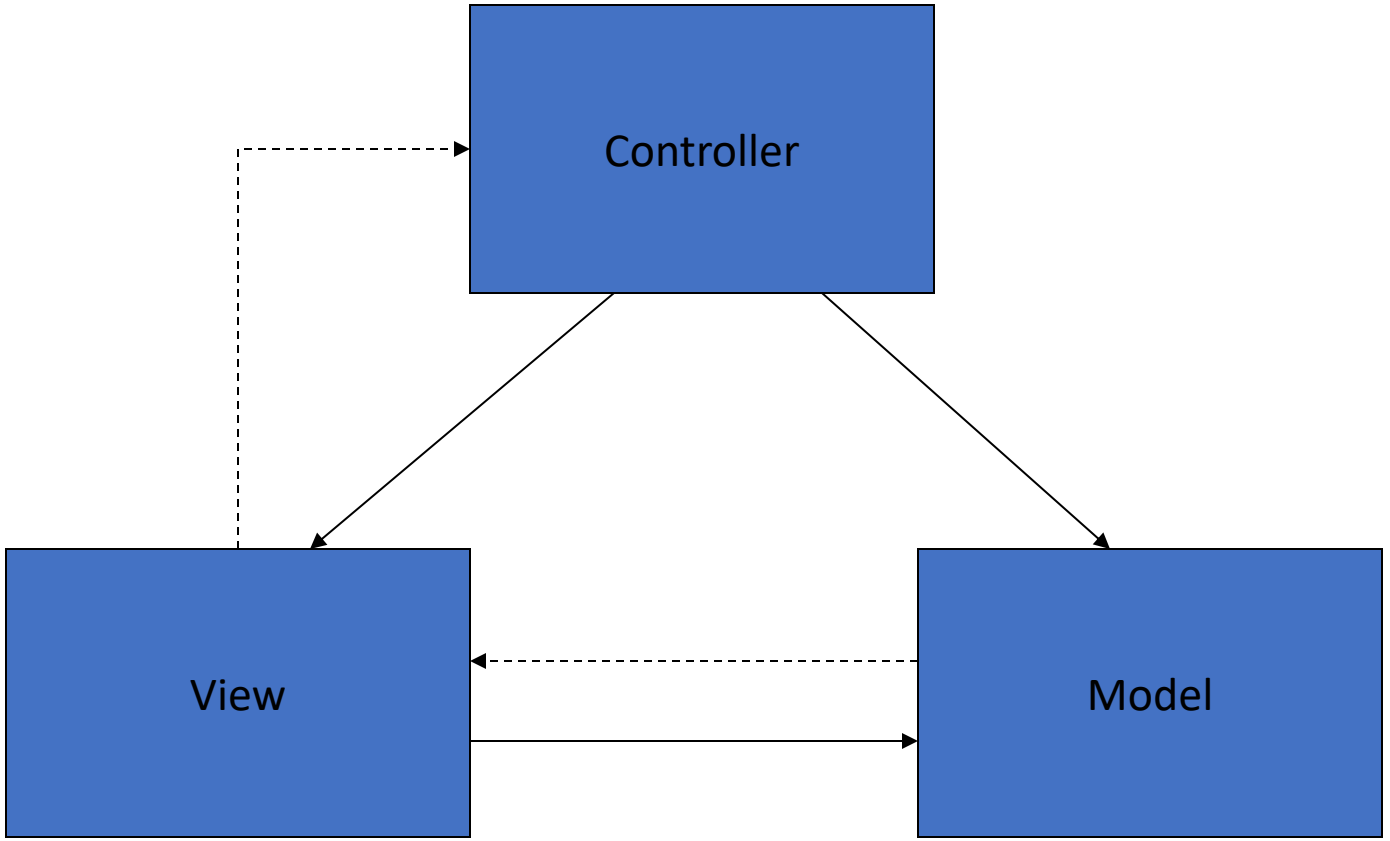
Subsystems as Singletons

```
public class PhysicsSingleton
{
        private static final PhysicsSingleton instance = new
                                                PhysicsSingleton(); // immutable constant

        // private constructor prevents user from instantiating
        private PhysicsSingleton()
        {
                // Initialize all fields for instance
        }

        public static PhysicsSingleton getInstance()
        {
                return instance; // provide access instead of via a global
        }
}
```

Model-View-Controller

- An architectural pattern
  - The same idea as a software pattern
  - Applies to complete program
- Used to isolate logic from user-interface
- Model
  - The information of the application
- View
  - The user interface and display of information
- Controller
  - Manages the communication of information and manipulation of the model

