# Sistema de Evolução de Monstros – Relatório Técnico de Refatoração

## Introdução

O sistema atual de desbloqueio e evolução de monstros encontra-se disperso em várias partes do código, dificultando manutenção e expansão. Este relatório propõe uma refatoração que centraliza toda a lógica de evolução em uma arquitetura coesa, inspirada em mecânicas consagradas de jogos como **Pokémon**, **Digimon**, **Path of Exile** e **Monster Rancher 2**. A nova abordagem combina: - **Evoluções Ramificadas e Condicionais (Digimon)** – múltiplos caminhos evolutivos baseados em atributos e condições específicas.  
- **Valores Individuais/Eforço e Herança de Ataques (Pokémon)** – cada monstro possui atributos únicos (IV) e pode treinar para melhorar (EV)[[1]](https://puo.fandom.com/pt-br/wiki/Sistema_IV_EV_Nature#:~:text=,at%C3%A9%2031%20para%20cada%20status), além de reter habilidades aprendidas ao evoluir.  
- **Árvore de Desenvolvimento e Builds (Path of Exile)** – a progressão permite escolhas estratégicas, com caminhos convergentes e divergentes formando uma “árvore” evolutiva complexa[[2]](https://www.pathofexile.com/passive-skill-tree#:~:text=It%20is%20a%20vast%20web,allocate%20a%20skill%20and).  
- **Aleatoriedade e Crescimento Orgânico (Monster Rancher 2)** – elementos de sorte influenciando o ganho de atributos por nível, tornando o crescimento de cada monstro único e imprevisível.

O objetivo é definir um **módulo único de Evolução** que concentre as definições das linhas evolutivas, requisitos e lógica de cálculo, facilitando futuras alterações e garantindo consistência em todo o jogo.

## Arquitetura Geral do Sistema de Evolução

A nova arquitetura adota uma estrutura centralizada em forma de **árvore de evolução**. Cada espécie de monstro é um nó nessa árvore, contendo metadados e suas possíveis evoluções. As evoluções são modeladas como arestas (ligações) entre nós, com condições (gates) que devem ser satisfeitas para que a transição ocorra. Essa estrutura pode ser representada como um grafo direcionado (não necessariamente uma árvore estrita, pois há **ramificações** e **convergências** onde múltiplas espécies levam a uma única forma evoluída).

**Componentes principais da arquitetura:**  
- **Módulo de Evolução (ex: EvolutionTree.lua ou Unlocks.lua)**: responsável por toda lógica de evolução. Ele carrega a estrutura de dados das espécies e evoluções, verifica condições, calcula chances e aplica evoluções nos monstros. Concentrar neste módulo as regras evita duplicação de código.  
- **Definições de Espécies**: uma estrutura (pode ser um tabela Lua ou arquivos de dados separados) listando cada monstro com seus atributos base, estágio evolutivo, família/raça, movimentos que aprende e modificadores de crescimento.  
- **Definições de Evoluções**: parte dos dados que indica, para cada espécie, quais evoluções estão disponíveis. Inclui o **destino** (espécie alvo), requisitos mínimos (atributos, vitórias, laços, etc.), e parâmetros de cálculo de chance para cada caminho. Essas definições podem ficar junto das espécies ou em estrutura separada, mas serão lidas centralmente pelo módulo de evolução.

A figura abaixo ilustra conceitualmente a estrutura de árvore evolutiva centralizada (cada caixa é uma forma de monstro, com setas indicando evoluções possíveis; requisitos resumidos ao lado das setas):

*Exemplo conceitual de árvore de evolução com ramificações e convergências inspirada em Digimon (várias formas possíveis a partir de uma base, dependendo de atributos e condições).*[[3]](https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412#:~:text=complicated,won%2C%20and%20previous%20evolutionary%20forms)[[4]](https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412#:~:text=While%20not%20immediately%20apparent%2C%20there,meeting%20five%20of%20Devimon%E2%80%99s%20quotas)

*(Na imagem conceitual acima, uma forma base pode evoluir para duas opções diferentes se cumprir certos atributos; ambas as formas intermediárias eventualmente convergem em uma forma final compartilhada.)*

Essa arquitetura permitirá representar evoluções alternativas (um monstro pode evoluir para formas distintas) e evoluções convergentes (formas diferentes levando à mesma evolução final) de forma clara. Toda a lógica de verificação e aplicação das evoluções será feita pelas funções do módulo EvolutionTree, garantindo que o sistema seja consistente em todos os lugares que acionam evolução (subida de nível, itens especiais, etc.).

## Caminhos Evolutivos e Condições

As **linhas de evolução** serão definidas com base em estágios e atributos do monstro, de modo semelhante às digievoluções do **Digimon**. Cada espécie pertence a um **estágio** (por exemplo: Filhote/Fledgling, Crescido/Rookie, Campeão/Champion, Ancião/Elder, Único/Unique) e evolui para espécies de estágio superior conforme determinadas condições:

* **Atributos Mínimos:** Os status do monstro (Força, Inteligência, Defesa, etc.) devem atingir certos thresholds exigidos pela forma evoluída. Por exemplo, uma evolução pode requerer pelo menos 100 de Força e 80 de Defesa.
* **Condições Extras:** Além de atributos, outros fatores condicionais podem ser considerados, inspirados em Digimon. Por exemplo: número de batalhas vencidas (vitórias mínimas), nível de laço/afinidade com o domador (bond), ou até limites de erros de cuidado (max. “care mistakes”) simulando mecânicas de **Digimon World**[[3]](https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412#:~:text=complicated,won%2C%20and%20previous%20evolutionary%20forms). Essas condições adicionais tornam o sistema mais orgânico e estratégico.
* **Ramificações Evolutivas:** Quando um monstro cumpre requisitos para mais de uma forma evolutiva alternativa, o sistema permitirá a *coexistência* dessas possibilidades. Diferente de Digimon World, onde o jogo aplica uma prioridade fixa se múltiplos requisitos são atendidos (forçando o jogador a evitar cumprir certos critérios para alcançar evoluções menos prioritárias[[4]](https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412#:~:text=While%20not%20immediately%20apparent%2C%20there,meeting%20five%20of%20Devimon%E2%80%99s%20quotas)), aqui adotaremos uma abordagem **probabilística** (detalhada adiante) para decidir o resultado quando várias evoluções são elegíveis simultaneamente.
* **Convergência:** O sistema permite que diferentes linhagens possam levar à mesma forma evoluída. Por exemplo, duas espécies de estágio *Campeão* distintas podem evoluir para o mesmo *Elder* único, caso ambas atendam aos requisitos dessa forma. A árvore centralizada facilitará esses cruzamentos, evitando duplicação de dados – a forma final convergente é definida uma vez e referenciada por múltiplos predecessores.

Em resumo, as evoluções deixam de ser sequências lineares rígidas para se tornar uma rede flexível de opções. O jogador poderá explorar caminhos variados e experimentar evoluções raras ao treinar atributos específicos ou cumprir condições especiais, aumentando a profundidade estratégica do jogo.

