5
Descrição da Implementação

A implementação do middleware declarativo *Maestro* teve início com a adaptação do Formatador HyperProp (Rodrigues, 2003), originalmente implementado na linguagem Java (Rodrigues, 2003), ao contexto da TV Digital. Para atender aos objetivos deste trabalho, discutidos no Capítulo 1, o formatador adaptado foi implementado na linguagem C++, utilizando uma abordagem orientada a objetos e visando plataformas que utilizam o sistema operacional Linux.

Os módulos Filtro de Seções, Sintonizador e DSM-CC, foram modelados de acordo com as discussões do capítulo anterior. Porém, a implementação desses módulos foi realizada pelo laboratório Lavid da Universidade Federal da Paraíba, dentro da proposta do middleware procedural FlexTV , por ela desenvolvido. A integração desses módulos ao middleware declarativo *Maestro* faz parte do processo de integração definido no projeto SBTVD. Os sub-módulos Gerenciador de Bases Privadas e Gerenciador DSM-CC foram modelados de acordo com as discussões realizadas no capítulo anterior, mas suas implementações foram deixadas para trabalhos futuros.

Como prova de conceito o middleware declarativo proposto foi integrado ao perfil simples de terminal de acesso definido pelo SBTVD, denominado Kalaheo, com a seguinte configuração: processador Celeron 700 MHz e 128 MB de memória RAM. O middleware foi também integrado a um terminal de acesso (Geode) de configuração inferior: processador AMD 233 MHz e 64 MB de RAM. A implementação do middleware com todas as suas funcionalidades foi testada em um computador pessoal, simulando um terminal de acesso de grande poder computacional: processador Pentium IV de 3.1 GHz e 1 GB de memória RAM.

As seções seguintes discorrem sobre as bibliotecas utilizadas pelo *Maestro* e a implementação do Núcleo de sua arquitetura modular, bem como dos seus exibidores. A versão atual dos módulos Núcleo e Exibidores do middleware *Maestro* conta com aproximadamente 280 classes. Maiores detalhes da

implementação podem ser encontradas em: http://www.telemidia.puc-rio.br/~marcio.

5.1. Bibliotecas Utilizadas

Além de DirectFB, discutida no Capítulo 2, foram utilizadas, na implementação do middleware *Maestro*, as seguintes bibliotecas: liburi, libxerces-c, liblua, libxine, libfusionsound, libpng, libjpeg e libmbrowser. Todas as bibliotecas utilizadas são livres e de código aberto.

A biblioteca liburi[9] oferece uma abstração para resolução de endereços dos documentos NCL a serem entregues ao Núcleo. Desenvolvida na linguagem C, a biblioteca foi projetada com o objetivo de obter um tempo mínimo de resposta e demandar pouco espaço no disco rígido. A função principal da biblioteca consiste em transformar uma URI (*Universal Resource Identifier*) específica em uma estrutura em C e vice-versa. A estrutura em C possui um campo para cada componente de uma URI: *scheme*, *path*, *query*, *params* e *host*.

Para realizar o *parser* de documentos NCL é utilizada a biblioteca libxerces-c[10], uma biblioteca portável e implementada na linguagem C++. Essa biblioteca foi desenvolvida com o objetivo de oferecer funcionalidades para a interpretação, validação, manipulação e criação de documentos XML. O *parser* da biblioteca libxerces-c foi desenvolvido com foco no desempenho, modularidade e escalabilidade.

Com o objetivo de interpretar códigos procedurais escritos na linguagem Lua, foi incorporada a biblioteca liblua[11]. Essa biblioteca é implementada como uma pequena biblioteca de funções C, escritas em ANSI C, que compila sem modificações na maioria das plataformas conhecidas. Os objetivos da implementação são simplicidade, eficiência, portabilidade e baixo impacto de inclusão em aplicações.

[9] http://www.senga.org

[10] http://xml.apache.org/xerces-c/

[11] http://www.lua.org

Normalmente, as decodificadoras MPEG-2 dos terminais de acesso para TV Digital não possuem capacidade para decodificar o programa principal e um objeto de vídeo simultaneamente. Dessa forma, para permitir a exibição de objetos de vídeo em paralelo com a exibição do programa principal decodificado por hardware, é utilizada a biblioteca libxine[12]. Essa biblioteca permite também que vídeos (inclusive do programa principal) sejam decodificados em plataformas que não possuem hardware de decodificação MPEG. A biblioteca libxine é implementada na linguagem C e possui capacidade para decodificar objetos de mídia conformes aos padrões MPEG-1 (ISO, 1993) e MPEG-2 (ISO, 2000a), entre outros (Xine, 2006).

