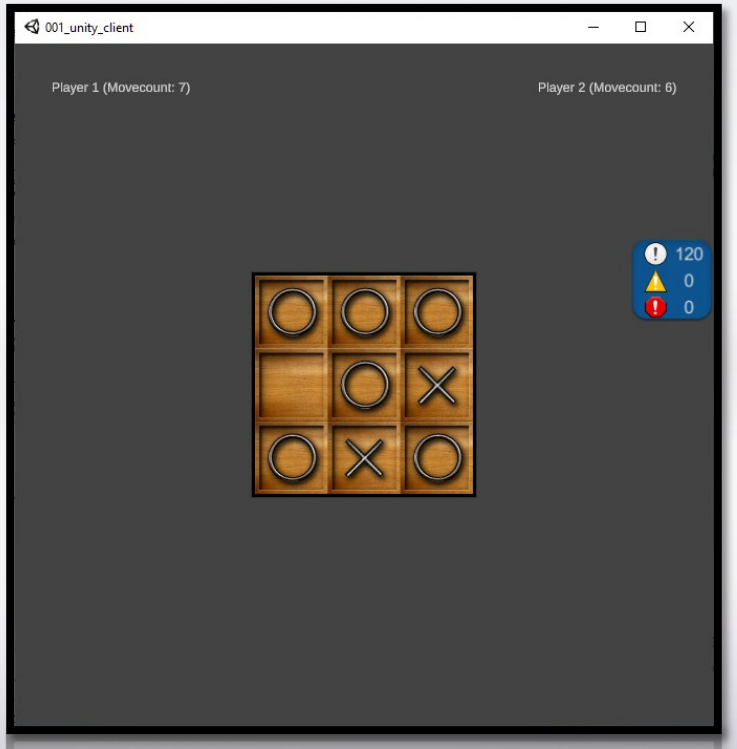
Networking lecture 4

Turn based application design



Recap

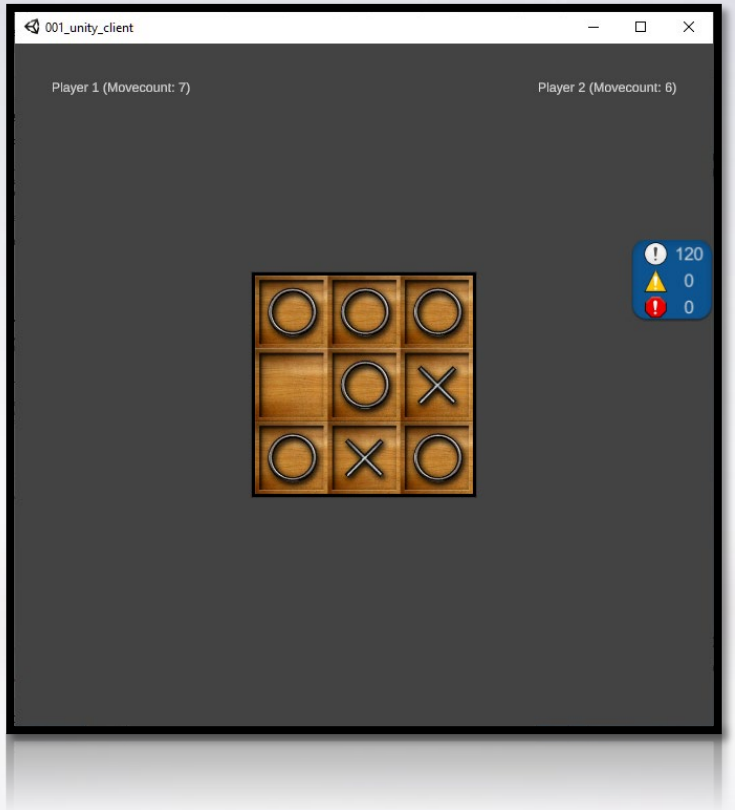
- Lecture 1 Basic networking & TCP communication
- Lecture 2 TCP challenges & pitfalls
- Lecture 3 Object oriented application protocols

Lecture 4 is all about ...

- An example of an (unfinished) turn based network application (because building a networked application is also about design, architecture, states & not losing our minds).

- The example shown this lecture is the base code for assignment 4

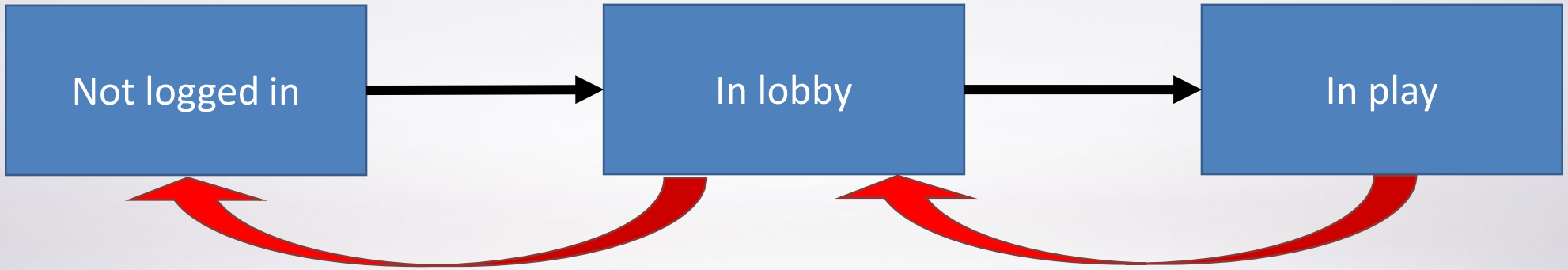
The example in 'action' : FlicFlacFlo*



Variation of TicTacToe, where the goal is to just click-click-click-click and dominate the board until you opponent gets tired and leaves

State based application

A player can be in any of these states
(which is reflected in both client & server architecture)



Note: going back is not an option (yet) !

FlicFlacFlo – The Good

- Functionally:
  - There is a login screen
  - After logging in we automatically join a lobby
  - When enough people are ready,
    the game starts automatically
- Technically:
  - Fairly robust error handling (except at the top level, so things can still go wrong)
  - Nice debug info

FlicFlacFlo – The bad

- Functionally:
  - Login information is not used anywhere (you are still John Doe in the lobby)
  - Lobby chat does not work (but it sure is peaceful)
  - There is no real gameplay (no win condition, no indication which players are playing, whose turn it is, etc).
- Technically:
  - Only one game can be started (which never ends ☺)

But it is a good starting point...

