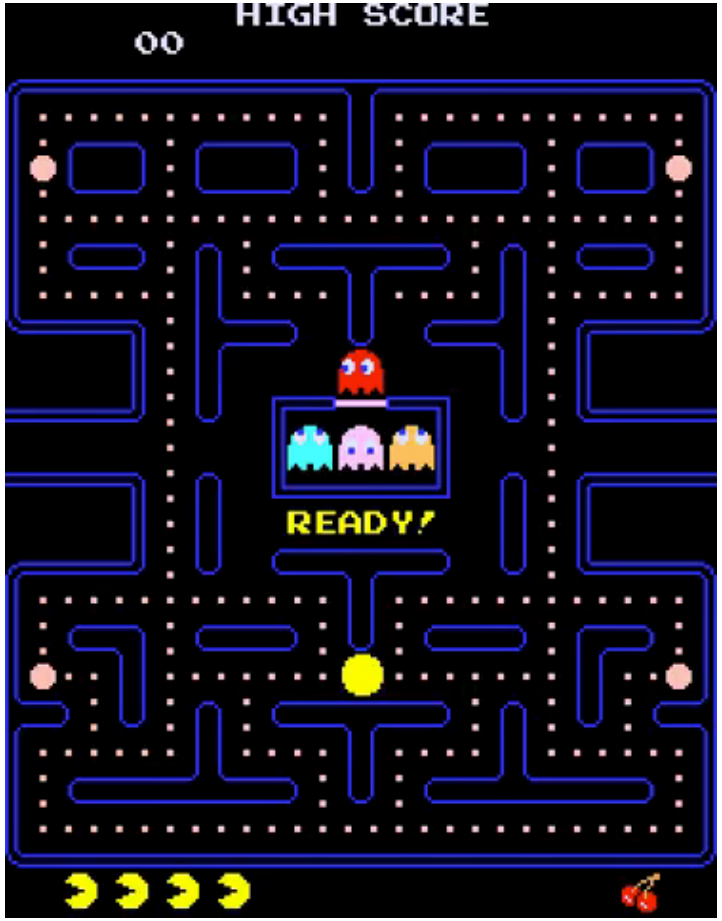
G54GAM Games

Building Games

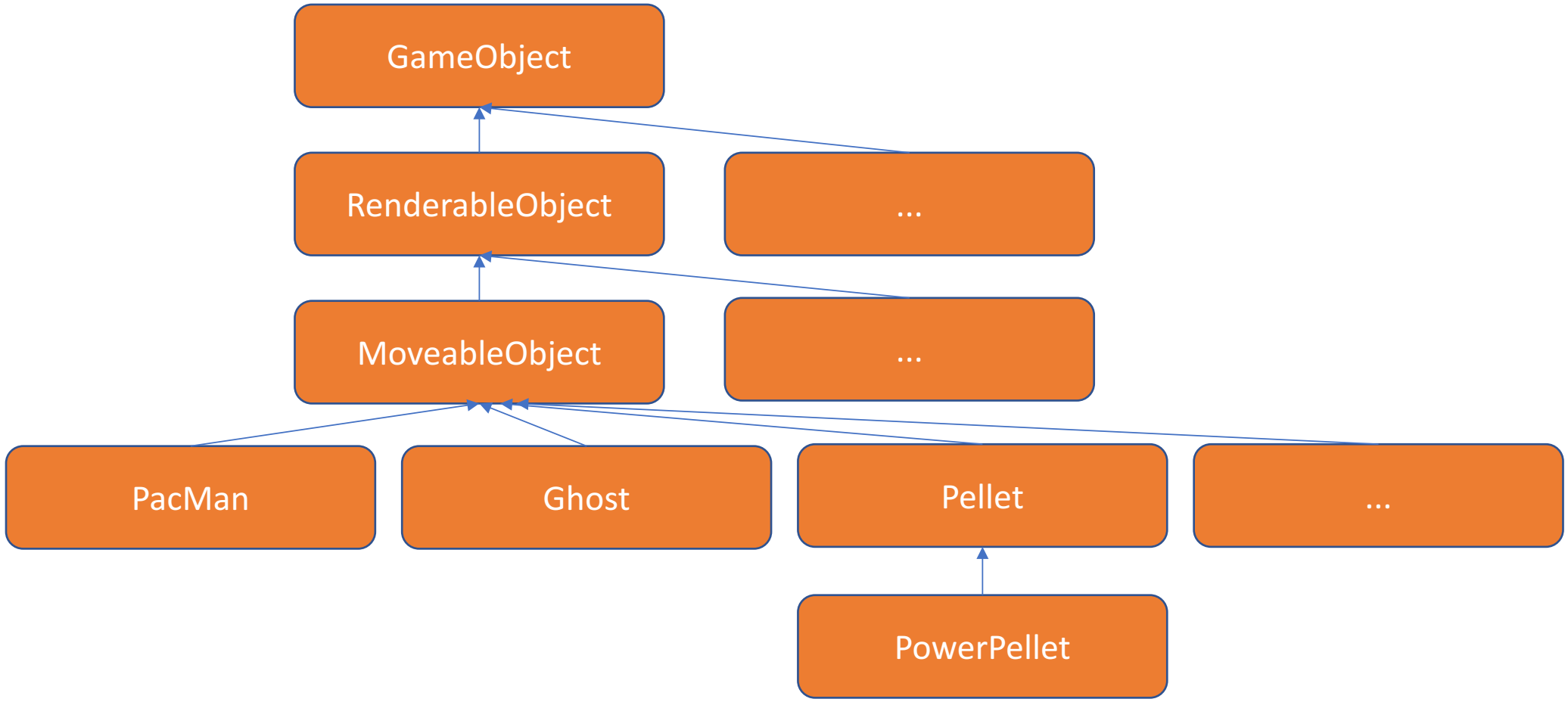
Software Patterns

Multiplayer

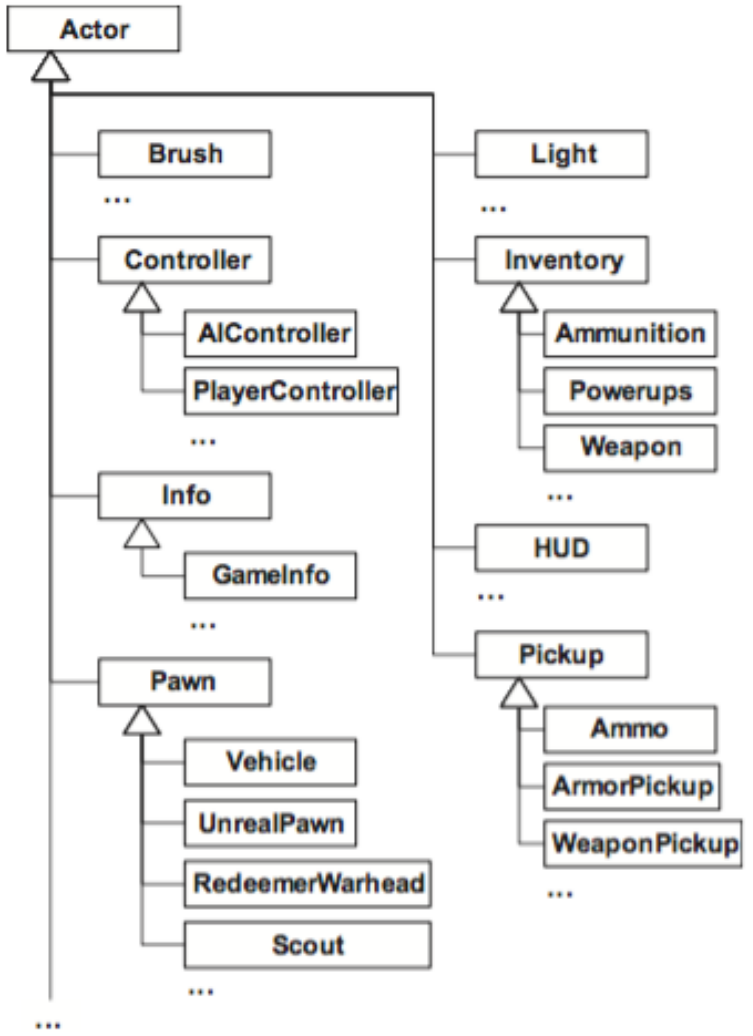
What is the class structure?