Updates model in response to events
Updates view with model changes

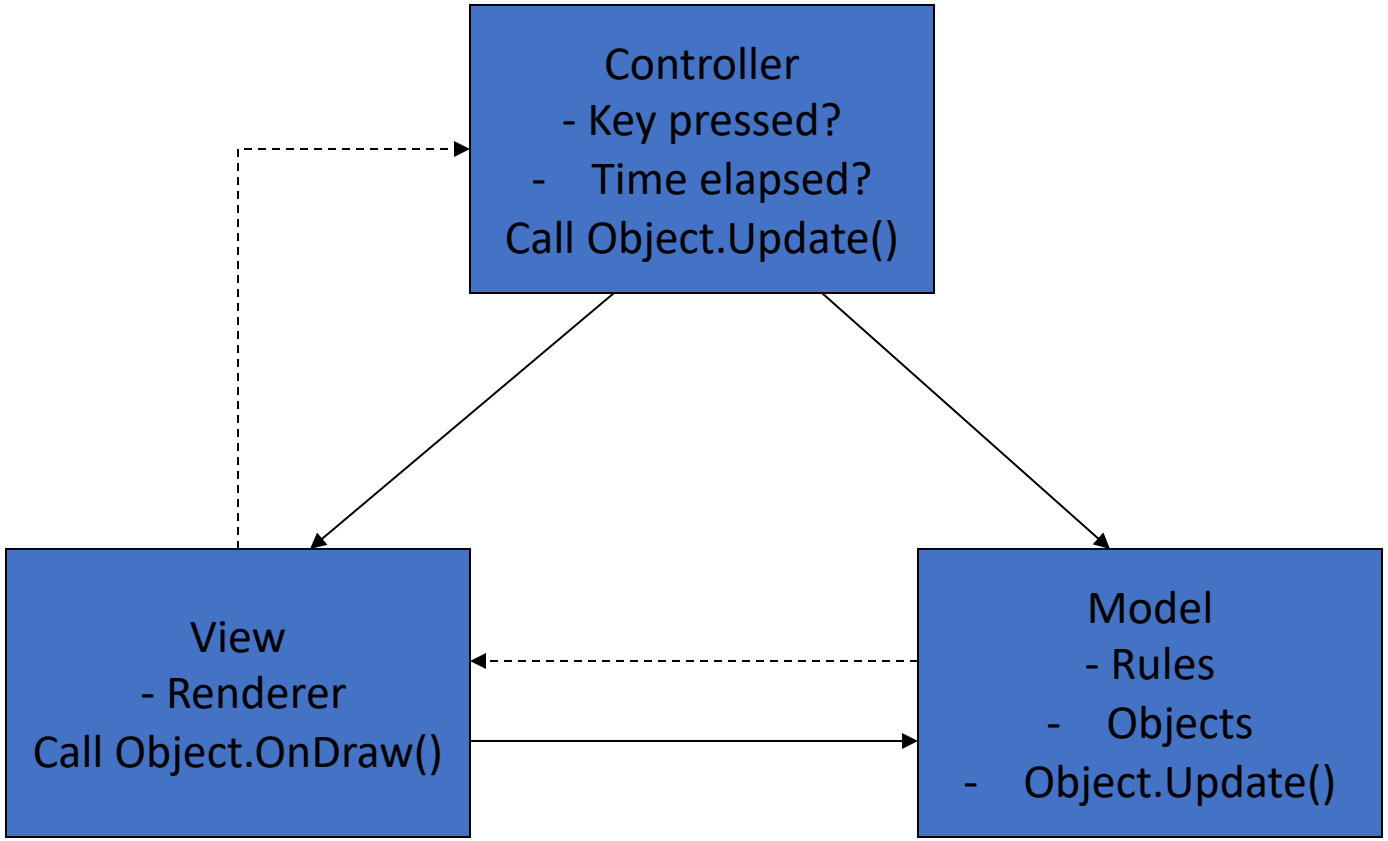


Displays model to the user
Provides interface for the controller

Defines the program data
Responds to the controller requests

Game MVC Architecture

- Model
  - The state of every game object and entity
    - Data pertaining to objects
    - Limit access (getter / setter, update)
  - The world simulation
    - Implements object logic
    - Complex actions on the model
      - E.g. “attack”, “collide”
  - Knows nothing about user input or display
- View
  - Renders the model to the screen
  - Uses the model to know where to draw everything
    - Draw the model from this camera position
- Controller
  - Process user input and call actions in the model
    - E.g. mouse, gamepad
  - Alters the game state
  - Traditional controllers are *lightweight*
- *Recall* User input -> Simulate Game -> Draw (game loop)

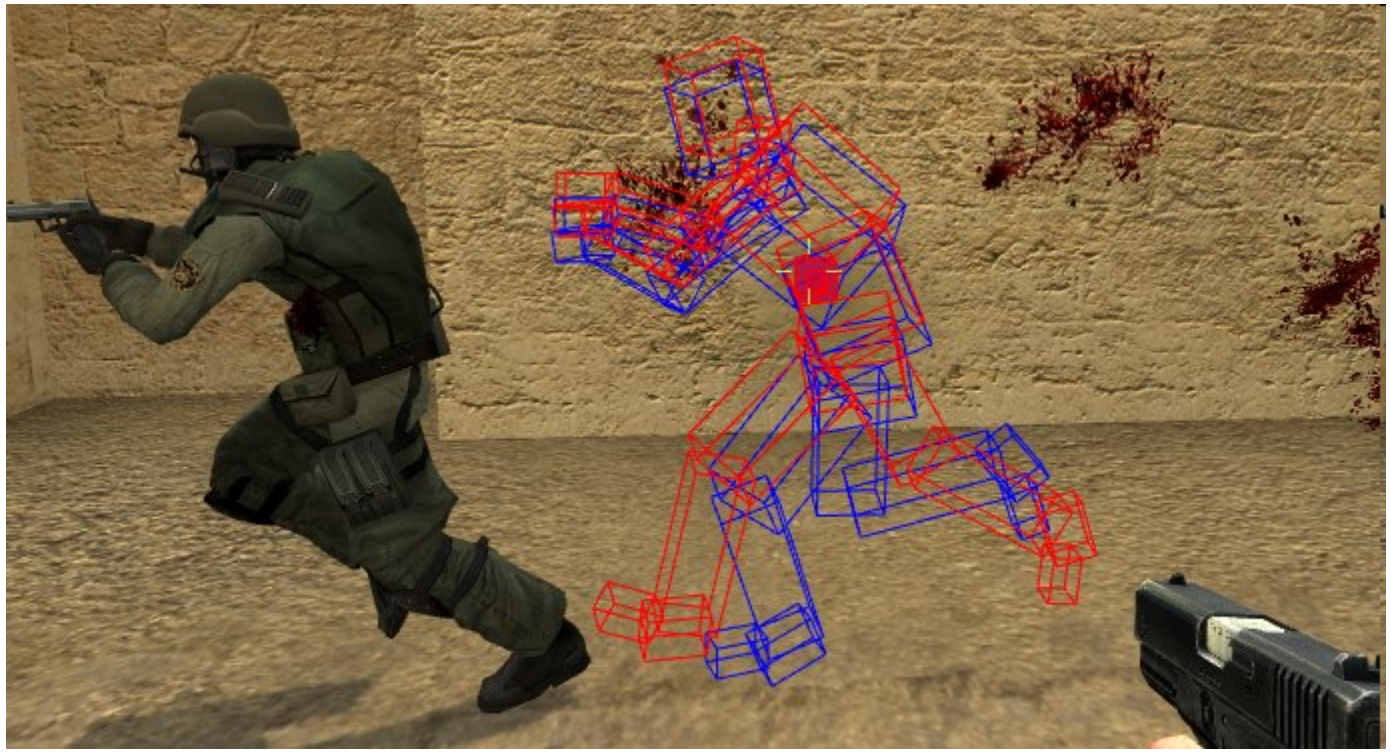

Controller
- Key pressed?
- Time elapsed?
Call Object.Update()
View
- Renderer
Call Object.OnDraw()
Model
- Rules
- Objects
- Object.Update()

