State MVC: Estendendo o padrão MVC para uso no desenvolvimento de aplicações para dispositivos móveis

Tiago Barros, Mauro Silva e Emerson Espínola

C.E.S.A.R – Centro de Estudos e Sistemas Avançados do Recife

{tgfb, mjcs, ele}@cesar.org.br

Resumo. *Aplicações para dispositivos móveis podem ser implementadas para várias plataformas diferentes, como J2ME, BREW, Symbian, Windows Mobile e Embedded Linux. No entanto, apesar de diferentes, estas plataformas possuem certa semelhança em sua arquitetura, pois todas são dirigidas a eventos. O Padrão SMVC tem por objetivo capturar estas semelhanças ao propor uma arquitetura em que possamos utilizar uma máquina de estados dentro do padrão MVC.*

Abstract. *There are many different platforms for mobile application development such as J2ME, BREW, Symbian, Windows Mobile and Embedded Linux. Although different, they have some common architecture, because each platform is event-driven. The SMVC pattern has the intent to catch these common elements of all platforms by include a state machine inside MVC.*

1. Introdução

Em desenvolvimento para dispositivos móveis as funcionalidades são muito centradas em cenários de uso baseados na interação com o usuário. Tais questões estão facilmente relacionadas com aspectos de manipulação de eventos originados pelos mesmos. Esses eventos, no contexto da codificação, são observados sob as seguintes vertentes: apresentação do modelo de dados; gerenciamento e controle dos eventos; e manipulação da interface com o usuário.

O padrão de arquitetura MVC *(Model-View-Controller)* [Krasner and Pope 1998] é bastante utilizado no desenvolvimento de aplicações para dispositivos móveis pois determina a separação de uma aplicação em três elementos. O *Model* é formado por entidades que representam os dados da aplicação. A *View* tem por objetivo realizar a apresentação destes dados e capturar os eventos do usuário; sendo representada pelas telas. O *Controller* faz a ligação entre o *Model* e a *View*, realizando o tratamento dos eventos, atuando sobre o *Model* e alterando os elementos da *View* para representar a nova forma dos dados.

Neste artigo, será apresentada uma extensão do padrão MVC para o desenvolvimento de aplicações para dispositivos móveis chamado *State MVC (SMVC)*. O padrão MVC será instanciado para o contexto de aplicações para dispositivos móveis e dois níveis

Copyright (c) 2007, Tiago Barros, Mauro Silva e Emerson Espinola. Permissão de cópia concedida para a conferência SugarLoaf-PLoP 2007. Todos os outros direitos reservados.

a mais serão sugeridos para que a manipulação de eventos seja realizada de maneira mais eficiente e escalável.

O SMVC é aplicado em cenários de desenvolvimento onde a mudança de interfaces e camada de controle necessitem de rapidez e eficiência, sem que o modelo arquitetural adotado seja um entrave à mudança. O público-alvo principal deste artigo são desenvolvedores de aplicações para dispositivos móveis.

2. SMVC

2.1. Objetivo

Fornecer uma arquitetura para desenvolvimento de aplicações para dispositivos móveis baseada numa extensão do MVC para uma maior eficiência, escalabilidade e melhor escrita de código.

2.2. Contexto

Ainda que o desenvolvimento para dispositivos móveis necessite de ambientes que são orientados a eventos, eles não dispõem de uma estrutura adequada para uma programação eficiente. É comum no desenvolvimento de aplicações em plataformas desta natureza [Forman and Zahorjan 1994] - BREW, J2ME, Symbian, Embedded Linux e Windows Mobile - que o código seja confuso, mesclando em um único lugar o tratamento de todos os eventos da aplicação. Neste código então, torna-se necessário adicionar diversas *flags* de controle potencializando o número de erros.

É importante ressaltar que as aplicações de interação com o usuário - especialmente para celulares - rodam em um único processo, não permitindo assim o bloqueio de uma aplicação em detrimento de outra. Para solucionar este conflito, as plataformas utilizam uma função de *callback*[1] que faz o tratamento dos eventos enviados para a aplicação.

Muito embora esta solução seja de grande valia do ponto de vista de programação, uma única função para tratar todos eventos torna a aplicação cada vez mais complexa. Neste momento, então, é natural que apareçam mecanismos de controle para representar os diferentes estados da aplicação, por exemplo: ao receber o evento `eventX` com o `flag1` ligado (`TRUE`), uma determinada ação deve ser tomada; recebendo o mesmo evento `eventX` com `flag1` igual a `FALSE` outra ação deve ser iniciada. Percebe-se que os valores das *flags* são de fato os estados existentes na aplicação e são eles que devem ser tratados.

Desta forma, um tratamento adequado para os estados e eventos, além de tornar a programação mais simples e intuitiva, proporciona um maior desacoplamento e garante uma melhor extensibilidade e manutenabilidade da aplicação.

