Composer 3: Ambiente de autoria extensível, adaptável e multiplataforma

Bruno Seabra Lima[1]　Roberto Gerson Azevedo[1]　Marcelo F. Moreno[1]　Luiz Fernando G. Soares[1]

[1] Lab. TeleMídia – DI – PUC-Rio
R. Marquês de São Vicente, 225
22453-900 Rio de Janeiro RJ Brasil
+55-21-3527-1500 ext. 3504

{bslima, robertogerson, moreno, lfgs}@telemidia.puc-rio.br

ABSTRACT

The interactive TV broadcasting chain involves many environments and actors, from the content producer to the broadcast operator, from the editing studio to the broadcaster head-end. They all have different needs on interactive content creation and editing. Even TV viewers in their homes are demanding tools to enrich content and to promote the so-called Social TV. Nowadays, a single authoring tool cannot fulfill their different requirements. This paper discusses the creation of a brand new version of the Composer authoring tool, towards to meet the different authors' requirements, relying on its extensibility, adaptability, robustness and scalability.

Categories and Subject Descriptors

I.7.2 [**Document Preparation**]: Hypertext/hypermedia
D.2.6 [**Programming Environments**]: Integrated environments

General Terms

Design, Languages.

Keywords

NCL, Composer, authoring tool, extensibility, scalability.

1. INTRODUÇÃO

A cadeia de produção e distribuição de conteúdo para TV digital Interativa (TVDi) envolve ambientes diversos e atores com diferentes perfis e necessidades. Desde os produtores de conteúdo, passando pelos operadores de transmissão e chegando até os próprios usuários domésticos (telespectadores) tem-se diferentes necessidades de criação, edição e interação com o conteúdo transmitido. Cada um dos atores envolvidos tem um papel e uma visão particular do processo, o que geralmente culmina em ferramentas focadas nesses atores e que visem suprir suas necessidades específicas.

No que se refere ao desenvolvimento ou edição de aplicações para a TVDi, existem diferentes grupos de usuários potencialmente interessados, e com perfis tão diversos quanto: usuários domésticos, designers gráficos, comunicadores e programadores experientes. Cada um desses grupos de usuários tem expectativas diferentes acerca de uma ferramenta de autoria de aplicações interativas. Mais ainda, esses grupos estão inseridos em diferentes ambientes de produção de conteúdo, conforme pode-se acompanhar pela Figura 1. Designers gráficos, no estúdio de autoria (2), esperam trabalhar em um nível de abstração mais próximo da apresentação final do que programadores inseridos em um ambiente de desenvolvimento e testes (3). Os profissionais no ambiente de radiodifusão (1) necessitam de uma ferramenta que possibilite a transmissão das aplicações e em diferentes redes de distribuição de conteúdo. O próprio telespectador também anseia por uma ferramenta para enriquecer o conteúdo com suas próprias anotações, promovendo a chamada TV Social [1].

Atualmente, ferramentas de autoria para aplicações de TVDi buscam atender certos perfis de usuários comentados acima. Nesse sentido, existem ferramentas que utilizam abstrações visuais baseadas em grafos compostos [2], grafos de cena [3], paradigma de linha temporal [4] e mesmo no conceito de "*What You See is What You Get*" [5], mais voltadas para os usuários iniciantes. Por outro lado, as ferramentas textuais trazem facilidades para programadores, seu público alvo [6]. Percebe-se assim, que um dos principais desafios ao se projetar um ambiente de autoria para TVDi é conseguir suprir as necessidades e expectativas de todos os diferentes perfis de usuários dentro da cadeia de produção e distribuição de conteúdo. E, mais ainda, é desejável que tal ambiente se adapte contínua e gradativamente, refletindo a própria evolução no conhecimento do usuário e o ambiente em que ele está inserido nessa cadeia de radiodifusão.

Como uma tentativa de agregar diferentes grupos de usuários em um mesmo ambiente, o Composer 2 [7] baseia-se no conceito de múltiplas visões. Cada uma de suas visões destaca uma particularidade do documento e pode ser mais adequada para um perfil de usuário. As visões são sincronizadas e algumas delas permitem inclusive que o usuário manipule o documento sem a necessidade de expertise na linguagem-alvo. Apesar do Composer 2 conseguir atingir uma parcela dos perfis de usuários em um ambiente de TVDi, o número de visões que ele disponibiliza é restrito e estático. Somado a isso, há vários diálogos que demandam conhecimento da linguagem NCL para serem preenchidos e, ainda, não houve a preocupação com requisitos não funcionais, como desempenho, portabilidade, adaptabilidade etc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference'10, Month 1–2, 2010, City, State, Country.
Copyright 2010 ACM 1-58113-000-0/00/0010…$10.00.

