



MÁQUINA DE ESTADOS: CONTROLANDO O JOGO, DO MENU À IA

WICA 2012

André Kishimoto

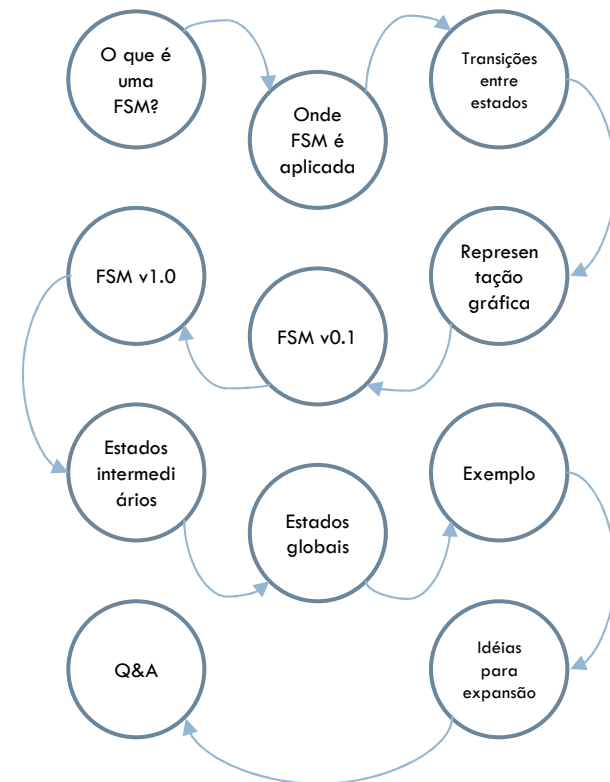
Hello, World!



- André Kishimoto
 - UNICSUL, Electronic Arts
 - andrekishimoto@yahoo.com.br

Agenda

- ❑ O que é uma FSM?
- ❑ Onde FSM é aplicada
- ❑ Transições entre estados
- ❑ Representação gráfica
- ❑ FSM v0.1
- ❑ FSM v1.0
- ❑ Estados intermediários
- ❑ Estados globais
- ❑ Exemplo
- ❑ Idéias para expansão
- ❑ Q&A



O que é uma FSM?



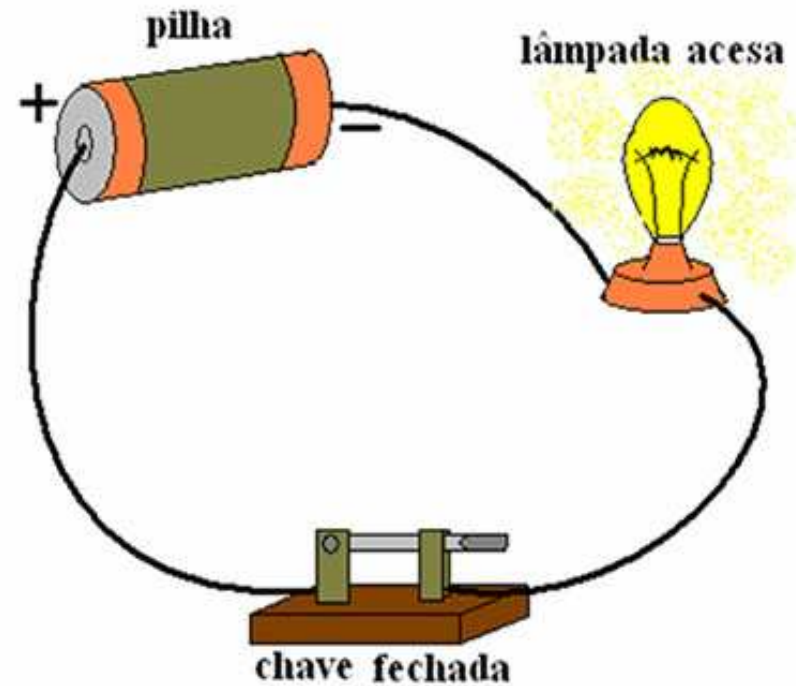
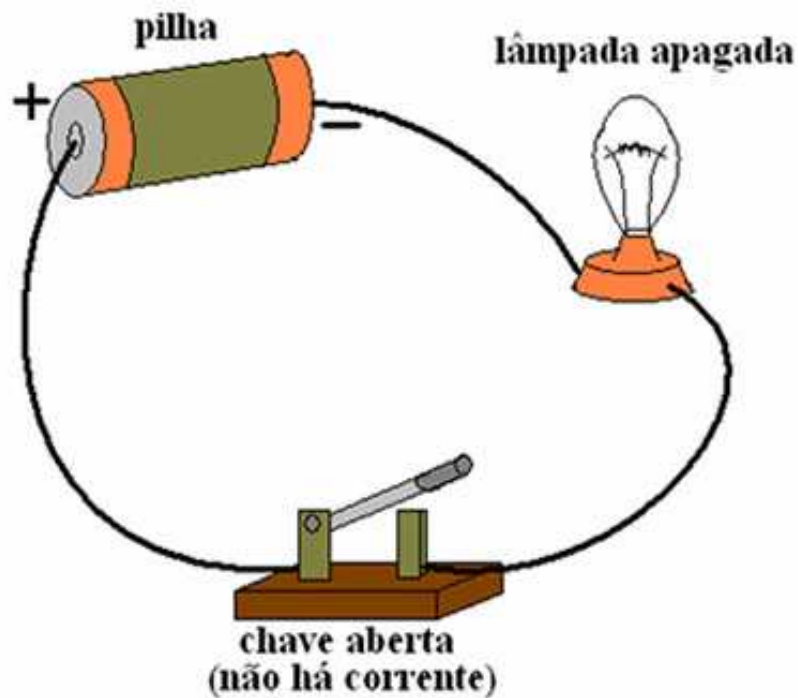
- FSM?
 - ▣ Finite State Machine
 - ▣ Máquina de Estados Finitos
- Historicamente
 - ▣ Modelo matemático usado para modelar problemas, projetar programas de computador e circuitos lógicos digitais
 - ▣ Máquina de Turing
- Máquina abstrata que possui um número finito de estados e diversas transições entre esses estados

O que é uma FSM? (cont.)