2.3. Problema

Como então manipular *eventos* e *estados* garantindo maior fator de produtividade (em linhas de código), facilidade na manutenção e, sobretudo, agregando escalabilidade a novas funcionalidades? Imprimir simplesmente o padrão MVC nas soluções para dispositivos móveis não garante o sucesso na implementação.

---

[1] Mecanismo utilizado para a realização de operações assíncronas. Uma função é passada como parâmetro e chamada quando a operação termina.

A adoção de padrões arquiteturais que garantem a separação eficiente entre interface e controle facilitam o tratamento dos desafios inerentes à computação móvel.

2.4. Forças

- A adição de novas funcionalidades deve ser facilitada através do desacoplamento do *Controller* e da *View*.
- Projetar a aplicação onde os estados sejam organizados como classes, com métodos para tratar cada evento.
- Deve minimizar a utilização de recursos (memória) pela aplicação, proporcionando um mecanismo para carregá-los quando necessário e liberá-los quando não estiverem mais em uso.
- A descentralização do tratamento de eventos deve ser atingida através da distribuição deste tratamento para os estados.
- A adição, remoção e modificação de estados da aplicação devem ser feitas de maneira a se evitar grande impacto na arquitetura.
- A reutilização de telas comuns deve ser garantida, através de um mecanismo que proporcione o gerenciamento destas telas.

2.5. Solução

Para resolver o problema apresentado, um padrão de projeto composto [Riehle 1997] chamado *State MVC* é sugerido. Este padrão consiste na extensão do padrão MVC, baseada na composição entre os padrões *State* [Gamma et al. 1994] e *Manager* [Sommerlad 1997] para representar o *Controller* do MVC, a fim de fornecer uma melhor manipulação e tratamento de eventos e estados. Além disto, um mecanismo para o controle de telas também baseado no padrão *Manager* implementa a *View*, proporcionando reutilização de telas e facilidade de manutenção.

2.6. Estrutura

A estrutura do SMVC é representada através do digrama de classes UML [Booch et al. 1998] da Figura 1.

Abaixo segue a descrição de cada classe participante do padrão:

- *Application*
  A classe `Application` representa o *Model* do padrão MVC. Além deste papel, esta classe também faz a interface com a plataforma alvo, representando o ponto de entrada da aplicação e possuíndo métodos de inicialização (`StartApp`) e finalização da aplicação (`StopApp`), bem como os métodos para pausar (`PauseApp`) e continuar (`ResumeApp`) a mesma.

- *StateManager*
  A classe `StateManager` representa uma máquina de estados, sendo responsável pela transição dos estados da aplicação e por chamar o método do estado que trata cada evento recebido pela aplicação. Dentro do padrão MVC, esta classe, junto com os estados propriamente ditos, representa o *Controller*.

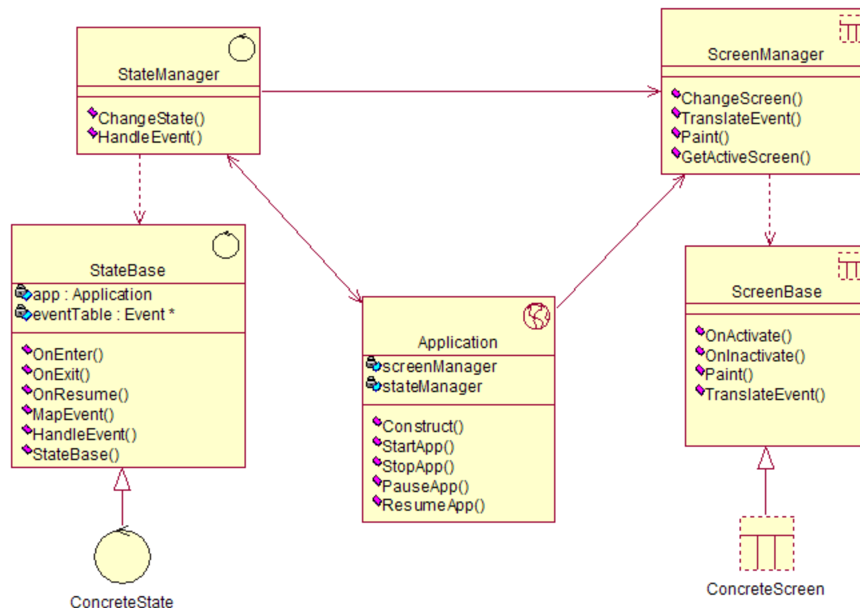

Figura 1. Diagrama de classes do padrão SMVC

- *StateBase*
  `StateBase` representa um estado da aplicação. Esta é uma classe abstrata, da qual todos os estados concretos devem herdar.

- *ConcreteState*
  Deve ser criada uma classe `ConcreteState` para cada estado da aplicação. Cada `ConcreteState` deve implementar um método para cada evento a ser tratado pelo estado.

- *ScreenManager*
  `ScreenManager` é responsável por gerenciar as telas da aplicação. Esta classe, junto com as telas propriamente ditas, representam a *View* do padrão MVC.

