**Документация к тестовому заданию.**

**Ниже приведено краткое описание используемых скриптов, кто за что отвечает и что делает, мой подход по управлению завязан на статичных событиях public static Action, почти все взаимодействие происходит благодаря им, мы просто подписываемся на событие где нам нужно и где нужно их вызываем, если на событие что то подписано то что то произойдет если нет то ничего не произойдет. Зависимости минимум или почти нет, и никаких проверок на null, соответственно и ошибок. Данный подход неплохо себя показал при онлайн файтинге и гонках.**

В проекте две сцены, MenuScen она стартовая и GameScen игровая, прогрузка сцен идет по имени, но сцена меню должна быть первой в иерархии сцен

*Сцена меню*

Класс: **MenuControll**

Главный скрипт в сцене Меню, в нем можно изменять максимальное количество игроков, например если мы в будущем вдруг захотели бы сделать бой на 4 игрока.

Скрипт отвечает за выбор режима игры Один игрок/Два Игрока - **метод** SelectGameMod(), выбор персонажа каждым игроком – **метод** SelectPlayer(), все выборы происходят нажатием кнопок в меню, и передачей string, так же в скрипте прописана логика асинхронной загрузки новой сцены (просто лень было создавать новый класс). Так же включает отключает объекты пользовательского интерфейса **Cont\_GameMod, Con\_PlayerSelect, Con\_StartGame;**

События: **отсутствуют**.

Присутствие в сцене: **обязательно** (только в сцене меню)

-----------------------------------------------------------------------------------------------------------------------------------------

Класс: **SaveLoadSystem**

Система сохранение\загрузка, класс инициализируется при старте, можно всегда получить доступ **SaveLoadSystem.Instance,** все сохраняет в Json, использую для передачи данных между сценами. **Метод** - SaveGameData(bool is\_Online,string \_Player\_1\_Tupe,string \_Player\_2\_Tupe) сохраняет параметры, булевая один игрок или два(с ботом или против игрока), выбор первого игрока и выбор второго игрока(если не is\_Onlain, сохраняет бот). **Метод** –LoadGameData() загружает и присваивает параметры.

События: **отсутствуют**.

Присутствие в сцене: **обязательно** (во всех сценах)

-----------------------------------------------------------------------------------------------------------------------------------------

*Сцена игры*

Класс: **Netvorck\_Generic**

Основной скрипт по мультиплееру, вся логика взаимодействий в игре построена на статичных событиях, которые отправляют сообщение на сервер (стандартный РПС метод, в котором храним имя операции, и идентификатор), а с сервера (или мастер клиента) отправляется РПС метод всем игрокам, метод при получении клиентом вызывает событие которое определяет действие, сравнивается идентификатор и если он совпадает то действие true, если нет то false;

“например игрок1 нажимает атака, он вызывает событие **GameEvent.on\_SendActionToServer?.Invoke**(  
 **typeAction**, -тип атаки (например легкая или тяжелая)  
 **Scen\_Model.Instance.curentPlayer.attackDamage**, - урон  
 **Scen\_Model.Instance.curentPlayer.playerId**); - свой ID

В данном скрипте на это событие подписан метод **GameEvent.on\_SendActionToServer -= SendActionToServer** , который отправляет РПС на сервер, а с сервера возвращается всем остальным **RpcPerformActio(),** и вызывает событие на каждом клиенте, в классе **PlayerController** игрок сравнивает ид и если оно совпадает он атакует (например проигрывается анимация нужной атаки, легкая\тяжелая), если есть оружие или патроны им присваевается урон, в нашем случае игрок у которого не совпал id просто получает урон (так же он знает какой тип атаки был произведен и можно отыграть анимацию определенного получения урона, например отшатнулся или упал).

Если мы например выбрали барьер или регенерацию то игрок с совпавшим id будет активировать например барьер.

”

Так же в данном скрипте присутствуют методы готовности игрока которые так же отправляют РПС **NetPlayerReady(int i) и onReadyPlayer(int i).**

Тем что все взаимодействие по сити прописано в одном скрипте можно в любой момент поменять логику (например использовать Mirror или Photon), но меняя логику не стоит забывать что нужные компоненты на префабы игроков по типу PhotonView нужно добавлять и кишировать ручками отдельно на каждый префаб, так же спавн игроков (в данном случае он прописан в другом скрипте так как игра локальна но можно прописать и в этом или создать отдельный статический класс с методом прогрузки, но это было уже лень и небыло необходимости).

Подход с событиями хорошо показал себя при файтингах, когда сетевая синхронизация на стороне фотона прихрамывала, а тут по сути весь контроль на каждом клиенте идет через сервер, игрок по сути не управляет своим персонажем а отправляет РПС –шки на сервер. Так же этот подход неплох при шутерах например событие зажал кнопку двигаться вперед, и событие отпустил кнопку двигаться вперед, просто более обширный диапозон событий, единственное при шутерах придется синхронизировать поворот в реальном времени, но это специфика жанра.

