5
Desenvolvimento da NCLite

Neste capítulo apresentamos a implementação de todos os módulos incluídos no escopo de implementação (Seção 3.7) da NCLite. Destacamos as classes principais e as tecnologias utilizadas em cada um deles. No final, exemplificamos a utilização da ferramenta implementada através do mesmo cenário de uso do Capítulo 3.

5.1
Plataforma de desenvolvimento

A linguagem de programação utilizada é *Java* e o *Eclipse Rich Client Platform* (RCP)[1] é a plataforma de desenvolvimento da ferramenta de autoria. Selecionamos essa plataforma devido à grande variedade de *plugins* disponíveis e ao melhor desempenho se comparado a outras APIs Java para interfaces gráficas, como o *Swing*.

Cada *plugin* do *Eclipse RCP* funciona como um módulo separado que é unido aos demais por um núcleo em comum. Os *plugins* têm o nome do pacote que define univocamente o componente. Como exemplo, temos o módulo do modelo de autoria nomeado como *br.pucrio.inf.serg.nclite*.

5.2
Arquitetura

A implementação é dividida em três camadas e cada uma delas contém um ou vários módulos. Sendo assim, temos a camada de conversão, a camada do modelo de autoria e a camada da interface gráfica. A Figura 5.1 apresenta a arquitetura da NCLite com as camadas mencionadas, os seus respectivos módulos e os relacionamentos entre eles.

A camada de conversão contém os módulos Parser, que gera a árvore DOM de uma aplicação NCL, e Persistência, que converte a árvore DOM para o modelo de autoria através do uso de conversores. Já a camada do modelo de autoria possui somente o módulo Modelo de Autoria com todas as classes e relacionamentos do modelo da NCLite.

[1] `http://wiki.eclipse.org/index.php/Rich_Client_Platform`

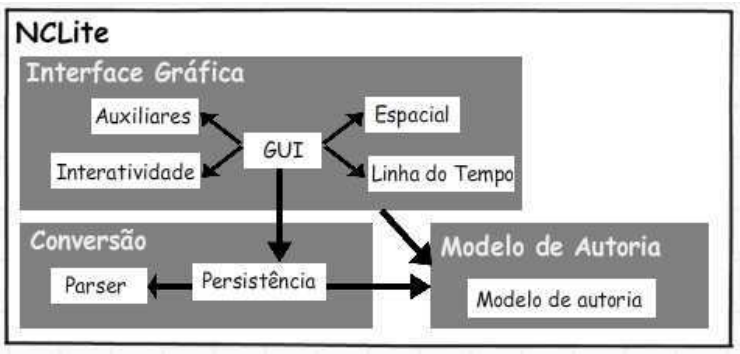

Figura 5.1: Arquitetura da NCLite

Finalmente, a camada da interface gráfica é constituída pelos módulos Componentes auxiliares, Espacial, Linha do tempo, Interatividade e GUI (do inglês *Graphical User Interface*). Todos eles tratam de componentes gráficos e o nome do módulo já deixa claro a sua função. O módulo GUI é o núcleo em comum necessário para unir os demais.

Como foi possível perceber, os módulos da implementação são separados de acordo com a definição apresentada no Capítulo 3. As exceções ficam por conta da separação da sincronização espaço-temporal (Seção 3.4.3) em dois módulos distintos para aumento da coesão do sistema e a transformação do módulo de conversão da NCL (Seção 3.4.1) em uma camada.

5.2.1 Camada de conversão

Essa camada se responsabiliza por gerar a árvore DOM de uma aplicação NCL, converter essa árvore de/para o modelo de autoria e serializar o modelo de volta para uma aplicação NCL. Os módulos Parser e Persistência contidos na camada são apresentados a seguir.

Parser

Esse módulo cria a árvore DOM que representa o documento da aplicação NCL e serializa a árvore de volta para um documento.

Utilizamos o *framework XMLBeans*[2] para gerar automaticamente um *parser* a partir de uma versão modificada do *XML Schema* da NCL. O *parser* contém classes que são mapeadas diretamente para a hierarquia do documento NCL.

O *XMLBeans* foi escolhido porque permite a navegação pela árvore tanto pelo DOM quanto por objetos Java, possui uma boa perfomance e a gama de *XML Schemas* compatíveis é consideravelmente maior que os demais.

[2] `http://xmlbeans.apache.org/`

Como mencionado, o parser gera classes Java para representar o código NCL. Cada classe tem uma correspondência um para um com os elementos da NCL. A Figura 5.2 apresenta um diagrama com classes geradas pelo *XMLBeans* para representar os elementos `<descriptor>` e `<descriptorParam>` da NCL.

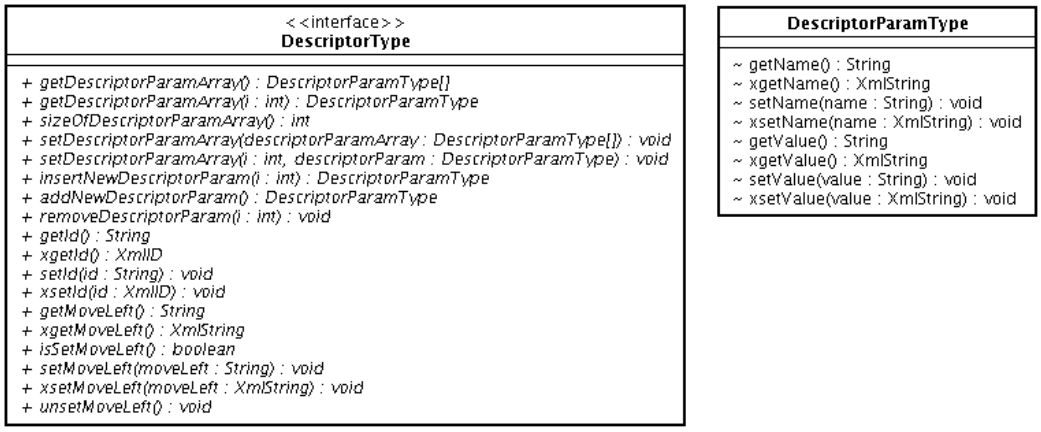

Figura 5.2: Classes que representam os elementos descriptor e descriptorParam

Como é possível perceber, um objeto do tipo `DescriptorType` possui os atributos e os filhos do elemento `<descriptor>`, sendo que esses filhos são os elementos `<descriptorParam>`, que são representados pela classe `DescriptorParamType`.

Ainda na Figura 5.2, os oito primeiros métodos de `DescriptorType` são exclusivos para manipulação dos objetos `DescriptorParamType`. Para cada conjunto de objetos filhos, existe um grupo de métodos para manipulação deles. Por exemplo, o método `getDescriptorParamArray()` retorna um vetor de `DescriptorParamType` que representa os elementos `<descriptorParam>` que o `<descriptor>` possui.

Para manipulação dos atributos obrigatórios, existe um conjunto de quatro métodos para realização da tarefa. No caso do atributo obrigatório `id`, por exemplo, temos: `getId()`, `xgetId()`, `setId(String)` e `xsetId(XmlString)`.

Já para os atributos não obrigatórios, existe uma coleção de seis métodos para manipulá-los. No diagrama, os seis útimos métodos pertencem à coleção para manipular o atributo não obrigatório `moveLeft`. Da mesma forma, apesar de ter sido omitido na figura, existem mais seis métodos para manipulação de cada atributo não obrigatório de `<descriptor>`.

Persistência

Como já mencionado, este módulo é responsável pela conversão entre os objetos gerados pelo *parser* representando o código NCL e o modelo de

autoria da NCLite. Para conseguir realizar essa conversão foram feitas diversas consultas a (Soares & Barbosa, 2009).

Utilizamos objetos chamados conversores para realizar essa conversão. A Figura 5.3 apresenta a classe `Converter` que representa um conversor e a classe `Document` que abstrai o documento NCL. No `Converter` encontramos dois métodos, o `createDomain(document)` responsável por criar um objeto do modelo de autoria a partir do documento NCL e o método `createXMLElements(document, domainObject)` que transforma o objeto do modelo em objetos do parser. Na mesma figura percebemos a existência de classes fábricas (padrão *Abstract Factory* (Gamma et al., 1995)) que auxiliam na abstração da criação dos conversores e dos documentos.