## Cálculo de Chance de Evolução

Quando as condições para uma evolução são alcançadas, a evolução **não acontece automaticamente**; em vez disso, o jogador recebe a opção de tentar evoluir o monstro manualmente. Se houver apenas uma forma possível, a tentativa tem uma determinada chance de sucesso; se houver múltiplas possibilidades, precisamos calcular a chance de cada uma de forma ponderada. O **cálculo de probabilidade** de evolução considerará vários fatores para equilibrar realismo, controle do jogador e um toque de aleatoriedade:

* **Grau de cumprimento dos atributos:** Quanto mais os atributos do monstro excederem os requisitos mínimos de uma determinada evolução, maior o peso dessa opção. Por exemplo, se uma forma A requer 50 de Inteligência e o monstro possui 80, enquanto a forma B requer 50 de Força e o monstro possui exatamente 50, a forma A receberá uma **chance proporcionalmente maior** por estar com sobra de atributos. Em termos de implementação, pode-se calcular um *score* para cada evolução baseada na proporção atributo\_atual / requisito (capada em 1.0 ou algum máximo), e usar esse score como base da distribuição de probabilidade.
* **Compatibilidade de Raça/Família:** Se a forma evoluída pertence à mesma família ou tipo do monstro base, ela pode ser intrinsecamente mais fácil de alcançar (uma evolução “natural”). Por outro lado, evoluções que mudam drasticamente de raça podem ter um peso reduzido. Essa compatibilidade pode ser modelada como um modificador fixo no cálculo. *Exemplo:* Um dragão evoluir para outro dragão pode ter +10% chance base, enquanto evoluir para uma criatura de tipo diferente teria -10%.
* **Chance Base da Espécie Destino:** Algumas formas podem ser mais raras por design do jogo. Assim, cada evolução terá um modificador de raridade influenciando a chance. Isso permite controlar ocorrências raras independentemente dos atributos. Por exemplo, uma evolução especial lendária pode ter no máximo 20% de chance mesmo com atributos perfeitos, enquanto evoluções comuns chegam a 80-90%.
* **Tentativas Limitadas por Nível:** Para impedir tentativas infinitas e forçar progressão, o sistema implementa um **limite de tentativas**. Por exemplo, apenas **1 tentativa de evolução por nível** do monstro. Se o jogador falhar na chance, deverá ganhar pelo menos um nível antes de tentar novamente. Esse mecanismo (inspirado em Monster Rancher, onde o crescimento ocorre gradualmente e com imprevisibilidade) adiciona tensão e significado ao ato de evoluir – não é garantido, e um fracasso significa investir mais treinamento ou níveis. Além disso, impede abuso de salvar/carregar ou spam de tentativa: é preciso merecer uma nova chance via progresso.

**Como funciona o cálculo internamente:** Suponha que um determinado monstro possa evoluir para duas formas (X e Y). O sistema calculará um peso para X e um peso para Y com base nos critérios acima. Exemplo numérico simplificado: Forma X tem atributos exigidos já alcançados com sobra (score 0.8), alta compatibilidade (+0.1) e raridade comum (+0.1), totalizando peso **1.0**. Forma Y mal atingiu os requisitos (score 0.5) e é de raça distinta (-0.1) mas também comum (+0.1), peso resultante **0.5**. Os pesos são então normalizados e convertidos em probabilidades (X teria 67%, Y 33% neste caso). Ao acionar a evolução, o jogo realiza um sorteio (p. ex., rolagem randomizada) com essas probabilidades para determinar em qual forma o monstro evolui – ou se falha, caso também exista a possibilidade de **nenhuma evolução** acontecer (podemos reservar, por design, uma pequena chance de falha mesmo com uma opção disponível, para refletir a incerteza da natureza).

Se a tentativa falhar (ou o sorteio resultar em “nenhuma evolução desta vez”), o monstro permanece em sua forma atual. O jogador poderá tentar novamente após cumprir a condição de nova tentativa (ex: subir de nível). Esse elemento de sorte controlada torna o sistema similar à criação de builds do **Path of Exile**, onde o resultado final não é linear: há espaço para planejar (maximizando chances via atributos treinados), mas também para surpresas e decisões de risco.

## Papel do Nível e do Treinamento

No novo design, o **nível do monstro é um fator secundário** em comparação ao **treinamento de atributos**. Inspirando-se no paradigma de Pokémon e Monster Rancher, o nível continuará existindo mas principalmente como um indicador de progresso e desbloqueio, enquanto o **poder real** do monstro virá dos seus atributos acumulados e esforço de treino do jogador.

Características deste modelo:  
- **Level-up não garante evolução:** Ao contrário de sistemas tradicionais onde atingir um certo nível automaticamente evolui o monstro, aqui **evoluir não é obrigatório nem garantido** ao subir de nível. Subir de nível pode habilitar a chance de evolução (se requisitos forem cumpridos), mas a evolução só ocorre mediante a tentativa manual do jogador conforme descrito. Assim, o jogador pode manter um monstro em determinada forma se preferir (para aprender certos ataques, por exemplo) e só tentar evoluir quando desejar.  
- **Desbloqueio de Habilidades e Conteúdo:** O nível servirá para liberar novas habilidades (golpes) conforme o monstro cresce, e possivelmente para controlar o acesso a tentativas de evolução e conteúdos do jogo (por exemplo, apenas monstros de nível X podem acessar determinada área ou desafio). Em termos de evolução, poderíamos exigir um nível mínimo além dos atributos para certas transformações – mas uma vez atingido esse nível, continuar subindo não aumenta drasticamente a força por si só, apenas permite mais tentativas ou desbloqueios.  
- **Treinamento de Atributos (EVs):** O incremento principal nos status virá de ações de treino e batalhas, similar ao sistema de **Effort Values** de Pokémon. Cada monstro poderá ganhar pontos de esforço ao lutar ou treinar, os quais aumentam seus atributos permanentemente[[5]](https://puo.fandom.com/pt-br/wiki/Sistema_IV_EV_Nature#:~:text=,lo). Esses ganhos independem do nível, ou seja, um monstro pode ficar significativamente mais forte em seus stats através de treino intensivo mesmo sem subir muitos níveis. Isso reflete o conceito de Monster Rancher onde você podia treinar um monstro extensivamente durante sua vida útil, e o nível (idade, no caso) não era o único determinante de força.  
- **Nível como Capacitador de Treino:** Podemos introduzir o conceito de *Stage Caps* – limites parciais de atributos de acordo com o estágio/nível atual do monstro. Por exemplo, um monstro no estágio **Rookie** (iniciante) só conseguiria treinar certos atributos até um valor máximo apropriado ao seu nível antes de evoluir. Isso evita que um jogador treine exageradamente um filhote para ter stats de nível avançado sem evoluir. Essa ideia já se esboça em nosso código atual (ex.: Races.StageCaps observados no sistema existente). Assim, o nível serve também para “liberar” maior potencial de treino quando a criatura evolui de estágio.

