ChatGPT

Wildfrost “Mad Family” Tribe Mod – Code Review & Documentation

1. Mod Project Structure (WildfrostMadFamilyTribe/ WildfrostBirthday)

Overview: The “Mad Family” mod is organized as a C# project (likely using the Wildfrost modding API). It defines a new Tribe (the Mad Family) with custom **cards** (leaders, companions, items), **charms**, and **status effects**. Each category is separated into subfolders (`Cards/`, `Charms/`, `Effects/`, `Tribes/`), and each content piece is defined in its own file with a static `Register` method. This structure follows a clear convention and best practices for Wildfrost modding, making it easy to maintain and extend.

Key files and folders:

- **WildfrostBirthday.sln / .csproj:** Solution and project files configuring the build. The project targets the Wildfrost modding framework.
- **WildFamilyMod.cs:** The main mod class (extends `WildfrostMod`) – responsible for initializing and integrating the mod with the base game.
- **Helpers/**: Utility classes (e.g. `ComponentRegistration`, `CardScriptHelpers`, etc.) to reduce repeated code and assist in data registration or copying.
- **Cards/**: Defines new card data for companions, leaders, and items. Each file (e.g. `Card_Alison.cs`, `Item_TackSpray.cs`) contains a static class with a `Register(mod)` method that uses a builder to create the card’s attributes and add it to the mod.
- **Charms/**: New charm data (similar structure to cards). Charms are essentially Card Upgrades (like game charms/crowns) that confer status effects to their bearer.
- **Effects/**: New status effects introduced by the mod. Each file defines a static `Register` that uses a `StatusEffectDataBuilder` to create the effect, often by leveraging an existing base effect class and customizing it.
- **Tribes/**: Contains `Tribe_MadFamily.cs` which sets up the new tribe (class) data, including leader options, starting deck, and reward pools.
- **assets/**: Contains images and a config. For example, `assets/images/` holds custom sprites for cards and charms (e.g. `companions/poppy0.png` and variations) and `tribe_madfamily.png` for the tribe icon. The `config.cfg` might include metadata for the mod loader.

Registration Pattern: All mod components use a reflection-based auto-registration. The `ComponentRegistration.RegisterAllComponents` method scans the assembly for any static class whose name starts with `Card_`, `Item_`, `Charm_`, `StatusEffect_`, or `Tribe_`, and invokes its `Register(WildFamilyMod)` method. This eliminates manual calls for each registration and ensures every defined component is loaded:

```
// WildfrostBirthday.Helpers.ComponentRegistration snippet
foreach (Type type in assembly.GetTypes()) {
    if (!type.IsAbstract || !type.IsSealed) continue;  // only static classes
    MethodInfo? registerMethod = type.GetMethod("Register", ...
typeof(WildFamilyMod) ...);
    if (registerMethod == null) continue;
    string typeName = type.Name;
    string? category = null;
    if (typeName.StartsWith("StatusEffect_")) category = "StatusEffect";
    else if (typeName.StartsWith("Card_")) category = "Card";
    // ... similar for Item_, Charm_, Tribe_ ...
    if (category == null) continue;
    registerMethod.Invoke(null, new object[] { mod });  // call Register method
}
```

The mod's **content classes** are all static with `Register` methods following this pattern. Notably, any class name containing "Template" is skipped in registration (to allow example/template files without activating them).

Integration with Base Game: The mod's design leverages the official mod API (via `WildfrostMod` base class and builder classes). It doesn't patch game code directly (no Harmony patches here), which is a **good practice** for stability. Instead, integration happens by creating new data objects and inserting them into the game's existing structures:

- New **ClassData** (tribe) is added to the game's tribe list.
- New **CardData** (for cards) and **CardUpgradeData** (for charms) are created and automatically included in addressable groups so the game recognizes them.
- Event hooks are used sparingly for specific issues (e.g., image display fix).
- All mod-added content is tracked (stored in `mod.assets`) so it can be unloaded cleanly if needed.

Build Output: After compilation, the mod produces `WildfrostBirthday.dll` (the mod code) and includes resources like images. The `assets/` folder suggests this is also a BepInEx or similar deployment where the DLL and images are packaged for the mod loader.

Suggestions: The structure is well-organized. One suggestion is to ensure consistent naming and GUID usage. The mod's GUID is `"madfamilymod.wildfrost.madhouse"` (as set in `WildFamilyMod.GUID`), and new data should ideally be prefixed or uniquely identified by the mod to avoid conflicts with base game or other mods. The provided builders likely handle prefixing automatically (e.g. via internal calls to `Extensions.PrefixGUID` or similar), but it's good to double-check that custom effects with names identical to base effects (e.g. "When Hit Apply Demonize To Attacker") don't accidentally clash. In practice, the mod uses `TryGet` to fetch base data when needed and typically creates new effects with either unique names or by cloning and renaming base ones via `StatusCopy`/`CardCopy` methods in `WildFamilyMod`. This approach is sound.

Overall, the project structure reflects a clear separation of concerns: each game element (card, effect, etc.) is defined in isolation, and they are all registered in a central way. This modular design is easy to navigate and aligns with recommended practices.

2. Code Logic & Integration with Base Game

WildFamilyMod: Mod Entry Point and Lifecycle

The `WildFamilyMod` class (in **WildFamilyMod.cs**) extends `WildfrostMod`, which is part of the modding API. This class serves as the entry point for the mod. Key aspects of `WildFamilyMod` include:

- **Mod Metadata:** It overrides properties like `GUID`, `Title`, `Description`, and `Depends`. For example, GUID is `"madfamilymod.wildfrost.madhouse"` and Title is `"MadHouse Family Tribe"`. These identify the mod and are used by the loader.

- **Load() Method:** This is called when the mod is loaded. In `Load`, the mod ensures it registers content once and then invokes base game modifications:

- The first time Load runs, it calls `ComponentRegistration.RegisterAllComponents(this)` to register all custom status effects, cards, items, charms, and the tribe. This populates the mod with all its custom data (each builder added to `mod.assets`).
- It then calls `base.Load()` to let the base class handle standard loading behavior (which likely builds the assets and integrates them).
- After that, it hooks an event: `Events.OnEntityCreated += FixImage;` – this attaches a handler (`FixImage`) that will run whenever an in-game entity is created. The purpose is to fix card image visibility issues for custom cards (ensuring the `mainImage` GameObject is active for cards without a "scriptable image"). This is essentially a bugfix for custom cards using static images.

- Finally, it integrates the new tribe into the game: it fetches the `GameModeNormal` object (the standard campaign mode) and appends the new ClassData for "MadFamily" tribe to `gameMode.classes` array. This means the new tribe will appear as an option on the tribe (clan) selection screen when starting a run.

- **Unload() Method:** This is called when the mod is unloaded (or game exits). It removes the mod's tribe from the game mode to clean up:

- It obtains `GameModeNormal` and calls `RemoveNulls` on its classes array, which filters out any null or mod-added entries. This ensures if the player returns to tribe selection without restarting the game, the removed mod tribe won't appear and cause errors.

- It also calls `UnloadFromClasses()`, a helper that iterates through all tribes' reward pools and removes any rewards marked with `ModAdded == this` (i.e., those introduced by this mod). This prevents stray references to mod cards or charms after unloading.

- **Utility Methods:** `WildFamilyMod` defines several useful helpers for content creation:

- `TryGet<T>(string name)`: A safe lookup that tries to find a game data object by name, searching both base game and mod-added content. It handles different types (StatusEffectData, KeywordData, etc.) appropriately. If not found, it throws an exception for debugging.
- `SStack(name, amount)` and `TStack(name, amount)`: Shorthand to create a `CardData.StatusEffectStacks` or `CardData.TraitStacks` with the given effect/trait (by name) and count. This is used when assigning status effects or traits to cards.
- `StatusCopy(oldName, newName)`: Clones an existing StatusEffectData from the base game (identified by `oldName`), and prepares a builder for a new status with `newName`. It copies all fields, resets target constraints, and sets the mod as the owner. This is handy when a new effect is very similar to an existing one – you can copy and tweak it.
- `DataCopy<Y, T>(oldName, newName)`: A generic method that clones any DataFile (card, class, etc.) to a new builder of type T, prefixing the new object's name with the mod GUID. This is the underlying method used by StatusCopy and the similar `TribeCopy` (which clones a ClassData).
- Quick builder methods for CardScripts: `GiveUpgrade("Crown")`, `AddRandomHealth(min,max)`, etc., which create ScriptableObject instances of built-in CardScript classes (like `CardScriptAddRandomDamage`) for use in leaders (explained below).

These utilities make the content definitions more concise and consistent.

- **FixImage(Event) Handler:** The `FixImage(Entity entity)` method is registered to `Events.OnEntityCreated`. In it, the code checks if the created entity's display is a `Card` and if that card does **not** have a "scriptable image" (i.e., it uses a static sprite). If so, it ensures the `card.mainImage` GameObject is active. This addresses an issue where custom card images might not show up because the game expects a ScriptableCardImage (an animated effect) and may disable the static image. By forcing `mainImage` visible, the static sprite (from `.SetSprites`) will display correctly. This is a smart integration fix, and the comment credits "Hopeful" for the method, indicating it's a known workaround in the modding community.

Best Practices Note: `WildFamilyMod` demonstrates good practice by using the provided events and APIs to integrate the mod: - It **does not hard-patch** game code; instead, it adds content through official channels (builders and event hooks). - It keeps track of all added content and removes them on unload, preventing persistence issues. - The use of reflection for registration is a clean automation of what would otherwise be many manual calls – just ensure all Register methods are properly named and static (the mod's structure does this).

Tribe Definition: ClassData for "Mad Family"

The new tribe is defined in **Tribe_MadFamily.cs**. This is one of the most important integration points, since a tribe (or "class") encapsulates leaders, starter deck, and progression for that clan.

In `Tribe_MadFamily.Register(mod)`, the mod creates a new `ClassData` using the builder pattern:

- It uses `ClassDataBuilder(mod).Create("MadFamily")` to start defining a new class with internal name "MadFamily". No base clone is used here; it's a fresh ClassData.
- `.WithFlag("Images/tribe_madfamily.png")` sets the banner icon for the tribe selection (pointing to the custom image in assets).

- `.WithSelectSfxEvent(<path>)` assigns a sound effect to play when this tribe is selected; in this case they use an existing SFX (“draw_multi” card draw sound).

After building the basic fields, they use `SubscribeToAfterAllBuildEvent` on the builder to finalize complex data once the ClassData object is created. Inside this subscriber lambda, they populate the fields:

- **ID and Character Prefab:** They set `data.id = "MadFamily"` explicitly (this might match the internal name, ensuring consistency). Then they create a character prefab for the tribe’s leader: they clone the base game’s “Basic” tribe character prefab as a starting point. This is done by `mod.TryGet<ClassData>("Basic").characterPrefab.gameObject.InstantiateKeepName()` – it duplicates the Basic class’s character GameObject (likely a template for leader units). They mark it `DontDestroyOnLoad` (so it persists across scene loads) and rename it to "Player (MadFamily)". Finally, they assign `data.characterPrefab` to the **Character** component on that cloned object. This character prefab is what the game uses to represent the leader on the map and in battles (animations, model, etc.). By cloning an existing one, they avoid building one from scratch, which is a clever shortcut (the tutorial suggests this method).