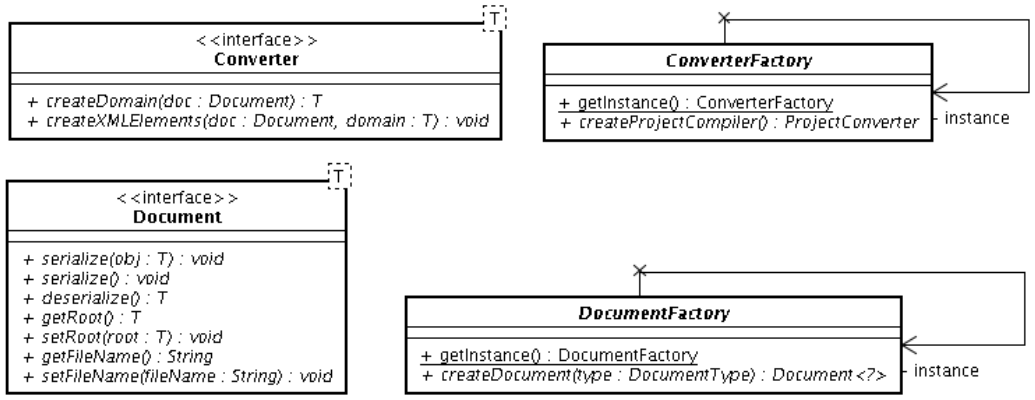

Figura 5.3: Interface de um conversor e de um documento

A implementação dos conversores é baseada no padrão *Builder* (Gamma et al., 1995). Este padrão é usado para construir os objetos de uma estrutura complexa de composição, como é o caso do modelo de autoria da NCLite (apresentado na Seção 3.2). O padrão sugere a utilização de uma composição idêntica a encontrada nos objetos que são criados. Na Figura 5.4 podemos verificar a composição dos conversores.

Sabemos que uma cena possui um Repositório espacial, uma Linha do tempo, os Recursos exibidos por esses repositórios e os Eventos de interatividade que levam às próximas cenas. Se repararmos no conversor `SceneConverter` que cria as cenas (`Scene`), observamos que a sua composição é idêntica a mencionada acima.

Percebemos que pelo nome do conversor já fica claro qual é o objeto do modelo criado. Por exemplo, o conversor `ProjectConverter` cria o objeto `Project` que representa um projeto. Os conversores `MediaListConverter` e `HypermediaTemporalGraphConverter` criam objetos `Media` e `Graph<HypermediaVertex, HypermediaEdge>` que serão apresentados com detalhes na Seção 5.2.2.

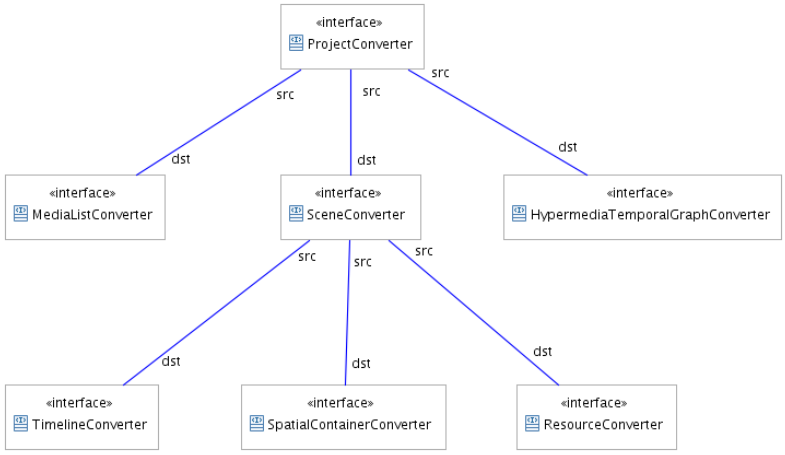

Figura 5.4: Conversores

Além disso, desenvolvemos alguns *Adapters* (Gamma et al., 1995) para converter os valores dos atributos dos elementos da NCL para um formato aceitável pela NCLite. Por exemplo, temos `DescriptorTypeHandle` que converte o atributo `explicitDur` de `String` para o tipo `Time`.

5.2.2 Camada do modelo de autoria

A camada do modelo de autoria contém somente o módulo de mesmo nome que concentra o modelo de domínio da NCLite.

A tecnologia utilizada na implementação é o *framework Eclipse Modeling Framework* (EMF)[3]. Ele cria o modelo de domínio através de um meta-modelo que define a estrutura e gera uma instância do modelo. Esse meta-modelo pode ser definido por um documento XMI (do inglês *XML Metadata Interchange*), por Java *annotations*, por um diagrama UML ou por *XML Schema*. Com o meta-modelo definido, podemos solicitar que o EMF gere a implementação das classes Java do modelo.

O EMF foi selecionado devido às seguintes vantagens: (1) o modelo de domínio é explicitamente criado, o que ajuda na legibilidade. (2) as classes geradas implementam o padrão *Observer* (Gamma et al., 1995) que se responsabiliza por notificar as mudanças do domínio. (3) o EMF também gera fábricas para abstrair a criação dos objetos (4) o modelo é gerado como interfaces Java com classes implementando-as (5) podemos gerar o código Java quantas vezes forem necessárias.

Neste trabalho, utilizamos um documento XMI para definir o meta-modelo porque esta forma de representação maximiza os benefícios dos itens 1 e 5 citados acima.

[3] `http://www.eclipse.org/emf`

O item 2 citado é utilizado da seguinte maneira na ferramenta: os componentes gráficos observam o modelo e, assim que este é alterado pelo usuário, todos eles são notificados. Essa abordagem evita a necessidade de comunicação entre os componentes gráficos para notificar alterações do usuário no modelo.

Visto a tecnologia utilizada e seus benefícios, apresentamos com detalhes o modelo de autoria da NCLite. Na Seção 3.2 exibimos a hierarquia principal do modelo, que consiste do projeto e das cenas. Agora apresentamos essa hierarquia e as demais com detalhes de implementação.

É importante destacar que todo objeto do modelo de autoria herda da classe `DomainModel`, que contém somente o atributo `id` que é o identificador único do objeto. Os diagramas de classes apresentados abaixo omitem o `DomainModel` para evitar repetição.

Projetos, Mídias e Recursos

A Figura 5.5 exibe a hierarquia dos objetos `Media` e `Resource`. Um objeto `Media` é a instância de uma mídia e o objeto `Resource` é composto de todos os elementos necessários para apresentar uma mídia: a própria mídia e os descritores e regiões associados a ela. Essa separação se fez necessária porque uma mídia pode ser inserida diversas vezes em uma mesma cena ou em cenas diferentes. Dessa forma, o objeto `Resource` é instanciado repetidas vezes e o objeto `Media` é único para a aplicação inteira.

Para ficar mais claro, podemos comparar com a NCL. O objeto `Media` seria equivalente ao elemento `<media>` e o `Resource` seria uma referência para o elemento `<media>` e a união dos elementos `<descriptor>` e `<region>`.

Ainda na Figura 5.5, percebemos que as hierarquias de `Media` e `Resource` são idênticas e que o `Resource` possui uma referência para a `Media`. Os tipos das mídias presentes como atributos na NCL foram transformadas em uma hierarquia de classes na NCLite. Sendo assim, para acrescentar novos tipos é preciso implementar a classe `Media` e `Resource`.

Na Seção 3.2 apresentamos superficialmente o objeto `Project`. Por isso, na Figura 5.6 acrescentamos as propriedades específicas do projeto que não foram detalhadas naquela seção.

Verificamos pela figura que o projeto contém a lista das mídias utilizadas (item 1), a cena ativa (item 2), a raiz do grafo de cenas (item 3) e o grafo temporal hipermídia da aplicação NCL (item 4).

