1 | Arquitetura de Games

Fábio Daniel e Tiago P. Teixeira

Arquitetura de sistemas é um tópico que, freqüentemente, causava repulsa em alguns desenvolvedores. Era comum, ainda na década de 90, encontrar programadores que viam o processo de análise, documentação e modelagem prévia de dados como uma enorme perda de tempo. Diziam que o tempo que se gasta nisso,era tempo que deixavam de realmente *programar* o sistema. E que, se no decorrer do projeto os diagramas e modelos precisam mudar de qualquer forma, então de nada adiantaria.

Mas a crescente competitividade do mercado foi acompanhada por uma forte demanda por cumprimento de prazos, metas e orçamentos. Nesse cenário, ficou evidente que os sistemas sem arquitetura ou planejamento demoravam mais tempo para ficar prontos, ou custavam muito mais do que o previsto, ou ambos. Mesmo nos casos onde uma versão utilizável era rapidamente disponibilizada, o custo posterior com manutenção e resolução de *bugs,* na ponta do lápis, provou-se enorme.

A profissionalização forçada delineou um novo cenário para a arquitetura de sistemas que, se não é ainda predominante, sem dúvida é a tendência de todos os desenvolvedores que buscam o sucesso a longo prazo. Todavia, estamos falando da arquitetura de sistemas convencionais, utilitários, voltadas ao usuário final. E nos games?

Nos games, a profissionalização e a adoção formal da arquitetura de sistemas como uma metodologia válida é... bem mais atrasada. Talvez por ser uma indústria de entretenimento, e um nicho essencialmente inovador – sempre explorando novas formas de interação e tecnologias – os desenvolvedores de games ainda não encaram os métodos formais de planejamento e documentação com a seriedade necessária.

O resultado, como vemos, são constantes atrasos no lançamento, estouros freqüentes do prazo (até em anos), elevadas despesas com re-programação de códigos mal estruturados, perda de controle sobre o trabalho dos funcionários, jogos que não atendem ao prometido. São poucas as desenvolvedoras que podem se dar ao luxo de passar por tais situações, e mesmo essas estão mais conscientes da necessidade de uma arquitetura sólida. Jogos bem-estruturados agilizam a criação de *mods* e expansões oficiais.

Vamos demonstrar nesse capítulo um exemplo bem simples de arquitetura de sistemas para games, usando técnicas de modelagem que irão ajudá-lo a produzir um projeto robusto e expansível, sem estourar seu orçamento ou seus prazos. Os diagramas aqui apresentados seguem linhas gerais da UML (Unified Modeling Language), mas simplificados. O interesse maior é direcioná-lo e inspirá-lo na criação de sua estrutura, e não ensinar UML. Ter uma noção de programação ajudará bastante na compreensão, mas tentaremos deixar as coisas mais claras possíveis.

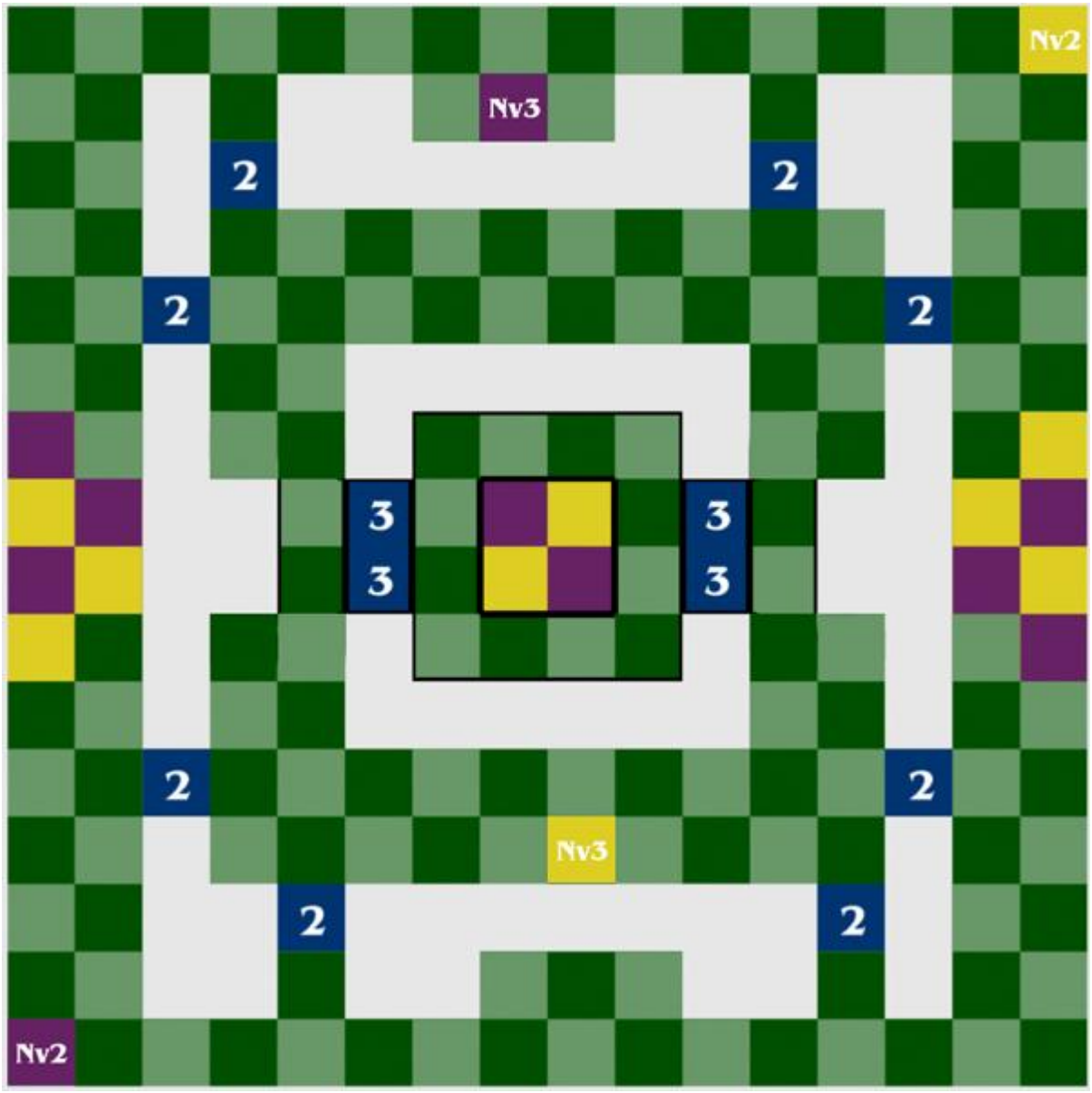
2 | Jogo: Espancobol

2.1 | Visão Geral

Espancobol é um jogo de estratégia por turnos, onde você comanda um time de personagens com habilidades distintas. Outro jogador comanda o time rival. O objetivo é agarrar a Bola no centro do campo e levá-la até a zona inimiga, ou tomar a Bola de seu adversário ou evitar que ele chegue com ela em sua zona.

O combate é fundamental para a vitória. Quando o seu personagem vence um combate, o derrotado é projetado para trás ou para os lados, e dificultando o avanço do time adversário. Se o derrotado for projetado para casas vazias (a água), você o elimina até a próxima partida, deixando seu adversário com um personagem a menos. Se o derrotado estava com a Bola, ela agora está com o seu personagem!

O campo de *Espancobol* é dividido por Casas e níveis de acesso. Seu time começa no nível 1, mas só terá acesso aos Níveis 2 e 3 se você conseguir levar um personagem certos botões localizados no mapa. Há um botão de nível para cada time, então você não ganha acesso ao nível 2, por exemplo, se posicionar um personagem no botão de nível 2 do adversário. Não há modo *single-player* em Espancobol.



2.2 | Personagens

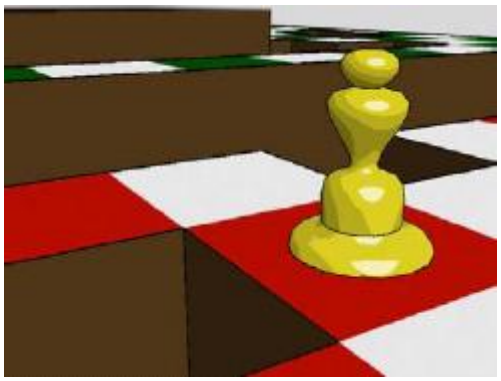
Todo time de Espancobol é composto por 6 personagens: 2 *Espancadores*, 2 *Corredores* e 2 *Muralhas*, cada um desempenhando uma função específica.

2.2.1 | Corredores

Esses são especialistas em correr, e servem fundamentalmente ao time. Eles geralmente são os responsáveis por ativar os botões de acesso aos níveis superiores, e estão sempre tentando adiantar-se em relação aos jogadores adversários. São rápidos, mas muito vulneráveis em combate.

- **Ataque 1;**
- **Defesa 1;**
- **Movimento 5**.

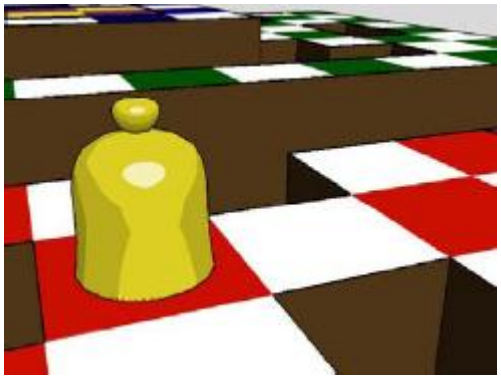
2.2.2 | Espancadores



Em um esporte onde a violência é permitida, é fundamental a presença de jogadores especialistas em agressão. Esses são os *Espancadores*, cuja função é evitar que os jogadores adversários se aproximem demais de seus colegas *Corredores*. Espancadores possuem os seguintes atributos:

- **Ataque 4;**
- **Defesa 1;**
- **Movimento 2.**

2.2.3 | Muralhas



Essas pilhas de músculo ambulante são extremamente úteis, pois servem para bloquear o avanço dos Espancadores adversários. Não são bons de briga, servem exclusivamente para conter os inimigos.

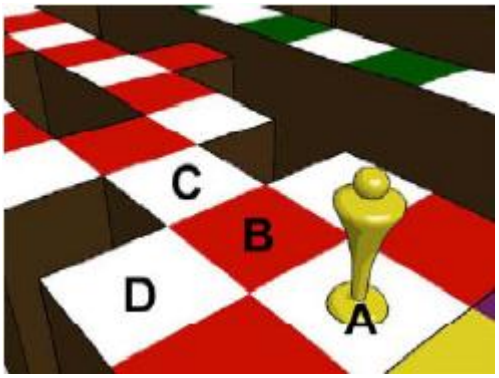
- **Ataque 1**
- **Defesa 4**
- **Movimento 2**

2.3 | Ações

Embora os times possuam 6 personagens, apenas 3 podem ser utilizados por rodada por cada usuário. E cada um desses 3 selecionados podem desempenhar até 2 ações, sendo que dessas necessariamente uma sempre é a de mover-se. Isso significa que, numa mesma rodada, um jogador pode mover e atacar ou mover e passar a bola, mas nenhum jogador pode numa mesma rodada atacar e passar a bola (ou vice versa). As ações de passe e de ataque não podem ser efetuadas antes de mover-se, de modo que se elas forem efetuadas, o jogador em questão não mais poderá deslocar-se na rodada.

2.3.1 | Movimento

Essa é a ação básica do jogo, e freqüentemente será a mais executada. Todo jogador pode, enquanto lhe restar Pontos de Movimento, deslocar-se *em cruz*, mas não pode deslocar-se em diagonal.