- **Leaders:** The tribe defines multiple possible leader cards. They set `data.leaders` to an array of `CardData` references for each leader. They use `mod.TryGet<CardData>("leader-alison")`, and similarly for Tony, Cassie, Caleb, Kaylee – these are the internal IDs of the leader cards defined in the mod (we will discuss these cards in the next section). By the time this runs, those leaders are already built and registered (the static Card_* classes will have run, adding their builders to mod assets). `TryGet` retrieves the built `CardData` by name. This leaders array makes those characters available as starting leader choices for the Mad Family tribe.

- **Starting Inventory (Starting Deck):** They create a new `Inventory` object for the tribe’s start. This includes the starting deck of cards, any reserve cards, and initial charms:

- `inventory.deck` is set to a `CardDataList` (a list-like collection of cards) containing specific cards fetched via `mod.TryGet<CardData>(...)`. For Mad Family, the starting deck includes base game cards: **SnowGlobe**, **Sword** (two copies), **Gearhammer** (two copies), **Sunlight Drum**, **Junkhead**, and two Mad Family companions: **Lulu** and **Poppy** (internal IDs “companion-lulu” and “companion-poppy”). This mix suggests the tribe starts with some standard items (SnowGlobe freezes, Sword and Gearhammer for damage), a Sunlight Drum (apply Spice), a Junkhead (summonable clunker), and their two unique companions. Including the mod’s companions here is crucial so the player starts with them.
- `inventory.reserve` is an empty CardDataList (no reserve cards at start, which is common).
- `inventory.upgrades` is a list of `CardUpgradeData` (charms) the tribe starts with. In this mod, they give one charm: `mod.TryGet<CardUpgradeData>("charm-bookcharm")`. “Book Charm” is one of the custom charms in this mod (we see it added to the charms pool later). Starting with a charm is unusual in base game (except the Frost Guardian tribe starts with the Frozen Heart Charm), so this is a special perk for Mad Family: presumably Book Charm provides some bonus from the outset.

They set `inventory.goldOwed = 0` (no debt). Finally, they assign this `inventory` to `data.startingInventory`. This ensures when a run begins with Mad Family, the player's initial deck and charm are exactly as defined.

- **Reward Pools:** Each tribe can have customized reward pools which determine what cards and charms appear as rewards (in chests, shops, etc.) during a run with that tribe. The mod defines several pools and assigns them to `data.rewardPools`. In `CreateRewardPools(mod)` helper, they construct:
- **Items Pool:** A `RewardPool` of type `"Items"` – containing item cards that can drop as rewards. They set `copies = 1` (each reward is one copy) and `isGeneralPool = true` (meaning it's used generally in runs, not a special event). They include a mix of base-game items and **all the new mod item cards**. For example: base items like Mimik, Blingo, BerryBlade, etc., and then `item-snowpillow`, `item-refreshingwater`, `item-wispmask`, and so on through every `item-...` defined by the mod. This integration is excellent – it means as you play the Mad Family tribe, you can find the new items in treasure chests or buy them in shops, alongside regular items.
- **Units Pool:** A pool of type `"Units"` for companion cards that can appear (usually in companion selection events or charmsmith, etc.). In the code they included many base-game companions (Foxee, Blunky, Dimona, etc.) as options. Notably, **they did not add the mod's own companions (Poppy, Lulu, etc.) to this units pool**. That means those companions might only be available at start and not show up as later recruits. This could be intentional (to make them exclusive to starting deck) or an oversight. If the design is to allow recruiting additional Mad Family companions, they would need to be in this pool. As it stands, playing Mad Family likely lets you recruit standard game companions as you progress. (From a design perspective, it might make sense – the Mad Family's unique characters are already with you).
- **Charms Pool:** A `"Charms"` pool listing charms that appear in charm chests. They start by adding essentially *every base-game charm* by its CardUpgradeData name (CardUpgradeBalanced, Barrage, Bling, etc.), and then they append all **custom charms** from the mod: charm-pugcharm, charm-goldenvialcharm, frostmooncharm, soda, pizza, plant, book, duck charms. Setting `isGeneralPool = true` means these charms are in the general charm pool for this tribe. So, when you open a charm chest as Mad Family, you could get any normal charm or one of the fun custom ones.
- **Snow Pools:** Interestingly, they define a couple of extra pools: "Snow items" and "Snow units" pools, each of type Items/Units with a couple of very specific entries (Snowcake & SnowGlobe, and Snobble & Snoffel). These might be intended for a special event or synergy (perhaps if the tribe has a thematic tie to snow, or maybe the Book Charm or other content triggers these pools). They set them as general pools too, which might make these appear in addition to normal pools – or possibly they intended to assign these to a specific event. It's a minor detail in the code that doesn't hurt; at worst it slightly increases the chance of snow items/units appearing for Mad Family.

Finally, the `CreateRewardPools` returns an array of all these pools, which the subscriber assigns to `data.rewardPools`.

The **integration point** is that once this ClassData is built and added to `GameModeNormal.classes` (in WildFamilyMod.Load), the game fully recognizes the new tribe: it appears in the clan selection UI (with the given icon and sound), shows the listed leaders and their preview decks (which the base game will compile from startingInventory), and uses the specified reward pools for run generation. All the heavy lifting of connecting to the game's progression is handled by providing a well-formed ClassData.

Best Practices & Improvements: The tribe setup is thorough. Cloning the character prefab is essential; without it, the new leaders might not have a proper in-game avatar or could cause errors. The mod does this correctly. They also were careful to include base game content in pools to keep the tribe from feeling isolated (players still find familiar cards, just with some new ones added).

One improvement might be to include the mod's unique companions in the Units reward pool if the design is to allow multiple of them or finding them later. If not (perhaps for balance), it's fine as-is. Also, the Snow-specific pools might be a relic of experimentation – if not used, they could be omitted; but since `isGeneralPool` is true, they'll just function as additional pools (in Wildfrost, multiple pools of the same type might all be considered when generating rewards).

Card Definitions: Leaders, Companions, and Items

The mod adds many new **CardData** entries. These are defined in the `Cards/` folder. The approach uses the provided `CardDataBuilder` to fluently configure each card, which is very readable. We'll break down examples for a **companion**, a **leader**, and an **item** card to illustrate the logic and integration.

Companions Example – *Poppy* (Companion)

File: **Card_Poppy.cs** – defines the companion *Poppy*. Inside `Card_Poppy.Register(mod)`:

```
string cardId = "companion-poppy";
var companionBuilder = new CardDataBuilder(mod)
    .CreateUnit(cardId, "Poppy")
    .SetSprites("companions/poppy0.png", "bg.png")
    .SetStats(11, 2, 4)  // HP, ATK, Counter
    .WithFlavour("Ferocious little guardian who fights back hard.")
    .WithCardType("Friendly")
    .WithValue(50)
    .SubscribeToAfterAllBuildEvent(data => {
        // Start with status effects
        data.startWithEffects = new[] {
            mod.SStack("When Hit Apply Demonize To Attacker", 2)
        };
        // Traits
        data.traits = new List<CardData.TraitStacks> {
            new CardData.TraitStacks(mod.TryGet<TraitData>("Smackback"), 1)
        };
        // Attach dynamic sprite-change script
        var script =
ScriptableObject.CreateInstance<CardScriptChangeMainOnCounter>();
        script.baseImagePath = "companions/poppy";
        // Add script to createScripts list
        if (data.createScripts != null) {
            var scripts = new List<CardScript>(data.createScripts) { script };
            data.createScripts = scripts.ToArray();
        } else {
            data.createScripts = new CardScript[] { script };
        }
    });
mod.assets.Add(companionBuilder);
```

Key points from this definition: - **CreateUnit(id, name):** This sets up a new **unit** card (as opposed to an item or enemy) with the given internal ID and display name. The internal ID “companion-poppy” is used to reference this card in code and reward pools, while the display name "Poppy" is what players see on the card. - **SetSprites(mainSprite, backgroundSprite):** It assigns the card’s images. Here `"companions/poppy0.png"` is the main sprite (the character image) and `"bg.png"` is a generic background. They plan to change the main sprite depending on counter (see CardScript below), so they provided multiple sprite files (poppy0, poppy1, poppy2, etc.) and here initially use poppy0. The background is presumably a default provided in assets (common to all their cards). This is integrated such that when the card is rendered in-game, it will use these sprites unless overridden. - **SetStats(HP, Attack, Counter):** Poppy is given 11 health, 2 attack, and a counter of 4. “Counter 4” means after acting it takes 4 turns to act again (in Wildfrost terms). Setting `hasAttack` and `hasHealth` is done implicitly by providing those values (the builder likely handles toggling those bools). These stats will reflect in battle. - **WithFlavour(text):** Sets the flavor text (italic text on card). For Poppy: *"Ferocious little guardian who fights back hard."* – likely hinting at her Smackback trait. - **WithCardType("Friendly"):** This categorizes the card. “Friendly” likely corresponds to the allies category (as opposed to “Leader” or “Enemy” or “Item”). Friendly means it’s a normal companion that can appear in the player’s team. This is important for game logic (e.g. it can be put in the companion limit, etc.). - **WithValue(50):** Likely sets the reward value or gold cost of the card. In Wildfrost, each card has a value (used when scrapping for gold or balancing rewards). 50 is a mid-range value for a companion, indicating moderate strength/rarity. - **AfterAllBuildEvent subscriber:** This is where they customize fields not directly covered by the fluent methods: - `startWithEffects`: Poppy starts with a status effect *“When Hit Apply Demonize To Attacker”* at 2 stacks. This means Poppy has a reactive effect: when Poppy is hit, the attacker gains Demonize(2). Demonize is a negative status (the attacker will take double damage). The mod created this status effect themselves for use here (we’ll see it in the Effects section). In base game, a similar effect exists for a enemy Gok (Demonize the attacker on being hit), so the mod likely mirrored that. By giving 2 stacks, Poppy applies Demonize(2), which is quite powerful (double damage taken, twice). This makes her a dangerous target to hit – fitting the “guardian fights back” flavor. Implementation-wise, this array of `StatusEffectStacks` gets assigned to the CardData’s `startWithEffects`, meaning whenever Poppy is spawned, the game will attach that StatusEffectData to her with count 2. - `traits`: They give Poppy the **Smackback** trait (1 stack). Smackback is a built-in trait that allows a unit to counterattack when hit. This synergizes with her Demonize effect – she’ll smack back and likely do double damage if the attacker is demonized. In code they do `mod.TryGet<TraitData>("Smackback")`, retrieving the base game trait, and wrap it in a `TraitStacks`. Traits in Wildfrost are like keywords (Smackback, Haste, etc.); adding it here means Poppy’s card will display the Smackback icon and have that behavior. - **Custom CardScript:** They create an instance of `CardScriptChangeMainOnCounter` (a custom script class) and set its `baseImagePath = "companions/poppy"`. This script (defined in Helpers) will, at runtime, adjust Poppy’s sprite as her counter changes. Specifically, `CardScriptChangeMainOnCounter.Run(card)` checks the card’s `counter` value and switches the main image to `poppy0.png`, `poppy1.png`, `poppy2.png`, or `poppy3.png` depending on the counter. This means as Poppy’s turn counter ticks down, the image could change (perhaps to show her getting more excited or angry). They also store `lastCounter` to avoid redundant changes. This is a clever cosmetic touch – making the card visually reflect its state. To attach this script to the card, they must add it to `data.createScripts`. The code handles whether `createScripts` already has entries or not, and then includes this new script.