Como podemos perceber, o projeto é a raiz do modelo da NCLite. Atualmente, a ferramenta trabalha com um projeto por vez e, por isso, só podemos carregar uma aplicação NCL que é convertida para o modelo

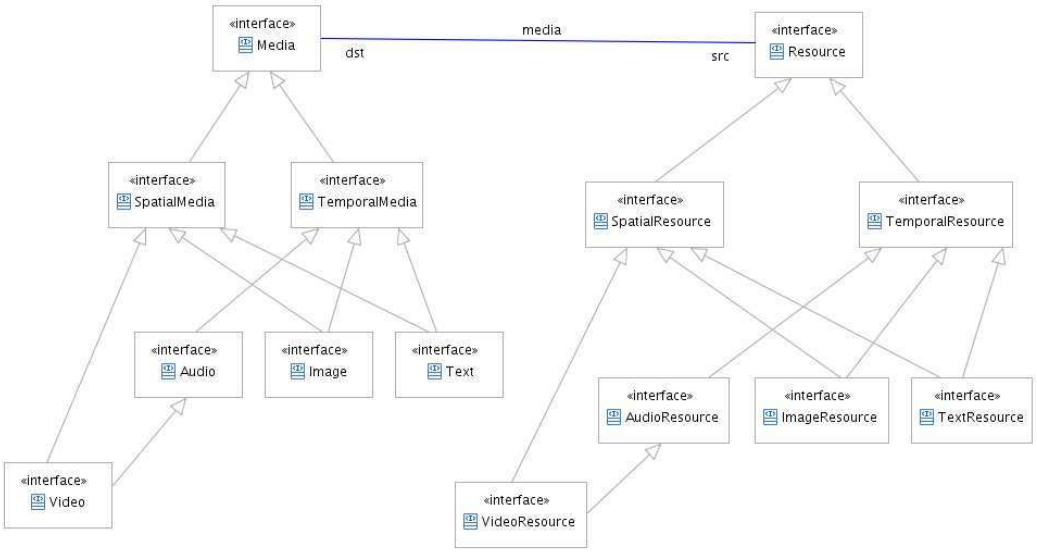

Figura 5.5: Mídias e recursos

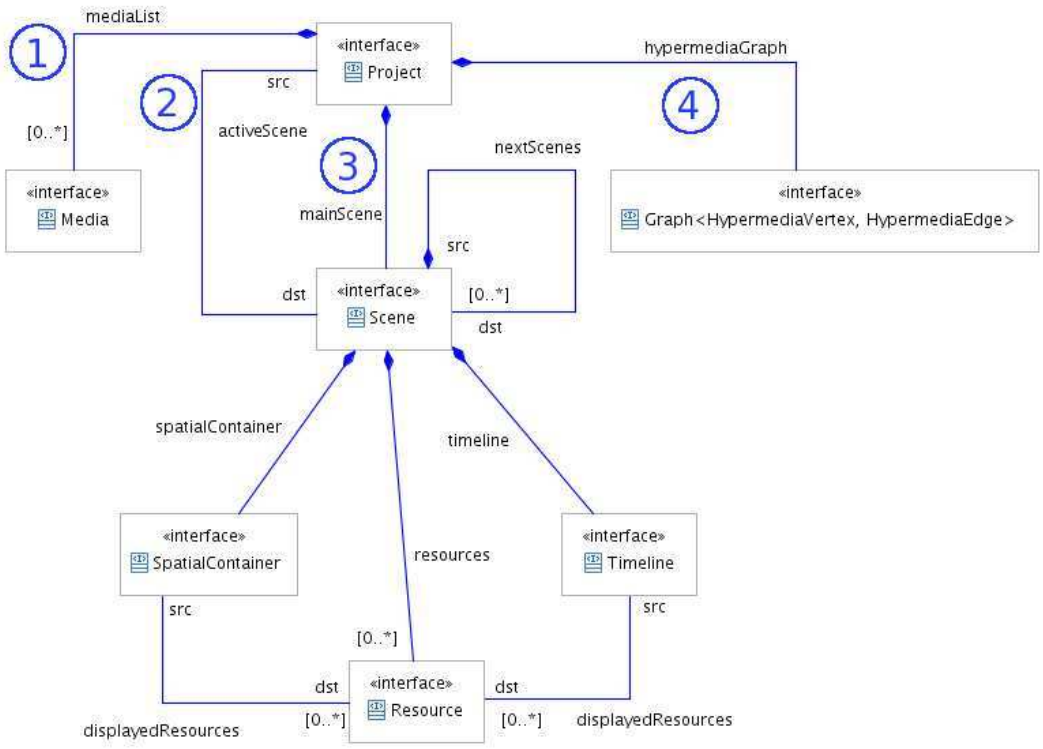

Figura 5.6: Composição do projeto

da ferramenta. Quando o usuário seleciona a opção de salvar o projeto, a ferramenta utiliza a camada de conversão para transformar o modelo em um documento NCL e salvá-lo no disco.

Grafo temporal hipermídia

Foram analisadas algumas estruturas de dados para controlar o comportamento temporal e interativo de uma aplicação hipermídia.

As estruturas de dados *SPANGRAPH* (Kim et al., 2000) e *Petri net* (Peterson, 1977) foram analisadas, mas proporcionavam, respectivamente, perda da expressividade da NCL e aumento considerável da complexidade de implementação.

Como comentado na Seção 3.4.4, utilizamos o grafo temporal hipermídia (Costa & Soares, 2007) na NCLite porque essa estrutura de dados é capaz de representar todas as características temporais e interativas oferecidas pela NCL.

Na NCLite, o grafo temporal hipermídia determina o tempo inicial e de duração dos recursos e também separa as cenas pela interatividade. Quando uma aplicação NCL é carregada geramos o grafo no conversor `HypermediaTemporalGraphConverter`. Ele é acionado pelo `ProjectConverter` que também associa o grafo ao objeto `Project`.

Foi criada uma hierarquia para abstrair a implementação do grafo temporal hipermídia (Figura 5.7). A classe `Graph<V,E>` corresponde a um grafo e possui diversos métodos para manipulação de vértices e arestas. Como exemplo, temos os métodos `setVisited(vertex)`, `setUnvisited(vertex)` e `isVisited(vertex)` para configurar a visita de um vértice. Além desses, encontramos também os métodos `getSource(edge)` e `getDestination(edge)` que informam, respectivamente, o vértice de origem e o de destino de uma aresta direcionada.

Essa abstração de grafo criada também é utilizada para representar o grafo de cenas. A classe concreta é a `SceneGraphAdapter` que implementa `Graph<Scene, Object>`. Como podemos perceber, o nó do grafo é a própria cena (`Scene`), e a aresta não possui representação.

Ainda na Figura 5.7, a classe concreta `JungDirectedGraphAdapter` implementa um grafo que utiliza a biblioteca JUNG[4] (do inglês *Java Universal Network/Graph Framework*) para tirar proveito das facilidades de manipulação e dos diversos algoritmos oferecidos por ela. Pelo nome da classe deixamos clara a utilização do padrão *Adapter* (Gamma et al., 1995).

A Figura 5.8 exibe o objeto `Graph<HypermediaVertex,HypermediaEdge>` que corresponde ao grafo temporal hipermídia e as classes que representam os seus vértices (`HypermediaVertex`) e as suas arestas (`HypermediaEdge`).

Um vértice no grafo temporal hipermídia contém o tipo da transição (*start*, *stop*, *abort*, *pause* ou *resume*), o tipo do evento (apresentação, seleção ou

