

## ECS Draft

### Beginner mistakes

**Using inheritance for components and using interfaces is a very common beginner mistake. Why is it bad for development?**

- **No benefit.** Abstraction at the component level gives you no real gain compared to ECS-style abstraction through data (I'll cover that next).
- **Adds limits.** It makes components harder to extend and the related logic harder to improve.
- **Leads to logic in components.** You start putting behavior inside a component, which breaks ECS rules.
- **Creates extra complexity.** You feel forced to inherit systems or write big switch/case blocks by type for no good reason.

One note: inheriting **systems** is not always a good idea, but it's not harmful by itself, unlike inheriting **components**. So if you feel the urge to inherit components - don't. Stop and think of another way to solve the problem.

---

### Not using ECS-style abstraction.



ECS abstraction means we put shared data (the stuff you'd "inherit" in OOP) into a separate component.

How to make a "child" of that data? Simple:

- keep the **BaseComponent** with the shared fields,
- add a new **InheritorComponent** with the extra fields you need,
- filter entities that have both **BaseComponent** and **InheritorComponent**.

That's it. If you notice common data across components or entities, you can almost always move it into its own component - and the earlier you do it, the better.

One of the most flexible ways to design components is the **atomic component** approach. In this style, each component has only one field. It takes practice to master, but once you do, it gives you a very flexible way to write code. Example:

```
1 //  BAD
2 [Game]
3 public sealed class Character : IComponent
4 {
5     public Vector3 position;
6     public Vector2 inputDir;
7     public float moveSpeed;
8     public bool isFrozen;
9     public bool isSprinting;
10 }
11
12 //  GOOD
13 [Game] public class Position : IComponent { public Vector3 Value; }
14 [Game] public class MoveDirection : IComponent { public Vector2 Value; }
15 [Game] public class MoveSpeed : IComponent { public float Value; }
16
17 // Tag components (no fields)
18 [Game] public class Frozen : IComponent { } // presence = frozen
19 [Game] public class Character : IComponent { }
20 [Game] public class Sprinting : IComponent { }
```

---

## Turning ECS into an absolute

Only Sith deal in absolutes!

Remember: OOP is not forbidden. Using a hybrid of OOP and ECS is a common practice.

When you work with ECS, don't push everything into it. That can be counterproductive. Some data structures don't map well to ECS. In those cases don't stress - just write a separate OOP class.

You can build certain parts (for example UI) outside of ECS in whatever way is comfortable, and then connect them to ECS with a small bridge. Also, helper tasks - loading configs, direct network calls, writing saves are often easier as OOP services that systems call directly, instead of forcing them into ECS.

Use common sense. ECS should help your development, not get in the way.

---

## Trying to port existing code to ECS word for word

Beginners often try to move their existing code to ECS line by line. That's not a good idea, because coding in ECS is different from traditional programming. The usual result is frustration with ECS and a very awkward codebase.

If you do need to move old code to ECS, the best way is to **rewrite the logic from scratch in ECS**. Use your knowledge and the old code as a guide for what the feature should do not as something to copy.

---

## Using delegates/callbacks or reactive logic inside systems

In ECS it can be risky to capture system logic and store it in a component for later, or to react immediately to changes (for example, one system reacting the moment another adds a component). This adds tight coupling between systems - they start to depend on outside calls and it breaks the clean data-processing pipeline by adding logic that we don't fully control when it runs.

A better option is to use **deferred reactivity**, which I'll explain in [Best Practices](#).

---

## Sorting files by type

When you start with ECS it's tempting to sort files by type: put components in a **Components** folder and systems in a **Systems** folder. With experience, you see this is not very effective. It's hard to navigate and hard to see which components relate to which systems.

A better way is **feature-based** folders. Put everything for one feature in one folder (you can keep a small inner split like components/systems). For example, all components and systems for health and damage go in a **Health** folder. This lets you see the main context of the data and systems at a glance and makes the project easier to navigate.

 **Best Practices**

## Tagging entities with marker components

In ECS a "marker component" (aka "tag component") is a component with no fields. It's only job is to mark an entity. You can think of it like a boolean flag on a class: it's either present (true) or absent (false).

Example: you have a bunch of units, and one of them is controlled by a player. You can mark that unit with an empty **Player** component. This lets you filter for player units and also lets you tell while iterating over all units whether you're dealing with a regular unit or the player-controlled one.

---

## Minimize places where a component is changed

The fewer places change a component the better. This follows the very useful **Don't Repeat Yourself (DRY)** principle. It has many benefits.

1. You understand data changes more clearly across the project, which makes debugging easier when something goes wrong.
2. When you update the logic for changing data you touch less code - ideally only one place.
3. There's a lower chance of data bugs appearing out of nowhere.