Finally, they add the builder to `mod.assets`. The builder pattern means that the actual `CardData` object isn't created immediately; it's created when the mod system finalizes assets (likely right after all Register calls, in base.Load or similar). At that time, each builder builds the `CardData` ScriptableObject, and because `WildFamilyMod.AddAssets` returns them, the game incorporates them.

Integration: Once built, "Poppy" is a fully functional companion in the game. It will show up in the Mad Family starting deck (they added `companion-poppy` to starting inventory) and potentially could be summoned or referenced elsewhere. All the status effects and traits reference base game or mod-defined data, which we ensure exist: - "When Hit Apply Demonize To Attacker" is defined in mod Effects. - Smackback trait is base game. - The CardScriptChangeMainOnCounter is mod-defined and will run on card creation (when you start a battle with Poppy, or if Poppy is summoned – though companions aren't summoned usually, but the script will run at battle start to set the initial image, and anytime the card's counter resets since they call it on certain events presumably).

This definition shows good practice: using **mod helpers** (SStack, TryGet) to wire in effects/traits, and adding custom behavior without altering core code (just by attaching a script in the allowed way).

Leader Example – *Alison* (Leader & Companion Duality)

The mod's approach for leaders is to often have a leader version and possibly a companion version of a character. In **Card_Alison.cs**, they actually register both a companion and a leader variant of Alison in one go. For brevity, let's focus on the **Leader** part to see what's unique:

```
var leaderBuilder = new CardDataBuilder(mod)
    .CreateUnit("leader-alison", "Alison")
    .SetSprites(getSpritePath(spritePath, cardId), "bg.png")
    .SetStats(9, 3, 3)
    .WithFlavour("Restore 2 HP on kill")
    .WithCardType("Leader")
    .WithValue(50)
    .SubscribeToAfterAllBuildEvent(data => {
        // Attack effects
        data.attackEffects = new[] {
            mod.SStack("On Kill Heal To Self", 2)
        };
        // Leader-specific create scripts
        data.createScripts = new CardScript[] {
            CardScriptHelpers.GetGiveUpgradeScript(mod),
            CardScriptHelpers.GetRandomHealthScript(-1, 3),
            CardScriptHelpers.GetRandomDamageScript(0, 2),
            CardScriptHelpers.GetRandomCounterScript(-1, 1)
        };
    });
mod.assets.Add(leaderBuilder);
```

Differences for a Leader: - **Internal ID and CardType:** “leader-alison” and `WithCardType("Leader")`. This marks the card as a leader unit. The game likely treats leaders slightly differently (e.g. exactly one can be chosen to start a run). The mod ensures to categorize it properly. - **Sprites:** For Alison’s leader, they use a `getSpritePath` helper that picks a random variant: `alison0.png`, `alison1.png`, or `alison2.png`. This means each run, Alison’s leader card might have one of a few appearances (maybe different outfit or pose). It’s a neat touch to give variety. All variants are in the `leaders/` images folder. This randomness is seeded in the Register method (they call `Dead.Random.Range(0,3)` – presumably a deterministic random for mod content). It’s fine because the sprite is chosen once when building the card. - **Stats:** 9 HP, 3 ATK, 3 Counter – a bit lower health than Poppy, and a faster counter. Leaders tend to be a bit weaker than average companions because you always have them. - **Flavour:** “Restore 2 HP on kill” – hinting at her effect. - **Attack Effects:** Instead of a start effect, Alison leader has an **attack effect**: On Kill Heal To Self (2). This means whenever Alison kills an enemy, she will heal herself for 2 HP. The mod created the “On Kill Heal To Self” status effect and gives it 2 stacks (for 2 HP heal). It’s placed in `attackEffects` because it triggers on performing an attack that kills an enemy. This directly ties to her flavor text. Integration-wise, when Alison deals a killing blow, the StatusEffectApplyXOnKill effect will fire and apply a Heal effect to Alison (we’ll see in Effects). - **Leader Create Scripts:** They override `data.createScripts` with a set of four scripts using the `CardScriptHelpers`: - `GiveUpgradeScript(mod)` which likely gives the leader a **Crown** upgrade at the start. Indeed, `GetGiveUpgradeScript(mod)` creates a `CardScriptGiveUpgrade` for a “Crown” upgrade. In Wildfrost, a Crown makes a card start on the board for free each battle. By crowning the leader, they ensure the leader is always on the field at battle start (which is normal in game – your leader is always deployed). Perhaps this script is redundant if the game auto-deploys leaders, but it might be required in the mod environment. It doesn’t hurt either way. - `GetRandomHealthScript(-1, 3)`: This will add a random amount between -1 and +3 to Alison’s health when the run starts. That means her 9 HP could become anywhere from 8 to 12 HP. It introduces a slight roguelike variance. - `GetRandomDamageScript(0, 2)`: Adds 0 to 2 attack randomly. So her 3 ATK could be 3, 4, or 5. - `GetRandomCounterScript(-1, 1)`: Adjusts her counter by -1 to +1. So her counter 3 could become as low as 2 or as high as 4.

These random stat adjustments mean each time you start a run with Alison, her exact stats might differ a bit. This is a design decision by the modder – perhaps to simulate the random charm you normally get for a leader, or just to make it interesting. It’s implemented via CardScripts that run on card creation (these scripts use Unity’s `Random.Range` as well). The use of these `CardScriptAddRandomX` classes is directly supported by the base game – they are ScriptableObjects that alter the card’s stats in `CardData.RunCreateScripts()` step. The mod attaches them so they execute once when the leader card is spawned for the run.

With these scripts, Alison leader might start one run with 10 HP 4 ATK, another with 8 HP 5 ATK, etc. The mod uses *existing* CardScript classes via helpers (no need to reinvent random stat logic, they leverage what’s available).

Leader Integration: The leader cards are added to the tribe’s ClassData as described. They use `"Leader"` type, which likely informs the UI to place them as selectable leaders (and not show up in normal card pools). The crown script ensures the leader card is auto-deployed (which is standard). The random stat scripts illustrate how to customize a leader's start without needing external configuration – all within the card's createScripts.

One thing to note: They did not give the leader Alison the Smackback trait like the companion version. Leaders are balanced separately; perhaps they chose not to make her too strong initially. The leader has the heal-on-kill effect for sustain instead.

Item Example – *Tack Spray* (Item Card)

File: **Item_TackSpray.cs** shows a new item card. Items in Wildfrost are single-use cards the player plays from hand (with a target or area effect). The mod defines several creative items. Tack Spray is described as "A spray that hits all enemies."

From `Item_TackSpray.Register(mod)`:

```
var builder = new CardDataBuilder(mod)
    .CreateItem("item-tackspray", "Tack Spray")
    .SetSprites("items/tackspray.png", "bg.png")
    .WithFlavour("A spray that hits all enemies.")
    .WithCardType("Item")
    .WithValue(45)
    .SetDamage(1)
    .SubscribeToAfterAllBuildEvent(data => {
        data.startWithEffects = new[] {
            mod.SStack("Hit All Enemies", 1)
        };
        data.canPlayOnHand = false;
        data.canPlayOnEnemy = false;
        data.playOnSlot = true;
    });
mod.assets.Add(builder);
```

Important details: - **CreateItem(id, name):** Denotes this as an item card. Item cards typically have `CardType = "Item"` and `PlayType = Play` in base game. The builder likely handles setting the appropriate flags for an item (e.g., items are one-time use by default). - **Stats:** They call `.SetDamage(1)` – giving the Tack Spray an attack value of 1. Items in Wildfrost often have a damage stat if they deal direct damage. This suggests Tack Spray does 1 damage per hit. - **Flavour:** Provided, matches description. - **Value:** 45, slightly less than a companion's 50, presumably balancing reward. - **After build:** - `startWithEffects`: They add **"Hit All Enemies" (1)**. This is presumably a status effect that modifies the card's behavior such that it targets or hits all enemies. In base game, there is a concept of *Bombard* or *Barrage*. "Hit All Enemies" might correspond to the *Bombard* effect (the sheet shows Bombard might have been implemented as status effects). It might also be a custom effect created by the mod (though the name suggests something straightforward). Actually, "Hit All Enemies" likely corresponds to the *Barrage* keyword (which hits all targets in a row) or *Bombard* (all enemies anywhere). However, they listed it under start effects instead of traits, which is interesting. Possibly "Hit All Enemies" is implemented as a status effect that, when on the card, causes its attack to hit everyone. If this effect is based on `StatusEffectApplyXPostAttack` or similar, it could loop through enemies. In any case, by giving Tack Spray this effect at start, whenever you play it, it will apply damage to all enemies rather than one. - They then adjust targeting flags: `data.canPlayOnHand = false` and `data.canPlayOnEnemy = false`, and `data.playOnSlot = true`. This means the card **cannot** be played on a card in hand or directly on an enemy card, but **can** be played on an empty slot on the board. In other words, Tack Spray is an item you place onto the battlefield (like a trap or area effect) rather than targeting a specific enemy. Likely, by playing it on a slot, it then executes its effect hitting all enemies. This is a bit unusual as most items target an enemy or ally or the board generally – but Wildfrost does allow items that you drop onto the ground (e.g. storms or traps). Setting these flags ensures the UI will only allow dropping it on a tile. The combination effectively says: you **must** choose a tile (slot) to play this, not a character or nothing. Once played, the "Hit All Enemies" effect triggers 1 damage to all enemies.

This item demonstrates how to use status effects to change an item's effect pattern. The mod likely defined "Hit All Enemies" in its Effects (or it exists as base "Bombard"). Checking the reference: the base "Bombard" might be tied to hitting random targets including allies, but perhaps "Hit All Enemies" is simpler (maybe implemented via `StatusEffectApplyXPostAttack` with flags for all enemies).