A biblioteca libfusionsound[13] é utilizada para decodificar objetos isolados de áudio (i.e. que não foram multiplexados com objetos de vídeo). A biblioteca é implementada na linguagem C e possui capacidade para decodificar objetos de áudio concomitantemente com a apresentação de outro objetos, incluindo o vídeo principal.

As bibliotecas libpng e libjpeg são utilizadas para decodificar imagens estáticas nos formatos PNG (*Portable Network Graphics*) e JPEG (*Joint Photographic Experts Group*), respectivamente. Ambas foram implementadas na linguagem C e, geralmente, são oferecidas nos pacotes de instalação dos sistemas Linux.

Finalmente, a biblioteca libmbrowser foi desenvolvida neste trabalho com o objetivo de integrar um exibidor de objetos HTML ao middleware *Maestro*. Essa biblioteca é baseada no navegador links[14], que foi adaptado ao contexto da TV Digital, como será discutido na Seção 5.3.4. O navegador links foi implementado na linguagem C e é capaz de interpretar páginas HTML 4.0, com suporte a CSS e JavaScript.

[12] http://www.xinehq.de

[13] http://www.directfb.org

[14] http://links.twibright.com

5.2. Núcleo

A estrutura central da implementação atual do middleware *Maestro* é representada pelo diagrama de classes apresentado na Figura 16. A entidade principal do middleware implementado é a classe `Formatter`, que corresponde ao módulo Núcleo da arquitetura do middleware *Maestro*, definida no capítulo anterior. Note, através da Figura 16, que um objeto dessa classe possui, obrigatoriamente, um escalonador (classe `FormatterScheduler`) e um adaptador (sub-sistema `Adapters`), que correspondem, respectivamente, aos sub-módulos Escalonador e Adaptador. Além disso, um objeto `Formatter` também mantém um contêiner (classe `Container`), discutido na Seção 4.5 e, opcionalmente, um layout (classe `FormatterLayout`), que é mantido pelo sub-módulo Gerenciador de Layout. Finalmente, para cada modelo (ou linguagem) de documentos que o objeto `Formatter` aceita como entrada (descrevendo os programas a serem apresentados), deve haver uma implementação de conversor (sub-sistema `Converters`) para o modelo de execução. Dessa forma, conversores de documentos NCL para o modelo NCM e do modelo NCM para o modelo de execução do Núcleo foram implementados.

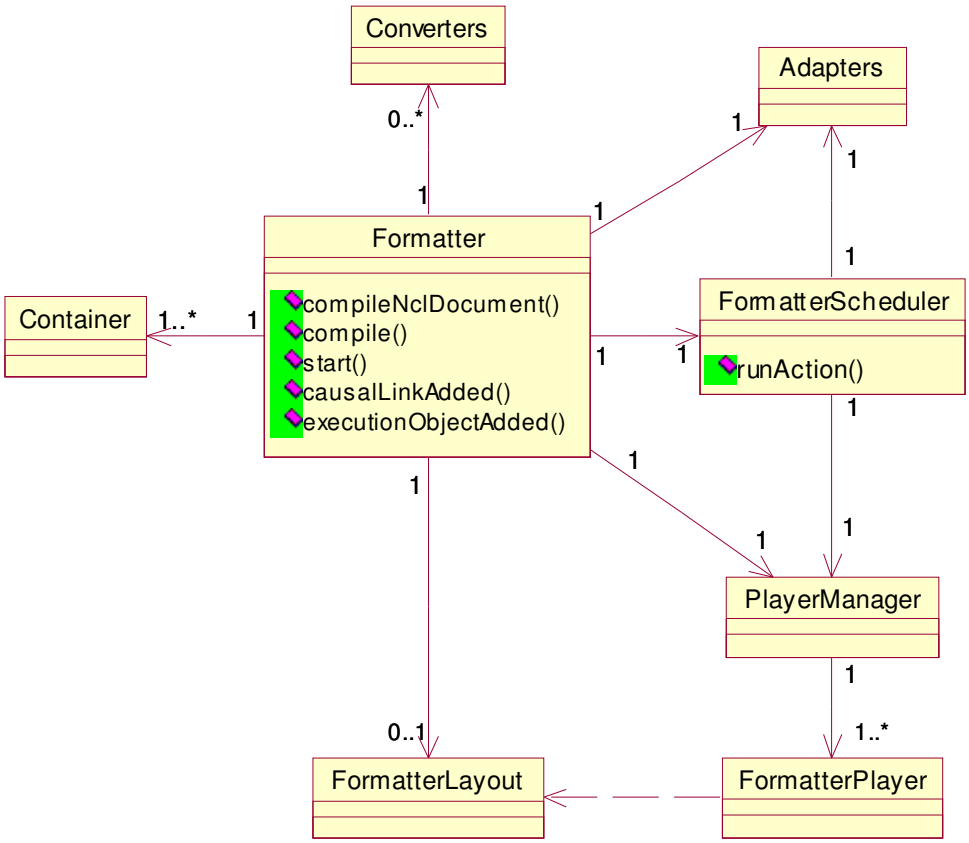

Figura 16: Diagrama de Classes central do Núcleo do *Maestro* com principais métodos.