FPS MVC Architecture

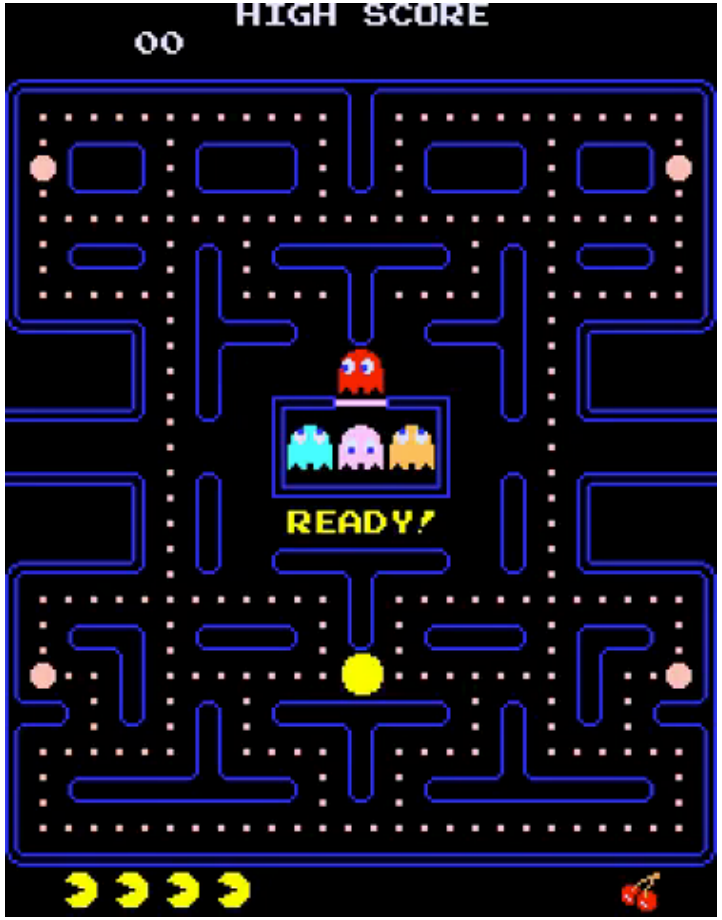
- Model
  - An *abstract* 3d environment
  - Positions and orientations change over time
- View
  - Render the 3d environment
  - Display complex avatars and animations
  - Fancy effects
- Controller
  - Tell the model that I want to move, shoot, jump
  - Tell the model that $1/30^{th}$ of a second has elapsed

FPS MVC Architecture

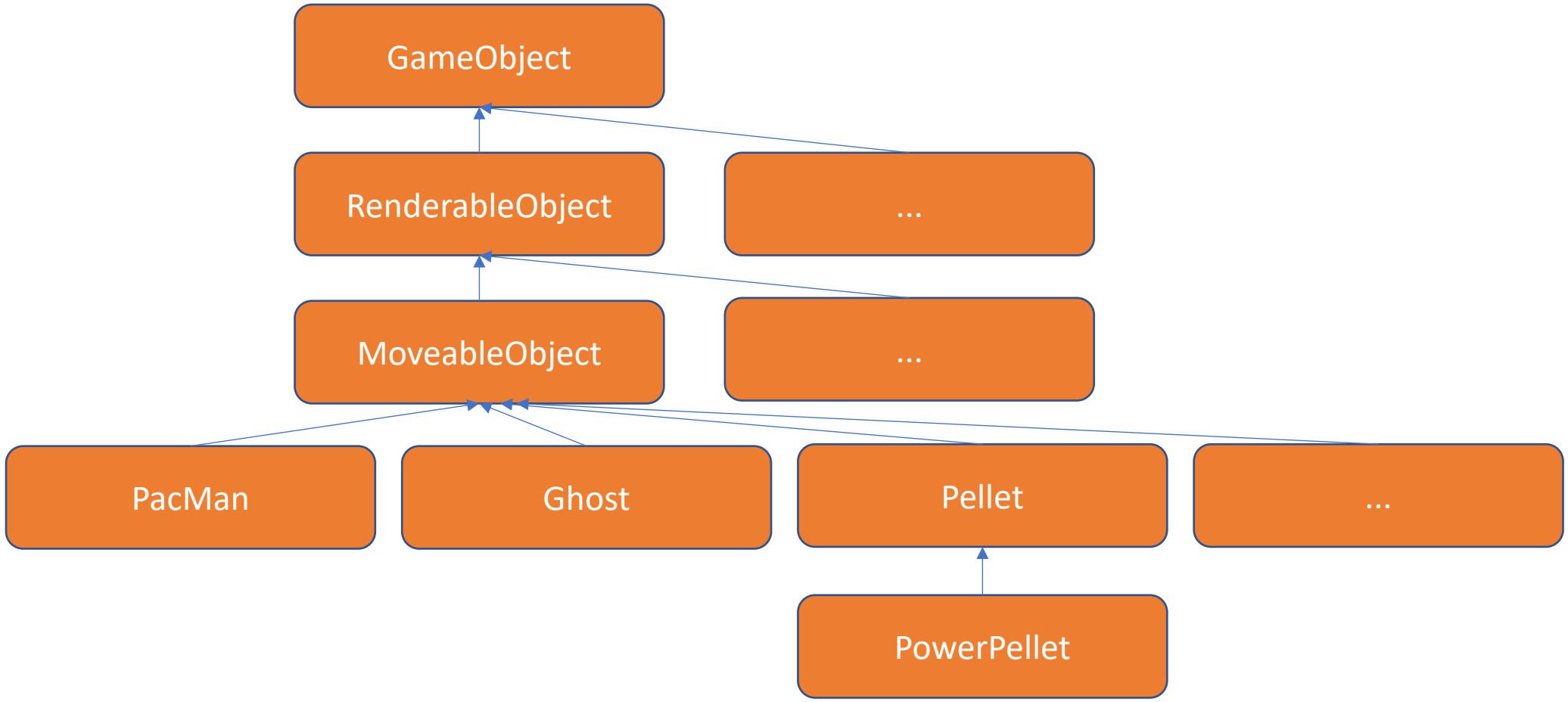
- Player
  - ID
  - Name
  - Transform
    - Position
    - Rotation
  - Hitboxes
  - Pistol
  - Health
  - Ammunition
  - Avatar resource ID