- *ScreenBase*
  A classe `ScreenBase` é uma classe abstrata da qual todas as telas da aplicação devem herdar. Ela possui métodos para exibição dos dados da aplicação na tela do dispositivo.

- *ConcreteScreen*
  Cada tela da aplicação é uma `ConcreteScreen`. Esta classe herda de `ScreenBase` e deve implementar seus métodos abstratos.

2.7. Dinâmica

A Figura 2 mostra o diagrama de seqüência da construção de uma aplicação que utilize o padrão SMVC. A aplicação tomada por exemplo implementa dois estados (`StateA` e `StateB`) e uma tela (`Screen1`), que são classes concretas (implementações de `ConcreteState` e `ConcreteScreen`) que herdam de `StateBase` e `ScreenBase`, respectivamente.

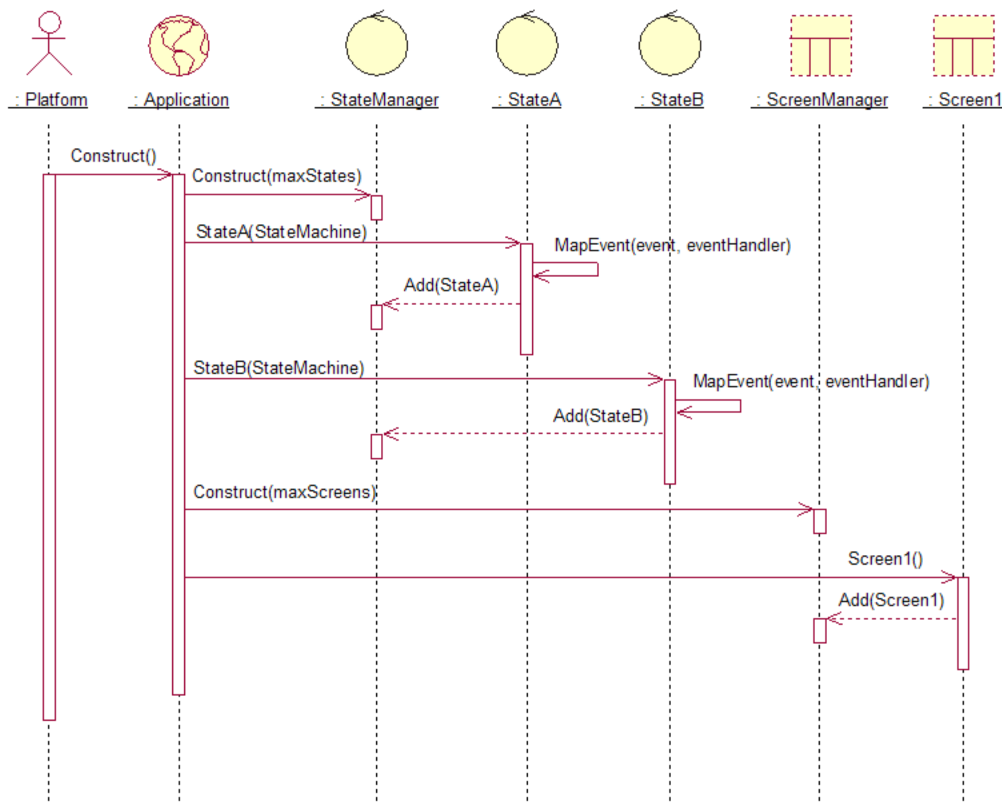

Figura 2. Construção da aplicação

O primeiro método chamado, ao inicializar a aplicação, é o método `Construct` da classe `Application`. A chamada deste método deve estar integrada com a plataforma de desenvolvimento escolhida de forma que ele seja chamado na inicialização da aplicação. Este método é responsável por instanciar os dados da aplicação, bem como todos os estados e telas.

Cada estado criado, é responsável por definir quais os eventos que ele vai tratar e quais os métodos responsáveis por tratar cada evento, através do metodo `MapEvent`.

Na Figura 3 podemos ver a seqüência de inicialização da aplicação. Depois de todos os estados e telas da aplicação serem instanciados, o método `StartApp` da classe `Application` é chamado, o qual deve definir o estado inicial do `StateManager`.

Ao definir o estado inicial, o método `OnEnter` deste estado é chamado, devendo inicializar os dados necessários ao estado e definir qual será a tela apresentada, através do método `ChangeScreen` de `ScreenManager`. O método `ChangeScreen` é responsável por chamar o método `OnInactivate` da tela anterior (caso exista uma

