Purify

Technical Design Document

Table of contents

Code Structure

Naming Convention

Kind	Rule	Example
private field	prefix + lowerCamelCase	m_previousEnergy
public field	UpperCamelCase	Player
protected field	UpperCamelCase	Owner
property	UpperCamelCase	CurrentTarget
local variable	lowerCamelCase	returnValue
parameter	lowerCamelCase	target
enum value	UpperCamelCase	Rare, Common,Uncommon
method names	UpperCamelCase	OnHover

Structure

Using the default C# code file / programm file structure.
See here

Game States

One part of the core module, that explains how the turns and phases works.
The Game consists out of 3 Game States. Each GameState takes care of specifics things, Timing and Events.

GameState is the base class and consists out of some methods

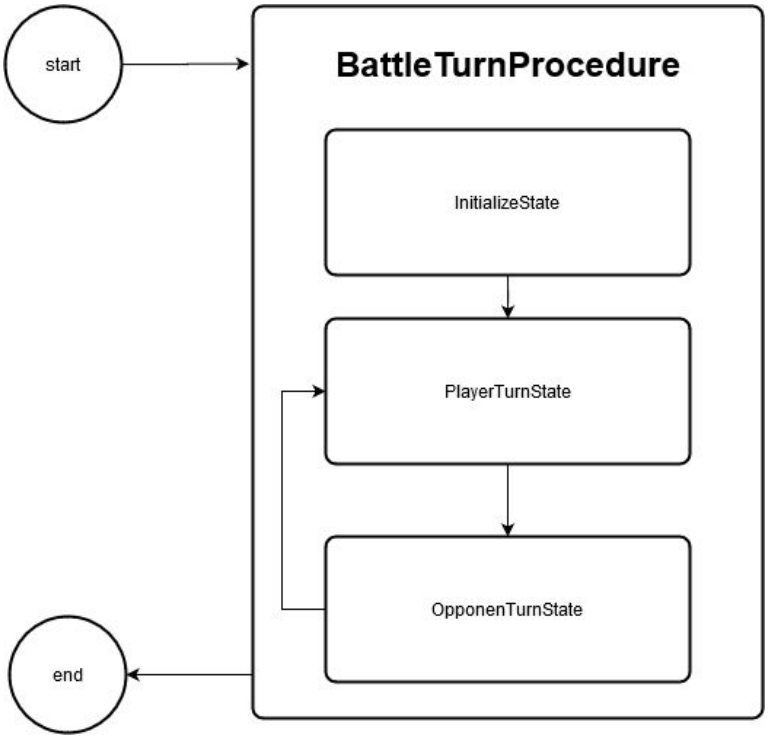
- *Execute()* handles the execution order of the following methods
- *Start()* at first and only once
- *Update()* called frequently
- Finish() at last and only once

InitializeState prepares a Battle, it creates a Deck, Enemies and the Player.

PlayerTurnState has different phases, each one takes care of a specific process.
For more information you can take a look at the code or Purify GDD.

OpponentTurnState starts after the PlayerTurn and handles each Enemy.

In **BattleTurnProcedure** all 3 game states comes together and are handled properly.



(Flowchart)

Effects

For our cards we need alot of effects and a simple to use interface to create new effects easily.

There are 2 different types of effects.

Condition Effects :
These are only used for Cards to create a special play oder discard condition that has to be met.