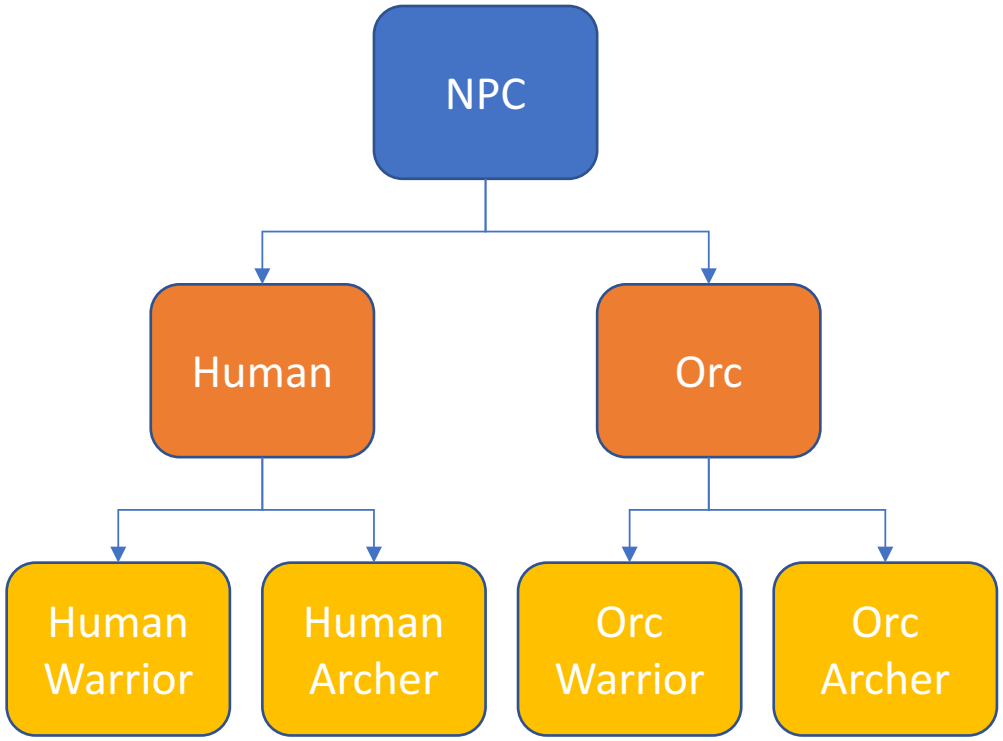
“Monolithic” Class Hierarchy



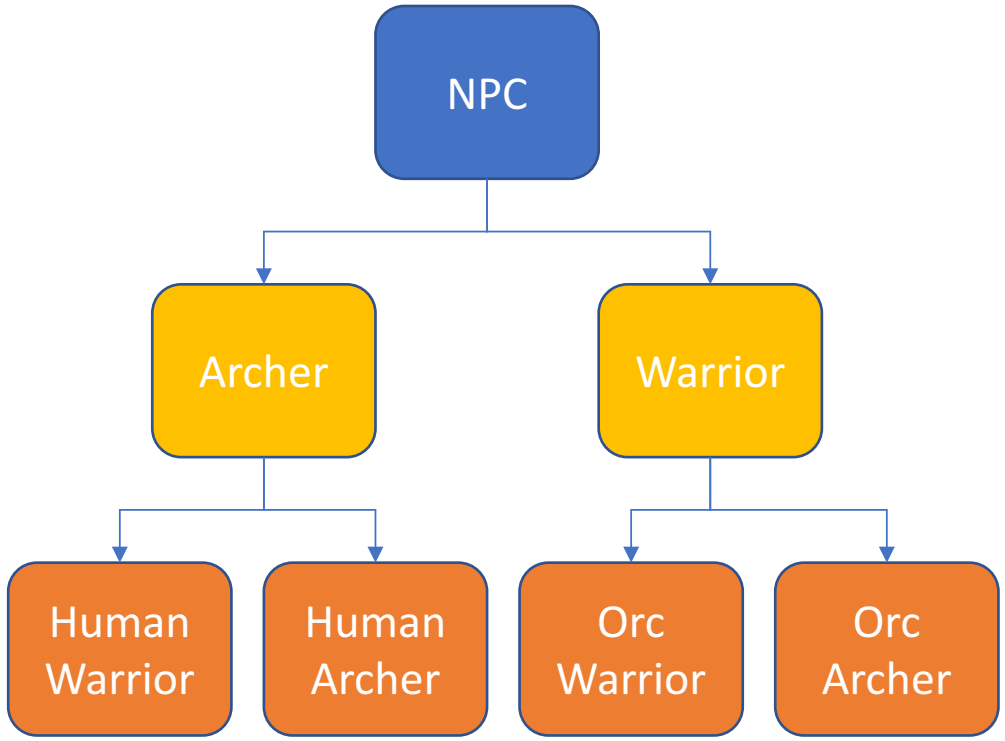
“Monolithic” Class Hierarchy



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

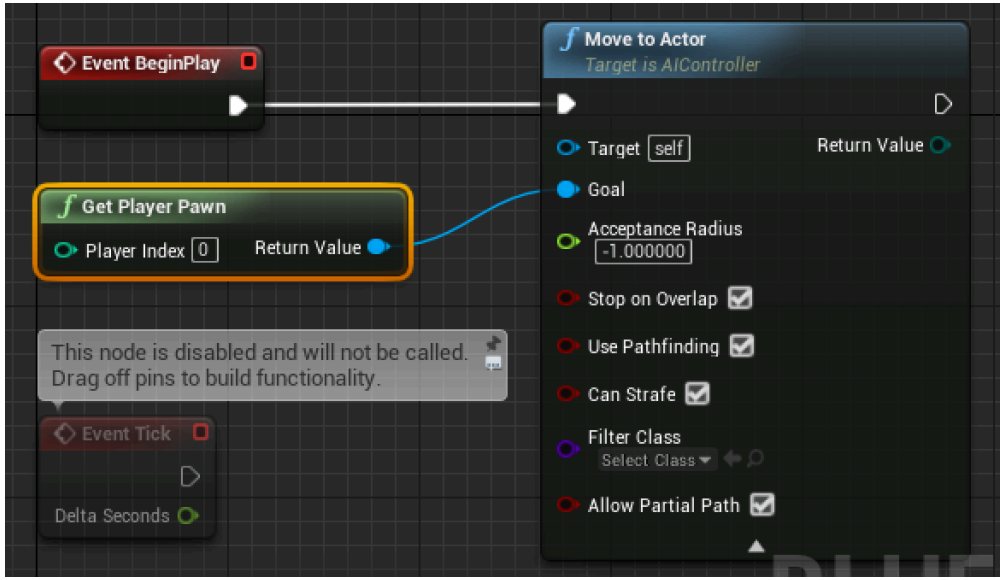
- Object centric
  - Attributes and behaviours
    - Encapsulated in classes
  - Game world is a collection of game object instances
- Conventional OOP approach to extensibility
- A class with some base functionality
  - Want to add additional functionality
  - Subclass original class
    - E.g. extending 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
  - 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

