Inteligência Artificial para Jogos

Aula 05 - Máquina de Estados Finitos

Apresentação da Aula

Olá, programador(a)! Bem-vindo(a) à aula 05! Você avançou bastante com as aulas anteriores. O jogador que você desenvolveu está ficando bem preparado. Ele já corre de um lado para o outro, sabe desviar de obstáculos, pode perseguir um adversário, sabe chutar... sabe até fazer besteira. Ops! 😁 Pois é, como dito na primeira aula de IA para Jogos, um dos objetivos da IA é simular uma situação de forma realista. No seu caso, você ensinou o jogador a "chutar pra fora". Fica mais parecido com o que acontece de fato. Muito bem!

Só que nos exercícios e atividades que você fez até agora, o jogador tinha apenas um comportamento: ou ele ficava correndo de um lado para o outro ou era um cobrador de pênaltis. Infelizmente, isso é limitado demais para um jogo de futebol. O jogador precisa ser mais polivalente, adotando diversos comportamentos em diferentes situações.

Na verdade, na aula anterior, você já começou a implementar um mecanismo simples de mudança de comportamento. Se você prestar atenção, o cobrador de pênaltis possui dois estados. Inicialmente, ele vai em direção da bola (1º estado) e depois, quando chega próximo dela (quando esse evento ocorre), chuta em um dos cantos do gol e a deixa de ir em sua direção (2º estado). Se não tivesse separado o comportamento do jogador em dois estados, ele continuaria indo atrás da bola mesmo após tê-la chutado. Isso não quebra o realismo se o pênalti for durante o jogo, mas se o pênalti for na decisão da partida após o tempo regulamentar, não faz muito sentido. Veja, então, a importância da organização de um comportamento maior (cobrar um pênalti) em estados (correr e, depois, chutar), até mesmo para comportamentos aparentemente simples.

Certamente, há jogos que não precisam de estados em seus personagens. Se você for implementar um jogo tipo Pong, não há necessidade de estados. A barra do jogo só precisa ir na direção da bola... o tempo todo! Esses jogos mais simples não requerem estratégias de IA. Porém, no momento em que os comportamentos começam a ser mais complexos, você precisa estruturá-los melhor. A organização

por meio de “estados” é um mecanismo simples, mas bastante eficiente para definição de personagens virtuais. Nessa aula, você irá elaborar e implementar comportamentos baseado em estados.

Pronto para começar?

Objetivos

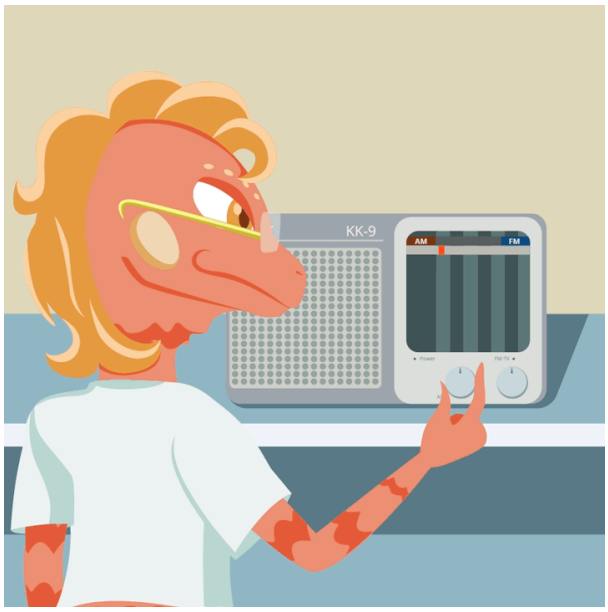
Entender o que é uma Máquina de Estados Finitos e suas diferentes formas de implementação no Unity;

Compreender como representar uma Máquina de Estados Finitos em forma de diagrama;

Aplicar uma Máquina de Estados Finitos para modelar o comportamento de um personagem de um jogo.

1. Máquina de Estados Finitos

A forma de organização de comportamentos através de estados não é algo exclusivo da área de jogos. Aliás, é um mecanismo que já existe bem antes dos jogos que você conhece atualmente. Esse mecanismo é encontrado até no radinho de pilha usado para escutar os jogos da copa de 50. Quando seu avô alternava as sintonias do rádio entre AM e FM, ele estava alterando o estado do funcionamento (ou comportamento) do aparelho. Como o radinho de pilha, se você prestar atenção, o comportamento de muitos dispositivos pode ser modelado através de estados; do funcionamento de um relógio-alarme à refrigeração de uma geladeira.



Em geral, o comportamento desses dispositivos é modelado através da técnica chamada de **Máquina de Estados Finitos** ou **MEF** (Esse conceito foi visto por você na aula 01). O termo finito advém do fato que o estado da máquina deve pertencer a um conjunto finito de possíveis estados. Além deles uma **MEF** é definida também pelas transições de um estado a outro, normalmente associadas a eventos que podem ocorrer. Por exemplo, o radinho de pilha do seu avô passa do estado AM para o estado FM quando o evento “botão de FM pressionado” ocorrer. Por fim, há também a possibilidade de se definir as ações a serem executadas quando a **MEF** entrar ou sair de um estado.

Para facilitar a representação de comportamentos baseados em estados, existe uma linguagem visual para descrever uma **MEF**, ou seja, ela pode ser representada através de um diagrama, como o da Figura 01.

Figura 01 - Diagrama de uma Máquina de Estados Finitos de um personagem de jogo

A Figura 01 é um exemplo de diagrama da **MEF** de um personagem de um jogo que possui três possíveis estados: o de patrulha, o de perseguição e o de ataque. A bola laranja do diagrama é para indicar o estado inicial do personagem: ele começa patrulhando uma determinada área. O comportamento do personagem nesse estado pode ser, por exemplo, o de visitar uma sequência pré-definida de pontos do cenário. Durante a patrulha, no momento em que um inimigo estiver à sua vista (o que caracteriza um evento), há uma transição desse estado e ele passa do estado patrulha para o de perseguição. Nesse estado, seu comportamento pode ser, por exemplo, ir na direção do inimigo. Estando nele, se o inimigo estiver perto o suficiente para poder atacá-lo, haverá uma nova transição, passando do de perseguição ao de ataque. Enquanto ele estiver nele, seu comportamento pode ser, por exemplo, o de dar golpes no inimigo. Se o inimigo morrer em um dos golpes, uma nova transição de estado é efetuada e o personagem volta à sua patrulha. Ainda no estado de ataque, se o inimigo estiver longe o suficiente para que os golpes não atinjam mais o personagem, então uma transição é efetuada e o personagem volta para o de perseguição.