- We have a very basic application with different states
- Implementation demonstrates a lot of core features

Goal of this lecture: explain the design, architecture and details of this application so you can extend it, in 3 parts:

SHARED

Serialization layer
Protocol objects
Model data

SERVER

Rooms

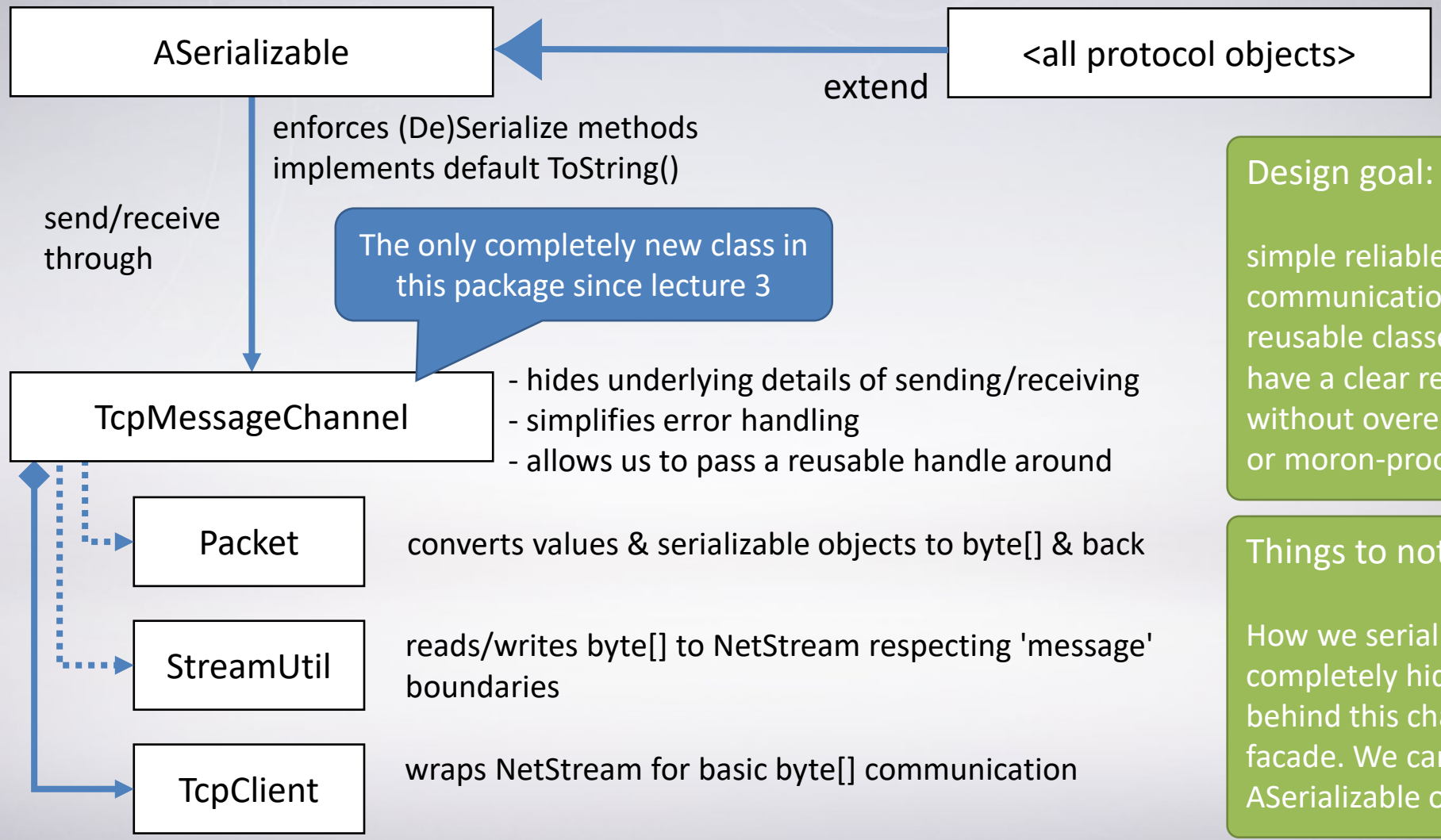
CLIENT

Finite State Machines
Views & view handling
Utility stuff (logging etc)

Shared classes

Serialization layer

Serialization layer



Design goal:

simple reliable object communication, with reusable classes that have a clear responsibility without overengineering or moron-proof clutter.

Things to note:

How we serialize is completely hidden behind this channel facade. We can send ***any*** ASerializable object.

TcpMessageChannel facade

- Hides details of sending/receiving messages
- Can be passed around as a single handle and 'reused' (TcpClient cannot be reused after being closed)

- Most important methods:
  - Connect(...)
  - SendMessage (...)
  - HasMessage()
  - ReceiveMessage() : ....
  - Connected (takes errors into account)
  - HasErrors() / GetErrors()
  - Close()

ASerializable = ISerializable + debugging

```
public abstract class ASerializable
{
    abstract public void Serialize(Packet pPacket);
    abstract public void Deserialize(Packet pPacket);

    public override string ToString()
    {
        StringBuilder builder = new StringBuilder();
        builder.Append("\n" + GetType().Name + ":");
        builder.Append("\n------------------------------------------------------------");

        IEnumerable<FieldInfo> publicFields = GetType().GetFields().Where(f => f.IsPublic);
        foreach (FieldInfo field in publicFields)
        {
            object value = field.GetValue(this);
            if (value is ICollection)
            {
                ICollection collection = value as ICollection;
                foreach (object item in collection) builder.Append(item.ToString());
            }
            else
            {
                builder.Append(String.Format("\nName: {0} \t\t\t Value: {1}", field.Name, value) + "");
            }
        }
        builder.Append("\n------------------------------------------------------------");
        return builder.ToString();
    }
}
```

StreamUtil Improvement

```
public static class StreamUtil
{
    private const int HEADER_SIZE = 4;

    /**
     * Optimized 'Available' check that FIRST checks if header data is available and THEN checks
     * if the actual data is available as well.
     */
    public static bool Available(TcpClient pClient)
    {
        if (pClient.Available < HEADER_SIZE) return false;
        byte[] sizeHeader = new byte[HEADER_SIZE];
        pClient.Client.Receive(sizeHeader, HEADER_SIZE, SocketFlags.Peek);
        int messageSize = BitConverter.ToInt32(sizeHeader, 0);
        return pClient.Available >= HEADER_SIZE + messageSize;
    }

    }
```

Log class

```
TCPGameServer.run():Starting server on port 55555

TCPGameServer.run():Accepting new client...

TcpMessageChannel..ctor():TCPMessageChannel created around System.Net.Sockets.TcpClient

LoginRoom.addMember():Client joined.


TcpMessageChannel.SendMessage():
RoomJoinedEvent:
------------------------------------------------------------
Name: room                      Value: LOGIN_ROOM
------------------------------------------------------------


TcpMessageChannel.ReceiveMessage():Receiving message...
TcpMessageChannel.ReceiveMessage():Received
PlayerJoinRequest:
------------------------------------------------------------
Name: name                      Value: Hans
------------------------------------------------------------

LoginRoom.handlePlayerJoinRequest():Moving new client to accepted...


TcpMessageChannel.SendMessage():
PlayerJoinResponse:
------------------------------------------------------------
Name: result                    Value: ACCEPTED
------------------------------------------------------------
```

What sort of messages?

A limited set of example messages is provided.