Exemplo de movimentação

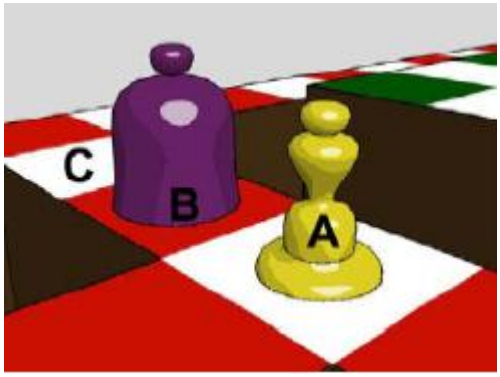
O Jogador Amarelo, no exemplo dado, pode deslocar-se do quadrado A até o quadrado B e C, mas não pode deslocar-se diretamente de A para D. Para chegar em D, ele teria que primeiro passar por B para aí sim deslocar-se para D. Numa mesma rodada, o jogador não pode ir de A para B e depois retornar para A novamente.

2.3.2 | Combate

É a segunda ação do jogo, mais complexa e definitiva. Para declarar tal ação, basta que o usuário desloque um de seus jogadores para o quadrado do alvo. O vencedor do duelo fica com o quadrado, e o perdedor será atordoado e deslocado. Existem três tipos de ataque: *Ataque frontal*, *Ataque lateral,* e *Arraste*.

Ataque Frontal

O ataque frontal é o mais comum, conforme ilustrado abaixo.



Exemplo de ataque frontal

No exemplo acima, temos um Espancador A confrontando um Muralha B. Como o *Ataque* do Espancador é 4 e a *Defesa* do Muralha também é 4, e nada acontece. Ataques de qualquer natureza só causam efeito se forem declarados contra alvos cuja Defesa seja inferior.

Um Espancador ataca um Corredor frontalmente.

Nesse novo exemplo, porém, temos uma diferença de níveis. O Espancador A, cujo *Ataque* é 4, declarou ataque ao Corredor B, cuja *Defesa* é 1. Sempre que ocorrem diferenças de níveis favoráveis ao atacante, essa diferença torna-se a distância por onde o alvo será deslocado para trás. Nesse exemplo, o Corredor seria arremessado 3 casas para trás (ATQ 4 – DEF 1 = 3); como não há um terceiro quadrado atrás do quadrado C, ele cai na água.

Caso o alvo esteja portando a Bola, ela passa a pertencer ao jogador que venceu o combate (no caso, o Corredor iria para a água sem a bola, que agora pertenceria ao Espancador). Nos casos em que depois de ser jogado para trás o alvo permanecer no tabuleiro, ele estará atordoado durante a rodada seguinte. Durante esse tempo ele continua ocupando o quadrado e obstruindo a passagem dos demais, mas não pode ser novamente atacado por um adversário. Caso haja um bloqueador de nível impedindo que o jogador seja arremessado para trás o suficiente, ele ocupará o quadrado adiante desse bloqueador.

Ataque lateral.

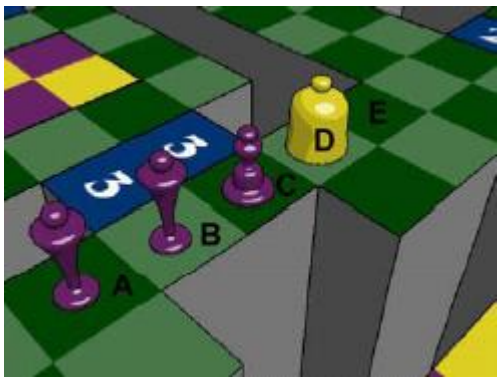
Exemplo de ataque pela esquerda

Na imagem acima, o Espancador A faz um *Ataque pela esquerda* no Corredor B. Como o Ataque do Espancador é 4 e a Defesa do Corredor é 1, essa ação pode ser efetuada. O Espancador assumiria o quadrado B enquanto o Corredor seria deslocado para o quadrado à sua esquerda (à direita do Espancador, seguindo a seta vermelha); como nesse caso não existe tal quadrado, o Corredor é lançado em água. Se o Corredor tivesse continuado no tabuleiro, na rodada seguinte ele estaria atordoado.

Ataques laterais (pela esquerda e pela direita), diferente dos frontais, não deslocam o alvo a vários quadrados de distância. Mesmo quando a diferença entre ATQ e DEF é superior a 1, o alvo sempre é deslocado para o quadrado ao lado.

Arraste

A última ação de combate chama-se *Arraste*. Ela consiste em combinar dois jogadores em linha para executar um ataque ou uma defesa. Quando a ação é declarada um Arraste é criado, e sua ação de mover tem como restrição o menor MOV da dupla. Vale lembrar que mover um Arraste significa utilizar 2 jogadores numa mesa rodada! Arrastes só são capazes de efetuar ataques frontais.



Exemplo de Arraste

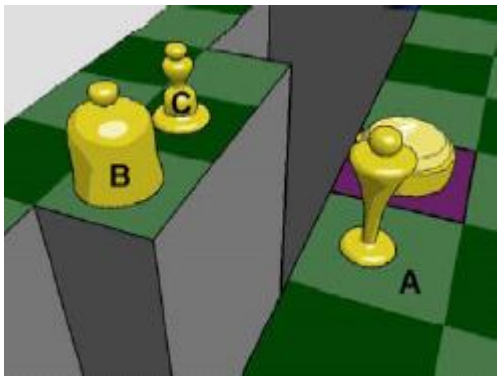
Quando um Arraste declara uma ação de ataque, soma-se os ATQs de ambos os jogadores, e compara-se o valor com a DEF do alvo. No exemplo acima, teríamos ATQ 5 x DEF 4, o que confere vitória ao Arraste. Note que o Arraste leva em conta a coma do ATQ do Corredor B e do Espancador C; o Corredor A, que está adjacente aos colegas, não faz parte da contagem. Se o Muralha tentasse declarar um ataque contra o Arraste sairia perdendo, pois não iria superar a DEF do mesmo (que seria 2, no caso). Se o time amarelo conseguisse deslocar o Arraste para trás, os 3 jogadores (incluindo o Corredor A) seriam deslocados para trás.

Vale lembrar que, obviamente, ataques frontais não podem ser declarados contra alvos que se encontrem encostados em bloqueadores de níveis, pois nesses casos eles não podem ser deslocados para trás.

Um Espancador pode atacar lateralmente o inimigo frontal de um Arraste, desde que esse não seja um Muralha. Se no exemplo anterior o Muralha D fosse um Espancador, este poderia atacar lateralmente o Espancador C e jogá-lo para fora do campo, desmanchando o Arraste. Ainda que sobre na seqüência 2 jogadores em fila, o usuário deles precisaria esperar sua rodada para declarar que eles formam um novo Arraste.

2.3.3 | Passe

Quando o Nível 3 for ativado e algum jogador vier a dominar a bola, ela passará a ser um importante elemento do jogo. O jogador que assim o fizer estará de posse dela, podendo assim permanecer até marcar ponto (ou ter a bola tomada por algum adversário), ou ele pode passar a bola para algum companheiro. Para isso, basta que ambas as miniaturas estejam adjacentes (exceto diagonais), e que a ação de *Passe* seja declarada.



Exemplo de passesl

A exceção dessa regra aplica-se aos Corredores. Esses jogadores, que servem especificamente para portar a bola, podem passá-la adiante em um raio de até 1 quadrado de distância, não havendo a necessidade de que o receptor esteja adjacente. Mas mesmo nessa situação, diagonais não se aplicam. No exemplo acima o Jogador B (que é um Muralha) pode passar a bola para o Jogador C (que é um Espancador), e o Corredor A pode passar a bola para o Muralha B, mas não pode passá-la ao Espancador C.

Vale lembrar que, no caso de uma ação de passe, o jogador que recebe a bola não se inclui entre os penalizados com tal restrição. Se um jogador A passa a bola para um jogador B antes de mover-se, ele não mais poderá fazê-lo nessa rodada; por outro lado, mesmo que o jogador B tenha recebido a bola antes de mover-se, ele poderá fazê-lo normalmente porque ainda não declarou nenhuma ação (logo, receber bola *não é* ação). É importante lembrar, também, que Arrastes são agrupamentos de 2 jogadores. Isso quer dizer que, ao executar qualquer ação com um Arraste (passar bola, atacar ou mover), implica dizer que 2 jogadores estão sendo utilizados; logo, na mesma rodada somente mais um jogador poderia ser utilizado. Numa mesma rodada não podem executar ações um Arraste mais dois jogadores distintos, muito menos 2 Arrastes!

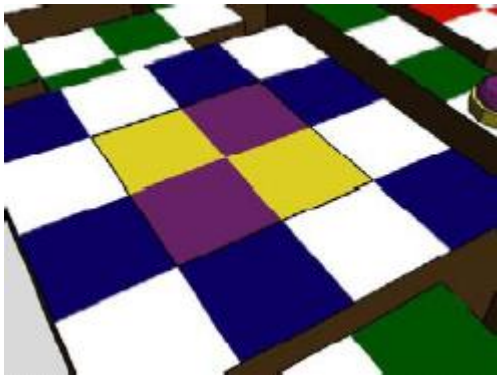
2.4 | O campo (mapa)

Níveis do Campo

O campo é dividido em 3 níveis, aumentando a complexidade da partida e aumentando suas possibilidades. Os níveis são divididos por marcações numeradas nos tabuleiros, que indicam o acesso aos níveis superiores (internos). Os níveis não interferem diretamente nas regras do jogo, basicamente apenas limitando o alcance de efeito dos *Botões* (ver adiante). Para os jogadores de ambos os times terem acesso ao próximo nível do tabuleiro, alguém precisa pressionar o respectivo *Botão de acesso* (ver adiante).

O campo é composto por 2 zonas específicas: *Zona de início* (onde os times iniciam e onde os adversários precisam chegar com a bola pra marcar pontos), e *Zona da bola* (onde ela se encontra, até que alguém a busque).

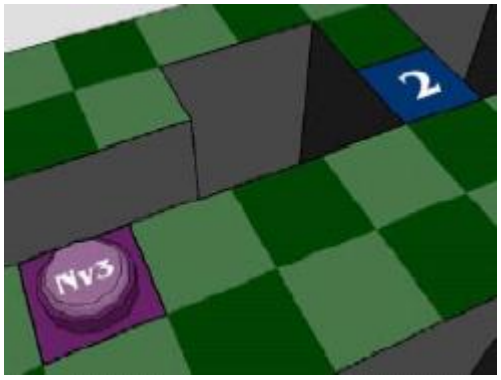
Exemplo de formação clássica, dentro da Zona de início.



Zona da bola, Nível 3.

Essa zona é o objetivo de ambos os times na primeira etapa da partida. Quando a partida se inicia, ambos os times precisam deslocar-se até ela para pegar a bola. Ela estará em algum ponto aleatório desntro dessa Zona. Marca ponto quem chegar na Zona de início adversária carregando a bola, e pisar em qualquer de seus quadrados. A partida termina quando um dos times marcar ponto.

2.4.2 | Botões e Bloqueadores de Níveis



Botão de acesso ao nível 3 (Nv3) e um Bloqueador de Acesso ao nível 2.

Os *Botões* servem para permitir o acesso aos níveis superiores. Existe um botão para cada time no nível 1, e um no nível 2. O do nível 1 permite acesso ao nível 2 (Botão Nv2), e um no nível 2 que dá acesso ao nível 3 (Botão Nv3). Vale lembrar que ele só pode ser pressionado por um jogador do respectivo time. Caso um jogador do time amarelo pise no botão Nv2 do time lilás, nada acontecerá.

Bloqueadores de acesso,

Como o nome sugere, servem para impedir o avanço dos times aos níveis superiores. Enquanto nenhum dos times pressionar algum botão, ambos estarão impedidos de acessar níveis superiores. Se apenas o time lilás tiver pressionado o respectivo botão Nv2, apenas eles poderão passar por tais bloqueadores; o time amarelo continuará sendo impedido.

3 | Metodologia

Antes de começarmos a efetivamente modelar e dar forma à estrutura de nosso jogo, vamos ver um pouco da metodologia de desenvolvimento que norteia um bom trabalho de arquitetura.

