6) Составить небольшой анализ реализации задания (это касается части, которая уже была реализована до вас):

- Какие недостатки и недоработки вам удалось найти?

Мне удалось найти следующие недостатки и недоработки:

1. Использование публичных полей. Любой код вне класса может напрямую изменять и читать значения этих полей без контроля и проверок. (CustomersController, Customer, GameController)
2. Логические блоки в методе Update(): В методе Update() используются вложенные условия, что может делать код менее читаемым и трудно поддерживаемым. Рекомендуется вынести логику проверки наличия свободных мест в отдельный метод для улучшения читаемости и поддерживаемости кода. (CustomersController, Customer)
3. Присутствие магических числе в коде (CustomersController)
4. Неявное использование ресурсов. Использование метода Resources.Load подразумевает, что ресурс находится в специальной папке Resources, что не всегда удобно и эффективно. Лучше использовать более явные способы загрузки ресурсов, такие как инъекция зависимостей или загрузка во время выполнения( OrdersController).
5. Взаимосвязь классов: Класс CookingTimer имеет прямую зависимость от класса FoodPlace. Хотя это может быть приемлемо в некоторых случаях, это также может сделать код более сложным для поддержки и тестирования. Рассмотрите возможность использования интерфейса или делегатов для снятия прямой зависимости.
6. Методы и поля в классе Customer переполнены различными функциями, что делает его сложным для понимания и поддержки в дальнейшем. Разделение на более мелкие компоненты или использование паттернов проектирования может помочь сделать код более модульным и управляемым.
7. Поля, такие как CustomerImage, CustomerSprites, и TimerBar, помечены как null, что предполагает, что они будут инициализированы в редакторе Unity. Однако, в реальности, если эти ссылки остаются неинициализированными, это приведет к ошибкам NullReferenceException при запуске игры.
8. Использование transform.localPosition: Присвоение Vector3.zero для локальной позиции клиента может привести к нежелательным результатам в зависимости от структуры иерархии объектов в сцене. Возможно, более безопасным подходом было бы присвоение глобальной позиции или использование другого метода размещения клиента.
9. Использование строки для имени еды: Использование строки FoodName для определения типа еды может быть не очень безопасным, так как она подразумевает, что тип еды будет сопоставлен с именем. Это может привести к ошибкам при вводе имен или к несогласованности в коде. Лучше использовать более надежные способы идентификации типа еды, например, перечисление (enum) или ссылку на объект типа еды.
10. Обработка исключительных ситуаций: В коде не предусмотрена обработка возможных исключительных ситуаций, таких как неправильное использование методов или неверные входные данные. Добавление обработки исключений поможет повысить надежность кода.(Food Presenter).
11. Дублирование кода: Проверка \_place.CurFood.CurStatus != Food.FoodStatus.Cooked дублируется в условии if (\_place.IsFree || (\_place.CurFood.CurStatus != Food.FoodStatus.Cooked)). Можно вынести эту проверку в отдельный метод или переменную для повышения читаемости и уменьшения дублирования.
12. Нарушение принципа единственной ответственности: Метод TryPlaceFood в классе OrderPlace выполняет сразу несколько задач: проверку возможности добавления еды, попытку размещения еды на месте и обновление списка возможных заказов. Это усложняет код и делает его менее модульным.
13. Неэффективные операции LINQ: Например, в методе UpdatePossibleOrders неэффективно использовать Count() == 0 вместо метода Any().

- Какие решения вы считаете сомнительными или нежелательными в более крупных проектах (и почему)?

В более крупных проектах я считаю сомнительными следующие вещи:

1. Прямые ссылки на объекты вместо использования интерфейсов: Класс GameplayController напрямую ссылается на объекты типа MenuWindow, WinWindow и LoseWindow, что делает его более жестко связанным с конкретными реализациями окон. Вместо этого лучше было бы использовать интерфейсы или абстрактные классы для уменьшения связанности и повышения гибкости.
2. Излишняя зависимость от глобального состояния: Класс GameplayController сильно зависит от глобального состояния (например, состояния CustomersController и других окон), что делает его сложным для тестирования и поддержки. Более модульный подход с минимизацией зависимостей обеспечивает более чистую и понятную архитектуру.
3. Неприменение паттерна проектирования Singleton: Хотя есть попытка реализовать синглтон с помощью статического поля Instance, реализация не является потокобезопасной. Это может привести к проблемам с множественной инициализацией экземпляров в многопоточной среде. Более безопасным подходом было бы использование более известного шаблона Singleton, такого как double-checked locking или использование синглтона из пакета UnityEngine.
4. Использование XML в качестве формата конфигурации. Хотя XML может быть удобен для небольших объемов данных, для больших проектов это может стать неэффективным и неудобным для обслуживания. Рассмотрите использование более современных форматов, таких как JSON или YAML.
5. Неоптимальная производительность: При большом количестве элементов в DestPlaces цикл foreach будет выполняться до тех пор, пока не будет найдено подходящее место для размещения еды. Это может стать проблемой при большом количестве мест назначения.
6. Прямые зависимости между классами: Прямые зависимости между классами, как в данном случае между CookingTimer и FoodPlace, могут привести к проблемам при масштабировании проекта или его изменении. Это усложняет поддержку и тестирование кода. Вместо этого лучше использовать принципы инверсии управления, такие как внедрение зависимостей (Dependency Injection) или использование интерфейсов, чтобы уменьшить прямые зависимости и сделать код более гибким.
7. Некоторые решения, например, частые проверки в методе Update(), могут привести к неэффективному использованию ресурсов и снижению производительности. В более крупных проектах это может привести к проблемам с производительностью приложения. Лучше оптимизировать такие участки кода или использовать асинхронные методы, если это возможно.
8. Класс Customer содержит много функциональности, что делает его трудным для понимания и поддержки. В больших проектах важно разделять функциональность на более мелкие и модульные компоненты, чтобы упростить разработку, тестирование и поддержку.