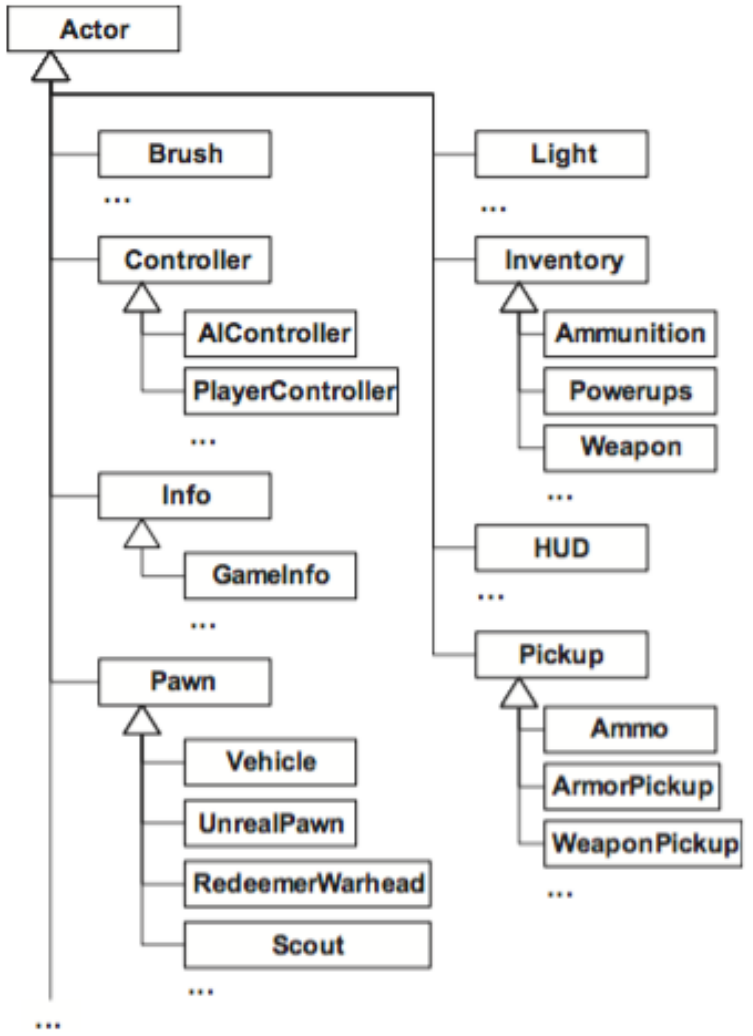
What is the class structure?