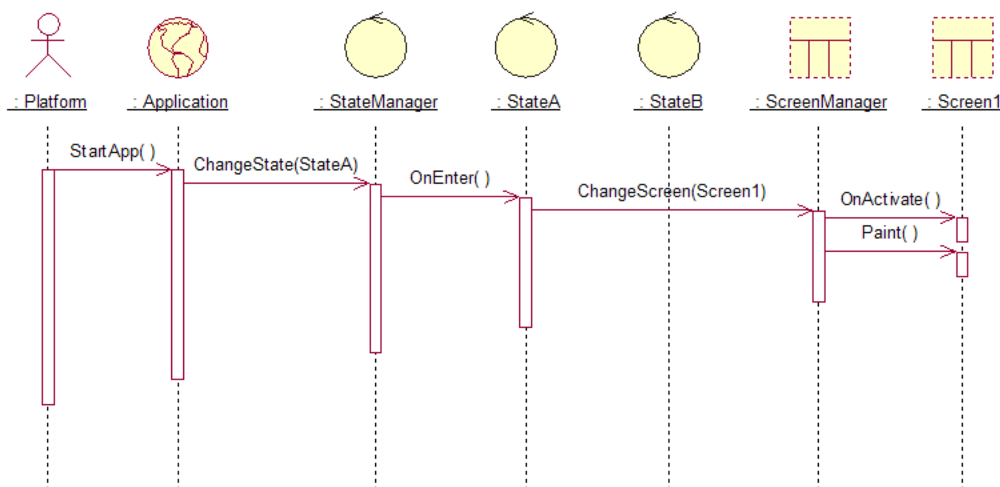

Figura 3. Inicialização da aplicação

tela anterior), para que seus dados sejam removidos da memória, e chamar o método `OnActivate` da tela atual, para que os dados desta tela sejam criados na memória. Depois disto, será chamado o método `Paint` para que a tela atual seja desenhada.

Depois desta inicialização, a aplicação aguardará por eventos para serem tratados pelo estado atual. A Figura 4 mostra o diagrama de seqüência para dois exemplos de tratamento de eventos pela aplicação, um evento de OK, e um evento de EXIT. A seqüência do tratamento de qualquer outro evento é análoga a estes mostrados.

Quando a aplicação recebe um evento, o método `HandleEvent` de `StateManager` é chamado. Este método verifica qual é o estado atual da aplicação e envia este evento para ser tratado, chamando o método `HandleEvent` deste estado. No estado, caso o evento seja um evento de tecla ele será traduzido para um evento significativo, de acordo com a tela que está sendo apresentada. No exemplo do primeiro evento, se a tecla pressionada for a *softkey* da esquerda e, na tela, esta tecla representa a função OK, o evento de *softkey* da esquerda será traduzido para OK.

Depois de traduzido, o estado irá verificar qual é o método responsável por tratar este evento e irá chamá-lo para que execute. A execução do método tratador do evento poderá acarretar em alterações no modelo ou na visualização da aplicação, através de chamadas de métodos de `Application` e `ScreenManager`, respectivamente. Também é possível mudar o estado da aplicação, alterando o estado atual através do método `ChangeState` de `StateManager`.

2.8. Conseqüências

O SMVC oferece as seguintes vantagens:

- Código Modular
  Assim como o MVC, o SMVC desacopla o comportamento do modelo de dados e da visualização. Além disto, o próprio comportamento é modularizado ao dividir o *Controller* em um conjunto de estados.
- Extensibilidade e Manutenabilidade
  Devido ao *Controller* ser implementado como máquina de estados, alterar ou adi-

