# Relatório Técnico – Refatoração do Sistema de Combate (Modularização de Movimentos)

## Justificativa da Refatoração

Atualmente, o sistema de **movimentos de combate** (módulo Moves) concentra todas as definições de ataques em um único arquivo monolítico. Cada movimento é definido como uma entrada numa grande tabela MOVES, contendo seus atributos (nome, poder, alcance, etc.) e tags de efeitos, conforme exemplificado pelo movimento **BasicAttack** definido no código existente[[1]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=w,power%3D30%2Ccooldown%3D4.0%2Cmin). Essa abordagem centralizada dificulta a manutenção e evolução do código por vários motivos:

* **Dificuldade de Localização e Manutenção:** Com todos os movimentos no mesmo módulo, o arquivo tornou-se extenso e complexo (centenas de linhas de definições contínuas). Encontrar um movimento específico ou ajustar seus parâmetros exige vasculhar o arquivo inteiro. Isso aumenta o risco de erros ao editar ou adicionar movimentos.
* **Escalabilidade Limitada:** À medida que novos movimentos são adicionados, a tabela central cresce indefinidamente. Alterações simples, como ajustar um atributo ou adicionar um efeito especial para um movimento, podem impactar a estrutura geral. Por exemplo, alguns movimentos exigiram *overrides* específicos para accuracy e custo de MP, implementados por meio de uma tabela extra e lógica dedicada[[2]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=%3A,nil%20then%20m.mpCost%3Dov.mpCost%20end). Esse tipo de remendo seria desnecessário em uma arquitetura modular, pois cada movimento poderia encapsular seus próprios ajustes.
* **Baixa Modularidade:** O código atual mistura **metadados** do movimento (números e propriedades estáticas), **lógica de combate** (efeitos de status aplicados, cálculo de acerto/crítico) e referências a **efeitos visuais/sonoros**, tudo no mesmo contexto. Essa falta de separação de responsabilidades torna mais difícil reutilizar ou modificar apenas um aspecto (por exemplo, trocar a animação de um movimento sem tocar na lógica de dano).
* **Integração complicada com outros sistemas:** O sistema de entidades/monstros do jogo aparentemente já é modularizado e organizado (segundo a referência do *“sistema de entidades fornecido”*). No entanto, o módulo de movimentos não segue o mesmo padrão, criando inconsistências na estrutura do projeto. Unificar o estilo de organização facilita o entendimento global do código e futuras extensões (por exemplo, no aprendizado de movimentos, evolução de criaturas, etc., que já fazem uso do módulo Moves atual).
* **Testabilidade e Evolução:** Em um design monolítico, é difícil isolar um movimento para testes unitários ou para experimentar alterações. Qualquer mudança exige recarregar todo o módulo Moves. Com módulos individuais, podemos carregar e testar um único movimento independentemente, aumentando a confiabilidade. Além disso, fica mais fácil introduzir funcionalidades novas (como tipos de efeitos não previstos originalmente, movimentos exclusivos de um chefe, etc.) implementando-as em um módulo de movimento específico sem arriscar efeitos colaterais em outros.

Diante desses pontos, justificamos a **refatoração** para um modelo totalmente modular de movimentos. A meta é melhorar a **organização**, **clareza** e **extensibilidade** do sistema de combate, alinhando-o às boas práticas de código modular e seguindo a mesma filosofia de organização do restante do projeto.

## Objetivos da Nova Arquitetura

1. **Modularização Completa dos Movimentos:** Cada movimento será isolado em seu próprio módulo, encapsulando seus dados e comportamentos únicos. Isso inclui:
2. **Metadados individuais:** cada movimento define localmente atributos como nome, poder, tempo de recarga (*cooldown*), alcance mínimo/máximo, tempo de conjuração (*castTime*), custo de mana, etc.
3. **Lógica de combate específica:** se o movimento causar efeitos únicos (dano especial, vários acertos, buffs/debuffs personalizados, teletransporte, etc.), sua implementação ficará dentro do módulo do movimento, em vez de espalhada em condicionais no código central.
4. **Efeitos visuais/sonoros próprios:** referência a partículas, animações ou sons específicos que o movimento deve executar. Cada módulo poderá indicar quais efeitos visuais (VFX) e sonoros (SFX) usar em suas fases (início da conjuração, projétil em movimento, impacto, fim da conjuração), mantendo esses detalhes de apresentação separados da lógica de jogo.
5. **Módulo Principal Moves como Mapeador:** Haverá um módulo principal central (um “mapper” de movimentos) responsável por:
6. Importar todos os módulos individuais de movimentos (por exemplo, MovesModules/BasicAttack.lua, MovesModules/VoidRay.lua, etc.) e registrá-los em uma coleção (por exemplo, uma tabela MOVES interna).
7. Fornecer funções utilitárias **genéricas**, como cálculo de chance de acerto (*accuracy*), chance de crítico, custo de mana (MP) baseados nos atributos do movimento. Essas funções, antes implementadas diretamente no módulo monolítico, permanecerão no módulo principal para uso geral.
8. Implementar lógica de apoio, como fornecer um movimento *fallback* (por exemplo, BasicAttack) caso um movimento específico não seja encontrado, e métodos para seleção aleatória de movimentos levando em conta raridade e categoria (família/raça) – funcionalidades que já existem no código atual e serão preservadas.
9. Expor uma interface pública semelhante à atual (métodos GetMove, GetPoolForRace, GetRandomMoves, etc.), de forma que outras partes do jogo continuem interagindo com Moves de forma consistente, sem precisar conhecer os detalhes de modularização interna.
10. **Organização de Pastas e Consistência:** Estruturar os arquivos em pastas de forma clara e consistente, inspirada no *design* já utilizado para entidades. A ideia é separar conceitos por diretórios:
11. Uma pasta (ou coleção) exclusiva para os módulos de movimentos (por exemplo, MovesModules/), contendo um arquivo por movimento.
12. O módulo principal Moves.lua separado, possivelmente na raiz dos módulos de jogo, atuando como integrador.
13. Uma estrutura dedicada para **efeitos visuais** dos movimentos (por exemplo, pasta MoveVFX/ ou um módulo MoveVFX.lua), para centralizar definições de VFX/SFX usados pelos movimentos. Isso segue o padrão de separar lógica de jogo dos detalhes de apresentação. No sistema atual, esses efeitos já são tratados em um módulo separado (MoveVFX), acessado durante a execução do movimento – iremos reforçar e organizar esse aspecto na nova estrutura.
14. **Exemplo Concreto e Consistente:** Demonstrar com um exemplo completo (usaremos o movimento básico BasicAttack) como ficará um módulo individual refatorado, incluindo sua definição de dados e quaisquer comportamentos especiais.
15. **Manutenção da Funcionalidade Existente:** Garantir que o comportamento atual do jogo não seja alterado pela refatoração. A refatoração visa melhorar organização interna sem mudar regras de jogo. Ou seja, cálculos de dano, chances de acerto/crítico, e seleção aleatória de movimentos continuarão funcionando igual – apenas distribuídos em módulos diferentes.
16. **Facilidade de Extensão Futura:** Preparar o terreno para futuras melhorias, como adição de tipagem estática, uso de metatables para padrões de projeto, criação de testes automatizados de movimentos e eventuais adaptações para ambientes *multiplayer* complexos. Essas recomendações futuras serão discutidas ao final.

