State-Driven Agent Design

(Design de agente dirigido por estados)

Máquinas de estados finitos, ou do inglês *Finite-State Machine*(FSM), como geralmente são referidas, tem sido, por muitos anos a ferramenta preferida dos desenvolvedores de *Inteligência Artificial* (IA) para dotar um agente de jogo com a ilusão de inteligência. Você encontrará as FSMs de um tipo ou de outro em quase todos os jogos lançados desde os primeiros dias dos videogames, e, apesar da crescente popularidade de arquiteturas de agentes mais esotéricas, elas continuaram a existir por muito tempo. Aqui estão apenas algumas das razões pelas quais:

Elas são rápidas e simples de codificar. Existem muitas maneiras de programar uma máquina de estados finitos e quase todas são razoavelmente simples de implementar. Você verá várias alternativas descritas neste capítulo, juntamente com os prós e contras de usá-las.

Elas são fáceis de depurar. Como o comportamento de um agente de jogo é dividido em pedaços facilmente gerenciáveis, se um agente começar a agir de maneira estranha, pode ser depurado adicionando código de rastreamento a cada estado. Dessa forma, o programador de IA pode acompanhar facilmente a sequência de eventos que precede o comportamento com defeito e agir de acordo.

Elas têm pouca sobrecarga computacional. As máquinas de estados finitos usam muito pouco tempo precioso do processador porque seguem essencialmente regras codificadas rigidamente. Não há realmente "pensamento" envolvido além do tipo de raciocínio "se isso, então aquilo".

Elas são intuitivas. É da natureza humana pensar em coisas como estando em um estado ou outro, e frequentemente nos referimos a nós mesmos como estando em tal estado. Quantas vezes você "entrou em um estado" ou se viu em "o estado de espírito certo"? Claro, os humanos não funcionam exatamente como máquinas de estados finitos, mas às vezes achamos útil pensar em nosso comportamento dessa maneira. Da mesma forma, é bastante fácil dividir o comportamento de um agente de jogo em vários estados e criar as regras necessárias para manipulá-los. Por essa mesma razão, as máquinas de estados finitos também facilitam a discussão do design de sua IA com não programadores (como produtores de jogos e designers de níveis, por exemplo), melhorando a comunicação e a troca de ideias.

Implementando uma máquina de estados finitos

Existem várias maneiras de implementar máquinas de estado finitos. Uma abordagem ingênua é usar uma série de instruções `if-then` ou o mecanismo ligeiramente arrumado de uma instrução `switch`. Usando um `switch` com um tipo `enum` para representar os estados se parece com o seguinte:

```
enum TipoEstado
{
	Fuga,
	Patrulha,
	Ataque
};

void Agente::atualizaEstado(TipoEstado estadoAtual)
{
	switch(estadoAtual)
	{
		case estado_Fuga:
			fugirDoInimigo();
			if (seguro())
			{
				alteraEstado(stado_Patrulha);
			}
			break;

		case estado_Patrulha:
			seguirCaminhoDePatrulha();
			if (ameacado())
			{
				if (maisForteQueInimigo())
				{
					alteraEstado(estado_Ataque);
				}
				else
				{
					alteraEstado(estado_Fuga);
				}
			}
			break;

		case estado_Ataque:
			if (maisFracoQueInimigo())
			{
				alteraEstado(estado_Fuga);
			}
			else
			{
				golpeiaInimigo();
			}
			break;
	}//end switch
}
```

Embora à primeira vista essa abordagem pareça razoável, quando aplicada na prática a algo mais complicado do que os objetos de jogo mais simples, a solução com `switch/if-then` se torna um monstro espreitando nas sombras, esperando para atacar. À medida que mais estados e condições são adicionados, esse tipo de estrutura rapidamente se transforma em algo que se assemelha a um emaranhado de espaguete, tornando o fluxo do programa difícil de entender e criando um pesadelo de depuração. Além disso, ela é inflexível e difícil de estender além do escopo de seu design original, caso isso seja desejável... e, como todos sabemos, isso ocorre na maioria das vezes. A menos que você esteja projetando uma máquina de estados para implementar um comportamento muito simples (ou seja um gênio), é quase certo que

você se encontrará ajustando o agente para lidar com circunstâncias não planejadas antes de aprimorar o comportamento para obter os resultados que você pensou que obteria quando planejou pela primeira vez a máquina de estados!