3.1 | Análise

Você precisa ser um analista. Você não precisa se dedicar totalmente à análise, mas precisa usá-la no dia-a-dia. Se você for um programador, terá uma excelente base para começar a modelar seu projeto. Se você não souber programar, ou souber muito pouco, recomendamos que ganhe algumas noções de lógica e organização de linguagens orientadas a objeto, sobretudo C++ e Java. Mas mesmo linguagens como Visual Basic ou Delphi serão suficientes para fazer seus primeiros diagramas e planejamentos.

Você pode começar suas análises usando texto, escrevendo suas idéias, desenhando à mão relacionamentos entre os elementos do jogo, montando tabelas de propriedades. Isso já ajudará. Você não precisa saber UML, mas esse padrão de modelação e planejamento visual fornece diagramas muito úteis, e facilita a comunicação com os programadores.

Então a análise é a primeira coisa que você fará no seu game. Ela irá lhe dar uma noção bem mais clara de que tipo de código será necessário, como tratar os eventos do jogo, e sobretudo da real dimensão do projeto e das dificuldades.

Um aspecto importante é que você nunca irá parar de fazer análise. No decorrer do projeto, mudanças na estrutura serão inevitáveis, e os diagramas idealizados anteriormente terão de ser refeitos. Então por que fazer a análise em primeiro lugar? Porque essas mudanças estruturais serão muito menos freqüentes. Tomarão menos tempo também, pois você automaticamente saberá que partes do sistema precisam ser modificadas.

3.2 | Expansibilidade e Prazos

Estamos sempre procurando colocar mais funções em nossos games. É fascinante quando você vê seus personagens se comportando do modo como você imaginou. Mas como você é criativo também está sempre imaginando coisas novas. O problema é a forma como você implementa as novas funções em seu projeto.

Sem um planejamento, você acabará tentando modificar os elementos de seu jogo sem ter concluído e testado outros elementos mais fundamentais. Ou precisará reescrever todo o código, porque a nova idéia não é compatível com a estrutura anterior. Quando a versão final estourar o prazo dado pelos investidores ou apresentar *bugs*, terá sido perdido tempo, dinheiro e credibilidade.

Precisamos de um plano de desenvolvimento por *Etapas*. Em cada Etapa, teremos uma versão completamente jogável (e, sobretudo, *comercializável*) que, embora não tenha as funções avançadas que você sonha como ideais, pode ser lançada se o prazo ou o orçamento encurtarem.

É recomendável que as Etapas sejam delineadas no momento inicial da análise, mas que mudanças nesse planejamento só ocorram para Etapas posteriores à que está sendo implementada. Não recomendamos que você mude o objetivo de uma Etapa enquanto a executa Dessa forma, mantêm-se a coesão do que está sendo implementado no presente, e planeja-se melhor o cronograma do futuro. Também recomendamos que você especifique as implementações por cada componente do sistema.

Usando nosso exemplo do Espancobol, iremos implementar o design básico do jogo como a Etapa 1, delineando as Etapas seguintes da seguinte forma:

Componentes	Etapa 1	Etapa 2	Etapa 3
Personagens	Espancador, Muralha e Corredor. Times de 6 personagens fixos.	Demolidor, Rebatedor, Espião, Aríete. Times variáveis conforme o mapa.	Canhoneiro, Capitão Guincho. Times escolhidos pelos jogadores.
Campo (mapa)	Casas: - níveis diferentes; - bloqueadores; - água (mortal); - dos times; - botões de acesso; - Marca da Bola. Um mapa fixo em 16x16. Texturas fixas para as Casas.	Casas: - botões de choque; - muros (não permitem passes de Bola); - casas explosivas. Diversos Mapas de tamanhos variáveis. Efeitos visuais para as Casas (choques, explosões).	Casas: - teleporte; - desmoronamento. - lama (movimento reduzido à metade). Texturas variáveis para as Casas (desmoronamento).
Itens	Bola	Capa de invisibilidade.	Cetro magnético.
Regras	Jogador move três personagens por turno. Pontua quem levar a Bola às casas do time. Repasse da Bola.	Personagem inativo fica invisível (Espião, poção de invisibilidade).	Limite de Tempo Placar: partidas que valem mais de um ponto. Múltiplas Bolas. Bolas por Time.
Combate	Ataque frontal, projeção variável após. Ataque lateral, projeção fixa (em 1). Arraste de ataque frontal com dois personagens.	Ataque lateral, projeção variável (para Rebatedor). Arraste de ataque frontal com mais personagens, ataque frontal de projeção fixa (para Aríete). Ataque com Explosão (para Demolidor).	Ataque dorsal (para Capitão Guincho). Ataque à distância (para Canhoneiro), limitado pelas Casas que também permitem Passes.
Extras	---	---	Editor de mapas

Repare que essa mesma metodologia pode ser usada também na parte artística: basta substituir os “componentes” em grupos de trabalhos como Texturas, Cenários, Animação, Modelos 3D, Música e etc.

Protótipo

O protótipo é uma implementação rápida das principais idéias do game, anterior à Etapa 1. É a “Etapa Zero”. O objetivo é verificar a viabilidade das idéias e excluir rapidamente *gameplays* ruins ou pouco divertidos. Por ser apenas uma demonstração, não é necessário que passe por um processo rigoroso de análise. Pode ser implementado diretamente, conforme se vai tendo idéias. Mas por causa disso o código dos protótipos não deve ser reutilizado (a não ser que esteja em conformidade com seu rigoroso planejamento nas Etapas subseqüentes).

Arquivos de Configuração XML

Quando olhamos para nossas Etapas, vemos que alguns componentes precisarão ler arquivos de configuração. O componente Campo é o caso mais óbvio, pois para implementar um Editor de Mapas cada um precisará ser salvo em um arquivo diferente. Mas estamos também criando novos Personagens e Itens, e como iremos ver, defini-los em um arquivo separado em vez de embutidos no código facilitará a criação de novas versões. A criação de *mods* pelos fãs também será fácil, e como se sabe, os *mods* são incríveis para dar maior sobrevida comercial ao jogo (e a seu faturamento).

Faremos então a opção pelo uso de arquivos XML. Há excelentes editores de XML no mercado, e o arquivo é facilmente alterável, e pode-se usar até mesmo o Bloco de Notas do Windows. Os atributos das *tags* XML e a implementação de novas *tags* também são ótimas quando na expansão de funções do programa. Pode-se, além disso, criar um site oficial que lista Campos extras de acordo com critérios como dimensões, Casas, tamanho do Time, etc, pois as principais linguagens de programação *web* possuem excelentes ferramentas para consulta e manipulação de XML.

3.3 | Componentização e Reuso de Código

Repare que algumas funções listadas nas Etapas subseqüentes utilizam os mesmos conceitos, e portanto, podem utilizar os mesmos pedaços de código. Por exemplo:

Função	Descrição	Reuso
Invisibilidade	Deixa o personagem invisível durante o turno dos times adversários.	A mesma função é utilizada de forma permanente pelo Espião, e pelo personagem que usar a Capa de Invisibilidade.
Explosão	Projeta o personagem por 3 Casas, em uma direção aleatória.	A função é usada pelo ataque do personagem Demolidor e pelas Casas Explosivas.
Atração	É um ataque dorsal. O adversário é atraído em direção ao jogador.	Cetro magnético e qualquer ataque dorsal. Para o Cetro magnético, a função é chamada quando o turno do jogador acaba.

A metodologia de reuso do código e componentização é fundamental para termos um sistema conciso e de manutenção mais fácil. Se nos exemplos acima fizermos um código de Explosão e o copiarmos dentro dos códigos de personagem e dos de Casas, e quisermos modificar a projeção para 2 Casas, teremos que lembrar de fazer isso e procurá-los em ambos. Não é algo tão difícil nesse caso, mas imagine um grande projeto onde uma consulta a dados em um arquivo de configuração precisa ser feita em quinze classes diferentes de itens. Repetiríamos o código em cada uma delas? O custo de uma manutenção em tempo e dinheiro seria muito maior que apenas mudar uma parte que se propaga em todas as outras. Ou imagine que o código de um é usado de forma ligeiramente diferente do outro, e uma mudança pode causar *bugs* se não for feita com cuidado.

Portanto, vamos desde já nos acostumar a reusar o código. No caso acima, criamos uma classe que guarda eventos especiais e fazemos com que tanto as Casas quanto os personagens os herdem. Se o código precisa ser usado de forma ligeiramente diferente, iremos montá-lo para aceitar parâmetros que modificam seu funcionamento interno. As formas de reuso do código variam conforme a linguagem que você usa. Há heranças, funções prontas, classes e muitas outras formas. A orientação a objeto é uma excelente ajuda nesse sentido, mas nem sempre pode ser usada: como é mais lenta que o código “seco”, alguns jogos de alta performance não podem se dar ao luxo de usá-la em toda a sua extensão. Mas mesmo isso não os impede de usar outros métodos de reuso, como os *imports*.

4 | A Arquitetura

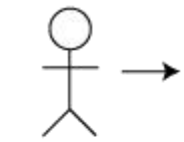
A análise e arquitetura demonstrada nesse capítulo *abrange a Etapa 1* de nosso projeto. Vamos ver como podemos estruturá-la, mas ao mesmo tempo vamos nos preparar e compatibilizar o sistema para as Etapas 2 e 3. Veremos, em linhas gerais, como a arquitetura montada previamente pode oferecer ampla expansibilidade em um projeto bem-estruturado desde o princípio.

4.1 | Casos de Uso

A primeira coisa de qualquer análise é estabelecer os casos de uso do sistema. Eles demonstram o que o usuário pode fazer, a que tipo de telas ele tem acesso, e, em linhas gerais, que componentes o sistema deve ter para atender à demanda.

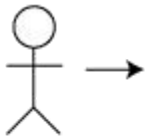
Se em sistemas convencionais eles são tão importantes quanto os diagramas de classes e objetos, em games os casos de uso são vitais. A ausência de casos de uso pode resultar em *gameplays* truncado, porque são neles que você começa a moldar a jogabilidade e entender o que pode e o que não pode dar certo, o que é intuitivo ou o que é confuso. Mesmo que você não tenha tempo ou experiência para estruturar as classes, componentes ou objetos do sistema do game, faça pelo menos seus casos de uso.

Em *Espancobol*, podemos delinear os seguintes casos de uso para nossos jogadores:



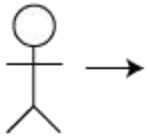
Caso de Uso: Interface

Interface
Campo: Montar a grade.
Campo: Renderizar as texturas e o cenário.
Campo: Renderizar os itens.
Campo: câmera- padrão
Personagens: Renderizar modelos.
Personagens: animações de combate, movimento, uso de itens.
Objetos do jogo: Coloca uma aura luminosa para diferenciar os objetos que estão sendo apontados pelo mouse.
Menus: Opções do jogo
Menus: Modos de câmera: isométrica, do topo ou acompanhando o personagem.
Áudio: música.
Áudio: efeitos sonoros.



Caso de Uso: Comandos do Jogador

Comandos do Jogador
Iniciar um novo jogo.
Clique Esquerdo: Selecionar os personagens. Não seleciona personagens atordoados.
Clique Direito: Aponta a Casa para onde o personagem deve se mover. O jogo calcula a rota mais rápida caso seja escolhido mais de uma Casa.
Clique Direito sobre Itens: o personagem pega o item.
Clique Direito sobre Personagens Rivais: um menu contextual oferece uma das opções de ataque disponíveis.
Clique Direito sobre Personagens Aliados: passar a Bola.



Caso de Uso: Interação entre Objetos

Interação entre Objetos
Personagem: Mover a quantidade de seu atributo Movimentação. Apenas para Casas adjacentes e que seu Time tenha acesso. Não move para a água.
Personagem: Passar a Bola. Pode ou não ser entre Casas adjacentes.
Personagem: Atacar sozinho, movimenta e gasta a ação de apenas de um personagem.
Personagem: Atacar em Arraste, movimenta e gasta a ação de dois personagens.
Personagem: Morte se atingir a água.
Combate: compara-se Ataque do agressor e Defesa do defensor.
Combate: Se em Arraste, soma-se o Ataque dos personagens que compõe o Arraste.
Combate, Vitória: Personagem ocupa o espaço e a Bola (se houver) do perdedor.
Combate, Derrota: Personagem é projetado para trás um número de Casas para o qual perdeu o combate, ou uma Casa se o ataque for lateral. A projeção acaba se o personagem atingir uma Casa de um nível inacessível ou a água.
Botão de Acesso: se um personagem entrar em sua Casa, libera o acesso de um time inteiro a um nível do campo.