Olhando o diagrama da Figura 01, é possível compreender rapidamente todo o comportamento descrito no parágrafo anterior, não é mesmo? É por isso que o diagrama de estados é bastante utilizado para representar uma **MEF**. Além de uma

organização clara do comportamento do personagem, rapidamente você consegue compreender ou fazer com que outros desenvolvedores compreendam como o personagem vai atuar.

O Unity possui uma ferramenta visual para facilitar a composição de animações dos objetos do jogo baseado em uma **MEF**. Pode-se utilizar essa ferramenta, porém há outras formas de se implementá-la. Veja, na próxima seção, algumas das possíveis estratégias.

2. Implementando uma MEF

Como mencionado no início da aula, você começou a implementar um mecanismo simples de mudança de comportamento. O cobrador de pênalti possui o estado de ir em direção à bola e o de chute (que não segue mais a bola). Na implementação realizada (na aula anterior), essa mudança de estado só foi possível porque você colocou uma "variável de controle" em um dos atributos do jogador. A variável, booleana, era usada no método *Update*() para saber quando o jogador devia ir em direção à bola ou não.

Essa forma de implementação é muito limitada porque uma variável booleana pode assumir apenas dois valores e, portanto, o jogador pode assumir apenas dois possíveis estados. Se você quiser que ele tenha um comportamento mais complexo, precisa de uma outra forma de implementar a **MEF**.

De uma forma geral, as variações referem-se principalmente a dois itens. O primeiro é **como definir o comportamento dos diferentes estados**, e o segundo é **como os eventos são avaliados durante a execução de um determinado estado**. Usando o Unity, você pode ainda utilizar o mecanismo de animação interno dele, chamado *Mecanim*, que nada mais é que uma ferramenta visual para implementar uma **MEF**. Apesar de estar direcionada à mudança de *sprites* ou de animações de modelos 3D, a *Mecanim* pode ser facilmente utilizada para modelar comportamentos simples de personagens. Porém, à medida que a complexidade do comportamento aumenta, outros mecanismos são necessários.

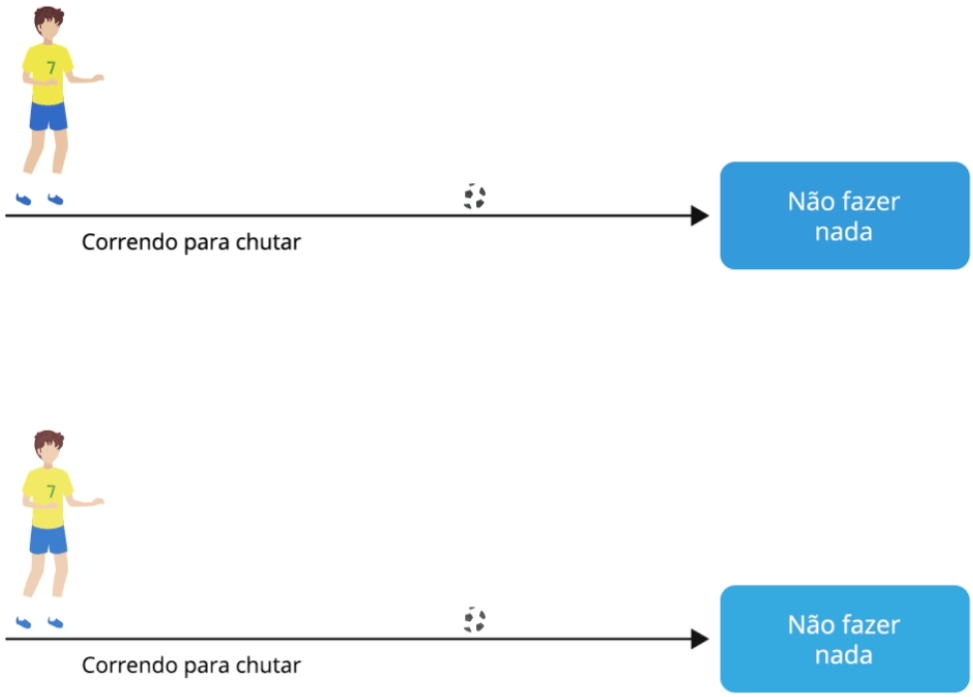
Nessa aula, serão apresentadas duas estratégias de implementação de uma **MEF** no Unity, ambas usando recursos de programação. De qualquer forma, vale salientar que a *Mecanim* é mais adequada para modelar comportamentos simples e traz a vantagem de usar uma ferramenta visual para especificar os estados. Essas formas de implementação não usam a *Mecanim* e são, portanto, mais flexíveis, mas perdem a organização visual. A primeira é implementada através de blocos de comandos e é voltada principalmente para comportamentos bem simples. A segunda faz uso de um padrão de projeto específico para mudança de estados, *State Pattern*. Essa estratégia é mais adequada em comportamentos com um número maior de estados e quando se deseja ter mais flexibilidade no reuso deles.

3. MEF usando blocos de comando

A essência de um comportamento baseado em estado é fazer com que, a cada atualização do personagem, as ações que ele executa sejam referentes ao que ele se encontra. Ou seja, **a ação que está sendo executada é baseada no estado atual**. Se o personagem altera seu estado, no laço de atualização do jogo, uma ação diferente passa a ser executada. Além disso, muitas vezes é importante você ter ações associadas ao momento em que o personagem entra em um estado e ações associadas ao momento que ele sai. Por exemplo, na cobrança de pênalti, você poderia ter os seguintes estados: “Correndo para chutar” e “Não fazer nada”. Durante o “Correndo para chutar”, a ação do jogador seria apenas a de ir na direção da bola. O chute seria a ação executada quando o jogador saísse desse estado, indo para o de “Não fazer nada”.

A Figura 02 ilustra essa descrição, mostrando que a ação “chute” é realizada no momento da transição de um estado para o outro.

Figura 02 - Diagrama de estados do cobrador de pênaltis



Uma estratégia para implementar essa ideia é reservar um atributo do personagem para armazenar o estado em que ele se encontra e, a cada atualização dele, executar um bloco de comandos diferente para cada respectivo estado. Se o personagem tiver apenas dois possíveis estados, o atributo pode ser uma variável booleana, tal como feito na aula anterior. Porém, se houver mais de dois possíveis estados para o personagem, você precisará então de um atributo que possa assumir mais valores.

Um mecanismo simples, mas bastante utilizado em jogos, é usar uma enumeração para delimitar os possíveis estados que o atributo pode assumir. Durante a atualização do personagem, um bloco de comandos é associado a cada possível estado, ou através de um *switch* ou de um *if...else*. O Código 01 mostra como essa abordagem pode ser implementada usando três possíveis estados.