Além disso, como um desenvolvedor de IA, muitas vezes você precisará que um estado execute uma ação específica (ou ações) quando ele é inicialmente ativado ou quando o estado é encerrado. Por exemplo, quando um agente entra no estado "Fugir", você pode querer que ele agite os braços no ar e grite "Arghhhhhhh!" Quando finalmente escapa e muda para o estado "Patrulha", você pode querer que ele solte um suspiro, limpe a testa e diga "Ufa!" Essas são ações que ocorrem apenas quando o estado "Fugir" é ativado ou encerrado e não durante a etapa de atualização usual. Portanto, essa funcionalidade adicional idealmente deve estar incorporada à sua arquitetura de máquina de estados. Fazer isso dentro do contexto de uma estrutura de switch ou if-then seria acompanhado por muitos dentes cerrados e ondas de náusea, e produziria um código muito confuso.

Tabelas de Transição de Estados

Estado atual	Condição	Estado de transição
Fuga	Seguro	Patrulha
Ataque	maisFracoQueInimigo	Fuga
Patrulha	Ameacado E maisForteQueInimgo	Ataque
Patrulha	Ameaçado E maisFracoQueInimigo	Fuga

Esta tabela pode ser consultada por um agente em intervalos regulares, permitindo-lhe fazer quaisquer transições de estado necessárias com base no estímulo que recebe do ambiente do jogo. Cada estado pode ser modelado como um objeto ou função separada existente externamente ao agente, proporcionando uma arquitetura limpa e flexível. Uma que é muito menos suscetível à "espaguetificação" do que a abordagem `if-then/switch` discutida na seção anterior.

Alguém certa vez me disse que uma visualização vívida e boba pode ajudar as pessoas a entender um conceito abstrato. Vamos ver se funciona…

Imagine um gatinho robô. Ele é brilhante e fofo, tem fios como bigodes e um compartimento em sua barriga onde cartuchos, análogos aos seus estados, podem ser conectados. Cada um desses cartuchos é programado com lógica, permitindo que o gatinho execute um conjunto específico de ações. Cada conjunto de ações codifica um comportamento diferente; por exemplo, "brincar com linha", "comer peixe" ou "fazer cocô no tapete". Sem um cartucho inserido em sua barriga, o gatinho é uma escultura metálica inanimada, capaz apenas de sentar ali e parecer fofo... de uma maneira meio "Metal Mickey".

O gatinho é muito ágil e tem a capacidade de trocar autonomamente seu cartucho por outro, se for instruído a fazê-lo. Ao fornecer as regras que ditam quando

um cartucho deve ser trocado, é possível concatenar sequências de inserções de cartuchos, permitindo a criação de todo tipo de comportamento interessante e complicado. Essas regras são programadas em um pequeno chip situado dentro da cabeça do gatinho, que é análogo à tabela de transição de estados que discutimos anteriormente. O chip se comunica com as funções internas do gatinho para obter as informações necessárias para processar as regras (como o quão faminto o gatinho está ou o quão brincalhão ele se sente).

Como resultado, o chip de transição de estados pode ser programado com regras como:

```
IF Kitty_Com_Fome E NÃO Kitty_Brincalhao TROCAR_CARTUCHO comer_peixe
```

Essas regras indicam que, se o gatinho estiver com fome e não estiver brincalhão, ele deve trocar seu cartucho atual pelo cartucho "`comer_peixe`". Isso representa uma instrução específica para o gatinho sobre como se comportar com base em seu estado atual.

Todas as regras na tabela são testadas a cada passo de tempo, e instruções são enviadas para o Kitty trocar os cartuchos de acordo. Esse tipo de arquitetura é muito flexível, tornando fácil expandir o repertório do gatinho adicionando novos cartuchos. Cada vez que um novo cartucho é adicionado, o proprietário só precisa usar uma chave de fenda na cabeça do gatinho para remover e reprogramar o chip de regras de transição de estado. Não é necessário interferir em qualquer outra parte da circuitaria interna.

