Unity / C# · Middle

Тестовое задание

Привет! Спасибо, что нашёл время. Ниже — короткое тестов

Внутри — 2 варианта, **выбери один**. Они равнозначны по сложности; выбирай тот, что ближе по душе. Делать нужно **с нуля**: создаёшь Unity-проект сам, базовые абстракции пишешь сам (это часть оценки).

Результат надо отправить до завтра 19:00 по Мск в Telegram @mattnastya.

Это домашний формат, поэтому не требуем идеальной точности по времени и жестких доказательств потраченного времени.
Если ты потратил больше/меньше 1,5 часов — это нормально, просто укажи это честно.
Для нас важнее прозрачность, чем попытка сделать идеально любой ценой.

Желаю удачи!

Технические требования (общие, обязательны)

- Unity **2022 LTS+**, C#.
- Разрешено подключать: **VContainer**, **UniTask**, **UniRx / R3** или своя реактивщина — на твой выбор. Опционально **Odin Inspector**, **DOTween**.

АРХИТЕКТУРА, В КОТОРУЮ НУЖНО ПОПАСТЬ

REACTIVE — UniRx/R3 или свой `ReactiveValue<T>` с сигнатурой `IDisposable Subscribe(Action<T> cb, bool invokeImmediately = true)`. Главное: каждая подписка должна корректно диспозиться, никаких голых `event Action` без отписки.

ASYNC — Все async-операции — **только UniTask** + `CancellationToken`. `async void`

запрещён везде, кроме Unity-колбэков.

DI **VContainer**. Все сервисы регистрируются как интерфейсы:

```
builder.Register<Impl>(Lifetime.Singleton).As<IInterface>();
```

Никаких `FindObjectOfType`, `Singleton.Instance`, `static` хранилищ состояния.

UI Своя пара `UIView : MonoBehaviour { Initialize(); Release(); }` + `UIView<TVm> : UIView where TVm : IUIViewModel`. **VM — обычный C#-класс, не** `MonoBehaviour`. Логика — в VM, View держит только Unity-ссылки и биндинги.

SERVICE Сервисы — `Service : IService` с асинхронным жизненным циклом:

```
UniTask InitializeAsync(CancellationToken)
UniTask ReleaseAsync(CancellationToken)
```

Все фоновые петли стартуют в `InitializeAsync` и гасятся через `CancellationTokenSource.Cancel()` в `ReleaseAsync`.

CONFIG Конфиги — `ScriptableObject` с суффиксом `*Settings`, регистрируются в `LifetimeScope` через `[SerializeField]` + `builder.RegisterInstance(_settings)`.

📦 Что сдать

- [ ] GitHub-репозиторий (или zip без `Library/`, `Temp/`, `obj/`).
- [ ] `README.md`: как запустить + 5–10 строк «что бы я доделал, будь у меня ещё 2 часа».
- [ ] `SELF_NOTES.md` — **обязательный** документ про твои собственные решения. Напиши своими словами:

  — Какие идеи ты рассмотрел и почему выбрал именно эту реализацию (а не альтернативы).
  — Какие места в коде ты придумал и написал сам, без AI, — и почему именно их.
  — Что в коде ты понимаешь до последней строки, а что осталось «магией» (это

нормально, но честно об этом скажи).
— Если бы кто-то завтра пришёл и спросил «почему здесь именно так?» — твой ответ на 2–3 ключевых решения.

- [ ] `AI_LOG.md` — какие промпты использовал, где AI ошибся, что переписал руками. Если AI не использовал — напиши, почему.

> ⚠️ Нам важно увидеть, что ты понимаешь свои решения, а не просто копируешь готовый ответ из AI.

🚫 Чего мы НЕ ждём

Красивого арта, анимаций, звука, мобильной сборки. Главное — слои, стиль и твоё понимание собственного кода.

🎯 Выбери вариант

A

«Boot Flow»

State machine из 3 стейтов

B

«Energy & Regen»

Реактивный сервис + UI

A Вариант A — «Boot Flow» state machine из 3 стейтов

Сделай загрузочный поток приложения через свою стейт-машину.

1 Базы стейт-машины

Свои `IState`, `IStatesController<TEnum>`, `StatesController<TEnum>` с методом `UniTask EnterStateAsync(TEnum code, CancellationToken ct)`. Контракт: сначала `await currentState.ExitAsync(ct)`, потом `await newState.EnterAsync(ct)`. `CancellationToken` пробрасывается насквозь и реально отменяет внутренние

ожидания.

2 **Три стейта**

SplashState — показывает лого, ждёт 1 с через `UniTask.Delay(..., ct)`, переходит дальше.

LoadState — имитирует загрузку (5 шагов по 200 мс), на стейте лежит `ReactiveValue<float> Progress` (0..1), который обновляется после каждого шага.

MenuState — показывает `MenuUIView` с одной кнопкой «Restart», по клику возвращает в `LoadState`.

3 **UI**

`LoadingUIView` подписан на `LoadState.Progress` и двигает прогресс-бар (DOTween или ручной Lerp — без разницы). При `ExitAsync` все подписки чисто диспозятся, никаких NRE при повторном входе.

B Вариант B — «Energy & Regen» реактивный сервис + UI

Сделай систему энергии для мобильной игры.

1 `EnergySettings` **(ScriptableObject)**

`MaxEnergy` (int), `RegenSeconds` (float, сек на 1 единицу). Создать ассет и заинжектить в `LifetimeScope`.

2 `IEnergyService` / `EnergyService : Service`

Поля: `IReadOnlyReactiveValue<int> Current`, `IReadOnlyReactiveValue<float> SecondsToNext` (доля до следующей единицы, 0..1). Методы: `bool TrySpend(int amount)`. Регенерация — фоновая UniTask-петля внутри сервиса, стартует в `InitializeAsync`, корректно гасится в `ReleaseAsync` через свой `CancellationTokenSource`.

> ⚠ Если `Current == Max`, петля «спит» эффективно (не крутит `Delay(0)` в while).

3 **UI**

`EnergyBarUIView<EnergyBarUIViewModel>` показывает: цифру `current / max`, прогресс-бар `Image.fillAmount = SecondsToNext`, кнопку «Потратить 10». Биндинги — через `ReactiveValue.Subscribe(...)`, отписки — в `Release()`.

> 🚫 **Запрещено** обновлять UI через `Update()`.