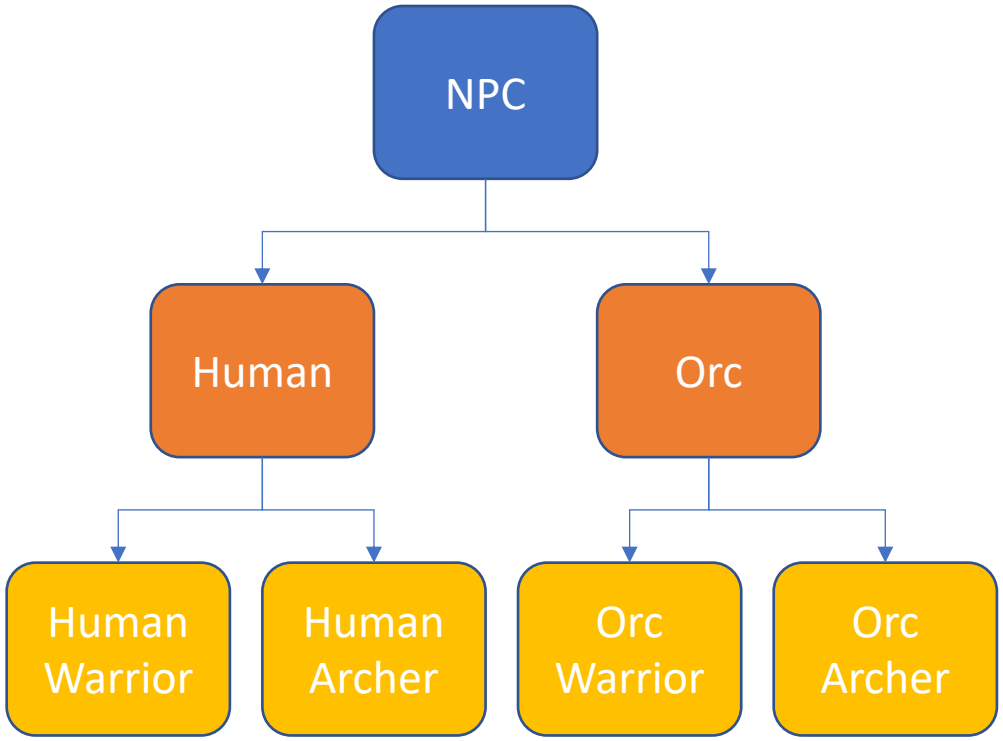
“Monolithic” Class Hierarchy



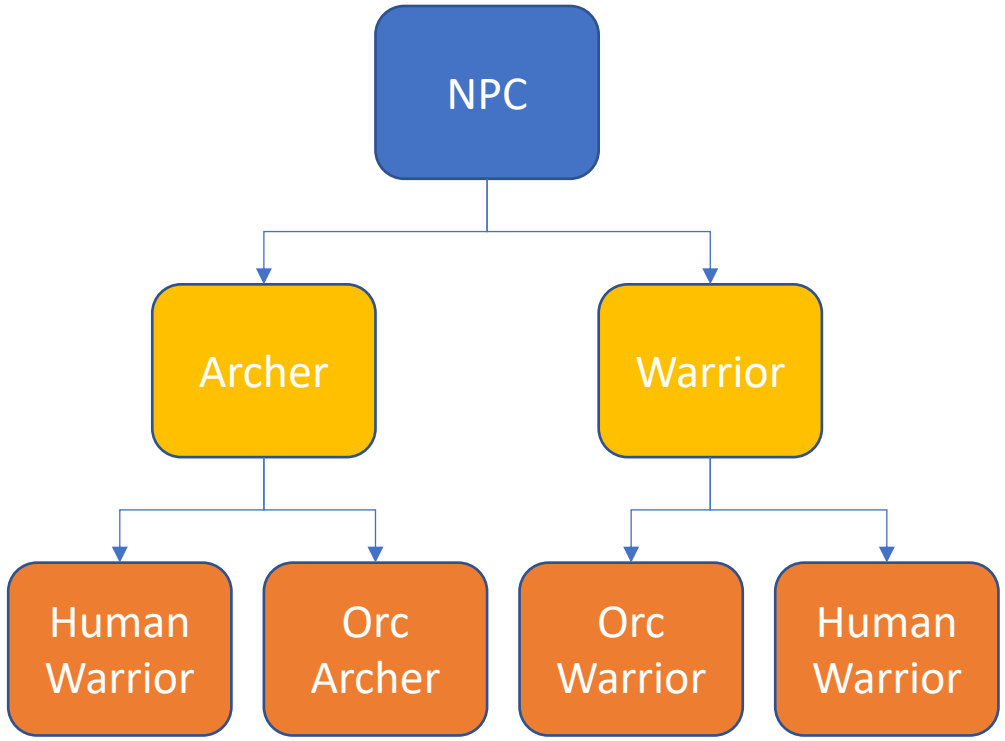
“Monolithic” Class Hierarchy



Describing Multidimensional Hierarchies



Redundant Behaviour



Redundant Behaviour

Issues with Games and OOP

- Object-oriented programming is *noun-centric*
  - Code must be organised into classes
  - Polymorphism *determines capability* via type
    - If it's an orc, it can do certain things
- OOP became popular with standard MVC pattern
  - Widget libraries are nouns implementing views
  - Data structures are all nouns
  - Controllers are not necessarily nouns, but are lightweight
- Games break this paradigm to some extent
  - View is animation (process) oriented, not widget oriented
  - Actions and capabilities are only loosely connected to entities / actors

Structuring the Game-Object-Model

- Object centric
  - Attributes and behaviours
    - Encapsulated in classes
  - Game world is a collection of game object instances
- Conventional OOP approach to extensibility
- A class with some base functionality
  - Want to add additional functionality
  - Subclass original class
    - E.g. extending GUI widgets
- Games have many classes
  - Each game entity is different
  - Needs its own functionality
  - Want to avoid redundancies
    - Makes code hard to change
    - Common source of bugs
- Property-centric
  - Game object as ID
  - Lookup-tables of properties and ids
    - If an object has the health property then it can be damaged

Revised MVC

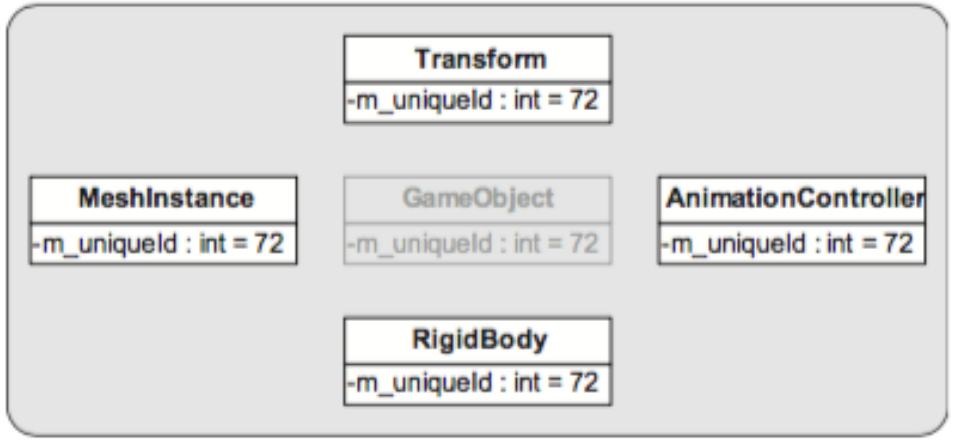
- Model
  - Store and retrieve **object data**
  - Lightweight
  - Limit access (getter / setter)
  - Only affects *this* object
- Controller
  - Heavyweight
  - Process **game actions**
    - Determine from input of AI
    - Find *all* objects effected
  - Process **interactions**
    - Look at current game state
    - Look for triggering events
    - Apply interaction outcome
- Doesn’t completely solve the problem

Issues with Games and OOP

- Classes and Types are Nouns
  - Method calls are sentences
    - Subject.verb(object)
    - Subject.verb()
  - Classes related by *is-a*
    - This object *is-a* monster
- Actions are Verbs (subsystem perspective)
  - Often just a simple function
    - Damage(object)
    - Collide(object1, object2)
  - Relates to objects via *can-it*
    - Orc *can-it* run away
    - Not necessarily tied to class
- Incorporate property-centric perspective?
  - Ideally capabilities over properties
  - Extend capabilities without necessarily changing *type*