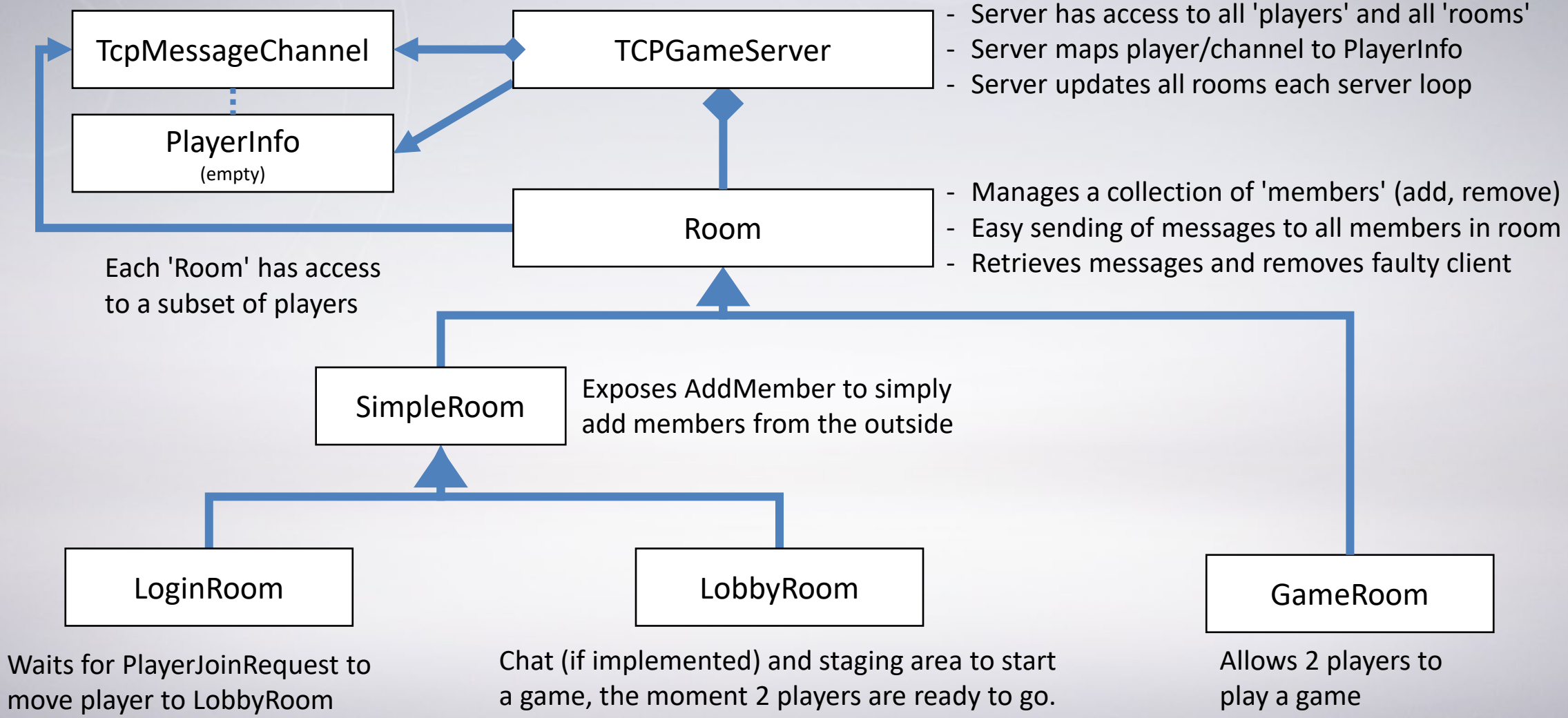
(see protocol folder)

Not logged in	In lobby	In play
PlayerJoinRequest PlayerJoinResponse	ChangeReadyStatusRequest LobbyInfoUpdate (ChatMessage)	MakeMoveRequest MakeMoveResult
(RoomJoinedEvent)	(RoomJoinedEvent)	(RoomJoinedEvent)

Server setup

Rooms and more ...

Server setup



Summing up

- Server is a sort of room manager:
  - accepts players & puts them in the correct starting room
  - manages and updates existing rooms
  - manages PlayerInfo for all connections (not used atm)
  - rooms take care of the rest
    - updating clients, getting messages, sending replies
    - moving clients to other rooms if required
    - removing clients when they are causing errors
    - each room encapsulates a specific piece of functionality (login, lobby, play)

Server setup

```
while (true)
{
    //check for new members
    if (listener.Pending())
    {
        //get the waiting client
        Log.LogInfo("Accepting new client...", this, ConsoleColor.White);
        TcpClient client = listener.AcceptTcpClient();
        //and wrap the client in an easier to use communication channel
        TcpMessageChannel channel = new TcpMessageChannel(client);
        //and add it to the login room for further 'processing'
        _loginRoom.AddMember(channel);
    }

    //now update every single room
    _loginRoom.Update();
    _lobbyRoom.Update();
    _gameRoom.Update();

    Thread.Sleep(100);
}
```

Note the 1 Game room limitation (1 room == 1 game!)

Base Room class 1/2

```
public virtual void Update()
{
    removeFaultyMembers();
    receiveAndProcessNetworkMessages();
}
```

```
protected void removeFaultyMembers()
{
    safeForEach(checkFaultyMember);
}
```

```
protected void receiveAndProcessNetworkMessages()
{
    safeForEach(receiveAndProcessNetworkMessagesFromMember);
}
```

```
protected void safeForEach(Action<TcpMessageChannel> pMethod)
{
    for (int i = _members.Count - 1; i >= 0; i--)
    {
        //skip any members that have been 'killed' in the mean time
        if (i >= _members.Count) continue;
        //call the method on any still existing member
        pMethod(_members[i]);
    }
}
```

Base Room class 2/2

The actual 'handling' is left to the subclass:

```
/**
 * Iterate over all members and get their network messages.
 */
protected void receiveAndProcessNetworkMessages()
{
    safeForEach(receiveAndProcessNetworkMessagesFromMember);
}

/**
 * Get all the messages from a specific member and process them
 */
private void receiveAndProcessNetworkMessagesFromMember(TcpMessageChannel pMember)
{
    while (pMember.HasMessage())
    {
        handleNetworkMessage(pMember.ReceiveMessage(), pMember);
    }
}

abstract protected void handleNetworkMessage(ASerializable pMessage, TcpMessageChannel pSender);
```

LoginRoom Example

```
protected override void handleNetworkMessage(ASerializable pMessage, TcpMessageChannel pSender)
{
    if (pMessage is PlayerJoinRequest)
    {
        handlePlayerJoinRequest(pMessage as PlayerJoinRequest, pSender);
    }
    else //if member send something else than a PlayerJoinRequest
    {
        Log.LogInfo("Declining client, auth request not understood", this);

        //don't provide info back to the member on what it is we expect, just close and remove
        removeAndCloseMember(pSender);
    }
}
```

```
private void handlePlayerJoinRequest (PlayerJoinRequest pMessage, TcpMessageChannel pSender)
{
    Log.LogInfo("Moving new client to accepted...", this);

    PlayerJoinResponse playerJoinResponse = new PlayerJoinResponse();
    playerJoinResponse.result = PlayerJoinResponse.State.ACCEPTED;
    pSender.SendMessage(playerJoinResponse);

    removeMember(pSender);
    _server.GetLobbyRoom().AddMember(pSender);
}
```

LobbyRoom

- Keeps track of amount of (ready) players in the lobby
- Updates client with lobby information
- Starts a game if enough players are ready.