[4] `http://jung.sourceforge.net/`

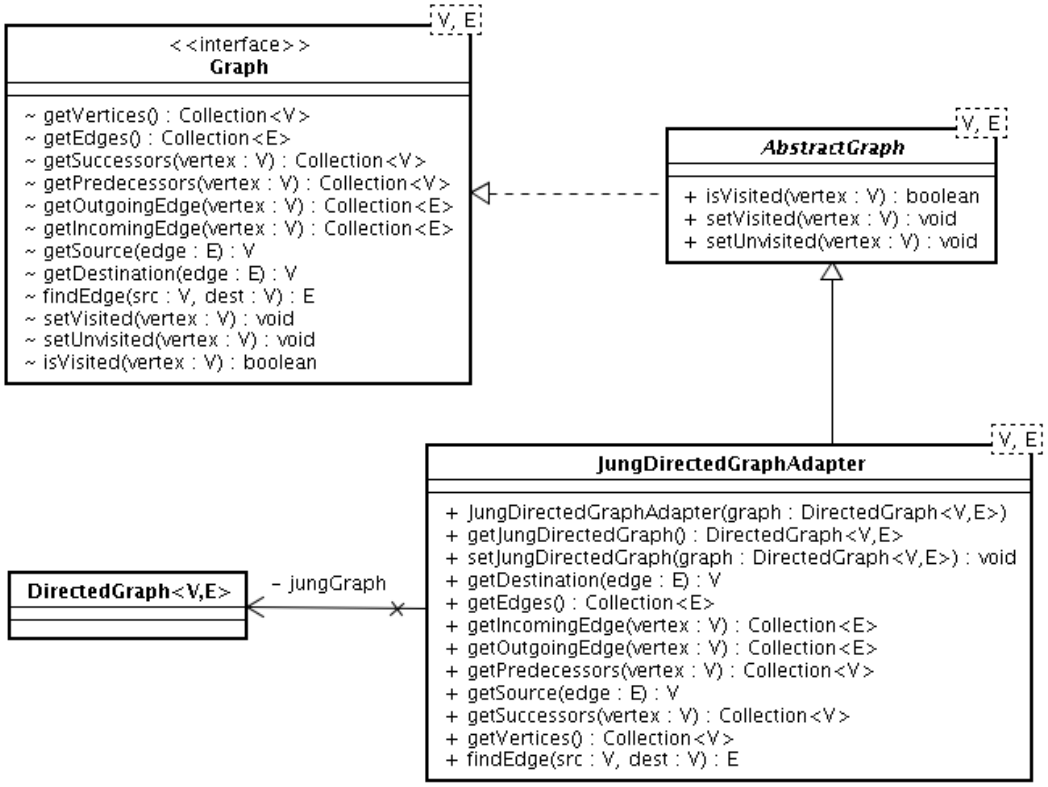

Figura 5.7: Generalização de um grafo

atribuição) e o conteúdo. A classe `MediaContentHypermediaVertex` é a implementação de um vértice com uma mídia como conteúdo. O vértice de atribuição é representado separadamente pela classe `AttributionHypermediaVertex` porque também precisa da propriedade e do valor que deve ser atribuído quando o vértice for visitado.

Já as arestas são diferenciadas pelo tipo de condição que deve ser satisfeita para que o vértice de destino seja alcançado. Atualmente, temos as arestas com condição temporal (`TimeConditionHypermediaEdge`) e de seleção (`SelectionConditionHypermediaEdge`).

O caminhamento no grafo foi resolvido pelo padrão *Visitor* (Gamma et al., 1995). Na Figura 5.9, verificamos que a classe `GraphVisitor` define os métodos `visit(vertex)` e `postVisit(vertex)` para serem implementados nas classes filhas para visitar um vértice. O caminhamento é realizado na classe `AbstractGraphVisitor` através do método `traverse(vertex)`. É possível escolher ainda o tipo (em profundidade ou em largura) e a ordem (busca de sucessores ou de predecessores) de caminhamento.

A classe `FindInteractivityInSceneVisitor` é um exemplo de uma classe visitante que faz o caminhamento no grafo temporal hipermídia para encontrar a interatividade de uma determinada cena. Outro exemplo é a classe `FindMediaInSceneVisitor` que é responsável por encontrar todas as mídias

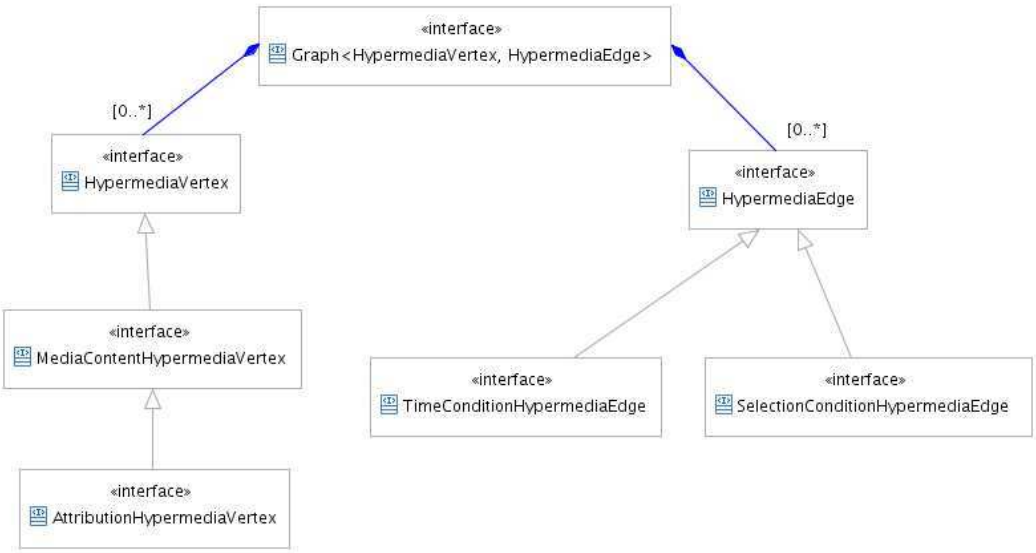

Figura 5.8: Classes do grafo temporal hipermídia

presentes em uma cena.

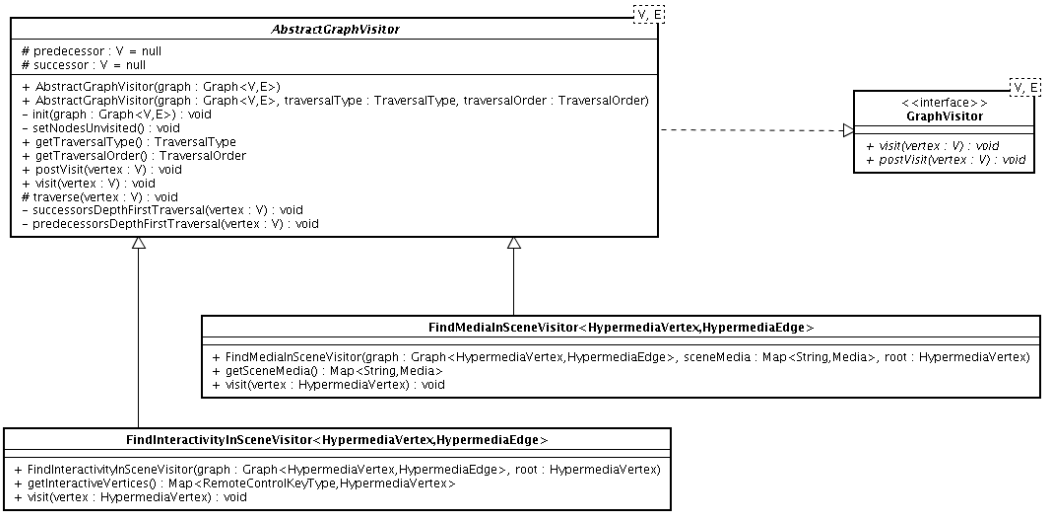

Figura 5.9: Visitantes de um grafo

Tocadores de vídeo e áudio

Foi necessário utilizar uma biblioteca de captura e de execução de vídeo e áudio para conseguir obter os dados a serem exibidos na visão espacial e na linha do tempo. As bibliotecas JMF[5] (do inglês *Java Media Framework*) e *Xuggler*[6] foram estudadas e testadas. A selecionada para ser utilizada na NCLite foi a Xuggler devido às seguintes vantagens: (1) suporte a diversos formatos de vídeo e áudio, inclusive MPEG4; (2) documentação satisfatória

[5] `http://java.sun.com/javase/technologies/desktop/media/jmf/`
[6] `http://www.xuggle.com/`

com diversos exemplos; (3) interfaces de alto e baixo nível disponíveis. O principal problema da JMF foi não suportar o formato MPEG4.

Para capturar e executar um vídeo e/ou áudio utilizamos objetos denominados tocadores de mídias. A Figura 5.10 exibe o diagrama de classes dos tocadores.

Pela figura podemos perceber que a classe `MediaPlayer` define todas as ações esperadas de um tocador de vídeo: *play*, *pause*, *stop* e *resume*. A classe abstrata `AbstractMediaPlayer` implementa o comportamento padrão dessas ações e cria *Template Methods* (Gamma et al., 1995) (ex.: `doInit` e `doPlay`) para inserção dos comportamentos específicos de cada classe concreta.

A captura do vídeo e do áudio são diferentes e, por isso, foram criadas as classes `VideoPlayer` e `AudioPlayer` para definirem os métodos específicos. A Figura 5.10 exibe o `VideoPlayer` com os métodos específicos `readFrameAt(time)` para leitura do quadro em um determinado tempo, `getCurrentFrame` para retornar o quadro corrente e `setVideo(video)` para configurar o vídeo que o tocador deve executar.