Integration: Tack Spray was added to the Mad Family items reward pool, so players can find it. When used in battle, the game will process its effects: - It sees `canPlayOnSlot=true`, so on play, it treats it like an "area" item. - The `attackEffects` of the card include the "Hit All Enemies" status effect. Likely, when the card "attacks" (items typically do a one-time action), that effect triggers and deals damage. If implemented as a status effect, the effect might subscribe to a PreAttack or OnPlay event. There is possibly a base `StatusEffectApplyXOnTrigger` they used. - `damage = 1` ensures each enemy gets 1 damage.

It's a straightforward but effective design: at play, do 1 damage to each enemy. This is akin to a weaker **Shroom Dust** (which hits all enemies with poison in base game). Value 45 suggests it's a decent item.

Other Cards: The mod defines many cards similarly: - **Companions**: Alison (companion version likely similar to leader minus scripts), Tony, Cassie, Caleb, Kaylee, Lulu, Wisp, Soulrose – each with unique stats and effects. For example, Lulu and Wisp appear in starting deck. Soulrose might be a summonable unit (there are effects summoning Soulrose). - **Items**: A range of creative items like Azul Torch, Berry Cake, Detonation Strike, Foam Rocket, etc. Each likely uses status effects: e.g., Foam Rocket might deal damage and apply **Overload**, Detonation Strike might damage and consume, etc. The consistent pattern: use `CreateItem`, set stats, then give either an `attackEffects` or `startWithEffects` that define what the item does on play, possibly adjusting targeting flags. - **Charms**: We cover in the next subsection, but charm cards are defined similarly in the Charms folder using `CardUpgradeDataBuilder`.

The **builder API** usage in all these cards is a strong point: it ensures that the new cards integrate seamlessly. When the mod's content is built: - Each card gets a unique `CardData` name (with mod GUID likely prefixed internally). - They are added to relevant pools so the game knows how to present them (leaders via ClassData, others via reward pools). - The stats and effects tie into existing game mechanics (through status effects and traits).

One best-practice suggestion: ensure that the textual descriptions (`WithText` in builder, or flavour) match the effects. The mod does a pretty good job (Poppy's flavour hints at Smackback, Alison's at healing on kill, Tack Spray's at hitting all enemies). For custom status effects, they use `WithText` to describe them. E.g., the Pug Charm had `.WithText("When an ally is hit, apply 1 frost to them")` – slight grammar aside, it explains the effect. Keeping these synchronized with the actual coded effect is important for player clarity.

Status Effects: Creation and Usage in Mod and Base

Status effects are central to Wildfrost's mechanics. In the base game, each named effect or keyword (e.g. **Snow**, **Demonize**, **Spice**, or triggered effects like **When Hit Apply Frost**) is defined as a `StatusEffectData` (often with specialized subclasses for triggered behavior). The mod **creates new status effects** primarily by using existing base classes that encapsulate certain trigger logic, then customizing them via builder.

Base Game StatusEffect System (Brief Overview)

In Wildfrost's code, `StatusEffectData` (in **StatusEffectData.cs**) is the abstract base for all status effects. Notable properties in base `StatusEffectData` include: - Flags like `isStatus` (typical status icon), `isReaction` (triggers on events), `isKeyword` (for keyword-only entries), as well as `offensive`, `makesOffensive`, `doesDamage`, `stackable`, `canBeBoosted`, etc., which define how the effect behaves (shown on card, affected by Spice, etc.). - `textKey` and `desc` for localization and description text, and possibly `applyFormat` for dynamic text like "<+{0}>" replacements. - Many **event hooks**: the class defines delegates for events like `OnTurnStart`, `OnAttack`, `OnHit`, `OnKill`, etc., and corresponding virtual methods `RunTurnStartEvent`, `RunHitEvent`, etc. By default, these do nothing (return true), but specific subclasses override them or subscribe to the events to execute logic.

How base game uses subclasses: The base game has *dozens* of concrete `StatusEffect...` classes (as seen in the Reference/Assembly-CSharp listing). For example, `StatusEffectApplyXOnKill` handles "when this unit kills something, apply X to Y target" logic, `StatusEffectSummon` handles summoning a unit on an event, `StatusEffectBlock` provides block stacks that negate hits, etc. Each such class typically overrides `Init()` to subscribe to the relevant event and sometimes override a `Run...Event` to put a condition on execution.

For instance, `StatusEffectApplyXOnKill.Init()` subscribes its internal event so that when any entity is destroyed, it calls its effect if the killer was the target entity. It overrides `RunEntityDestroyedEvent` to check that the `entity.lastHit.attacker == target` (ensuring the target of this status was the one who got the kill). If true, it triggers `Run(...)` which actually applies the effect (the logic for applying is in the base `StatusEffectApplyX` class's `Run` method). The base class `StatusEffectApplyX` provides a generalized implementation for applying an effect or damage to various targets when triggered, based on flags like `applyToFlags`, `scriptableAmount`, etc..

The takeaway: **To create a new triggered effect, modders usually don't subclass these directly at runtime (since that would require extending a Unity ScriptableObject class in the mod, which is possible but not necessary). Instead, the mod can instantiate one of these base classes and configure its fields.** This is exactly what the builder API allows: you pick a class like `StatusEffectApplyXOnKill`, create an instance via `.Create<StatusEffectApplyXOnKill>(name)`, and then set its properties (text, target, etc.).

Mod-Created Status Effects

The mod's **Effects/** folder contains static classes for each custom effect name. The pattern in these is:

```
var builder = new StatusEffectDataBuilder(mod)
    .Create<SomeBaseStatusEffectClass>("Effect Name")
    .WithText("Description with {0} etc", SystemLanguage.English)
    .WithTextInsert("<{a}><keyword=...>")
    .WithStackable(true/false)
    .WithCanBeBoosted(true/false)
    .WithOffensive(true/false)
    ... (other fluent config like MakesOffensive, DoesDamage)
    .SubscribeToAfterAllBuildEvent<SomeBaseStatusEffectClass>(data => {
         // set specific fields on data
         data.someField = X;
         data.anotherField = Y;
    });
mod.assets.Add(builder);
```

This yields a new StatusEffectData asset. Let's examine a few of them:

- **"On Kill Heal To Self"** – uses `StatusEffectApplyXOnKill`:
- Text: "Restore {0} on kill" in multiple languages.
- They mark it stackable and boostable (meaning multiple instances stack and Spice can increase it), and **not offensive** (since it's a buff effect).
- In the subscribe, they set `data.eventPriority = -1`. By default, eventPriority 0 means the effect triggers in normal order; -1 probably ensures it triggers slightly later than other kill-trigger effects (maybe to avoid interfering or to ensure heal happens after kill credit). They also set:
    - `data.effectToApply = mod.TryGet<StatusEffectInstantHeal>("Heal (No Ping)")` – this references a base game status effect that instantly heals without showing the "+HP" popup (the "No Ping" likely means no floating text). So, on kill, it will apply that heal effect.
    - `data.applyToFlags = Self` – meaning the effect is applied to the unit itself (the killer).
    - `data.waitForAnimationEnd = true`, `data.queue = true` – these control the sequencing. Likely it waits for the attack animation to finish before applying the heal, and queues the effect so it doesn't interrupt other actions.

- This status effect is then given to Alison's leader card as we saw (attackEffects). When Alison kills an enemy, the `StatusEffectApplyXOnKill` sees that event, and because `RunEntityDestroyedEvent` returns true (the attacker was Alison), it will run and apply the `InstantHeal` to Alison for the given amount. The amount `{a}` is specified via text insert "<{a}><keyword=health>" meaning it uses the effect's own amount as the placeholder. They didn't explicitly set a `scriptableAmount` or base amount here, but by default, `StatusEffectApplyXOnKill` likely uses the stacks on the card as the amount (they gave 2 stacks in card data). So 2 stacks * Heal effect => heal 2 HP. The text "Restore {0} on kill" will replace {0} with 2 in-game. This matches the intention.

- **“When Hit Apply Demonize To Attacker”** – uses `StatusEffectApplyXWhenHit`:

- Text: provided in multiple languages, “When hit, apply {0} to the attacker”, with insert `<{a}><keyword=demonize>`.
- Marked stackable and boostable, offensive = false (because as a **starting effect** on an ally, they treat it as a buff, not something that makes the unit offensive).
- Subscribe sets: `data.desc` explicitly to a string with formatting (maybe to ensure translation or color, they included `<{0}><keyword=demonize>` in desc), `data.hiddenKeywords = { "Hit" }` (so the card will also show the “When Hit” keyword icon from base game, presumably),
    - `data.effectToApply = mod.TryGet<StatusEffectDemonize>("Demonize")` – uses the base game Demonize effect data. This means when triggered, it will apply the standard Demonize status (which doubles damage taken) to the target.
    - `data.applyToFlags = Attacker` – apply to the attacker that hit the unit.
    - `data.queue = true`, `data.targetMustBeAlive = false` – queue the effect (so it doesn’t happen mid-damage but right after, probably), and allow it to trigger even if the unit with this effect dies from the hit (so Poppy can still demonize her killer even if she is killed by that hit).

- This effect is given to Poppy (2 stacks) and effectively it’s exactly what base Gok’s ability was (they copied base name and logic, just tweaked priority/queue). The result: any time Poppy (or any unit with this effect) is hit, the attacker gets Demonize(2). The game’s event flow: The unit is hit -> `StatusEffectApplyXWhenHit` sees OnHit event -> applies Demonize to the attacker. Since `targetMustBeAlive=false`, even if that hit killed Poppy, the effect still applies (because by default, if the target died, you might not trigger their effects, but they allow it in this case).

- **“On Card Played Deal Random Damage To Target (1-6)”** – uses `StatusEffectApplyXOnCardPlayed`:

- This effect is used by the **Dice of Destiny** item. They gave it a name including “(1-6)” to clarify its range.
- It’s configured with text “Deal {0} random damage (1-6) to the target when played” and insert `<+{a}><keyword=attack>`. The `<+{a}>` likely shows as “+X” damage icon, and keyword=attack for styling.
- Marked not stackable (makes sense, you don’t stack multiple instances of this on one card), not boostable (Spice won’t increase it), but **offensive = true** and **makesOffensive = true**. That means this effect is considered an attack effect; it will cause the card to be treated as an attack (so if it’s on a card in hand, the card is considered offensive).
- In subscribe, they set:
    - `data.dealDamage = true` (so instead of applying a status, this effect will directly deal damage).
    - `data.applyToFlags = Target` (it affects the chosen target of the card).
    - `data.scriptableAmount = new ScriptableAmountRandomRange(1, 6)` – here they introduce a **custom ScriptableAmount** class they made (nested in this file) to produce a random value from 1 to 6 each time. `ScriptableAmountRandomRange` overrides `Get(Entity)` to return a random number in [min, max] on the spot. By assigning this, each time the card is played, the effect will fetch a random damage amount 1-6.