Excerpt from LobbyRoom.handleReadyNotification

```
if (_readyMembers.Count >= 2 && !_server.GetGameRoom().IsGameInPlay)
{
    TcpMessageChannel player1 = _readyMembers[0];
    TcpMessageChannel player2 = _readyMembers[1];
    removeMember(player1);
    removeMember(player2);
    _server.GetGameRoom().StartGame(player1, player2);
}
```

GameRoom

- maintains (serializable) board data
- converts MakeMoveRequests into moves on the board
- communicates board state back to clients
- lacks real game logic:

```
private void handleMakeMoveRequest(MakeMoveRequest pMessage, TcpMessageChannel pSender)
{
    //we have two players, so index of sender is 0 or 1
    //playerID becomes 1 or 2
    int playerID = indexOfMember(pSender) + 1;
    //make the requested move (0-8) on the board for the player
    _board.MakeMove(pMessage.move, playerID);

    //and send the result of the boardstate back to all clients
    MakeMoveResult makeMoveResult = new MakeMoveResult();
    makeMoveResult.whoMadeTheMove = playerID;
    makeMoveResult.boardData = _board.GetBoardData();
    sendToAll(makeMoveResult);
}
```

Client setup

Finite state machines, views & more ...

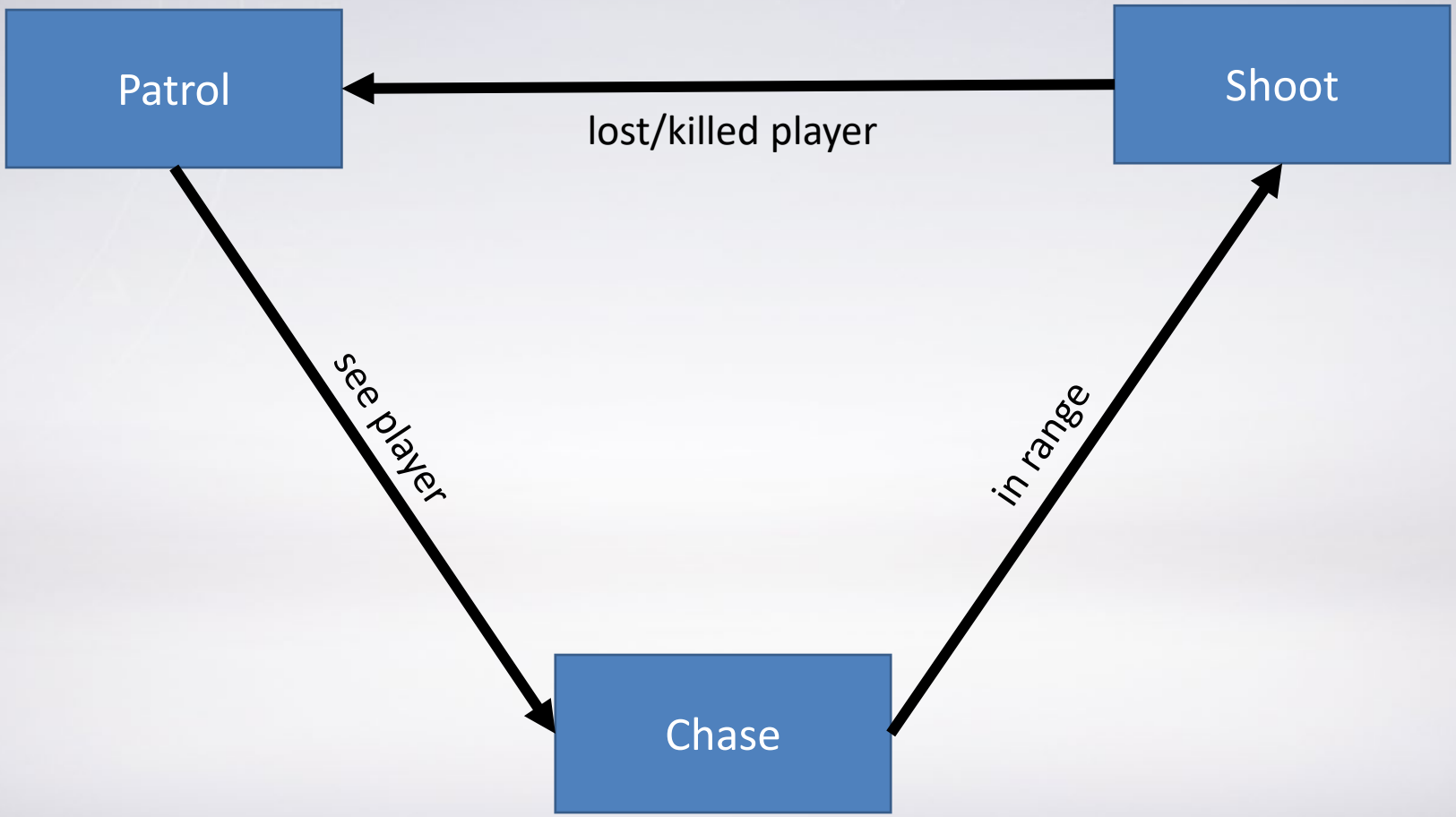
Client state

- On the serverside a client moves between rooms based on the messages it sends to the room it is currently in.
- But what happens on the client side?
  - Clientside state should reflect the client's server state
  - Different ways, this example simply sends *RoomJoinedEvent*s
  - RoomJoinedEvents tell client which state it's currently in
- We'll explore the client first and then get back to some server-client communication details.