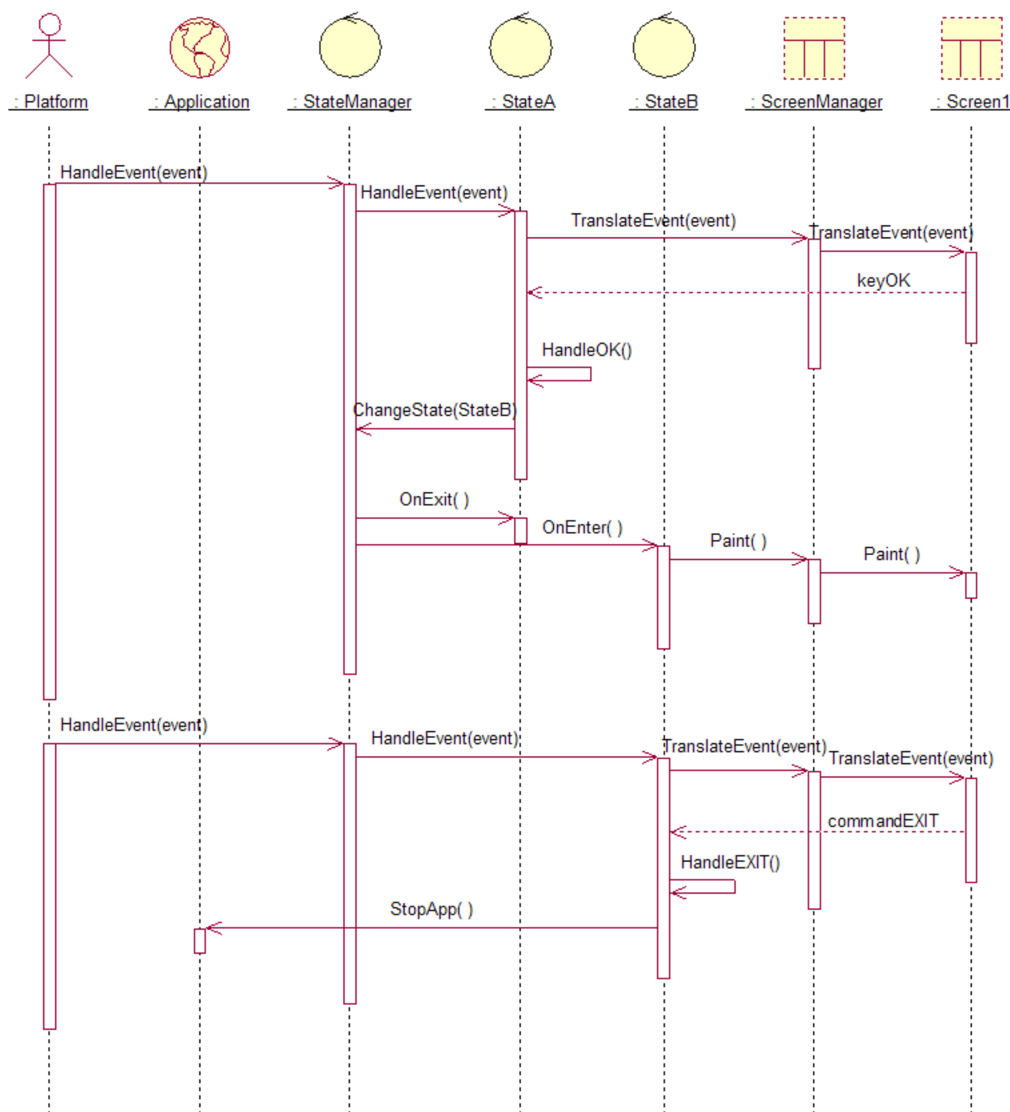

Figura 4. Dinâmica do tratamento de eventos

cionar novas funcionalidades consiste em alterar ou adicionar os estados correspondentes, minimizando bastante o impacto destas mudanças na aplicação completa.

- Reutilização de telas
  Como o código de tratamento de eventos está implementado nos estados (*Controller*), podemos utilizar uma mesma tela em vários estados, evitando a inserção de *flags* no código das telas.
- Redução da memória utilizada
  Os estados da aplicação podem carregar os dados necessários ao seu processamento quando tornam-se ativos e liberar esta memória ao tornarem-se inativos, proporcionando um melhor aproveitamento da memória do dispositivo ao evitar que os dados necessários a todos os estados estejam sempre na memória.
- Descentralização do tratamento de eventos

Os eventos são tratados por métodos específicos de cada estado, evitando-se escrever uma única função para tratar todos os eventos da aplicação.

Em consequência disto, também oferece as seguintes desvantagens:

- Aumento do número de classes
  Com a introdução de `StateManager` e `ScreenManager` além da implementação das classes concretas para os estados e telas, há um aumento no número de classes. Para aplicações pequenas, a Variação 1 (Seção 2.10) do padrão pode ser considerada.
- Duplicação de código
  Eventos tratados em vários estados podem ter seu código de tratamento duplicado na implementação dos estados. Neste caso, é sugerida a Variação 2 do padrão (Seção 2.10), que implementa os estados de forma hierárquica, reusando eventos comuns em níveis de estados intermediários.

2.9. Implementação

Para implementar o padrão SMVC, devemos seguir os seguintes passos:

1. Definir, dentro da plataforma escolhida, de que classe herdaremos a classe `Application`. Por exemplo, em J2ME [J2ME 2007], seria a classe `Midlet` e em BREW [BREW 2007] a estrutura `AEEApplet`.
2. Implementar os dois gerenciadores (`ScreenManager` e `StateManager`). Como estas classes são classes genéricas, deverão ser implementadas completamente desacopladas da aplicação, pois poderão ser reutilizadas nas próximas aplicações desenvolvidas.
3. Modelar a aplicação como uma máquina de estados, verificando quais eventos devem ser tratados por cada estado e quais telas serão apresentadas. Neste ponto, podemos definir se vamos utilizar as variações sugeridas neste artigo.
4. Implementar cada estado modelado, definindo seus métodos tratadores de eventos.
5. Implementar as telas da aplicação.

2.9.1. Exemplo

Foi escolhida a plataforma BREW para demonstração do uso do padrão SMVC. Será mostrada a implementação de uma aplicação tradicional em BREW. Depois mostraremos a implementação do padrão proposto. Esta abordagem visa realizar uma análise comparativa da utilização do SMVC.

Uma aplicação em BREW consiste nos seguintes elementos:

- *Estrutura base da aplicação*
  Deve ser criada uma estrutura para a aplicação que contenha a estrutura `AEEApplet` como primeiro elemento. Isto faz-se necessário visto que `AEEApplet` é a base para qualquer aplicação BREW e é utilizada internamente nas funções de criação da aplicação. Declarando `AEEApplet` como primeiro elemento da estrutura da nossa aplicação permite que se faça um *cast* da estrutura da nossa aplicação para a estrutura `AEEApplet`. Desta forma, pode-se passar a estrutura da nossa aplicação como parâmetro para as funções do framework de BREW.