## Estrutura de Pastas Proposta

Após a refatoração, o sistema de movimentos será organizado em uma hierarquia de arquivos mais granular. Abaixo está a estrutura de diretórios e módulos sugerida:

Modules/ -- Pasta principal de módulos (já existente no projeto)  
├── Moves.lua -- Módulo principal agregador de movimentos (mapper)  
├── MovesModules/ -- Pasta contendo módulos individuais de cada movimento  
│ ├── BasicAttack.lua  
│ ├── VoidRay.lua  
│ ├── CosmicThread.lua  
│ ├── ... (demais movimentos, um por arquivo)  
│ └── [NovoMovimento].lua (futuras adições)  
└── MoveVFX/ -- Pasta (ou módulo) para efeitos visuais e sonoros dos movimentos  
 ├── MoveVFX.lua -- Módulo principal de VFX (resolver/gerenciar pacotes visuais)  
 ├── BasicAttackVFX.lua -- (Opcional) módulo com definição VFX do BasicAttack  
 ├── VoidRayVFX.lua -- (Opcional) módulo com definição VFX do VoidRay  
 └── ... -- etc., ou estrutura alternativa conforme necessidade

**Detalhes dessa organização:**

* **Modules/Moves.lua:** Este continuará sendo o ponto de acesso central para consultas de movimentos. Ele atuará principalmente como um *conteiner* que carrega os módulos de MovesModules e mantém uma tabela interna MOVES com todos os movimentos disponíveis. Além disso, reterá as funções utilitárias (cálculos de accuracy, crítico, custo de MP, randomização, etc.) e as constantes default (por exemplo, DEFAULTS.baseAccuracy, limites de mínimo/máximo) que antes estavam no topo do módulo original[[3]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=local%20Moves%3D,min%28b%2Cx%29%29end%20local%20function)[[4]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=computeAccuracy%28m%29local%20acc%3DDEFAULTS,45%20then%20crit). Dessa forma, lógica comum permanece num lugar único. O módulo principal *não* conterá mais listas extensas de movimentos – estas ficarão nos arquivos individuais.
* **Modules/MovesModules/:** Pasta que abriga os **módulos individuais de movimentos**. Cada movimento do jogo terá seu próprio arquivo (por exemplo, BasicAttack.lua, VoidRay.lua, etc.), onde define:
* Seus **metadados**: ID (identificador único, usado internamente e para vincular com VFX e referências), nome legível, raridade, poder base, tempo de recarga, alcance mínimo/máximo, tempo de conjuração, flags como lockMovement (se o movimento imobiliza o usuário ao conjurar), categorias/tags (e.g. "Melee", "Magic", "AoE", etc.), e lista de efeitos de status que ele provoca (e.g. {"Stun", "Slow"} para um movimento que atordoa e lentifica o alvo). Todos esses campos antes estavam no grande table MOVES[[5]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=VoidRay%3D%7Bid%3D,Debuff)[[6]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=k,NanoSwa); agora ficam encapsulados no retorno de cada módulo.
* **Lógica de combate única (quando aplicável):** se o movimento requer um comportamento especial que não seja apenas “causar X de dano e aplicar status Y” – por exemplo, um movimento que desloca o usuário (reposição), ou que atinge multiplos alvos, ou cujo dano depende de alguma condição – essa lógica pode ser implementada dentro do próprio módulo. Podemos definir, por convenção, uma função como Execute(attacker, target) ou ApplyEffects(attacker, target) dentro do módulo do movimento para tratar desses efeitos únicos. No caso de movimentos simples (como ataques básicos), essa função pode nem ser necessária (podemos usar a lógica genérica de dano do sistema de combate). Mas ter a capacidade de incluir funções permite flexibilidade: o módulo VoidStep poderia ter, por exemplo, uma função que teleporta o usuário atrás do alvo e lhe concede invulnerabilidade temporária (conforme suas tags Reposition e Invulnerability), enquanto o módulo PhotonBarrage (que tem tag MultiHit) poderia definir internamente que dispara múltiplos projéteis em sequência. A ideia é que **cada módulo saiba fazer o que for exclusivo daquele movimento**, evitando “ifs” espalhados no código central.
* **Referências a efeitos visuais e sonoros:** cada movimento pode indicar qual pacote de VFX/SFX ele utiliza. Aqui temos duas abordagens possíveis:
  1. **Via Configuração Externa (MoveVFX):** O movimento apenas especifica um identificador visual, ou usa seu próprio id como chave, e o módulo central de VFX (MoveVFX) contém um mapeamento dessa chave para os assets visuais correspondentes (partículas, animações, sons). Esse é o modelo já existente: por exemplo, quando um movimento é executado, o sistema de combate chama Bridge.PlayImpact(attacker, target, move.id), e internamente o Bridge usa MoveVFX.Resolve(moveId) para obter os efeitos visuais daquele movimento. Neste caso, o módulo do movimento não precisa ter detalhes de VFX – ele apenas tem um id consistente com a entrada no MoveVFX.
  2. **Via Associação no Próprio Módulo:** Alternativamente, poderíamos incluir no retorno do módulo um campo (por exemplo, vfx) que contenha as definições visuais para aquele movimento (ou referência a elas). Por exemplo, BasicAttack.lua poderia retornar uma tabela com seus atributos e um subtabela vfx = { castEffect = "BasicCast", impactEffect = "BasicImpact", sfxHit = "HitSound", ... }. Essa informação visual seria então consumida pelo sistema de renderização (cliente) ou pelo módulo MoveVFX principal. Essa abordagem favorece uma coesão maior (os dados de um movimento, inclusive visual, ficam juntos), mas pode não ser ideal se quisermos manter uma separação clara entre lógica (server) e apresentação (client). Dado que **no Roblox geralmente mantemos os assets visuais separados do código de lógica**, a primeira abordagem (mapa central de VFX) é mais adequada – e podemos apenas continuar usando-a, organizando melhor seus arquivos.
* **Modules/MoveVFX/:** A existência dessa pasta sugere separar em nível de arquivo os pacotes de efeitos visuais dos movimentos. Por exemplo, podemos ter um módulo MoveVFX.lua central que exporta a função Resolve(moveId) (semelhante ao atual) e internamente sabe quais efeitos aplicar em cada fase (*cast*, *travel*, *impact*, *endcast*) de cada movimento. A diferença é que, para modularidade, podemos deslocar as configurações de cada movimento para sub-módulos. Exemplo: um arquivo BasicAttackVFX.lua poderia retornar uma tabela definindo os efeitos de BasicAttack (por exemplo: partícula de ataque corpo-a-corp simples, sem efeito de travel, som de espadada no impacto, etc.). O módulo principal MoveVFX.lua então faria require desses submódulos e montaria a estrutura de dados (um pouco análogo ao que faremos no Moves.lua com os movimentos). Contudo, essa subdivisão de VFX é opcional – poderíamos também manter o MoveVFX.lua como um único módulo com todas as definições (caso o volume não seja tão grande). A refatoração foca mais na lógica de movimentos; o importante é garantir que os **efeitos visuais permaneçam desacoplados** da lógica de combate. Eles residirão sob a pasta MoveVFX e poderão ser mantidos pelo time de arte/UX sem interferir nos módulos de lógica.