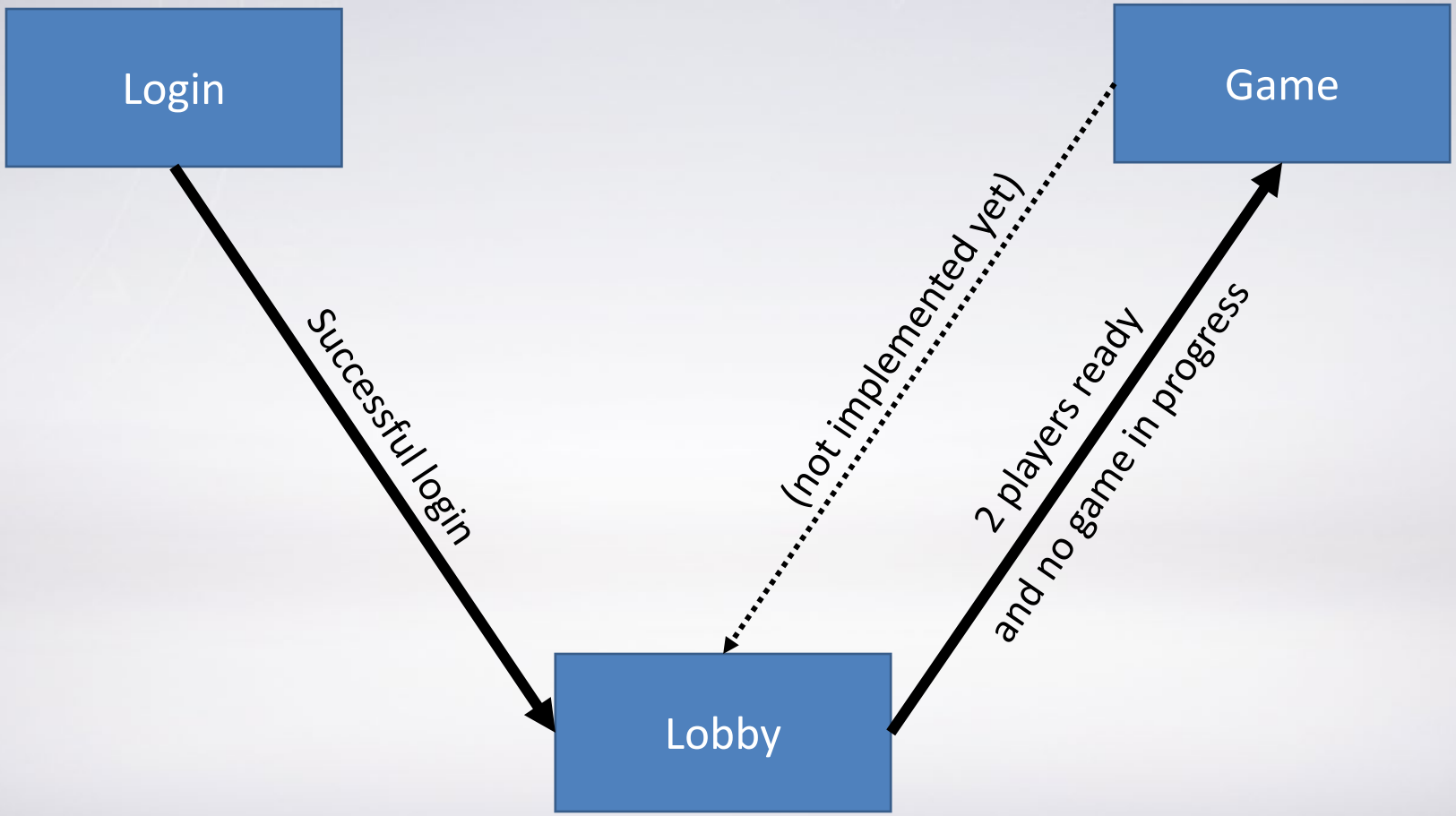
Before we dive into the code..

- Problem:
  - How do you model complex behavior and interactions without writing unreadable/spaghetti code ?
- Possible solution:
  - Split code into clear states & transitions
  - AKA the ***Finite State Machine*** design pattern

Example 1 Enemy AI behaviour



Example 2 Unity client behavior ☺



Advantages?

- Idea is that thinking about your code in terms of states & transitions will lead to cleaner code
- Code is more cohesive, each state implements a specific task (Patrol OR Chase OR Attack and not Patrol&Chase&Attack)

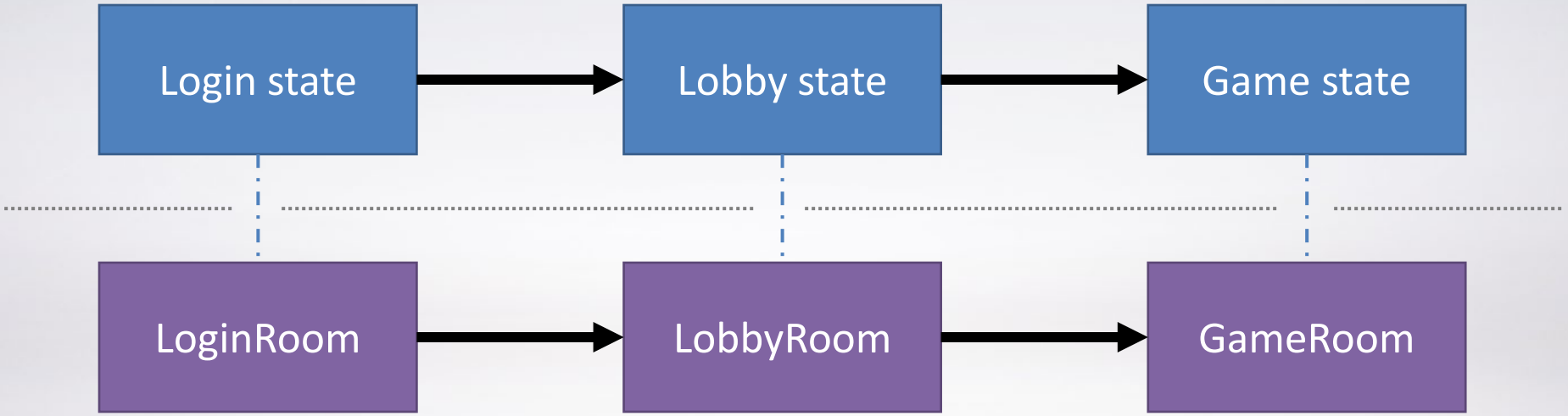
- Of course:
  - they don't solve everything
  - they are not moron proof

Example client states

- Login state:
  - you only worry about possible log-in protocol messages
- Lobby state:
  - you only worry about possible lobby protocol messages
- Game state:
  - you only worry about possible game protocol messages
- etc...

Client setup mirrors server setup

Clientside, client is in state ...



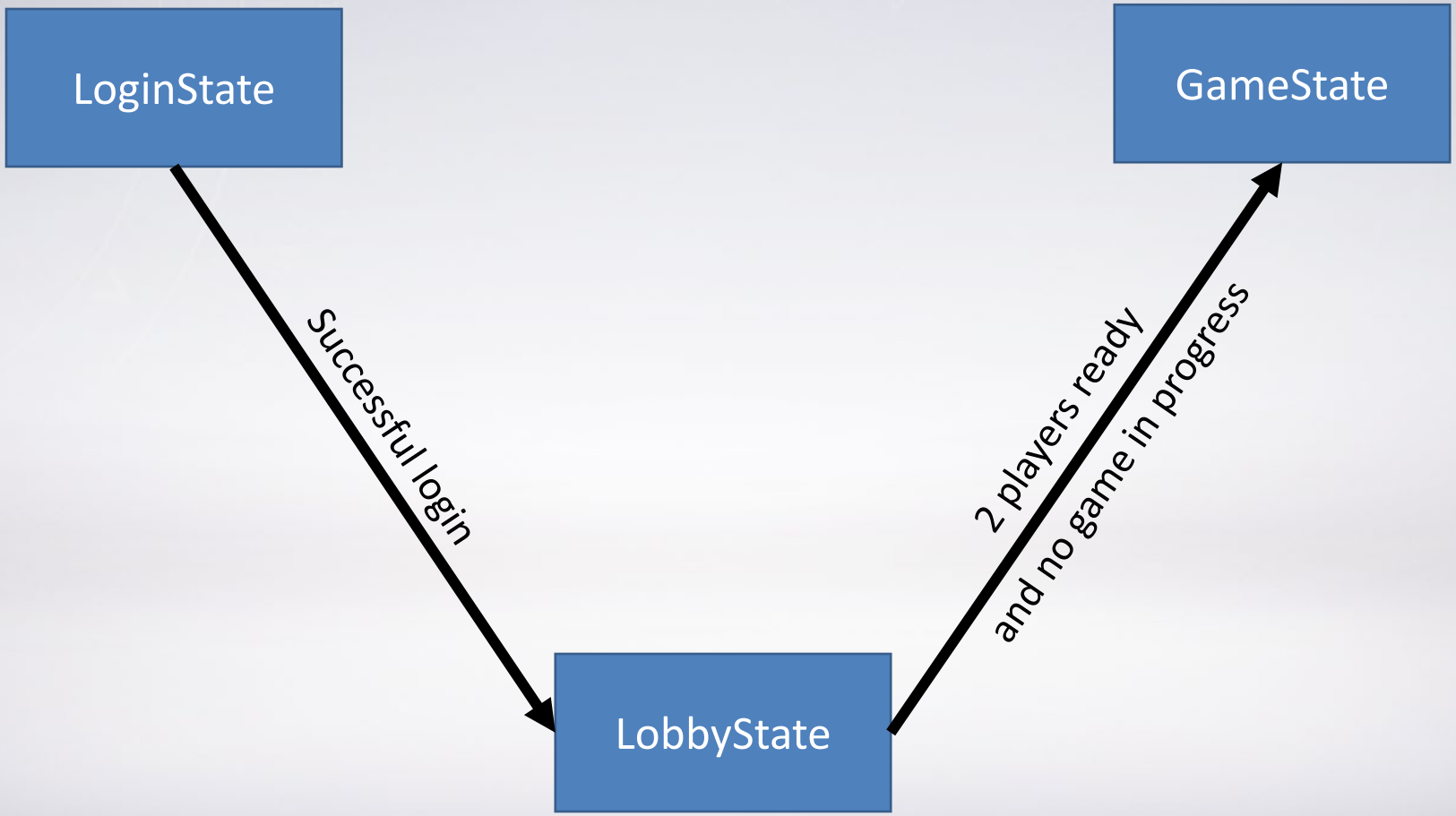
Serverside, client is in room ...