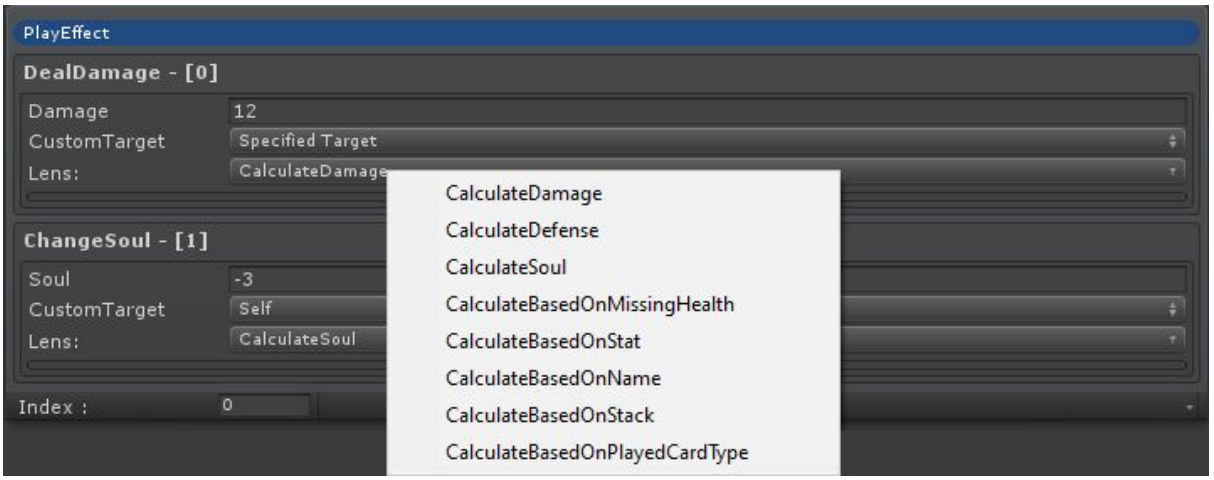
Effects :
Effects are used for cards and enemy Attacks.

```
public void Use(Unit target, Unit from, TargetType targetType)
{
  DefineExecution(target, from, targetType);
}


protected virtual void DefineExecution(Unit target, Unit from, TargetType targetType)
{
  BattleInfo.SolveTargetSelection(target, targetType, from, Execute);
}

public abstract void Execute(Unit target, Unit from);

protected int UseLens(Unit unit, Unit target, int value)
{
  if (unit == null || Lens == null)
  {
     return value;
  }

  return Lens.Calculate(unit, target, value);
}

public abstract object Value(Unit from, Unit target);
```

To define different execution just override *DefineExecution.*
Execute contains content that fullfill an effect.
UseLens uses lenses to calcualte values, these values can be returnd in *Value.*
Value are used to retrvieve Effect information like values.

Lens

Lenses are a common used feature in Functional programming.
But fits perfect for a problem we had during development.
Values from card effects needs to be manipulated based on specific situation.
Instead of code duplication, the Game Designer can choose another lens for a Effect.
The Lens will manipulate the value the Effect is using.




```
public class CalculateSoul : Lens
{
  public override int Calculate(Unit unit, Unit target, int value)
     => StatsFormula.CalculateSoul(value, unit);
}
```

Status (Buffs/Debuffs)

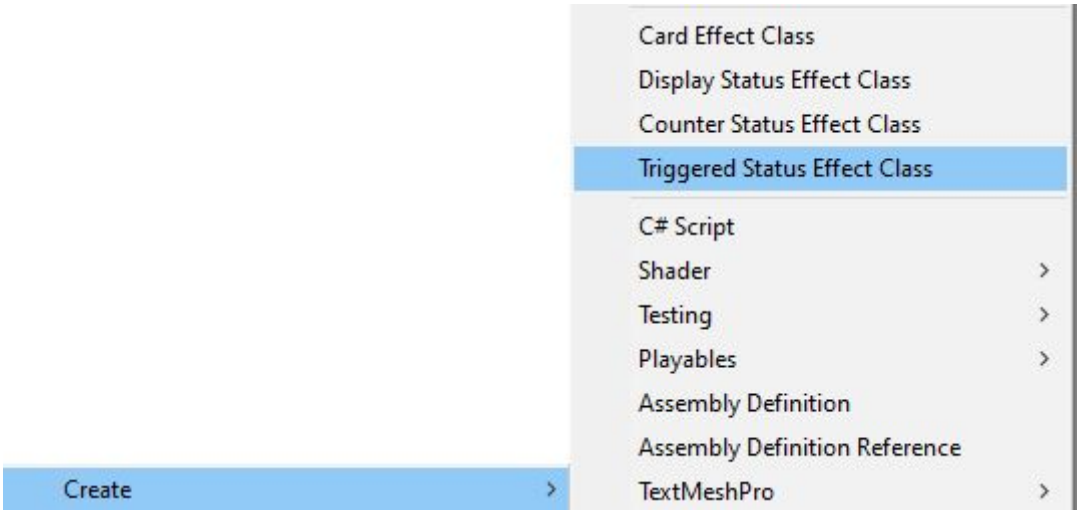
Status primarily are split into 2, data and behaviour. StatusData as the base class for all, contains information for the status behaviour. It also contains attributes for serialization/deserialization.