Example: instead of modifying `Health` component in every system that deals damage, create a single `DamageSystem` whose job is to apply damage to entities with `HealthComponent`. Damage itself also can be an entity with a `DealDamageRequest` component on it.

---

## Use the “Component” suffix only when it helps

The “Component” suffix is useful for beginners because it reminds you “this is data only.” Over time, you don't need that reminder, but your code stays noisy with “Component” everywhere. My advice: you can safely drop the suffix - it adds little value, aside from maybe making autocomplete/IntelliSense searches a bit easier.

For example, `CurrentHealthComponent` becomes just `CurrentHealth` and the code reads a bit more fluently: `entity.hasCurrentHealth`.

**However** sometimes you want to store a reference to a `MonoBehaviour` inside an ECS component, and you'll likely want to name that component the same as the Unity-side type. To avoid name conflicts in that case, you should add the word “Component” to the ECS component's name, like this:

```
1 [Game] public class EnemyViewComponent : IComponent { public EnemyView  
Value; }
```

---

## Storing a reference to another entity inside a component

You might be wondering: “How do we link entities in ECS? Do we tag them with components and then search in a loop?” Of course not. It's much simpler: we store a reference - just in a different form.

We create a separate **ID** component:

```
1 [Game] public class Id : IComponent { [PrimaryEntityIndex] public int  
Value; }
```

---

Attach this component to every new entity:

```
1 CreateEntity.Empty()  
2 .AddId(_idService.Next())  
3 // add other components...  
4 ;
```

Now every entity in the game has a unique ID.

When we want to point to another entity, we don't store a reference to its component or to `GameEntity` itself. We store its **id**. This keeps links simple, serializable and easy to look up.

---

## Deferred reactivity and one-frame components

As noted in “Beginner mistakes” reactive code in ECS can cause problems. But in game dev you often need to react to events. The answer is **deferred reactivity**.

**Deferred reactivity** means: instead of calling logic right away when the event happens, you write data that says the event happened. Any system that cares will react to that data at the right time for it. Think of it like an OOP “dirty” flag: anyone can set `SetDirty(true)`, but the logic reads and handles it later.

Let’s say `Health` component changes, the entity gets a `HealthChanged` **marker component** added. Such markers live for **one frame**: they notify systems this frame, then get removed so the logic doesn’t repeat next frame. The component can be removed by the system that adds it, or by a separate cleanup system that clears all `X` markers where you need it.

**Example:** we have a `DealDamageByRequestSystem`. To tell it how much damage to apply, we add a `DealDamageRequest` component with the damage amount and the ID of the target entity.

`DealDamageByRequestSystem` iterates entities with `Health` and `DealDamageRequest`, finds the target by ID, applies damage, removes `DealDamageRequest`, and adds a `WasDamaged` marker to the target. That marker lets later systems know the entity took damage. At the end of the frame, a cleanup system removes `WasDamaged` so it won’t be processed again next frame.

```
1 public class DealDamageByRequestSystem : IExecuteSystem
2 {
3     private readonly GameContext _game;
4     private readonly IGroup<GameEntity> _requests;
5     private readonly IGroup<GameEntity> _targets;
6
7     public DealDamageByRequestSystem(GameContext game)
8     {
9         _game = game;
10
11         _requests = game.GetGroup(GameMatcher
12             .AllOf(
13                 GameMatcher.DealDamageRequest));
14
15         _targets = game.GetGroup(GameMatcher
16             .AllOf(
17                 GameMatcher.Id,
18                 GameMatcher.CurrentHP,
19                 GameMatcher.Alive)
20             .NoneOf(
21                 GameMatcher.Dead));
22     }
23
24     public void Execute()
25     {
26         foreach (GameEntity request in _requests)
27         {
28             GameEntity target =
29                 _game.GetEntityWithId(request.TargetId);
30
31             if (_targets.ContainsEntity(target))
32             {
33                 float newCurrentHp = target.CurrentHP -
34                     request.DamageAmount;
35                 newCurrentHp = Mathf.Clamp(newCurrentHp, 0,
36                     target.HasMaxHP ? target.MaxHP : newCurrentHp);
37                 target.WasDamaged = true;
38             }
39
40             request.isDestroyed = true;
41         }
42     }
43 }
```

```
38     }
39     }
40 }
```

---

## Requests/Events as a systems API

Building on one-frame components, we can use them to express a simple API for systems: **Request** components to ask a system to do work, and **Event** components to tell everyone what happened. The system can control both lifecycles: delete **Requests** right after handling them, and clear **Events** before firing new ones. This way your features can “speak” to each other and still be completely decoupled.

**Example:** take the `DealDamageByRequestSystem` from the previous section. We send a damage request with a `DealDamageRequest` component. Same way we can notify others with a `DamagedEntityEvent` component on a new event entity. The system’s flow is:

1. Clear all `DamagedEntityEvent` from the last frame (so the event makes a full loop through systems).
2. For each entity with a request, apply damage.
3. Remove the request
4. Create new entity and add `DamagedEntityEvent` on it.