- A idéia é “quebrar” o comportamento de um objeto em estados (ou “pedaços”) facilmente gerenciáveis
- Em um dado momento, a máquina pode estar somente em um único estado
 - ▣ Current state (estado atual)
- No geral:
 - ▣ Fáceis de depurar
 - Código quebrado em pedaços
 - ▣ Não necessitam de muito processamento
 - If-else
 - ▣ Flexíveis
 - Novos estados e transições podem ser adicionados à FSM

O que é uma FSM? (cont.)

- `bool lampadaAcesa = false;`
- `lampadaAcesa = true;`



Onde FSM é aplicada



- Fluxo de telas (estados) do jogo
- Elementos de interface (UI)
- Personagens
 - ▣ Controlados pelo jogador
 - ▣ NPC's
- Objetos
- Fases
- Resumindo: em todo o jogo

Onde FSM é aplicada (cont.)

□ Fluxo de telas (estados) do jogo

- Splash
- Loading inicial
- Vídeo/Animação de introdução
- Menu principal
- Créditos
- Opções
- Ajuda
- Novo Jogo
- Gameplay
- Pause
- Game over
- Fim de jogo
- Loading entre fases/telas
- Lobby
- Etc.



Onde FSM é aplicada (cont.)

- Elementos de interface (UI)
 - ▣ Textos, Botões, Links, Campos de texto, Menus, Etc.

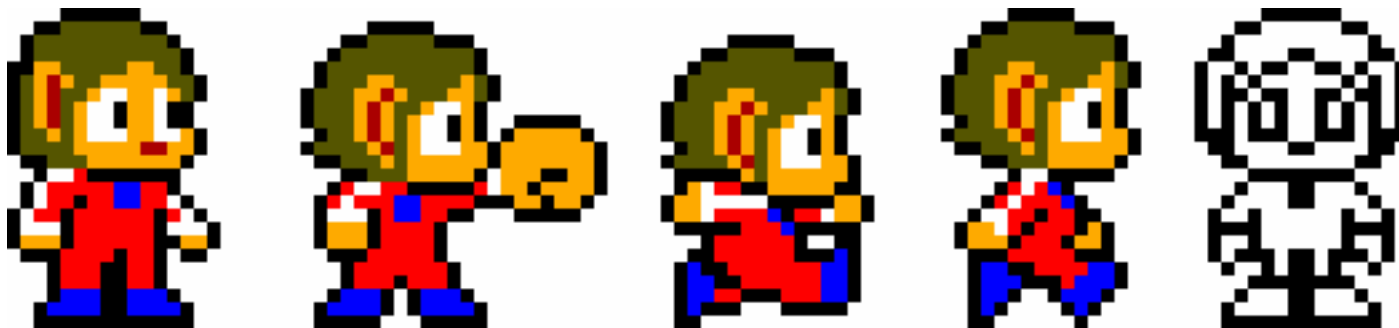
- ▣ Ativado
- ▣ Desativado
- ▣ Selecionado
- ▣ Não selecionado
- ▣ Animado
- ▣ Vísivel
- ▣ Etc.



Onde FSM é aplicada (cont.)

□ Personagens

- Idle/Parado
- Andando
- Correndo
- Pulando
- Golpeando
- Morto
- Etc.



http://www.sprisers-resource.com/other_systems/alexxiddmw/sheet/10911

Onde FSM é aplicada (cont.)

□ Objetos

- ▣ Estático
- ▣ Animado
- ▣ Quebrado
- ▣ Visível
- ▣ Invisível
- ▣ Etc.



<http://activeden.net/item/breakable-crate-prefab/1585898>

Onde FSM é aplicada (cont.)

- Fases
 - ▣ Início de fase
 - ▣ Fim de fase
 - ▣ Etc.



| POS | DRIVER | BEST LAP | TOTAL TIME |
|-----|--------------|-----------|------------|
| 1 | chris148apps | 01:28.040 | 03:08.154 |
| 2 | L. Potter | 01:30.710 | +00:01.232 |
| 3 | T. Lim | 01:31.163 | +00:02.450 |
| 4 | L. Romeo | 01:30.950 | +00:02.705 |
| 5 | A. Moppel | 01:30.925 | +00:02.885 |
| 6 | H. Shilton | 01:31.585 | +00:05.933 |

(Importância dos estados em GUI)

- GUI: Graphical User Interface
- Nem todas as libs/API's/SDK's/frameworks possuem suporte nativo de componentes GUI, sendo necessário desenvolver tais componentes
- Flash
 - ▣ ex. Button (Up, Over, Down)
- Java desktop
 - ▣ AWT/Swing
- OpenGL/DirectX/XNA
 - ▣ ???
- FSM para cada componente
 - ▣ Set/get: ativado/desativado, mouse over/out, clique down/up, focus/blur, etc.