Talvez você tenha reparado que esses diagramas de casos de uso não são iguais aos da UML. A linguagem UML oferece diagramas muito úteis para isso, mas procuramos mostrar a coisa de forma mais concisa e objetiva.

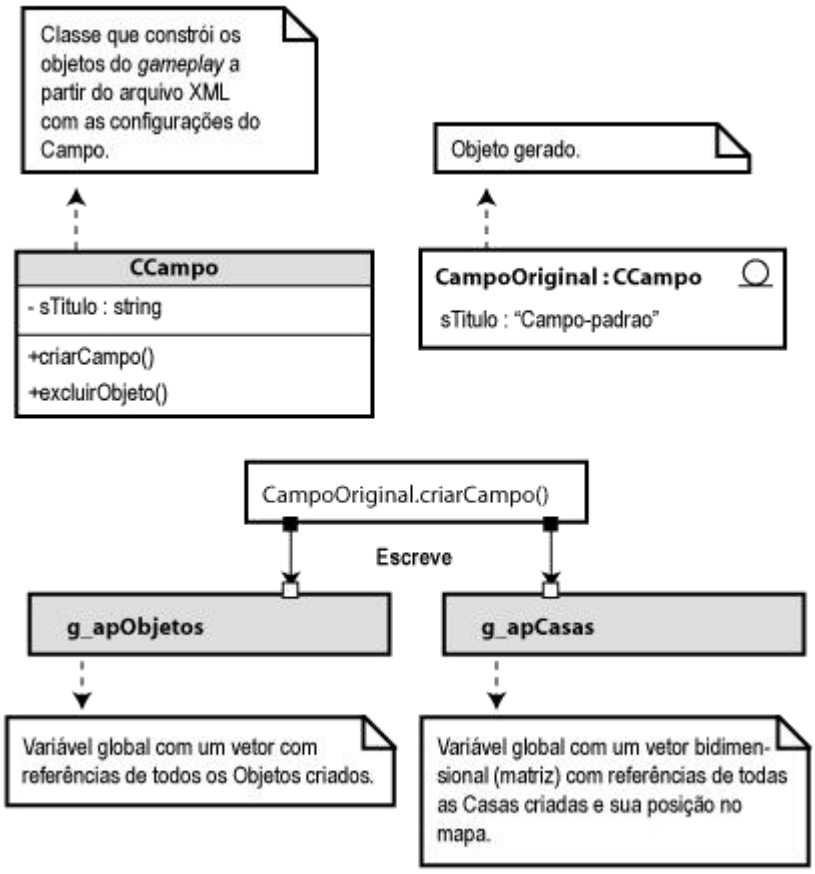
4.2 | Classes e Fluxogramas

Então temos agora o que nosso jogador pode fazer. É hora de traduzir modelar isso em estruturas e fluxogramas. Como a parte de componentes de exibição, renderização na tela e áudio constitui um assunto muito amplo para o escopo desse capítulo, abrangendo interfaces com componentes DirectX, OpenGL, DirectSound e outros, vamos tratá-los de forma mais superficial e nos concentrar nos componentes de *gameplay*. Vamos também ignorar o controle de erro (por exemplo, quando atribuímos mais personagens que o número de Casas dos Times na configuração de um Campo) para focar no principal.

4.2.1 | O Campo

Imediatamente, vemos a necessidade de criar funções que montem o Campo de jogo ao início de uma nova partida. O Campo-padrão, da Etapa 1, é uma grade de 16x16 Casas (de cinco tipos). No entanto, mais tarde nossos mapas terão tamanhos diferentes, inclusive irregulares (8x16, 9x5, etc), e haverá um Editor de Mapas. Portanto, os Campos, desde o Campo-padrão, serão montados a partir de arquivos XML dentro da pasta *campos* do diretório do jogo. Há um arquivo para cada Campo.

A construção do Campo começa na classe CCampo. A partir do nome do campo requerido, a classe chama as demais classes do jogo e criar todos os objetos do gameplay – personagens, Casas e itens – de acordo com as configurações no arquivo XML. Ele também grava um vetor com a referência a todos os Objetos criados numa variável global, e outro vetor bidimensional global com as referências aos Objetos Casas criados.



Dentro da classe CCampo, o arquivo XML do Campo é transformada nas seguintes variáveis:

Variável	Tipo	Função
iCampo_Altura	Integer	Quantas Casas de altura.
iCampo_Largura	Integer	Quantas Casas de largura.
iCampo_Bola	Integer	Quantas Bolas pode haver no campo. Valores de 0 e 1 sempre representam uma Bola.
aCampo_Casas	Array (vetor)	A configuração de cada Casa do Campo. Os Campos tem que ter sempre pelo menos uma Casa que seja Marca da Bola e Casas que acomodem os jogadores de cada Time.
iTime_Tamanho	Integer	A quantidade de jogadores em cada Time.

aTime_Prs	Array (vetor)	A configuração de cada Time no Campo.

Os membros do vetor aCampo_Casas são estruturas com as seguintes propriedades:

Variável	Tipo	Função
aCampo_Casas[i].sMaterial	String	O Material de que é feita a Casa.
aCampo_Casas[i].sItem	String	O Item que está sobre ela. Se for vazio, não há itens. Não pode haver Itens sobre Materiais do tipo “Mortal”.
aCampo_Casas[i].iTime	Integer	O Time relacionado à casa. Um valor de zero indica que a Casa não está relacionada a nenhum time (a maioria das Casas são assim).
aCampo_Casas[i].iNivel	Integer	O nível onde a Casa se encontra.

Os membros do vetor aTime_Prs são estruturas com as seguintes propriedades:

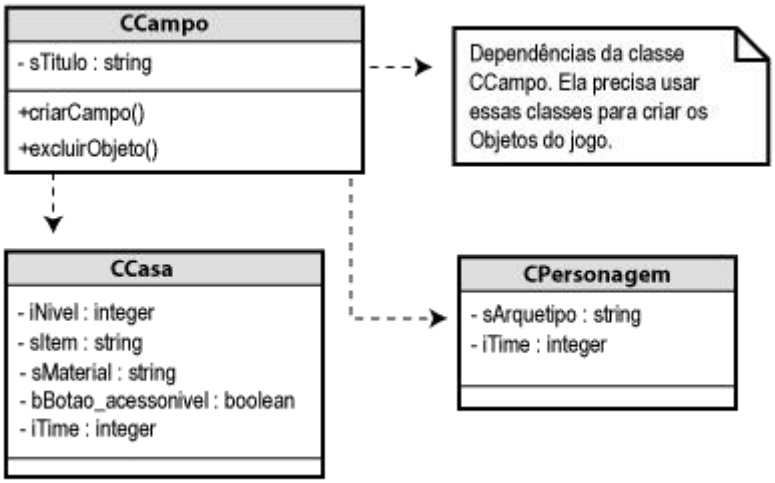
Variável	Tipo	Função
aTime_Prs[i].iTime	Integer	O número que identifica um time.
aTime_Prs[i].sArquetipo	String	O nome do tipo de personagem que deve ser inserido.

Na Etapa 2, essa estrutura permitirá que os tipos de personagem disponíveis e o tamanho dos times varie de acordo com o Campo. Deixando **aTime_Prs[i].sArquetipo** *vazio* poderemos tratar também os casos da Etapa 3 em que o jogador poderá escolher o tipo de personagem que integrará o time. Será também possível distribuir e configurar as Casas e os itens nelas como for conveniente.

Repare que não há uma propriedade que determina se a Casa deve ser ou não a *Zona da Bola*. Isso não é necessário, porque essa Zona não ativa nenhum evento no jogo. A *Zona de início* dos Times, por outro lado, ativa a vitória de um Time ao levar a Bola até ela, então precisa de uma identificação.

A diferenciação visual da Zona da Bola dá-se pelo uso de um Material diferente do comum. O posicionamento aleatório da Bola em sua Zona ocorre porque as quatro Casas possuem o Item “Bola” configurado no XML, mas **iCampo_Bola** permite um número menor (no Campo-padrão, apenas uma). O objeto CampoOriginal, por padrão, sorteia o número de Bolas entre as Casas configuradas para contê-la.

A classe CCampo depende das seguintes classes para construir os Objetos do jogo:



As setas pontilhadas de classe para classe significam que a classe CCampo depende das outras para funcionar. A seta pontilhada até o comentário é apenas uma indicação visual.

Veremos a seguir o funcionamento da classe CCasa criar um Objeto para cada Casa.

XML dos Campos

<campo valor= "*nome do mapa***" bola= "***1***">**

<dimensoes altura= "*16***" largura= "***16***">**

<casa nivel = "*1***" material = "***nome do material***" item = "***nome do item***" time = "***0***" />**

Cada tag <casa> determina os atributos de uma das Casas do mapa. As Casas são numeradas da esquerda para a direita, de cima para baixo. Assim, a Casa 3 de uma mapa de 8x8 será a terceira da esquerda para a direita da primeira linha, mas a Casa 9 será a primeira da segunda linha.

<casa nivel = "1" material = "nome do material" item = "nome do item" time = "0" />
<casa nivel = "1" material = "nome do material" item = "nome do item" time = "0" />
. . .

</dimensoes>

<time valor= "*6***">**
Quantos personagens podem participar de cada time.

<prs time= "*1***" arquetipo= "***Corredor***" />**
Determina o arquétipo do personagem de um time Y. (Ex: "Muralha").

<prs time= "1" arquetipo= "Muralha" />
<prs time= "2" arquetipo= "Corredor" />
. . .

</time>
</campo>

4.2.2 | Casas

Cada Casa do jogo possui propriedades que a distingue. Mas não vamos programar subclasses da classe CCasa para cada tipo. Vamos, em vez disso, colocar algumas dessas propriedades em *Materiais* configurados em XML (*material.xml*), que definem não só a textura que deve ser aplicada,

Antes de determinar a modelagem das Casas, é melhor fazer uma rápida matriz de propriedades que irão compor as Casas:

Casa	Acesso	Acesso por Time	Passe	Eventos	Efeitos
Normal	Não	Sim	Sim	---	Muda a textura
Bloqueadores de nível	Não	Sim	Sim	---	Muda a textura
Água	Atordoados	Não	Sim	Mata o Personagem.	---
Do Time	Sim	Não	Sim	Se um personagem adversário entrar com a Bola, o time dele vence a partida.	---
Botão de acesso	Não	Sim	Sim	Permite acesso de um time a um nível. Funciona apenas uma vez.	Muda a textura. Somente uma vez.
Zona da Bola	Sim	Não	Sim	---	---
Botões de choque	Não	Sim	Sim	Desfere um choque elétrico nos times adversários e os Atordoa. Funciona apenas uma vez.	Animação de raios. Somente uma vez.
Muros	Não	Não	Não	Não permitem passes de Itens ou ataques à distância através de sua Casa.	---
Explosivo	Não	Sim	Sim	Projeta o personagem 3 casas numa direção aleatória e o deixa Atordoado. Funciona apenas uma vez.	Explosão. Somente uma vez.
Teleporte	Não	Sim	Não	Muda o personagem para outro lugar no mapa. É unidirecional. (apenas leva, não trás o personagem de volta)	Animação de raios.
Desmoronamento	Não / Atordoados	Sim / Não	Sim	Mata os personagens depois que o atravessam um certo número de vezes.	Muda a textura a cada vez que um Personagem entra,