*Nota:* No design atual, o módulo MoveVFX original já provê valores padrões e construção de pacotes de efeito para cada movimento, preenchendo valores faltantes automaticamente (ex.: usar efeitos genéricos de *cast* e *impact* caso não definidos)[[7]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=stage.effect%3D%3Dnil%20and%20stage._stage%3D%3D,local)[[8]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=endcast,Resolve%28moveId%29return%20buildResolved%28moveId%29end%20return%20M). Na refatoração, manteremos essa funcionalidade – seja no mesmo módulo central de VFX ou distribuída – para evitar duplicação de informações. Ou seja, se um movimento não define explicitamente algo visual, usaremos defaults (padrões globais) assim como antes.

Em suma, a estrutura proposta cria uma correspondência **1:1 entre movimento e módulo**, alinhada com a filosofia de design modular do projeto (conforme visto no sistema de entidades). Isso melhora a **coerência**: por exemplo, se existe uma pasta Races/ ou Monsters/ onde cada criatura é definida isoladamente, agora teremos MovesModules/ onde cada golpe/habilidade também é definido isoladamente. A centralização permanece através dos módulos Moves.lua e MoveVFX.lua que atuam como **facades** (fachadas) para o resto do código usar sem se preocupar com os detalhes internos.

## Módulos de Movimentos Individuais (MovesModules)

Cada arquivo em MovesModules/ define integralmente um movimento. Vamos detalhar o conteúdo típico de um módulo de movimento, usando o exemplo do **BasicAttack** (Ataque Básico), que é o movimento padrão e mais simples do jogo:

* **Metadados e Atributos Básicos:** Logo no início do módulo, definimos uma tabela Lua com todos os campos descritivos do movimento. Por exemplo:
* id: Identificador único (string). *Exemplo:* "BasicAttack".
* name: Nome do movimento (pode ser usado para exibição em interface). *Ex:* "Ataque Básico".
* rarity: Raridade do movimento (Amateur, Advanced, Specialist, Ascended, Primordial, etc.). *Ex:* "Amateur" (Comum) para Ataque Básico.
* power: Poder base (dano base ou fator de efeito). *Ex:* 10 para Ataque Básico (conforme código atual[[9]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=any%7D%3D%7BBasicAttack%3D%7Bid%3D,power%3D30%2Ccooldown%3D4.0%2Cmin)).
* cooldown: Recarga em segundos. *Ex:* 1.2 segundos.
* minRange e maxRange: Alcance mínimo/máximo em studs/unidades de jogo. *Ex:* 0 a 6 (Ataque Básico pode ser usado corpo a corpo até 6 unidades de distância).
* castTime: Tempo de conjuração/lançamento. *Ex:* 0.4 seg.
* castSpeed: (Opcional) Fator de velocidade da animação de conjuração. *Ex:* 1.0 (padrão).
* lockMovement: (Opcional) Booleano indicando se o usuário fica parado enquanto conjura. *Ex:* true para Ataque Básico (o atacante não se move durante o golpe).
* tags: Lista de tags/categorias do movimento. *Ex:* { "Melee", "Universal" } – Ataque Básico é corpo-a-corpo e Universal (disponível para qualquer raça/família).
* effects: Lista de efeitos de status que o movimento aplica. *Ex:* { } vazio para Ataque Básico (não tem efeito extra além do dano). Em movimentos mais avançados, aqui estariam strings como "Stun", "Slow", "Burn", etc., indicando quais *StatusEffects* serão aplicados no alvo ao acertar (o sistema central de combate mapeará essas strings para efeitos reais[[10]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=CombatManager,LocationParts%3D%7Btarget.PrimaryPart%20or%20ta)).
* **Dados Derivados (calculados ou customizados):** Alguns campos, como accuracy (precisão) e critChance (chance de crítico), podem ser definidos automaticamente pelo sistema com base nos atributos acima e tags. No módulo atual, isso é feito pós-definição via funções computeAccuracy e computeCrit[[4]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=computeAccuracy%28m%29local%20acc%3DDEFAULTS,45%20then%20crit)[[11]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=clamp%28acc%2CDEFAULTS,maxCrit%29end%20local%20function). Na nova arquitetura, podemos seguir duas abordagens:
* **Cálculo Centralizado:** Não definir accuracy e critChance nos módulos individuais, deixando-os nil, e permitir que o módulo principal Moves os calcule após carregar todos os movimentos (usando as mesmas fórmulas atuais). Esse método garante consistência e centraliza a lógica de cálculo. Por exemplo, o módulo Moves ao importar BasicAttack, detectaria que BasicAttack.accuracy não foi definido e aplicaria computeAccuracy para obter ~0.96 de precisão (levando em conta que é Melee e castTime 0.4)[[2]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=%3A,nil%20then%20m.mpCost%3Dov.mpCost%20end).
* **Definição Explícita no Módulo:** Fornecer já no módulo valores como accuracy = 0.96 e mpCost = 0 (custo de mana) para casos especiais, seguindo a tabela de overrides antiga[[2]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=%3A,nil%20then%20m.mpCost%3Dov.mpCost%20end). Isso elimina a necessidade de uma tabela de override separada. Podemos calcular manualmente esses valores ou extrair do resultado do sistema atual. Em geral, para evitar duplicação de lógica, preferimos a abordagem anterior (cálculo centralizado), mas não há impedimento em definir manualmente valores garantidos (no caso do Ataque Básico, sabemos que sempre será sem custo de mana e muito preciso).
* **Conclusão:** A solução híbrida seria: deixar o cálculo default no central para a maioria, mas se um módulo quiser sobrescrever (como era o caso de alguns movimentos raros), ele pode fornecer o valor e o normalizador do Moves principal respeitará o que já estiver preenchido. De fato, no código atual já havia essa lógica: só calculava se o campo fosse nil[[12]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=ipairs%28m.tags%29do%20mult%20,B). Vamos manter o mesmo princípio. Portanto, em BasicAttack podemos explicitamente definir mpCost = 0 e accuracy = 0.96 para conservar seu status especial de “golpe gratuito e confiável”, ou deixar para o cálculo automático – ambos resultarão no mesmo valor final dado o tratamento de override.
* **Funções Específicas do Movimento (Opcional):** Se o movimento requer lógica de execução própria, definimos funções no módulo. Exemplos de possíveis funções:
* BasicAttack.execute(attacker, target): Poderia simplesmente invocar o mecanismo genérico de dano do jogo (já presente no CombatManager) ou aplicar um modificador simples. Provavelmente o Ataque Básico não precisa de rotina custom – ele apenas causa dano normal. Por isso, podemos omitir essa função no BasicAttack; entretanto, vale a pena citar para movimentos futuros.
* VoidRay.execute(attacker, target): Poderia ser definida para causar dano penetrante que ignora defesa (**exemplo hipotético** para ilustrar). Nesse caso, a função custom faria cálculo de dano direto e aplicaria o efeito “Debuff” no alvo.
* QuantumSlip.execute(attacker, target): Esse movimento tem efeitos de *Invulnerability* e *Reposition*; uma função custom aqui poderia manipular o estado do atacante (torná-lo invulnerável por alguns segundos, talvez usando o sistema de StatusEffects com um efeito personalizado) e deslocá-lo instantaneamente a outra posição (por exemplo, trocar de lugar com o alvo ou teletransportar alguns studs adiante).

O importante é que **o módulo principal de combate checará se o movimento possui alguma função de execução customizada e, em caso positivo, chamará essa lógica**, senão aplicará apenas a lógica genérica. No design atual, grande parte dos efeitos é tratada genericamente no CombatManager.ApplyMoveEffects com base nas tags em move.effects[[10]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=CombatManager,LocationParts%3D%7Btarget.PrimaryPart%20or%20ta). Podemos estender isso: manter esse tratamento para efeitos padrão (buffs, debuffs simples), mas dar aos movimentos individuais a chance de executar código adicional antes ou depois. Isso torna o sistema mais flexível para movimentos futuros sem inflar o código central.