- With this effect, when the card is played on a target, the `StatusEffectApplyXOnCardPlayed` will trigger and because `dealDamage=true`, it creates a `Hit` with that random damage to the target. This was given to the Dice of Destiny item as a start effect (1 stack). So Dice of Destiny effectively does 1-6 damage to whoever you target with it. Integration-wise, that's straightforward: the UI will let you target an enemy (since it's an offensive item), and on play the effect executes.

These examples show how the mod uses **base classes** like `StatusEffectApplyXWhenHit`, `StatusEffectApplyXOnKill`, `StatusEffectApplyXOnCardPlayed`, etc., to implement custom logic without writing new complex C# classes themselves. They just configure the parameters: - Choose the right trigger class. - Set text for clarity. - Tune flags (offensive, stackable, etc.). - If needed, set an `effectToApply` (for applying another effect) or `dealDamage=true` and a custom amount for direct damage. - Set `applyToFlags` to control who gets affected (Self, Attacker, All Enemies, etc., as defined by the enum in base code). - Use `queue` or `waitForAnimationEnd` to synchronize with animations when appropriate. - If the effect is meant to be persistent or stackable (like status icon), leave `stackable` true; if it's a one-time reaction (like these on-hit/on-kill triggers) often they are not meant to stack multiple separate instances on one card (so they sometimes set stackable=false to keep it one instance).

Base game references: Many of the mod's new effects correspond to things in base: - "When Ally is Hit Apply Frost to Attacker" (for Pug Charm) parallels other reaction effects. Base likely has similar for e.g. Ice Charm. - "When Enemy Is Killed Apply Health to Attacker" (they have WhenEnemyIsKilledApplyHealthToAttacker.cs) might mirror the **Muncher charm** effect. - "When Hit Gain Attack (No Ping)" – perhaps a variant of Spice or other buff on hit. - "SummonFoamBullet" or "SummonWisp" etc. – likely using `StatusEffectSummon` class to spawn a unit (like how Snowcake summons Snowboy, etc.). Indeed, they have StatusEffectSummonWisp, which probably sets a `summonCard` to Wisp's CardData and triggers on some condition (maybe a countdown or when destroyed). - The mod also clones or uses existing keyword effects when possible (e.g., Demonize, Snow (freeze), Overload, etc., are in base and they reference them via TryGet).

By using the modding API's builder, all these StatusEffectData objects are created at mod load and added to the game's database. Each is identified by name (with the mod's GUID likely prefixed internally). The mod's cards then refer to these by using `mod.SStack("EffectName", amount)`, which calls `TryGet<StatusEffectData>("EffectName")` under the hood. `TryGet` is smart: for StatusEffects, it first tries `Get<StatusEffectData>(name)` which checks mod-added ones, and if not found, it falls back to base game ones (because they derive DataFile, loaded in Addressable "StatusEffectData" group). This means the mod can retrieve base effects or its own seamlessly. For example, `mod.SStack("Snow", 1)` would yield the base Snow effect.

Creating New vs. Cloning: The mod sometimes could have used `StatusCopy` to clone a base effect and modify it slightly. But the builder approach works similarly and perhaps more cleanly. For instance, "When Hit Apply Demonize" already existed in base for an enemy named Gok with 1 stack. The mod wanted 2 stacks and slightly different behavior (ensuring it triggers even on death), so they created a new effect with the same name. Ideally they could have cloned Gok's effect and changed stack default and targetMustBeAlive, but either method is fine. They just have to ensure the name doesn't collide – since base had exactly "When Hit Apply Demonize To Attacker", there is a slight risk of name collision. However, their `DataCopy` function prefixes clones with GUID, and for Create... it might automatically scope it to mod (not 100% certain without seeing `StatusEffectDataBuilder.Create` implementation). Likely, the mod's content gets their mod reference so even if names match, the GUID in the full internal name differs. So it's safe.

Status Effect Use in Cards: The new status effects the mod defines are assigned to cards in two ways: - **startWithEffects**: The card spawns with that status on itself. E.g., companions have reactive effects like Demonize attacker (Poppy) or apply Ink on turn, etc. These typically are `isStatus=true` so they appear as icons on the card in battle. - **attackEffects**: The effect triggers when the card attacks (or is played). E.g., Alison's heal on kill is an attack effect; Dice of Destiny's random damage is an attack/play effect for an item. Attack effects also show as icons (sword icons) on cards. - **traits**: Some effects might instead be implemented as traits/keywords (though in this mod they mostly treat everything as status effects or static traits like Smackback). Traits like Haste, Wild, etc., are separate `TraitData`. The mod references some base traits (Smackback, Wild, etc.) directly.

The interplay between **StatusEffects and CardScripts**: It's worth noting that **CardScripts** (from Cart**Scripts** in the question, presumably a typo for CardScripts) are used for immediate one-time modifications (like random stat adjustments or giving a crown). They run in `CardData.RunCreateScripts()` when a card entity is instantiated. **StatusEffects** (even "instant" ones) typically either attach to an entity and can persist or trigger over time/events. For example, a `StatusEffectInstantHeal` might apply immediately and then remove itself. In the mod, they rely on status effects for anything that involves game events or repeated triggers, which is correct. CardScripts cannot listen to "on kill" etc., they're not event-driven beyond creation time.

CartScripts vs CardScripts Clarification

The user question mentioned **"CartScripts"** but based on context, this seems to be a typo or misunderstanding. It's likely referring to **CardScripts**, since those are used in the mod (there is no concept of "Cart" scripts in Wildfrost). We have already touched on CardScripts in the leader and companion sections, but here is a clearer overview:

CardScript (Base Class): In Wildfrost, `CardScript` (from **CardScript.cs** in base, likely an abstract class derived from ScriptableObject) defines a method `Run(CardData target)`. The game uses CardScripts primarily in `CardData.createScripts`: after a CardData is cloned for battle, `RunCreateScripts()` will call each script's Run on that card. This allows modifying the card's properties dynamically at the moment it enters play.

Base game implements specific CardScript subclasses such as: - `CardScriptGiveUpgrade` – adds a charm (CardUpgradeData) to the card (e.g., crown or specific charm). - `CardScriptAddRandomHealth` / Damage / Counter – adjust stats randomly (used in daily challenge or charm effects). - Possibly others like `CardScriptChangeCardStats` etc., used in tutorials or events.

The **mod's use of CardScripts:** - **Custom Script:** `CardScriptChangeMainOnCounter` in Helpers is a new subclass the mod created (it inherits CardScript). It overrides `Run(CardData card)` to swap the card's sprite depending on the `card.counter` value. This script does *not* automatically run on some event by default – it runs when `RunCreateScripts` is called (so at card creation). Their logic only does something when the counter differs from lastCounter; since `lastCounter` starts -1, on creation it will definitely set the sprite for the current counter. But how will it update when the counter changes later? Notice the comment *"This script should be called manually from a counter change event or similar"*. Simply putting it in createScripts means it runs once at battle start. To actually update on each turn countdown, the mod would have to call Run again whenever the counter changes. The mod does not show an explicit event hook for counter changes. Perhaps the base game triggers `RunCreateScripts` again on counter changes (less likely), or more plausibly the mod might plan to manually trigger it via some Harmony patch or future addition (they left a note). In any case, on spawn it sets an appropriate image (e.g. if Poppy's counter is 4, it sets the sprite to `poppy3.png` after clamping, etc.). That's an example of using a CardScript for a custom visual effect. - **Base Scripts via Helpers:** As seen with the leader, they use static helper methods to instantiate base game CardScripts. The `CardScriptHelpers` provides `GetGiveUpgradeScript`, `GetRandomHealthScript`, etc., which simply create the respective ScriptableObject and set their parameters. For example, `GetGiveUpgradeScript(mod, "Crown")` creates a CardScriptGiveUpgrade and assigns `upgradeData = mod.TryGet<CardUpgradeData>("Crown")` (fetching the base Crown upgrade). By including that in a card's createScripts, they ensure the card gets that upgrade at runtime. This is how the mod crowns leaders.

Integration of CardScripts: When the game instantiates the leader/companion card at the start of battle: - It clones the CardData (the clone inherits base stats). - Calls `RunCreateScripts()`, which runs each attached CardScript's Run method on that CardData clone. - The RandomHealth script will randomly adjust `card.hp`. The GiveUpgrade script will attach a Crown upgrade to `card` (making it deploy immediately). These modifications happen before the battle fully begins. - After scripts, the card is placed either in hand or on board as appropriate (leaders on board, companions might start on bench or in hand, etc. Leaders being crowned ensures on board).

CardScripts **do not persist** as status icons or effects; they typically self-destruct or are one-time. For example, after giving the crown, that script isn't needed further. Status effects, however, persist on the character (like Poppy's Demonize-on-hit stays on her as a status effect icon with a counter of 2).

The mod's combined use of **CardScripts for static one-time modifications** and **StatusEffects for ongoing reactive or keyword effects** shows an understanding of each system's role.

Summary of Creating and Managing Mods (StatusEffects & CardScripts)

From the above, a new modder should take away: - Use **StatusEffectDataBuilder** to define new effects that trigger on specific game events (turn, kill, hit, etc.) by selecting the appropriate base class. Set the properties so the game knows what the effect does (target, amount, etc.). The mod provides many templates: e.g., to create an effect "on X event do Y", there's likely a base class for it (OnKill, OnTurn, OnDeploy, etc.) and you just fill in Y (the effect to apply or damage). - Use **CardDataBuilder** to create new cards and attach those effects to the card either at start (passive effect on the unit) or on attack. - Use **CardScript** instances to handle things that should happen only once when a card is created or drawn (like giving a charm or randomizing stats). Do not use CardScripts for things that need to happen repeatedly or conditionally in battle – that's what status effects (reactions) are for. - The modding API's builders take care of linking these with the mod, so that all new content is recognized by the game.

Charm Definitions and Usage

Charms (and crowns) in Wildfrost are represented by `CardUpgradeData`. The mod adds several new charms (e.g. Pug Charm, Frost Moon Charm, etc.). These are defined in **Charms/** as static classes using `CardUpgradeDataBuilder`.