Em resumo, o monstro fica mais forte principalmente pelo **crescimento orgânico de stats via treino e experiência**, enquanto o nível age como um suporte – dando acesso a novas habilidades, possibilitando tentativas de evolução e elevando limites de atributos, mas não ditando sozinho o quão poderoso o monstro é. Isso incentiva o jogador a interagir mais com os sistemas de treino e combate para desenvolver suas criaturas, ao invés de apenas grindar níveis.

## Aprendizado de Habilidades e Herança de Ataques

No sistema refatorado, o aprendizado de habilidades (golpes, técnicas) e sua transferência através das evoluções será tratado cuidadosamente para premiar o progresso do jogador sem penalizar indevidamente por evoluir “cedo ou tarde demais”. As diretrizes principais são:

* **Habilidades por Nível (Learnset):** Cada forma de monstro possui uma lista de ataques que pode aprender em determinados níveis, similar ao *learnset* de Pokémon. Por exemplo, a forma básica pode aprender um ataque simples no nível 5, outro no nível 10, etc., enquanto formas evoluídas desbloqueiam habilidades mais poderosas em níveis mais altos. Esses níveis de aprendizado serão centralizados nos dados da espécie (ex.: tabela de movimentos em cada forma), facilitando consulta e manutenção.
* **Herança de Ataques ao Evoluir:** Quando um monstro evolui, **ele herda os golpes da forma anterior**, mantendo parte do conjunto de habilidades já adquirido. Isso significa que o jogador não “perde” tudo que conquistou antes da evolução. Por exemplo, se um monstro aprendeu **Garra Dupla** e **Investida** em sua forma inicial, ao evoluir ele continuará com esses ataques disponíveis. Além disso, a nova forma adiciona seus próprios ataques potenciais (que poderão ser aprendidos conforme alcance os níveis requisitados).
* *Observação:* Em alguns casos, pode-se decidir que certas habilidades muito específicas de uma forma não fazem sentido na forma evoluída (por questões de anatomia ou temática). Nesses casos, a herança poderia ser parcial – porém, isso deve ser exceção e deixado claro para o jogador. De modo geral, **a maior parte dos ataques será carregada adiante**. Esse conceito vem da ideia de Pokémon de que você pode evoluir sem perder golpes aprendidos, e também de Digimon, onde técnicas aprendidas muitas vezes persistem entre digievoluções.
* **Benefício de Esperar para Evoluir:** Para equilibrar e dar opção ao jogador, se decidir **não evoluir imediatamente**, a forma atual pode aprender alguns ataques *mais cedo* ou exclusivos. Por exemplo, suponha que a forma básica aprenda “Ataque Rápido” no nível 12, mas sua evolução aprenderia esse mesmo ataque apenas no nível 8 (por ser mais forte, aprende antes) – nesse caso não há perda. Porém, pode haver ataques que **só a forma básica pode aprender** (inacessíveis se pular aquela forma). Isso encoraja decisões táticas: às vezes atrasar a evolução para obter certo golpe, ou evoluir logo para ganhar acesso a outros atributos e habilidades.
* **Implementação da Herança:** Quando o monstro evolui, seu objeto/dados de monstro pode simplesmente manter a lista de ataques conhecidos intacta. A única diferença é que agora sua espécie mudou, então ao ganhar níveis daqui em diante, o jogo buscará na nova espécie quais ataques destravar. Para garantir que ataques exclusivos de formas anteriores não sejam esquecidos, podemos armazená-los junto a um flag de origem. Por exemplo, um ataque herdado poderia permanecer na lista mas marcado como “legacy” se a nova forma normalmente não teria acesso; assim, sistemas de equilíbrio poderiam lidar com isso se necessário.
* **Ataques Inatos vs Aprendidos:** Algumas formas podem ter ataques inatos (por exemplo, um dragão adulto pode já saber “Sopro de Fogo” ao evoluir, mesmo sem ter aprendido antes). O sistema de evolução pode conceder automaticamente certos golpes ao completar a evolução como bônus. Esses casos seriam definidos nos dados de evolução – por exemplo, a entrada de evolução poderia listar *ataques bônus a conceder*. Isso serve para tornar evoluções mais empolgantes (o jogador evolui e imediatamente ganha um novo poder característico daquela forma).

Em síntese, a **herança de ataques** garante continuidade na progressão do jogador. O treinamento investido em aprender habilidades não é desperdiçado pela evolução; ao contrário, a evolução expande o repertório, preservando o que já foi conquistado. Assim, o jogador sente que cada estágio constrói sobre o anterior, refletindo a progressão natural (inspirado na transferência de técnicas dos Digimon e no sistema de IV/EV do Pokémon, onde um Pokémon evoluído mantém os EV treinados e continua construindo em cima daquele fundamento[[1]](https://puo.fandom.com/pt-br/wiki/Sistema_IV_EV_Nature#:~:text=,at%C3%A9%2031%20para%20cada%20status)).

## Estrutura de Dados Centralizada da Árvore de Evolução

Para suportar todas as mecânicas acima, é crucial termos os dados estruturados de forma clara e acessível. A proposta é criar uma **estrutura de arquivos** modular para definir todas as informações de formas de monstro e suas evoluções, evitando lógica espalhada. A seguir, sugere-se uma possível organização:

* **Arquivo de Definições de Espécies (ex: MonsterForms.lua):** Contém uma tabela/lista de todas as formas de monstros no jogo. Para cada espécie, armazenamos:
* Nome interno e nome exibido (display name).
* Estágio (Fledgling, Rookie, Champion, etc.).
* Família/Tipo base (p. ex.: Dragão, Besta, Elemental) – usado para compatibilidade de raça.
* Atributos base iniciais (força, defesa, etc. no nível 1, ou um valor base para cálculo).
* Multiplicadores ou ajustes por forma (ex.: forma X tem 1.2× HP ganho por nível, etc.).
* Dados de crescimento de atributo (por exemplo, dados de rolagem por nível para cada stat, ou classificação de crescimento).
* Lista de ataques que pode aprender, e o nível para cada (learnset).
* Possivelmente flags como isSummonable (se pode ser obtido diretamente ou só via evolução).
* **Arquivo de Definições de Evolução (ex: EvolutionTree.lua):** Estrutura que define as ligações evolutivas. Pode ser uma tabela mapeando o nome da espécie atual para uma lista de possíveis evoluções. Cada entrada de evolução detalha:
* **target**: nome da espécie alvo (forma evoluída).
* **requiredStage**: estágio requerido atual para que a evolução seja válida (por exemplo, só pode evoluir quando a atual for Champion – isso previne “pular” estágios sem ordem).
* **gates**: condições necessárias (sub-tabela contendo possivelmente minStats com valores mínimos por atributo, minWins para vitórias, minBond para laço, maxCareMistakes etc., conforme discutido nas condições). Se qualquer condição não for atendida, essa evolução não fica disponível.
* **chanceWeights** (ou função weights): os parâmetros de cálculo de chance específicos daquela evolução. Isso pode ser uma estrutura fixa (ex.: peso base e modificadores) ou uma função/lambda que calcula o peso dado o contexto do monstro (ctx). No código existente já havia exemplos de weights=function(ctx)... calculando um valor[[6]](file://file_00000000d82061f58e24d2bf2f45c240#:~:text=totalGates%3DtotalGates%20%2B%201%20end%20end,minWins%29progress%3Dprogress) – podemos formalizar isso. Por exemplo, definir uma fórmula padrão mas permitir override por espécie se necessário para casos especiais.
* **inheritMoves**: opcionalmente, lista de ataques a herdar ou bônus ao evoluir (caso alguma forma conceda um golpe automaticamente).
* **otherEffects**: outros efeitos da evolução, se houver (ex.: resetar idade, modificar expectativa de vida no caso de um sistema de longevidade, etc.).