State implementation options?

- if then else (preferably using enums)
  - if (state == State.Login) doLogin();
- delegates/functions pointers:
  - state = doLogin;
  - Update() { state(); }
- class based:
  - LoginState (with Enter/Update/Exit methods)

- More information (good read !!):
  - https://gameprogrammingpatterns.com/state.html

Client is a class based FSM



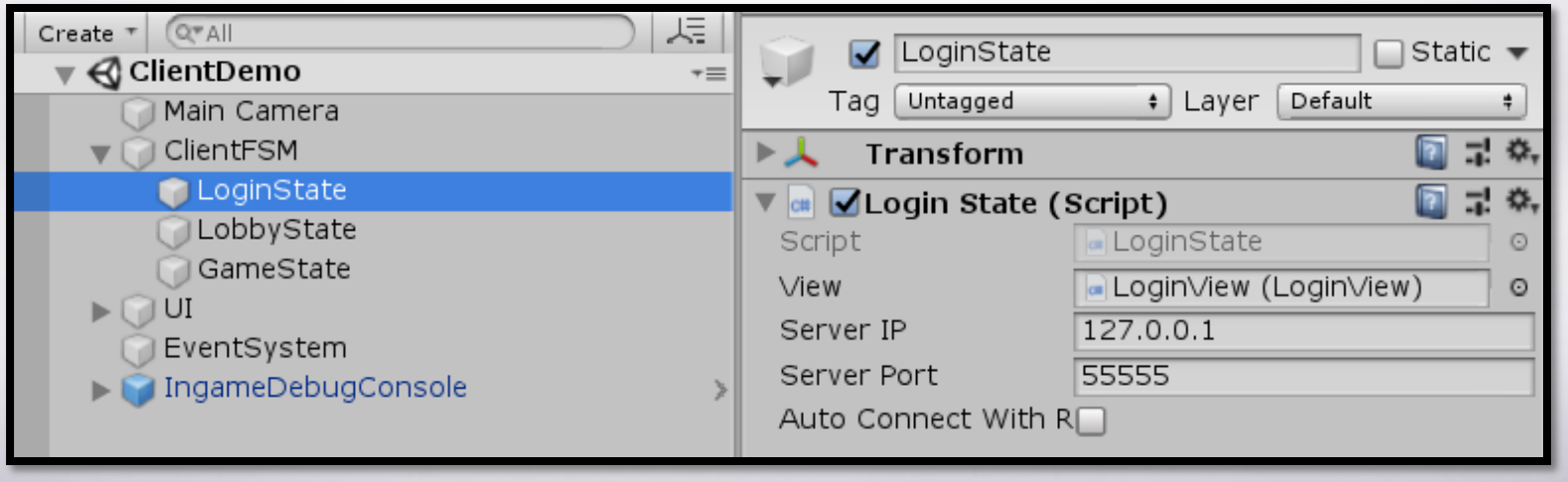
Why class based ?

- Multiple disjunct sets of incoming/outgoing messages that need to be send/received in Update loop
- Each client state might have different graphics/ui/etc
- For anything but the simplest state machines,
  classes are often the best option

States in Unity

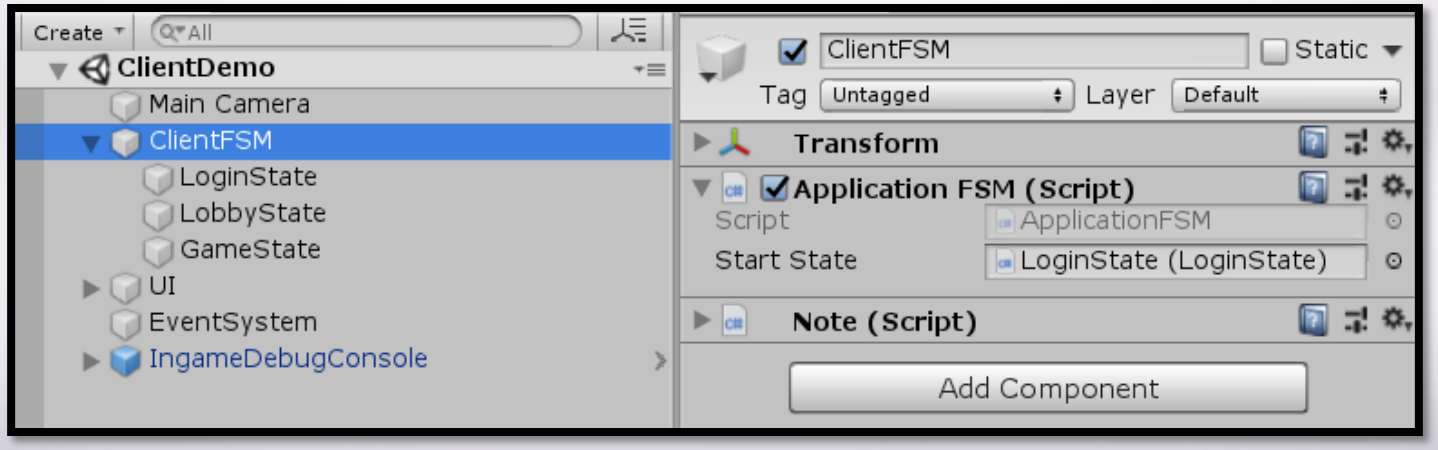
One possible approach:

- Each state is a GameObject with a ...State script
- Each ...State script has EnterState(), ExitState() & Update() methods
  - EnterState() → gameObject.SetActive(true)
  - ExitState() → gameObject.SetActive(false);