Taking **Charm_PugCharm.cs** as an example:

```
var builder = new CardUpgradeDataBuilder(mod)
    .Create("charm-pugcharm")
    .AddPool("GeneralCharmPool")
    .WithType(CardUpgradeData.Type.Charm)
    .WithImage("charms/pug_charm.png")
    .WithTitle("Pug Charm")
    .WithText("When an ally is hit, apply 1 frost to them")
    .WithTier(2)
    .SubscribeToAfterAllBuildEvent(data => {
        data.effects = new CardData.StatusEffectStacks[] {
            mod.SStack("When Ally is Hit Apply Frost To Attacker", 1)
        };
    });
mod.assets.Add(builder);
```

- **Create(id):** Gives an internal ID “charm-pugcharm”. By convention, base game charms often have IDs like “CardUpgradeXYZ” or a unique name; using “charm-name” is fine and consistent in this mod.
- **AddPool("GeneralCharmPool"):** This likely tags the charm to be included in the general distribution (so it can appear in normal charm rewards). It’s possible the modding API uses this to auto-add to the pool, but the mod also manually added these charms to the Charms reward pool for the tribe. Perhaps "GeneralCharmPool" is a base grouping used by the game when any charm can drop.
- **WithType(Charm):** Ensure it’s marked as a Charm type (as opposed to a Crown or other upgrade type).
- **WithImage:** Path to its icon.
- **WithTitle and WithText:** Title is the charm’s name and Text is the effect description. They wrote “apply 1 frost to them” which might be a minor mistake (it should be “to the attacker” given the effect’s name). Or possibly they intended “to them” meaning to the ally? Actually, the effect name says “To Attacker”, so likely the text might be off. Ideally, it should read “apply 1 <keyword=frost> to the attacker” to be precise. But that’s a minor wording issue.
- **WithTier(2):** Charms have tier (rarity) levels 1, 2, or 3 in Wildfrost. Tier 2 suggests this is a relatively rare/strong charm.
- **After build:** They assign `data.effects` – this is how charms apply their effect to a card. A charm’s `effects` array contains `StatusEffectStacks` that will be **granted to any card that has this charm equipped**. So here they give one stack of “When Ally is Hit Apply Frost To Attacker”. That status effect (presumably a new one mod created) will be attached to the unit that equips the Pug Charm. Because the effect is “When Ally is Hit…” and likely has logic applying Frost to the attacker, the net result: *the ally who has this charm, when hit, will apply 1 Frost to whoever hit them.* (Basically a mini Snow charm that only gives 1 Frost retaliatory, but doesn’t prevent damage like Snow does).

> This is similar to base **Snowbelle Charm** (hit back with 1 Snow) or **Spice Charm** (on hit, gain Spice, etc.), but specifically frost on attacker.

By adding this charm's data to mod assets and the reward pool, players can find "Pug Charm" and equip it on a companion. The effect will then be active via the status effect on that companion.

All other charms in the mod follow suit: - Frost Moon Charm, Golden Vial Charm, etc., likely attach different status effects (e.g., maybe GoldenVialCharm might grant "On Kill, gain bling" or something, Frost Moon might grant a Summon or apply Frostburn). We see each charm file uses `.WithText` and assigns some `data.effects`. - They also possibly use different Tier values to indicate power.

One notable integration: The mod's ClassData starting inventory gave **Book Charm** at start. Book Charm likely has a beneficial effect (maybe something like apply Demonize on kill or who knows). We see "charm-bookcharm" is defined and added in code.

Charms vs Crowns: Crowns (which make a card start in hand each battle) are actually also CardUpgradeData of type Crown. The mod doesn't add new crowns (just uses the base crown in scripts). They did include a charm that removes charm limit (as seen in reward pool: CardUpgradeRemoveCharmLimit – which is a base charm). They included all base charms by their IDs in the pool assignment to ensure variety.

Traits/Keywords vs StatusEffects: Some charms or card effects could be implemented either as a trait or a status effect. For example, **Bombard** is implemented as a status effect in base (as seen in Effects sheet: Bombard 1 and 2 used by Krunker, and also listed under Traits in card sheet with (1) or (2) stacks). The line between trait and status is blurred: generally, if it's something that stacks or can be affected by Spice, it's a status effect; if it's binary (has or not) and doesn't stack, it might be a trait. The mod primarily used status effects for new mechanics, likely because the builder doesn't explicitly have a ".WithTraitEffect" and status effects give more control. They did use actual TraitData for things like Smackback, Wild, etc., which are static traits.

For new keywords, the mod might not have added any new TraitData (we don't see evidence of new trait definitions). They mostly re-used existing ones. For example, "Wild" trait (double attack power up after killing something) is base – maybe they gave that to a card. If the mod wanted a new keyword icon, they could define a TraitData and assign it. The provided *Wildfrost Reference.xlsx* has a **Keywords** sheet listing what each keyword does for reference.

Given the scope, the mod didn't introduce wholly new keyword mechanics (aside from naming their effects like "Hit All Enemies" which might correspond to Barrage but not sure if they intended a new keyword icon for it or just an internal effect).

They did include in the docs (Wildfrost Reference.xlsx) a **Keywords** sheet for reference – presumably to ensure they leverage or avoid duplicates.

Comparison with Pokefrost (Reference Example)

The user specifically requested comparing with **Pokefrost** (from WildfrostMods-master references). Pokefrost is another mod which, from a quick glance, uses a more direct approach (perhaps Harmony patching and custom logic). Notable differences:

- **Structure & Style:** Pokefrost's code (e.g., `CardScripts.cs` in Pokefrost) shows classes directly inheriting `CardScript` and overriding `Run` to implement effects (like a foresight ability scanning future nodes). It suggests Pokefrost might be adding new complex behaviors not supported by simple builders (like custom campaign node outcomes or abilities that require scanning game state). It uses Harmony (`using HarmonyLib;`) which means it patches game methods to insert new features (for example, they might patch the campaign map generation to add special battles).
- In contrast, **Mad Family mod avoids Harmony** and sticks to the mod API. This is generally safer and more maintainable, especially if the base game updates. The Mad Family mod doesn't modify base classes, it only extends via provided hooks. Pokefrost's approach is powerful (one can do anything with Harmony), but it requires careful maintenance.

- **Registration:** Pokefrost might not use the reflection auto-register; it might manually patch or register things. It likely was made before the builder API was robust. For instance, we see `AddressableExtMethods.cs` and various custom classes in Pokefrost folder, suggesting it might manipulate addressables or game data directly in code rather than through high-level builders.

- **Integration:** The Mad Family mod integration is clean – just adding new data. Pokefrost adds new campaign nodes (like `CampaignNodeTypePortal.cs` in Tutorial7 or Pokefrost's `CampaignNodeTypeSpecialBattle.cs`). It's more invasive by necessity (for example, adding a new map node type might require patching the map generation). Mad Family mod doesn't alter game flow, it just adds content to existing flows (new tribe, new cards in existing battles, etc.).

Comparison Summary: For an intermediate programmer new to modding: - Mad Family mod serves as a great example of using provided frameworks (builders, events) to add content (**content modding**). - Pokefrost is an example of **behavior modding** – altering game logic, which often requires patching. It's more complex and can break more easily. You can see Pokefrost defines new `CombineCardInBattleSystem`, custom commands, etc., indicating a deeper hack of the game. - If your goal is to create a new tribe or cards, following the Mad Family approach is recommended: minimal patching, use the official mod API. Only patch (like using Harmony) when you need to change something the API doesn't expose (Pokefrost needed it for new map mechanics, which weren't officially supported).

The WildfrostModTutorials presumably cover both approaches – basic content addition vs advanced patching (there is a **Tutorial4-Patching** project in the references, showing when to use Harmony). The Mad Family mod largely sticks to what's taught in Tutorial 1-3 and 5 (basic mod, adding cards/effects/charms, and creating a tribe), which is exactly what an intermediate programmer should start with. Patching (Tutorial4) can be learned later if needed.

Base Game Data Reference (Wildfrost Reference.xlsx)

The provided reference spreadsheet contains a trove of base game data. Key sheets of interest:

- **Cards:** Lists every card's internal name, in-game name, type, and its effects/traits. For example, it shows bosses with their effects ("ImmuneToSnow(1)", etc.), companions with traits like Wild or Barrage, etc. A modder can consult this to see what existing status effects and trait names do. E.g., if you want to give your unit a Barrage (hit all in a row) effect, you'd see in the Cards sheet that e.g. "Krunker" (ClunkerBoss) has trait "Bombard (1)" in Traits column, and in Effects sheet Bombard is a status effect class with notes. This hints that giving Bombard(1) as a starting effect might replicate that. The Mad Family mod indeed used "Hit All Enemies" which likely corresponds to Bombard.
- **Effects:** Lists effect name, description, who uses it, what class it is implemented by, etc.. This is extremely useful for modders to identify which StatusEffectData class and fields to use. For instance, seeing **"When Hit Apply Demonize To Attacker"** in the Effects sheet tells you it's implemented by `StatusEffectApplyXWhenHit` and used by "Gok(1)". A modder reading this knows to use that class and effect for their own card (which the mod did for Poppy).

The spreadsheet shows many "Unused" effects as well – these might be vestigial or reserved. The mod could potentially activate them or copy them. For example, "On Turn Apply Demonize To Enemies" is marked unused – a mod could clone that for a charm or boss.

- **Keywords & Traits:** The **Keywords** sheet likely lists traits and what they do (Smackback, Haste, etc.) and possibly internal names vs display. The **Traits** sheet possibly lists similar info (in Wildfrost, sometimes "TraitData" and "KeywordData" overlap concepts – e.g., Smackback might be a TraitData with a keyword icon).

If a modder wanted to add a new trait/keyword, they could define a new TraitData with behavior. However, creating new behavior for a trait likely requires patching since traits are often checked in code (for example, the game code checks if a unit has the "Haste" trait to reduce its counter).

The Mad Family mod doesn't add new traits, it reuses base ones, which is wise for simplicity.

- **StatusTypes or Status Icons:** Possibly lists which status effects are considered Status vs Reaction vs Keyword. (The sheet names "StatusTypes" might enumerate categories).
- **Reward Pools, CardTypes:** These sheets show base definitions of reward pools and card types. Knowing these ensured the modder uses correct strings like `"Items"`, `"Units"`, `"Charms"` for pool types, and `"Friendly"`, `"Leader"`, `"Item"` for card types. The mod indeed uses those exact strings, matching base data.

Overall, the **Wildfrost Reference.xlsx** serves as a dictionary for the mod creator: - Want to apply a certain effect? Look it up by name to find if it exists or what it's called internally. - Want to ensure you use correct pool names or class names? Check the reference. - It prevents guesswork – for example, knowing that the "Basic" class ID is "Basic" was used to clone the prefab.

For documentation purposes, using this reference can help explain definitions: - **Example:** *Demonize* – In the Effects sheet, Demonize has `offensive=true, makesOffensive=true` and is used by various sources. This means a unit with Demonize is considered a negative status (makes them take double damage). The mod did not modify Demonize, but used it. - **Snow (Freeze)** – There might be an entry

“ImmuneToSnow” which is a status effect making a unit immune to being frozen (the bosses had it). If a mod wanted to create, say, “Immune to Demonize”, they could mimic that structure.

For an intermediate programmer: learning to cross-reference base game data is crucial. The modder of Mad Family clearly did so – they didn’t hardcode values blindly; they often fetched base data via `TryGet` and copied or adjusted. This ensures compatibility (e.g., using the game’s definitions of traits and keywords means the mod’s cards will automatically benefit from any game logic tied to those traits).