* **Retorno do Módulo:** No final, o módulo retorna a tabela do movimento (com todos os campos definidos e funções). Por convenção, podemos retornar apenas a tabela (como um objeto estático), ou retornar uma função construtora se imaginássemos instâncias do movimento – porém, neste caso, não há necessidade de instanciar movimentos, já que eles são conceitos estáticos. Mantemos simples: return BasicAttack (supondo a tabela se chama BasicAttack internamente).

Em resumo, cada arquivo em MovesModules atua como a **fonte da verdade** para aquele movimento específico, tornando fácil localizar e editar qualquer aspecto desse movimento sem impactar outros.

### Exemplo: Módulo BasicAttack.lua Refatorado

A seguir, apresentamos um exemplo completo ilustrativo de como ficaria o módulo do movimento **BasicAttack** após a refatoração:

-- File: Modules/MovesModules/BasicAttack.lua  
  
-- Definição do movimento Ataque Básico em uma tabela  
local BasicAttack = {  
 id = "BasicAttack",  
 name = "Ataque Básico",  
 rarity = "Amateur",  
 power = 10,  
 cooldown = 1.2,  
 minRange = 0,  
 maxRange = 6,  
 castTime = 0.4,  
 castSpeed = 1.0,  
 lockMovement = true,  
 tags = { "Melee", "Universal" },  
 effects = { }, -- Sem efeitos de status adicionais (apenas dano puro)  
  
 accuracy = 0.96, -- Precisão alta (override do cálculo padrão para este movimento)  
 mpCost = 0, -- Custo de mana nulo (movimento básico não consome MP)  
}  
  
-- (Opcional) Função de execução customizada.  
-- Ataque Básico não tem efeitos especiais além do dano normal,   
-- portanto não precisamos alterar a lógica padrão de dano.  
-- Poderíamos, se desejado, ainda definir um stub para fins de demonstração:  
-- function BasicAttack.execute(attacker, target)  
-- -- Dano e acerto são resolvidos pelo sistema genérico, então nada especial aqui.  
-- end  
  
return BasicAttack

**Explicação do exemplo:** Observamos que quase todos os campos presentes originalmente na tabela MOVES["BasicAttack"] do módulo monolítico aparecem aqui de forma declarativa. Incluímos explicitamente accuracy e mpCost para cobrir o override que antes era aplicado separadamente[[2]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=%3A,nil%20then%20m.mpCost%3Dov.mpCost%20end) – assim o módulo principal não precisará corrigir esses valores posteriormente. A função execute comentada indica onde colocaríamos lógica própria se necessário; no caso do Ataque Básico, optou-se por usar a implementação genérica de combate (dano calculado pelo sistema de combate, sem efeitos adicionais). Manter o stub (mesmo que vazio) pode ser útil por padronização, mas não é obrigatório.

Um módulo de movimento mais complexo, digamos VoidRay.lua, seguiria estrutura semelhante: dados de definição (ex.: poder 30, alcance 6-18, castTime 0.9, tags {"Magic","Ranged"} etc conforme[[13]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=xRange%3D14%2CcastTime%3D0.75%2Ceffects%3D%7B,po)) e poderia ter uma função execute que, por exemplo, aplica dano perfurante em linha reta (*lore* do Void Ray) além de marcar o alvo com um *debuff* (conforme effects={"Debuff"} já indicado nos metadados). A beleza do novo sistema é que toda essa lógica do VoidRay ficaria confinada ao seu módulo, e outros movimentos não precisariam se preocupar com “se for VoidRay, faça X” – reduzindo condicionais no código central.

## Módulo Principal Moves (Mapper) – Ajustes na Implementação

Com os movimentos definidos individualmente, o módulo principal Moves.lua precisará ser adaptado para construir sua base de dados a partir desses módulos. Suas responsabilidades principais serão:

1. **Carregar os Módulos de Movimento:** Quando Moves.lua for executado (via require em alguma parte do jogo, provavelmente no início), ele deve **importar todos os movimentos disponíveis** na pasta MovesModules/. Há duas formas de fazer isso:
2. Listar manualmente cada movimento. Exemplo:

* local MOVES = {}  
  MOVES.BasicAttack = require(game.ReplicatedStorage.Modules.MovesModules.BasicAttack)  
  MOVES.VoidRay = require(game.ReplicatedStorage.Modules.MovesModules.VoidRay)  
  MOVES.CosmicThread = require(game.ReplicatedStorage.Modules.MovesModules.CosmicThread)  
  -- ... e assim por diante para todos os movimentos.
* Isso é claro e explícito, mas exige atualizar esta lista toda vez que um novo movimento for adicionado.

1. Carregamento dinâmico iterando os filhos da pasta. Exemplo em Roblox Lua:

* local MOVES = {}  
  local movesFolder = game.ReplicatedStorage.Modules:WaitForChild("MovesModules")  
  for \_, moduleScript in ipairs(movesFolder:GetChildren()) do  
   if moduleScript:IsA("ModuleScript") then  
   local moveData = require(moduleScript)  
   if moveData.id then  
   MOVES[moveData.id] = moveData  
   end  
   end  
  end
* Assim, qualquer novo ModuleScript colocado dentro de MovesModules será automaticamente registrado, sem precisar editar o código do mapper. Essa abordagem é escalável e diminui a chance de esquecer de adicionar um movimento novo ao mapa.

1. Podemos optar por uma mistura: carregar dinamicamente mas também pode haver certos movimentos base que queremos garantir ordem (ex.: talvez garantir BasicAttack carregado primeiro caso outros dependam, embora isso não deva acontecer já que não há dependência direta entre movimentos).

Em ambos os casos, ao final desse passo teremos uma tabela interna MOVES contendo entradas do tipo MOVES["BasicAttack"] = {...dados do BasicAttack...}, etc., para todos os movimentos.