Transições entre estados



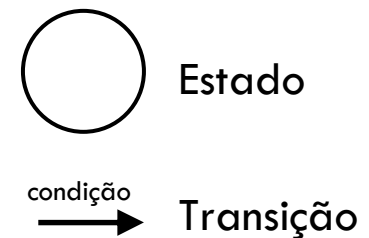
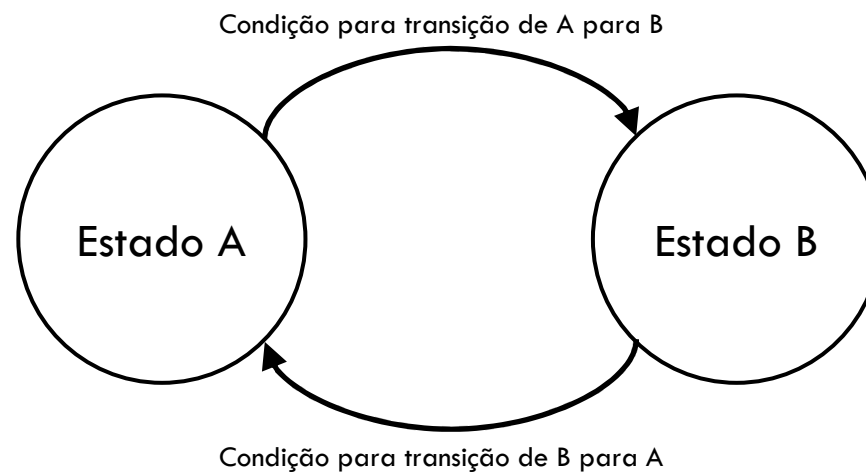
- Relação entre dois estados, indicando que o objeto sai do estado atual e passa para o novo estado em resposta a um evento ou condição
 - ▣ Lâmpada acesa em decorrência da chave fechada (ativando passagem de corrente)
- Define de qual estado para qual estado a máquina deve mudar

Transições entre estados (cont.)

- Tabela de transições de estados
 - ▣ Mecanismo para organizar estados e suas transições

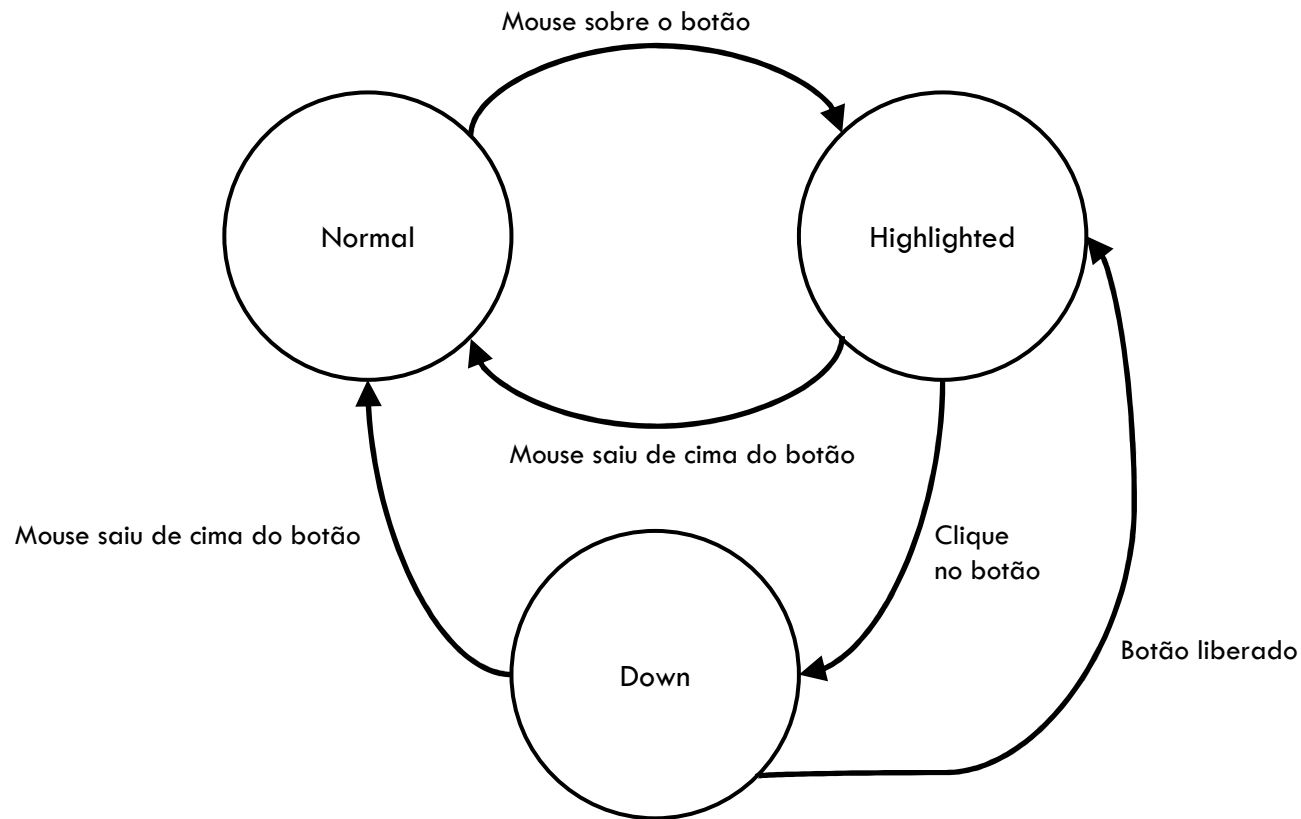
| Estado atual | Condição | Transição para estado |
|--------------|---|-----------------------|
| Patrulhando | Inimigo à vista && mais forte que inimigo | Atacando |
| Patrulhando | Inimigo à vista && mais fraco que inimigo | Fugindo |
| Atacando | Mais fraco que inimigo | Fugindo |
| Atacando | Inimigo morto | Patrulhando |
| Fugindo | São e salvo | Patrulhando |