- *Função de inicialização*
  Esta função é responsável por inicializar (alocar memória) todos os dados da aplicação.
- *Função de finalização*
  Esta função é chamada quando a aplicação termina e é responsável por desalocar toda a memória alocada na função de inicialização.
- *Função de tratamento de eventos*
  É responsável por receber todos os eventos enviados à aplicação. Geralmente é implementada como um grande *switch* que escolhe qual o tratamento adequado, de acordo com o evento recebido.

Abaixo veremos um exemplo de código da estrutura base de uma aplicação BREW, bem como o código das funções de inicialização e finalização da aplicação.

```
typedef struct _HelloWorld
{
    AEEApplet a ; // First element of this structure must be AEEApplet
    AEEDeviceInfo  DeviceInfo; // the hardware device information

    int appScreen; // holds application screen ID
} HelloWorld;


// this function is called when your application
// is starting up
boolean HelloWorld_InitAppData(HelloWorld * pMe)
{
    // Get the device information for this handset.
    pMe->DeviceInfo.wStructSize = sizeof(pMe->DeviceInfo);
    ISHELL_GetDeviceInfo(pMe->a.m_pIShell,&pMe->DeviceInfo);

    return TRUE;
}

// this function is called when your application
// is exiting
void HelloWorld_FreeAppData(HelloWorld * pMe)
{
    // insert your code here for freeing any
    // resources you have allocated...
}
```

Quando a aplicação é inicializada, a função `HelloWorld_InitAppData` será chamada para que os recursos necessários à aplicação sejam alocados. Depois disto, toda a execução da aplicação passa a acontecer na função `HelloWorld_HandleEvent`, que será mostrada a seguir.

```
static boolean HelloWorld_HandleEvent(HelloWorld* pMe,
                                      AEEEvent eCode,
                                      uint16 wParam,
                                      uint32 dwParam)
{
    // switch event code
    switch (eCode)
    {
        // App is told it is starting up
        case EVT_APP_START:
```

```
    // application goes to first screen
    pMe->appScreen = FIRST_SCREEN;
    // send an event to this app to paint the
    // screen
    ISHELL_PostEvent(pMe->a->m_pIShell,
                     HELLOWORLD_CLSID,
                     EVT_USER_REPAINT,
                     0,
                     0);

return(TRUE);

// App is told it is exiting
case EVT_APP_STOP:
// do nothing, just return TRUE
// meaning event was recognized and app
// agrees to be terminated
return(TRUE);

// A key was pressed
case EVT_KEY:
// verify current screen
if (pMe->appScreen == FIRST_SCREEN)
{
    // if key is softkey 1,
    // goto sencond screen
    if (wParam == AVK_SOFT1)
    {
        pMe->appScreen == SECOND_SCREEN;
    }
}
else if (pMe->appScreen == SECOND_SCREEN)
{
    // if key is softkey 1,
    // goto first screen
    if (wParam == AVK_SOFT1)
    {
        pMe->appScreen == FIRST_SCREEN;
    }
    // if key is softkey 2,
    // exit application
    else if (wParam == AVK_SOFT2)
    {
        ISHELL_CloseApplet(pMe->a->m_pIShell, FALSE);
    }
}
// send an event to this app to paint the
// screen
ISHELL_PostEvent(pMe->a->m_pIShell,
                 HELLOWORLD_CLSID,
                 EVT_USER_REPAINT,
                 0,
                 0);
return(TRUE);

case EVT_USER_REPAINT:
```

```
        if (pMe->appScreen == FIRST_SCREEN)
        {
            // Draw first screen
        }
        else if (pMe->appScreen == SECOND_SCREEN)
        {
            // Draw second screen
        }


        //-------------------------------------------
        // All other events comes here.
        // Once application becomes complex, this
        // function will become very big.
        //-------------------------------------------
    }

    return FALSE;
}
```

O código acima é um exemplo típico de construção de aplicações para dispositivos móveis. Podemos perceber claramente que a variável `appScreen` representa o controle da tela ativa na aplicação. No entanto, analisando o tratamento do evento `EVT_KEY`, visualizamos que esta variável também é utilizada para representar o estado da aplicação.

O crescimento de complexidade desta aplicação irá implicar na adição de mais variáveis como esta para determinar o controle das diversas situações de tela e estado. Esta adição de *flags* proporciona uma pior manutenabilidade e uma maior sucessão a erros, por parte do desenvolvedor, além do excesso de diretivas `if` para verificar estes valores.