A primeira etapa na apresentação de um documento NCL é a chegada de uma solicitação para o início da sua exibição. Essa solicitação, que atualmente é proveniente de uma interface externa ao middleware *Maestro*, consiste em chamadas a três métodos do objeto `Formatter`: *compileNclDocument()*, na qual a descrição do documento (ou parte dele) é passada como parâmetro; *compile()*, na qual um contêiner é passado como parâmetro; e *start()*.

O método *compileNclDocument()* é responsável por retornar um contêiner, resultado da tradução da especificação NCL para o modelo de execução do Núcleo. Essa tradução é realizada pelo sub-sistema Conversores. O contêiner obtido é passado como parâmetro na chamada ao método *compile()*.

O método *compile()* é responsável por incluir no container do objeto `Formatter`, todas as entidades presentes no contêiner recebido (possivelmente: elos, objetos de execução e layout). O objeto `Formatter` é um observador do seu contêiner. Assim, cada elo causal (elos que possuem uma condição específica que, quando satisfeita, implica no disparo de uma ação) (Rodrigues, 2003) adicionado no contêiner do objeto `Formatter`, gera uma notificação. Essa notificação

consiste em uma chamada a um método do objeto `Formatter`, denominado *causalLinkAdded()*. Esse método é responsável por registrar o objeto `FormatterScheduler` como observador desses elos. Além disso, cada objeto de execução inserido no container do formatador gera uma notificação, que consiste em uma chamada a um método do objeto `Formatter`, denominado *executionObjectAdded()*, passando como parâmetro o objeto de execução adicionado. Nesse método, são avaliadas, caso existam, as alternativas de objetos de execução que podem ser resolvidas estaticamente (antes de iniciar a apresentação). As alternativas de objetos de execução são associadas a regras que devem ser avaliadas. Dessa forma, para avaliação das alternativas, o serviço de um objeto do sub-sistema `Adapters` é requisitado. Como as regras podem depender de parâmetros definidos na plataforma do terminal de acesso (capacidade de processamento, recursos disponíveis etc.), assim como preferências do usuário telespectador, o adaptador consulta o contexto de apresentação através de um componente denominado `ContextInfoProxy`. Na implementação corrente, esse *proxy* de informações contextuais é uma implementação simples de gerente de contexto que mantém informações da plataforma, carregadas a partir de arquivos textos locais.

Ainda no método *executionObjectAdded()*, se houver um descritor associado ao objeto de execução (recebido como parâmetro), o objeto `FormatterLayout` pode ser solicitado para converter as informações obtidas das janelas e regiões do layout do documento NCL para superfícies (interface *IDirectFBSurface* que faz parte de uma interface *IDirectFBWindow*) e sub-superfícies (estrutura recebida através da referência *IDirectFBSurface->GetSubSurface()*) oferecidas pela biblioteca DirectFB. Além disso, no método *executionObjectAdded()* é solicitado ao objeto `PlayerManager`, apresentado no diagrama de classes da Figura 16, que crie um exibidor (classe `FormatterPlayer`, na Figura 16) apropriado para o objeto de execução (recebido como parâmetro). O objeto `PlayerManager` corresponde ao sub-módulo Gerenciador de Exibidores.

Finalmente, o método *start()* (do objeto `Formatter`) é responsável por disparar o escalonador (através de uma chamada ao método *start()* do objeto `FormatterScheduler`), passando como parâmetro um evento de apresentação para iniciar a exibição dos objetos de execução presentes no contêiner recém compilado no objeto `Formatter`.

Durante a apresentação do documento, o objeto `FormatterScheduler` permanece como um observador dos elos causais, sendo notificado sempre que uma condição for satisfeita. Essa notificação consiste em uma chamada ao método *runAction()* (do objeto `FormatterScheduler`), passando como parâmetro a ação especificada no elo causal satisfeito. Esse método é responsável por coordenar o disparo da ação (recebida por parâmetro), bem como verificar a necessidade de um objeto de execução ter o evento de apresentação de seu conteúdo preparado, ou mesmo ter sua ocorrência iniciada.