Finalmente, a figura apresenta ainda a classe concreta `XugglerMediaToolVideoPlayer` que implementa o tocador que utiliza a biblioteca *Xuggler* para capturar e executar os vídeos. Os métodos e atributos auxiliares desta classe foram omitidos.

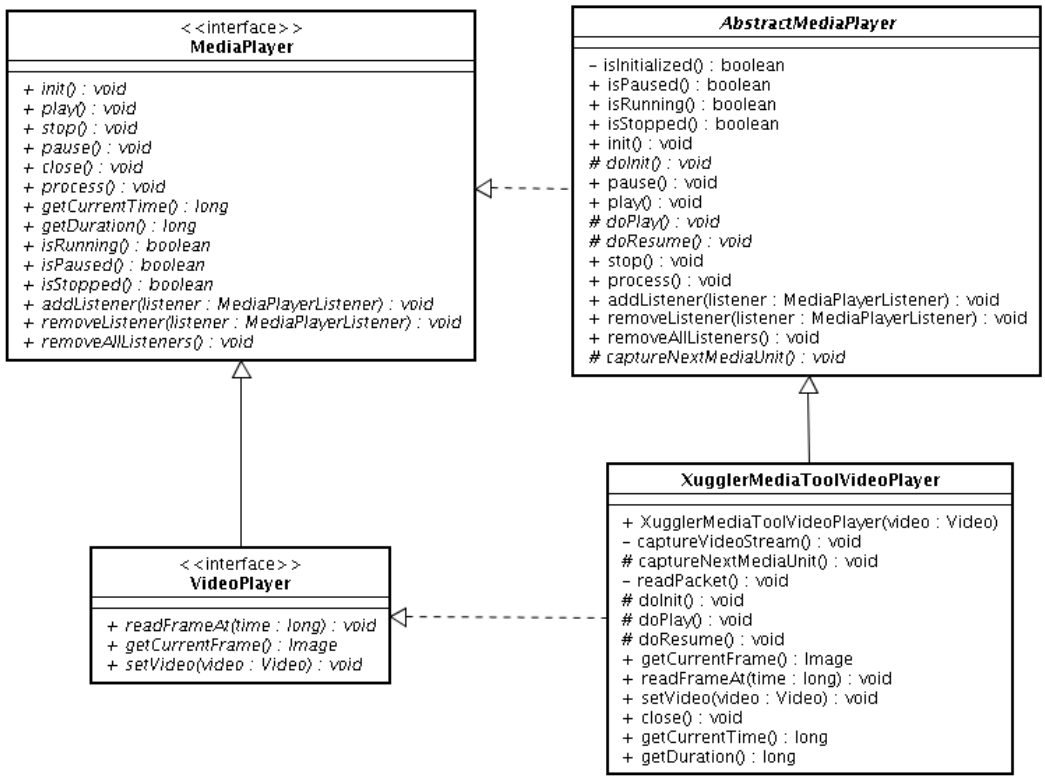

Figura 5.10: Tocadores das mídias

5.2.3 Camada da interface gráfica

Essa camada é responsável por toda a interface gráfica da NCLite. Os módulos são separados de acordo com os componentes gráficos. Dessa forma, temos os seguintes módulos: Componentes auxiliares, Espacial, Linha do tempo e Interatividade. Por último, encontramos ainda o módulo GUI (do inglês *Graphical User Interface*), que agrupa todos os outros módulos.

Além da separação de modelo e visão entre as camadas, cada interface gráfica também possui a separação interna de acordo com o padrão MVC (do inglês *Model-view-controller*). Por exemplo, na visão espacial existem os objetos `EditPart` que representam os controladores e fazem a comunicação com o modelo de autoria. Temos também o objeto `SpatialViewGEFEditor` que representa a visão e possui os *widgets* gráficos. Por fim, o próprio modelo de autoria faz o papel de modelo na visão espacial.

Componentes auxiliares

É composto pelos componentes gráficos Propriedades e Biblioteca de Mídias. O primeiro exibe as propriedades do recurso selecionado e o segundo disponibiliza uma tabela com todas as mídias utilizadas no projeto.

No *Eclipse-RCP* existe um *plugin* que cria um framework com componentes gráficos que exibem as propriedades dos elementos. Para utilizar esse *plugin* é necessário implementar provedores (*Provider*) responsáveis por adicionar descritores (*Descriptor*) que convertem as propriedades dos objetos para o padrão esperado pelos componentes.

Uma outra vantagem do *EMF* (Seção 5.2.2) é que ele gera automaticamente esses provedores a partir das configurações do meta-modelo. É criada também uma fábrica responsável por instanciar todos os provedores. A Figura 5.11 apresenta o diagrama de classes dos provedores da NCLite. Como é possível verificar, a hierarquia dos provedores é semelhante a dos recursos apresentados na Seção 5.2.2.

Além dos provedores apresentados na figura, foi preciso que os controladores da visão espacial (`SpatialResourceEditPart`) e da linha do tempo (`TimelineModel`) que manipulam os recursos implementassem a interface `IAdaptable` para retornar o provedor adequado para cada recurso.

Com o objetivo de evitar repetição de código, criamos a classe `DomainModelPropertyAdaptable` para retornar o provedor de um objeto `DomainModel`. Sendo assim, os controladores da visão espacial e da linha do tempo delegam a responsabilidade de gerar o provedor dos recursos para o objeto `DomainModelPropertyAdaptable`.

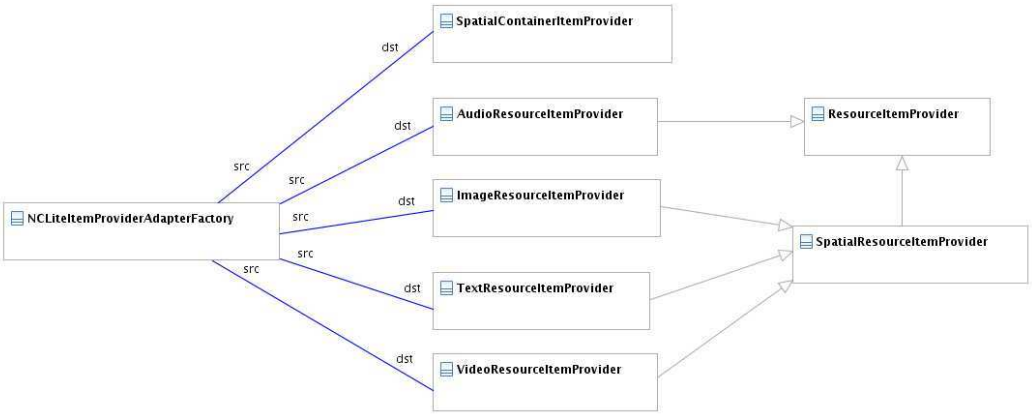

Figura 5.11: Provedores de propriedades

A Biblioteca de mídias é responsável por agrupar todas as mídias da aplicação NCL para proporcionar uma maior rapidez e facilidade em encontrar uma mídia. A implementação atual contempla somente a exibição em detalhes das mídias, deixando para versões futuras a exibição em árvore e com prévia da mídia. Para a forma de exibição implementada, o componente gráfico adequado é a tabela.

No *Eclipse-RCP*, existem os provedores de conteúdo, denominados `ContentProvider` e `LabelProvider`, que separam o modelo da sua representação gráfica e exercem o papel dos controladores do padrão MVC. O componente da Biblioteca de mídias implementa esses objetos para criar a tabela com as mídias.

A seguir apresentamos as classes mencionadas acima e a que exerce o papel da visão do MVC:

- `ArrayContentProvider`: implementação de `ContentProvider` que trata o conteúdo da tabela como um vetor de elementos, no nosso caso é um vetor de mídias. Essa implementação já é pertencente ao *Eclipse-RCP*;
- `MediaLibraryLabelProvider`: implementação de `LabelProvider` que transforma os atributos das mídias em valores esperados pela tabela;
- `MediaLibraryView`: visão da Biblioteca de Mídias responsável por criar a tabela (`TableViewer`) com os objetos `ArrayContentProvider` e `MediaLibraryLabelProvider`. Além disso, insere a lista de mídias na tabela quando é notificada de mudanças no modelo de autoria.

Espacial

Este módulo é responsável por todo o componente da Visão espacial. Iniciamos apresentando a tecnologia usada e depois detalhamos as classes desenvolvidas.