Com essa separação, o **módulo EvolutionTree** carrega ambos os conjuntos de dados. O ciclo básico é: dado um monstro de espécie X, consulta EvolutionTree[X] para obter as evoluções possíveis. Para cada opção, verifica os gates usando os atributos e status atuais do monstro. As que não satisfazem são descartadas; as válidas são então avaliadas pela fórmula de chanceWeights para produzir probabilidades. O módulo então expõe métodos como CanEvolve(monster), GetEvolutionOptions(monster), AttemptEvolve(monster, choice) etc., que o restante do jogo pode chamar.

**Vantagens da centralização dos dados:**  
- *Facilidade de ajuste:* designers podem equilibrar o jogo mexendo nesses arquivos de dados (por exemplo, aumentando um requisito de atributo ou alterando uma chance base) sem tocar na lógica de código.  
- *Clareza:* visualizar todas as evoluções em um só lugar facilita entender todo o esquema evolutivo do jogo, similar a olhar a Pokédex ou uma wiki de Digimon com todas linhas evolutivas.  
- *Reutilização:* formas convergentes não precisam ser definidas múltiplas vezes. No data, duas espécies diferentes podem ter o mesmo target em sua lista de evoluções – o sistema tratará isso naturalmente.  
- *Validação:* O módulo pode incluir uma função de validação que ao iniciar percorre toda a EvolutionTree e checa consistência (por exemplo, referências a espécies que não existem em MonsterForms, ou ciclos de evolução inválidos). Isso ajuda a detectar erros de configuração rapidamente.

Como exemplo de definição, poderíamos ter algo assim (em pseudocódigo Lua simplificado):

-- MonsterForms.lua (exemplo resumido)  
MonsterForms = {  
 Hopling = {stage="Fledgling", baseFamily="Beast", isSummonable=true,  
 baseStats={HP=30, Str=10, Def=8, Int=5, Speed=12, Skill=10},  
 growthDice={HP="d6", Str="d4", Def="d4", Int="d3", Speed="d6", Skill="d6"},  
 learnset = { [3]={"Coice"}, [7]={"Mordida"}, [12]={"Chute Duplo"} } },  
 Pugilhare = {stage="Rookie", baseFamily="Beast", isSummonable=false,  
 baseStats={HP=45, Str=18, Def=12, Int=6, Speed=20, Skill=15},  
 growthDice={HP="d8", Str="d6", Def="d4", Int="d3", Speed="d8", Skill="d8"},  
 learnset = { [5]={"Soco Rápido"}, [10]={"Fúria do Coelho"} } },  
 -- ... outras definições ...  
}

-- EvolutionTree.lua (exemplo resumido)  
EvolutionTree = {  
 Hopling = {  
 { target="Pugilhare", requiredStage="Fledgling",  
 gates={ minStats={Skill=50, Speed=45} },  
 chanceWeights=function(ctx)  
 local s=ctx.stats;   
 local skillScore = math.min(s.Skill / 50, 1);   
 local speedScore = math.min(s.Speed / 45, 1);  
 -- peso base 0.5 + média dos scores (até +0.5)  
 return 0.5 + 0.5 \* ((skillScore+speedScore)/2);  
 end  
 }  
 },  
 Pugilhare = {  
 { target="Strikeron", requiredStage="Rookie",  
 gates={ minStats={Strength=100, Skill=90}, minWins=6 },  
 chanceWeights=function(ctx)   
 local s=ctx.stats;  
 local strScore = math.min(s.Strength/100,1);  
 local sklScore = math.min(s.Skill/90,1);  
 local winsScore = math.min((ctx.wins or 0)/6, 1);  
 return 0.4 + 0.6 \* ((strScore+sklScore+winsScore)/3);  
 end  
 },  
 -- (Imagine que Pugilhare pudesse ter outra evolução alternativa aqui, seria listado como outro item do array)  
 },  
 -- ... continua para outras espécies ...  
}

No exemplo acima, vemos **Hopling** evoluindo para **Pugilhare** se atingir 50 de Skill e 45 de Speed (condição nos gates). A fórmula de chance dá 50% base + até 50% extra conforme os atributos atendam o requisito (se estiver exatamente no mínimo, 50%; se exceder ambos bastante, chega perto de 100%). Já **Pugilhare** para **Strikeron** requer Strength e Skill elevados e pelo menos 6 vitórias; a chance base é 40% + até 60% conforme quão perto dos stats e vitórias necessárias ele está. (Se houvesse outra alternativa de evolução para Pugilhare, seria outro objeto na lista EvolutionTree.Pugilhare.)

Essa estrutura modular e declarativa torna clara a progressão e facilita tanto ajustes manuais quanto futuras extensões do sistema (por exemplo, adicionar itens especiais que interfiram em evoluções – isso poderia ser incorporado nos gates ou na função de weights, verificando algum campo em ctx que indique uso de item).

## Exemplo de Linha Evolutiva Detalhada

Para ilustrar o funcionamento completo do sistema proposto, considere a seguinte linha evolutiva hipotética de uma família de monstros do tipo “Besta Lutadora”:

* **Fase 1 – Hopling (Fledgling):** Um pequeno monstro coelho boxeador. Atributos base moderados em velocidade e técnica (Skill), aprende golpes básicos de combate corpo-a-corpo. É uma forma **Invocável** (pode ser obtida diretamente).
* *Evolução:* Ao treinar Hopling até ter **Skill ≥ 50** e **Speed ≥ 45**, o jogador pode tentar evoluí-lo. Essas condições representam tornar-se ágil e técnico o bastante. Suponha que Hopling atingiu nível 8 com treinamento, alcançando Skill=52 e Speed=47. Ele cumpre os requisitos mínimos, então a evolução é habilitada.
* *Tentativa:* O jogador clica no “botão de Evolução” disponível (1 tentativa por nível). Com os atributos um pouco acima do mínimo, o sistema calcula, por exemplo, **70% de chance de sucesso** para evoluir em **Pugilhare**. Se falhar, Hopling permanece assim até o próximo nível, quando poderá tentar de novo (com chance talvez melhor se os atributos aumentarem mais).
* **Fase 2 – Pugilhare (Rookie):** Forma evoluída de Hopling, um coelho mais robusto e bípede que luta com socos. Ao evoluir, herdou os ataques de Hopling (ex: **Coice**, **Mordida**, **Chute Duplo**), e pode agora aprender ataques novos exclusivos de Pugilhare (como **Soco Rápido** no lvl 5, **Fúria do Coelho** no lvl 10, etc., conforme sua lista). Seus atributos base aumentaram – por exemplo, um multiplicador de +20% em HP e +15% em Força foi aplicado ao evoluir, além de ganhar um bônus fixo de +5 em Defesa simbolizando “fortalecimento” pela evolução. Todos os atributos são recalculados: os valores de Hopling são ajustados pelos novos multiplicadores de Pugilhare e somados eventuais bônus, resultando em um monstro bem mais forte imediatamente após evoluir.
* *Evoluções:* Pugilhare, como Rookie, possui pelo menos uma evolução possível: **Strikeron (Champion)**. Para evoluir, requer **Strength ≥ 100**, **Skill ≥ 90** e pelo menos **6 vitórias** em batalha. Vamos supor que ao treinar Pugilhare até nível 15, o jogador alcança esses stats (graças a treino intenso em força e diversas lutas vencidas na arena). Então, a evolução para Strikeron fica disponível.
* *Tentativa:* Agora suponha que, além de Strikeron, os designers colocaram uma evolução alternativa secreta para Pugilhare: **Longhare** – um lutador mais focado em velocidade – que requer **Speed ≥ 100**, **Skill ≥ 80** e pelo menos **4 vitórias**, e é de família diferente (Talvez uma mistura de Besta com Fada, por exemplo). Se o jogador tiver treinado Pugilhare de forma equilibrada, é possível que ambos os conjuntos de requisitos estejam cumpridos, habilitando **duas opções de evolução**. O sistema então calculará chances relativas: Strikeron (força bruta, mesma família “Besta”) vs Longhare (foco em agilidade, família distinta). Digamos que o Pugilhare do jogador tenha Strength 102, Skill 95, Speed 101, 10 vitórias. Ele excede ambos requisitos; para Strikeron os atributos-chave estão ~100% e vitórias mais que suficiente, para Longhare a Speed está 101% do req e Skill 118% do req mínimo, vitórias ok. O módulo atribui pesos – Strikeron ganha bônus por ser da mesma família (compatibilidade alta) e talvez ligeiramente maior por mais vitórias que o necessário; Longhare ganha peso por Speed bem alta mas talvez penalidade por família menos compatível. O resultado: Strikeron 55% vs Longhare 45% de chance, por exemplo. Ao clicar “Evoluir”, o jogo sorteia o resultado. **Ambos os resultados são bons, mas diferentes**: Strikeron terá stats focados em força/defesa e alguns ataques únicos de pugilismo, Longhare seria mais veloz/esquivo com golpes de agilidade. O jogador influenciou a distribuição através de como treinou o monstro, mas há um elemento de surpresa.
* **Fase 3 – Strikeron (Champion):** A forma evoluída padrão de Pugilhare, um “campeão” boxeador, bípede musculoso. Se Pugilhare evoluiu para Strikeron, este herdará **todos os golpes aprendidos até então** (inclusive qualquer especial que Hopling tinha passado para Pugilhare) e agora desbloqueará golpes de nível Champion (por exemplo **Gancho Demolidor** no lvl 20, **Defesa Férrea** no lvl 25, etc.). Strikeron também ganha outro incremento de stats ao evoluir (p. ex.: outro +20% HP, +10% Força, etc.).
* *Evoluções:* Strikeron poderá evoluir para **Monarchare (Elder)**, um mestre lutador, caso alcance **Strength ≥ 180**, **Skill ≥ 160** e **12 vitórias**. (Novamente, poderia haver alternativa, mas vamos seguir linear para o exemplo.) Ao atingir esses parâmetros, o jogador tem 1 tentativa por nível para evoluir. Supondo sucesso, chega ao estágio Elder.
* **Fase 4 – Monarchare (Elder):** Uma forma veterana, “Monarca das Lebres Lutadoras”, combinando força e técnica em altíssimo nível. Herdou golpes de Strikeron e anteriores, e aprende talvez suas últimas habilidades nos níveis altos (próximos do limite do jogo). Como Elder, resta apenas evoluir para um possível estágio Unique, se existir.
* *Evolução:* Monarchare poderia ter uma evolução final **SteelBoxer (Unique)** se atingir **Strength ≥ 250**, **Skill ≥ 220** e um alto nível de **laço (Bond ≥ 50)** com o treinador, indicando extrema confiança. Além disso, não pode ter mais que 3 erros de cuidado (parâmetro maxCareMistakes = 3 por exemplo) – um requerimento inspirado diretamente em Digimon World, onde cuidar bem do Digimon influencia suas evoluções[[7]](https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412#:~:text=HP%2C%20MP%2C%20attack%2C%20defense%2C%20speed%2C,won%2C%20and%20previous%20evolutionary%20forms)[[6]](file://file_00000000d82061f58e24d2bf2f45c240#:~:text=totalGates%3DtotalGates%20%2B%201%20end%20end,minWins%29progress%3Dprogress). Esses requisitos tornam a evolução final desafiadora. O jogador provavelmente terá que maximizar EVs de força e habilidade, lutar muitas batalhas para aumentar o bond e evitar deixar o monstro infeliz. Quando todas condições são satisfeitas, a chance de evolução pode ser alta (ex: 90%, dado o esforço enorme investido). O jogador tenta evoluir e, se tudo der certo, obtém a forma final.
* **Fase 5 – SteelBoxer (Unique):** Forma final convergente de toda a linhagem “coelho boxeador” – um guerreiro de aço. Não possui evoluções posteriores. Nesta fase, o monstro atinge seu potencial máximo. Os atributos recebem um último acréscimo/multiplicador, e todos os golpes anteriores permanecem disponíveis. SteelBoxer pode ter também alguns ataques exclusivos finalísticos (por exemplo **Impacto Meteórico**) aprendidos nos níveis mais altos.

Nesse exemplo completo, vemos diversos aspectos em ação: thresholds de atributos, condições extras (vitórias, laço, erros), chance envolvendo múltiplas opções, herança de habilidades e crescimento de stats por etapas. A implementação centralizada proposta suportaria tudo isso configurando os dados adequadamente no MonsterForms e EvolutionTree. O resultado final para o jogador é uma experiência rica: há **transparência** (eles podem saber requisitos, talvez via uma interface de dicas), mas também **imprevisibilidade suficiente** para tornar cada evolução emocionante. Além disso, a estrutura facilita incluir conteúdo novo – imagine adicionar um item especial que, se usado, aumenta temporariamente a compatibilidade com certa família para melhorar chance de uma evolução rara, por exemplo – isso seria apenas mais um fator no cálculo de chance sem alterar a estrutura geral.