A próxima seção trata em mais detalhes a instanciação dos exibidores de mídia, durante a apresentação dos documentos NCL, bem como as particularidades na implementação realizada de exibidores para o middleware *Maestro*.

5.3. Exibidores

No middleware *Maestro* foram implementados exibidores para tratar os seguintes tipos de conteúdo: HTML, Lua, imagens estáticas (JPEG, GIF e PNG), áudio (WAVE, MPEG-1 e MPEG-2) e vídeo (MPEG-1 e MPEG-2). A Figura 17 ilustra um diagrama de classes resumido para a implementação dos exibidores.

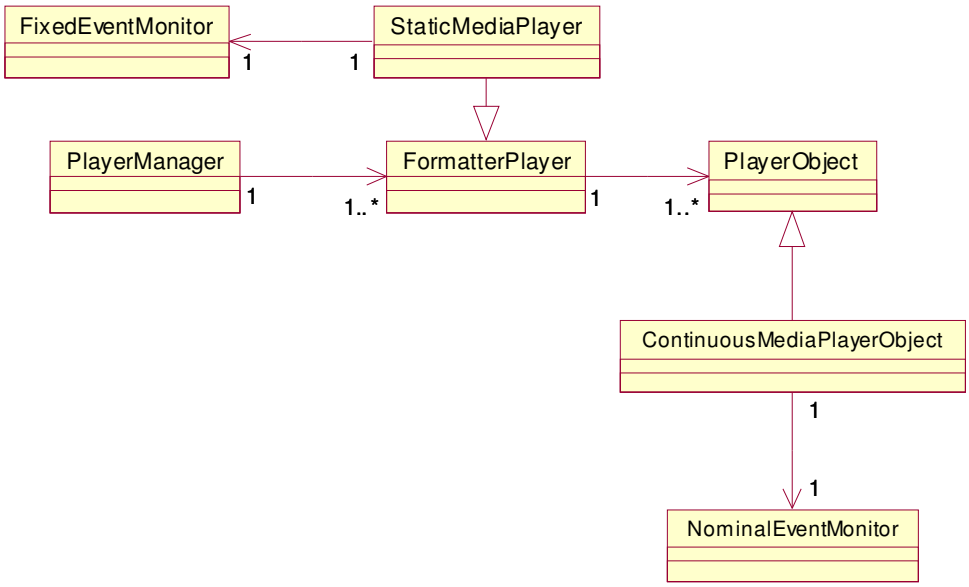

Figura 17: Diagrama de Classes para a implementação dos Exibidores.

Todo exibidor implementado deve especializar, direta ou indiretamente, os atributos e métodos da classe `FormatterPlayer`. Esses métodos permitem ao

Núcleo requisitar a execução das ações sobre eventos controlados pelo exibidor (Rodrigues, 2003). O Núcleo do middleware *Maestro* prevê que um mesmo exibidor possa ser utilizado para controlar a apresentação de mais de um objeto, esses objetos de exibição são modelados pela classe `PlayerObject`. Os exibidores de mídia contínua devem especializar a classe `ContinuousMediaPlayerObject`, que possui uma instância da classe `NominalEventMonitor`. Essa classe é responsável por monitorar temporalmente trechos do conteúdo apresentado pelo objeto que especializa a classe `ContinuousMediaPlayerObject`. Já os exibidores de mídia estática devem especializar a classe `StaticMediaPlayer`, que possui uma instância da classe `FixedEventMonitor`. Essa classe é responsável por monitorar eventos específicos na apresentação do objeto que especializa a classe `StaticMediaPlayer`.

A forma com que os exibidores foram implementados e como utilizam o diagrama de classes ilustrado na Figura 17 é discutida nas seções a seguir.

5.3.1. Áudio e Vídeo

O exibidor de áudio e vídeo foi implementado através das classes `AVPlayerAdapter`, `AVPlayerObject` e `AVPlayer`.

A classe `AVPlayer` foi implementada de forma a reconhecer um dispositivo de decodificação MPEG-2 (i.e. decodificação MPEG-2 por hardware) e controlar a exibição de fluxos de áudio e vídeo por esse dispositivo. Além disso, a classe `AVPlayer` foi implementada de tal forma a tratar também a decodificação e exibição de fluxos MPEG por software. Isso significa que o exibidor de áudio e vídeo permite que fluxos (inclusive do programa principal) sejam apresentados em plataformas que não possuem hardware de decodificação MPEG, ou que vídeos e áudios possam ser exibidos paralelamente ao programa principal decodificado por hardware. Evidentemente, a resolução dos vídeos decodificados por software irá depender da capacidade de memória e processamento do terminal de acesso.