A implementação utiliza o *framework Graphical Editing Framework* (GEF)[7]. Ele permite a criação de um editor gráfico para um domínio específico. Como exemplo temos um editor de circuitos digitais. Ele é composto por dois *plugins*: Draw2D (visão) que realiza a renderização gráfica através de objetos chamados `Figure` e o GEF (controlador) que realiza a conversão entre o modelo do domínio e os objetos `Figure` através de objetos denominados `EditPart`.

A seleção do GEF ocorreu por causa das seguintes características: implementação do padrão MVC; utilização do padrão *Command* (Gamma et al., 1995); *API* completa de renderização (*clipping*, *painting*, *layout*, *z-order*); *API* completa de eventos (interação com *mouse* e teclado); paleta com ferramentas para selecionar e criar elementos e integração com o EMF.

Um `Figure` pode ser descrito como o objeto gráfico que representa um elemento do domínio e o `EditPart` é responsável por criar esses objetos `Figure`. Na NCLite, os objetos `Figure` são mapeados para os recursos. Para ficar mais claro, a inserção de um novo elemento do domínio pode ser descrito da seguinte forma:

1. o usuário seleciona em uma paleta o elemento do domínio que deseja inserir e o arrasta para o editor gráfico;

2. o editor gráfico solicita um novo objeto `Figure` para o `EditPart` associado ao elemento do domínio;

3. o `EditPart` cria e retorna um objeto `Figure` a partir do elemento de domínio correspondente.

Para agrupar todos os `Figure`, o editor gráfico precisa de um objeto raiz que funciona como um repositório desses objetos. A raiz deve ser o elemento do domínio que contém todos os demais elementos representados pelos `Figure`. No caso da NCLite, o elemento raiz do editor gráfico é o repositório espacial.

As classes criadas com base no *framework* do GEF são listadas a seguir. Vale destacar que foi preciso implementar novos objetos `Figure` para contemplar a representação gráfica que a ferramenta necessitava.

[7] `http://www.eclipse.org/gef/`

- `SpatialViewGEFEditor`: responsável por criar o componente gráfico do GEF (`GraphicalViewer`) com a fábrica de controladores `SpatialEditPartFactory`. Além disso, insere o repositório espacial adequado no componente do GEF quando é notificado de mudanças na cena corrente;
- `ExtendImageFigure`: objeto `Figure` que estende a classe `ImageFigure` do GEF responsável por apresentar uma imagem no editor gráfico. Adiciona a funcionalidade de que as propriedades espaciais da imagem SWT[8] precisam ser idênticas ao do objeto `Figure`. É utilizado para representar um recurso de imagem (`ImageResource`);
- `VideoImageFigure`: objeto `Figure` que estende tanto a classe `ExtendImageFigure` quanto o observador `MediaPlayerListener` do tocador de vídeo. Essa classe se resposabiliza em atualizar a imagem SWT de acordo com o quadro lido do vídeo. É utilizado para representar um recurso de vídeo (`VideoResource`);
- `SWTControlFigure`: objeto `Figure` para apresentar um componente SWT como um objeto `Figure` no editor gráfico. O recurso de texto (`TextResource`) no formato *HTML* é renderizado pelo componente `Browser` do SWT e apresentado como um `SWTControlFigure` no editor gráfico;
- `SpatialContainerEditPart`: controlador responsável por criar o elemento raiz do editor gráfico correspondente ao objeto de domínio `SpatialContainer`;
- `SpatialResourceEditPart`: controlador que cria os objetos `Figure` associados aos recursos. Utiliza o padrão *Visitor* para criar o `Figure` correto para cada tipo de recurso;

Para ficar mais claro, a Figura 5.12 apresenta a hierarquia dos objetos `EditPart`. A classe `AbstractEMFGraphicalEditPart` é responsável por se conectar como observador ao objeto do domínio correspondente através dos métodos `addListenToModel(model)` e `removeListenToModel(model)`. Além disso, essa classe também gera o provedor dos objetos do domínio para o correto funcionamento do componente Propriedades (Seção 5.2.3).

Ainda na Figura 5.12, temos as classes `SpatialContainerEditPart` e `SpatialResourceEditPart` mencionadas acima. Ambas herdam de `AbstractEMFGraphicalEditPart` e implementam o método `createFigure()` para criar os objetos `Figure` de acordo com o seu objeto de domínio.

[8] `http://www.eclipse.org/swt/`

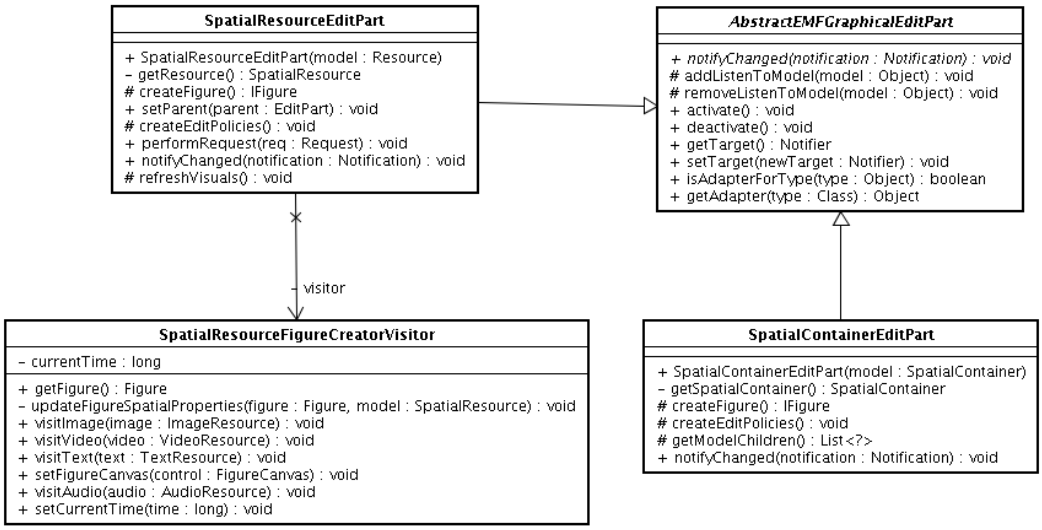

Figura 5.12: Controladores da visão espacial

Continuando na Figura 5.12 temos também a classe visitante `SpatialResourceFigureCreatorVisitor`, que implementa o padrão *Visitor*, para que seja possível visitar os diferentes tipos de recursos e criar os `Figure` corretos. Por exemplo, os recursoes de imagens são visitados pelo método `visitImage(imageResource)`.

Como mencionado, o editor gráfico do GEF realiza o tratamento da interação do usuário com o *mouse* e teclado. A funcionalidade de simulação da interatividade (Seção 3.4.4) precisa que o duplo clique em um recurso seja tratado. Para que isso fosse feito, foi preciso implementar o método `performRequest(req)` de `SpatialResourceEditPart`.

Linha do Tempo

Este módulo é responsável por todo o componente da Linha do tempo. Inicialmente, apresentamos a tecnologia usada e depois detalhamos as classes implementadas.

Utilizamos o *framework Jaret Timebars*[9] que disponibiliza uma visão na forma de um grafo de Gantt ou uma linha do tempo para representar um modelo de dados contendo intervalos de tempo. As características que se mostraram importantes para a escolha dessa biblioteca foram: (1) renderização tratada por objetos `Renderer` que podem ser completamente customizados; (2) implementação do padrão MVC; (3) componentes auxiliares para a linha do tempo: guia vertical e sincronização com outras visões; (4) suporte à edição visual, como *drag-and-drop* e redimensionamento dos intervalos; (5) suporte a mudanças na escala do tempo.

[9] `http://www.jaret.de/timebars/`

No *Jaret Timebars*, a classe `TimeBarViewer` é responsável por criar o *widget* gráfico da linha do tempo a partir do modelo definido na `TimeBarModel`. Os intervalos de tempo são representados pela `Interval` e armazenados no modelo citado. Além disso, temos a classe `ITimeBarChangeListener` que funciona como um observador das mudanças dos intervalos e da guia do tempo.

Na NCLite, foi preciso implementar cada uma dessas classes para se adequarem ao modelo de autoria da ferramenta. Na Figura 5.13 apresentamos o diagrama de classes com essa implementação.