## Crescimento de Atributos e Sistema de IV/EV

Para dar suporte a todo esse sistema complexo de evolução e garantir diversidade entre indivíduos, o crescimento de atributos de cada monstro combinará abordagens determinísticas e aleatórias, inspirando-se fortemente no sistema de **IV/EV de Pokémon** e nas mecânicas de **Monster Rancher**:

* **Atributos Base por Raça:** Cada espécie (forma) tem um *baseline* de atributos que define seu perfil. Alguns são naturalmente mais fortes, outros mais rápidos, etc. Esses valores base servem tanto para o nível inicial quanto para escalar o potencial. Por exemplo, se Hopling tem base 30 HP e Strikeron base 120 HP, sabemos que um Strikeron típico é quatro vezes mais resistente que um Hopling típico. Esses status base correspondem aos *Base Stats* do Pokémon[[1]](https://puo.fandom.com/pt-br/wiki/Sistema_IV_EV_Nature#:~:text=,at%C3%A9%2031%20para%20cada%20status), e no cálculo final de atributos podem ser somados a outros componentes para obter o total.
* **Multiplicadores por Forma/Evolução:** Além do valor base, podemos definir **multiplicadores de crescimento** específicos por raça ou por estágio evolutivo. Por exemplo, ao evoluir, aplicar um multiplicador imediato nos atributos atuais (como descrito nos exemplos) ou ajustar a fórmula de ganho por nível. Esses multiplicadores refletem tendências raciais – p.ex., um dragão pode ter um multiplicador de HP maior que uma fada, indicando que dragões ganham mais HP por nível. No código, isso pode ser um campo tipo growthMult dentro de MonsterForms indicando percentuais extra para cada stat na evolução ou por nível.
* **Rolagem Aleatória de Stats por Nível:** Para incorporar **aleatoriedade e crescimento orgânico**, implementaremos ganhos semi-aleatórios de atributos a cada subida de nível. Cada espécie terá um “dado de vida” ou valores aleatórios de incremento para cada stat. Por exemplo, Hopling ao subir de nível rola um **dado de 6 faces (d6)** para determinar quanto HP ganhará, enquanto uma espécie maior como um Golem poderia rolar um **d10** para HP. Da mesma forma, pode-se rolar um d4 ou d3 para atributos como inteligência que crescem mais lentamente. Isso significa que dois Hoplings no nível 10 podem ter HP ligeiramente diferente conforme a sorte das rolagens acumuladas – trazendo individualidade, como em Monster Rancher 2 onde os ganhos de treino nunca eram exatamente os mesmos entre monstros. Esse método é inspirado no sistema de **hit dice** de RPGs de mesa e confere um elemento de surpresa no leveling. Importante: definiremos limites e médias para que a aleatoriedade não quebre o equilíbrio (por exemplo, um Hopling nível 10 típico terá ~+30 HP totais, variando talvez ±5).
* **Sistema IV (Valor Individual):** Ao nascer/ser gerado, cada monstro recebe um modificador oculto e permanente em seus atributos – seus **IVs**. Em Pokémon, IVs variam por status de 0 a 31 e representam a genética única do Pokémon[[1]](https://puo.fandom.com/pt-br/wiki/Sistema_IV_EV_Nature#:~:text=,at%C3%A9%2031%20para%20cada%20status). Aqui podemos adotar lógica similar: gerar um pequeno bônus fixo para cada atributo do monstro, podendo ser na faixa de, digamos, 0 a 5% do valor base ou alguns pontos extras. Isso fará com que, mesmo entre monstros da mesma espécie e nível, haja diferenças individuais (um pode ter nascido mais forte, outro mais resistente, etc.). Esses IVs não mudam com o tempo e ao evoluir continuam afetando os stats do monstro. Por exemplo, um Hopling com IV de +3 em Velocidade ficará ligeiramente mais rápido que a média para sempre, inclusive quando virar Pugilhare e além (podemos recalcular seu impacto conforme a nova forma, mas a vantagem percentual permanece).
* **Sistema EV (Valor de Esforço):** Complementando os IVs, os **EVs** representam o esforço de treinamento do jogador. Cada batalha ou sessão de treino concede pontos de esforço em um ou mais atributos, acumulando ao longo da vida do monstro[[5]](https://puo.fandom.com/pt-br/wiki/Sistema_IV_EV_Nature#:~:text=,lo). Esses EVs aumentam gradualmente os atributos – por exemplo, a cada 4 pontos de EV em Ataque, o monstro ganha +1 no atributo de Ataque (similar à regra clássica de Pokémon[[5]](https://puo.fandom.com/pt-br/wiki/Sistema_IV_EV_Nature#:~:text=,lo)). Podemos equilibrar diferente (talvez 1 ponto de atributo por cada 1 EV, ou usar escalas maiores), mas o conceito é: quanto mais você treina um atributo específico, mais aquele atributo vai crescendo além do crescimento por nível. Há normalmente um limite para EVs por monstro ou por atributo, para evitar crescimento infinito – por exemplo, poderíamos impor que no máximo 50% do ganho total de um atributo venha de EV, ou um valor absoluto teto. Isso é detalhe de balanceamento. O importante é que EVs dão ao jogador **controle direcionado**: se quiser um monstro tanque, foque treinos em HP e Defesa; se quiser um atacante, treine Força e Speed, etc.
* **Interação com Evolução:** Quando o monstro evolui, seus IVs e EVs **permanecem** com ele (afinal, fazem parte de sua individualidade e histórico de treino). Contudo, o efeito numérico deles pode mudar sutilmente porque a fórmula de cálculo dos atributos agora usa o base/multiplicador da nova forma. Em outras palavras, ao evoluir recalculamos o atributo total do monstro somando: (Base da nova forma \* nível) + (bônus aleatórios ganhos por nível rolados) + (IV) + (EV). Se a base da nova forma for maior, os EVs investidos naquele atributo podem render mais pontos absolutos (um EV poderia valer mais em uma forma avançada). Isso **recompensa** o treinamento prévio: um monstro bem treinado antes da evolução fica proporcionalmente mais forte na nova forma do que um monstro que evoluiu “cru”. Essa propriedade é análoga ao Pokémon – treinar EV antes de evoluir faz o Pokémon evoluído ficar mais forte do que se não tivesse EVs.
* **Exemplo de Cálculo de Stat:** Suponha um Pugilhare nível 15 evoluindo para Strikeron. Antes da evolução, ele tinha 100 de Força. Digamos que a base de Força de Pugilhare é 50 e de Strikeron é 80 (valores hipotéticos), e que Pugilhare tinha acumulado 20 EVs em Força e possuía IV 5 em Força. Como Pugilhare nível 15, sem EV/IV ele teria talvez ~75 de força baseado no crescimento natural + rolagens; os 20 EVs podem ter dado +5 e IV +5, chegando em 85, e talvez algum bônus de evolução para bater 100. Agora, ao se tornar Strikeron, recalculamos: Strikeron base 80 com nível 15, mais IV 5, mais EV (mantém 20, que continuam valendo +5 se mantivermos mesma conversão), e aplicamos um ajuste de evolução (talvez multiplicador 1.1 se definimos que Strikeron tem 10% a mais em Força por nível que Pugilhare). O resultado poderia ser algo como 120 de Força após evoluir. Portanto, o monstro fica significativamente mais forte, carregando todo benefício de sua genética e treino.