3. How to Build a New Tribe Mod (Tutorial-Style Guidance)

Finally, consolidating all of the above into a step-by-step guide for someone with moderate programming experience new to Wildfrost modding:

Step 1: Setup Your Mod Project

- Use the Wildfrost modding template or API (as seen in WildfrostModTutorials). Set up a new C# class library project that references Wildfrost’s assemblies (so you have access to game types like CardData, StatusEffectData, etc.). The Mad Family mod’s `.csproj` likely includes references to the game’s Assembly-CSharp.dll and modding support DLLs.
- Create a main mod class inheriting `WildfrostMod`. Give it a unique GUID and name. This class will manage loading your content.

Step 2: Plan Your Tribe and Content

Decide on: - The **Tribe Name** and theme (e.g., “Mad Family” focused on retaliation and chaos). - **Leader characters** (at least one, could be multiple for variety). - **Starting deck** cards (companion(s) and items the tribe begins with). - Any new **companions** the tribe introduces. - Any new **items** or **charms** that fit the theme. - New **status effects** needed for unique mechanics (or reuse base ones where possible). - You may list base game effects/traits you want to leverage (using the reference to find exact names and behaviors).

For example, Mad Family decided on multiple leaders (family members), a couple of family pet companions (Lulu the dog, Poppy the cat perhaps), a variety of homemade items (foam bullets, tack spray, etc.), and some charms representing family mementos. They aimed for effects like demonizing attackers, summoning a spirit (Soulrose), random damage (dice), etc.

Step 3: Implement Custom Status Effects (if any)

Implement **Effects** first, because your cards will reference them: - For each new effect, create a static class in your `Effects/` folder, e.g. `StatusEffect_OnTurnSummonSoulrose`. - In the static `Register(WildFamilyMod mod)` method, use `new StatusEffectDataBuilder(mod)` and choose a base class with `.Create<>()`. - If no base class does what you need, consider combining multiple or in worst case, you might create a new subclass of `StatusEffectData` (advanced). But the base game covers a lot: *OnTurn*, *OnKill*, *OnHit*, *WhenXApplied*, *Summon*, etc. - Use the fluent API to set the effect’s text (with placeholders `{0}` for stack values and use `<keyword=XYZ>` to insert icons if needed). - Determine properties: `.WithStackable(true/false)`, `.WithCanBeBoosted(true/false)` depending on if Spice (effect bonus) should amplify it. Typically: - Buffs on allies: boostable by Spice (so true if it’s like healing or strength increase). - Debuffs on enemies: often `offensive=true` so that having it makes a unit considered a threat, and `makesOffensive=true` if just having the effect makes the holder considered an attacker (usually for effects that cause damage). - If the effect shouldn't stack multiple separate instances, set stackable false (but you can still give it a count >1 as one instance). - In `SubscribeToAfterAllBuildEvent`, set any specific fields of the effect: - For trigger effects, set `effectToApply` (which itself is a StatusEffectData, e.g., Demonize or a Heal) and appropriate flags (`applyToFlags` like Self, Attacker, AllEnemies, etc.). - Or if the effect directly deals damage, set `dealDamage=true` and provide a `scriptableAmount` (could be a fixed ScriptableAmount or a custom one like random range as done in Dice of Destiny). - If needed, adjust `eventPriority`. A lower priority (e.g., -1) means it triggers later than default, a higher means earlier. For example, multiple "On Kill" effects – you might want buff effects to apply after damage effects, etc. - Use `mod.TryGet<StatusEffectData>("Name")` or `<TraitData>` to fetch base effects/traits if you want to re-use them.

Example: Creating a *"On Kill Gain Shield"* effect:

```
public static class StatusEffect_OnKillGainShield {
    public static void Register(WildfrostMod mod) {
        var builder = new StatusEffectDataBuilder(mod)
            .Create<StatusEffectApplyXOnKill>("On Kill Gain Shield")
            .WithText("Gain {0} <keyword=shield> on kill",
SystemLanguage.English)
            .WithTextInsert("<{a}>")
            .WithStackable(false)
            .WithCanBeBoosted(true)
            .WithOffensive(false)
            .SubscribeToAfterAllBuildEvent<StatusEffectApplyXOnKill>(data => {
                data.effectToApply = mod.TryGet<StatusEffectBlock>("Block"); //
"Block" is the shield effect in game.
                data.applyToFlags = StatusEffectApplyX.ApplyToFlags.Self;
            });
        mod.assets.Add(builder);
    }
}
```

This would create an effect so that if a unit has it, whenever it kills an enemy it gains Block (shield) stacks. We reuse the base "Block" status effect (which prevents the next hit) by fetching `StatusEffectBlock`.

Make sure to add each builder to `mod.assets`. Once all effects are defined, they'll be auto-registered by your ComponentRegistration (discussed later).

Step 4: Define Cards (Companions, Leaders, Items)

For each new **card**, create a static class in `Cards/` (or separate subfolders if you like, as done in the mod).

Companions & Leaders (Unit cards): - Use `CardDataBuilder.CreateUnit(cardId, displayName)`. - Use `.SetSprites(mainSpritePath, backgroundPath)`. Put your art in the mod `assets/images` and reference it. Ensure to include the file extension in path if required by the API. - Set stats via `.SetStats(health, attack, counter)`. If a unit has no attack (support unit), you can set hasAttack false by either passing null or using a separate call, but the builder likely handles null as “no stat”. Similarly for health if needed. Usually all companions have HP. - Choose `WithCardType("Friendly")` for normal companions, `"Leader"` for leader units, `"Enemy"` if you were adding an enemy (not in this mod’s scope), or `"Clunker"` for clunkers (if you add a clunker-type unit). - Set value via `.WithValue(x)` to approximate its strength (this affects game economy). - Flavor text `.WithFlavour("...")` (optional but nice). - Add any innate **traits**: If your unit has a keyword like Haste (acts immediately) or Barrage (hits a whole row), you might add that as a trait stack. E.g., `.SubscribeToAfterAllBuildEvent(data => { data.traits = new List<TraitStacks> { new TraitStacks(mod.TryGet<TraitData>("Haste"), 1) }; });`. - Add **status effects** for passive or triggered abilities: - Use `data.startWithEffects` for effects that the unit has inherently (these will show as status icons on the card). E.g., Poppy’s demonize-on-hit went here. If you gave a unit Snow (it starts frozen), that would go here too. - Use `data.attackEffects` for effects that trigger when the unit attacks (or is played, in case of items). E.g., Alison’s heal-on-kill is an attackEffect. If you wanted a unit to apply poison on hit, that could be an attackEffect (triggered on their attack hitting someone). - Attach **CardScripts** if needed: - For leaders, it’s typical to attach the Crown script and any stat randomizer if desired (like the mod did for fun). The Crown script ensures the leader is auto-deployed. However, note that the game may automatically deploy the leader anyway, so this might be redundant unless you want the crown icon on them. (The mod likely did it to guarantee it, or because the modding API might not automatically mark leaders as deployed). - If you have a unit that should have a dynamic sprite or other special behavior at creation, attach a custom CardScript. Otherwise, most companions don’t need createScripts except leaders. - Finalize with `mod.assets.Add(builder)`.

Items: - Use `CreateItem(cardId, name)`. - SetSprites, Flavour, CardType("Item"), Value similarly. - If the item deals direct damage or has an attack stat, use `.SetDamage(x)`. If it doesn’t deal damage directly (e.g., only applies a status or healing), you might leave attack 0 or not set attack. - Determine how it’s used: - If the item targets an **enemy or ally** directly, usually you don’t need to change `canPlayOn...` flags (by default an item needs a target if `needsTarget=true`). You might set `data.needsTarget = true/false` via builder if needed. - If the item should be played on the board (like a trap or zone), use flags as in Tack Spray: `playOnSlot = true` and disable the others accordingly. If it affects all allies or all enemies without a target (like a global effect), you could set `canPlayOnBoard=true` (board meaning anywhere?) or simply require clicking the bell (some items, like **Storm Globe** in base, target no one and affect all enemies – how they handle that might be `needsTarget=false` and no specific flags, meaning you can play it without selecting a target). - Most item effects will be in `attackEffects` or `startWithEffects`: - If the item’s effect occurs *upon playing it*, you might treat it as an `attackEffect` because when you “attack” with the item, it triggers. However, in the code, both `attackEffects` and `startWithEffects` of items are applied when played. The distinction is usually: - `attackEffects` apply to the target of the item’s attack (for example, **Ink** status on target on hit). - `startWithEffects` on an item will be present on the item itself – which can modify how the item behaves (like the “Hit All Enemies” modifies the attack to splash). - For clarity: If item simply **applies a status to target**, you could make an effect “On Hit Apply X to target” and put it in attackEffects (since the item hitting the target will cause that effect). If item affects multiple or doesn’t require hitting, you might use startWithEffects like Tack Spray did. - Items can also have `data.targets` constraints if needed (to restrict what they can target, e.g., only enemies, only allies). The mod didn’t explicitly set `targetConstraints` in code, but it could have. Instead, they used the boolean flags to broadly control usage. The base game’s items often have TargetMode which is set via scriptable asset; the modding API might allow `.WithTargetMode()` if needed. - After setting up, add builder to assets.

Charms: - Use `CardUpgradeDataBuilder.Create(id)` with a descriptive id. - Set `WithType(Charm)` (or Crown if making a crown). - Set image, title, and description text (text should describe what effect it gives). - Tier for balance. - In `SubscribeToAfterAllBuildEvent`, assign `data.effects` = an array of `StatusEffectStacks` (or `TraitStacks` via a different field if a charm grants a trait rather than status). - If a charm is very simple like "+2 Attack", the game uses a different mechanism (there's a "Boost" charm in base that just adds attack – that might be handled via `data.statBoosts` or something in CardUpgradeData). The mod didn't add pure stat charms, they added effect charms. If you did want a stat charm, you could possibly clone an existing one or use `data.statAdd` fields. - For effect charms, giving a status effect is straightforward. For example, Pug Charm gave the status that applies Frost on hit. A hypothetical **"Thorns Charm"** could give a "When Hit deal 1 damage to attacker" effect (which base has as **Spice Spike** or something). - Add the charm builder to mod assets and consider adding the charm to a reward pool (like they did to Charms pool for the tribe). If you want it to appear for all tribes, you could possibly hook it into a global pool (the mod's AddPool("GeneralCharmPool") might attempt to do that globally, but I suspect it's more for organization; manual pool insertion as they did is more direct).