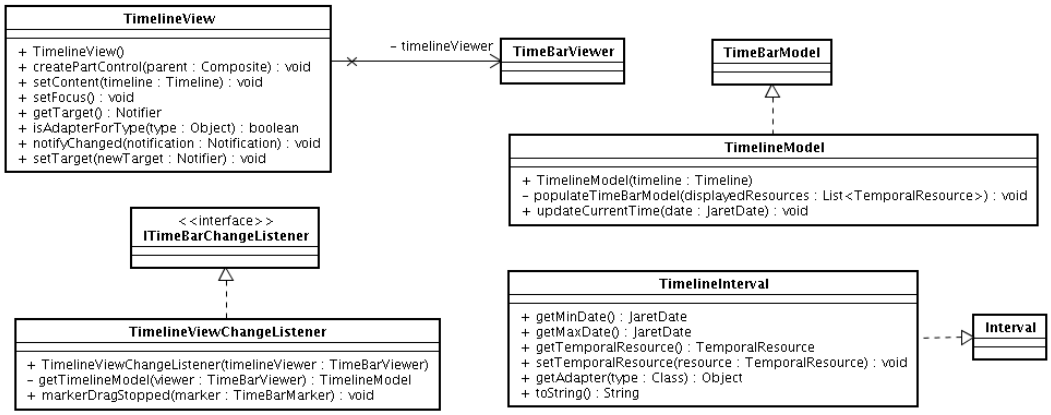

Figura 5.13: Hierarquia da linha do tempo

A classe `TimelineView` se responsabiliza em criar o *widget* gráfico `TimeBarViewer` com a linha do tempo (`Timeline`) do modelo de autoria corrente. Além disso, também altera essa linha do tempo de `TimeBarViewer` quando é notificado de mudanças na cena corrente.

Para o objeto `Timeline` do modelo de autoria com o *widget* `TimeBarViewer` temos as classes `TimelineModel`, `TimelineInterval` e `TimelineViewChangeListener`, que implementam, respectivamente, `TimeBarModel`, `Interval` e `ITimeBarChangeListener`. Um exemplo desse sincronismo é o método `updateCurrentTime(date)` de `TimelineModel` que atualiza o `Timeline` quando a guia do tempo é movida pelo usuário.

Interatividade

É constituído por todos os componentes que exibem ou alteram a interatividade da aplicação NCL. Atualmente temos os componentes Mídias Interativas e Cenas interativas.

O componente Mídias Interativas proporciona uma associação direta entre as mídias e as suas propriedades interativas. Assim como a Biblioteca de Mídias, a forma de exibição desse componente pode ser em detalhes, em

árvore ou com prévia das mídias. A implementação atual contempla somente a exibição em detalhes e o *widget* gráfico adequado para isso é a tabela.

Na Seção 5.2.3 apresentamos os provedores de conteúdo `ContentProvider` e `LabelProvider` que têm a responsabilidade de separar o modelo e a sua representação gráfica. Para a criação da tabela das mídias interativas é preciso implementar esses provedores para contemplar o sincronismo com as mídias.

Dessa forma, implementamos as seguintes classes para o correto funcionamento do componente:

- `InteractiveMediaContentProvider`: implementação da classe `ContentProvider`, que recebe como conteúdo a cena corrente e extrai um vetor das mídias com interatividade que devem ser exibidas na tabela;
- `InteractiveMediaLabelProvider`: implementação de `LabelProvider`, que transforma os atributos das mídias em valores esperados pela tabela;
- `InteractiveMediaView`: visão do componente Mídias Interativas, responsável por criar a tabela (`TableViewer`) com os objetos `InteractiveMediaContentProvider` e `InteractiveMediaLabelProvider`. Além disso, insere a cena corrente no *content provider* quando é notificada de mudanças no modelo de autoria.

Para implementação do grafo de cenas utilizamos o *framework* para visualização de grafos *Zest*[10]. Ele é baseado no SWT e no Draw2D e suporta o conceito de provedores de conteúdo apresentado na Seção 5.2.3.

O Zest oferece diversos gerenciadores de leiaute, como o em árvore e o em grade, para determinar como os nós de um grafo são organizados na tela de exibição. Além disso, podemos inserir filtros nesses gerenciadores para definir quais nós e arestas devem ser exibidos.

Não foi possível tirar proveito da facilidade oferecida pelos provedores de conteúdo no Zest porque precisávamos customizar os componentes gráficos que representavam nós e arestas. Dessa forma, temos somente a visão `InteractiveSceneView` que cria o *widget* gráfico que representa o grafo de objetos `Scene`.

[10] `http://www.eclipse.org/gef/zest/`

Interface Gráfica do Usuário (GUI)

Todos os módulos detalhados acima são implementados como *plugins* do *Eclipse*. Como a NCLite é uma aplicação autônoma, foi necessário criar este módulo como um núcleo para unir todos os módulos e gerar uma aplicação separada do *Eclipse*.

Essa abordagem facilita a inclusão de novos componentes gráficos na ferramenta porque precisamos somente inserir o novo *plugin* com o componente no núcleo. Os demais plugins não sofrem nenhuma alteração.

5.3 Cenário de uso

Com o objetivo de testar a implementação da NCLite, utilizamos alguns exemplos da segunda edição do Tutorial NCL (Tutorial NCL 3.0, 2007). Dentre os exemplos selecionados, apresentamos a seguir o exemplo 9 porque é o mesmo utilizado na avaliação formativa (Capítulo 4). Dessa forma, podemos fazer uma comparação inicial entre o projeto e a implementação da NCLite.

A Figura 5.14 exibe o exemplo 9 no contexto de uma partida de futebol no projeto da NCLite. Já a Figura 5.15 apresenta o mesmo exemplo na implementação. Vale lembrar que a implementação atual reduziu o escopo do projeto proposto.

A maior diferença é a presença do recurso `video2` na cena 1 da implementação. Isso ocorre porque a implementação do exemplo 9 utilizada atribui o valor `false` para a visibilidade do `video2` na cena em questão. Dessa forma, esse recurso faz parte da cena, mas não é exibido. No esboço do projeto considerávamos que esse recurso não pertenceria à cena 1 e só estaria presente na cena 2 porque a interatividade que seria responsável por acionar a sua execução.

Nos demais componentes verificamos que a implementação é semelhante ao projeto. Destacamos também os componentes Mídias Interativas e Biblioteca de Mídias que só tiveram uma forma de visualização implementada. Além disso, o componente Propriedades apresenta, na implementação, todas as propriedades em uma única aba.

Com o exemplo já carregado na NCLite, podemos realizar algumas alterações e gerar uma nova aplicação NCL modificada. A Figura 5.16 apresenta alterações espaciais nos recursos `botaoVermelho` e `video1` da cena 1.

Em seguida, salvamos as modificações na nova aplicação NCL e a executamos na máquina virtual com a implementação de referência do *middleware* Ginga-NCL (Figura 5.17).