Em resumo, **cada monstro é único** e o sistema de atributos reflete isso: a base da espécie define seu potencial, mas dois indivíduos da mesma espécie podem divergir pelos IVs (diferenças inatas) e pelos EVs (diferenças de treinamento). O uso de aleatoriedade controlada nos ganhos por nível adiciona variação extra, evocando a sensação de *“este monstro cresceu de forma especial”*. A combinação desses fatores provê profundidade de RPG: jogadores engajados podem otimizar treino de EV para builds específicas (como no Pokémon competitivo), enquanto jogadores casuais ainda verão progresso significativo apenas lutando e subindo nível. A centralização desses cálculos pode ficar em um módulo de **Status/Stats**, ou dentro do próprio Evolution/Forms, contanto que seja acessível para consulta (por ex., ao exibir a ficha do monstro mostrando IVs, EVs e atributos totais).

## Integração com o Sistema de Combate e Status

Como o sistema de combate já foi modularizado no projeto, a integração do novo sistema de evolução deve ser feita de modo a **não quebrar funcionalidades existentes**, mas sim comunicar-se claramente com os módulos de combate e status. Pontos-chave de integração:

* **Atualização de Atributos pós-evolução:** Quando um monstro evolui, seus atributos base, modificadores e possivelmente nível de estágio mudam imediatamente. O módulo de evolução deve acionar uma rotina de recalculo dos status do monstro. Se há um módulo separado para cálculo de atributos (por exemplo, uma função UpdateStats(monster) que re-soma base+IV+EV+equipamentos, etc.), ele deve ser chamado. Isso garantirá que o **sistema de combate** passe a usar os novos valores (HP máximo, ataque, defesa, etc.) na próxima batalha. Importante: se o monstro estiver **no meio de uma batalha** durante a evolução (caso se permita evoluir em batalha, como alguns jogos fazem em momentos específicos), deve-se decidir se a evolução recupera HP ou não, reseta status negativos, etc. Uma abordagem comum é restaurar pelo menos parte do HP ao evoluir (no anime/jogos Digimon, evoluir costuma curar completamente; em Pokémon não há evolução no meio da batalha exceto mega-evoluções temporárias). Essas regras devem ser definidas claramente e implementadas. No contexto atual, talvez a evolução ocorra fora de combate, então é mais simples: podemos simplesmente atualizar tudo instantaneamente.
* **Persistência de Status Efetivos:** Se o monstro tinha status temporários antes de evoluir (buffs, debuffs, estados alterados), precisamos decidir o que acontece. Provavelmente, se a evolução é manual fora de combate, não haverá efeitos ativos. Mas se implementarmos evolução em combate, por exemplo, um buff de ataque contínuo deveria ou não permanecer? Em Digimon, ao digievoluir normalmente zera-se quaisquer modificadores temporários, pois é “um novo corpo”. Poderíamos seguir essa lógica: evolução remove condições negativas e positivas, dando um “fresh start” na luta. Isso deve ser documentado e comunicado ao jogador para não haver confusão.
* **Identidade e Referências no Sistema de Combate:** O monstro evoluído pode ser representado internamente pelo mesmo objeto/ID, apenas com seu campo de espécie alterado. O sistema de combate e outros sistemas (UI, inventário, etc.) que referenciam aquele monstro devem receber notificação da mudança para atualizar nomes, ícones, modelos 3D/2D, etc. Uma forma de implementar isso é o módulo de evolução disparar um **evento** global, por exemplo Event.MonsterEvolved(monster, oldSpecies, newSpecies). Listeners como a interface gráfica de batalha, o módulo de renderização de personagem, ou o log de eventos poderiam capturar isso e reagir (atualizando o retrato do monstro, escrevendo “Seu Hopling evoluiu para Pugilhare!” no feed, etc.). Essa abordagem mantém o acoplamento baixo – o Evolution module não precisa saber quem vai atualizar a UI, apenas emite o evento.
* **Compatibilidade com Sistemas de Combate Existentes:** É importante verificar se o combate leva em conta algo como categoria de monstro, família, tamanho, etc., pois a evolução pode alterar isso. Por exemplo, se existirem vantagens elementais ou de espécie (supers efetivo vs tipo Dragão, etc.), uma mudança de família pode mudar fraquezas e resistências. Precisamos assegurar que o módulo responsável (talvez um DamageCalculator) consulte o **tipo atual** do monstro no momento do cálculo. Isso já deve acontecer, mas reforçamos: após evoluir, as propriedades de tipo/família devem ser atualizadas para que dano e efeitos sejam corretamente aplicados.
* **Status e Combate Modularizados:** O texto cita que o sistema de combate e status já é modular. Provavelmente há módulos como Moves (como vimos no código), Races ou Monster classes. A refatoração da evolução deve **se integrar sem invadir** esses módulos. Por exemplo, ao calcular dano de um ataque, o combate consulta o ataque no módulo Moves e consulta os stats do agressor/defensor. Esses stats agora vêm potencialmente influenciados pelos EV/IV e forma evoluída – mas se centralizamos a lógica de stat em um lugar (talvez no objeto monstro ou em um Stats module), o combate continuará chamando da mesma forma, apenas obterá valores atualizados. Assim, do ponto de vista do combate, nada muda exceto os números (que agora podem ser maiores ou com escalas diferentes). Testes devem ser feitos para calibrar que os valores não saiam da curva esperada de balanceamento – caso saiam, podemos ajustar as fórmulas/multiplicadores nos dados de evolução sem mexer no combate.
* **Sinergia com Sistema de Movimentos:** Quando o monstro evolui e herda movimentos, isso deve refletir imediatamente na lista de ações disponíveis no combate. Se usamos um sistema de **Slots de habilidades** (por exemplo, um monstro pode ter 4 ataques equipados de um conjunto maior aprendido, como Pokémon), precisamos verificar se a evolução altera o número de slots ou reseta a lista. Geralmente não faria sentido resetar – manteríamos os mesmos ataques selecionados, apenas agora possivelmente podendo trocar ou aprender novos. O importante é que, se o monstro ganhar um ataque automaticamente ao evoluir (como mencionado, certos ataques inatos de forma evoluída), ele deve ser **adicionado** à lista de moves do monstro. Podemos notificar o jogador disso via interface (“Aprendeu ataque X ao evoluir”). Novamente, qualquer parte do sistema de movimento que dependia do nome ou categoria do monstro deve ser atualizada. Pelo que vimos no código, há um Moves:GetMovesForRace(race)[[8]](file://file_00000000d82061f58e24d2bf2f45c240#:~:text=EvolutionDefs%3Drequire%28game.ReplicatedStorage.Modules.EvolutionDefs%29local%20moveNames%3DEvolutionDefs%3AGetMovesForRace%28race%2C%7B%7D%29if%20moveNames%20and%20,then%20local%20moveData%3DMoves%3AGetMove%28race%2CmoveName%29if) – possivelmente usado para determinar movimentos iniciais ou aprendíveis. Certamente, após evoluir, chamar essa função com a nova raça retornará os movimentos corretos. O design deve garantir que o monstro não perca moves já aprendidos: isso pode implicar em armazenar moves aprendidos no próprio objeto do monstro, ao invés de derivar apenas da raça. Se ainda não for assim, a refatoração do sistema de evolução pode englobar armazenar no monstro uma lista dinâmica de ataques aprendidos.

