



UI PROGRAMMER

UBISOFT TEST



HELLO!

I am Joan Ginard

I'm here because I love to make games. My life's quote is: "Genius is 1% talent and 99% hard work".



TABLE OF CONTENTS

1. Project Structure
2. Project Extras
3. Difficulties Encountered
4. Final Words



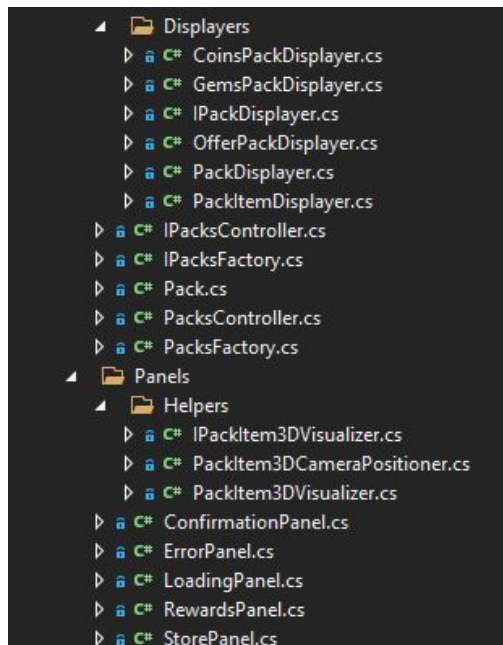
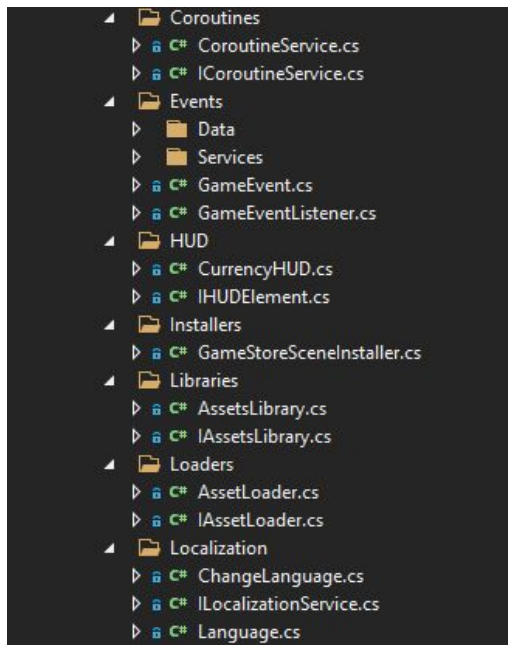
1.

PROJECT STRUCTURE



PROJECT OVERVIEW

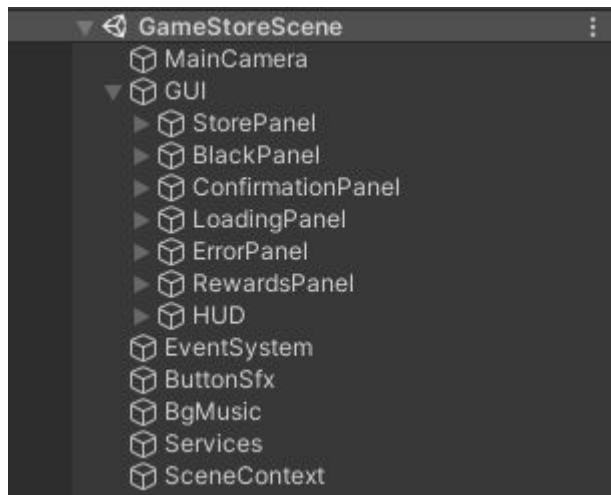
- Respecting the **single responsibility principle**





SCENE HIERARCHY

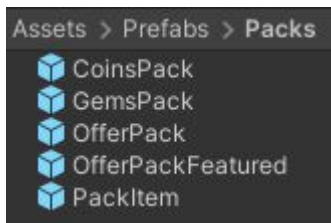
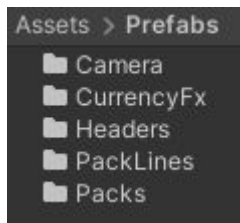
- Clean structure for easy modification





PREFABS MANAGEMENT

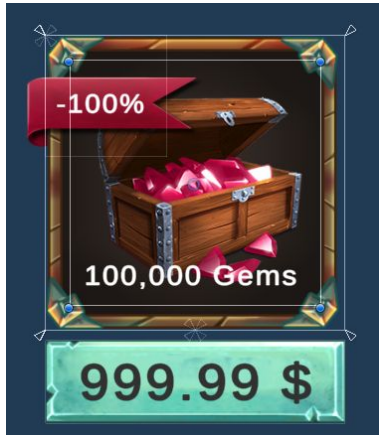
- **Prefabs** are the key to avoid scene conflicts and for designers and artists to test/make updates without interfering in the programmers work!