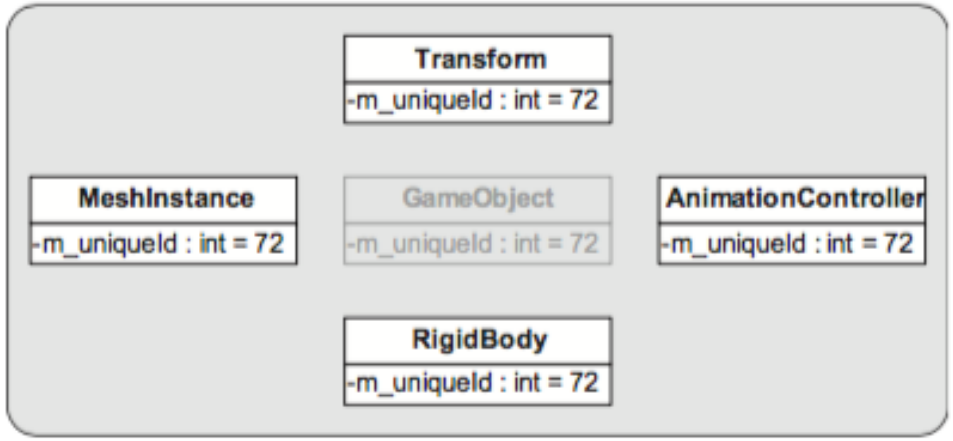
Issues with Games and OOP

- 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*

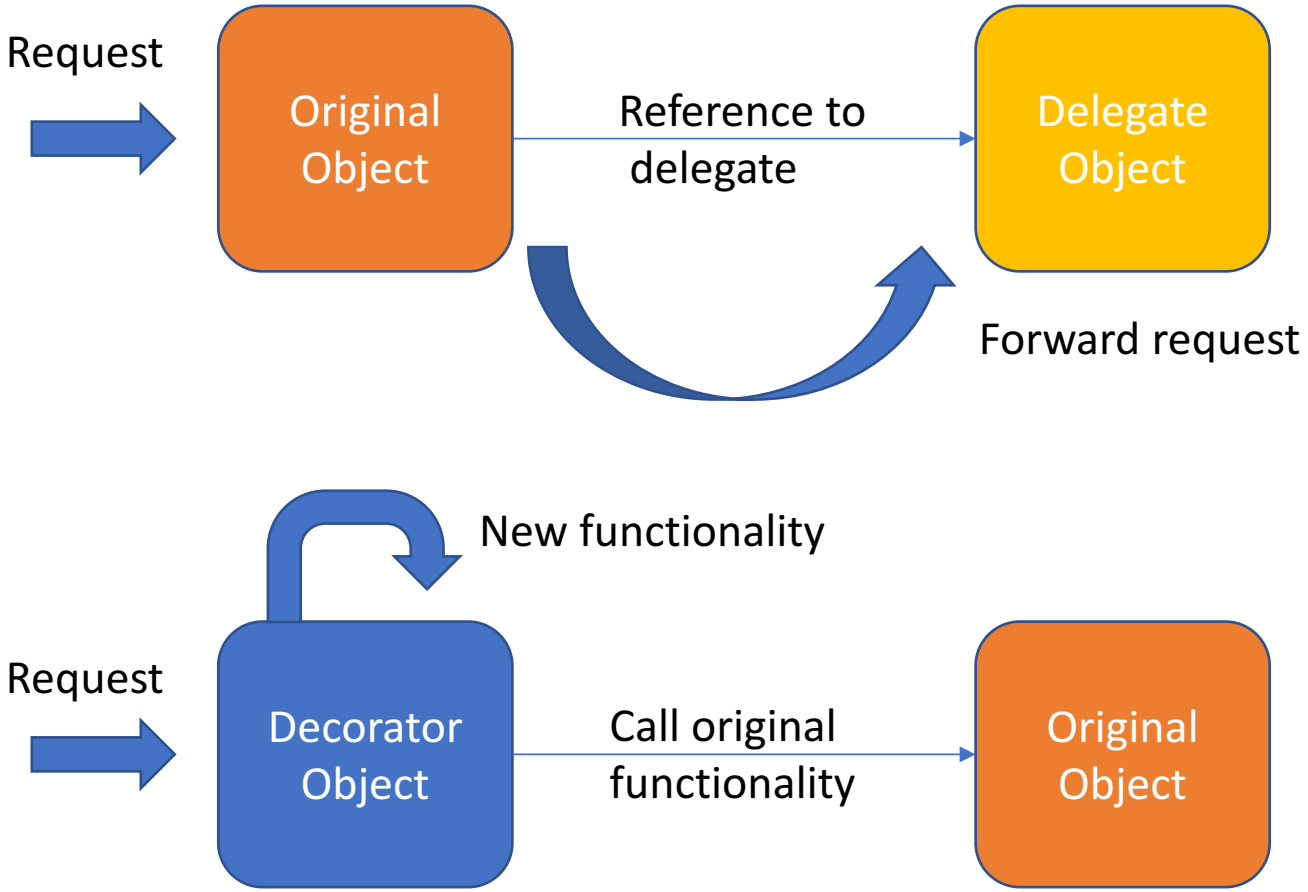


Pure Component based approach

```
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;
```



Delegation and Decorator Patterns



Delegation Pattern

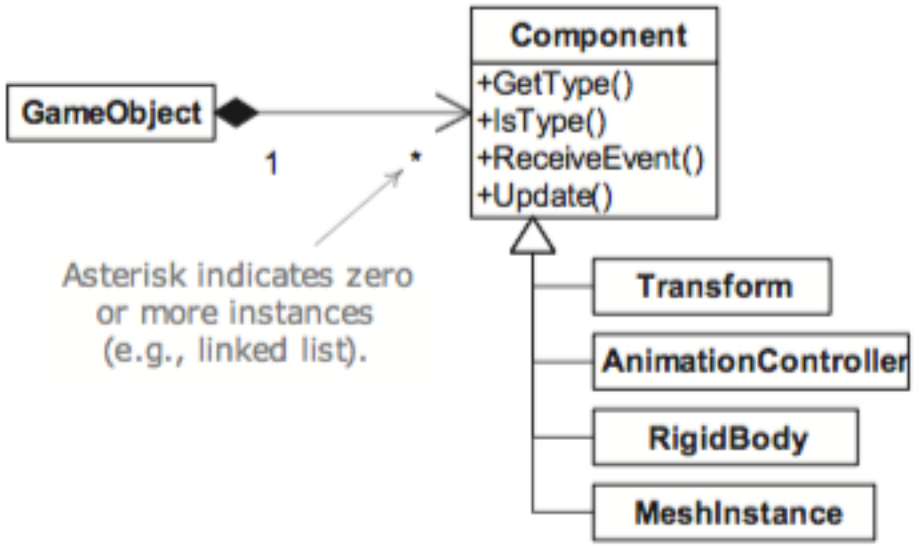
```
public class SortableArray extends ArrayList {

        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)
}
```