Código 01 – Exemplo de código implementando uma MEF usando blocos de códigos

```
public class MyBehavior : MonoBehaviour {

    enum State {
        State_A,
        State_B,
        State_C
    };

    State state;

    void Start () {
        state = State.State_A;
    }

    void ChangeState(State newState) {
        switch (state) {
            case State.State_A:
            // ação a ser realizada ao entrar no estado A
            break;
            case State.State_B:
            // ação a ser realizada ao entrar no estado B
            break;
            case State.State_C:
            // ação a ser realizada ao entrar no estado C
            break;
        }
        state = newState;
        switch (state) {
            case State.State_A:
            // ação a ser realizada ao entrar no estado A
            break;
            case State.State_B:
            // ação a ser realizada ao entrar no estado B
            break;
            case State.State_C:
            // ação a ser realizada ao entrar no estado C
            break;
        }
    }

    void FixedUpdate () {
        switch (state) {
            case State.State_A:
            // ação a ser realizada durante o estado A
            break;
            case State.State_B:
            // ação a ser realizada durante o estado B
            break;
            case State.State_C:
            // ação a ser realizada durante o estado C
            break;
```

```
52        }
53      }
54    }
```

O que faltou apresentar no código 01 é em que momento o método *ChangeState*() é chamado, ou seja, quando os eventos geram as transições de estado. Há basicamente duas possibilidades: ou **se registra um evento**, por exemplo um associado ao *collider* do personagem ou relacionado a uma mensagem que ele possa receber de outro objeto, ou **se testa se o evento ocorreu durante a atualização do personagem** (*FixedUpdate*()).

Serão apresentados mais detalhes sobre o gerenciamento de eventos na última seção dessa aula. De qualquer forma, você pode ver aqui como a captura do evento pode ocorrer. O Código 02 ilustra duas formas de captura de evento. No caso, quando o personagem colidir com algum objeto que possua um *collider* que é um gatilho (*trigger*), esse evento será capturado e o método *OnTriggerEnter2D*() é chamado. Nesse momento, se o personagem estiver no estado **A**, ele é alterado para o **B**. Por sua vez, durante a atualização do personagem no estado **B**, se uma determinada condição for verdadeira, ele passa para o **A**.

Código 02 – Captura do evento de transição de estado seja dentro do laço de atualização ou "escutando" o evento

```
public class MyBehavior : MonoBehaviour {

    // ...
    void FixedUpdate () {
        switch (state) {
            case State.State_A:
            // ...
            break;
            case State.State_B:
            if (/* condição */)
                ChangeState(State.State_A);
            break;
            case State.State_C:
            // ...
            break;
        }
    }

    void OnTriggerEnter2D (Collider2D coll) {
        if (state == State.State_A) {
            ChangeState(State.State_B);
        }
    }
}

```

Para ficar mais claro, aplique o código genérico descrito anteriormente no contexto do jogo. Para começar, os estados que você tem são o de **correr atrás da bola**, que se chama de *SeekBallState* e o de **não fazer nada** (pois o jogador já chutou a bola e não precisa mais sair correndo atrás dela), que se chama de *IdleState*.

A classe *IdleState* implementa um estado que "não faz nada". Parece estranho implementar algo para não fazer nada, não é? Mas os desenvolvedores de jogos aprenderam com os mestres zen-budistas que não fazer nada, às vezes, é essencial. 😁

Muitos jogos possuem esse estado para seus personagens. Por exemplo, imagine um jogo de plataforma em que os personagens estão dispostos em várias partes do cenário. Porém, eles só vão atuar quando o jogador estiver à sua vista.

Enquanto ele não se aproximar do personagem, em que estado ele vai estar? Em meditação profunda, como os mestres ensinaram. Da mesma forma, você vai colocar o jogador em estado de *idle* após sair do *SeekState*.

No vídeo 01 abaixo, você encontrará uma situação similar à que foi descrita.

Vídeo 01 - Cena do jogo Dark Souls

Fonte: VITO VENUE. **Dark souls defeating a black knight**. 2011. Disponível em: https://www.youtube.com/watch?v=5VnK6SY3ZTU. **Acesso em:** 04 abr. 2018.

Por enquanto, não se preocupe, pois em breve, você incrementará ainda mais o comportamento do jogador, definindo outros estados, por exemplo, um será para o jogador "sair pra galera" depois de fazer o gol. No momento, dois já são suficientes.

Para os dois estados (*SeekBallState* e *IdleState*), é necessário adaptar o método de atualização e o de captura de colisões existentes, além de criar o *ChangeState*(). Para organizar melhor o novo código, passe o código de "chute da bola" implementado na aula anterior para um método próprio chamado *ShotBall*(), encontrado no Código 03 abaixo.

Código 03 – Versão do *script* do cobrador de pênalti baseado em estado usando blocos de comando

```
public class SoccerPlayer : MonoBehaviour {

    enum State {
        SeekBall,
        Idle
    };
    State state;

    // ... parte omitida porque não houve alteração

    public void Reset() {
        state = State.SeekBall;
        steer.Reset();
    }

    void FixedUpdate () {
        switch (state) {
            case State.Idle:
                break;
            case State.SeekBall:
                GameObject ball = GameManager.instance.ball;
                target = ball.transform.position;
                break;
        }
        Vector2 steering = steer.Arrival(target);
        steer.ApplySteering(steering);
    }

    void ChangeState(State newState) {
        // ações executadas na saída de um estado
        switch (state) {
            case State.Idle: // não faz nada
                break;
            case State.SeekBall: // chuta a bola quando for sair do estado
                ShotBall();
                break;
        }
        state = newState;

        // ações executadas na entrada de um estado
        // não fazem nada, mas deixamos o código aqui para ilustrar
        // como a entrada de estado poderia ser implementada
        switch (state) {
            case State.Idle:
                break;
            case State.SeekBall:
                break;
        }
    }

    void ShotBall() {
```

```
52          GameObject goal = GameManager.instance.goal;
53          GameObject ball = GameManager.instance.ball;
54          Bounds bounds = goal.GetComponent<BoxCollider2D>().bounds;
55          Vector2[] sides = {
56              new Vector2(bounds.center.x, bounds.max.y), // à direita
57              new Vector2(bounds.center.x, bounds.min.y)  // à esquerda
58          };
59          int sideId = Random.value < 0.5? 0 : 1;
60          Vector2 side = sides[sideId];
61          Rigidbody2D ballBody = ball.GetComponent<Rigidbody2D>();
62          ballBody.AddForce((side - ballBody.position) * kickForce);
63          OnShot(); // dispara o evento
64      }
65
66      void OnTriggerEnter2D(Collider2D coll) {
67          if (coll.gameObject.name == "Ball" && state == State.Seek) {
68              ChangeState(State.Idle);
69          }
70      }
71  }
```

Porém, imagine como seria o seu código se você adotasse esse esquema em um personagem que precisasse ter 20 possíveis estados. 😱 Caso seu código necessite de um ajuste, como resolver um bug, por exemplo, por favor, não peça para um amigo dar manutenção! Provavelmente, você terá um "código spaghetti (um código tão bagunçado que parece um emaranhado de uma macarronada!)" tão difícil de dar manutenção, que tal demanda pode ser capaz de abalar laços de amizades. 😁 Em vez de perder um amigo, é melhor procurar uma solução alternativa.