Instâncias `AVPlayer`, que processam fluxos audiovisuais decodificados por software, utilizam a interface *IDirectFBVideoProvider* (da biblioteca DirectFB) para renderizar o conteúdo visual do fluxo na superfície definida através do método *setSurface()* (superfície definida na camada gráfica do modelo de

apresentação). A interface *IDirectFBVideoProvider* foi integrada à biblioteca libxine para decodificar os objetos MPEG. Para iniciar a exibição do fluxo decodificado, um método denominado *playVideo()* é utilizado.

Instâncias `AVPlayer` que processam fluxos audiovisuais por hardware possuem um funcionamento mais complexo. Para possibilitar a exibição do fluxo de áudio e vídeo principal decodificado por hardware simultaneamente aos outros exibidores, foram utilizadas as funcionalidades de uma API do sistema operacional Linux, conhecida como Video4Linux (Cox, 2000).

No contexto da API Video4Linux, todo o conteúdo gerado pela decodificação por hardware é enviado para um dispositivo denominado "dispositivo de vídeo" (*video*) (Cox, 2000). Esse dispositivo é utilizado, através de chamadas definidos pela API Video4Linux, para o controle de dispositivos de entrada e saída `ioctl` (Cox, 2000). Nesse caso específico a chamada para iniciar o uso do dispositivo é `open("/dev/video", O_RDONLY)`, que retorna um número inteiro que funciona como um descritor. No entanto, para exibir as informações no dispositivo de vídeo, é necessário abrir para utilização um outro dispositivo denominado "saída do dispositivo de vídeo" (*vout*), através da chamada `open("/dev/vout", O_RDONLY)`, que também retorna um descritor representado por um número inteiro.

Uma vez aberto o dispositivo *vout*, é necessário definir uma área na tela para que as informações sejam exibidas. Essa área é definida através da especificação de uma cor chave. A área utilizada na implementação é definida por uma interface *IDirectFBSurface*, possuindo a cor magenta, uma vez que essa é a cor chave padrão definida na API Video4Linux. Assim, essa superfície (que faz parte de uma *IDirectFBWindow*) consiste na camada de vídeo do modelo de apresentação.

Para conseguir sobrepor algum conteúdo sobre a camada de vídeo, é necessário, segundo a API Video4Linux (Cox, 2000), definir algumas informações denominadas de informações de *overlay*. Isso é realizado através da seguinte chamada: `ioctl(fd_vout, IOCTL_SET_OVERLAY_INFO, &ovr_info)`. Nessa chamada as informações de *overlay* são especificadas através da estrutura `ovr_info`. Resumindo, através das informações de *overlay* (cor chave da camada de vídeo, entre outras (Cox, 2000)) é definida a camada gráfica do modelo de apresentação. Porém, quando deseja-se eliminar (terminar, esconder, etc.) o conteúdo que sobrepõe a camada de vídeo, é necessário refazer as definições

dessa camada, uma vez que foi perdida a cor chave outrora presente na área utilizada pelo exibidor desse conteúdo específico. Para isso, a API Video4Linux permite definir informações de *alpha_blend* (área, forma de transição entre o conteúdo da camada gráfica e da camada de vídeo (Cox, 2000)), através da chamada: `ioctl(fd_vout, GEODE_IOCTL_SET_ALPHA_BLEND_INFO, &alp_info)`. As informações alpha_blend foram utilizadas também, na implementação da classe `AVPlayer`, para redimensionamento da área de apresentação do vídeo exibido na camada de vídeo.

O adaptador de áudio e vídeo (classe `AVPlayerAdapter`) foi implementado com o objetivo de compatibilizar o exibidor de áudio e vídeo implementado (classe `AVPlayer`), com a API de troca de mensagens entre os módulos Exibidores e Núcleo (Rodrigues, 2003). Entre as principais funções dos adaptadores de áudio e vídeo estão: sinalizar ao Núcleo eventos de apresentação (início da exibição da região de uma âncora, fim da exibição da região da âncora etc.) e executar ações sobre a apresentação como, por exemplo, aumentar volume do áudio, suspender a apresentação, reiniciar, acelerar etc., comandadas pelo Núcleo.

O adaptador de áudio e vídeo é uma especialização da classe `FormatterPlayer`, podendo controlar um ou mais objetos de exibição de áudio e vídeo (instâncias da classe `AVPlayerObject`, que especializa a classe `ContinuousMediaPlayerObject`).