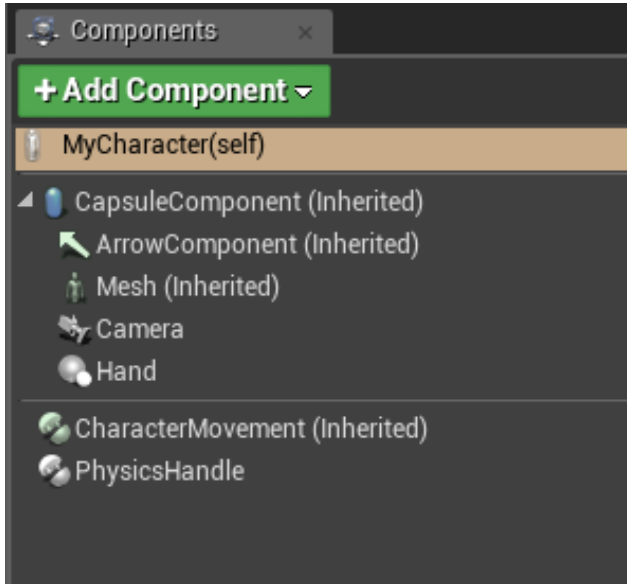
Delegation Pattern

```
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

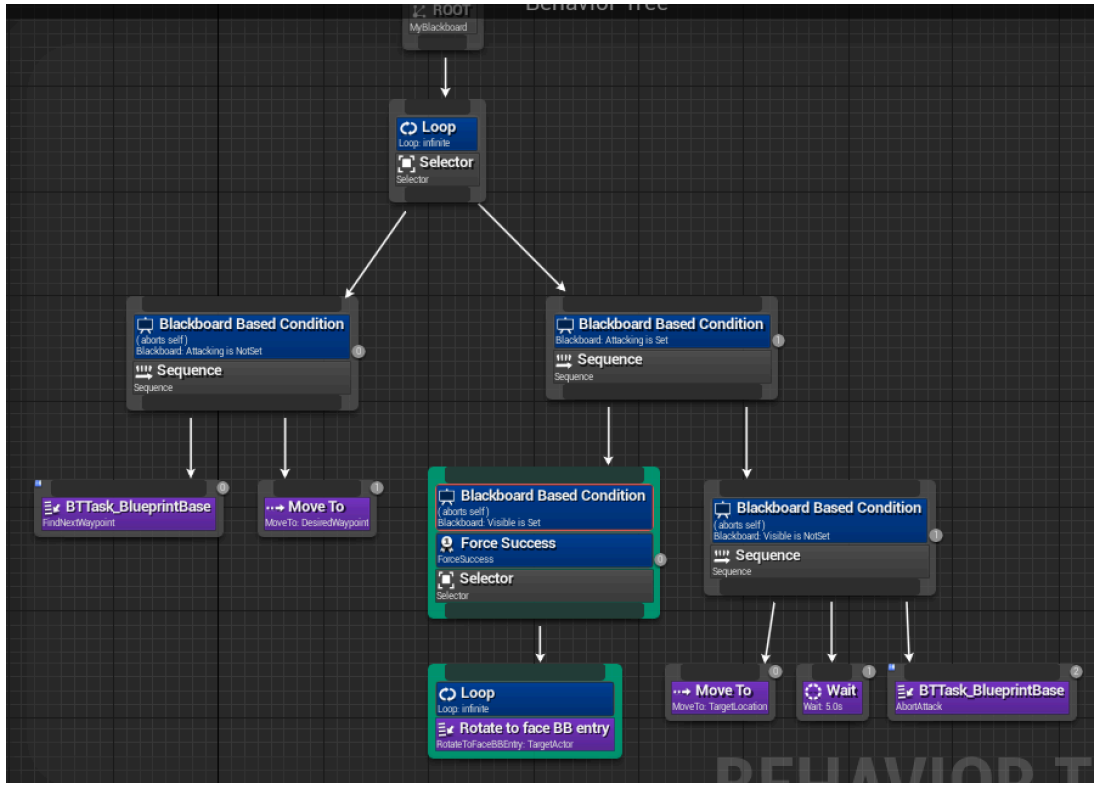
- Delegation
  - Applies to *original object*
    - *We* design the object class
    - Requests made through object
  - Modular solution
    - Each method can have its own delegate implementation
  - Limited to classes that *we* make





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

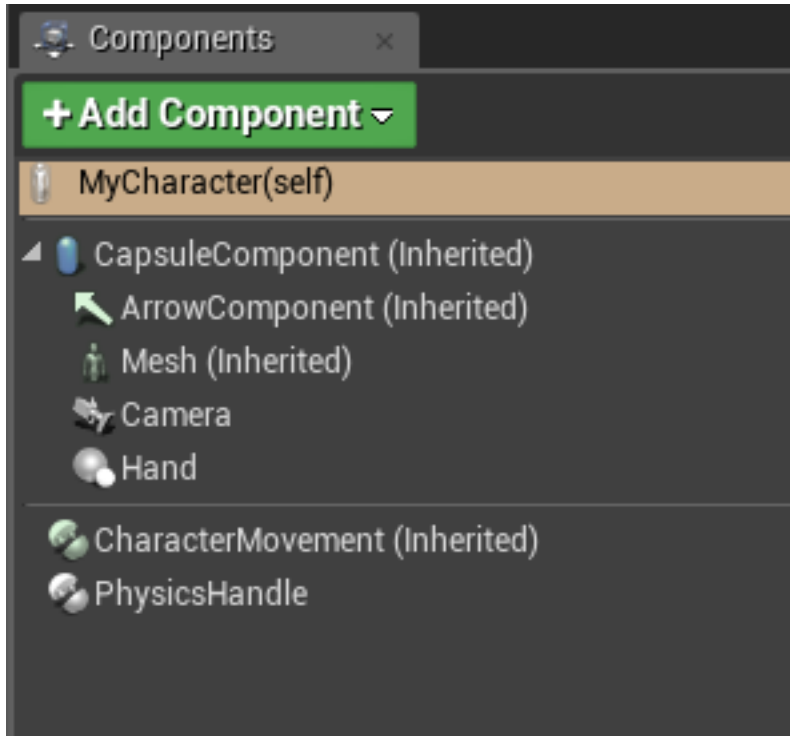


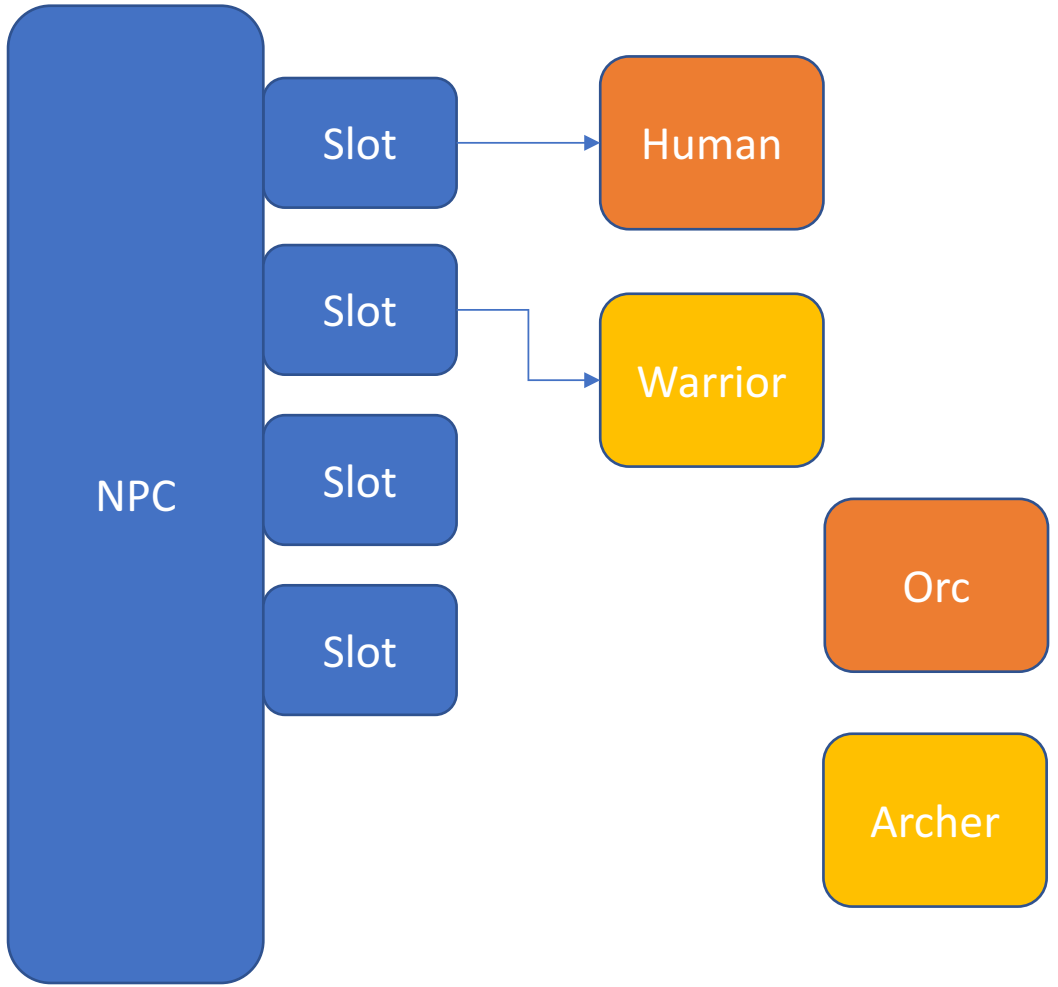
Partial Component based approach

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





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

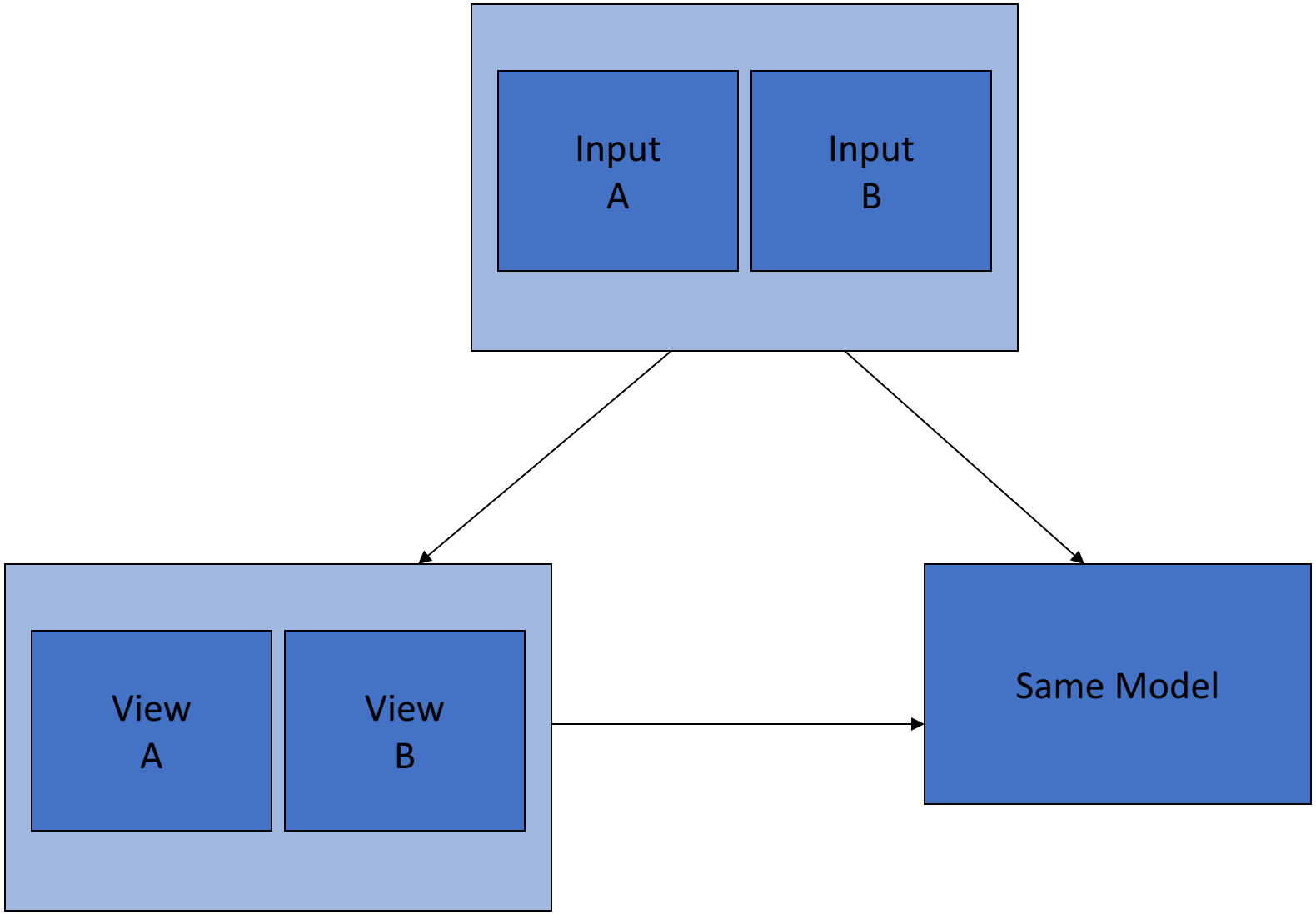
Multiplayer Games

- Single Player
  - Pre-defined challenges, “campaigns”
  - AI controlled opponents
  - Simple MVC design
  - Player vs “the game”
- Multi-Player
  - More than one player can play in the same environment at the same time
    - “Shared virtual environment”
  - Interaction with other players forms a key challenge of the game play