- Как можно было бы доработать прототип, чтобы улучшить его производительность на очень слабых мобильных устройствах, не жертвуя качеством картинки и без кардинальных изменений в коде или устройстве сцены?

Для того, чтобы улучшить его производительность, можно сделать следующее:

1. Отказаться от многократного использования FindObjectsOfType: В методах Restart() и Play() происходит повторное выполнение операции поиска всех объектов типа AbstractFoodPlace с помощью FindObjectsOfType<AbstractFoodPlace>(). Это может быть дорогостоящей операцией, особенно если в сцене много таких объектов. Более эффективным подходом было бы сохранение ссылок на эти объекты в переменной во время инициализации и повторного использования этих ссылок при необходимости.
2. Вместо частого создания и уничтожения объектов в методе GenerateCustomer() можно использовать объекты пула, чтобы избежать накладных расходов на сборщик мусора.
3. Внутри методов, где часто используются ссылки на другие компоненты или объекты, такие как OrdersController.Instance, GameplayController.Instance и CustomerPlaces, эти ссылки можно кэшировать один раз при инициализации или при старте скрипта, чтобы избежать повторных запросов.
4. Предварительное вычисление значений: Если некоторые значения не изменяются во время выполнения, их можно предварительно вычислить и сохранить, чтобы избежать повторных вычислений.
5. Пакетная инициализация и обработка ресурсов: Если ресурсы часто используются вместе, их можно инициализировать пакетами, чтобы избежать частых обращений к хранилищам ресурсов.
6. Если Update() вызывается каждый кадр, это может быть излишним для проверки состояния Place.IsCooking. Вместо этого можно использовать события или другие механизмы обратного вызова, чтобы вызывать метод только при необходимости.
7. Использование раннего выхода из цикла: Если еда уже размещена успешно, нет необходимости продолжать итерацию по оставшимся местам. Выход из цикла сразу после успешного размещения улучшит производительность.
8. Уменьшение числа аллокаций памяти: Избегать создания временных объектов внутри циклов или методов, особенно если они вызывают сборку мусора. Вместо этого, использовать пулы объектов или переиспользовать существующие объекты.

- Где можно изменить числовые параметры: время ожидания посетителей, время готовки/сгорания блюд, количество посетителей?

Время ожидания посетителей (CustomerWaitTime) и количество посетителей (CustomersTargetNumber) можно изменить в скрипте CustomersController, который находится на объекте сцены [CustomersController].

Время готовки/сгорания блюда можно менять в скрипте FoodPlace, за это отвечают переменные CookTime и OverCookTime. В рамках данного прототипа этот скрипт находится на дочерних объектах <Название продукта>Pan(HotDogPan и BurgerPan соответственно) Place<Номер>. Сами объекты находятся на Canvas.

- Что нужно сделать, если будет необходимость реализовать в рамках этого прототипа еще одно блюдо (луковые кольца, жареные во фритюре) с новой механикой:

1) Игрок тапает по фритюрнице, начиная готовку

2) Фритюрница запускает таймер на пять секунд

3) По истечению таймера на стол автоматически выкладывается три порции луковых колец. (максимальное количество порций на столе – 3) (достаточно описать порядок действий, не обязательно реализовывать эту механику)

Для того, чтобы реализовать в рамках этого прототипа еще одно блюдо, а именно луковые кольца, жаренные во фритюре, нужно сделать:

1. Создать данный рецепт в файле Orders.xml
2. Разместить на сцене объект OnionRingsPun, который будет иметь следующие дочерние объекты: OnionRingsPlacer, Place0, Place1, Place2.
3. В OnionRingsPlacer, который будет представлять фритюрницу необходимо добавить следующие скрипты: FoodPlace и FoodTransfer. В FoodPlace необходимо установить в параметр Cook Time значение, равное 5, так как механика не подразумевает пережаривание, то параметр Overcook Time нужно выставить в 0.

В скрипте FoodTransfer нужно указать места, где будет появляться готовая еда, а именно Place0, Place1, Place2. Добавляем компонент EventTrigger и настраиваем его. Выбираем Pointer Up, Runtime Only, дальше нужно выбрать наш объект, а именно OnionRingsPlacer, и выбрать метод TryTransferFood. На OnionRingsPlacer накладываем спрайт фритюрницы, и добавляем в дочерние объекты CookingTimer

1. В CookingTimer добавляем одноименный скрипт и настраиваем его необходимым образом. В качестве параметра Place указываем OnionRingsPlacer. Для того, чтобы порции выкладывались автоматически на объект также необходимо поставить скрипт FoodServer
2. В Place0, Place1, Place2 необходимо добавить скрипты FoodPlace, Food Presenter. В Food Presenter мы указываем то, какие объекты мы будет визуализировать, а именно Raw и Coocked, и где мы будем это делать, поэтому каждый Place должен содержать 2 дочерних объекта Raw и Cooked, отвечающие за визуализацию «сырого» и «готового» состояний блюда. Для этого им необходимо прикрепить скрипты Food Visualizer и настроить их правильным образом.
3. В папке Prefabs/Orders создать префаб заказа, чтобы посетитель мог его заказать.