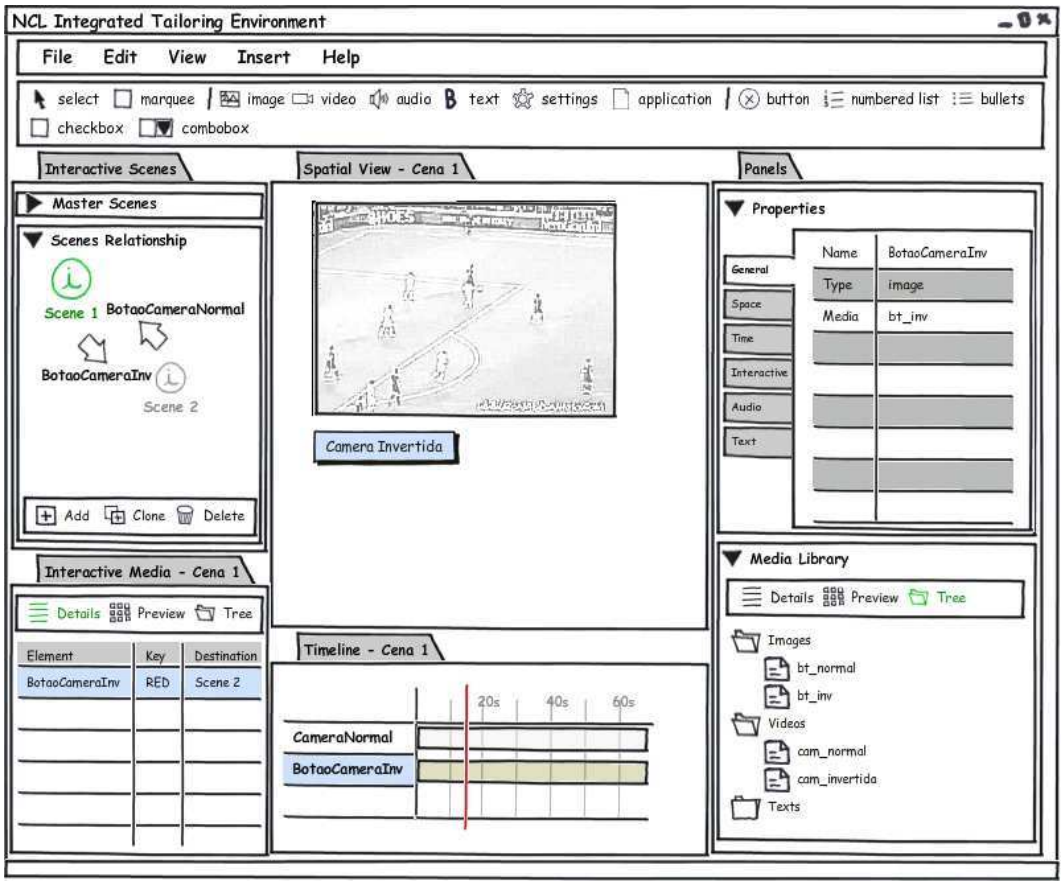

Figura 5.14: Exemplo 9 no projeto da NCLite

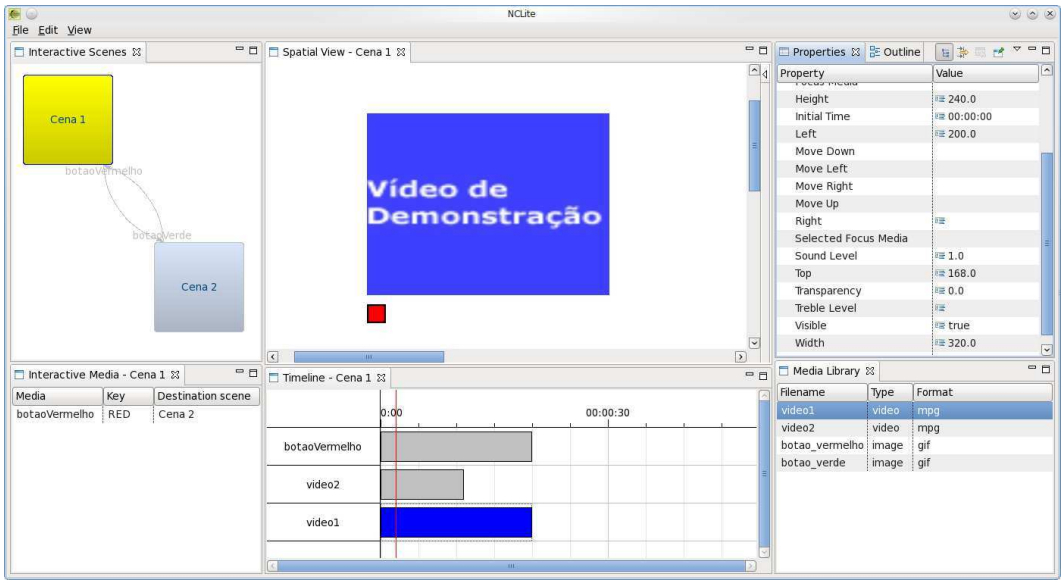

Figura 5.15: Exemplo 9 na implementação da NCLite

O cenário aqui apresentado é uma das possibilidades de interação que a NCLite proporciona aos seus usuários. Alguns outros foram realizados com base nos exemplos do Tutorial de NCL v2.0 (Tutorial NCL 3.0, 2007) e verificamos que a NCLite facilita e agiliza a autoria do autor.

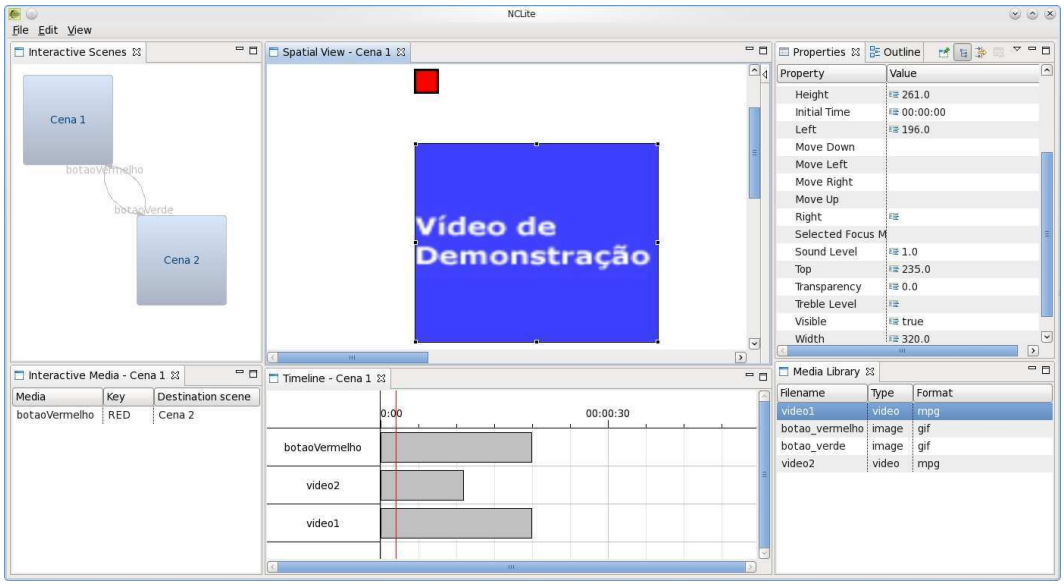

Figura 5.16: Exemplo 9 alterado na NCLite

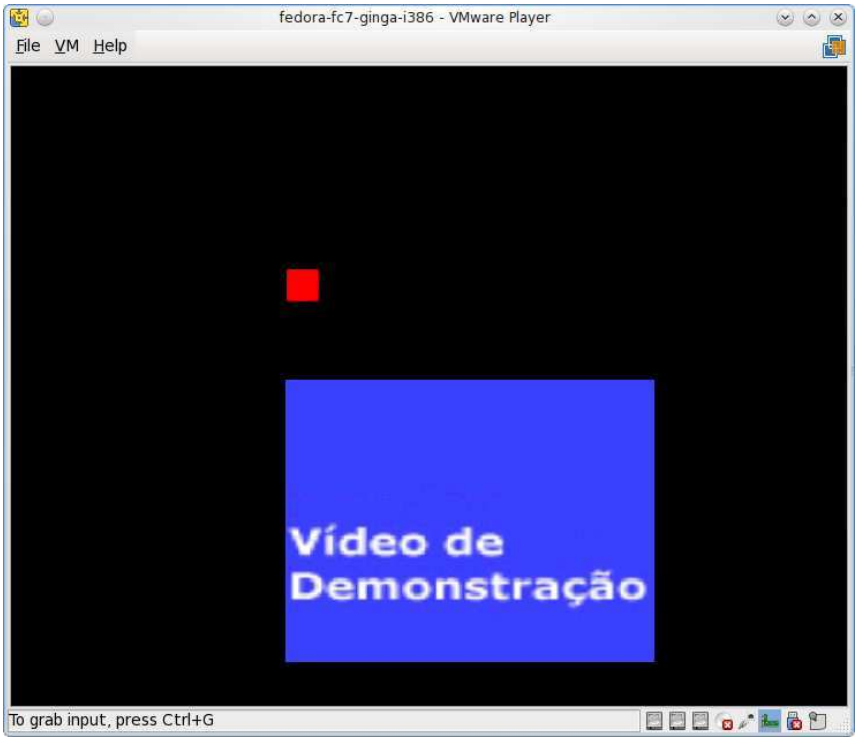

Figura 5.17: Execução do exemplo 9 alterado

Entretanto, devemos destacar que esses exemplos focam na família de aplicações NCL que, quando representadas no grafo temporal hipermídia, são apresentadas como uma árvore e não um grafo. Dessa forma, sabemos que a NCLite é muito útil para esses tipos de aplicações. Para outras famílias é nenessário validar o conceito atual de cena e, se preciso, adaptá-lo.