  - How should we build such a system?
  - What are some of the technical issues that arise?

Where is the view?

- Consider the physical experiential context
- Local
  - Players are co-located
  - Share the same console / screen / pc
    - Multiple *controllers* / input-mechanisms
  - Share or split screen into two or four sections
  - Arcade games, racing, fighting, co-operative shooters
- Networked / Online
  - Players are physically separated
  - Game play is shared over the network / Internet
  - Many combinations of players 2 -> ??
  - FPSs, MMORPGs

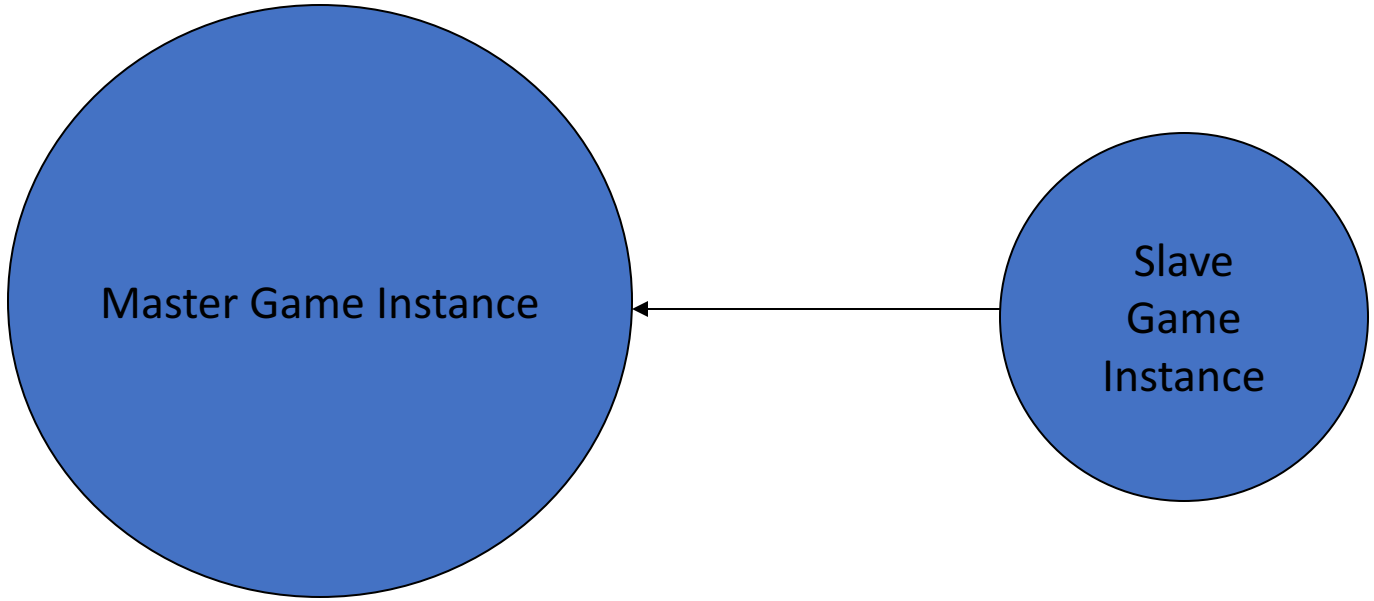
Split-Screen View – Mario Kart

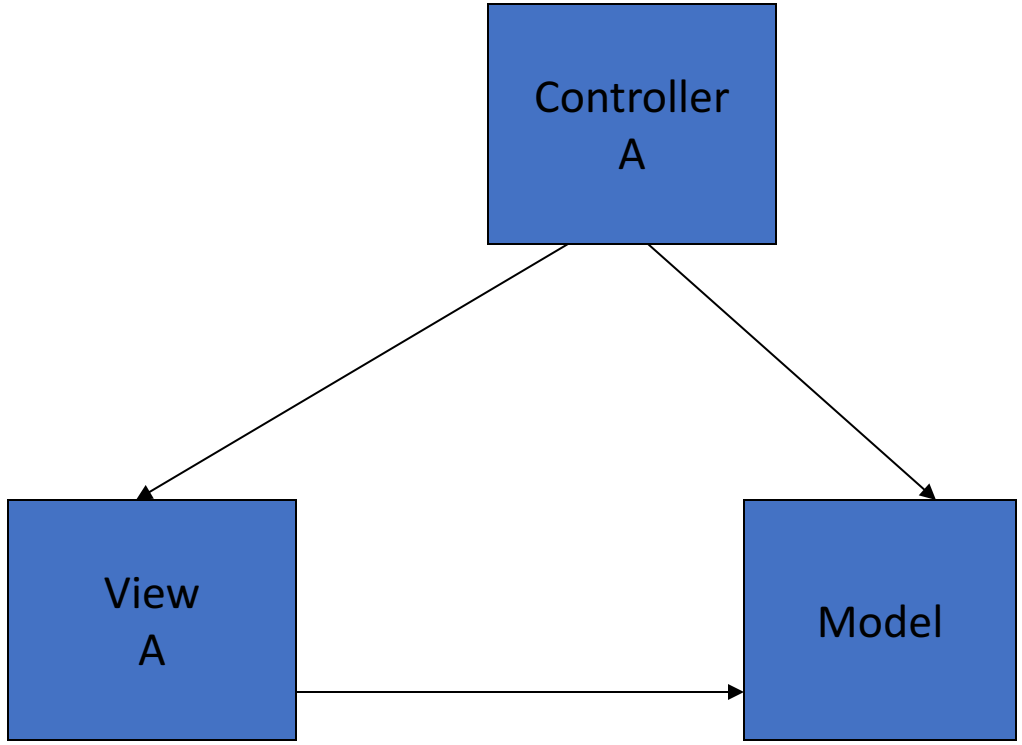

Input
A
Input
B
View
A
View
B
Same Model

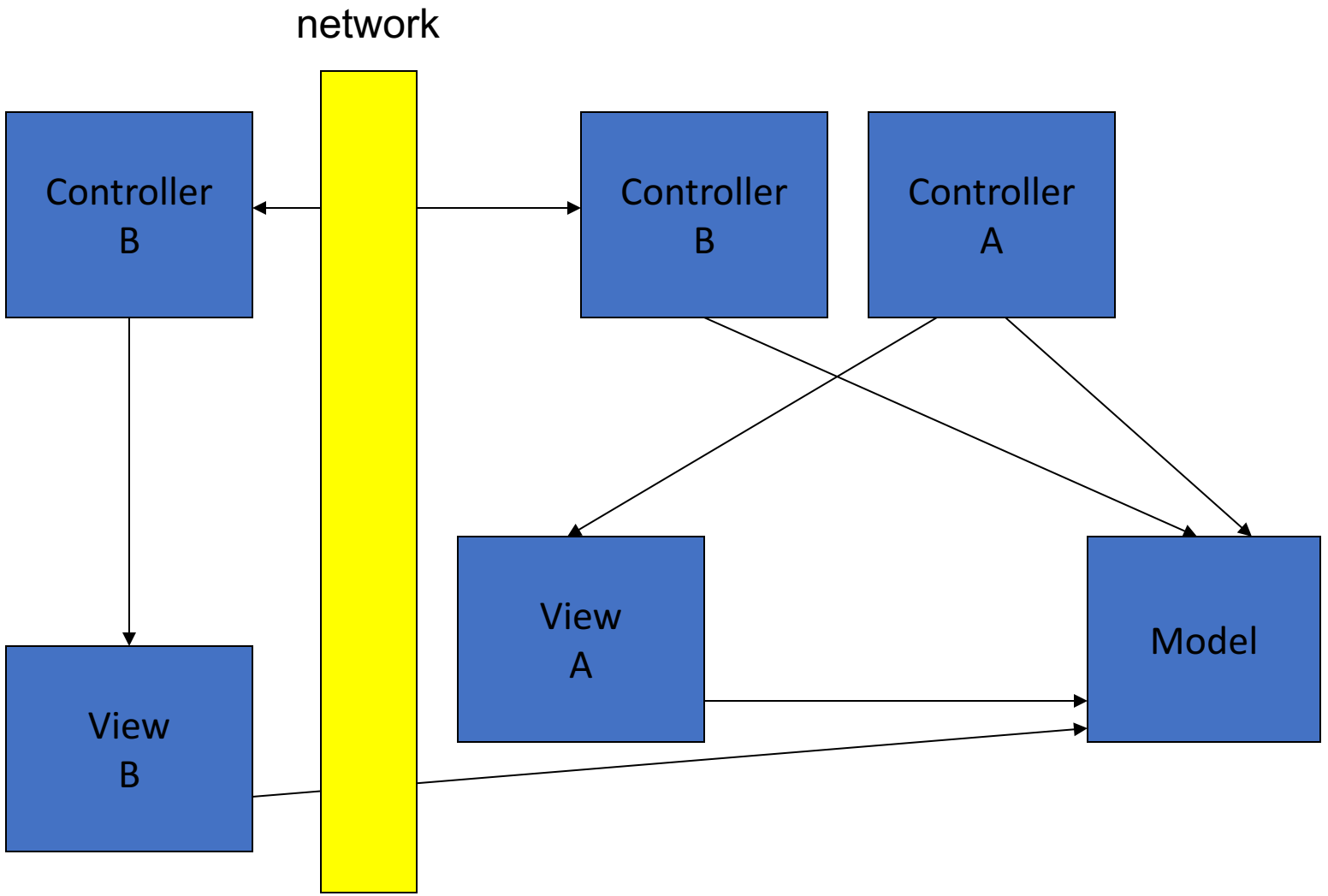
Networked / Online Game Play

- Players are physically separate
  - Communicating sufficient information to maintain a perceptually shared reality, while minimising both bandwidth use and perceived violations of the integrity of the simulation / fiction