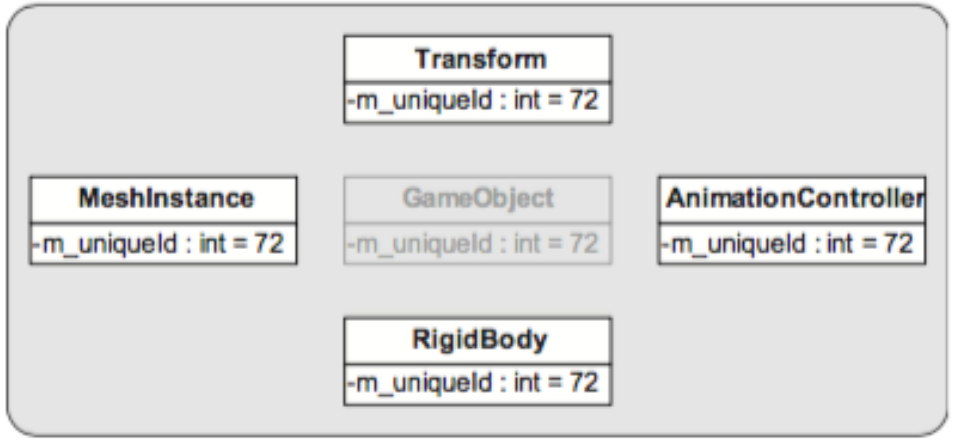
Pure Component based approach

```
struct AllGameObjects
{
        U32 m_aUniqueId [MAX_GAME_OBJECTS];
        Vector m_aPos [MAX_GAME_OBJECTS];
        Quaternion m_aRot [MAX_GAME_OBJECTS];
        float m_aHealth [MAX_GAME_OBJECTS];
        // ...
}
AllGameObjects g_allGameObjects;
```

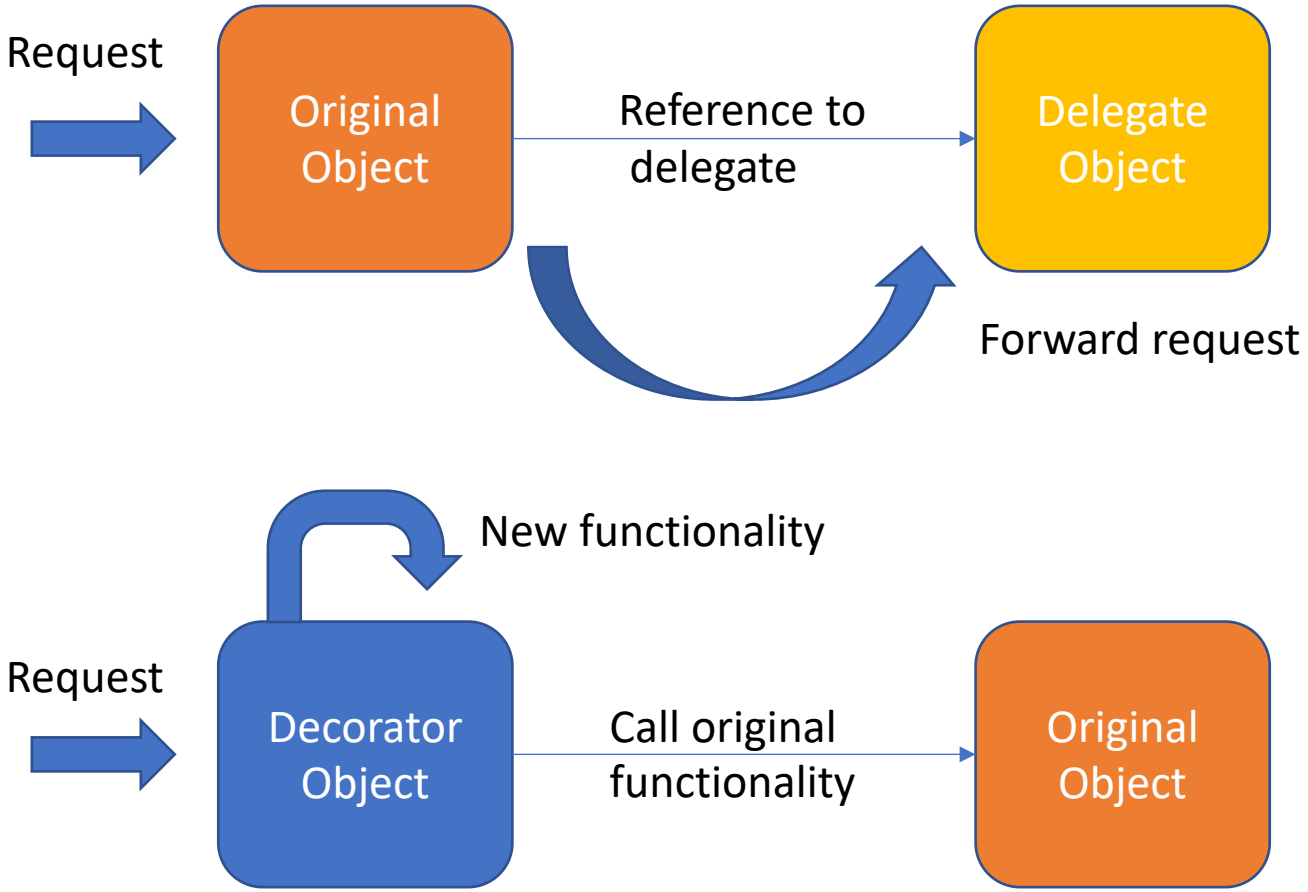


Pure Component based approach

```
struct AllGameObjects
{
        U32 m_aUniqueId [MAX_GAME_OBJECTS];
        Vector m_aPos [MAX_GAME_OBJECTS];
        Quaternion m_aRot [MAX_GAME_OBJECTS];
        float m_aHealth [MAX_GAME_OBJECTS];
        // ...
}
AllGameObjects g_allGameObjects;
```