Mostraremos abaixo, como resolver estes problemas ao aplicar o padrão SMVC na construção de aplicações para dispositivos móveis.

```
class Application : public AEEApplet
{
public:
    // Application entry point for event handling
    static bool HandleEvent(Application *app,
                            UINT16 evCode,
                            UINT16 wParam,
                            UINT32 dwParam);

    // App memory allocation
    int         Construct();

    // App memory deallocation
    static void FreeAppData(Application *app);

    // Method that is called when app starts
    void StartApp();
    // Method that is called when app ends
    void StopApp();
    // Method that is called when app is suspended
    void SuspendApp();
    // Method that is called when app resumes its
    // execution after being suspended
```

```
    void ResumeApp();

private:
    // State Manager object
    StateManager      *iStateManager;

    // Screen Manager object
    ScreenManager     *iScreenManager;

    // Application data goes here...
};
```

A classe `Application` possui a interface com a plataforma BREW, ao herdar da estrutura `AEEApplet`. Além disto esta classe deve implementar os métodos necessários a sua execução, como os métodos de alocação e desalocação da memória utilizada pela aplicação (`Construct` e `FreeAppData`), bem como o método `HandleEvent`, responsável por tratar todos os eventos recebidos.

```
bool Application::HandleEvent(Application *app,
                              UINT16 evCode,
                              UINT16 wParam,
                              UINT32 dwParam)
{
    UINT16 event, ret = FALSE;

    switch (evCode)
    {

        case EVT_APP_START:
        {
            app->StartApp();
            return(TRUE);
        }

        case EVT_APP_STOP:
        {
            app->StopApp();
            return(TRUE);
        }

        case EVT_APP_RESUME:
        {
            app->ResumeApp();
            return(TRUE);
        }

        case EVT_APP_SUSPEND:
        {
            app->SuspendApp();
            return(TRUE);
        }

        case EVT_KEY_PRESS:
        case EVT_KEY:
        case EVT_KEY_RELEASE:
        // translate the key event in the current dialog
        event = app->iScreenManager->TranslateEvent(aeCode,
```

```
                                                         awParam,
                                                         adwParam);

        // if event is not translated, use it "as is"
        ret =  app->iStateManager->HandleEvent(event,
                                               awParam,
                                               adwParam);
        app->iScreenManager->Repaint();
        return ret;

        // default: send the event to stateManager
        default:
        {
            return app->iStateManager->HandleEvent(evCode,
                                                   awParam,
                                                   adwParam);
        }

    } // switch evCode

    return(FALSE);
}
```

Nesta implementação, o método `HandleEvent` ao receber os eventos de `Start`, `Stop`, `Suspend` e `Resume`, irá chamar os métodos de `Application` responsáveis por tratá-los.

Caso seja um evento de tecla, este evento será primeiramente traduzido pelo `ScreenManager`, de acordo com a tela que está sendo apresentada, e depois enviado ao `StateManager`. Qualquer outro evento será enviado diretamente ao `StateManager`.

O método `HandleEvent` do `StateManager` será então responsável por enviar o evento ao estado ativo, para que seja tratado pelo método correspondente.

Isto descentraliza completamente o tratamento de eventos, evitando a verificação de estados e telas atuais e proporcionando uma maior modularização e extensibilidade do código. Abaixo temos o código do método `HandleEvent` do `StateManager`.

```
int StateManager::HandleEvent(UINT16 evCode,
                              UINT16 wParam,
                              UINT16 dwParam)
{
    int ret = FALSE;

    // look for current state and send event to it
    if(this->iCurrentState)
        ret = this->iCurrentState->HandleEvent(aeCode,
                                               awParam,
                                               adwParam);

    return ret;
}
```

O `StateManager` também possui o método `ChangeState`, que é responsável por alterar o estado ativo, chamando os métodos `OnExit` e `OnEnter` do estado anterior e do novo estado, respectivamente.

```
int StateManager::ChangeState(const UINT16 &aID)
```

```
{
    int ret = FALSE;

    StateBase *state = this->Get(aID);

    if (state)
    {
        // call OnExit from previous state
        if (this->iCurrentState)
            this->iCurrentState->OnExit();

        // change the state
        this->iCurrentState = state;

        // call OnEnter from new state
        this->iCurrentState->OnEnter();

        ret = TRUE;
    }
    return ret;
}
```

A implementação do `StateBase` pode ser vista a seguir. O método `MapEvent` é responsável por mapear eventos em métodos do estado. Este mapeamento pode ser feito utilizando uma tabela de eventos e métodos. Esta tabela associa cada evento tratado pelo estado a um método e pode ser consultada posteriormente para chamar o método desejado.

```
// Event Handler method pointer
typedef bool (StateBase::*EventHandler)(UINT16 wParam, UINT32 dwParam);

int StateBase::MapEvent(UINT16 aEventCode,EventHandler aEventHandler)
{
    int ret = FALSE;
    if ( ( iCurrNumEvents >= 0 ) &&
         ( iCurrNumEvents < this->iMaxEvents ))
    {

        iEventTable[iCurrNumEvents].iEventCode = aEventCode;
        iEventTable[iCurrNumEvents].iEventHandler = aEventHandler;
        iCurrNumEvents++;
        ret = TRUE;
    }
    return ret;
}
```