Representação gráfica



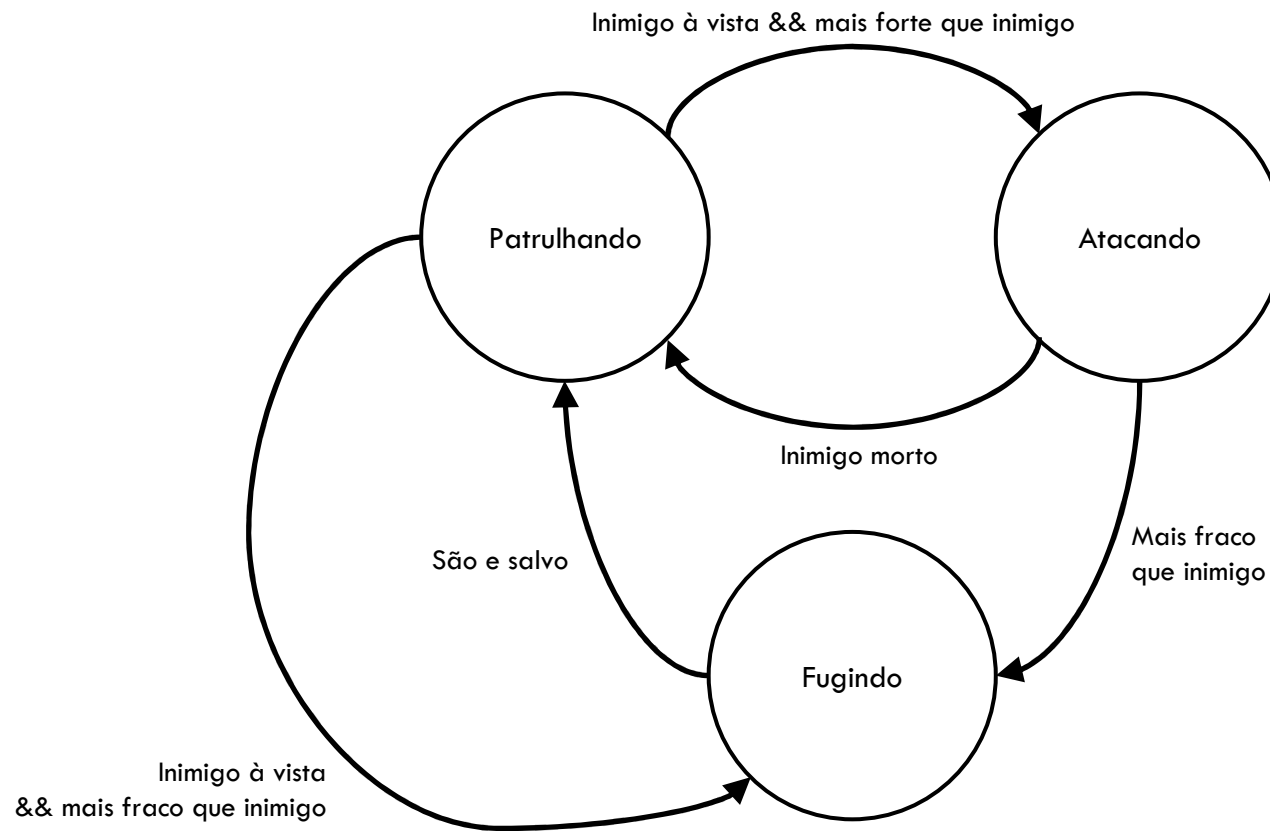
Representação gráfica (cont.)

□ Exemplo: botão



Representação gráfica (cont.)

□ Exemplo: personagem



FSM v0.1

□ Uso de if-else e switch-case

```
// Enumera os estados e define estado atual como "Patrulhando"
enum FSM_States { Patrulhando, Atacando, Fugindo };
FSM_States currentState = Patrulhando;

// Verifica o estado atual da FSM e faz transições de acordo com as condições apresentadas
switch (currentState)
{
    case Patrulhando:
        if (inimigoAVista())
        {
            if (maisForteQue(inimigo))
                estadoAtual = Atacando;
            else
                estadoAtual = Fugindo;
        }
        break;
}
// continua no próximo slide
```