Delegation and Decorator Patterns



Delegation Pattern

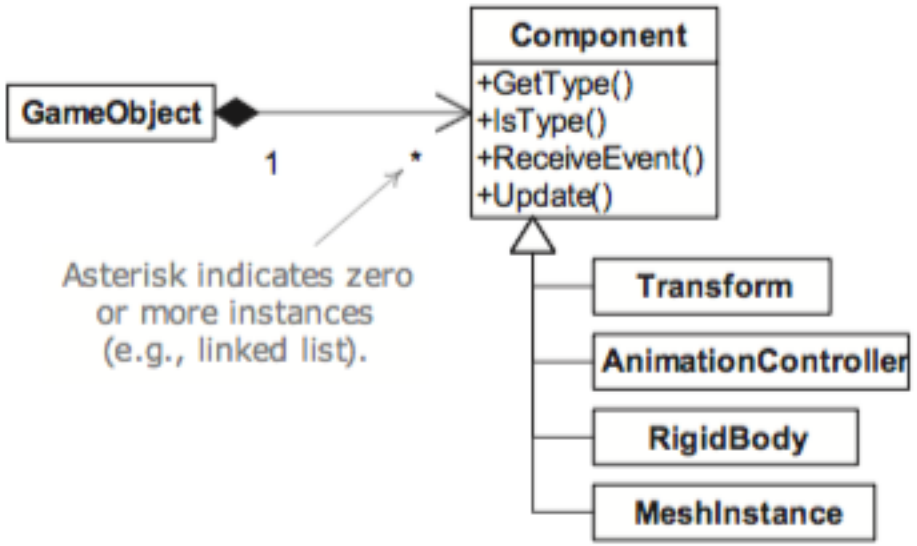
```
public class SortableArray extends ArrayList {

        private Sorter sorter = new MergeSorter();

        public void setSorter(Sorter s) { sorter = s;}
        public void sort() {
                Object[] list = toArray();
                sorter.sort(list);
                clear();
                for (o:list) { add(o); }
        }
}

public interface Sorter {
        public void sort(Object[] list)
}
```

Delegation Pattern

```
public class SortableArray extends ArrayList {

        private Sorter sorter = ~~new MergeSorter();~~
                              = new QuickSorter();
        public void setSorter(Sorter s) { sorter = s;}
        public void sort() {
                Object[] list = toArray();
                sorter.sort(list);
                clear();
                for (o:list) { add(o); }
        }
}

public interface Sorter {
        public void sort(Object[] list)
}
```

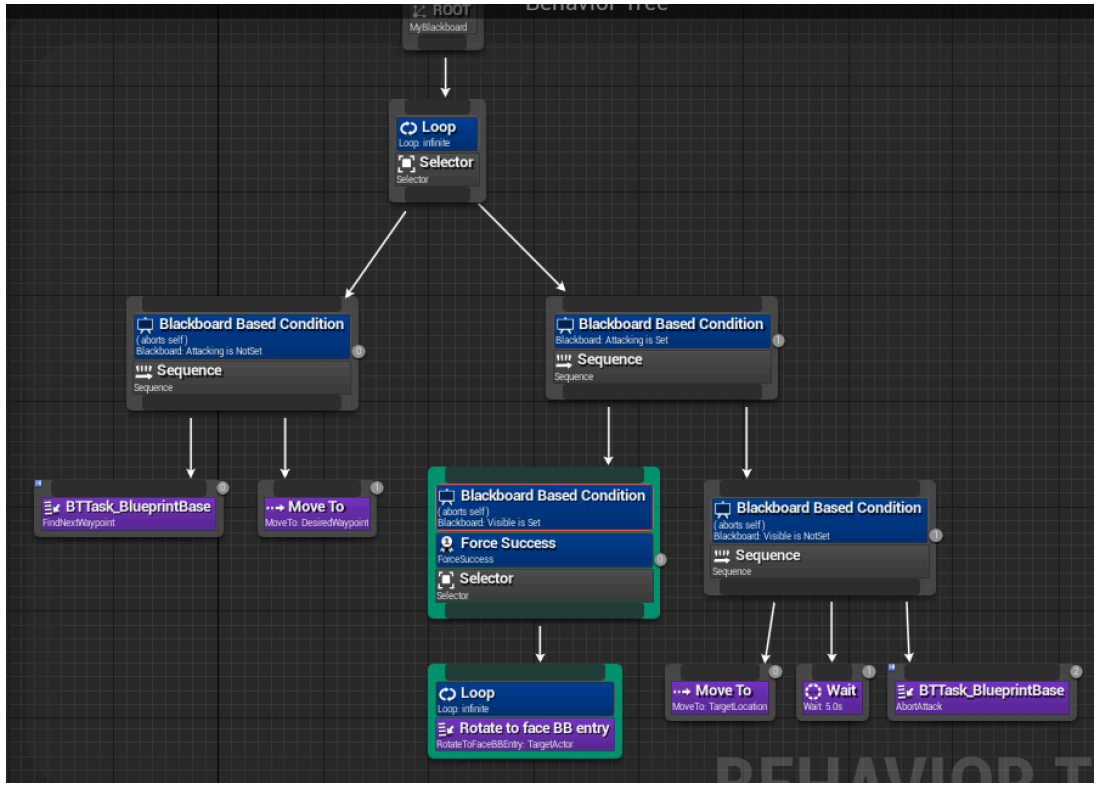
Delegate

- Delegation
  - Applies to *original object*
    - *We* design the object class
    - Requests made through object
  - Modular solution
    - Each method can have its own delegate implementation
  - Limited to classes that *we* make



Decorator

- Given the original object
  - Requests made *through* decorator
  - Adds functionality without necessarily knowing what the original object does
- *Monolithic* solution
  - Decorator has all methods
  - Layer for more methods
    - e.g. Java I/O classes
      - InputStream
      - Reader
      - BufferedReader
- Works on *any* object/class
  - Even those that we haven't made ourselves
  - E.g. AI functionality