Regras Embutidas

Uma abordagem alternativa é *incorporar as regras para as transições de estado diretamente nos próprios estados*. Aplicando esse conceito ao Robo-Kitty, o chip de transição de estados pode ser dispensado e as regras podem ser movidas diretamente para os cartuchos. Por exemplo, o cartucho para "**brincar com a linha**" pode monitorar o nível de fome do gatinho e instruí-lo a trocar para o cartucho "**comer peixe**" quando perceber que a fome está aumentando. Por sua vez, o cartucho "**comer peixe**" pode monitorar o intestino do gatinho e instruí-lo a trocar para o cartucho "**fazer cocô no tapete**" quando perceber que os níveis de cocô estão perigosamente altos.

Embora cada cartucho possa estar ciente da existência de qualquer um dos outros cartuchos, cada um é uma unidade autônoma e não depende de lógica externa para decidir se deve ou não permitir sua troca por uma alternativa. Como resultado, é uma questão simples adicionar estados ou até mesmo trocar todo o conjunto de cartuchos por um conjunto completamente novo (talvez aqueles que façam o pequeno Kitty se comportar como um raptor). Não há necessidade de usar uma chave de fenda na cabeça do gatinho, apenas em alguns dos cartuchos.

Vamos dar uma olhada em como essa abordagem é implementada no contexto de um videogame. Assim como os cartuchos do Kitty, os estados são encapsulados como objetos e contêm a lógica necessária para facilitar as transições de estado. Além

disso, todos os objetos de estado compartilham uma interface comum: uma classe virtual pura chamada "Estado". Aqui está uma versão que fornece uma interface simples:

```
class Estado
{
      public:
            virtual void executa (Troll* troll) = 0;
};
```

Agora, imagine uma classe `Troll` que possui variáveis de membro para atributos como saúde, raiva, stamina, etc., e uma interface que permite a um cliente consultar e ajustar esses valores. Um `Troll` pode ser dotado da funcionalidade de uma máquina de estados finitos adicionando um ponteiro para uma instância de um objeto derivado da classe State, e um método que permite ao cliente alterar a instância para a qual o ponteiro está apontando.

```
class Troll
{
      /* ATRIBUTOS OMITIDOS */
      Estado* m_pEstadoAtual;

      public:
            /* INTERFACE PRA ATRIBUTOS OMITIDOS */
            void atualiza()
            {
                  m_pEstadoAtual->executa(this);
            }

            void alteraEstado(const Estado* pNovoEstado)
            {
                  delete m_pEstadoAtual;
                  m_pEstadoAtual = pNovoEstado;
            }
};
```

Quando o método "`atualiza`" de um `Troll` é chamado, ele, por sua vez, chama o método "`executa`" do tipo de estado atual com o ponteiro "`this`". O estado atual pode então usar a interface do `Troll` para consultar seu proprietário, ajustar os atributos de seu proprietário ou efetuar uma transição de estado. Em outras palavras, como um `Troll` se comporta quando atualizado pode depender completamente da lógica em seu estado atual. Isso é melhor ilustrado com um exemplo, então vamos criar alguns estados para permitir que um `troll` fuja de inimigos quando se sente ameaçado e durma quando se sente seguro.

```
//---------------------------------Estado_Fuga
class Estado_Fuga : public Estado
{
      public:
            void executa(Troll* troll)
            {
                  if (troll->seguro())
                  {
                        troll->alteraEstado(new Estado_Dormir());
                  }

                  else
                  {
                        troll->afataseDoInimigo();
                  }
            }
};

//---------------------------------Estado_Dormir
class Estado_Dormir : public Estado
{
      public:
            void executa(Troll* troll)
            {
                  if (troll->ameacado())
                  {
                        troll->alteraEstado(new Estado_Fuga())
                  }
                  else
                  {
                        troll->cochila();
                  }
            }
};
```

Como você pode ver, quando atualizado, um `troll` se comportará de maneira diferente, dependendo de qual dos estados o "`m_pEstadoAtual`" está apontando. Ambos os estados são encapsulados como objetos e ambos fornecem as regras que afetam a transição de estados. Tudo muito organizado e arrumado.

Essa arquitetura é conhecida como o padrão de design de estado e fornece uma maneira elegante de implementar um comportamento orientado por estados. Embora isso seja um desvio da formalização matemática de uma FSM, é intuitivo, simples de codificar e facilmente extensível. Também torna extremamente fácil adicionar ações de entrada e saída a cada estado; tudo o que você precisa fazer é criar métodos "`Entrada`" e "`Saida`" e ajustar o método "`alteraEstado`" do agente de acordo. Você verá o código que faz exatamente isso em breve.