FSM v0.1 (cont.)

```
// continuação do slide anterior
case Atacando:
    if (matou(inimigo))
        currentState = Patrulhando;
    else if (maisForteQue(inimigo) == false)
        currentState = Fugindo;
    break;

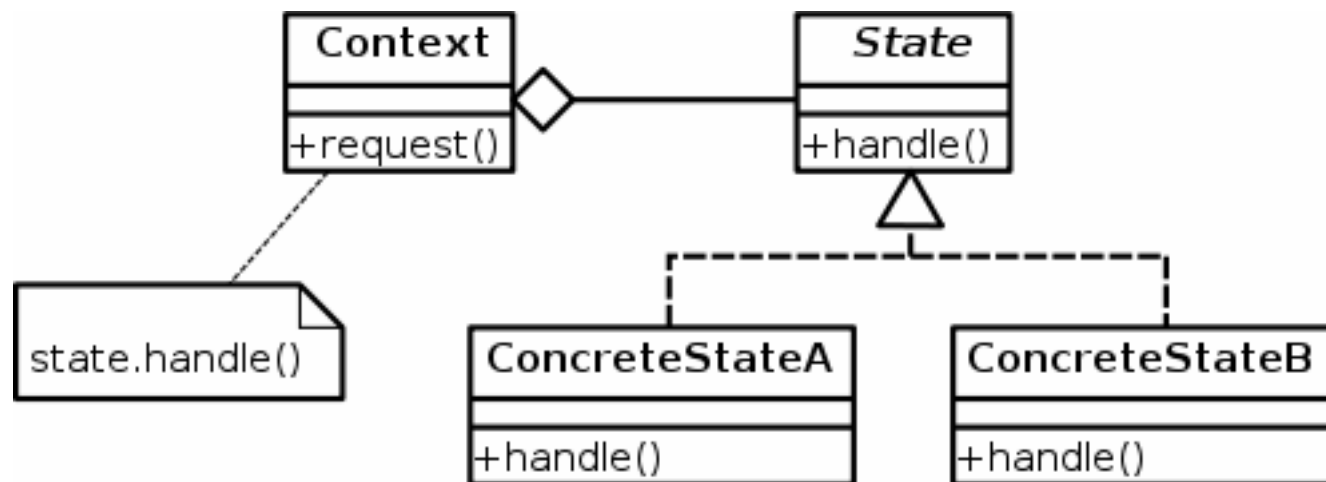
case Fugindo:
    if (saoESalvo())
        currentState = Patrulhando;
}
```

FSM v0.1 (cont.)

- Straight forward
 - ▣ Para FSM's bem simples...
- Alguns problemas
 - ▣ Manutenção e flexibilidade
 - Desastres e dor-de-cabeça
 - ▣ Spaghetti code (bagunça!)
 - Muitos if-else's e switch's aninhados
 - ▣ Transição direta
 - Não há ações de entrada e saída para cada estado (enter/exit – veremos em breve)

FSM v1.0

- Existe um design pattern (padrão de projeto) comportamental chamado State



FSM v1.0 (cont.)

- Padrão de Projeto:

- ▣ State

- Objetivo:

- ▣ Permitir que um objeto altere seu comportamento quando seu estado interno é alterado.

- Aplicação:

- ▣ Use o padrão de projeto State em um dos seguintes casos:

- O comportamento de um objeto depende de seu estado e precisa mudar seu comportamento em tempo de execução, dependendo do estado
 - Há códigos de decisão complexos e extensos que dependem do estado do objeto

FSM v1.0 (cont.)

- Padrão de Projeto:

- ▣ State

- Vantagens:

- ▣ Fácil de localizar responsabilidades de estados específicos, remove if's e switch's monolíticos
 - ▣ Mudanças de estado explícitas (getters/setters ao invés de associar valores à variáveis)
 - ▣ Estados (objetos do tipo State) podem ser compartilhados
 - ▣ Facilita a expansão de estados

- Desvantagens:

- ▣ Aumenta o número de classes (subclasses) e é menos compacto que uma única classe

FSM v1.0 (cont.)

```
// Interface para encapsular o comportamento associado a um estado particular de Context
public abstract class IState
{
    public abstract void Handle();
}

// Cada subclasse concreta implementa o comportamento associado a um estado de Context
public class ConcreteStateA : IState
{
    public override void Handle()
    {
        // Do something...
    }
}

// continua no próximo slide
```

FSM v1.0 (cont.)

```
// continuação do slide anterior
// Cada subclasse concreta implementa o comportamento associado a um estado de Context
public class ConcreteStateB : IState
{
    public override void Handle()
    {
        // Do something different than A...
    }
}

// Mantém uma instância de uma subclasse concreta que define o estado atual
public class Context
{
    protected IState mCurrentState;

    public void Request()
    {
        mCurrentState.Handle();
    }
}
```

FSM v1.0 (cont.)



- ❑ Para ajudar e facilitar a definição do estado atual de um objeto, a criação de uma classe do tipo FSM é recomendada
- ❑ Tal classe é responsável por manter o estado atual do objeto, além de realizar as transições de estados
- ❑ Objeto contém uma instância da classe do tipo FSM

FSM v1.0 (cont.)

```
// Algumas linhas de código foram omitidas para dar ênfase no assunto
public class StateMachine // FSM
{
    protected IState mCurrentState;
    protected IState mPreviousState;

    public void SetState(IState state) {
        mPreviousState = mCurrentState;
        mCurrentState = state;
    }

    public void RevertToPreviousState() {
        SetState(mPreviousState);
        mPreviousState = null;
    }

    public void Request() {
        if (mCurrentState != null)
            mCurrentState.Handle();
    }
}
```

FSM v1.0 (cont.)

```
// Algumas linhas de código foram omitidas para dar ênfase no assunto
public class MyObject
{
    protected StateMachine mFSM;

    public void SetState(IState state) {
        mFSM.SetState(state);
    }

    public void Update() {
        mFSM.Request();
    }
}
```

FSM v1.0 (cont.)

- ❑ O método `SetState()` recebe como parâmetro um `IState state`, que é a instância de uma das classes concretas que implementam `IState`
- ❑ Para compartilhar um estado entre vários objetos, basta uma única instância do estado
 - ▣ É possível usar o padrão de projeto criacional chamado Singleton



FSM v1.0 (cont.)

- Padrão de Projeto:
 - ▣ Singleton
- Objetivo:
 - ▣ Garante que uma classe possua apenas uma instância e fornece um ponto global de acesso ao objeto
- Aplicação:
 - ▣ Use o padrão de projeto Singleton onde:
 - Deve-se existir exatamente uma instância de uma classe, que deve ser acessível pelos usuários a partir de um ponto de acesso conhecido

FSM v1.0 (cont.)