1. **Normalizar/Completar Dados dos Movimentos:** Após carregar, o módulo principal deve iterar por MOVES e aplicar quaisquer cálculos ou preenchimentos necessários:
2. **Accuracy, Crit, mpCost:** Conforme discutido, para cada movimento em MOVES deveremos garantir que existam valores para precisão (accuracy), chance de crítico (critChance) e custo de mana (mpCost). Usaremos as funções utilitárias baseadas nas já existentes:
   * computeAccuracy(move) – calcula precisão baseando-se em DEFAULTS.baseAccuracy ajustado por penalidades de cast time e alcance, e bônus das tags (ex.: +0.04 se tag Melee, -0.03 se Ranged, etc.)[[4]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=computeAccuracy%28m%29local%20acc%3DDEFAULTS,45%20then%20crit)[[14]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=m,tags%20then%20for%20_%2Ct%20in). Já existem tabelas de ajuste como tagAccuracyAdj no código[[15]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=Crit%3D0.35%2CrarityMpMult%3D%7BAmateur%3D1.00%2CAdvanced%3D1.10%2CSpecialist%3D1.20%2CAscended%3D1.35%20%2CPrimordial%3D1.50%7D%2CtagAccuracyAdj%3D%7BMelee%3D0.04%2CRanged%3D,castAccuracyPenaltyPerSec%3D0.06%2CrangeAccuracyPenaltyPer10%3D0.015), que serão mantidas. O Moves principal aplicará move.accuracy = clamp(calculado, minAccuracy, maxAccuracy) se move.accuracy não estiver definido pelo módulo.
   * computeCrit(move) – calcula chance de crítico base (DEFAULTS.baseCrit) com ajustes por tags (ex.: +0.01 se Melee, etc. conforme tagCritAdj)[[16]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=clamp%28acc%2CDEFAULTS,45%20then%20crit) e pequenos incrementos por poder alto (no código original, +0.02 se poder>=35, +0.01 extra se >=45)[[17]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=computeCrit%28m%29local%20crit%3DDEFAULTS,maxCrit%29end%20local%20function). Novamente, preencherá move.critChance se não definido.
   * computeMpCost(move) – calcula custo de mana aproximado usando fórmula base (mpCostBase + power\*mpPerPower) ajustada por multiplicador de raridade e tags (ex.: movimentos mágicos custam mais, movimentos Melee menos)[[18]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=computeMpCost%28m%29local%20power%3Dmath,B). Garante mínimo e arredonda o valor. No código original, havia lógica especial: se power==0 então base=6 (movimentos utilitários têm custo base 6)[[18]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=computeMpCost%28m%29local%20power%3Dmath,B). Manteremos essas regras. Preencherá move.mpCost se não definido.
   * O Moves principal executará essas funções em cada movimento logo após o carregamento. Assim, todo movimento sai com esses atributos derivados prontos para uso. *(No exemplo BasicAttack acima, definir manualmente foi mais por simplicidade; poderíamos removê-los e deixar o Moves calculá-los igualmente.)*
3. **Aplicar Overrides Específicos:** No design original, após normalizar, havia uma tabela OVERRIDES para corrigir alguns movimentos excepcionais[[2]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=%3A,nil%20then%20m.mpCost%3Dov.mpCost%20end) (e.g. garantir ArchitectShield.accuracy=1.0 e mpCost=14, etc.). Com o novo sistema, podemos eliminar essa estrutura separada, pois podemos definir esses valores já nos módulos respectivos. Caso optemos por não defini-los manualmente, podemos carregar uma lógica equivalente: por exemplo, após computar tudo, fazer:

* if MOVES["ArchitectShield"] then MOVES["ArchitectShield"].mpCost = 14; MOVES["ArchitectShield"].accuracy = 1.0; end
* Mas isso é basicamente replicar o override. Melhor é já ter posto no ArchitectShield.lua: accuracy=1.0, mpCost=14. Portanto, idealmente a fase de override seria desnecessária. **Recomendação:** transferir todos valores de override para os módulos individuais durante a refatoração (como feito com BasicAttack, GenesisBeam, etc.), simplificando o mapper.

1. **Outros ajustes:** Garantir que certos movimentos fundamentais estejam sempre presentes. Exemplo: no sistema original, ao construir o **pool** de movimentos por raça, eles sempre adicionam BasicAttack manualmente ao pool para garantir que todo conjunto inclua um ataque básico[[19]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=%2Ctags%3D%7B,Spec). Isso continua válido. Podemos marcar no Moves principal para ao final fazer MOVES.BasicAttack presente globalmente (o que já estará). Além disso, pode-se definir a propriedade BasicAttack.rarity = "Amateur" explicitamente (já definido) para fins de sorteio.
2. **Atualizar Funções de Acesso:** O módulo principal expõe métodos usados em diversas partes do jogo:
3. Moves:GetMove(raceOrNil, id): hoje simplesmente retorna MOVES[id][[20]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=buildPoolForFamily,then%20local%20moveData%3DMoves%3AGetMove%28race%2CmoveName%29if). Isso permanece igual, apenas internamente estará acessando a tabela preenchida via módulos.
4. Moves:GetPoolForRace(race): constrói um subconjunto de MOVES adequado à raça/família fornecida[[21]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=Races%3AGetBaseRaceFamily,return%20result%20end%20end%20local). Essa função filtra movimentos cujas tags incluem a família ou "Universal". Na implementação original, isso itera sobre MOVES e seleciona os que batem a condição, e adiciona BasicAttack no pool[[22]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=,race%29end%20return%20nil%20end%20function)[[23]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=family%20then%20local%20pool%3D,EvolutionDefs%29local). Continuaremos com essa lógica, só que MOVES agora vem dos módulos. Podemos aprimorar tornando a lista de famílias e correspondências mais explícita ou data-driven, mas isso foge do escopo – manteremos a busca direta nas tags como está.
5. Moves:GetRandomMoves(race, count, minRarity, weightsOverride): monta aleatoriamente uma lista de movimentos para, por exemplo, recompensas ou escolha de IA, respeitando pesos por raridade[[24]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=then%20return%7B%7Bmove%3DbasicAttack%2Crarity%3D,random%28%29local%20acc%3D0%20local)[[25]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=chosenRarity%3A%20string%3F%3Dnil%20for%20_%2Cr%20in,move%3Dmv%2Crarity%3Dmv.rarity%7Dend%20return%20picked%20end%20Moves.RARITY%3DRARITY). Essa função usa internamente:
   * GetPoolForRace(race) para restringir quais movimentos considerar.
   * Uma tabela de pesos padrão por raridade (DEFAULT\_RARITY\_WEIGHTS) e possivelmente overrides fornecidos.
   * Sorteia count movimentos aleatoriamente conforme distribuição. Garante que se nada for encontrado em dada raridade, cai para Amateur.
   * Sempre garante que BasicAttack possa ser escolhido se tudo falhar. Essa função permanece praticamente igual, pois não depende de detalhes de implementação do MOVES, apenas de seu conteúdo. Apenas certificaremos que as constantes e ordens de raridade (ORDERED\_RARITIES) estejam definidas no Moves principal (como no original) e usar MOVES atualizado.
6. Moves:getStarterMoves(race): determina quais movimentos iniciais uma criatura de certa raça terá ao nascer/evoluir[[20]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=buildPoolForFamily,then%20local%20moveData%3DMoves%3AGetMove%28race%2CmoveName%29if). No original, ele tenta usar Moves:GetStarterMoves (talvez definido em outro lugar ou sobescrito), se não, busca em EvolutionDefs um conjunto de nomes de movimentos e então monta a lista final. Essa parte não muda exceto que as chamadas a Moves:GetMove(race, moveName) continuarão funcionando. A refatoração pode inclusive facilitar se quisermos mover a definição de starters para um local mais declarativo (ex.: talvez cada raça define seus starters, ou usar a constante STARTERS\_BY\_FAMILY que existia[[26]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=end%20end%20local%20STARTERS_BY_FAMILY%3D%7BProgenitor%3D%7B,return%20pool%20end%20local%20function)). No entanto, isso já está implementado com sucesso, então manteremos conforme está, apenas adaptando a obtenção dos moves da nova estrutura.
7. **Mantendo a Interface via Metatable:** No código original, o módulo retorna return setmetatable({}, Moves)[[27]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=mv%3Dbucket%5Bmath.random%281%2Cmath.max%281%2C,Moves). Isso significa que quem faz local Moves = require(MovesModule) obtém uma tabela vazia com Moves como metatable, de modo que as chamadas Moves:GetMove(...) invocam na verdade Moves.\_\_index.GetMove. Essa técnica de ocultar a tabela interna e expor só métodos é interessante para evitar modificações externas nos dados. Podemos manter esse padrão: definiremos Moves.\_\_index = Moves no topo (como já tinha) e retornaremos setmetatable({}, Moves) no final. Assim, nenhum código externo acessa diretamente MOVES ou outros detalhes – usam somente as funções públicas. **Nota:** Ao implementar a carga dinâmica, precisamos ter cuidado para inicializar MovesModules antes do retorno. Provavelmente faremos todas as requires e normalizações no corpo do módulo Moves.lua, depois configurar metatable e retornar instância. Essa ordem garante que no momento que alguém requer Moves, já esteja tudo pronto.
8. **Integração com Efeitos Visuais:** O Moves principal em si talvez não interaja diretamente com MoveVFX (quem faz isso é o Bridge no combate). Porém, podemos adicionar utilidades no Moves principal para depuração ou conveniência, como Moves:PlayMoveVisual(moveId, stage, where) chamando internamente MoveVFX.Resolve(moveId). Isso foge um pouco do escopo, mas menciono para notar que, na refatoração, a conexão entre movimento e visual continua sendo via move.id. Desde que não alteremos os IDs, o módulo MoveVFX continuará funcionando. Se optarmos por modularizar MoveVFX também, o Moves principal não precisaria saber – o Bridge continuará usando MoveVFX como antes. Em suma, **nenhuma alteração é necessária** no mecanismo de VFX devido a essa refatoração, exceto realocar arquivos. Para clareza, podemos documentar dentro do Moves principal qual é a correspondência de nomes (por exemplo, um comentário listando “-- Visual effects for moves are defined in MoveVFX modules with matching keys”).