---

## Extracting repeated logic into DI services/extensions

Over time you’ll notice you repeat the same logic in different systems. Usually, that’s a sign it’s time to make a new system :D

But sometimes the repeated logic is helper code tied to one or two specific components/entities and its result is used for different purposes, for example a special way to read data from a component. Some developers put such helper logic right inside the component (like getters), but we are cool guys and we want to keep ECS clean so I suggest another route: **DI services** or **extension methods** that systems call.

Example: you have a `Team` component that stores the team’s color. You might need to check in multiple systems whether two entities are on the same team. Create a static `TeamExtensions` class with a method `IsInSameTeam(GameEntity, GameEntity)` and put the shared comparison logic there. This keeps components as data only and avoids duplication.

Same applies for the DI services. Let’s say you have a complex logic of checking whether an enemy is owning a specific type of a weapon. This logic may require you to access other DI services even, so in this case it’s better to be moved into a `EnemyUtilsService` DI service. This way you can inject any other dependency for this logic.

---

## Grouping systems by when they run

As you know, system order matters in ECS, so it’s useful to group systems at the top level by their place in the frame.

For example, first you can run all input-related systems (**InputFeature**) to collect user input and convert it to ECS data. Next comes the gameplay group (**GameplayCoreFeature**), which interprets the input and updates the ECS world. Finally, you run a cleanup group (**GameCleanupFeature**) that should execute after all gameplay logic.

---

## Should you split components/systems into small pieces?

This is a tricky question.

Simple answer is: **Yes - split**. In 99% of cases you won't regret about it. As said earlier, this is the "atomic" style. The extreme form is: each component has only one field.

This gives you maximum composition in the project, reduces refactors, and removes the "how do I combine entities with property X?" problem.

#### Downsides:

- The number of components grows fast, which can cause confusion on large projects if you don't organize things well.
- It's harder to "see" what an entity **is** from many small components (you can fix this with a clear marker/tag component).

#### When you can skip splitting: events and requests.

In the `DealDamageOnRequestSystem` example, the `DealDamageRequest` component has several fields, because the handling system usually needs all of them at once. I wouldn't call this a strict rule, though. You might later wish you had split a request into several components, but for requests/events it's often fine to keep them together.

#### If you don't split components, you'll hit these problems:

- **More time to reach an ECS-friendly shape.** If a designer adds a new entity with similar data and you need to split later, the refactor cost grows with the amount of code already tied to that data.
- **Less freedom when building entities.** With many small components, you assemble entities like LEGO from small behavior pieces. With fat components, you lose that flexibility.

---

#### How to assemble an Entity?

It's very convenient to build entities using the **Factory** pattern. Especially if you follow the atomic component style. This way you can always check what components a given entity has. If the entity needs new components you don't have to hunt through many creation sites. You just open one Factory class and adjust what you need.

```
1 public class HeroFactory : IHeroFactory
2 {
3     private readonly IIdentifierService _identifiers;
4     private readonly IConfigsService _configs;
5
6     public HeroFactory(IIdentifierService identifiers, IConfigsService
7         configs)
8     {
9         _identifiers = identifiers;
10        _configs = configs;
11    }
12
13    public GameEntity CreateHero()
14    {
15        HeroConfig config = _configs.GetHeroConfig();
16
17        return CreateEntity.Empty()
18            .AddId(_identifiers.Next())
19            .With(x => x.isHero = true)
20
21            .AddLevel(1)
22            .AddExperience(0)
23
24            .AddMovementSpeed(config.MovementSpeed)
25            .With(x => x.isTransformMovement = true)
```

```

25
26         .AddCurrentHealth(config.Health)
27         .AddMaxHealth(config.Health);
28     }
29 }

```

## How to store components?

As said before it's better to split your logic in features. Then components related to this feature can be put in a single file inside this feature folder.

For example we have **InventoryFeature**. Folder structure would look like this:

```

1 Inventory
2 - Systems
3 - Services
4 - InventoryFeature.cs
5 - InventoryComponents.cs

```

So all the inventory components would go in **InventoryComponents.cs** :

```

1 using System.Collections.Generic;
2 using Entitas;
3
4 namespace Code.Gameplay.Features.Loot.ItemDrop.Inventory
5 {
6     [Game, Watched] public class InventoryComponent : IComponent {
7         public Dictionary<ItemId, int> Value; }
8
9     [Game]
10    public class LoseItemInInventoryRequest : IComponent
11    {
12        public int TargetId;
13        public ItemId TypeId;
14        public int Amount;
15    }
16
17    [Game]
18    public class AddItemInInventoryRequest : IComponent
19    {
20        public int TargetId;
21        public ItemId TypeId;
22        public int Amount;
23    }
24 }

```