- Padrão de Projeto:
 - ▣ Singleton
- Vantagens:
 - ▣ Permite o controle sobre como e quando os usuários acessam a instância do objeto
 - ▣ Várias classes Singleton podem obedecer a mesma interface (seleção de Singleton em tempo de execução)
 - ▣ Mais flexível que métodos estáticos por permitir polimorfismo
- Desvantagens:
 - ▣ “Variável global”
 - ▣ Memory leak (C++)

FSM v1.0 (cont.)


```
public class ConcreteStateA : IState
{
    private static ConcreteStateA mInstance = null;

    public static ConcreteStateA Instance()
    {
        if (mInstance == null)
            mInstance = new ConcreteStateA();

        return mInstance;
    }

    // Resto da classe ConcreteStateA
}
```

FSM v1.0 (cont.)



```
// Exemplo de como mudar estado de um objeto com State e Singleton
// Algumas linhas de código foram omitidas para dar ênfase no assunto

MyObject objTest = new MyObject();

objTest.SetState(ConcreteStateA.Instance());
```

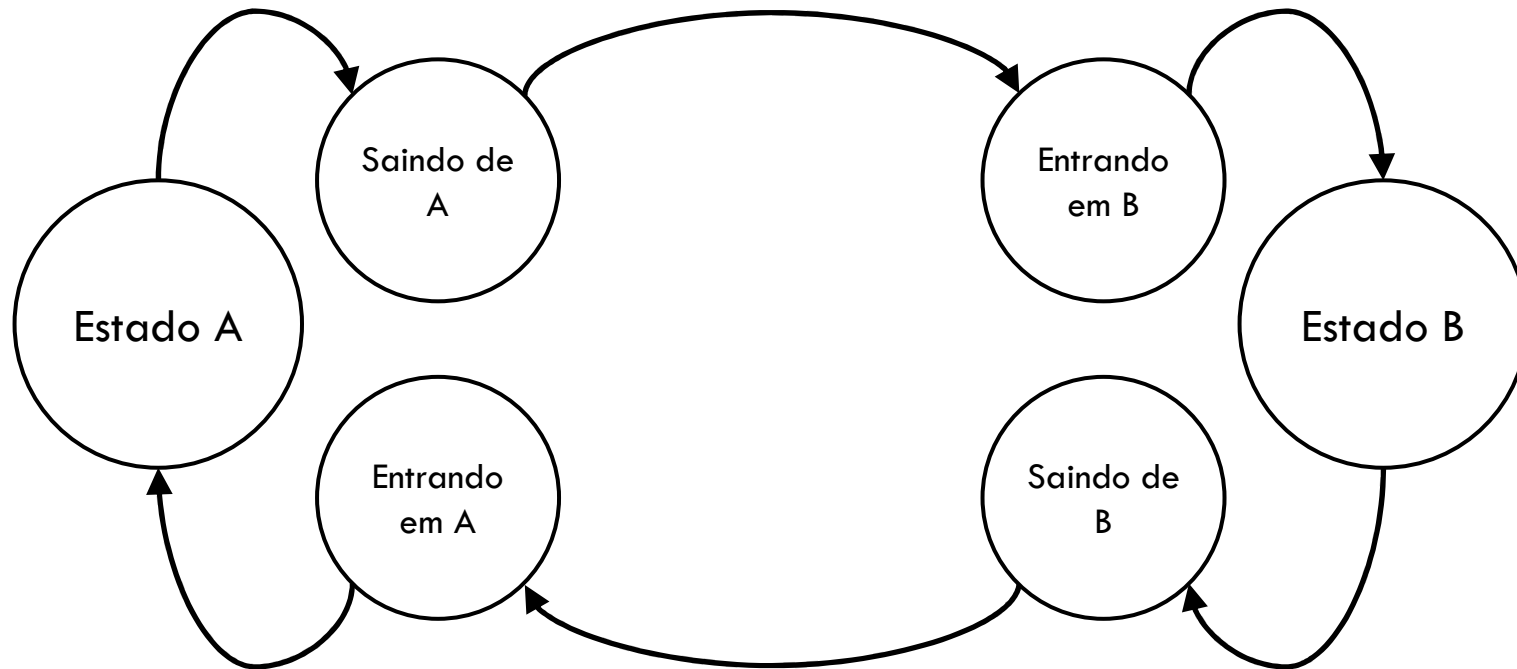
Estados intermediários

- Um dos pontos negativos do FSM v0.1 foi:
 - ▣ Transição direta
 - Não há ações de entrada e saída para cada estado (enter/exit – veremos em breve)
- O ideal é que a transição entre um estado e outro não seja direta/brusca, mas sim que exista um processo de saída do estado atual e um processo de entrada do novo estado durante a transição

Estados intermediários (cont.)

□ Representando graficamente:

Condição para transição de A para outro estado.
Define que próximo estado é B.



Condição para transição de B para outro estado.
Define que próximo estado é A.

Estados intermediários (cont.)

- Uma vez que a interface `IState` e a classe `StateMachine` já foram criadas, basta:
 - ▣ Adicionar dois métodos (`Enter()` e `Exit()`) na interface `IState`,
 - ▣ Implementá-los nas subclasses concretas e
 - ▣ Chamá-los na transição de estados (método `SetState()` da `StateMachine`)

Estados intermediários (cont.)