**Resumindo as mudanças no Moves.lua:** ele deixa de ter a grande tabela de movimentos e a tabela de overrides. Em vez disso, ele: - Importa todos os movimentos de arquivos externos. - Usa as mesmas funções (refatoradas como locais ou métodos) para completar os dados de cada movimento. - Mantém as estruturas de raridade, pesos e métodos *Get*/*Select* praticamente inalterados. - Retorna o objeto Moves configurado.

Depois dessa refatoração, qualquer parte do jogo que faça, por exemplo, Moves:GetMove(nil, "VoidRay") ou Moves:GetRandomMoves("CosmicWeaver") irá através do metatable encontrar a função no Moves principal, que por sua vez operará sobre a nova tabela interna preenchida com os módulos. Ou seja, para o restante do sistema, o comportamento será equivalente – não é necessário alterar chamadas externas, somente certificar que o módulo Moves está sendo requerido em algum ponto (o que já ocorria).

## Estrutura dos Efeitos Visuais (MoveVFX)

Embora a pergunta foque na modularização dos movimentos em si, foi solicitada orientação sobre **onde colocar os efeitos visuais** nessa nova estrutura. Como mencionado, o projeto já possui um módulo separado para VFX (MoveVFX), provavelmente com uma tabela de pacotes de efeitos por movimento e funções utilitárias para combiná-los (aplicar partícula, reproduzir som, etc.). Vamos alinhar a organização de VFX com a dos movimentos:

* **Manter o Módulo Central de VFX:** Continuaremos com um módulo principal MoveVFX.lua que fornece pelo menos uma função pública Resolve(moveId) – a qual retorna a configuração de efeitos para aquele movimento. Esse módulo conhece os diferentes *stages* da execução (cast, travel, impact, endcast) e qual efeito deve ocorrer em cada um. No código atual do Bridge, vemos que após obter pack e adv via MoveVFX.Resolve, ele primeiro checa se há uma função override para aquela fase (em adv), senão pega os dados em pack[stage] e executa os efeitos. Vamos preservar essa interface para não ter que refatorar também o Bridge ou outros consumidores de VFX.
* **Modularizar definindo sub-módulos (opcional):** Se o módulo de VFX estiver muito grande ou se quisermos seguir a mesma filosofia de “1 movimento = 1 arquivo”, podemos transformar MoveVFX em uma pasta:
* MoveVFX/MoveVFX.lua – Módulo principal, similar ao Moves.lua, que faz require dos submódulos.
* MoveVFX/BasicAttackVFX.lua, MoveVFX/VoidRayVFX.lua, etc. – cada qual retorna a tabela de efeitos daquele movimento, por exemplo:
* -- BasicAttackVFX.lua (retorno exemplo)  
  return {  
   cast = { effect = "BasicSwingEffect", sfxStart = "SwordWhoosh" },  
   travel = nil, -- BasicAttack não tem projétil viajante  
   impact = { effect = "BasicHitEffect", sfxHit = "Hit" },  
   endcast = nil -- sem efeito de término  
  }
* e assim por diante. O módulo principal iteraria sobre esses, similar ao Moves, para compor um mapa PACKS[moveId] = { cast=..., travel=..., impact=..., endcast=... }. Poderíamos também ter, para casos especiais, funções de efeito avançado (como no atual adv – talvez uma tabela ADVANCED com funções personalizadas para alguns movimentos específicos).
* Se a escolha for não dividir, então simplesmente ter MoveVFX.lua único é aceitável. Nesse caso, apenas garantiríamos que ele esteja na pasta Modules ou acessível da mesma forma. Talvez possamos movê-lo para Modules/MoveVFX.lua (raiz) ou mantê-lo em Modules/MoveVFX/MoveVFX.lua e ajustar o require no Bridge. O importante é não quebrar a chamada:
* local MoveVFX = require(ReplicatedStorage.Modules:WaitForChild("MoveVFX"))
* Se MoveVFX virar uma pasta, WaitForChild("MoveVFX") retornará a pasta, e o require falhará a menos que a gente aponte para um ModuleScript dentro. Logo, recomendação: **manter um ModuleScript nomeado MoveVFX** (seja na raiz ou dentro da pasta) para continuar sendo requerido diretamente. Esse ModuleScript pode internamente carregar sub-definições se existirem.
* **Localização dos arquivos de VFX:** Colocar a pasta MoveVFX dentro de Modules (como sugerido na estrutura) deixa claro que se trata de dados de módulos, replicados no jogo (disponíveis tanto no servidor quanto no cliente, caso necessário). Alternativamente, se preferido, MoveVFX poderia residir em outra hierarquia (por exemplo, dentro de uma pasta Effects&SFX se existente). Contudo, para manter a organização paralela, é válido tê-lo em Modules/MoveVFX/. Isso não conflita com Modules/MovesModules/, pois um lida com lógica do movimento e outro com visual do movimento.
* **Associação movimento <-> VFX:** Permanecerá **por nome/ID**. Ou seja, o move.id definido em cada módulo de movimento deve corresponder a uma entrada no MoveVFX. No exemplo BasicAttack, id = "BasicAttack" e no MoveVFX haverá um pacote para "BasicAttack". Essa convenção simples evita ter que armazenar referências cruzadas nos módulos. Também significa que se quisermos renomear um movimento (ID), devemos lembrar de atualizar sua entrada de VFX. Em teoria, poderíamos incluir no módulo de movimento um campo vfxId caso o nome difira, mas provavelmente não há caso disso no design atual – então simplesmente seguir a regra de mesmo ID em ambos lugares.

Em resumo, **a recomendação para os efeitos visuais** é manter o sistema existente, apenas reestruturando arquivos se necessário para limpeza. A principal mudança é organizacional: assegurar que há um local claro para editar os visuais de um movimento (por exemplo, *“Precisa ajustar a cor do efeito de VoidRay?”* – saber que isso estará no VoidRayVFX.lua em vez de perdido no meio de um grande módulo). Isso também facilita colaboração entre programadores e artistas. Tecnologicamente, o funcionamento continua: o CombatManager/Bridge chama MoveVFX.Resolve(id) -> obtém pacote -> Bridge playStage lida as partículas/sons.