Há também um outro problema associado à solução apresentada. Imagine que você tenha dois comportamentos que são diferentes, mas que possuem muitos estados em comum. Da forma em que está, as partes do código referentes aos estados comuns precisam ser duplicadas em cada comportamento. Isso dificulta ainda mais a manutenção porque uma simples alteração precisa ser replicada em vários arquivos. Novamente, mais um motivo para não incomodar seu amigo!

Enfim, a implementação apresentada é uma solução adequada quando se sabe que o número de estados é pequeno e que os estados não são reusados em diferentes comportamentos. Um jogo no estilo *PacMan* estaria de bom tamanho para usar essa estratégia nos seus fantasmas. Porém, se você adotar essa estratégia em um jogo com mais estados, prepare-se para ter dor de cabeça. Nesse caso, é melhor usar uma solução que será apresentada na seção seguinte.

3. MEF usando blocos de comando

3.1 MEF usando o padrão de projeto State

Na aula passada, você viu rapidamente o que são padrões de projeto. Inclusive, o padrão de projeto ***Singleton***, não foi? Pois bem, nesta seção, será apresentado um outro padrão de projeto, chamado ***State***. Como o nome diz, tem tudo a ver com estados. 😁

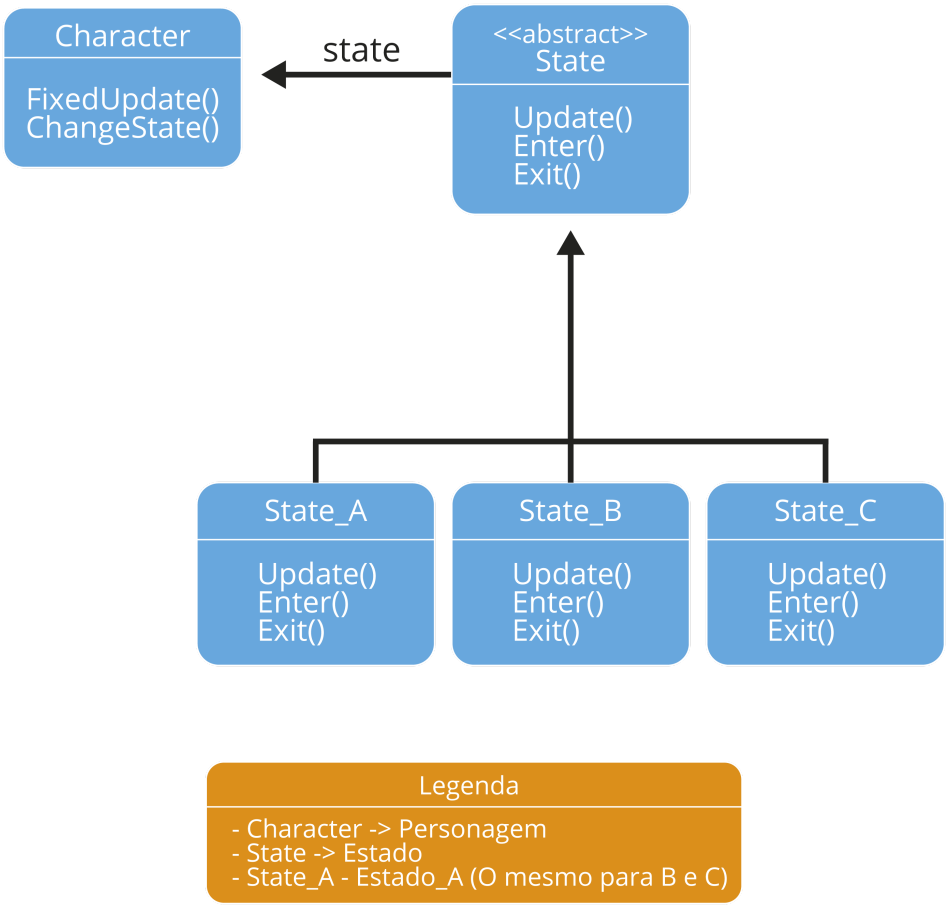
Um **padrão de projeto** é uma solução generalizada para um problema recorrente. No caso do *Singleton*, o problema recorrente é quando se precisa que uma classe tenha uma única instância e que ela seja facilmente acessível pelos demais objetos da aplicação. Quando você se deparar com essa necessidade em algum momento da programação, não precisa mais perder tempo tentando achar uma solução adequada. Use o padrão *Singleton* e está resolvido.

Da mesma forma, sempre que for necessário que um objeto altere seu comportamento quando alterações internas ocorrerem (percebeu uma relação com o que está sendo estudado nessa aula?), você não precisa mais perder tempo procurando uma solução. Basta usar o padrão State para esse problema recorrente. Simples assim! O que você precisa, agora, é saber como esse padrão funciona e como ele é implementado.

Primeiramente, o padrão *State* separa o comportamento do objeto. Nesse caso, é como se o comportamento não fizesse parte do personagem. Ele apenas o utiliza, ou seja, o personagem "possui" um comportamento. Quando o estado do personagem é alterado, o que ocorre é que ele deixa de "possuir" o comportamento atual e passa a "possuir" um outro. Faz sentido, não é mesmo?

A implementação dessa estratégia, na qual personagem e comportamento encontram-se separados, passa pelo uso de mecanismo de composição de classes (associação). Com essa separação, é possível reutilizar o mesmo comportamento em diferentes personagens. Porém, como no Unity se trabalha com C#, que é uma linguagem de tipagem estática, para que o atributo de associação do personagem ao comportamento possa assumir diferentes estados, eles precisam ser do mesmo tipo. Por essa razão, em C#, o padrão State usa também os mecanismos **herança de classes** e **polimorfismo**.

Figura 03 - Diagrama de classe ilustrando a estrutura do padrão de projeto *State*



Observe a Figura 03, ela apresenta um diagrama de classe de um padrão de projeto *State* genérico com três possíveis estados. *Character* é a classe representando o personagem (ou seu comportamento). Essa classe possui o atributo *state* da classe *State*, e métodos de atualização *Update*() e de mudança de estado *ChangeState*(). A classe *State*, por sua vez, possui declarações de métodos para implementar as ações a cada atualização do personagem (*Update*()), bem como para quando o personagem entrar (*Enter*()) e sair do estado (*Exit*()). Porém, *State* é uma classe abstrata, ou seja, não há objetos dela propriamente ditos. Ela deve, entretanto, possuir subclasses que não são abstratas (*State_A, State_B* e *State_C*). Estas, sim, podem possuir objetos. Cada uma dessas subclasses representa um possível estado e, portanto, implementa suas próprias ações de atualização, entrada e saída do estado.