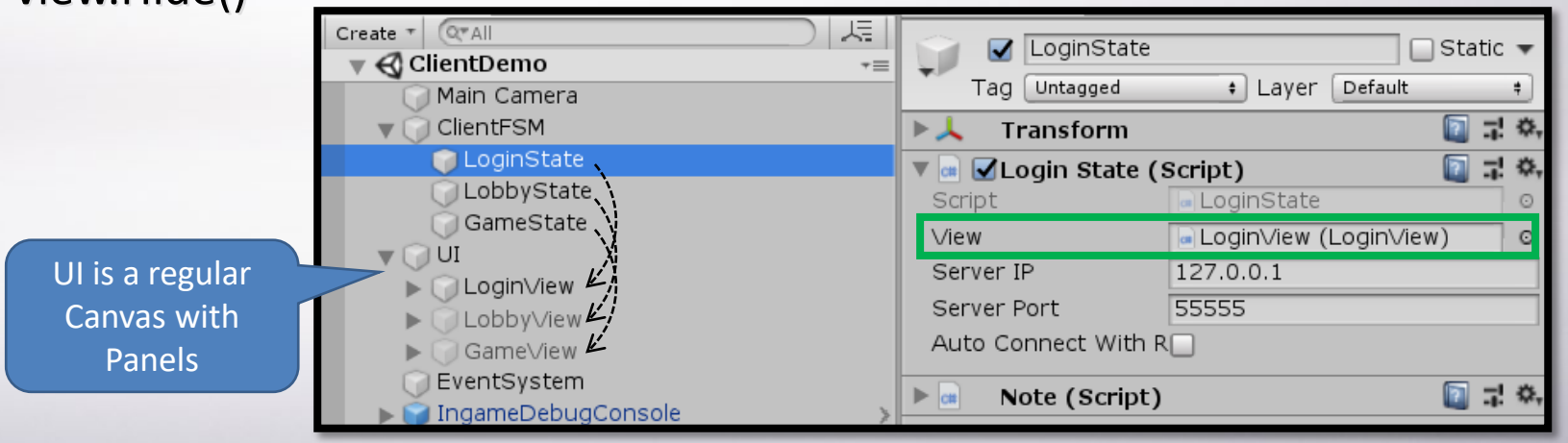
State machine in Unity

- Main 'application' object with FSM script:
  - Finds all state instances on child objects
  - Manages those states through ChangeState(..) calls
  - Calls EnterState()/ExitState() as required
  - (Update() is called automatically of course)



States can also have views/UI

- Bit clumsy in Unity since regular Transforms and RectTransforms are in separate 'trees'
- Simple solution here:
  - each state references a single view
    - EnterState () → view.Show()
    - ExitState() → view.Hide()

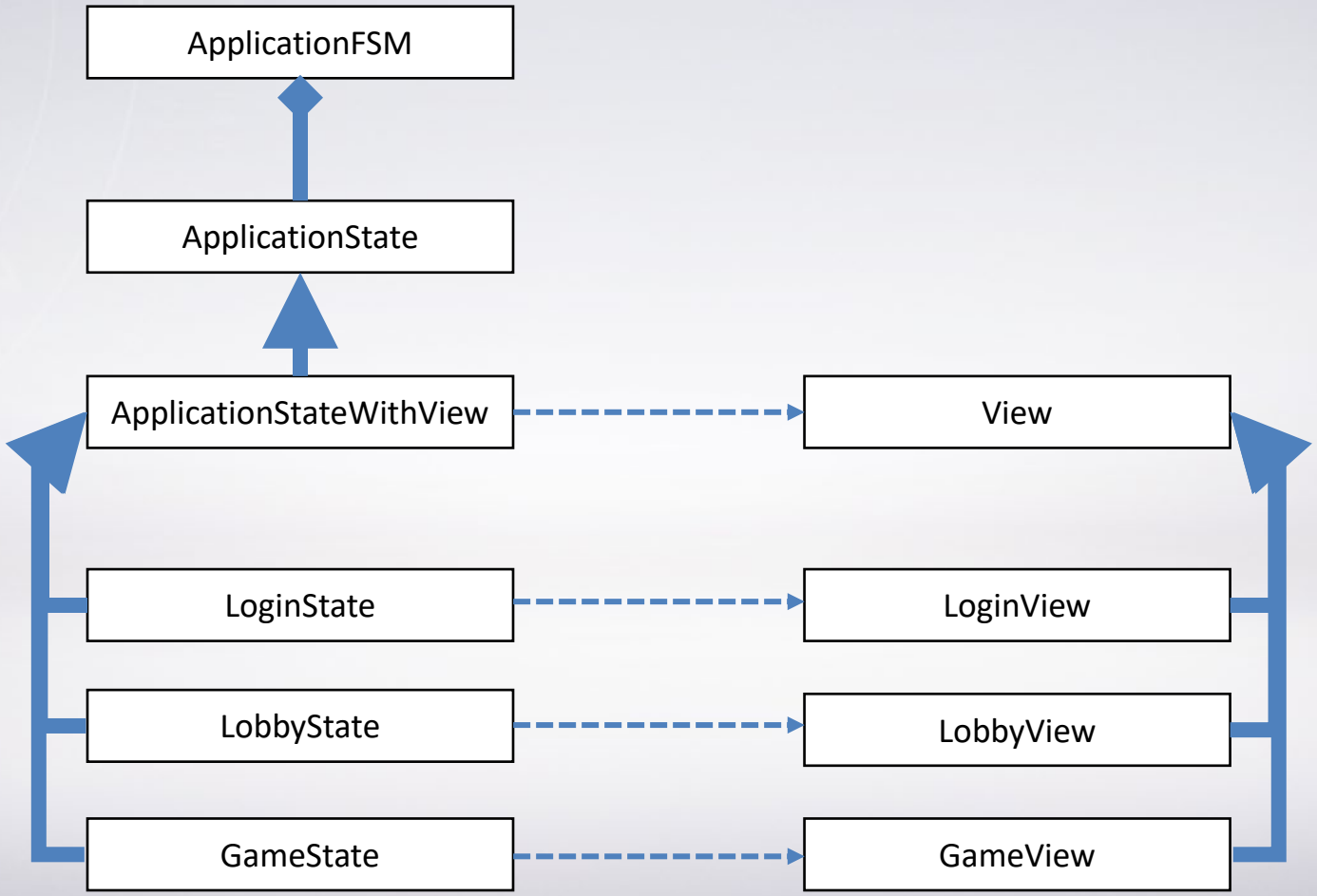


What is a View?