Conforme discutido na Seção 5.2, durante a apresentação de um documento NCL, quando um exibidor apropriado deve ser selecionado para exibir um objeto de execução, o objeto `PlayerManager` é consultado. O passo seguinte é solicitar ao exibidor (instância de uma classe que herde de `FormatterPlayer`) a preparação da apresentação do objeto, através de uma chamada ao método *prepare()*. Esse método é responsável, entre outras coisas (Rodrigues, 2003), pela criação de uma instância de objeto de exibição de áudio e vídeo (através do método *createPlayerObject()* do adaptador).

O objeto de exibição de áudio e vídeo (`AVPlayerObject`) cria, em seu método construtor, um objeto `AVPlayer`, passando como parâmetro uma referência (obtida através do objeto de execução) para o fluxo de áudio e vídeo a ser exibido. Além disso, o objeto de exibição de áudio e vídeo pode definir no objeto `AVPlayer` (através do método *setSurface()* implementado) uma região (i.e. uma superfície) especificada pelo descritor obtido também através do objeto de

execução. Uma vez preparada a exibição, as ações de apresentação de iniciar (*start*), pausar (*pause*), retomar (*resume*) e encerrar (*stop* ou *abort*) a apresentação são mapeados do adaptador de áudio e vídeo para o objeto `AVPlayer` de forma trivial.

O objeto de exibição de áudio e vídeo monitora temporalmente a apresentação do conteúdo (através da classe `NominalEventMonitor`). Para isso, uma pthread (Nichols et al, 1996) é criada para observar e sinalizar ocorrências temporais passadas inicialmente como parâmetro para o monitor. Fazendo uso de chamadas ao método *getMediaTime()* do objeto de exibição de áudio e vídeo, esse monitor observa se um determinado trecho teve sua exibição iniciada ou terminada, para sinalizar tal fato ao Núcleo. A classe `AVPlayerObject` implementa essa chamada simplesmente consultando a instância de `AVPlayer` através de um método com a mesma assinatura.

5.3.2. Imagens Estáticas

O exibidor de imagens estáticas foi implementado através das classes `ImagePlayer` e `ImagePlayerObject`. Para utilizar esse exibidor, as imagens estáticas podem ser enviadas no mesmo carrossel que o documento NCL é transmitido, ou mesmo serem obtidas via canal de retorno. Nos testes da implementação, as imagens foram dispostas diretamente no sistema de arquivos.

A classe `ImagePlayer` foi implementada como uma especialização da classe `StaticMediaPlayer`, podendo instanciar e controlar um ou mais objetos de exibição de imagens estáticas (instâncias da classe `ImagePlayerObject`, que especializa a classe `PlayerObject`).

Para iniciar a exibição de uma imagem estática, é utilizado o método *start()* de um objeto `ImagePlayer`, que recebe como parâmetro um objeto de execução. Nesse método é criada uma instância que implementa a interface *IDirectFBImageProvider*, através da biblioteca DirectFB. Essa instância é utilizada para renderizar (na camada gráfica do modelo de apresentação) a imagem estática, cuja referência é obtida através do objeto de execução, na região (sub-superfície) especificada pelo descritor obtido, também, através do objeto de execução passado como parâmetro. No processo de renderização, a interface

IDirectFBImageProvider utiliza as bibliotecas libpng e libjpeg para decodificar a imagem estática especificada.

Para finalizar a exibição de uma imagem estática, é utilizado o método *stop()* de um objeto `ImagePlayer`, responsável por liberar os recursos utilizados na exibição. Existe ainda a possibilidade do autor, na especificação de um documento NCL, definir um tempo explícito de exibição da imagem estática. Nesse caso, é função do objeto `FixedEventMonitor` controlar esse tempo e sinalizar para que a exibição seja encerrada quando decorrida duração.

5.3.3. Lua

O exibidor Lua foi implementado através das classes `LuaPlayer` e `LuaPlayerObject`. Para utilizar o exibidor Lua, um arquivo, contendo código procedural especificado nessa linguagem, pode ser enviado no mesmo carrossel que o documento NCL é transmitido. No entanto, nos testes da implementação, arquivos contendo código Lua foram dispostos diretamente no sistema de arquivos. Além disso, na especificação do documento NCL um nó de mídia do tipo "Lua", deve fazer referência a esse tipo de arquivo específico. Esse nó pode referenciar funções Lua através de atributos.