Como as classes mencionadas são subclasses de *State*, seus objetos são também do tipo *State* e podem, portanto, ser associados ao atributo *state* do personagem. Assim, o que o método *Update*() de *Character* faz é chamar o método *Update*() do estado em que o personagem se encontra. Quando o atributo *state* for alterado, o *Update*() chamado será outro e, portanto, o comportamento também será outro. Por fim, o que o método *ChangeState*() de *Character* faz é chamar os métodos *Exit*() e *Enter*() do antigo e do novo estado, respectivamente. Veja o código 04 a seguir.

Código 04 – Métodos *FixedUpdate*() e *ChangeState*() da classe *Character*

```
1  void FixedUpdate () {
2      state.Update();
3  }
4
5  void ChangeState(State newState) {
6      state.Exit();
7      state = newState;
8      state.Enter();
9  }
```

O Código 04 mostra como a atualização e a mudança de estado são implementadas. Essa parte do código da classe *Character* não muda, independentemente do tipo de estado, pois o que muda é o objeto associado ao atributo *state*. Se o objeto for da classe *State*_**A**, o método *Update*() chamado será dessa classe. Se o objeto for da classe *State*_**B**, o *Update*() será dela, e assim por diante.

Talvez seja difícil entender todo o mecanismo apenas descrevendo-o, não é mesmo? Então, aplique o padrão no exemplo do cobrador de pênalti para que você possa compreender melhor a ideia.

Da mesma forma que na implementação da **MEF** usando blocos de comandos, nessa implementação você terá também dois possíveis estados. O comportamento do jogador será composto, portanto, de uma classe abstrata para generalizar todos seus possíveis estados e duas subclasses para especificar cada possível estado. Denominadas de *SoccerState* a classe abstrata, e *SeekState* e *IdleState* as subclasses para **ir em direção à bola** e **não fazer nada**, respectivamente.

Você deve estar se perguntando por que irá implementar a *SoccerState* como uma classe abstrata e não uma interface. O fato é que, da mesma forma que o personagem possui uma referência para seu estado (através do atributo *state*), o estado também precisa ter uma referência para o personagem, de forma que ele possa consultar seus atributos e executar seus métodos.

Há duas opções para que o estado tenha a referência do personagem; ou **o estado a recebe a cada chamada de seus métodos *Update*(), *Enter*() e *Exit*()**, ou **você a armazena em um atributo**. Escolha a última opção e vá, portanto, guardar a referência para o personagem em um atributo. Você deve saber que interfaces possuem apenas definições de métodos para que as classes possam implementar. Sendo assim, o estado não pode ser uma *interface*, mas uma classe abstrata. Os métodos precisam ser, entretanto, virtuais para que o **polimorfismo** possa funcionar. O Código 05 apresenta a classe implementada.

Código 05 – A classe *SoccerState* (classe abstrata da qual todos os possíveis estados do jogador irão derivar)

```
1   public abstract class SoccerState {
2      protected SoccerPlayer owner;
3
4      public SoccerState(SoccerPlayer o) {
5         owner = o;
6      }
7
8      public virtual void Update(){}
9      public virtual void Enter(){}
10     public virtual void Exit(){}
11  }
```

Note que os métodos *Update(), Enter()* e *Exit()* são virtuais. Isso permite que você tenha uma implementação *default* para cada método sem precisar que as subclasses sejam obrigadas a implementá-los. No caso, sua implementação *default* não faz nada (o bloco de comandos está vazio).

A classe *SeekState* deve derivar da *SoccerState* e implementar os métodos necessários para detalhar seu comportamento nesse estado, ou seja, durante a atualização *Update*() e na saída do estado *Exit*(). Durante a atualização, ele atualiza o alvo do jogador e o envia em direção à bola através do comportamento de navegação *Seek*. Na saída do estado, ele manda o jogador chutar a bola.

O Código 06 apresenta o código resultante. Lembre-se de que *onwer* é um atributo herdado da classe *SoccerState* e possui a referência para o jogador.

Código 06 – Código resultante em que a classe *SeekState* é um subtipo de *SoccerState* (é uma subclasse)

```
1   public class SeekState : SoccerState {
2       public SeekState(SoccerPlayer o) : base(o) {
3       }
4
5       public override void Update() {
6           GameObject ball = GameManager.Instance.ball;
7           Vector2 steering = owner.steer.Seek(ball.transform.position);
8           owner.steer.ApplySteering(steering);
9       }
10
11      public override void Exit() {
12          owner.ShotBall();
13      }
14  }
```

O estado zen-budista de “não fazer nada” (IdleState) é implementado de forma similar ao *SeekState*. Porém, no seu método de atualização, ele deve ir parando as ações anteriores. Como o jogador estava correndo no estado anterior devido ao comportamento de navegação *Seek*, ele não pode simplesmente parar de um momento para o outro, pois há uma inércia envolvida. A atualização deve, portanto, aplicar uma força de navegação para pará-lo. Sua implementação, apresentada no Código 07, usa o comportamento de navegação *Stop*(), que não foi introduzido na aula de Comportamentos de navegação, mas cujo conceito é bastante simples. O que ele faz é reduzir, a cada atualização, 1/4 da velocidade multiplicando o inverso

da velocidade atual por 3/4. Quando a velocidade atingir uma velocidade mínima, você retornará como força o inverso de sua velocidade atual, fazendo com que o jogador pare completamente.

Código 07 – A classe *IdleState* é um subtipo de *SoccerState* (subclasse) em que comportamento é parar o que está fazendo

```
1   public class IdleState : SoccerState {
2
3       public IdleState(SoccerPlayer o) : base(o) {
4       }
5
6       public override void Update() {
7           Vector2 steering = owner.steer.Stop();
8           owner.steer.ApplySteering(steering);
9       }
10  }
```

Código 08 – Método *Stop*() da classe *SteeringBehavior* usado pelo *IdleState.*

```
1   public Vector2 Stop() {
2       if (body.velocity.magnitude < minDist)
3           return -body.velocity;
4       return -body.velocity * 3f/4f;
5   }
```

Por fim, falta implementar as alterações necessárias na classe *SoccerPlayer*. Você precisa inicialmente retirar a enumeração *SoccerState*, uma vez que agora você tem uma classe com esse nome, e o atributo *state* será uma instância da classe. Consequentemente, os acessos ao atributo precisam ser alterados, como no método *Reset*() que, em vez de inicializar o estado do jogador com uma constante da enumeração, inicializa o estado atribuindo-lhe uma instância da classe *SeekState*. O Código 09 mostra a nova versão da classe *SoccerPlayer*.