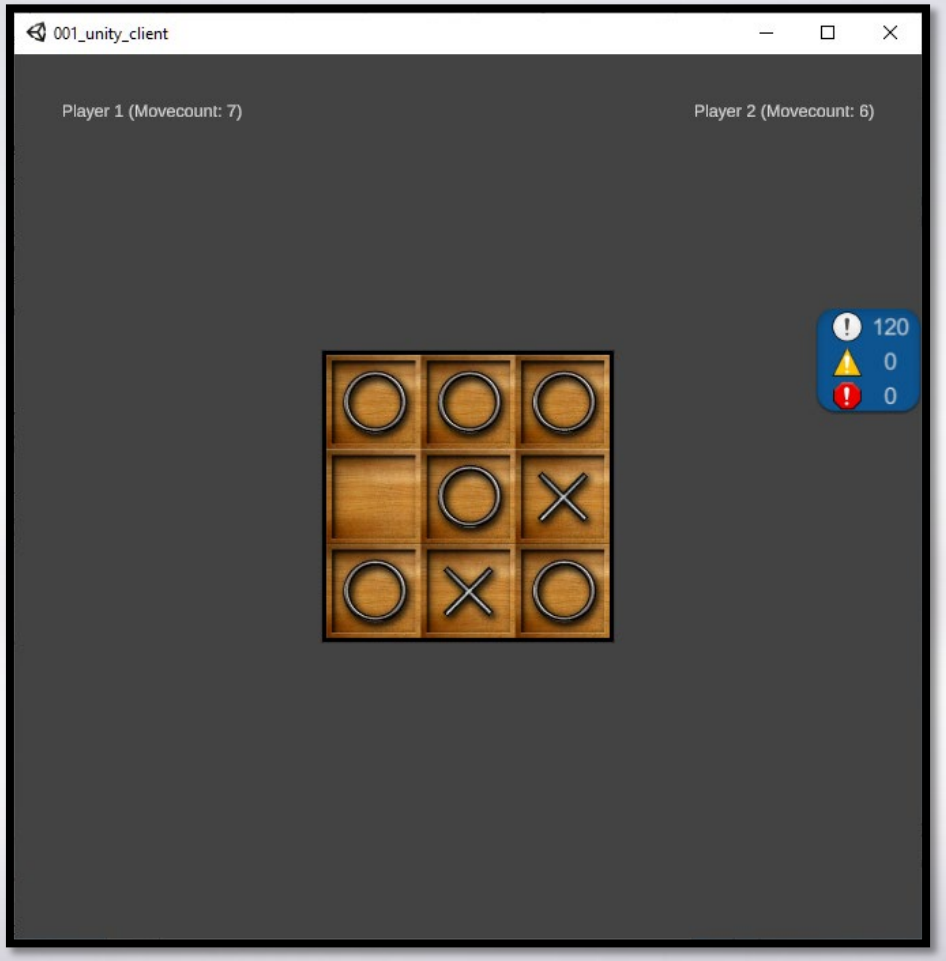
- Wrapper around a specific panel and all its components to hide its implementation details from the rest of the application and make life easier on you as an application programmer.

- Check out:
  - GameView
  - LobbyView
  - LoginView

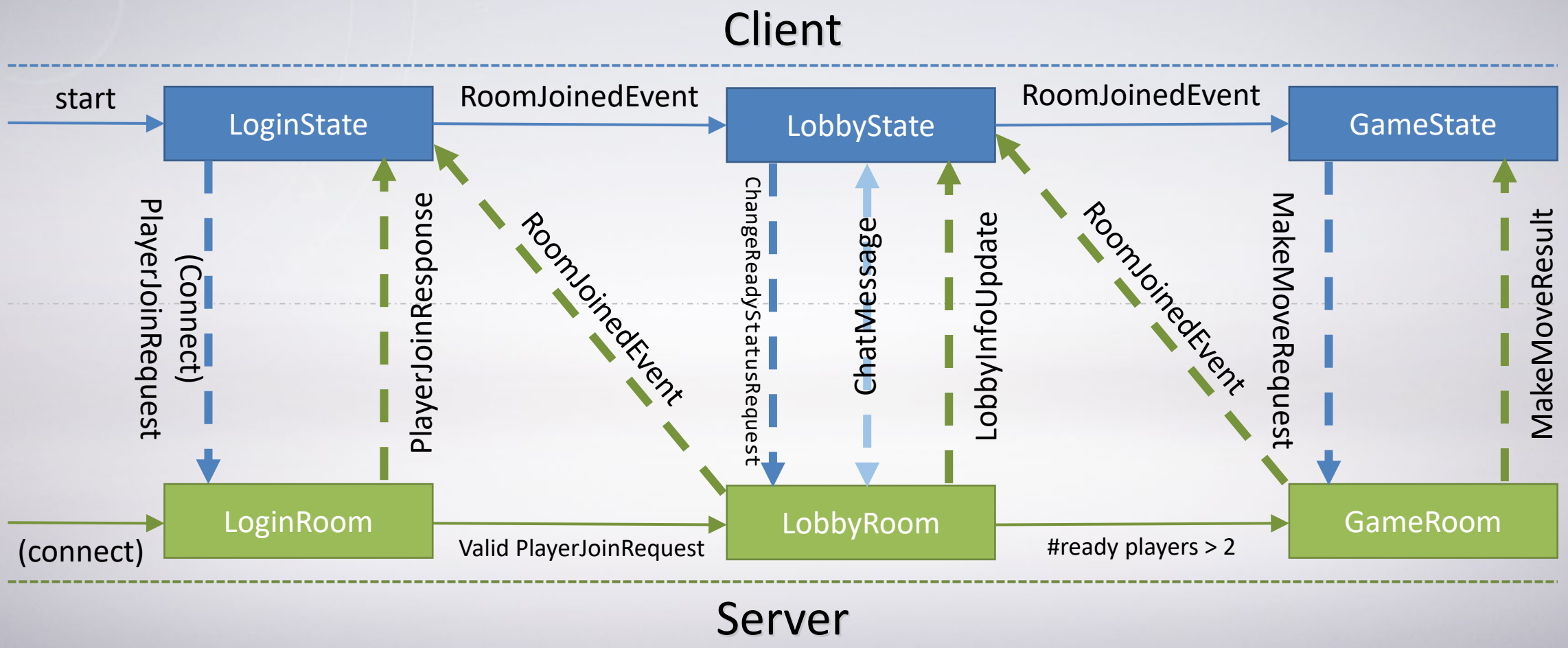
Client class diagram



Guided tour of the rest of the application



State/protocol flow sketch



Things to note (neat things)

- View selection in editor
- Log class with pretty colors ☺
- Console redirection
- In game debugger
- Documentation and notes

Last but not least: Boards & board data

- TicTacToeBoard → Server side wrapper of TicTacToeBoardData
- TicTacToeBoardData → Serializable board state client/server
- GameBoard → Client side UI class

Assignment 4 – Flac Flac Flo

To fix what is 'broken'

An recap of some broken things ...

- Only 1 game can be started
- Game never ends:
  - it's not really a game yet!
  - clients are 'stuck' in game forever
- Client names are not shown
  - Clients don't know who they are/who they play against
  - Clients don't know whose turn it is

Your task ...

- ... is to create a robust server for a turnbased multiplayer network game of your choice (Tic tac toe *is* allowed).
- All starting code + Getting started/requirements on blackboard

- It might sound like a lot to take in/do, but, keep in mind:
  - this is an assignment for the last ***3*** – ***4*** weeks: 3.6, 3.7, 3.8, 3.9 (assessment/deadline: Monday week 3.10)
  - 4 points of your total grade: excellent is basically 66% of all the other assignments' excellents combined
  - Expect 20-40 hours of work (based on 5-10 hours per week)