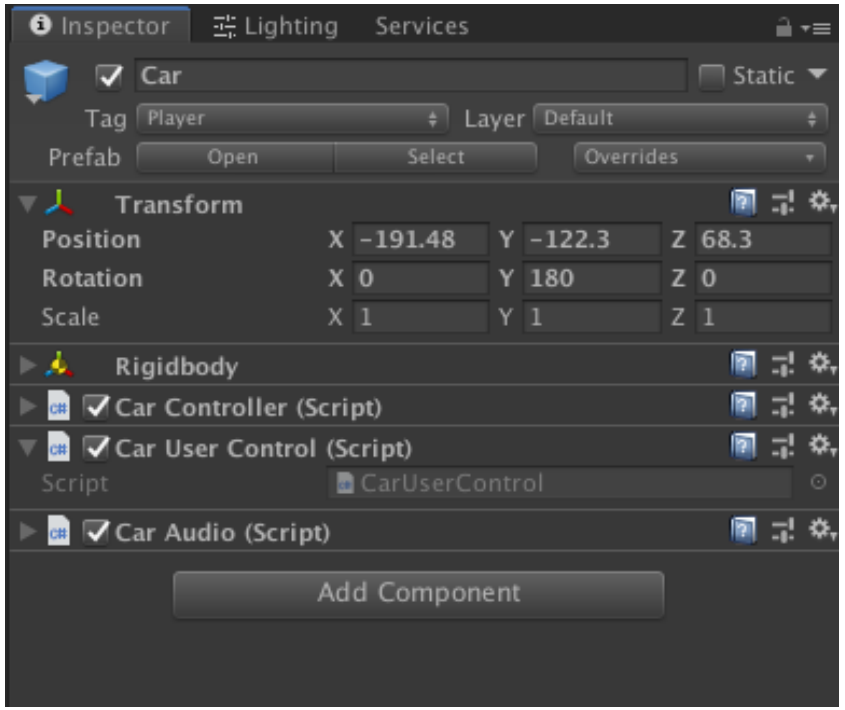


Partial Component based approach

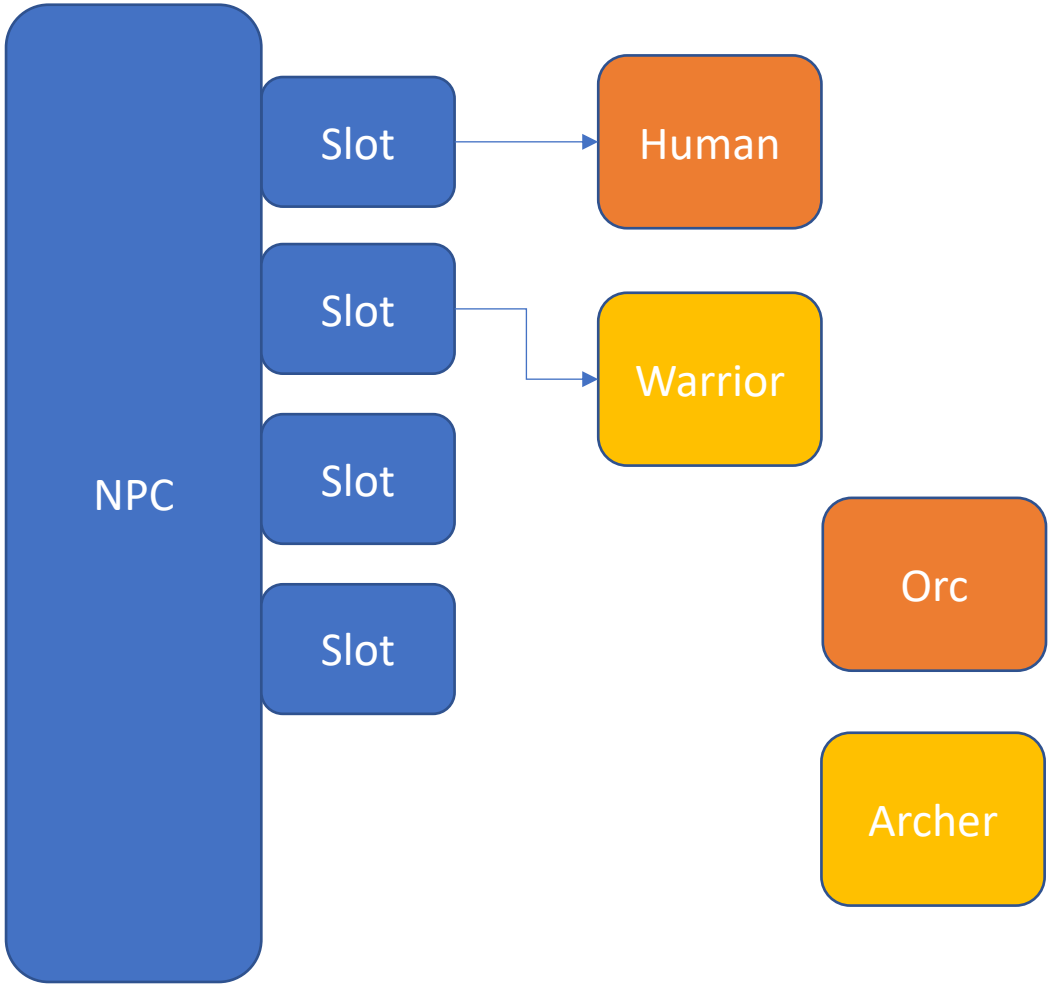
Partial Component based approach

- Entity
  - Needs both *is-it* and *can-it* approach
- Add a field storing a single delegate / collection of delegates as *roles*
  - A role is a set of *capabilities*
    - Class with very little data
    - A collection of methods
      - Things that the object will be able to do
  - Add to object as delegate
    - Object gains those methods
  - *Can-it* search object roles
    - Keep a table of all objects with X capability
    - Better than duck-typing (if orc instanceOf Orc)

Partial Component based approach




```
using UnityEngine.AI;

public class State_Patrol: IState
{
    EnemyController owner;
    NavMeshAgent agent;
    Waypoint waypoint;
```



What should the structure be?

MVC Revisited

- Model
  - Store / retrieve object data
  - Data may include delegates
  - Determines *is-a* properties
- Controller
  - Process interactions
    - Look at current game state
    - Look for triggering events
    - Apply interaction outcome
- Components relevant for both model and controller
  - Process game actions
    - Attached to an entity (model)
    - Use the model as context
    - Determines *can-it* properties for the controller

Summary

- Games naturally fit a specialised MVC pattern
  - Lightweight models
    - Aids with serialisation
      - Networking
        - Who needs to know about what to *transmit* the game
        - The smaller the amount of data the better
  - Heavyweight controllers for the game loop
- Design leads to unusual OOP
  - Subclass hierarchies are unmanageable
  - Component-based design to model actions

Reading

- Game Engine Architecture, Jason Gregory 2014, chapter 14