Código 09 – Nova versão da classe *SoccerPlayer* usando a **MEF** implementada através do padrão *State*

```
1  public class SoccerPlayer : MonoBehaviour {
2
3      SoccerState state;
4
5      // ... parte omitida porque não houve alteração
6
7      public void Reset() {
8          state = new SeekState(this);
9          steer.Reset();
10     }
11
12     void FixedUpdate () {
13         state.Update();
14     }
15
16     void ChangeState(SoccerState newState) {
17         state.Exit();
18         state = newState;
19         state.Enter();
20     }
21
22     void OnTriggerEnter2D(Collider2D coll) {
23         if (coll.gameObject.name == "Ball" && state is SeekState) {
24             ChangeState(new IdleState(this));
25         }
26     }
27 }
```

4. Gerenciando eventos

A Máquina de Estados Finitos que você desenvolveu até agora possui apenas dois estados e uma transição. A transição é realizada no objeto *SoccerPlayer*, quando o *BoxCollider2D* dele entra no *CircleCollider2D* da bola, fazendo-o mudar do estado de **ir em direção da bola** para o estado de **não fazer nada**. Na saída do estado de "ir em direção à bola", a ação "chutar" é realizada. Veja, então, que essa não é uma situação complexa. Os estados e as transições são fáceis de gerenciar. Agora, imagine um cenário mais complexo, com um maior número de estados e de transições. E que nele tenha o documento de *Game Design* indicando duas situações: o jogador **fazendo gol** e "saindo pra galera", e em seguida voltando à posição inicial para cobrar outro pênalti, e **ele não fazendo o gol**, errando o chute, e voltando à posição inicial, mas sem comemorar.

Imaginou? Vá para a melhor das situações, que é a de o jogador fazer gol. Afinal, você quer ver a comemoração do jogador, não é mesmo? :) O Vídeo 01 ilustra como colocar essa comemoração em prática. Nele, você notará que na execução apresentada, a ação do goleiro será retirada de forma que todos os chutes realizados pelo batedor sejam gol.

Vídeo 02 - Exemplo de cobrança de pênalti com comemoração do jogador

Conteúdo interativo, acesse o Material Didático.

O esquema da **MEF** da Figura 04 apresenta uma possível máquina de estados para o cenário proposto. Como você vai gerenciar todos esses eventos? Onde colocar os testes para chamar as transições? Veja a seguir.

Figura 04 - Diagrama de estados mais completo do cobrador de pênaltis

Como o diagrama da Figura 02 representa o comportamento do jogador cobrador de pênalti, os estados apresentados são próprios do jogador. Porém, se você prestar atenção nos eventos que geram as transições, há tanto **eventos internos**, próprios do jogador, quanto **eventos externos**, que dependem de outros elementos do jogo. Por exemplo, “término da comemoração” é algo interno. O próprio estado pode gerenciar quando o jogador terminará sua comemoração. Mas saber se a bola que ele chutou foi gol ou não depende de outros objetos. Você vai distinguir esses dois tipos de eventos porque eles possuem mecanismos diferentes.

Os **eventos internos** são mais fáceis de gerenciar. Como eles dependem apenas do próprio estado, as transições podem ser realizadas nos métodos de atualização do estado. Por exemplo, o estado de comemoração pode ser

implementado como no Código 10. A comemoração implementada faz o jogador ir em uma direção aleatória durante meio segundo, depois muda para outra direção aleatória durante meio segundo e assim por diante, realizando esse processo seis vezes (count > 5).

Código 10 – Código do estado de comemoração do jogador

```
1  public class CelebrateState : SoccerState {
2
3     float   time;
4     int     count;
5     Vector2 target;
6
7     public CelebrateState(SoccerPlayer o) : base(o) {
8     }
9
10    public override void Enter() {
11     time = Time.time;
12     count = 0;
13     target = Random.insideUnitCircle + (Vector2) owner.transform.position;
14    }
15
16    public override void Update() {
17     if (Time.time - time < 0.5) {
18        Vector2 steering = owner.steer.Seek(target);
19        owner.steer.ApplySteering(steering);
20     } else {
21        count++;
22        time = Time.time;
23        target = Random.insideUnitCircle + (Vector2)owner.transform.position;
24        if (count > 5) {
25          owner.ChangeState(new BackToInitialPosState(owner));
26        }
27     }
28    }
29 }
```

A classe *CelebrateState* possui três atributos:

- *Time*: gerencia o tempo do jogador quando ele estiver indo em uma direção;
- *Count*: gerencia o número de vezes que o jogador mudará de direção;
- *Target*: define a direção em que o jogador está indo.

A variável *count* será incrementada sempre que o tempo de meio segundo for ultrapassado, e quando seu valor for maior que cinco, você fará a transição para o próximo estado, *BackToInitialPosState*.

Se o estado de comemoração só depende dele, por outro lado, as transições do estado de aguardar o resultado do chute dependem, exclusivamente, de outros objetos. Como você irá, então, implementar isso?

Suponha que você coloque testes na atualização do estado "aguardar o resultado do chute", verificando se a bola chutada foi gol ou não. A princípio, não há mal algum em fazer isso. Pode-se averiguar, a cada atualização do estado, se o círculo envoltório da bola se sobrepõe à caixa envoltória do gol. Se isso ocorrer em algum momento, é porque foi gol. E se o chute não resultar em gol? Você precisará fazer algo similar, testando a cada atualização se o círculo envoltório da bola se sobrepõe ao círculo envoltório do goleiro. E se isso acontecer em algum momento, é porque o goleiro segurou a bola.