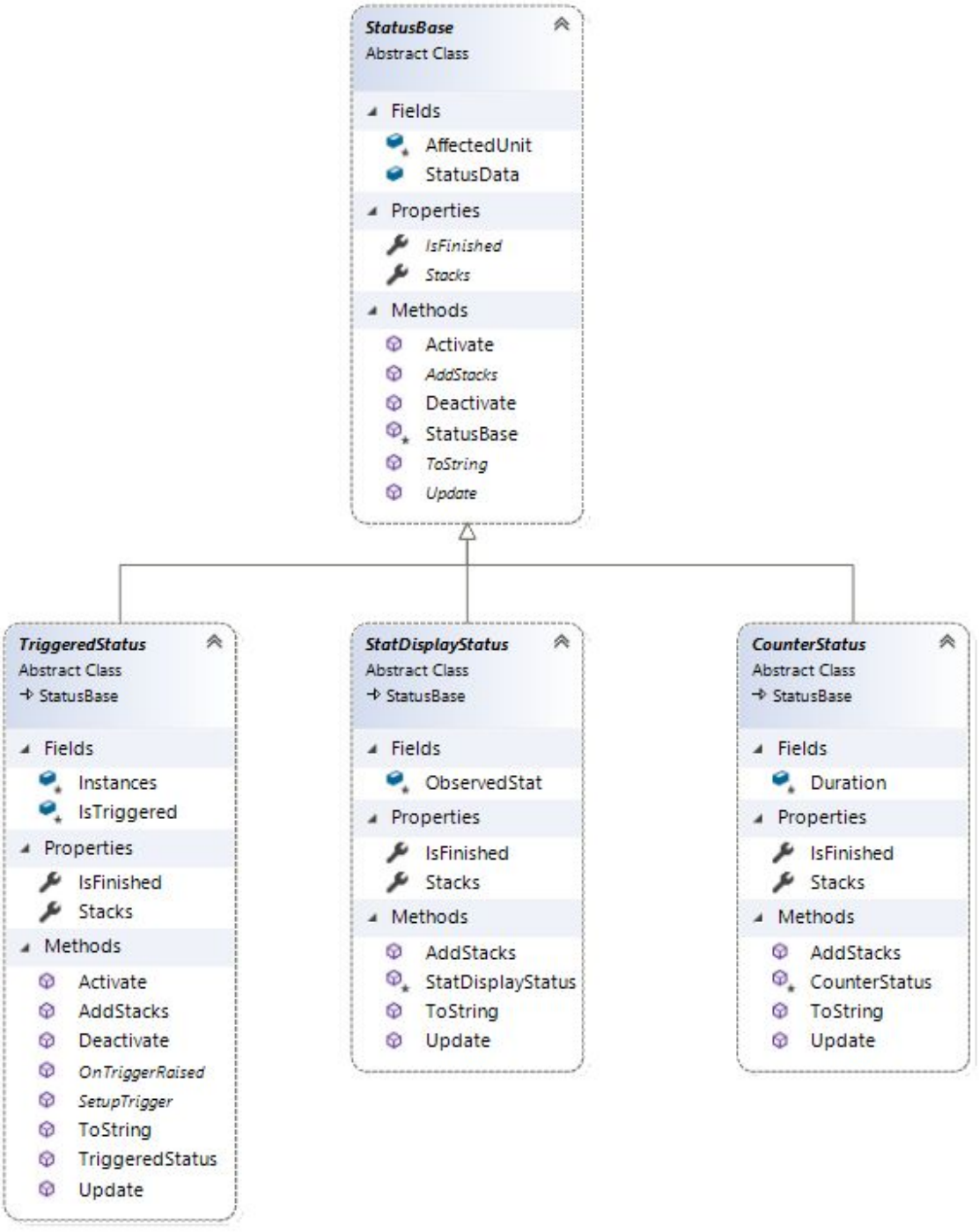
StatusBase is base class but there are 3 different classes that create a interface for better useability.

CounterStaus is used as view for stats, it only reads and displays the stats values.
TriggeredStatus is used for *TriggeredActions* in combination with the *EventLog.*
CounterStatus is used for status that tic's e.g each round.

To minimize the error rate, there is a way to create Templates. These Templates contains a minimum of code foreach Status.This helps to use the correct methods.

Simply right click in Project tab, go on create and pick one.