Em suma, a integração se resume a **comunicação e consistência**: O módulo de evolução informará os demais sistemas das mudanças (seja via eventos, seja chamando métodos do próprio monstro para atualizar stats e moves). Como resultado, o monstro evoluído será tratado corretamente no combate (com novos stats, possivelmente novo tipo influenciando fraquezas) e na interface (novo nome/visual, mesma posição na party etc.). Testes unitários e integrados serão necessários para garantir que, por exemplo, evoluir um monstro com pouca vida e alto ataque não cause algum overflow, ou que evoluir durante um efeito X não cause bugs. Mas esses são detalhes de implementação; conceitualmente, o sistema modularizado torna mais fácil realizar essas mudanças isoladamente.

## Conclusão

A refatoração proposta fornece um **sistema de evolução centralizado, flexível e rico**, atendendo às diretrizes desejadas. Organizamos a lógica em torno de um módulo único, inspirado nas melhores características de jogos clássicos, porém adaptado às necessidades do nosso jogo. Os benefícios esperados desta arquitetura incluem:

* **Manutenibilidade:** Com todos os dados evolutivos em um lugar e funções claras para evolução, será mais simples adicionar novas espécies, ajustar balanceamentos ou corrigir bugs sem efeitos colaterais inesperados.
* **Profundidade de Jogo:** Os jogadores experimentarão uma mecânica de evolução mais envolvente, com múltiplos caminhos, requisitos variados e resultados que dependem de suas escolhas de treino (evocando o planejamento de builds de Path of Exile). Isso aumenta o engajamento e o fator replay – diferentes escolhas produzem criaturas distintas.
* **Familiaridade e Novidade:** Ao mesmo tempo que aproveita conceitos familiares (EV/IV, árvore de evolução, chance aleatória), o sistema combina esses elementos de forma inédita, dando uma identidade própria ao jogo. Por exemplo, teremos a *estratégia* de Pokémon EVs[[1]](https://puo.fandom.com/pt-br/wiki/Sistema_IV_EV_Nature#:~:text=,at%C3%A9%2031%20para%20cada%20status), a *surpresa e complexidade* das digievoluções de Digimon[[9]](https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412#:~:text=To%20get%20the%20evolution%20you,out%20a%20simple%20concrete%20example), a *liberdade de build* de Path of Exile[[2]](https://www.pathofexile.com/passive-skill-tree#:~:text=It%20is%20a%20vast%20web,allocate%20a%20skill%20and) e a *variabilidade orgânica* de Monster Rancher – tudo coexistindo harmoniosamente.
* **Integração Suave:** A transição para o novo sistema pode ser feita garantindo compatibilidade com o combate e outras partes já prontas, como detalhado, de forma que a mudança seja transparente exceto pelas melhorias de funcionalidade.

Em conclusão, centralizar e refatorar o sistema de evolução seguindo esta proposta tornará o código mais **modular, escalável e alinhado** à visão de design do jogo. Os monstros não serão meros acumuladores de XP para evoluir no nível certo – eles serão criaturas com trajetórias únicas, moldadas tanto por sua genética quanto pelas ações do jogador, garantindo uma experiência de jogo profunda e recompensadora.

**Referências Utilizadas:** Jogos e sistemas de inspiração analisados, incluindo mecânicas de Pokémon (EV/IV e aprendizado de golpes)[[1]](https://puo.fandom.com/pt-br/wiki/Sistema_IV_EV_Nature#:~:text=,at%C3%A9%2031%20para%20cada%20status), Digimon (requisitos múltiplos e não linearidade de evoluções)[[3]](https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412#:~:text=complicated,won%2C%20and%20previous%20evolutionary%20forms)[[9]](https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412#:~:text=To%20get%20the%20evolution%20you,out%20a%20simple%20concrete%20example)[[4]](https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412#:~:text=While%20not%20immediately%20apparent%2C%20there,meeting%20five%20of%20Devimon%E2%80%99s%20quotas), Path of Exile (sistema de árvore de habilidades e builds variados)[[2]](https://www.pathofexile.com/passive-skill-tree#:~:text=It%20is%20a%20vast%20web,allocate%20a%20skill%20and) e Monster Rancher 2 (elementos de aleatoriedade e desenvolvimento orgânico de stats). Essas referências guiaram as decisões de design para equilibrar familiaridade e inovação no nosso sistema.

[[1]](https://puo.fandom.com/pt-br/wiki/Sistema_IV_EV_Nature#:~:text=,at%C3%A9%2031%20para%20cada%20status) [[5]](https://puo.fandom.com/pt-br/wiki/Sistema_IV_EV_Nature#:~:text=,lo) Sistema IV EV Nature | Wiki PUO | Fandom

<https://puo.fandom.com/pt-br/wiki/Sistema_IV_EV_Nature>

[[2]](https://www.pathofexile.com/passive-skill-tree#:~:text=It%20is%20a%20vast%20web,allocate%20a%20skill%20and) Passive Skill Tree - Path of Exile

<https://www.pathofexile.com/passive-skill-tree>

[[3]](https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412#:~:text=complicated,won%2C%20and%20previous%20evolutionary%20forms) [[4]](https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412#:~:text=While%20not%20immediately%20apparent%2C%20there,meeting%20five%20of%20Devimon%E2%80%99s%20quotas) [[7]](https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412#:~:text=HP%2C%20MP%2C%20attack%2C%20defense%2C%20speed%2C,won%2C%20and%20previous%20evolutionary%20forms) [[9]](https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412#:~:text=To%20get%20the%20evolution%20you,out%20a%20simple%20concrete%20example) Digimon Breeding Is So Complicated, I Had to Make Spreadsheets - Kotaku

<https://kotaku.com/digimon-breeding-is-so-complicated-i-had-to-make-sprea-1770411412>

[[6]](file://file_00000000d82061f58e24d2bf2f45c240#:~:text=totalGates%3DtotalGates%20%2B%201%20end%20end,minWins%29progress%3Dprogress) [[8]](file://file_00000000d82061f58e24d2bf2f45c240#:~:text=EvolutionDefs%3Drequire%28game.ReplicatedStorage.Modules.EvolutionDefs%29local%20moveNames%3DEvolutionDefs%3AGetMovesForRace%28race%2C%7B%7D%29if%20moveNames%20and%20,then%20local%20moveData%3DMoves%3AGetMove%28race%2CmoveName%29if) MonsterGame.lua

<file://file_00000000d82061f58e24d2bf2f45c240>