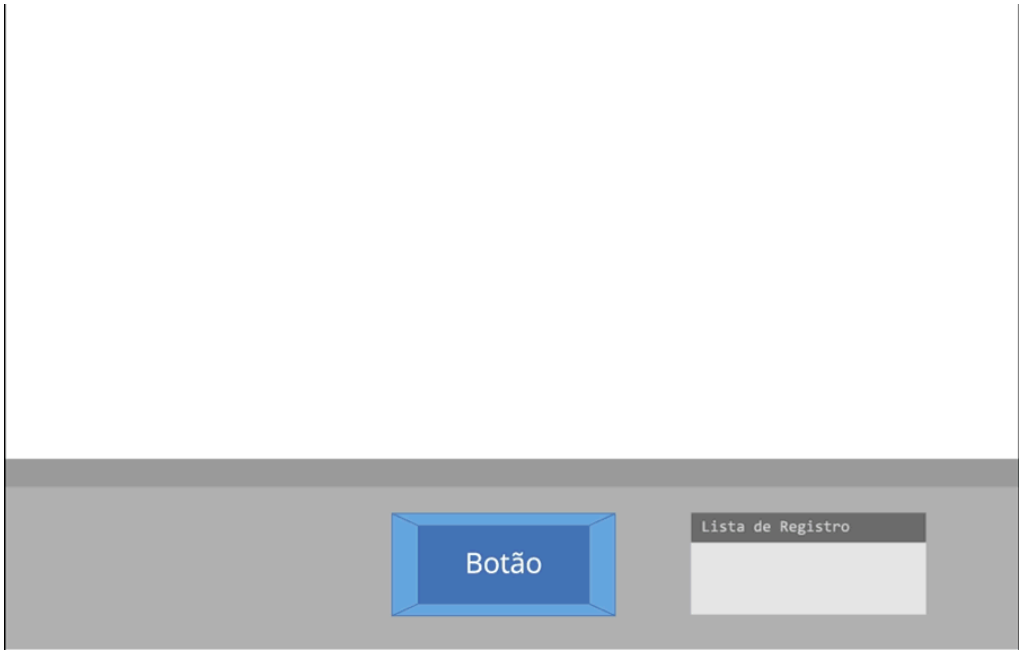
Bom... mas imagine isso com os 22 jogadores em campo, todos ansiosos para saber se o chute resultou em gol ou não. 😱 Se você implementar dessa forma, terá as mesmas verificações 22 vezes, consumindo um tempo desnecessário do processador. Você precisará, então, inverter o processo de verificação. Em vez de cada um testar se a bola se sobrepõe a outros elementos, você fará com que apenas a bola execute esse teste. Porém, os demais objetos que estejam interessados em saber do gol precisam se "cadastrar" para serem informados sobre o evento. Caso a bola constate que um gol ocorreu, ela passará essa informação a todos os objetos previamente cadastrados.

Esse mecanismo descrito é outro padrão de projeto chamado de ***Observer*** (você já viu dois nessa disciplina: *Singleton* e *State*) e é amplamente utilizado para implementar sistemas baseados em eventos. Por exemplo, têm-se duas aplicações de sistemas baseados em eventos, uma com um sistema de *interface* que possui menus e botões e uma que executa rotinas quando um pacote de dados chega da rede. Enfim, a maioria das aplicações reais são baseadas em eventos. Por isso, conhecer esse padrão é muito importante.

O padrão ***Observer*** pode ser implementado de duas maneiras. Em ambas, há os conceitos de "observável" e "observador". No caso de uma aplicação com botões à espera de serem clicados pelo usuário, os botões são os elementos observáveis, e os

observadores são os objetos ou sub-rotinas que se registram para serem chamados quando o evento do clique do botão ocorrer. Normalmente, registra-se observadores passando uma sub-rotina (função ou método) para o objeto observável. Pode-se ter, inclusive, várias sub-rotinas registradas para o mesmo evento no objeto observável. Quando o evento ocorrer, todas as sub-rotinas registradas serão chamadas.

Figura 05 - Esquema ilustrando o registro de dois observadores no padrão Observer



As sub-rotinas registradas no observável são chamadas de *callbacks* porque você passa suas referências a fim de serem "chamadas de volta", quando o evento ocorrer. Porém, nem toda linguagem de programação dá suporte à passagem de sub-rotinas. Nesse caso, você passa um objeto que possua um método com uma assinatura específica (pois o método será chamado quando o evento ocorrer). Mas, como garantir que o objeto a ser registrado tenha obrigatoriamente um método com assinatura específica? Fazendo com que o objeto implemente uma *interface* predefinida.

Felizmente, C# dá suporte à passagem de sub-rotinas, especificando a assinatura através da palavra-chave *delegate*. Só que melhor ainda, C# dá suporte ao registro das sub-rotinas e à chamada das sub-rotinas registradas através de um tipo de elemento chamado *event* (tudo a ver 😁). Você já utilizou o *delegate* e o *event* anteriormente, mas apenas agora todos os seus detalhes estão sendo explicados. Enquanto o delegate permite especificar a assinatura da sub-rotina que

será chamada quando um evento ocorrer, ou seja, quais são seus parâmetros e seu tipo de retorno, o *event* permite registrar sub-rotinas (ou remover seu registro) que atendem a assinatura especificada pelo *delegate* e chamar todas as sub-rotinas quando for necessário.

Você pode ver o funcionamento dos mecanismos através dos códigos a seguir. Na implementação do *script* da bola (Código 11), foi definido um *delegate* chamado *GoalEvent* que especifica a assinatura de uma sub-rotina que não recebe nenhum parâmetro e não retorna nada (*void*). Em seguida, determinados dois tipos de eventos; um para ser disparado quando ocorrer o gol (*OnGoal*) e outro para quando o chute falhar, ou seja, quando não for gol (*OnShotFailed*). Quando a bola colidir com o gol (que possui a tag *Goal*), o evento *OnGoal* será disparado, chamando todas as sub-rotinas que foram registradas no evento. De maneira similar, quando a bola colidir com o goleiro (que possui a tag *GoalKeeper*), o evento *OnShotFailed* será disparado. Esses eventos foram definidos como públicos porque serão usados por outros objetos para se registrarem.

Código 11 – Código da bola (Ball.cs) o estado de comemoração do jogador

```
1  public class Ball : MonoBehaviour {
2
3    public delegate void GoalEvent();
4    public event GoalEvent OnGoal;
5    public event GoalEvent OnShotFailed;
6
7    private Rigidbody2D body;
8    private Vector2 initialPosition;
9
10   // Use this for initialization
11   void Awake () {
12     body = GetComponent<Rigidbody2D>();
13     initialPosition = body.position;
14     Reset();
15   }
16
17   public void Reset() {
18     body.position = initialPosition;
19     body.velocity = Vector2.zero;
20   }
21
22   void OnTriggerEnter2D(Collider2D coll) {
23     if (coll.gameObject.name == "Goal") {
24       OnGoal();
25     }
26     else if (coll.gameObject.name == "GoalKeeper") {
27       OnShotFailed();
28     }
29   }
30 }
```

Na implementação do estado de espera pelo resultado do chute (Código 12), o construtor armazena a referência para o *script* da bola para que você possa registrar sub-rotinas do estado para quando os eventos *OnGoal* e *OnShotFailed* ocorrerem. Isso é realizado no momento em que o jogador entra no estado (o método *Enter()* é chamado). O registro é feito através do operador +=, passando os métodos *OnGoal()* e *OnShotFailed()* do estado, e que, por sinal, atendem a assinatura do *delegate* especificada nos eventos (ou seja, não recebe parâmetro e não retorna nada). Na saída do estado (quando método *Exit()* é chamado), os métodos *OnGoal()* e *OnShotFailed()* são removidos dos registros do evento. É importante removê-los porque, apesar de o objeto do estado não ser mais utilizado, ele não poderá ser liberado pelo gerenciador de memória da linguagem (vulgo coletor de lixo ou *garbage collector*), por ainda ter referências para ele.