****

A partir daqui não irei mais traduzir nomes de classes, funções, métodos, atributos, variáveis, propriedades ou exemplos.

****

O Projeto West World

Como um exemplo prático de como criar agentes que utilizam máquinas de estados finitos, vamos dar uma olhada em um ambiente de jogo onde os agentes habitam uma cidade de mineração de ouro no estilo Velho Oeste chamada West World. Inicialmente, haverá apenas um habitante - um mineiro de ouro chamado Miner Bob - mas mais tarde, neste capítulo, sua esposa também fará uma aparição. Você terá que imaginar os arbustos rodando, os adereços rangendo e a poeira do deserto soprando em seus olhos, porque West World é implementado como um aplicativo de console simples baseado em texto. Qualquer mudança de estado ou saída das ações de estado será exibida como texto na janela do console. Estou usando essa abordagem apenas de texto porque ela demonstra claramente o mecanismo de uma máquina de estados finitos sem adicionar a desordem de código de um ambiente mais complexo.

Existem quatro locais em West World: uma mina de ouro, um banco onde Bob pode depositar as pepitas que ele encontra, um saloon onde ele pode saciar sua sede e sua casa, onde ele pode dormir para descansar da fadiga do dia. Exatamente para onde ele vai e o que faz quando chega lá é determinado pelo estado atual de Bob. Ele mudará de estados dependendo de variáveis como sede, fadiga e quanto ouro ele encontrou cavando na mina de ouro.

Antes de mergulharmos no código-fonte, confira a seguinte saída de exemplo do executável WestWorld1.

```
Miner Bob: Pickin' up a nugget
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Goin' to the bank. Yes siree
Miner Bob: Depositin’ gold. Total savings now: 3
Miner Bob: Leavin' the bank
Miner Bob: Walkin' to the gold mine
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Boy, ah sure is thusty! Walkin' to the saloon
Miner Bob: That's mighty fine sippin liquor
Miner Bob: Leavin' the saloon, feelin' good
Miner Bob: Walkin' to the gold mine
Miner Bob: Pickin' up a nugget
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Goin' to the bank. Yes siree
Miner Bob: Depositin' gold. Total savings now: 4
Miner Bob: Leavin' the bank
Miner Bob: Walkin' to the gold mine
Miner Bob: Pickin' up a nugget
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Boy, ah sure is thusty! Walkin' to the saloon
Miner Bob: That's mighty fine sippin' liquor
Miner Bob: Leavin' the saloon, feelin' good
Miner Bob: Walkin' to the gold mine
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Goin' to the bank. Yes siree
Miner Bob: Depositin' gold. Total savings now: 5
Miner Bob: Woohoo! Rich enough for now. Back home to mah li'l lady
Miner Bob: Leavin' the bank
Miner Bob: Walkin' home
Miner Bob: ZZZZ...
Miner Bob: ZZZZ...
Miner Bob: ZZZZ...
Miner Bob: ZZZZ...
Miner Bob: What a God-darn fantastic nap! Time to find more gold
```

Na saída do programa, cada vez que você vê o Miner Bob mudar de local, ele está mudando de estado. Todos os outros eventos são as ações que ocorrem dentro dos estados. Vamos examinar cada um dos estados potenciais do Miner Bob em um momento, mas por enquanto, deixe-me explicar um pouco sobre a estrutura do código do exemplo.

A classe BaseGameEntity

Todos os habitantes de West World são derivados da classe base "`BaseGameEntity`". Esta é uma classe simples com um membro privado para armazenar um número de ID. Ela também especifica uma função de membro virtual pura, "`Update`", que deve ser implementada por todas as subclasses. "`Update`" é uma função que é chamada a cada etapa de atualização e será usada pelas subclasses para atualizar sua máquina de estados, juntamente com quaisquer outros dados que precisam ser atualizados a cada etapa de tempo.