  - User input -> model, model -> client view for rendering
- Where is the “game”?
  - An *authoritative model* somewhere (game state)
    - **Never** trust the client
      - **I am here <> I would like to move here**
  - Master and slave
    - One player’s game takes responsibility for the model
    - Usually two players, local network
    - Dynamic master and slave
  - Dedicated server and Clients
    - Dedicated software / machine is responsible for the model
    - Multiple players, local network, internet
  - Peer-to-peer
    - Shared responsibility for the model

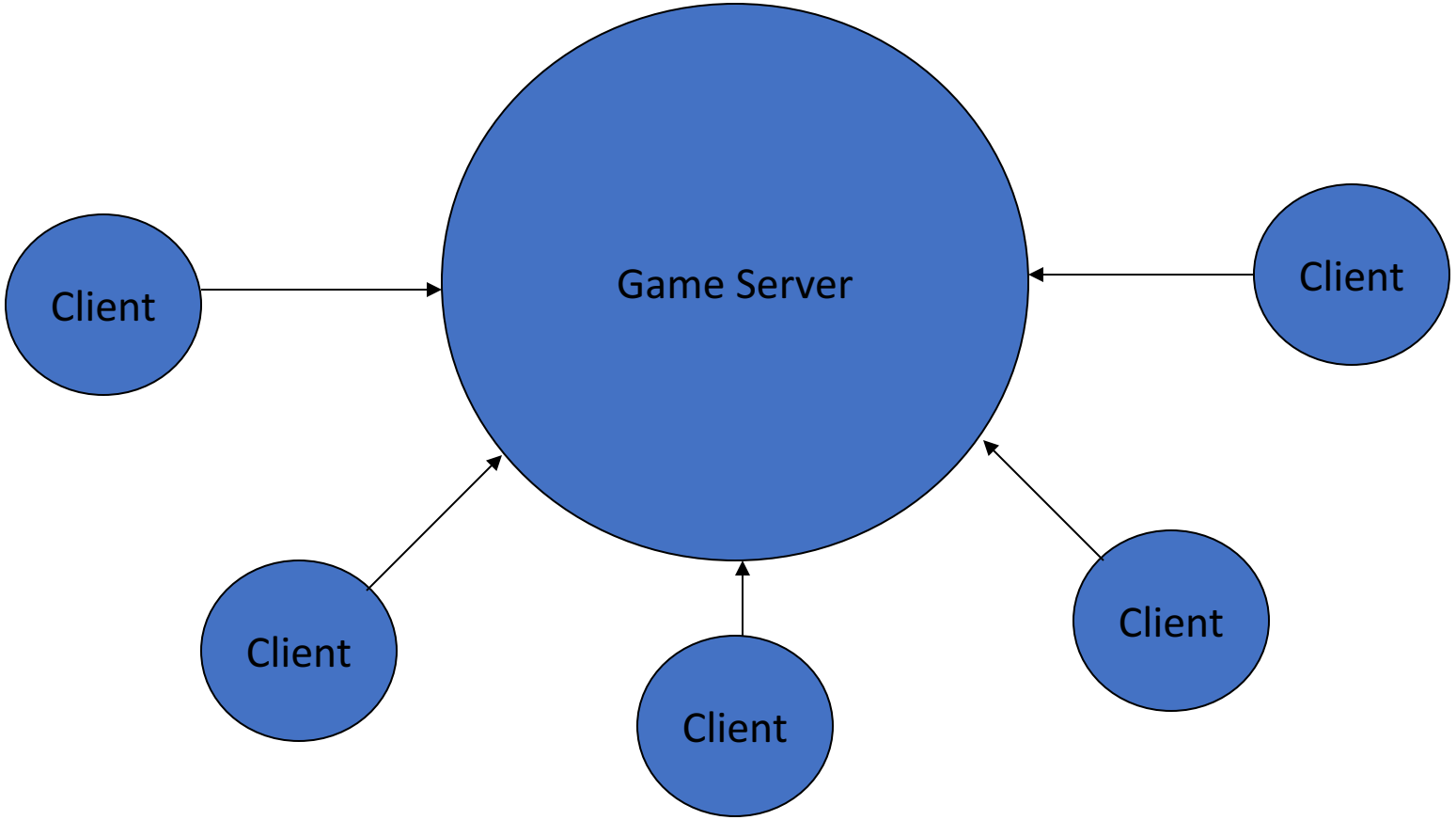
Master and Slave

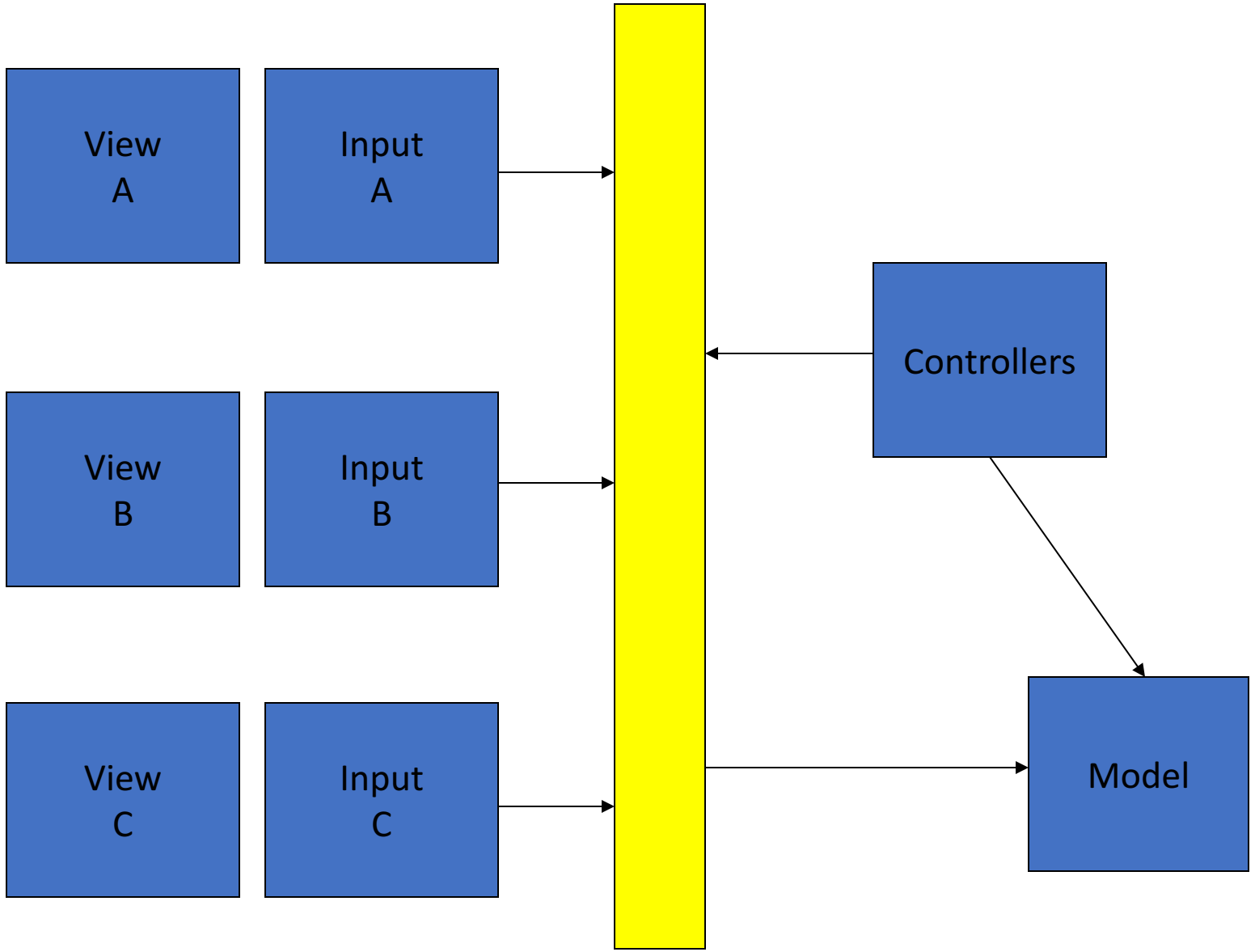



Controller
A
View
A
Model

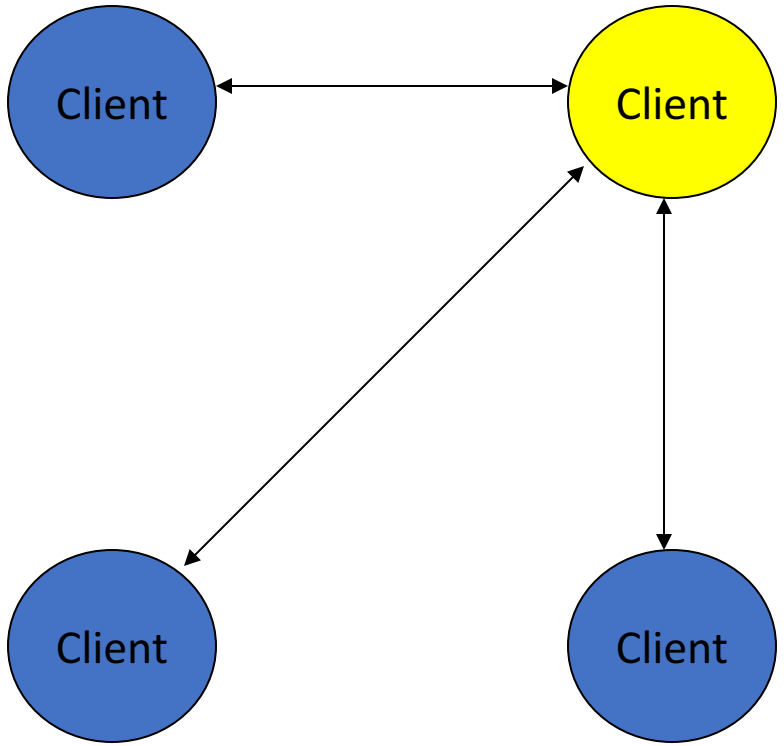

network
Controller
B
Controller
B
Controller
A
View
A
Model
View
B

Client Server

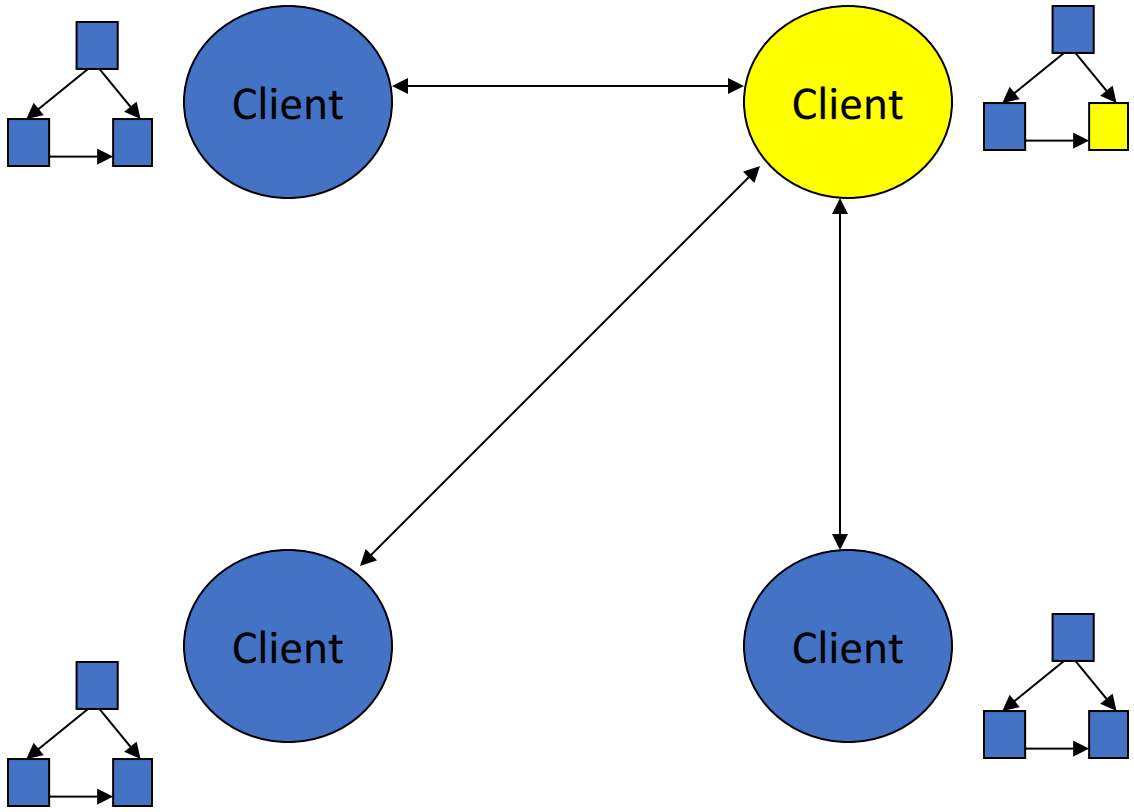



View
A
Input
A
View
B
Input
B
View
C
Input
C
Controllers
Model

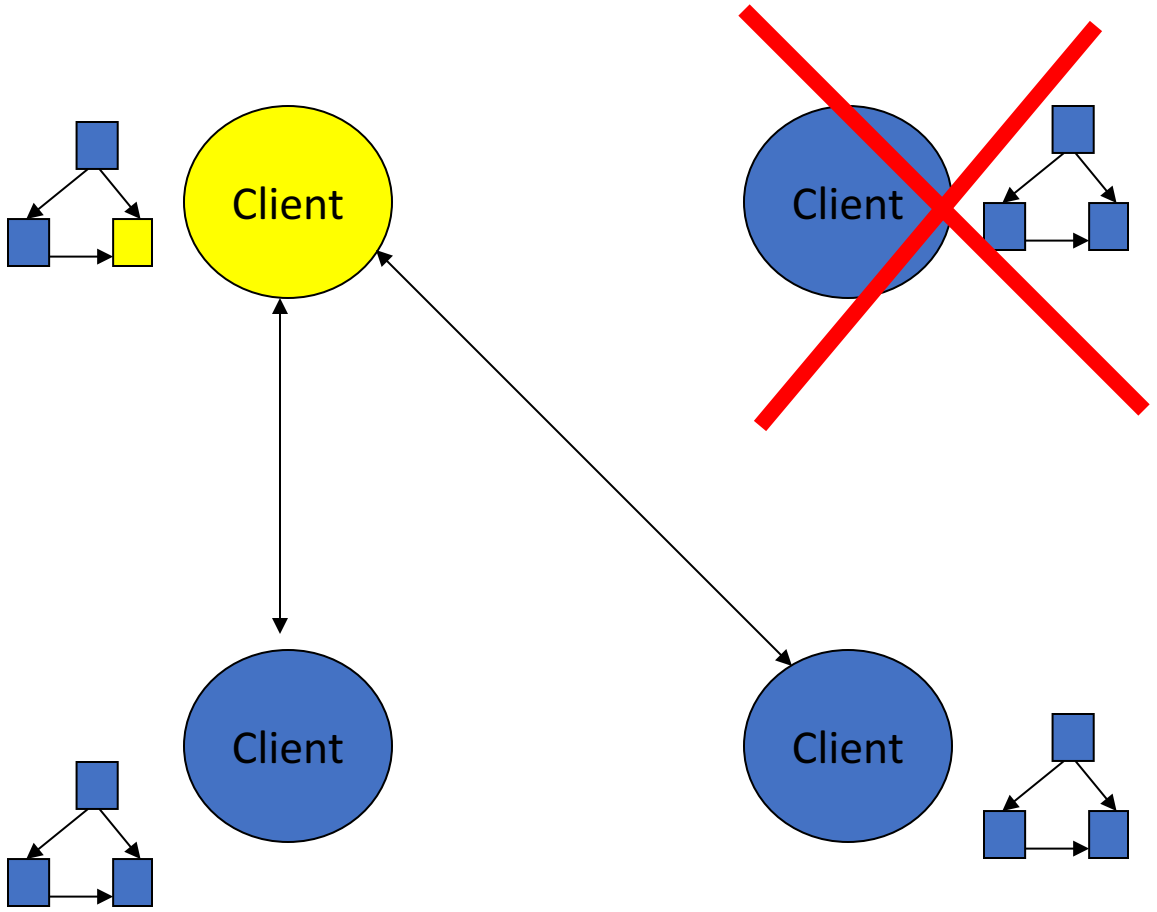
Dynamic Master-Slave

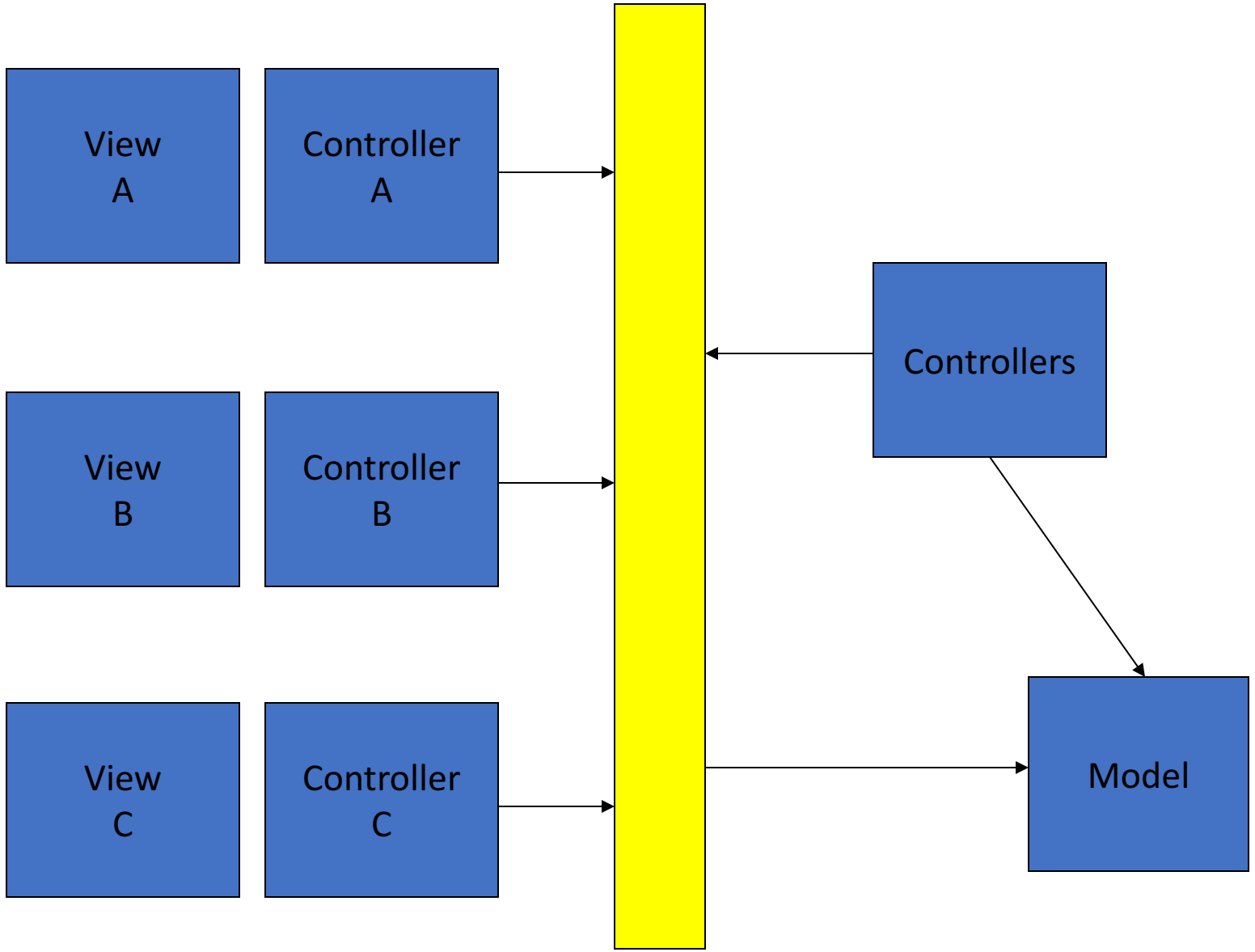


Dynamic Master-Slave