STORE BUILDING PROCESS

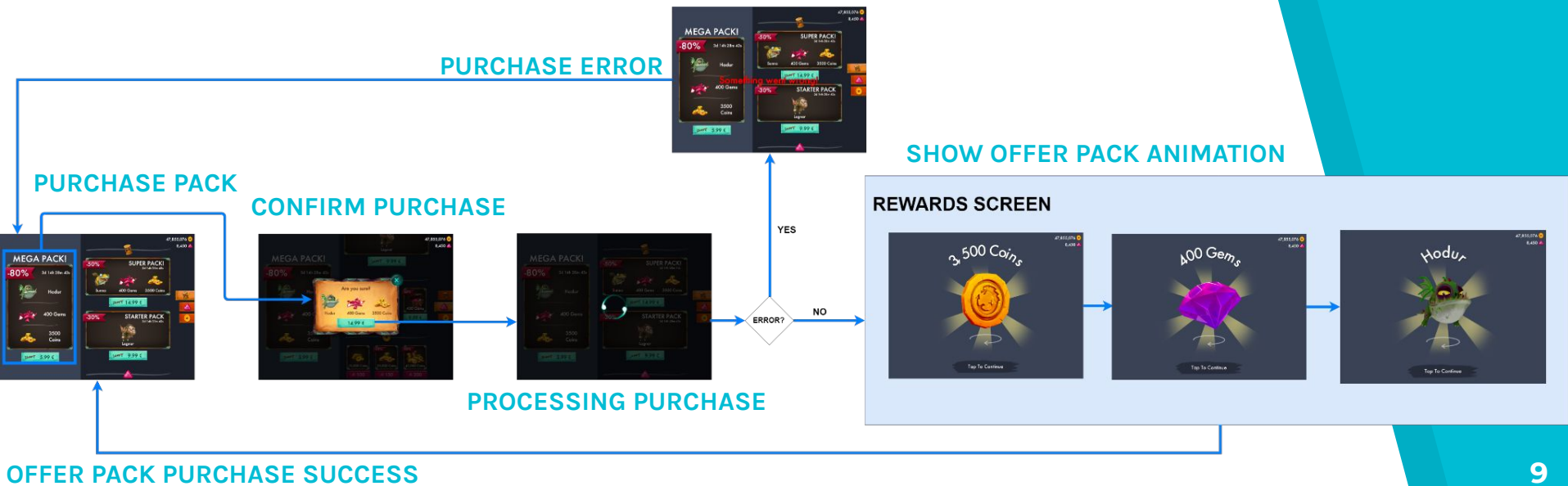
- For starters, prepare the assets adjusting **anchors** and **pivots** for compatibility between multiple mobile devices resolutions





STORE BUILDING PROCESS

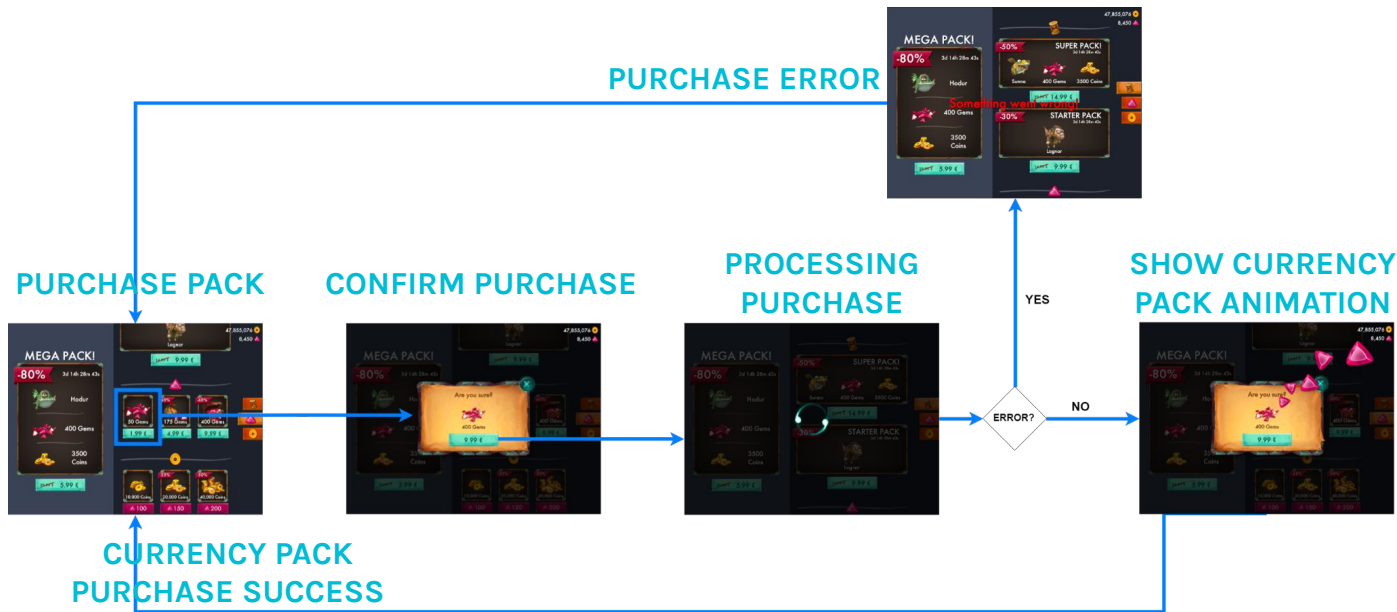
- Then, make an **events** system to go from one step to another of the flow and code the API