*(Vale notar que a refatoração dos movimentos praticamente não afeta o cliente nem as animações, apenas reorganiza como os dados são fornecidos. Os mesmos assets e funções Roblox (\_G.EFFECTS:PlaySound, EmitterEffect, etc.) serão usados como antes. Portanto, o jogador não percebe diferença – mas os desenvolvedores sim, positivamente na manutenção.)*

## Recomendações Futuras e Considerações Técnicas

Ao concluir a refatoração para módulos individualizados de movimentos, abre-se espaço para aprimoramentos futuros no sistema de combate. Listamos aqui algumas recomendações e ideias de evolução:

* **Uso de Tipagem Estática (Type Annotations):** Aproveitar o suporte a *type checking* do Luau (Roblox Lua) para definir tipos formais para os objetos movimento. Por exemplo, declarar um tipo MoveDefinition que descreva o shape esperado:
* export type MoveDefinition = {  
   id: string,  
   name: string,  
   rarity: string,  
   power: number,  
   cooldown: number,  
   minRange: number,  
   maxRange: number,  
   castTime: number,  
   castSpeed: number?,  
   lockMovement: boolean?,  
   tags: {string},  
   effects: {string},  
   accuracy: number?,  
   critChance: number?,  
   mpCost: number?,  
   execute: (attacker: Model, target: Model) -> ()?  
  }
* e usar esse tipo nas funções (e.g. function Moves:GetMove(race: string?, id: string): MoveDefinition?). Isso ajudará o autocompletar do editor e previne alguns erros (por exemplo, esquecimento de um campo obrigatório ao adicionar um novo movimento). Durante a refatoração, já podemos inserir essas anotações nos módulos e no Moves principal.
* **Metatables e POO:** Considerar adotar um padrão de projeto mais orientado a objetos para movimentos. Atualmente, cada movimento é uma simples tabela de dados. Poderíamos transformar cada um em um objeto com métodos. Uma abordagem seria definir uma “classe base” Move e setar como metatable de cada movimento, provendo métodos comuns. Por exemplo, um método move:GetDisplayName() ou move:IsOffensive() (checando se tem tag de ataque) poderia ser útil. Outra ideia com metatable: implementar \_\_index nos movimentos para retornar valores default caso acessem algo não definido (embora com os cálculos iniciais isso seja desnecessário). Se o jogo for expandir com lógica mais complexa de movimentos (ex.: cooldowns dinâmicos, upgrades de movimentos, etc.), pensar em representar cada movimento possivelmente como instância de uma classe poderia facilitar (por ora, não parece necessário porque movimentos são fixos).
* **Isolamento para Testes Automatizados:** Com cada movimento em seu módulo, fica viável escrever testes unitários para eles. Recomenda-se criar casos de teste para:
* Verificar que os metadados estão consistentes (ex.: soma de ranges faz sentido, cooldown não negativo, etc.).
* Testar que computeAccuracy/computeCrit/computeMpCost produzem valores esperados para cada movimento (podemos fixar cenários).
* Se existir função execute custom em algum movimento, testar em condições simuladas (por exemplo, mock de attacker/target) se o efeito aplicado corresponde ao design. Pode-se integrar uma *suite* de testes usando frameworks como TestEZ ou semelhantes no Roblox, garantindo que futuras modificações nos movimentos não introduzam regressões. Cada módulo individual facilita rodar teste isolado (ex.: require do movimento, aplicar função, checar resultados).
* **Extensão para Multiplayer e Segurança:** Em ambientes multiplayer, é importante assegurar que os movimentos funcionem de forma determinística e segura. Como muitos desses cálculos ocorrem no servidor (por exemplo, o acerto, dano e aplicação de status via CombatManager), o refactoring não muda isso – o servidor continuará sendo autoridade. No entanto, podemos:
* Verificar se alguma parte deveria rodar no cliente para fluidez (como animações). Em geral, a execução visual já está replicada via efeitos. Se no futuro permitirmos que o cliente antecipe algo (ex.: previsões de hit), devemos manter a fonte da verdade no servidor.
* Sanitizar inputs: se um cliente solicita usar um movimento, validar que esse movimento existe em Moves e que a criatura/jogador realmente o possui disponível. A modularização não altera isso, mas ter movimentos organizados torna trivial iterar e validar IDs.
* Sincronização de novos movimentos: Com módulos separados, adicionar novos movimentos requer distribuí-los no jogo (por exemplo, via update no Roblox). Deve-se garantir que versão de servidor e cliente tenham os mesmos movimentos carregados para evitar descompasso de animações ou expectativas. A estrutura modular não dificulta isso – apenas siga os mesmos processos de publicação.
* **Possibilidade de Conteúdo Dinâmico/Plugins:** Uma vantagem indireta da modularização é que poderíamos pensar em *plugins* ou DLCs adicionando movimentos simplesmente colocando novos ModuleScripts na pasta. O sistema de carregamento dinâmico os incluiria automaticamente. Isso torna o jogo mais extensível sem mudar código existente. É interessante planejar se alguma parte do jogo (como evolução ou loot) deveria automaticamente reconhecer novos movimentos – possivelmente integrando com configurações (ex.: adicionar novo movimento em uma família de raça para sorteio). Com a estrutura modular, tais alterações localizam-se no lugar certo (por exemplo, se for um movimento para uma família nova, adicionar a tag correspondente garante que GetPoolForRace o considere).
* **Organização por Categoria (Opcional):** Se a lista de movimentos crescer muito, podemos sub-organizar a pasta MovesModules em subpastas por categoria ou família. Por exemplo: MovesModules/Progenitor/WeavePulse.lua, MovesModules/Construct/CoreBash.lua, etc. Isso pode espelhar a organização de raças. Entretanto, isso complica um pouco o require dinâmico (precisa percorrer subpastas) e talvez seja desnecessário se o número de movimentos for gerenciável. Por ora, manter todos em uma pasta flat pode ser suficiente, mas fica a sugestão para futuro se houver dezenas de movimentos adicionais.
* **Documentação e Comentários:** Como melhoria de engenharia, documentar cada movimento no topo do seu módulo, explicando o que ele faz, pode ajudar designers e devs. Por exemplo, comentário dizendo “-- VoidRay: Dispara um raio de energia que causa dano penetrante e aplica Debuff (redução de defesa) no alvo.” Isso não impacta código, mas a modularização encoraja, já que agora há um lugar específico para escrever sobre cada movimento.
* **Referência Cruzada com Status Effects:** Notamos que muitos movimentos dependem de efeitos de status (como *Stun*, *Burn*, *Shield*, etc.), aplicados via sistema de StatusEffects no CombatManager[[10]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=CombatManager,LocationParts%3D%7Btarget.PrimaryPart%20or%20ta)[[28]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=~%3D,dur%3D5.0%7D%2Cattacker%29if%20_G.EFFECTS). Poderíamos no futuro melhorar a integração: talvez permitir que o movimento especifique duração ou intensidade do efeito (ex.: um movimento *Freeze* que congela 2s em vez do padrão 6s). Atualmente, essas durações estão fixas no código (Stun = 1s, outros = 6s)[[28]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=~%3D,dur%3D5.0%7D%2Cattacker%29if%20_G.EFFECTS). Com módulos, fica mais fácil passar parâmetros – por exemplo, um movimento poderia ter effects = { {name="Stun", dur=2.5} } em vez de apenas "Stun". A refatoração não implementa isso agora, mas a estrutura modular deixaria essa extensão mais orgânica no futuro.
* **Desempenho:** A princípio, dividir um módulo grande em vários menores não traz impacto negativo significativo no desempenho do jogo. O carregamento inicial via requires adicionais é mínimo frente ao total. Em compensação, ganhamos em *performance de desenvolvimento* (mais fácil achar e corrigir problemas). Entretanto, é bom continuar monitorando a eficiência do CombatManager, pois é onde acontece a execução de movimentos em tempo real. Certifique-se de que chamar possivelmente uma função move:execute() extra não degrade nada perceptível. Provavelmente não, mas teste cenários com vários movimentos simultâneos para confirmar.