Dynamic Master-Slave

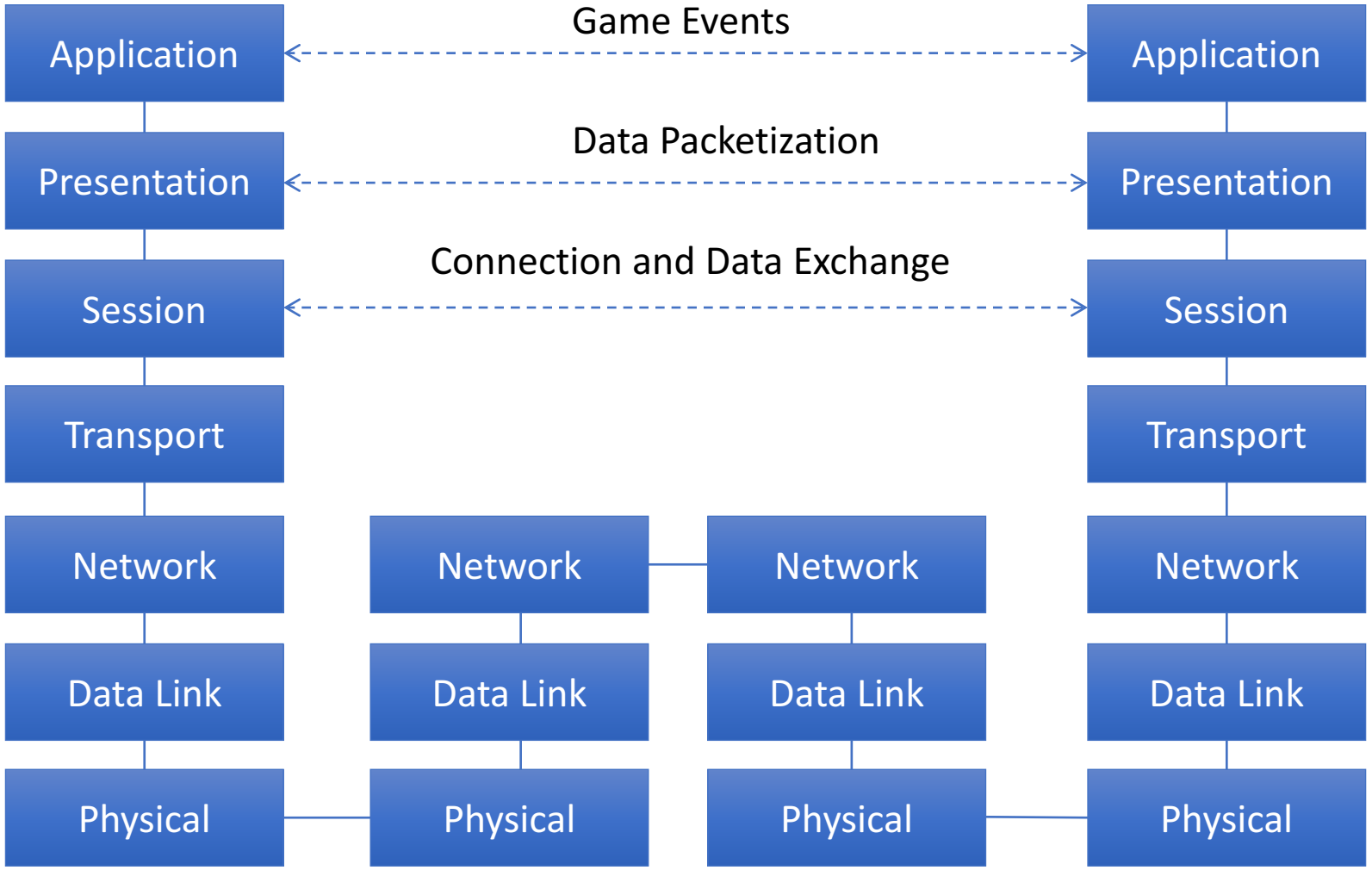



View
A
Controller
A
View
B
Controller
B
View
C
Controller
C
Controllers
Model

Sadly it’s not that simple

- **Real time simulation**
  - Game loop runs at 30 Hz
  - Renderer redraws at 50-100 Hz
- Each input has to…
  - Travel from the client to the server
  - Be processed by the server
  - Wait for the server loop to update the model
  - Travel from the server to the client
  - Be drawn onscreen
  - **Round trip time**
- For several players on the Internet playing a fast paced game
  - **Latency** – Networks can be slow
  - **Bandwidth** - Each client needs to know about the current state of the model every loop
  - **Reliability** - Packets get lost or delayed or arrive out of order

Protocol stack



Common Simplifying Approaches

- Lockstep
  - Deterministic / literal input passing
    - Turn taking, waiting for all clients to have sent input
  - Common for games with a strict split between input and simulation
- Reliable transport protocols
  - Requires high bandwidth or simple networked state
    - For N players, O(N*N) data needs to be sent
  - TCP requires high latency tolerance
- Send all networked state as a single blob (atomically)
  - E.g. Quake 3 model
  - Works well as along as the total networked state is not too large

Lag / Consistency

- Network delays lead to logical Inconsistencies
- A player shoots at another player
  - The player/client thinks they should have hit
  - The **fire** command takes some time to get to the server
  - The server thinks the player missed
  - How do we resolve this?
- Two players try to pick up the same object
  - Does the player with the lowest latency get it?
  - What does the player with the highest latency see?
  - How do we resolve the inconsistency of both players trying to pick it up?
- Implementation needs to have
  - Speed (otherwise it feels slow and jerky)
  - Synchronisation (to avoid logical inconsistencies)
  - Not use too much bandwidth (remember dial-up)
  - Cope with packet loss

Reading

- Game Engine Architecture, Jason Gregory 2014, chapter 14