StatusBase
Abstract Class
Fields
AffectedUnit
StatusData
Properties
IsFinished
Stacks
Methods
Activate
AddStacks
Deactivate
StatusBase
ToString
Update
TriggeredStatus
Abstract Class
StatusBase
Fields
Instances
IsTriggered
Properties
IsFinished
Stacks
Methods
Activate
AddStacks
Deactivate
OnTriggerRaised
SetupTrigger
ToString
TriggeredStatus
Update
StatDisplayStatus
Abstract Class
StatusBase
Fields
ObservedStat
Properties
IsFinished
Stacks
Methods
AddStacks
StatDisplayStatus
ToString
Update
CounterStatus
Abstract Class
StatusBase
Fields
Duration
Properties
IsFinished
Stacks
Methods
AddStacks
CounterStatus
ToString
Update

Serialization/Deserialization

We are using [Messagepack] to serialize / deserialize data.
Only a few steps are necessary to do this. Classes need to be marked with

```
[MessagePackObject( keyAsPropertyName: true)]
30 usages  Oliver Sradnick  4 exposing APIs
public class CardData
```

abstract classes needs to be marked with attributes that describe the inheritors.

```
[Union( key: 0, typeof(AttackBuffData))]
8 usages  1 inheritor  Oliver Sradnick  2 exposing APIs
public abstract class BuffData
```

In some cases custom Formatters have to be written.
Sprites are not supported by MessagePack but can be simply implemented.
In combination with AssetDatabase and Resource Loader it fits perfect our needs.

```
1 usage  Oliver Sradnick
public sealed class SpriteFormatter : global::MessagePack.Formatters.IMessagePackFormatter<global::UnityEngine.Sprite>
{
    0+38 usages  Oliver Sradnick
    public int Serialize(ref byte[] bytes, int offset, global::UnityEngine.Sprite value, IFormatterResolver typeResolver)
    {
        var path = AssetDatabase.GetAssetPath(value);
        if (path != string.Empty)
        {
            path = path.Replace( oldValue: "Assets/Cards/Resources/",  newValue: "");
            path = path.Substring( startIndex: 0,  length: path.IndexOf("."));
        }

        return MessagePackBinary.WriteString(ref bytes, offset, path);
    }

    0+42 usages  Oliver Sradnick
    public global::UnityEngine.Sprite Deserialize(byte[] bytes, int offset, IFormatterResolver typeResolver, out int readSize)
    {
        var path = MessagePackBinary.ReadString(bytes, offset, out readSize);
        var sprite = Resources.Load<global::UnityEngine.Sprite>(path);
        return sprite;
    }
}
```

PersistentJson.cs and ***PersistentData.cs*** are wrapper classes to provide a simple interface to load/save data to binary for e.g save games or json files e.g card pools.

Cards, card pools, enemy behaviour and the starter deck definition are saved as json.
For all of them i created a Editor so there is no need to directly edit the json file.

Animation

To simplify the creation of Animation, we are using DoTween .
Creating sequences to create big animations or a simple one with a few commands.

To give some accessibility without changing the code there are always some fields to change in the Inspector, like the duration, target position,rotation,scale and the easing type.

```
m_onEnter = DOTween.Sequence();

m_onEnter.OnStart(() =>
{
  m_transform.eulerAngles = Vector3.zero;
  m_transform.localScale = Vector3.one * m_scaleFactor;
  m_isPlaying = true;
});

m_onEnter.Join
(
  m_transform.DOMove(GetHoverPos()
                , m_duration)
     .SetEase(EaseType)
);

m_onEnter.OnComplete(() =>
{
  m_keywordHandler.EnableKeywords();
  m_isPlaying = false;
});

m_onEnter.Pause();
```

Dependency Injection

As a Dependency Injection we used Extenject .
This help us to couple the code base loosely, see *BattleInstaller.cs* .
In most cases it is used to inject stuff in the Battle scene.

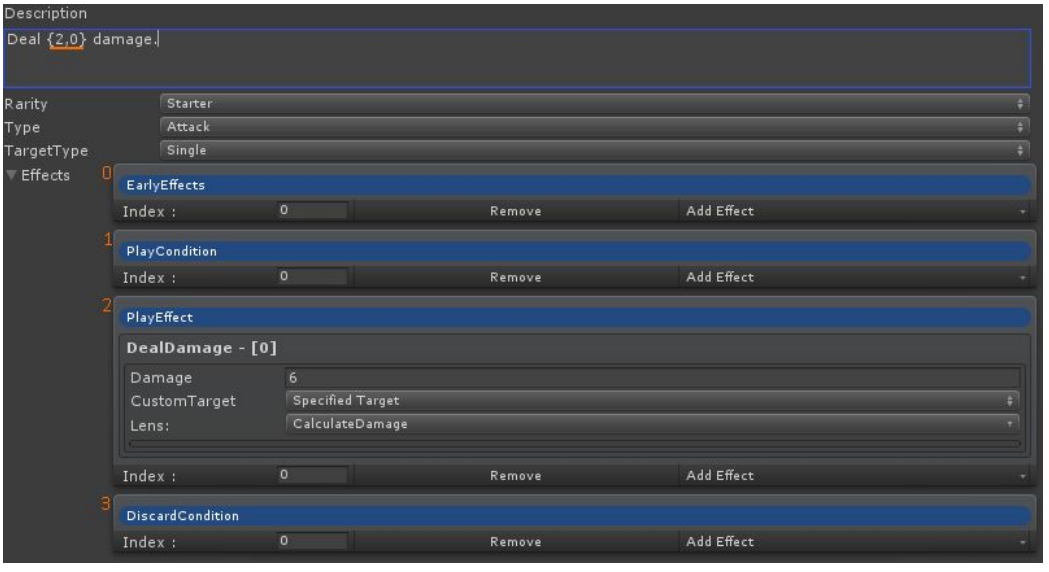
```
Container
  .Bind<Player>()
  .FromComponentInHierarchy(true)
  .AsSingle();


Container
  .Bind<General.BattleTurnProcedure>()
  .FromComponentInHierarchy(true)
  .AsSingle();

Container
  .Bind<RestMenu>()
  .FromComponentInHierarchy(true)
  .AsSingle();
```