A declaração da classe "`BaseGameEntity`" se parece com isso:

```
class BaseGameEntity
{
      private:
            //Cada entidade possui um número de indentificação único
            int m_ID;

            //Este é o próximo ID válido. Cada vez que uma instância de
            //BaseGameEntity é criada, este valor é atualizado
            static int m_iNextValidID;

            //Isso é chamado dentro do construtor para garantir que o ID
            //seja configurado corretamente.
            //Ele verifica se o valor passado para o método é maior ou igual
            //ao próximo ID válido antes de definir o ID e incrementar o próximo
            //ID válido
            void SetID(int val);
      public:
            BaseGameEntity(int id)
            {
                  SetID(id);
            }
            virtual ~BaseGameEntity(){}

            //Todas as entidades devem implementar um função Update
            virtual void Update()=0;
            int ID()const{return m_ID;}
};
```

Por razões que se tornarão óbvias mais tarde no capítulo, é muito importante que cada entidade em seu jogo tenha um identificador único. Portanto, na instanciação, o ID passado para o construtor é testado no método "`SetID`" para garantir que seja único. Se não for único, o programa encerrará com uma falha de assertiva (assertion failure). No exemplo fornecido neste capítulo, as entidades usarão um valor enumerado como seu identificador exclusivo. Esses valores enumerados podem ser encontrados no arquivo *EntityNames.h* como "`ent_Miner_Bob`" e "`ent_Elsa`".

A classe Miner

A classe "`Miner`" é derivada da classe "`BaseGameEntity`" e contém membros de dados que representam os vários atributos que um Mineiro possui, como sua saúde, nível de fadiga, posição, e assim por diante. Assim como o exemplo do `troll` mostrado anteriormente no capítulo, um "`Miner`" possui um ponteiro para uma instância de uma classe "`State`", além de um método para alterar para qual `State` esse ponteiro aponta.

```
class Miner : public BaseGameEntity
{
      private:
            //Um ponteiro para uma instancia de um State
            State* m_pCurrentState;

            // O local que o mineiro está atualmente situado
            location_type m_Location;

            //Quantas pepitas de ouro o mineiro tem nos bolsos
            int m_iGoldCarried;

            //Quanto dinheiro o mineiro depositou no banco
            int m_iMoneyInBank;

            //Quanto maior o valor, com mais sede o mineiro está
            int m_iThirst;

            //Quanto maior o valor, mais cansado o mineiro está
            int m_iFatigue;
      public:
            Miner(int ID);

            //Este método ainda dever implementado
            void Update();

            //Este método altera o estado atual para o novo estado
            void ChangeState(State* pNewState);

      /* A maior parte da interface foi omitida */
};
```

O método `"Miner::Update"` é direto; ele simplesmente incrementa o valor de `"m_iThirst"` antes de chamar o método `"Execute"` do estado atual. Parece assim:

```
void Miner::Update()
{
      m_iThirst += 1;
      if (m_pCurrentState)
      {
            m_pCurrentState->Execute(this);
      }
}
```

Agora que você viu como a classe “`Miner`” opera, vamos dar uma olhada em cada um dos estados em que um mineiro pode se encontrar.

Os estados do mineiro

O minerador de ouro será capaz de entrar em um dos quatro estados. Aqui estão os nomes desses estados, seguidos de uma descrição das ações e transições de estado que ocorrem dentro desses estados:

- **EnterMineAndDigForNugget (Entrar na Mina e Cavucar Pepitas)**: Se o mineiro não estiver localizado na mina de ouro, ele muda de local. Se já estiver na mina de ouro, ele cava em busca de pepitas de ouro. Quando seus bolsos estiverem cheios, Bob muda de estado para "VisitBankAndDepositGold" (Visitar o Banco e Depositar Ouro), e se, enquanto cava, ele ficar com sede, ele parará e mudará de estado para "QuenchThirst" (Matar a Sede).

- **VisitBankAndDepositGold (Visitar o Banco e Depositar Ouro)**: Neste estado, o mineiro caminhará até o banco e depositará qualquer pepita que esteja carregando. Se ele se considerar rico o suficiente, ele mudará de estado para "GoHomeAndSleepTilRested" (Ir para Casa e Dormir Até Descansar). Caso contrário, ele mudará de estado para "EnterMineAndDigForNugget" (Entrar na Mina e Cavucar Pepitas).
- **GoHomeAndSleepTilRested (Ir para Casa e Dormir Até Descansar)**: Neste estado, o mineiro retornará à sua cabana e dormirá até que seu nível de fadiga caia abaixo de um nível aceitável. Ele então mudará de estado para "EnterMineAndDigForNugget" (Entrar na Mina e Cavucar Pepitas).
- **QuenchThirst (Matar a Sede)**: Se a qualquer momento o mineiro sentir sede (cavar ouro é um trabalho sedento, você sabe), ele mudará para este estado e visitará o saloon para comprar um uísque. Quando sua sede estiver saciada, ele mudará de estado para "EnterMineAndDigForNugget" (Entrar na Mina e Cavucar Pepitas).