// Algumas linhas de código foram omitidas para dar ênfase no assunto

```
public abstract class IState
{
    public abstract void Handle();
    public abstract void Enter();
    public abstract void Exit();
}

public class StateMachine // FSM
{
    public void SetState(IState state)
    {
        mCurrentState.Exit();

        mPreviousState = mCurrentState;
        mCurrentState = state;

        mCurrentState.Enter();
    }
}
```

Estados globais

- Há certas operações que são consideradas globais, isto é, ocorrem independente do estado atual do objeto
 - ▣ Pessoa envelhece a cada momento, independente do que está fazendo
 - ▣ Repetição de código em todos os estados
- Para essas operações, é possível criar um (ou mais) estado global que sempre é executado junto com o estado atual
 - ▣ StateMachine terá um novo IState mGlobalState

Estados globais (cont.)

```
// Algumas linhas de código foram omitidas para dar ênfase no assunto
public class StateMachine // FSM
{
    protected IState mGlobalState;
    protected IState mCurrentState;
    protected IState mPreviousState;

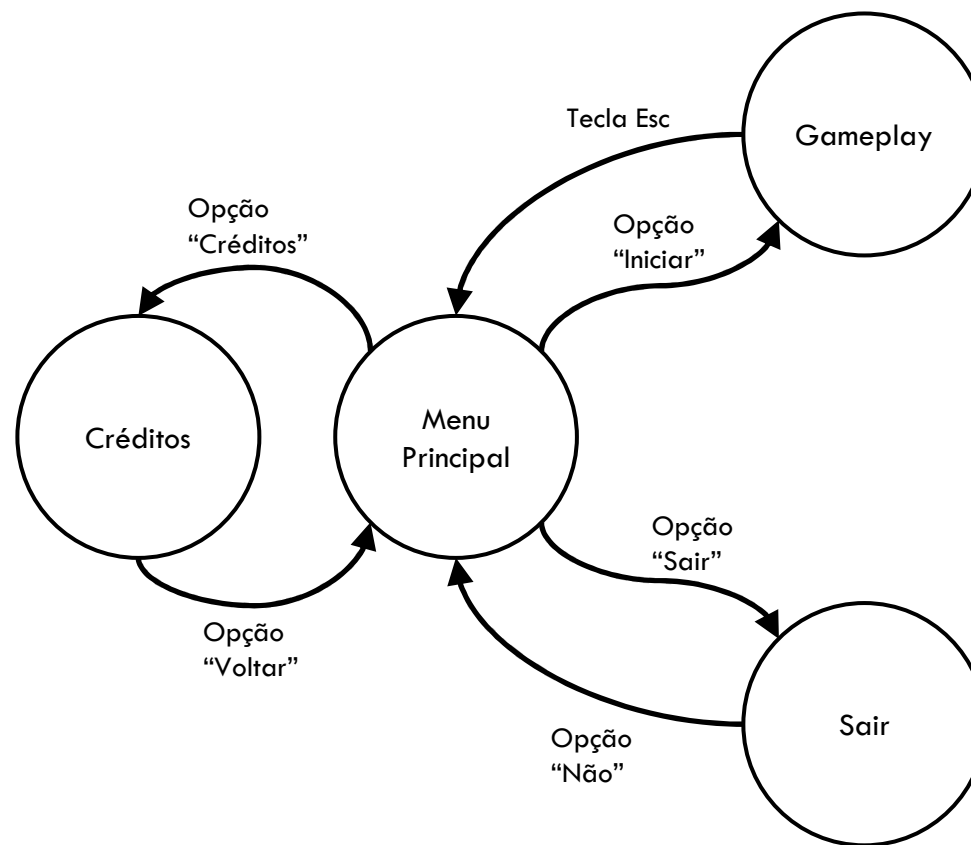
    public void SetGlobalState(IState globalState) {...}

    public void Request() {
        if (mGlobalState != null)
            mGlobalState.Handle();

        if (mCurrentState != null)
            mCurrentState.Handle();
    }
}
```