STORE BUILDING PROCESS

- Then, make an **events** system to go from one step to another of the flow and code the API





WHY USE EVENTS? BECAUSE...

- They make your code **decoupled**
(“Yay, I hate dependencies!”)
- It lets you **adapt to change** and customize the flow (which we know will happen)
- You can send an event from **anywhere** in the code, it’s magic! (e.g. Send “Refresh Health Bar” event on player hit.)
- It’s **easy** to implement



MY ROUTINE WORK PROCESS

UNDERSTAND REQUIREMENTS



MAKE IT WORK



CLEAN AND OPTIMIZE IT



2.

PROJECT
EXTRAS



PROJECT EXTRAS

- Animations
- Visual FX
- Sound FX
- Language Change Option
- Events System
- Dependency Injection
- Unit Tests





DEPENDENCY INJECTION

Why I'm not a big fan of **singletons**?

- Because you are **limited** to one instance
(e.g. Imagine that in January the requirements are to have 1 avatar replayer, but then in May they change to 3 avatar replayers per scene, you are screwed my friend...)
- Because they are **public** to everyone
(This can lead to misuse, specially for junior devs.)
- Because they are **immutable**
(e.g. I have a singleton that does X, but on some cases I want to do Y, so I have to constantly change it.)
- Refactoring a singleton is **painful**
(If you need to manually change every “.Instance” in your code base for another class “.Instance”, it could take years to complete the refactor...)



DEPENDENCY INJECTION

What do I prefer to do instead?

- Program to an interface and use a **dependency injection** framework
(either code your own or use an existing one)



Singletons



Dependency
Injection



DEPENDENCY INJECTION

✗ NOT RECOMMENDED WAY

```
Ⓜ UnityScript (1 asset reference) | 0 references
public class Player : MonoBehaviour
{
    0 references
    public void OnPlayerHit()
    {
        // immutable implementations
        EventSystemManager.Instance.TriggerEvent("PlayerHit");
        AudioManager.Instance.PlayAudioClip("PlayerHitFx");
        // ...
    }
}

Ⓜ UnityScript (1 asset reference) | 0 references
public class Button : MonoBehaviour
{
    0 references
    public void OnPress()
    {
        // immutable implementation
        AudioManager.Instance.PlayAudioClip("ButtonPressFx");
    }
}
```