При данном подходе мы не нагружаем сервер излишней синхронизацией, и по минимуму используем запросы к серверу.

Может есть подходы и лучше и проще, но данный по моему мнению пока что самый бережливый и производительный.

События:

**Подписан:**

**GameEvent.on\_SendActionToServer +=** SendActionToServer; **GameEvent.on\_NetRedyPlayer +=** NetPlayerReady;

**Вызывает:**

**GameEvent.on\_StartGame**

**GameEvent.on\_Actions**

**GameEvent.on\_PlayerReady**

Присутствие в сцене: **обязательно** (сцена игры)

PS: в данном скрипте не реализована сетевая логика(она вообще в проекте не реализована), все что нужно для сетевой логики я закоментировал, и прописал лишь в качестве примера. Написано для тех кто понимает.

-----------------------------------------------------------------------------------------------------------------------------------------

Класс: **InputControl**

В нем хранятся методы которые мы вызываем пользовательским интерфейсом, тип действия записывается и вызывается событие GameEvent.on\_SendActionToServer

События:

**Подписан:**

-

**Вызывает:**

**GameEvent.on\_ReadyAction**

**GameEvent.on\_SendActionToServer**

Присутствие в сцене: **обязательно** (сцена игры)

-----------------------------------------------------------------------------------------------------------------------------------------

Класс: **Scen\_Controll**

Основной класс игровой сцены, через него проходят основные события, так же он хранит префабы игроков, спавнит игроков **метод**- StartGame(), переключает игроков **метод** - ChangePlayer(), проверяет окончание хода **метод** - CheckProgress(), рестарт игры **метод** - ResetGame(),

События:

**Подписан:**

**GameEvent.on\_UpdateUI** += CheckProgress;  
**GameEvent.on\_StartGame** += StartGame;  
**GameEvent.on\_ReadyAction** += ChangePlayer;  
**GameEvent.on\_PlayerReady** += CheckReadyAllPlayer;

**Вызывает:**

**GameEvent.on\_NetRedyPlayer**

**GameEvent.on\_UpdateUI**

**GameEvent.on\_StartAction**

**GameEvent.on\_Winner**

**GameEvent.on\_Loser**

Присутствие в сцене: **обязательно** (сцена игры)

-----------------------------------------------------------------------------------------------------------------------------------------

Класс: **Scen\_Model**

Хранит данные сцены, имена и типы игроков, их стартовые позиции, какой тип игры, онлайн или офлайн, прогружает данные с **SaveLoadSystem.Instance,** класссереализуется **public static Scen\_Model Instance** и доступен в любом месте **Scen\_Model.Instance**

События: **отсутствуют**.

Присутствие в сцене: **обязательно** (сцена игры)

-----------------------------------------------------------------------------------------------------------------------------------------

Класс: **Scen\_Model**

Отвечает за визуал в сцене, должен активировать различные анимации и проигрывать звуки.

В данном случае я только меняю текст.

События: **отсутствуют**.

Присутствие в сцене: **обязательно** (сцена игры)

-----------------------------------------------------------------------------------------------------------------------------------------

Класс: **PlayerController**

-Связан с компонентом public PlayerModel \_model; private PlayerView \_view;

**- Зависимость** \_view = GetComponent<PlayerView>();

-PlayerModel – обьект добавляется руками в префаб так как это ScriptableObject

обьект

Отвечает за контроль на игроке, подписывается на события действия, принимает событие сравнивает id и производит действие, для визуализации вызывает методы из **PlayerView.**

События:

**Подписан:**

**GameEvent.on\_Actions += ActionPlayer;  
GameEvent.on\_StartAction += ActionReadyAllPlayer;**

**Вызывает:**

**GameEvent.on\_UpdateUI**

Присутствие в сцене: не**обязательно** (префаб игрока)

-----------------------------------------------------------------------------------------------------------------------------------------

Класс: **PlayerModel**

**Это ScriptableObject, создается в ассетах**

Хранит данные игрока (здоровье, урон и тп)

События: **отсутствуют**.

Присутствие в сцене: **обязательно** (на префабе игрока)

-----------------------------------------------------------------------------------------------------------------------------------------

Класс: **PlayerView**

**Зависимость –** его ищет PlayerController, необходим на префабе игрока

Для визуализации действий, просто методы с действиями которые выводят сообщение в консоль

События: **отсутствуют**.

Присутствие в сцене: **обязательно** (на префабе игрока)

-----------------------------------------------------------------------------------------------------------------------------------------

Класс: **GameData**

Побочный класс создан для сохранений, хранит только два string и булевую, ничего не делает

События: **отсутствуют**.

Присутствие в сцене: не**обязательно**

-----------------------------------------------------------------------------------------------------------------------------------------

Класс: **GameEvent**

Статичный класс который просто хранит все статичные события.

События: **отсутствуют**.

Присутствие в сцене: не**обязательно**