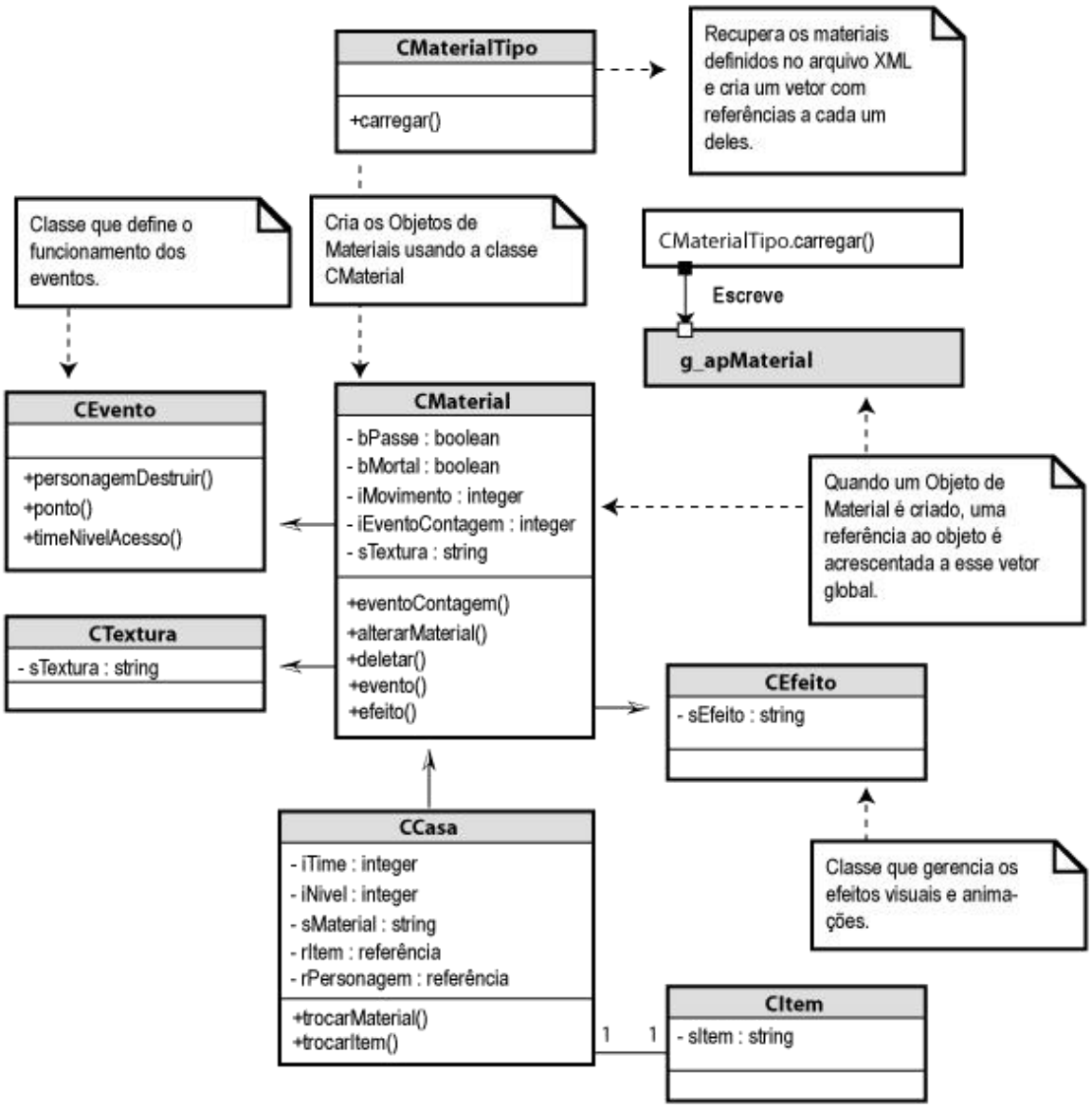
					terminando em abismo.
Lama	Não	Sim	Sim	Entrar na Casa custa dois movimentos ao personagem.	---

Precisamos preparar a estrutura de nossos Materiais para suportar eventos e efeitos especiais. Primeiro, constatamos que as Casas que matam personagens só podem ser acessadas por personagens Atordoados, porque os jogadores não podem comandar seus personagens a entrarem nessas Casas.

Temos Eventos e Efeitos que são ativados infinitas vezes ou em um número limitado de vezes. Mas os mesmos Efeitos podem ser utilizados uma ou infinitas vezes. As Casas são acessíveis por Time, ou são sempre acessíveis, ou nunca são acessíveis. Casas Mortais nunca são inacessíveis a qualquer time. Quase todas as Casas permitem que os Itens (principalmente a Bola) sejam passados através dela, mas nem todas. A Casa pode exigir um ou mais movimentos para ser acessada. Algumas Casas tem o mesmo comportamento de outras, mudando apenas a textura.

Por fim, toda Casa pode ou não ter um personagem ocupando-a. Casas ocupadas por um personagem não pode ser ocupada por outro.

Com isso em mente, seguimos em nossa modelagem:



As setas cheias entre uma classe e outra significam herança entre elas.

Os atributos de CCasa significam:

CCasa	Tipo	Função
iNivel	Integer	O nível ao qual a Casa pertence.
rItem	Referência	Referencia um Objeto Item que esteja sobre a Casa. Retorna NULL se não houver Item algum.
sMaterial	String	O Material que a Casa utiliza.
rPersonagem	Referência	Referencia um Objeto Personagem que esteja ocupando a Casa. Retorna NULL se não houver personagem algum.
iTime	Integer	O número do Time a que a Casa pertence. Se for 0, não pertence a time algum (quase todas as casas são zero).

Já os atributos de CMaterial significam:

CMaterial	Tipo	Função
bPasse	Boolean	Se um Passe de Item ou um ataque à distância pode ser feito através da Casa.
bMortal	Boolean	Se a Casa mata um personagem. Chama o evento *personagemDestruir().*
iMovimento	Integer	Quantas unidades de movimento de um personagem são necessárias para entrar na Casa. O padrão é 1. Com 0, a Casa do Material *não pode* ser acessada por personagens (o Muro teria iMovimento zero).
iEventoContagem	Integer	Contador que soma quantas vezes um personagem já entrou na Casa. Usado para regular eventos. Por exemplo, quando tivermos uma Casa que desmorona, podemos chamar o método *personagemDestruir()* quando o valor chegar a 4 ou 5. Somente os Materiais que invoquem o método *eventoContagem()* fazem esse controle. Os demais tem iEventoContagem sempre zero.
sTextura	String	Nome da textura que será desenhada.

Alguns métodos de CEfeito e CMaterial que valhem a pena ser comentados:

Evento	Função
Ponto()	Confere um ponto ao Time. Normalmente isso termina a partida, mas na Etapa 3 o evento será chamado várias vezes.
timeNivelAcesso()	Libera o acesso de um nível aos personagens de um time.
efeito()	Invoca o efeito visual de um evento()
personagemDestruir()	Apaga o Objeto de um dos personagens da memória. O personagem só voltará a existir na próxima partida. Todas as casas mortais chamam esse evento.

Confira a formatação do arquivo *material.xml*. A seguir, veremos as classes que moldam os Objetos Itens.

material.xml

<material valor= "*nome do Material***" >**

<passe valor= "*true***" />**

<mortal valor= "*false***" />**

<textura valor= "*lama***" />**

<movimento valor= "*2***" />**
O custo em movimento para que um personagem acesse a Casa. Com valor zero, nenhum personagem pode entrar na Casa.

<evento valor= "*0***">**

Contagem para aplicação do efeito.

<metodo comando= "*pontuar***" valor= "***false***" />**
Cada tag representa um dos eventos disponíveis para o Material. Caso algum novo evento de Etapas posteriores seja adicionado e as tags de um Material não sejam atualizadas não haverá problema, porque a classe CMaterialTipo considera só executa as tags encontradas de valor *true*.

<metodo comando= "*timeAcessoNivel***" valor= "***false***" />**
<metodo comando= "*personagemDestruir***" valor= "***false***" />**
<metodo comando= "*eventoContagem***" valor= "***false***" />**

. . .

</evento>

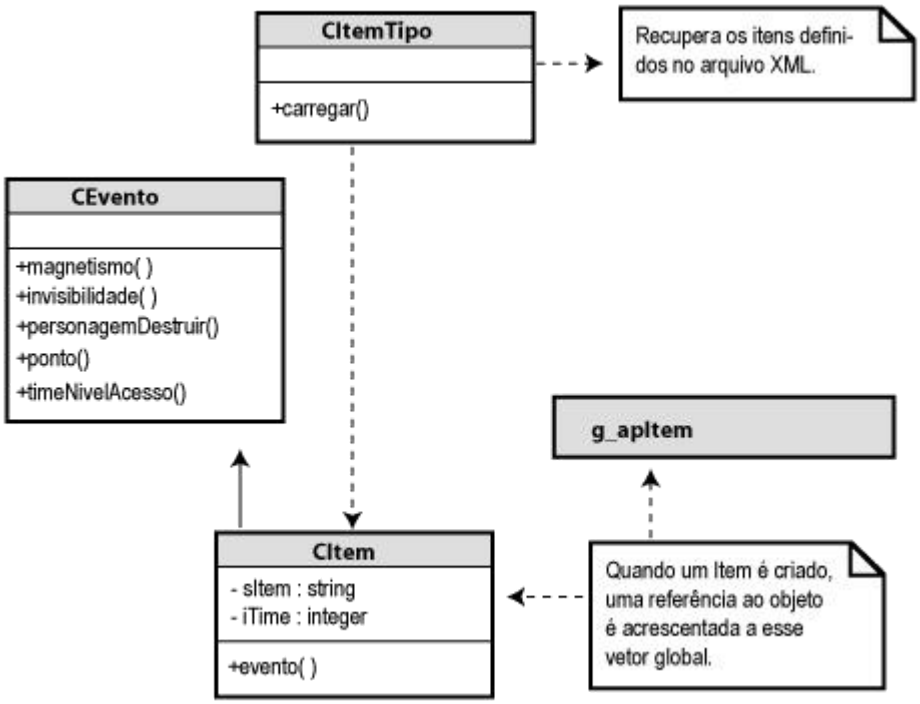
</material>

4.2.3 | Itens

Na Etapa 1, o único Item do jogo é a Bola. Mas iremos implementar uma estrutura que permitirá a criação de diversos outros Itens.

A primeira coisa dos Itens é que só há um Item por Casa. A segunda é que só há um Item por Personagem. A terceira é que os personagens podem ou não ser capazes de Passar um Item para outro personagem aliado, mas somente se o personagem já não tiver outro Item. A quarta é que as Casas "mortais" ou não podem guardar Itens. A quinta é que nas Etapas 2 e 3 eles poderão ser veiculados a Times e eventos estranhos (ativados ao fim do turno do jogador). Por último, os Itens não interagem entre si.

Repare agora que quase toda a mecânica do Item é definida nos outros Objetos do jogo, sobretudo Casa e Personagem. Os atributos do Item propriamente dito são poucos:



Repare que estamos utilizando a classe CEvento, a mesma do diagrama de Casas. Isso é intencional. Podemos compartilhar eventos entre Casas, Itens e personagens (a seguir), possibilitando a construção de elementos realmente estranhos como um arquétipo de personagem com magnetismo permanente (similar a um famoso mutante...). Os métodos que acrescentamos aqui, no entanto, estão vazios, e só serão implementados em suas respectivas Etapas.

O atributo **CItem.iTime** determina que um objeto seja privativo de um dos Times. Os demais personagens não poderão pegá-lo.

item.xml

<item valor= "*nome do Item***" >**

<evento>

<metodo comando= "*magnetismo***" valor= "***false***" />**
Cada tag representa um dos eventos disponíveis para o Item. Como os Materiais, caso algum novo evento de Etapas posteriores seja adicionado e as tags de um Item não sejam atualizadas não haverá problema, porque a classe CitemTipo considera só executa as tags encontradas de valor *true*.

<metodo comando= "*invisibilidade***" valor= "***false***" />**
. . .