Parsing

Card Description

To link a value from a Card effect there is a simple code that will be replaced with the appropriated value.
Using `{x,y}` in a description will be replaced.

Where x defines what kind of Effect (EarlyEffect,PlayCondition,PlayEffect,DiscardCondition) and y defines at which index the effect is.



Keywords

Keywords are colored words that explain a mechanic.
To each Keywords belongs a description.

To define a Keyword look at the ScriptableObject KeywordList, it contains all used Keywords.
These are pased if you enter in a card description `[n]` where n is the keyword from the list.

Keywords are parsed with the *KeywordParser.cs.*

Naming Convention

Folder Naming:

We use UpperCamelCase. That means all words are written together, each word starts with a capitalized letter.

Examples:

- Buffs
- Cards
- Deck
- Units

Asset Naming:

We use **UpperCamelCase** for Assets. The first letter of a word is always capitalized.

Examples:

- VFXHit
- PlayerPortrait
- CardTemplate

- MatchBackground
- IconEffectAttack

Scripts: (based on code architecture)

- GameState
- CardPile, DrawPile
- PersistentData

Tip / Hint:

→ DescriptiveClass_Variation
→ DescriptiveClassSubClass_Variation
→ ClassDescriptive_Variation
→ ClassDescriptiveSubClass_Variation
→ ClassDescriptiveSubClass
→ ClassDescriptive

Version control

General

Version control will be done with Mercurial and TortoiseHg as graphical interface.
The repository can be used to push all directly related project things. A good structured repository will improve the teams' efficiency.

Commits

There are certain things you have to follow before you push a Commit !

- No **unnecessary file changes** e.g personal settings, changes to Game Objects you don’t need for this commit
- check out your file size, big files waste our repository storage
- every pushed file will be in the repository, don't **push carelessly**
- consider the correct **naming convention**
- use the **correct branch**
- encountering a **merge conflict**, talk to the person / department that is responsible for it

Commit Messages

There are 2 headlines Added, Updated and **done**.

added

Is for things you just put in the project, things you just started to working on.
updated
For things that maybe changed frequently, restructuring folder, renamed files, script changes, prefab changes.

```
added:
- background music
- weapon model xx

updated:
- player movement speed
- player turn speed
- Lobby Room Panel, can now be used as a button

done:
- Pistol Projectile VFX
- background music volume
- player movement script
```

Branches

Branches should be named accordingly to the feature you are working on.
UpperCamelCase is our naming convention.

Consider **splitting your working directory** to put **each feature on its own Branch**.

Be aware of **Merge Conflicts** if two or more people **working on the same file,** Scene, Prefab, Material and so on.

Example :
VFX
MainMenu
Cards
CardPileUI

Branches can be merged if a feature is done and complete, consider to ask for a second opinion to check if your branch is cleaned up before you **merge with the default branch.**

Folder Structure

Using the Object Oriented approach for structuring the folder Hierarchy.
In some places a Resources Folder is placed, this is necessary to load data.

- Assets
  - Buffs
    - General
    - Trigger
    - Types
    - Resources
      - Icons
  - Cards
    - Effects
    - Filter
    - Resources
      - Icons
      - Illustration
  - Deck
  - Stats
    - General
    - Types
    - Environment
  - Utilities
  - Units
    - Enemy
      - Prefabs

- UI
- General
- Player
- General

Technical Information

We are using :

- Unity 2019.2.8f1
- Tortoise Hg
- JetBrains Rider
- Slack