DEPENDENCY INJECTION



RECOMMENDED WAY

```
UnityScript (1 asset reference) | 0 references
public class Player : MonoBehaviour
{
    [Inject] private IEventsSystemService _eventsSystemService;
    [Inject] private IAudioPlayingService _audioPlayingService;

    0 references
    public void OnPlayerHit()
    {
        _eventsSystemService.TriggerEvent("PlayerHit");
        _audioPlayingService.PlayAudioClip("PlayerHitFx");
        // ...
    }
}
```

```
UnityScript (1 asset reference) | 0 references
public class Button : MonoBehaviour
{
    [Inject]
    private IAudioPlayingService _audioPlayingService;

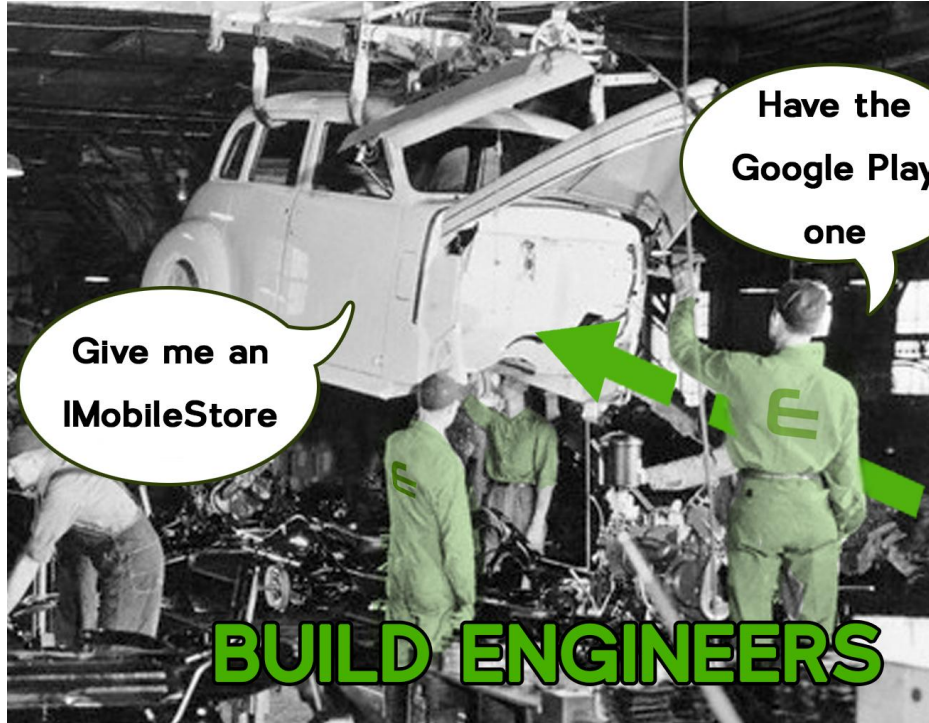
    0 references
    public void OnPress()
    {
        _audioPlayingService.PlayAudioClip("ButtonPressFx");
    }
}
```

```
UnityScript (1 asset reference) | 0 references
public class DependencyInjectionInstallerExample : MonoInstaller
{
    9 references
    public override void InstallBindings()
    {
        // With just one line of code you can change all the dependencies from the code base!
        Container.Bind<IEventsSystemService>().To<DefaultEventsSystemService>().AsSingle();
        Container.Bind<IAudioPlayingService>().To<AudioStorePlayingService>().AsSingle();

        // You can do cool stuff such as:
        #if UNITY_EDITOR
            Container.Bind<IMobileStore>().To<TestMobileStore>().AsSingle();
        #elif UNITY_ANDROID
            Container.Bind<IMobileStore>().To<GooglePlayStore>().AsSingle();
        #elif UNITY_IOS
            Container.Bind<IMobileStore>().To<AppleAppStore>().AsSingle();
        #endif
    }
}
```



DEPENDENCY INJECTION



IMobileStore Implementations:



AmazonAppStore



AppleAppStore

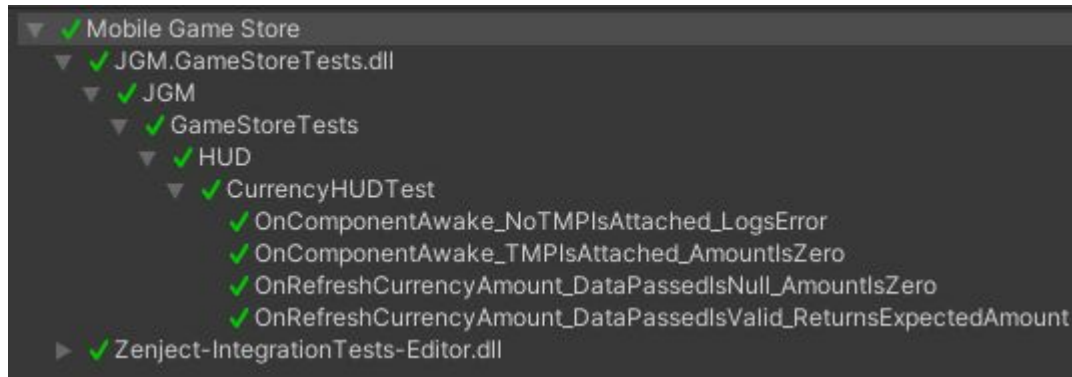


GooglePlayStore



UNIT TESTS

- TDD approach is always a good habit





3.

**DIFFICULTIES
ENCOUNTERED**



DIFFICULTIES ENCOUNTERED

- **Time** has been the biggest issue, as I have a full-time job during the week and had a birthday celebration on the weekend, which didn't help...





4.

FINAL WORDS



FINAL WORDS

- Doing the test has actually been **super fun**! Hours went by and I didn't even notice! Hehe
- I wish I had more time to **polish** the code more (remove duplicated code, open-closed principle, polish prefabs, etc.)
- But, one thing is for sure, with **passion** and **hard work** coding any feature is possible!



THANKS!

Thank you for your time.

Have a great day!