</evento>

</item>

4.2.4 | Personagens

Os personagens são os Objetos mais importantes de *Espancobol*. Eles são o melhor exemplo de como uma arquitetura bem estruturada pode facilitar a expansão posterior do game. Vamos primeiro analisar a estrutura do Objeto personagem, e posteriormente iremos juntá-los com os Itens, Casas e Campo em fluxogramas que irão formar o jogo em si.

Quando fazemos o planejamento prévio de nosso game, observamos que o atributo Ataque de um personagem divide-se em outros dez atributos:

- Atributo Ataque que avalia o resultado em um combate;
- Um atributo Projeção para cada direção do ataque (frontal, lateral, dorsal);
- Três atributos que dizem se cada Projeção é relativa ou absoluta (se o inimigo derrotado anda uma quantidade de Casas igual à diferença entre Projeção e Defesa, ou se anda uma quantidade de Casas igual à Projeção).
- Três atributos que declaram o alcance em Casas de cada direção do ataque (frontal, lateral, dorsal);

Na Etapa 1, todos os personagens realizam ataques frontais relativos, ataques laterais absolutos, e não realizam ataques dorsais (valor zero). O algoritmo de combate da Etapa 1 sequer trata os ataques dorsais.

Podemos construir a seguinte tabela de atributos dos personagens:

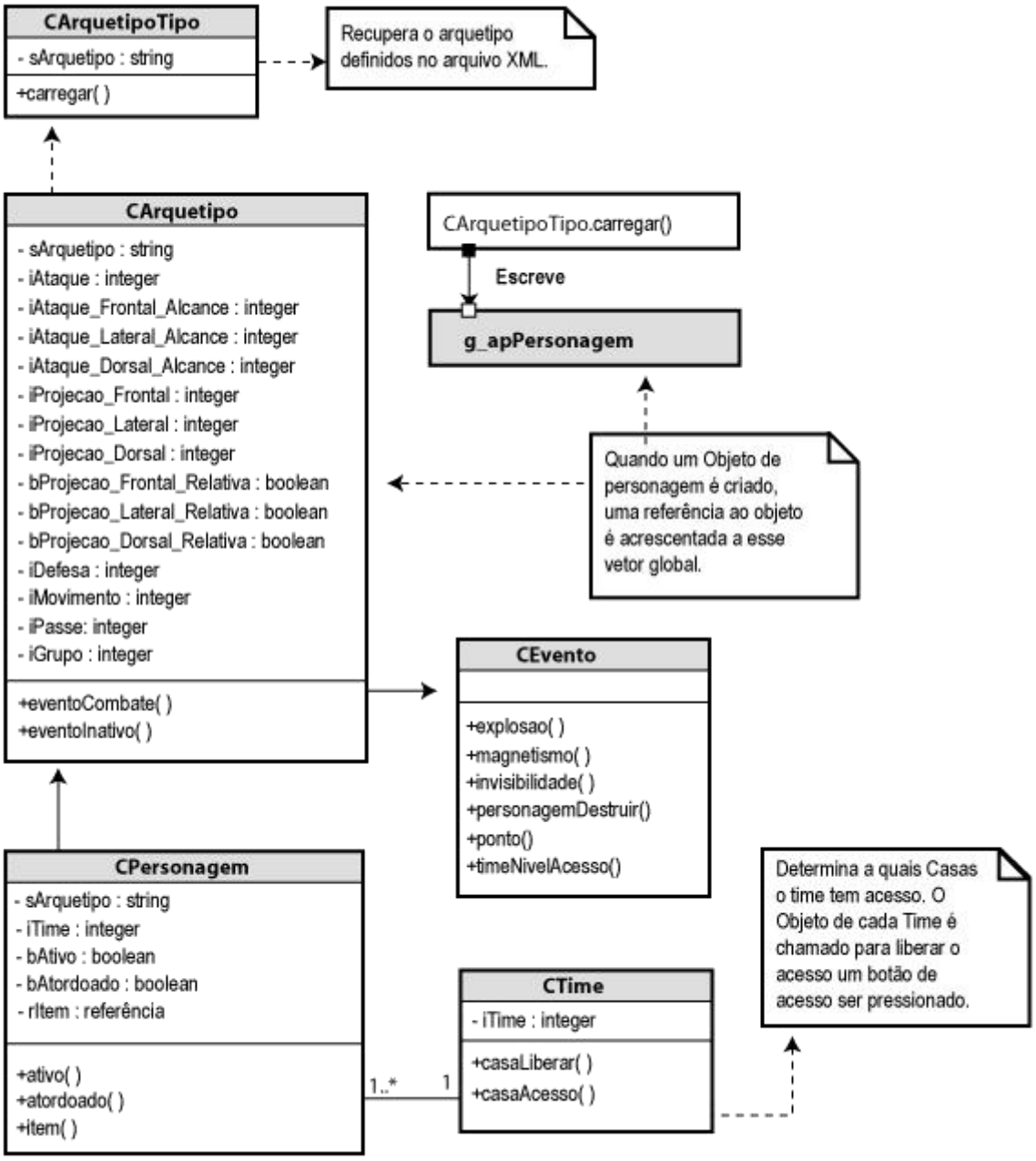
Atributo	Tipo	Funções
iAtaque	Integer	O poder de ataque do personagem, usado na avaliação do resultado de um combate.
iAtaque_Frontal_Alcance	Integer	A que distância podem ser realizados os ataques frontais.
iAtaque_Lateral_Alcance	Integer	A que distância podem ser realizados os ataques laterais.
iAtaque_Dorsal_Alcance	Integer	A que distância podem ser realizados os ataques dorsais.
iProjecao_Frontal	Integer	A projeção de um ataque frontal do personagem.
bProjecao_Frontal_Relativa	Boolean	Se a projeção frontal é relativa (*true*) ou absoluta (*false*).
iProjecao_Lateral	Integer	O poder de ataque lateral do personagem.
bProjecao_Lateral_Relativa	Boolean	Se a projeção lateral é relativa (*true*) ou absoluta (*false*).
iProjecao_Dorsal	Integer	O poder de ataque dorsal do personagem.
bProjecao_Dorsal_Relativa	Boolean	Se a projeção dorsal é relativa (*true*) ou absoluta (*false*).
iDefesa	Integer	A defesa do personagem.
iMovimento	Integer	A quantidade de Casas que o personagem

		pode se movimentar seguidamente.
bAtordoado	Boolean	Se o personagem está ou não Atordoado.
bAtivo	Boolean	Se o personagem está ou não Ativo para ser comandado pelo jogador.
iTime	Integer	O time ao qual o personagem pertence.
rItem	Referência	O Item que o personagem está carregando. NULL se não houver item algum.
iPasse	Integer	A quantas Casas de distância o personagem pode Passar um Item para outro.
iGrupo	Integer	Quantos outros personagens aliados o personagem pode juntar em um Arraste.

Por que criar tantos atributos de ataque? Não seria melhor simplificar, estabelecendo que todos os modos de ataque tem o mesmo alcance? Talvez, mas então perderíamos possibilidades interessantes para novos personagens. O Capitão Guincho da Etapa 3, por exemplo, tem, um alcance de ataque dorsal 3 e uma projeção dorsal 1. É um personagem que usa as Casas mortais de forma inversa, tentando atrair e manipular os inimigos.

Ou talvez fosse melhor abandonar o atributo iAtaque, e avaliar a vitória de um combate usando os atributos iProjecao_<direção> de acordo com o tipo de ataque. Não seria, se iAtaque é a forma de avaliar uma vitória, podemos ter personagens que projetam pouco mas sempre ganham um Combate, adicionando um elemento estratégico extra.

Os tipos de personagens, denominados *Arquétipos*, são configurados no arquivo *arquetipo.xml*. Podemos modelar os arquétipos e os atributos dos personagens da seguinte forma:



A classe CPersonagem sempre consulta o objeto da classe CTime para avaliar em quais Casas ele terá acesso. Os objetos (instâncias) de CTime sempre monitora e responde se os membros dos times têm acesso a determinado nível ou não. O evento timeNivelAcesso() sempre avisa esses objetos das mudanças de acesso.

Novamente, expandimos nossa classe CEvento para acomodar o método de explosão, ainda se implementação na Etapa 1. De forma similar, muitos atributos da classe CArquetipo também só serão implementados nos algoritmos das Etapas 2 e 3. Como sempre, o que estamos fazendo é lançar as bases de compatibilidade futura.

Repare agora como tudo faz sentido na modelagem de nossos personagens de qualquer Etapa. As estatísticas para os personagens das Etapas 2 e 3 não precisam ser exatas agora, mas você precisa de parâmetros quaisquer para avaliar a expansibilidade da modelagem:

Personagem	Atributos	Observações
Corredor	▪ iAtaque: *1*	Possui um *Passe* mais longo

	▪ iAtaque_Frontal_Alcance: *1* ▪ iAtaque_Lateral_Alcance: *1* ▪ iAtaque_Dorsal_Alcance: *0 (sem ataque)* ▪ iProjecao_Frontal: *1* ▪ bProjecao_Frontal_Relativa: *true* ▪ iProjecao_Lateral: *1* ▪ bProjecao_Lateral_Relativa: *false* ▪ iProjecao_Dorsal: *0* ▪ bProjecao_Dorsal_Relativa: *false* ▪ iDefesa: *1* ▪ iMovimento: *5* ▪ iPasse: *2* ▪ iGrupo: *1*	que os outros da Etapa 1.
Espancador	▪ iAtaque: *4* ▪ iAtaque_Frontal_Alcance: *1* ▪ iAtaque_Lateral_Alcance: *1* ▪ iAtaque_Dorsal_Alcance: *0* ▪ iProjecao_Frontal: *4* ▪ bProjecao_Frontal_Relativa: *true* ▪ iProjecao_Lateral: *1* ▪ bProjecao_Lateral_Relativa: *false* ▪ iProjecao_Dorsal: *0* ▪ bProjecao_Dorsal_Relativa: *false* ▪ iDefesa: *1* ▪ iMovimento: *2* ▪ iPasse: *1* ▪ iGrupo: *1*	---
Muralha	▪ iAtaque: *1* ▪ iAtaque_Frontal_Alcance: *1* ▪ iAtaque_Lateral_Alcance: *1* ▪ iAtaque_Dorsal_Alcance: *0* ▪ iProjecao_Frontal: *1* ▪ bProjecao_Frontal_Relativa: *true* ▪ iProjecao_Lateral: *1* ▪ bProjecao_Lateral_Relativa: *false* ▪ iProjecao_Dorsal: *0* ▪ bProjecao_Dorsal_Relativa: *false* ▪ iDefesa: *4* ▪ iMovimento: *2* ▪ iPasse: *1* ▪ iGrupo: *1*	---
Demolidor	▪ iAtaque: *2* ▪ iAtaque_Frontal_Alcance: *1* ▪ iAtaque_Lateral_Alcance: *0* ▪ iAtaque_Dorsal_Alcance: *0* ▪ iProjecao_Frontal: *0* ▪ bProjecao_Frontal_Relativa: *false* ▪ iProjecao_Lateral: *0* ▪ bProjecao_Lateral_Relativa: *false* ▪ iProjecao_Dorsal: *0* ▪ bProjecao_Dorsal_Relativa: *false* ▪ iDefesa: *2* ▪ iMovimento: *3* ▪ iPasse: *1* ▪ iGrupo: *1*	Quando ataca (método *eventoCombate*), chama o evento *explosao()* na Casa do adversário.