A classe `LuaPlayer` foi implementada, inicialmente, como uma especialização da classe `StaticMediaPlayer`, podendo instanciar e controlar um ou mais objetos de exibição Lua (instâncias da classe `LuaPlayerObject`, que especializa a classe `PlayerObject`). Ao ser instanciado, um objeto de exibição Lua solicita a iniciação do ambiente Lua (liblua), caso o mesmo não esteja iniciado.

Para iniciar a interpretação do código procedural Lua, é utilizado o método *start()* de um objeto de exibição Lua. Existe a possibilidade de exibir componentes gráficos relacionados com a interpretação do código Lua. Assim, o código procedural Lua pode utilizar uma interface *IDirectFBFont* (da biblioteca DirectFB). A exibição é realizada em uma região (sub-superfície), na camada gráfica do modelo de apresentação, também especificada pelo código procedural Lua. As chamadas às funções Lua existentes no código procedural interpretado podem ser realizadas através de elos. Esses elos podem conter atributos a serem passados como parâmetros para as funções Lua.

Para finalizar o ambiente Lua, bem como a possível exibição do resultado da interpretação de um código Lua específico, é realizada uma chamada ao método *stop()* de um objeto `LuaPlayer`. Esse método tem o objetivo de fazer com que o exibidor Lua demande o mínimo de recursos possíveis, finalizando o ambiente Lua e liberando os recursos utilizados na possível exibição do resultado. Existe ainda a possibilidade do autor, na especificação de um documento NCL, definir um tempo explícito de exibição do resultado. Nesse caso, é função do objeto `FixedEventMonitor` controlar esse tempo e sinalizar para que a exibição seja encerrada quando decorrida duração.

5.3.4. HTML

O navegador links[15] foi a base para implementação do navegador HTML do middleware *Maestro*. Entretanto, o código do navegador links precisou ser alterado, porque sua implementação não permite que uma aplicação usuária especifique a região exata na tela onde a janela do navegador deve ser aberta. Além disso, o navegador links não foi implementado como uma aplicação do tipo multi-aplicação (i.e. aplicações que permitem que a interface gráfica seja compartilhada por outras aplicações). Para resolver essas limitações, foram realizadas as seguintes modificações: cada instância do navegador passou a consistir em uma nova *pthread*; e a região que o navegador utiliza passou a poder ser passada como parâmetro, utilizando a classe `FormatterLayout`.

O exibidor HTML foi implementado através das classes `TextPlayer` e `TextPlayerObject`. Para utilizar o exibidor HTML, um nó do documento NCL, deve ser do tipo "*text*".

De forma análoga aos exibidores de mídia estática, discutidos nas seções anteriores, a classe `TextPlayer` foi implementada como uma especialização da classe `StaticMediaPlayer`, podendo instanciar e controlar um ou mais objetos de exibição HTML (instâncias da classe `HTMLPlayerObject`, que especializa a classe `PlayerObject`).

[15] http://links.twibright.com/

Para iniciar a exibição de uma página HTML, é utilizado o método *start()* de um objeto `TextPlayer`, que recebe como parâmetro um objeto de execução. Nesse método é criada uma instância do navegador, através da biblioteca libmbrowser, passando como parâmetro a URL, obtida através do objeto de execução, bem como uma região (sub-superfície) onde o navegador será exibido, especificada pelo descritor, também obtido através do objeto de execução.

O método *stop()* de um objeto `TextPlayer` pode ser utilizado para finalizar a exibição de uma página HTML. Esse método é responsável por liberar os recursos utilizados na exibição da página. Caso o autor, na especificação de um documento NCL específico, defina um tempo explícito de exibição da página HTML, é função do objeto `FixedEventMonitor` controlar a exibição da página, como discutido nas seções anteriores.

5.3.5. Eventos do Usuário Telespectador

Conforme discutido na Seção 5.2, um objeto `FormatterLayout`, presente no diagrama de classes da Figura 18, pode ser solicitado para converter as informações de posicionamento espacial, obtidas dos próprios objetos de execução, para superfícies (controladas pela classe `FormatterWindow`) e sub-superfícies (controladas pela classe `FormatterRegion`) oferecidas pela biblioteca DirectFB. Ao ser instanciada, uma janela `FormatterWindow` solicita a um gerenciador de eventos (classe `EventManager`, apresentada no diagrama de classes da Figura 18) uma referência para um *buffer* responsável por manter eventos gerados pelo usuário telespectador. Esse gerenciador de eventos é obtido através do método *getInstance()*, uma vez que a classe `EventManager` foi implementada como *singleton*. Para isso, um objeto `EventManager` mantém um *buffer* (*IDirectFBEventBuffer*, introduzido no Capítulo 3), responsável por receber eventos de todos os dispositivos de entrada (teclado, mouse, controle remoto etc.) presentes na plataforma.