## Conclusão

A refatoração proposta redesenha o sistema de movimentos de combate com foco em **modularidade e organização**, sem alterar a mecânica de jogo do ponto de vista do jogador. Cada movimento ficará isolado em seu próprio módulo contendo sua definição completa, o que facilita manutenção e futuras evoluções. O módulo principal Moves se torna um mapeador leve, responsável por agregar os movimentos e fornecer utilitários comuns (como cálculo de accuracy, crit, etc.), conforme o comportamento já existente[[4]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=computeAccuracy%28m%29local%20acc%3DDEFAULTS,45%20then%20crit)[[29]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=computeCrit%28m%29local%20crit%3DDEFAULTS,maxCrit%29end%20local%20function). Essa separação de responsabilidades segue a arquitetura já empregada em outros subsistemas do jogo (como entidades/raças), tornando o projeto mais uniforme e de fácil compreensão.

Ao implementar essa refatoração, esperamos os seguintes **benefícios imediatos**: - Redução drástica do tamanho e complexidade do arquivo Moves.lua. - Facilidade para adicionar, remover ou ajustar movimentos sem risco de efeitos colaterais nos demais. - Maior legibilidade: desenvolvedores podem focar em um movimento por vez, entendendo todas as suas nuances em um único arquivo pequeno. - Possibilidade de colaboração paralela (várias pessoas podem trabalhar em diferentes movimentos simultaneamente, já que estão em arquivos distintos, minimizando conflitos de merge). - Alinhamento com princípios de **Design Modular** e **SOLID** (em particular, o Single Responsibility Principle – cada módulo de movimento lida apenas com aquele movimento).

Por fim, as recomendações futuras garantem que o sistema continue evoluindo de forma sustentável: adotando tipagem para evitar bugs, permitindo polimorfismo de movimentos via metatables se necessário, e assegurando que tanto a lógica de jogo quanto a apresentação visual permaneçam bem desacopladas e configuráveis. Com uma base modular sólida, o jogo pode crescer em conteúdo (novos movimentos, novas mecânicas) com menos esforço e menor propensão a quebrar funcionalidades já estabelecidas.

**Referências de Código:** Utilizamos trechos do código original do módulo Moves e relacionados para embasar a proposta. Esses trechos estão referenciados entre colchetes ao longo do texto, por exemplo, mostrando como era definida a tabela de movimentos[[1]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=w,power%3D30%2Ccooldown%3D4.0%2Cmin) e como certos valores eram ajustados via overrides[[2]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=%3A,nil%20then%20m.mpCost%3Dov.mpCost%20end). Tais referências servem para comparar o *antes* e *depois* e assegurar que a refatoração preservará a lógica de negócio existente. Em especial, mantivemos atenção à parte de cálculo de atributos derivados[[4]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=computeAccuracy%28m%29local%20acc%3DDEFAULTS,45%20then%20crit) e aplicação de efeitos durante o combate[[10]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=CombatManager,LocationParts%3D%7Btarget.PrimaryPart%20or%20ta), para que o comportamento final dos movimentos refatorados seja equivalente ao sistema legado, agora com uma estrutura muito mais modular e organizada.

[[1]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=w,power%3D30%2Ccooldown%3D4.0%2Cmin) [[2]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=%3A,nil%20then%20m.mpCost%3Dov.mpCost%20end) [[3]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=local%20Moves%3D,min%28b%2Cx%29%29end%20local%20function) [[4]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=computeAccuracy%28m%29local%20acc%3DDEFAULTS,45%20then%20crit) [[5]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=VoidRay%3D%7Bid%3D,Debuff) [[6]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=k,NanoSwa) [[7]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=stage.effect%3D%3Dnil%20and%20stage._stage%3D%3D,local) [[8]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=endcast,Resolve%28moveId%29return%20buildResolved%28moveId%29end%20return%20M) [[9]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=any%7D%3D%7BBasicAttack%3D%7Bid%3D,power%3D30%2Ccooldown%3D4.0%2Cmin) [[10]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=CombatManager,LocationParts%3D%7Btarget.PrimaryPart%20or%20ta) [[11]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=clamp%28acc%2CDEFAULTS,maxCrit%29end%20local%20function) [[12]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=ipairs%28m.tags%29do%20mult%20,B) [[13]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=xRange%3D14%2CcastTime%3D0.75%2Ceffects%3D%7B,po) [[14]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=m,tags%20then%20for%20_%2Ct%20in) [[15]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=Crit%3D0.35%2CrarityMpMult%3D%7BAmateur%3D1.00%2CAdvanced%3D1.10%2CSpecialist%3D1.20%2CAscended%3D1.35%20%2CPrimordial%3D1.50%7D%2CtagAccuracyAdj%3D%7BMelee%3D0.04%2CRanged%3D,castAccuracyPenaltyPerSec%3D0.06%2CrangeAccuracyPenaltyPer10%3D0.015) [[16]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=clamp%28acc%2CDEFAULTS,45%20then%20crit) [[17]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=computeCrit%28m%29local%20crit%3DDEFAULTS,maxCrit%29end%20local%20function) [[18]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=computeMpCost%28m%29local%20power%3Dmath,B) [[19]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=%2Ctags%3D%7B,Spec) [[20]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=buildPoolForFamily,then%20local%20moveData%3DMoves%3AGetMove%28race%2CmoveName%29if) [[21]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=Races%3AGetBaseRaceFamily,return%20result%20end%20end%20local) [[22]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=,race%29end%20return%20nil%20end%20function) [[23]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=family%20then%20local%20pool%3D,EvolutionDefs%29local) [[24]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=then%20return%7B%7Bmove%3DbasicAttack%2Crarity%3D,random%28%29local%20acc%3D0%20local) [[25]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=chosenRarity%3A%20string%3F%3Dnil%20for%20_%2Cr%20in,move%3Dmv%2Crarity%3Dmv.rarity%7Dend%20return%20picked%20end%20Moves.RARITY%3DRARITY) [[26]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=end%20end%20local%20STARTERS_BY_FAMILY%3D%7BProgenitor%3D%7B,return%20pool%20end%20local%20function) [[27]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=mv%3Dbucket%5Bmath.random%281%2Cmath.max%281%2C,Moves) [[28]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=~%3D,dur%3D5.0%7D%2Cattacker%29if%20_G.EFFECTS) [[29]](file://file_000000009ae8620eb2d60b4f6a3a151a#:~:text=computeCrit%28m%29local%20crit%3DDEFAULTS,maxCrit%29end%20local%20function) MonsterGame.lua

<file://file_000000009ae8620eb2d60b4f6a3a151a>