Espião	▪ iAtaque: *2* ▪ iAtaque_Frontal_Alcance: *1* ▪ iAtaque_Lateral_Alcance: *1* ▪ iAtaque_Dorsal_Alcance: *0* ▪ iProjecao_Frontal: *2* ▪ bProjecao_Frontal_Relativa: *true* ▪ iProjecao_Lateral: *1* ▪ bProjecao_Lateral_Relativa: *false* ▪ iProjecao_Dorsal: *0* ▪ bProjecao_Dorsal_Relativa: *false* ▪ iDefesa: *1* ▪ iMovimento: *3* ▪ iPasse: *1* ▪ iGrupo: *1*	Permanece com o evento *invisibilidade()* ligado enquanto não está disponível par aser comandado pelo jogador (método *eventoInativo*).
Rebatedor	▪ iAtaque: *4* ▪ iAtaque_Frontal_Alcance: *1* ▪ iAtaque_Lateral_Alcance: *1* ▪ iAtaque_Dorsal_Alcance: *0* ▪ iProjecao_Frontal: *1* ▪ bProjecao_Frontal_Relativa: *false* ▪ iProjecao_Lateral: *4* ▪ bProjecao_Lateral_Relativa: *true* ▪ iProjecao_Dorsal: *0* ▪ bProjecao_Dorsal_Relativa: *false* ▪ iDefesa: *1* ▪ iMovimento: *2* ▪ iPasse: *2* ▪ iGrupo: *1*	Especialista em ataques laterais. Seu ataque frontal sempre projeta uma casa. Passes longos.
Aríete	▪ iAtaque: *5* ▪ iAtaque_Frontal_Alcance: *1* ▪ iAtaque_Lateral_Alcance: *0* ▪ iAtaque_Dorsal_Alcance: *0* ▪ iProjecao_Frontal: *2* ▪ bProjecao_Frontal_Relativa: *false* ▪ iProjecao_Lateral: *0* ▪ bProjecao_Lateral_Relativa: *false* ▪ iProjecao_Dorsal: *0* ▪ bProjecao_Dorsal_Relativa: *false* ▪ iDefesa: *1* ▪ iMovimento: *1* ▪ iPasse: *1* ▪ iGrupo: *2*	Ataques frontais com projeção fixa e Arrastes poderosos, mas sem ataques laterais.
Canhoneiro	▪ iAtaque: *4* ▪ iAtaque_Frontal_Alcance: *4* ▪ iAtaque_Lateral_Alcance: *0* ▪ iAtaque_Dorsal_Alcance: *0* ▪ iProjecao_Frontal: *4* ▪ bProjecao_Frontal_Relativa: *true* ▪ iProjecao_Lateral: *0* ▪ bProjecao_Lateral_Relativa: *false* ▪ iProjecao_Dorsal: *0* ▪ bProjecao_Dorsal_Relativa: *false* ▪ iDefesa: *1* ▪ iMovimento: *1* ▪ iPasse: *1* ▪ iGrupo: *0*	Ataques frontais poderosos e à distância, mas sem ataques laterais. Não faz Arrastes.

Capitão Guincho	▪ iAtaque: *2* ▪ iAtaque_Frontal_Alcance: *1* ▪ iAtaque_Lateral_Alcance: *1* ▪ iAtaque_Dorsal_Alcance: *4* ▪ iProjecao_Frontal: *3* ▪ bProjecao_Frontal_Relativa: *true* ▪ iProjecao_Lateral: *1* ▪ bProjecao_Lateral_Relativa: *false* ▪ iProjecao_Dorsal: *2* ▪ bProjecao_Dorsal_Relativa: *false* ▪ iDefesa: *3* ▪ iMovimento: *2* ▪ iPasse: *1* ▪ iGrupo: *1*	Ataques dorsais de projeção fixa.

Finalmente, vamos modelar nossos arquétipos na XML. A seguir, veremos como colocar isso tudo junto, usando fluxogramas e objetos de controle.

arquetipo.xml

<arquetipo valor= "*nome do arquétipo***" >**

<ataque valor= "*1***" />**

<direcao valor= "*frontal***">**

Há três tags <direcao> com os seguintes atributos: frontal, lateral e dorsal. Para cada tag há configurações diferentes que determinam o funcionamento dos sentidos de ataque. Caso o arquétipo não tenha uma tag, o game avalia o valor faltante como zero (o personagem não pode executar um ataque naquela direção).

<alcance valor= "*1***" />**

<projecao valor= "*1***" relativa= "***true***" />**

</direcao>

<defesa valor= "*1***" />**

<movimento valor= "*5***" />**

<grupo valor= "*1***" />**

<passe valor= "*1***" />**

<evento tipo= "*combate***" />**

O atributo "tipo" será lido pelo jogo para checar quando o evento deve ser tratado. Cada tag representa um dos eventos disponíveis para o Personagem.

<metodo comando= "*explosao***" valor= "***true***" />**
Cada tag representa um dos eventos disponíveis para o Personagem. Como os Materiais, caso algum novo evento de Etapas posteriores seja adicionado e as tags de um Item não sejam atualizadas não haverá problema, porque a classe CArquetipo só executa tags encontradas e de valor *true*.

<metodo comando= "*invisibilidade***" valor= "***false***" />**
. . .

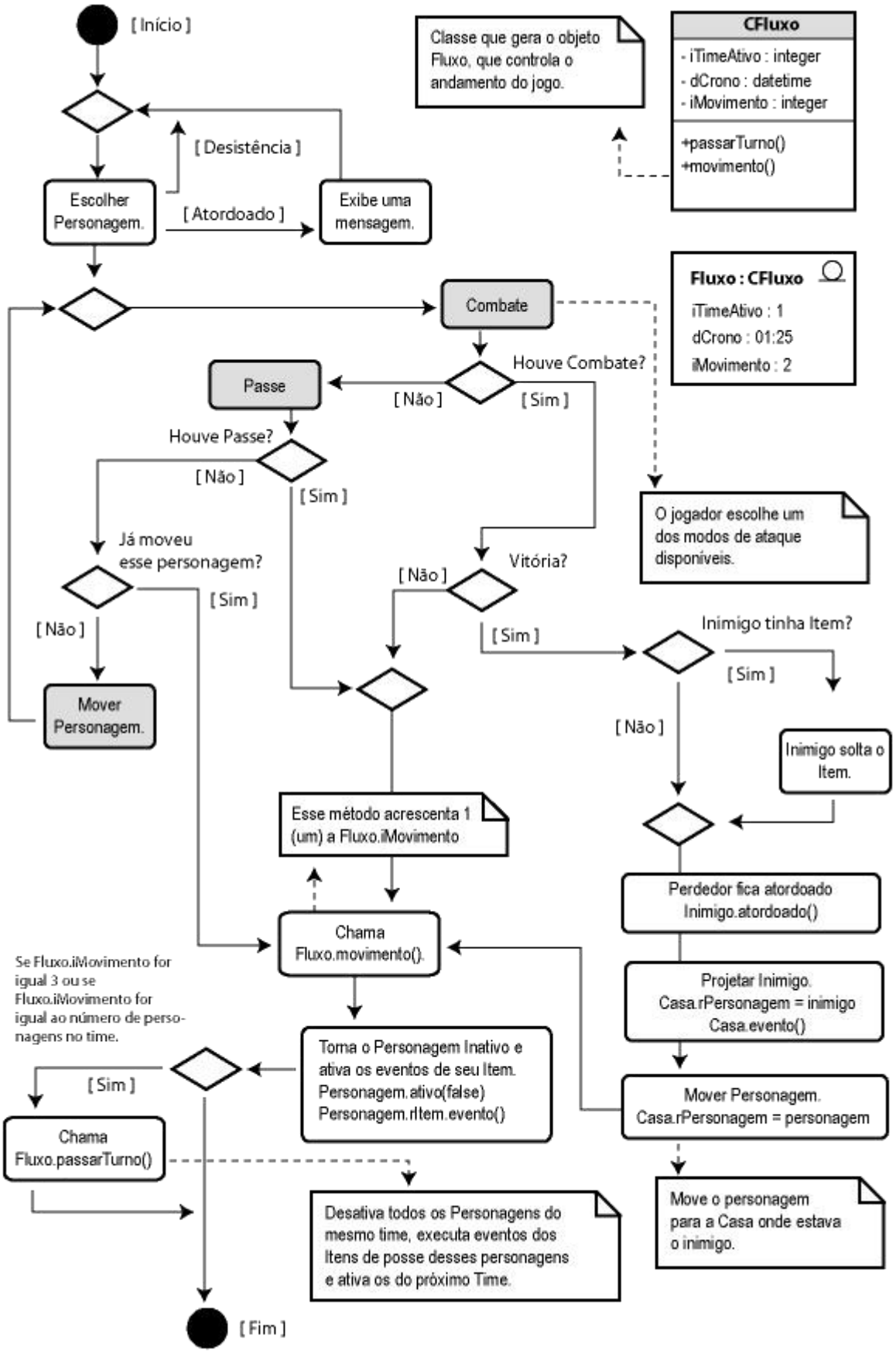
```
        </evento>

</ arquetipo >
```

4.2.5 | Fluxo

Vamos tratar aqui de como efetivamente colocar os Objetos em interação, suas relações e funcionamento em cadeia. Ou seja, como juntar tudo em um jogo. Faremos isso em fluxogramas, e à medida que avançamos, veremos que novas classes, objetos e métodos para as classes já existentes vão surgindo.

O primeiro fluxograma é o que trata da atividade do jogador: selecionar um personagem, movê-lo, atacar. Para controlar o ritmo do jogo – qual o Time ativo, quantos movimentos faltam –criamos a classe CFluxo, instanciada no objeto Fluxo. Como pretendemos que na Etapa 3 possamos limitar jogos pelo tempo decorrido, também vamos adicionar um cronômetro.

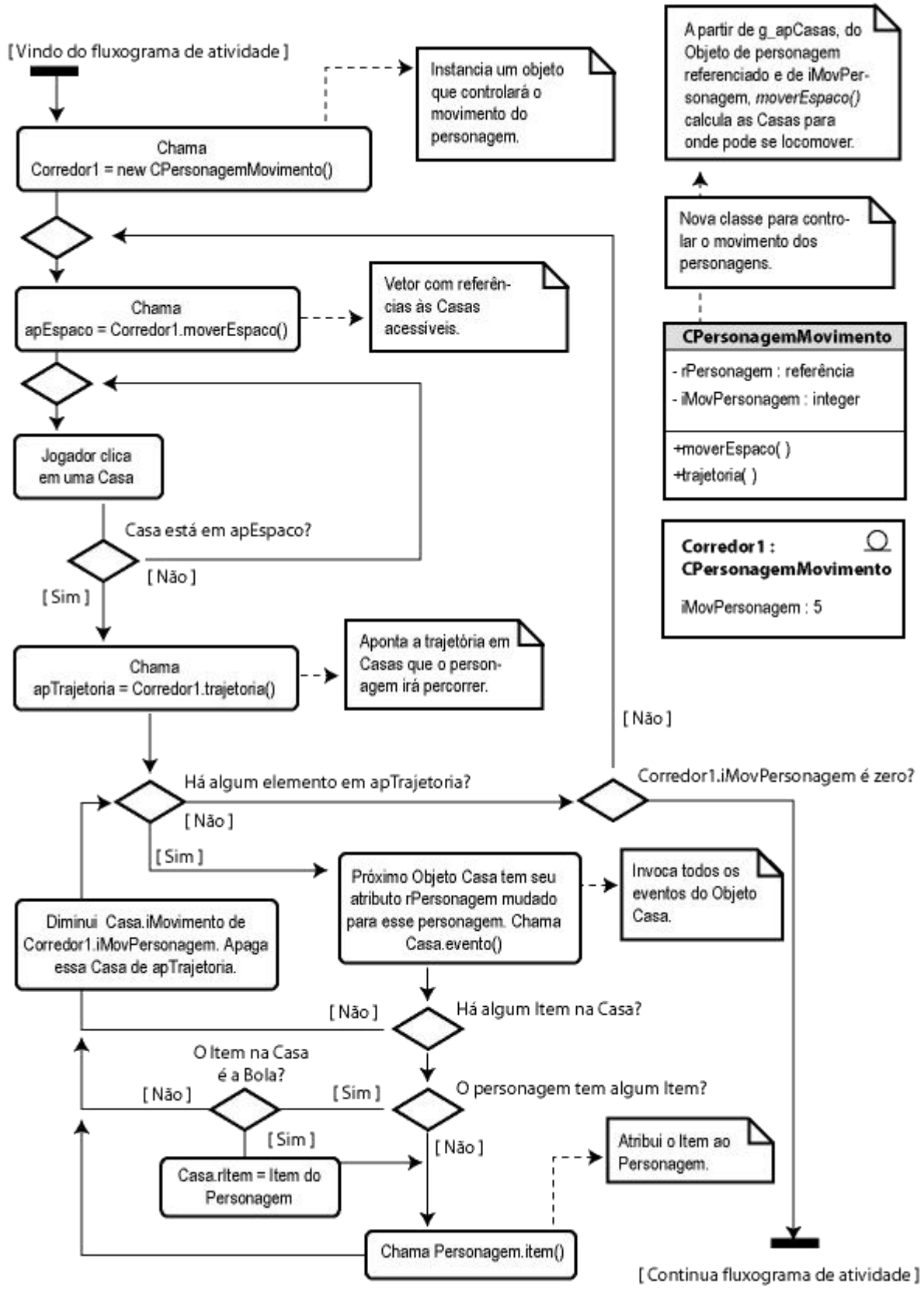
Nota: as bolinhas pretas significam um comportamento cíclico: quando o fluxo chega na bolinha “Fim”, ele na verdade volta à bolinha “Início” e o ciclo se reinicia.