Quando uma exibição é iniciada, pelo método *start()* de uma das especializações de `FormatterPlayer`, discutidas nas seções anteriores, um controlador de eventos do usuário telespectador (classe `KeyHanler`, apresentada no diagrama de classes da Figura 18) é instanciado. Esse objeto especializa a classe `KeyListener` (segundo o diagrama de classes da Figura 18), e cadastra-se

no gerenciador de eventos como *listener* de eventos do usuário telespectador, através do método *addKeyEventListener()* do objeto `EventManager`.

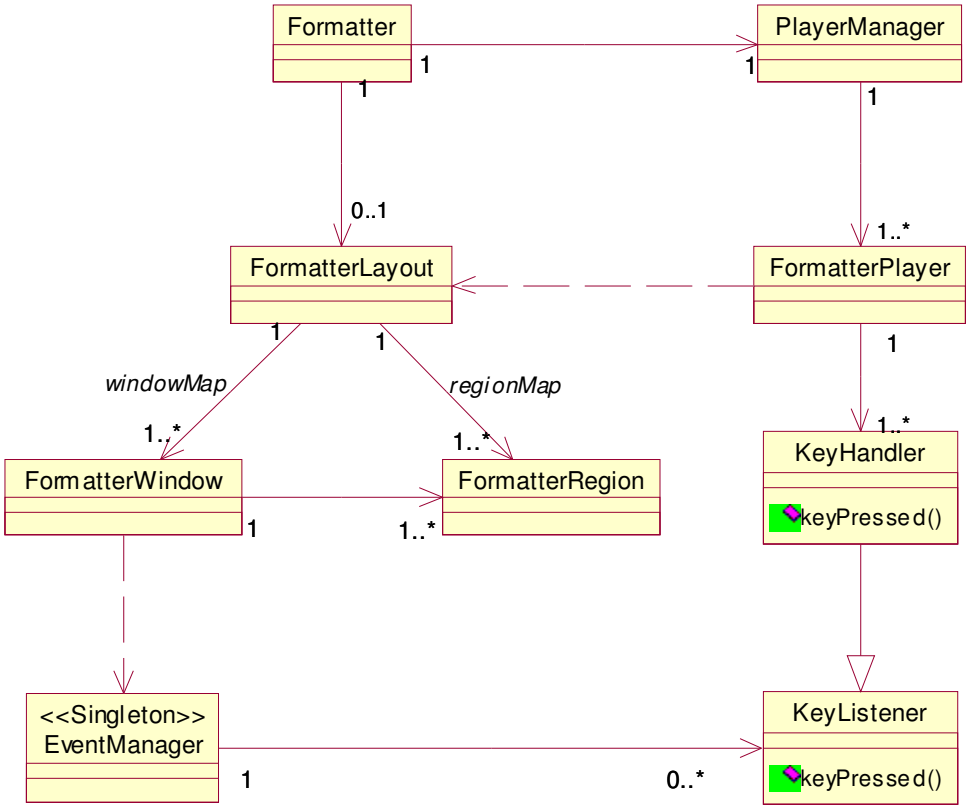

Figura 18: Diagrama de Classes do Gerenciador de Layout.

Quando uma janela `FormatterWindow` é exibida, uma *pthread* é criada para esperar eventos do usuário telespectador, que possivelmente serão gerados pelos dispositivos de entrada da plataforma, através da janela que possuir o foco (qualquer evento gerado em uma região é reportado à janela, uma vez que cada região é uma sub-superfície da superfície da janela). Essa *pthread* existe apenas enquanto a janela estiver em exibição. Ao ser gerado um evento, em qualquer janela que tiver o foco, todos os *listeners* de eventos do usuário telespectador são notificados pelo gerenciador de eventos, passando esse evento como parâmetro através de uma chamada ao método *keyPressed(),* implementado por esses *listeners* (objetos *KeyHandler*). Esse método verifica se o objeto de execução, passado como parâmetro ao instanciar um `KeyHandler`, possui alguma âncora relacionada ao evento recebido. Isso é feito para disparar um evento de seleção específico, que será tratado pelo objeto `FormatterScheduler`, conforme discussões realizadas na Seção 5.2.

Tudo isso significa que, independente de qual janela tenha o foco, todas as entidades em exibição serão notificadas do evento de usuário telespectador gerado, garantindo que as ações relacionadas a ocorrência do evento, naquele instante, sejam realizadas de acordo com o especificado na autoria.