O método `HandleEvent` é responsável por verificar se o estado trata o evento recebido e chamar o método correspondente.

```
int StateBase::HandleEvent(UINT16 evCode,
                           UINT16 wParam,
                           UINT32 dwParam)
{
    int ret = FALSE;

    for (int i=0; i<iCurrentNumEvents; i++)
    {
        // search for event handler in event table
```

```
        if ((iEventTable[i].iEventCode == evCode) &&
            (iEventTable[i].iEventHandler != NULL))
        {
            // Call event handler for the event evCode
            ret = (this->*(iEventTable[i].iEventHandler))(awParam, adwParam);
        }
    }

    return ret;
}
```

O `ScreenManager` também é implementado de acordo com o padrão de projeto *Manager*. O método `ChangeScreen` é responsável por mudar a tela que está sendo exibida, bem como chamar o método `Repaint` para que a nova tela seja desenhada.

```
int ScreenManager::ChangeScreen(UINT16 aId)
{
    int ret = FALSE;
    ScreenBase *screen = this->GetActiveScreen();

    if (screen != NULL) screen->OnInactivate();

    this->iActiveScreenId = aId;

    screen = this->GetActiveScreen();

    if (screen)
    {
        ret = screen->OnActivate();

        if (ret == TRUE)
        {
            this->Repaint();
        }
    }

    return ret;
}
```

2.10. Variações

Na Variação 1, é possível alterar o padrão SMVC para que possua um único *Manager*. Desta forma, cada tela corresponderia a um único estado. Isto implica num menor número de classes e redução do *overhead*, sendo recomendado para aplicações pequenas.

A Variação 2 consiste em utilizar uma *Hierarchical State Machine* [Samek 2002] para representar o *Controller*. Pertencem a uma *super classe* de estados, os eventos que são tratados da mesma forma em vários estados. Isto faz com que os estados da aplicação herdem destes *super estados*, evitando, assim, a duplicação de código no tratamento destes eventos.

2.11. Usos Conhecidos

O padrão SMVC vem sendo utilizado no desenvolvimento de diversas aplicações para dispositivos móveis no CESAR. Por questões de confidencialidade, não podemos listar nominalmente as aplicações, no entanto é possível ter uma idéia dos domínios de aplicações que utilizaram este padrão:

- Aplicações multimídia;
- Compartilhamento de imagem;
- Sincronização de informações pessoais;
- Aplicações de *slide-show*.

2.12. Padrões Relacionados

- MVC [Krasner and Pope 1998]. O SMVC extende o MVC ao fazer a composição deste padrão com os padrões *State* e *Manager*.

- *State* [Gamma et al. 1994]. O *Controller* do SMVC é implementado como uma máquina de estados usando o padrão *State*.

- *Manager* [Sommerlad 1997]. A *View* e o *Controller* do SMVC utilizam o padrão *Manager* para gerenciar as telas e estados da aplicação.

- *Hierarchical State Machine* [Samek 2002]. Este padrão pode ser utilizado como forma de reusar eventos comuns em níveis de estados intermediários na Variação 2 desse padrão.

- *Observer* [Gamma et al. 1994]. Este padrão e o Controller do SMVC têm objetivos semelhantes. Para cada estado, ambos padrões devem modificar seu comportamento, ou seja, o comportamento de um objeto depende de um estado.

- Strategy [Gamma et al. 1994]. O SMVC e o Strategy se assemelham por terem classes relacionadas que diferem somente nos seus comportamentos. Tais comportamentos são encapsulados e são implementados como uma hierarquia de algoritmos.

Agradecimentos

Este trabalho foi suportado pelo C.E.S.A.R - Centro de Estudos e Sistemas Avançados do Recife.

Agradecemos especialmente a Alexandre Sztajnberg, nosso *shepherd*, pelos comentários e sugestões importantes que proporcionaram melhorias ao nosso padrão.

Referências

Booch, G., Rumbaugh, J., and Jacobson, I. (1998). *The Unified Modeling Language User Guide*. Reading, MA. Addison-Wesley.

BREW (2007). Qualcomm brew - binary runtime environment for wireless. Disponível em http://www.qualcomm.com/brew/.

Forman, G. H. and Zahorjan, J. (1994). The challenges of mobile computing. *IEEE*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Object-Oriented Software*. Addison-Wesley.

J2ME (2007). Sun - java 2 micro edition. Disponível em http://java.sun.com/j2me/.

Krasner, G. and Pope, S. (1998). A cookbook for using the model view controller user interface paradigm in smalltalk-80. In *Journal of Object-Orientated Programming*, volume 1(3), pages 26–49.

Riehle, D. (1997). Composite design patterns. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 218–228, New York, NY, USA. ACM Press.

Samek, M. (2002). *Practical Statecharts in C C++*. CMP Books.

Sommerlad, P. (1997). The manager pattern. In Martin, R., Riehle, D., and Buschmann, F., editors, *Pattern Languages of Program Design 3*. Addison-Wesley.