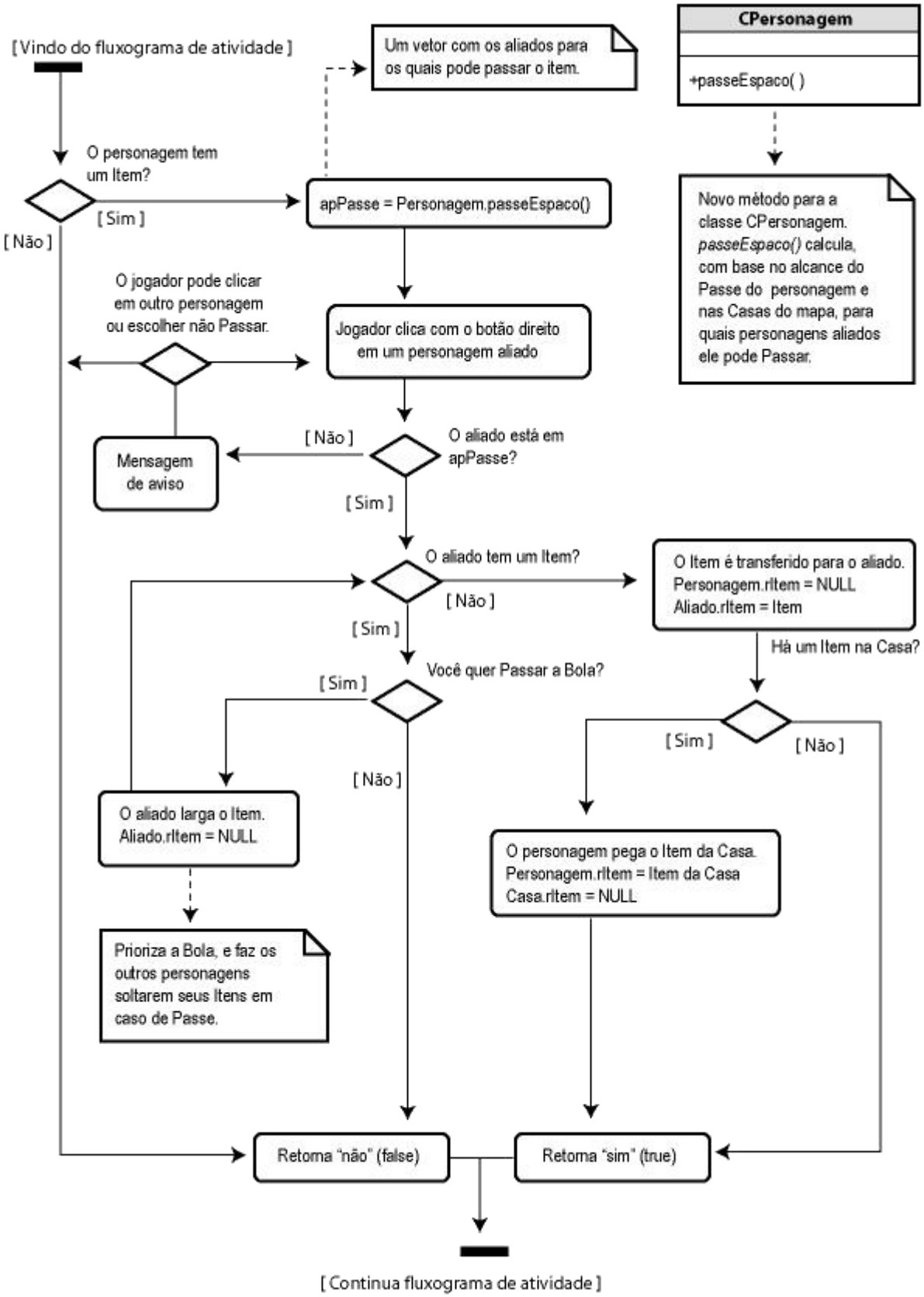
Vamos ver com mais detalhes as fases em cinza. Primeiro, temos o tratamento da movimentação dos personagens. Antes de montarmos o fluxograma, ajuda estabelecer uma matriz de interação personagem / propriedades das Casas:

Atributo da Casa	Personagem
iNivel	Os personagens só podem entrar se seu Time tiver acesso ao nível.
iTime	Se o personagem entrar carregando o Item Bola, seu Time ganha marca um ponto.
rPersonagem	Se houver outro personagem referenciado pela Casa, não permite a entrada. O combate é tratado adiante.
rItem	Se houver um Item, o personagem o pega. Personagens que já tenham um Item não podem pegar outro.
bPasse	Avalia se personagens que Passam Itens à distância (como o Corredor) podem “jogá-los” através dessa Casa.
bMortal	Se true, somente personagens atordoados podem acessa a Casa. O personagem é destruído.
iMovimento	Quanto de movimento o personagem terá que gastar para entrar na Casa.
iEventoContagem	Cada vez que um personagem qualquer entrar numa Casa cujo Material invoque o método *eventoContagem()*, esse número é acrescido em um.

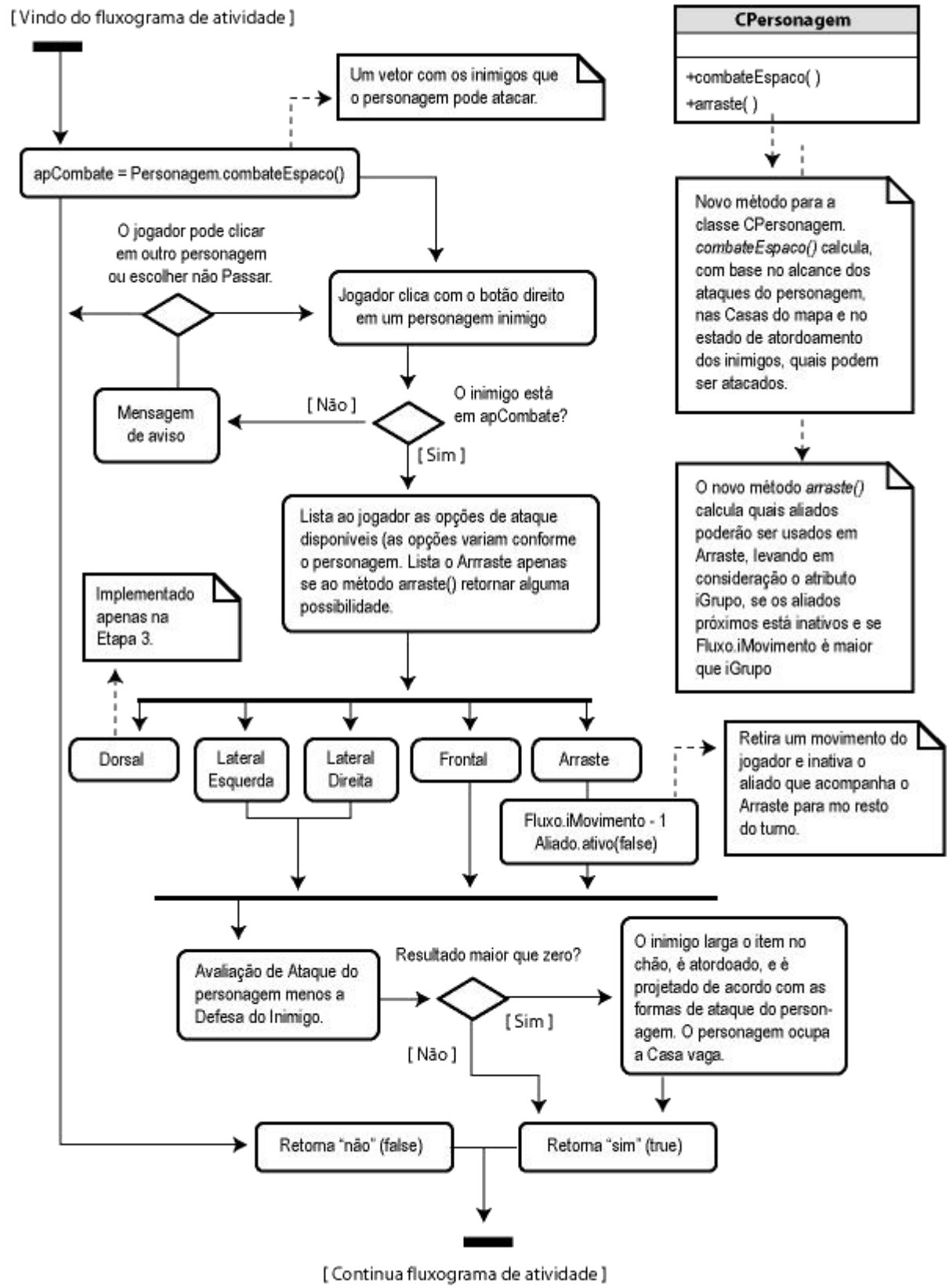
Sabemos também que os personagens não podem se deslocar em diagonal. Com essas informações, projetamos agora nosso fluxograma de movimentação:



Com a classe CPersonagemMovimento criada, podemos utilizá-la em nosso fluxograma que avalia se o personagem pode ou não fazer um Passe:



Por fim, uma visão geral do fluxograma de combate:



Repare que no final não estamos falando que um inimigo derrotado passa seu Item para o vencedor, e sim que ele larga o Item na Casa, é projetado, e só então o personagem ocupa a Casa e, naturalmente pega o Item dela. Essa diferença é importante, pois pretendemos implementar ataques à distância no futuro, e os Itens não poderão ser simplesmente transferidos.

5 | Conclusão

Nesse capítulo, utilizamos um exemplo simples para demonstrar como você deve planejar a arquitetura do seu game. Demonstramos como pensar sua estrutura a partir de análises com casos de uso e diagramas de funcionamento, que podem revelar problemas insuspeitos muito antes de você começar a escrever os códigos. Demonstramos também como tornar sua base expansível para eventuais adições e *mods*, e como componentizá-la para reusar códigos e diminuir custos com manutenção e correção de *bugs*. E porque nos preocupamos com prazos e imprevistos que possam afetar a qualidade de nosso produto, colocamos tudo dentro de um planejamento e execução em Etapas seqüenciais.

Mesmo que você não queira projetar diagramas e fluxogramas como os aqui descritos, seja porque seu projeto é bem pequeno, seja porque prefere ir logo ao código, faça pelo menos os casos de uso para o seu game. É importantíssimo ter uma idéia clara, desde o início, do que seu jogador será capaz de ver, comandar e selecionar.

Por fim, ressaltamos que análises e projetos expansíveis e componentizados irão economizar muito tempo de programação no futuro. Temos consciência que eles demandam mais trabalho no começo, e nem sempre os benefícios são visíveis de imediato. Mas acredite: a não ser que você esteja em um projeto bem pequeno sem muitas perspectivas, ou com o prazo estourado, o esforço na estruturação e padronização vale a pena.