Código 12 – Código do estado de esperar o resultado do chute (WaitShotResultState.cs) mostrando como o eventos são registrados, capturados e removidos do registro

```
1  public class WaitShotResultState : SoccerState {
2
3    Ball ball;
4
5    public WaitShotResultState(SoccerPlayer o) : base(o) {
6      ball = GameManager.Instance.ball.GetComponent<Ball>();
7    }
8
9    public override void Enter() {
10     ball.OnGoal += OnGoal;
11     ball.OnShotFailed += OnShotFailed;
12   }
13
14   public override void Exit() {
15     ball.OnGoal -= OnGoal;
16     ball.OnShotFailed -= OnShotFailed;
17   }
18
19   public override void Update() {
20     Vector2 steering = owner.steer.Stop();
21     owner.steer.ApplySteering(steering);
22   }
23
24   public void OnGoal() {
25     owner.ChangeState(new CelebrateState(owner));
26   }
27
28   public void OnShotFailed() {
29     owner.ChangeState(new BackToInitialPosState(owner));
30   }
31 }
```

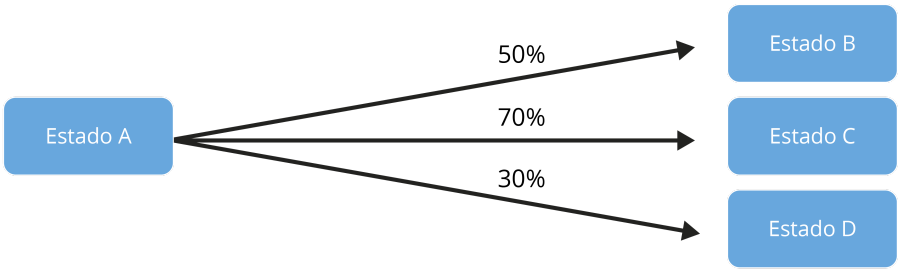
Quando a linguagem de programação utilizada não possibilita passar uma sub-rotina, pode-se registrar um objeto de uma classe que implementa uma interface específica.

Bem, até agora você conheceu o básico de uma **MEF**, mas há inúmeras variações que podem ser exploradas... e que serão exercitadas nas atividades. Uma versão bastante utilizada é a **MEF Probabilística**.

Uau! Que nome bonito, não é? **Máquina de Estados Finitos Probabilística!**

Ela consiste apenas em fazer com que um mesmo evento durante um estado possa gerar um ou mais estados. Então, quando o personagem se encontra em um estado que possui transições para dois ou mais estados a partir do mesmo evento, cada transição poderá ter sua probabilidade associada, como ilustra a Figura 06.

Figura 06 - Exemplo de Máquina de Estados Finitos Probabilística



Atividade 01 – Implementando os demais estados

Agora que você já sabe como especificar os estados usando o padrão *State* e eventos que realizam as transições usando o padrão *Observer*, implemente os demais estados do diagrama da Figura 04. Além do cobrador de pênalti, implemente estados para o goleiro usando o padrão *State*. Ele deve também comemorar quando agarrar o pênalti e, depois, voltar à sua posição inicial para tentar segurar um novo.

Resumo

Nessa aula, você compreendeu como elaborar comportamentos mais complexos para os jogadores. De fato, você está começando a explorar um mundo novo, onde os personagens podem adaptar seus comportamentos em função do estado em que eles se encontram. O uso de **Máquinas de Estados Finitos** (MEF) permitem essa adaptação. Você viu também como implementar uma **MEF** através de duas abordagens: **usando blocos de comandos** e **usando o padrão de projeto *State***. Esta última é mais flexível e versátil do que o bloco de comandos porque separa o personagem do seu comportamento, permitindo reutilizá-lo.

Além do padrão *State*, você viu igualmente outro padrão de projeto amplamente utilizado em sistemas computacionais, em especial nas aplicações cujo funcionamento depende de eventos externos. O padrão *Observer* serve para registrar sub-rotinas ou métodos de objetos a serem chamados quando um elemento "observável" da aplicação gerar um "evento". Nesse momento, todos os observadores registrados são executados. Felizmente, a própria linguagem C# já fornece as ferramentas necessárias para implementar o padrão *Observer*.

Bom... sei que você deve estar contente porque o comportamento do jogador está ficando mais polido. Mas imagino o que você deve estar pensando: "Caramba! Vários arquivos de código só para a cobrança de um pênalti! Imagine como será para implementar todos os estados de uma partida de futebol completa".

Se você acha que comportamentos mais complexos produzam também uma complexidade grande para gerenciar e dar manutenção no código, você está coberto de razão. Certamente, o uso de **MEF** não é um problema para jogos de plataforma, nos quais os personagens inimigos possuem basicamente três estados: aguardar, perseguir e atacar. Também não é um problema nos jogos RPG mais simples em que, por exemplo, o responsável por uma taverna fica aguardando o usuário para iniciar um diálogo e passar-lhe uma missão. Depois que o usuário retornar, ele vai dialogar novamente em função de a missão ter sido cumprida ou não. Enfim, em inúmeros estilos de jogos o uso de uma **MEF** é a solução ideal. Porém, no caso de jogo de futebol, o número de possíveis estados pode crescer de forma a tornar o código não mais gerenciável.

Há várias alternativas para esse problema e serão apresentadas algumas delas nas aulas seguintes, inclusive usando variações da **MEF**. Por exemplo, há um tipo de **MEF** que se chama **Máquina de Estados Finitos Hierárquica**. Ela permite que um estado possua outra **MEF** dentro dela. Assim, o jogador pode entrar, por exemplo, no estado de "cobrar pênalti" e, dentro desse estado, ter todos os estados apresentados no diagrama da Figura 02. Para comportamentos mais complexos, a **MEF Hierárquica** é uma solução adequada.

Por enquanto, fica o seguinte aprendizado: "é essencial aprender várias técnicas para saber escolher a mais adequada para o seu problema". A **MEF** é uma das técnicas mais utilizadas na indústria de jogos (se não for a mais), mas não é solução para todos os problemas. Lembre-se de não ser como martelo. Nem tudo na vida é prego. 😁

Até a próxima!

Referências

MILLINGTON, Ian; FUNGE, John. **Artificial intelligence for games**. 2. ed. Massachusetts: Morgan Kaufmann Eds, 2009.

MADHAV, Sanjay. Game **Programming algorithms and techniques**: a plataform-agnostic approach. Massachusetts: Addison-Wesley, 2014.