Step 5: Register the Tribe (ClassData)

Create a static class for your tribe, e.g. `Tribe_MyNewTribe.cs`: - Use `ClassDataBuilder(mod).Create("TribeInternalName")`. - `.WithFlag("Images/yourtribe_flag.png")` for the banner icon. - `.WithSelectSfxEvent(...)` if you want a custom sound (you can reuse an existing event as shown, or if you have a custom sound, that's more involved to add). - AfterAllBuildEvent to set: - `data.id = "TribeInternalName"` (sometimes needed if not set by Create; ensure it matches what you used in Create). - `data.characterPrefab`: If you have a custom prefab or want to clone one: - You can clone the base game's prefab of a similar tribe as done (Basic or Snowdin). For example, if your tribe theme is similar to Snowdwellers, you might clone "Snowdwellers" ClassData's prefab. Cloning Basic is a safe default (Basic is the default human I believe). - Or if you have made a custom prefab (advanced: requires Unity asset creation, outside scope of code), you'd load and assign it. - Use `UnityEngine.Object.DontDestroyOnLoad(gameObject)` on the clone and assign. - `data.leaders`: an array of your leader CardData(s). Use `mod.TryGet<CardData>("leader-id")` for each. Ensure those leader cards are registered before the tribe (the reflection registration doesn't guarantee order, but usually if you name files such that Tribe is registered last or call RegisterAllComponents in a controlled sequence as they did, it works out – in `RegisterAllComponents`, tribes are registered after others due to naming order logic). - `data.startingInventory`: create an Inventory for the starting deck. You can initialize `deck` with `new CardDataList { card1, card2, ... }`. The code in mod simply did `CardDataList list = new CardDataList { ... }` by collection initializer, which works because CardDataList implements ICollection. - Include your intended starting cards. Usually 0/1 leaders (leader not in deck list), some companions, items, etc. Balance the number (base game typically ~6 cards + leader). - `reserve` can be used if you want some cards to be set aside (less commonly used; base Shademancers keep Junkhead in reserve to summon later). - `upgrades`: list any charms to start with. Usually 0, but you can give one as a bonus. - `goldOwed`: if you want the tribe to start in debt (rare, maybe not needed). - `data.rewardPools`: You can either: - Build arrays of RewardPool as the mod did, or - Clone an existing tribe's pools and then modify. For example, maybe your tribe has mostly the same pool as Shademancers but plus one extra card – you could copy Shademancer's ClassData.rewardPools and just insert your card. The mod did a manual assembly likely for full control. - For each RewardPool, instantiate via `ScriptableObject.CreateInstance<RewardPool>()`, set its `type` ("Items", "Units", "Charms", or "Clunkers" for clunker selection, or "Modifiers" for frozen travelers events, etc.), `copies = 1` (almost always 1 except some special multi-drop cases), and populate the `list` with DataFiles (CardData or CardUpgradeData). - Use `mod.TryGet` to gather references. Include your mod cards where appropriate: * Items pool: all item CardData you created that you want to be findable. * Units pool: any companion CardData you made (unless you purposely restrict them). If you want your companions only as starters, leave them out as this mod did. * Charms pool: your new charms (they did). * You can also include base game content to ensure those still drop. The mod basically merged their content into base pools rather than isolating them, which is good to avoid too thin pool or to allow variety. - If your tribe should not get certain base items (maybe thematic reasons), you could omit them. - Ensure `isGeneralPool = true` for pools that are to be used in normal reward generation. If not, the game might ignore that pool (or treat differently). - Add any special pools if needed (like their Snow pools, or a “Pets” pool if your tribe has unique mechanics). - No need to set `data.unlockConditions` or such because mod tribes are usually unlocked by default (unless you implement a progression system).

- Add the ClassData builder to mod assets.

Step 6: Main Mod Class Integration

In your `WildfrostMod` subclass (e.g. MyMod.cs), implement: - `public override void Load()`: Inside: - If using a reflection-based registration like `ComponentRegistration` (as provided in the mod’s Helpers), call `RegisterAllComponents(this)` early to register your StatusEffects, Cards, Items, Charms, Tribe in one go. - To use this, you’d include the `ComponentRegistration` class from the mod (it’s generic enough to reuse, or write your own variant). The Mad Family mod’s `ComponentRegistration` filters by name prefixes which assume you followed the naming convention (`Card_`, `Item_`, etc.). If you did, it will work out of the box. If not, you can manually call each Register: e.g. in Load do `Card_MyUnit.Register(this); ... Tribe_MyTribe.Register(this);` manually. - Call `base.Load()` to let the mod loader do default handling (this likely triggers the building of assets and such). - Add your tribe to the game’s tribe selection:

```
GameMode gameMode = TryGet<GameMode>("GameModeNormal");
gameMode.classes =
gameMode.classes.Append(TryGet<ClassData>("MyTribeInternalName")).ToArray();
```

This appends your ClassData to the list of selectable classes. (The mod did exactly this with "MadFamily".) - If you need to hook any events (like OnEntityCreated to fix images, or others): - For instance, if one of your status effects or mechanics needs an event hook (maybe you want to trigger something when a specific card is drawn, etc.), you can subscribe here using `Events.[Something] += YourHandler`. - Use as sparingly as needed. The provided mod only needed the image fix event. - You might also register any custom **KeywordData** or UI text if you added truly new keywords (the mod didn’t, and doing so might be complex because the game’s UI might not know your new keyword icon unless you supply it and patch the keyword system). It’s usually easier to piggyback on existing keywords.

- `public override void Unload()`: Clean up what you added:
- Remove your tribe from `GameModeNormal.classes` (the mod provided `RemoveNulls` to filter it out).

- Optionally remove any events: e.g., `Events.OnEntityCreated -= YourHandler` if you added one (to avoid it persisting if the mod unloads).
- Remove mod content from reward pools: The mod's `UnloadFromClasses()` loops through all ClassData rewardPools and removes entries where `item.ModAdded == this`. This works because presumably when your mod builds assets, it flags them internally as ModAdded with your mod reference. This ensures once mod is unloaded, those cards won't appear for other tribes inadvertently. Including the mod's provided Helper for this is wise. If not, you'd manually iterate similar to their code.

- Ensure your mod class's `GUID` is unique and doesn't conflict. It's used in naming and identifying mod-added addressable entries (the mod builder likely concatenates GUID and names when finalizing, as seen in `DataCopy` method).

Step 7: Build and Test

Compile your mod DLL, place it and your assets in the Mods folder as required by the Wildfrost mod loader (the exact structure may depend on the loader; if using BepInEx, you put the DLL in plugins and ensure your images are in the right relative path for Addressables to load).

Run Wildfrost and verify: - Your tribe appears in the new game selection (with correct name, icon, and available leader(s)). - Starting deck is correct (cards, stats, effects present). Check that any custom status icons appear and have the right tooltip text. - Play a battle: ensure your status effects trigger as intended (e.g., your OnKill heal actually heals, your OnHit frost applies frost, etc.). Use logging (`UnityEngine.Debug.Log`) if necessary to debug triggers – the mod did for FixImage or warnings in AddAssets. - Check that your items can be played and do what they should. - Check charms: pick up your charm from a chest and equip it, see if it grants the effect. - Balance and adjust: maybe a status lasted too long or stacked incorrectly – you can tweak stack counts or boostability based on testing.

Step 8: Documentation and Refinement

Document each component for players or other modders: - Describe each new card and charm's effect in plain terms (some of this goes into card text already). - If releasing, mention if the tribe is balanced around certain strategies.

From a code perspective, consider comments in code (as the mod did) to explain tricky parts (they left notes referencing tutorials, and crediting sources for methods, which is good practice).

Additional Best Practices and Suggestions:

- **Naming Conventions:** Follow the established patterns. It helps auto-registration and also clarity. For instance, prefixing static classes with category (Card_, Item_, etc.) and effect classes with StatusEffect_ makes it immediately clear what they contain.
- **Mod Compatibility:** Using the mod API and not patching means your mod should be compatible with others that do the same. Only one mod can modify GameModeNormal.classes at a time, but appending to the array is fine (multiple mods can add tribes; there is no inherent conflict except UI

might overflow if too many tribes, but that's hypothetical). Patching, on the other hand, can conflict if two mods patch the same method.

- **Avoid Hardcoding References:** Use `TryGet` to fetch base game data rather than duplicating them. The Mad Family mod reuses base card and effect data for many things (traits, status, base cards in pools) – this ensures if the game updates those, the mod automatically uses updated versions. For example, instead of creating a separate "Demonize" data, they used `TryGet<StatusEffectData>("Demonize")` to get the game's Demonize effect. This is efficient and less error-prone.
- **Performance:** The mod's approach of pre-registering all components in Load and using events is efficient. There's minimal overhead. Reflection for registration happens only once on load. The rest are just object creations. That's fine for this scale of mod.
- **Using Tutorials & Docs:** The WildfrostModTutorials and provided docs (Tutorial1-7) are extremely useful. The mod author clearly followed them (comment references like "see docs/TribeLogicOverview.md" in code). For a new modder, reading those markdown tutorials is recommended. They cover basic mod setup, adding cards and effects (Tutorial2), charms (Tutorial3), patching (Tutorial4 if needed), creating a tribe (Tutorial5), using addressable assets (Tutorial6, e.g., adding new art or localization), and adding new map nodes/events (Tutorial7). The AI can assist by summarizing or clarifying these if any step is confusing.

Important: Always test each part incrementally. For instance, get your new cards working in an existing tribe or practice mode before integrating the tribe, which is a bigger change. Use logs to confirm that Register methods are being called (the mod prints logs when auto-registering each component which can help).

By following the structure exemplified by the Mad Family mod, you ensure your code logic is organized, your new content integrates naturally with the base game, and you maintain readability for future updates or other modders.

4. Conclusion

The **Wildfrost Mad Family Tribe mod** is a well-designed example that showcases how to extend a game with new content while respecting its framework: - It **mirrors the base game's patterns** (using builders to create ScriptableObjects for data, just like the game uses ScriptableObjects for cards and effects). - It keeps mod-specific logic self-contained (no hacky global variables or spaghetti code). - It demonstrates thoughtful use of events and cleanup to remain stable on load/unload. - Code is commented and structured by feature, which is a good practice in mod projects.

Suggestions for Improvement: are minor – e.g., double-checking text vs effect consistency ("them" vs "attacker" in charm text), possibly invoking the sprite-change script on counter update events (to fully utilize that feature), and maybe including mod companions in the recruit pool if desired. But functionally, the mod integrates with Wildfrost well.

By following this documentation and the example provided by the mod, an intermediate programmer should feel comfortable creating a new tribe with custom cards and effects. Always refer to base game references for any effect or trait you want to use – often the functionality already exists and you just need to plug it into your cards. And when in doubt, use the community tutorials or inspect how existing mods like this one handle similar features.

Sources:

- WildfrostMadFamilyTribe Mod source code (for structure and examples)
- Wildfrost Assembly-CSharp decompiled code (for understanding base classes like CardData, StatusEffectData, etc.)
- WildfrostModTutorials (guidance on mod architecture)
- Wildfrost Reference.xlsx (for base game data on cards/effects/keywords to ensure correct usage)