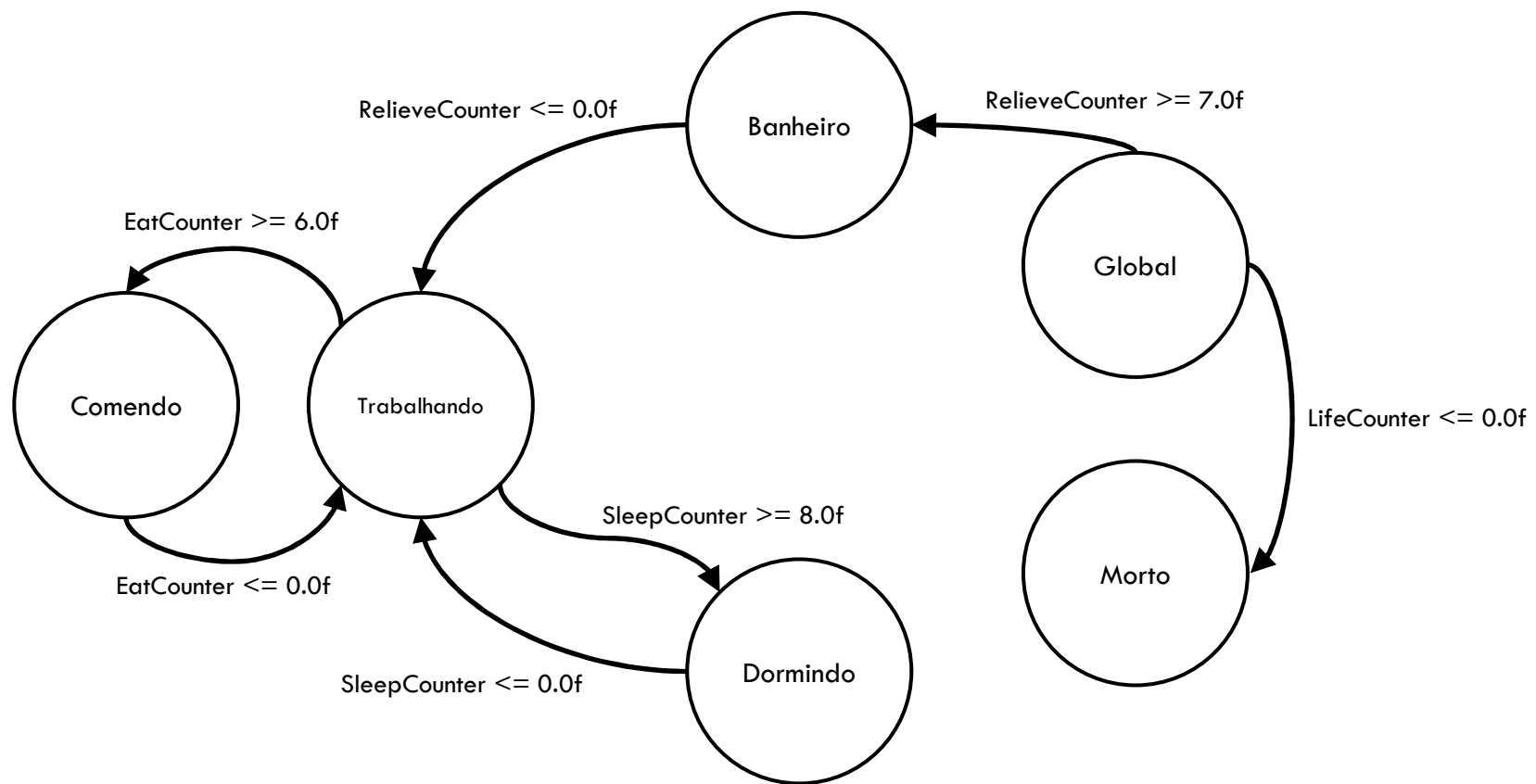

Exemplo

□ Run, Game, Run!



Exemplo (cont.)

□ Run, Game, Run!



Exemplo (cont.)

- Onde FSM é aplicada
 - ▣ Fluxo de telas (estados) do jogo – classes principais
 - ScreenManager
 - Contém StateMachine, SetState(), instância das telas e tela atual
 - GuiScreenStateMainMenu
 - GuiScreenStateCredits
 - GuiScreenStateExit
 - GuiScreenStateGame
 - ▣ Botões – classes principais
 - GuiButton
 - Contém StateMachine e SetState()
 - Cada tela possui um array de GuiButton
 - GuiButtonStateActivated
 - GuiButtonStateSelected
 - GuiButtonStateUnselected

Exemplo (cont.)

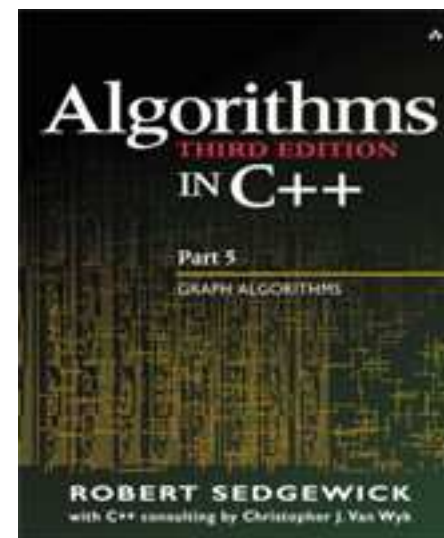
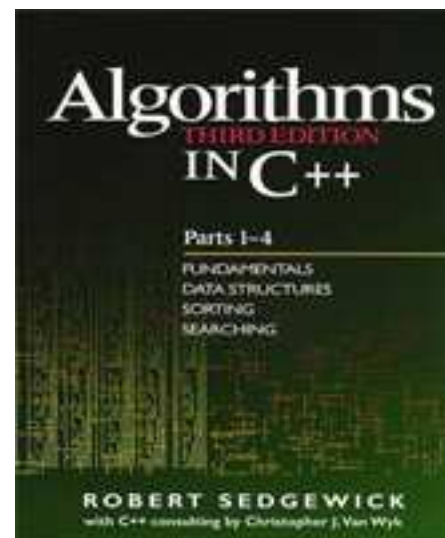
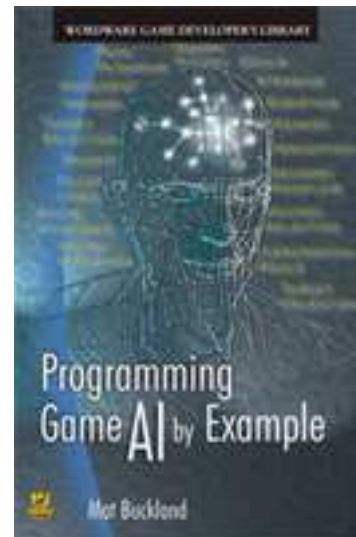
- Onde FSM é aplicada
 - ▣ NPC – classes principais
 - GameNPC
 - Contém StateMachine e SetState()
 - GameNPCStateWorking
 - GameNPCStateEating
 - GameNPCStateSleeping
 - GameNPCStateRelieving
 - GameNPCStateDead
 - GameNPCStateIdle

Idéias para expansão



- Mensagens/callback
 - ▣ Comunicação entre objetos
- Filas / filas com prioridade
 - ▣ The Sims
- Grafos
 - ▣ Transições entre estados com Teoria dos Grafos
- Script
 - ▣ Comportamento dos estados via script (Lua, Python, Javascript, etc.)

Idéias para expansão (cont.)



Q&A



- Obrigado!