Às vezes, é difícil acompanhar o fluxo da lógica de estados lendo uma descrição em texto como esta, portanto, muitas vezes é útil pegar papel e caneta e desenhar um diagrama de transição de estados para seus agentes de jogo. A Figura 2.2 mostra o diagrama de transição de estados para o minerador de ouro. As bolhas representam os estados individuais e as linhas entre eles as transições disponíveis.

O Padrão de State Design Revisitado

Você viu uma breve descrição deste padrão de design algumas páginas atrás, mas não vai doer recapitular. Cada estado de um agente de jogo é implementado como uma classe única, e cada agente mantém um ponteiro para uma instância de seu estado atual. Um agente também implementa uma função membro chamada `"ChangeState"` que pode ser chamada para facilitar a troca de estados sempre que uma transição de estado for necessária. A lógica para determinar quaisquer transições de estado está contida em cada classe `"State"`. Todas as classes de estado são derivadas de uma classe base abstrata, definindo assim uma interface comum. Até agora tudo bem. Você já sabe disso.

Anteriormente neste capítulo, foi mencionado que é geralmente favorável para cada estado ter ações associadas de entrada e saída. Isso permite que o programador escreva lógica que é executada apenas uma vez na entrada ou saída do estado e aumenta significativamente a flexibilidade de uma FSM. Com esses recursos em mente, vamos dar uma olhada em uma classe base `"State"` aprimorada.

```
class State
{
      public:
            virtual ~State(){}

            //Este método é executado quando se entra no estado
            virtual void Enter(Miner*)=0;

            //Este método é chamado pela função de atualização (Update)
            //do mineiro a cada etapa de atualização
            virtual void Execute(Miner*)=0;

            //Este método é executado quando se sai do estado
            virtual void Exit(Miner*)=0;
}
```

Esses métodos adicionais só são chamados quando um `“Miner”` muda de estado. Quando ocorre uma transição de estado, o método `“Miner::ChangeState”` primeiro chama o método `“Exit”` do estado atual, depois atribui o novo estado ao estado atual e finaliza chamando o método `“Enter”` do novo estado (que agora é o estado atual). Acredito que o código seja mais claro do que palavras neste caso, então aqui está o código para o método `“ChangeState”`:

```
void Miner::ChangeState(State* pNewState)
{
      //Certifique-se que ambos os estados sejam válidos
      //antes de tentar chamar seus métodos
      assert (m_pCurrentState && pNewState);

      //Chama o método de saída do estado existente
      m_pCurrentState->Exit(this);

      //Altera o estado para o novo estado
      m_pCurrentState = pNewState;

      //Chama o método de entrada do novo estado
      m_pCurrentState->Enter(this);
}
```

Observe como um Mineiro passa o ponteiro `"this"` para cada estado, permitindo que o estado utilize a interface do Mineiro para acessar quaisquer dados relevantes.

TIP **O padrão de state design também é útil para estruturar os principais componentes do fluxo do seu jogo. Por exemplo, você pode ter um estado de menu, um estado de salvamento, um estado de pausa, um estado de opções, um estado de execução, etc.**

Cada um dos quatro possíveis estados que um Mineiro pode acessar é derivado da classe `"State"`, resultando nessas classes concretas: `"EnterMineAndDigForNugget"`, `"VisitBankAndDepositGold"`, `"GoHomeAndSleepTilRested"` e `"QuenchThirst"`. O ponteiro `"Miner::m_pCurrentState"` pode apontar para qualquer um desses estados. Quando o método `"Update"` de `"Miner"` é chamado, ele por sua vez chama o método `"Execute"` do estado atualmente ativo com o ponteiro `"this"` como parâmetro. Essas relações de classe podem ser mais fáceis de entender se você examinar o diagrama de classe UML simplificado mostrado na Figura 2.3.