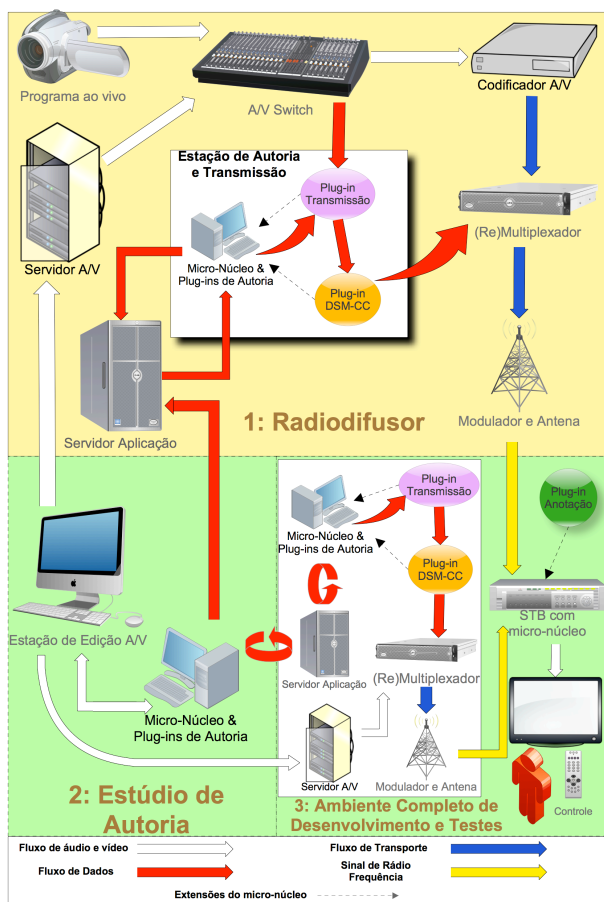

Figura 1 – Diferentes ambientes de produção de conteúdo

Este artigo propõe a base para a construção de um ambiente de autoria integrado (IDE), denominado Composer 3, que seja capaz de se adaptar aos vários perfis de usuário citados e com suporte a requisitos não funcionais. Os programadores experientes devem ser capazes de estendê-lo criando funcionalidades de cunho mais específico. A proposta deste artigo resulta em uma arquitetura baseada em micro-núcleo, responsável pela troca de mensagens entre os diferentes módulos que estendem esse ambiente.

O artigo está organizado da seguinte forma: a Seção 2 discute a importância de requisitos não-funcionais e descreve os aspectos que motivaram a definição da nova arquitetura; a Seção 3 apresenta a arquitetura proposta, discorrendo detalhadamente sobre cada um de seus módulos; a Seção 4 evidencia a interface de programação para extensão do micro-núcleo, detalhando a maneira como um novo *plug-in* é definido.; a Seção 5 exemplifica a extensão do micro-núcleo através da construção de um *plug-in* que implementa uma visão de leiaute. Finalmente, a Seção 6 é destinada às considerações finais.

2. ASPECTOS DE PROJETO

Diversos trabalhos [8][9][10] discutem os requisitos funcionais desejáveis em ferramentas de autoria para aplicações hipermídia. Tais requisitos são reforçados e estendidos em [7]. Entretanto, nenhum desses explora quais trata adequadamente requisitos não-funcionais, que também são de suma importância para uma ferramenta de autoria. Um dos objetivos deste artigo é levantar tais requisitos. Confiabilidade, manutenabilidade, desempenho, adaptabilidade, escalabilidade, portabilidade e extensibilidade são alguns dos requisitos não funcionais que devem ser considerados no projeto de ferramentas de autoria hipermídia.

Mesmo as ferramentas baseadas em múltiplas visões, como Composer 2 e LimSee3 [4] podem ainda ser incompletas ou proibitivamente complexas para um determinado perfil de autor ou aplicação. Por exemplo, um designer gráfico parece se satisfazer com uma visão estrutural seguindo um paradigma de "*StoryBoard*", enquanto um comunicador prefere trabalhar em uma visão mais simples, arrastando e posicionando graficamente os objetos de mídia ao longo de uma linha do tempo. Por outro lado, tais ferramentas pecam principalmente em não despender a devida atenção no suporte a requisitos não-funcionais.

Isso pôde ser particularmente evidenciado com o retorno dado pela comunidade Ginga[1] ao utilizar o Composer 2. Identificou-se que os principais problemas reportados diziam respeito principalmente a requisitos não-funcionais, como instabilidade e declínio de desempenho no uso contínuo da ferramenta, seguidos pela falta de uma visão textual avançada como a existente no NCLEclipse [11].

Os perfis de usuários e os ambientes em que as ferramentas de autoria para TVDi devem executar são bastante diversificados. Por isso, uma ferramenta ideal de autoria deve ser extensível, permitindo que programadores possam acrescentar facilmente novas funcionalidades. Ao ser extensível, tal ferramenta abre espaço para a incorporação de novas técnicas de autoria que ainda estão para ser desenvolvidas. Além disso, como demonstrado na Figura 1, a ferramenta também deve se adaptar ao ambiente de produção ao qual está inserida. Isso demanda que somente partes da ferramenta sejam modificadas para trabalhar com diferentes tipos de equipamentos, sejam estes relacionados a transmissão, exibição ou anotação do conteúdo.

Além de ser extensível, a arquitetura de uma ferramenta de autoria para TVDi também deve ter como foco o desempenho, permitindo escalabilidade, e permitindo que o ambiente acople um número grande de novas funcionalidades sem degradação de performance. Adicionalmente, tal ambiente também deve fornecer mecanismos para que as extensões sejam livres de qualquer especificidade de plataforma, ou seja, multiplataforma.

Todos os requisitos anteriormente levantados se tornam obsoletos se a confiabilidade da ferramenta não for colocada como prioridade. A arquitetura deve estar voltada para minimização na ocorrência de falhas de software e também ser tolerante a falhas de seus componentes, dando suporte a mecanismos de recuperação e evitando, na medida do possível, a perda de trabalho e de tempo por parte do autor.

Com base nos requisitos funcionais já levantados para o Composer 2 e os requisitos não-funcionais aqui discutidos, a seção seguinte apresenta uma proposta de arquitetura para o Composer 3.

3. ARQUITETURA DO COMPOSER 3

Em [4] é apresentado um padrão arquitetural baseado em micro-núcleo e extensões. Esse padrão é aplicado a sistemas de software

[1] Comunidade Ginga no Portal do Software Público Brasileiro: http://www.softwarepublico.gov.br

que devem ser capazes de adaptar-se de acordo com diferentes requisitos. Ele agrupa as funcionalidades mínimas do sistema em um micro-núcleo, separando-o das partes especificas a esses requisitos e suas possíveis extensões. O micro-núcleo serve como um hospedeiro para as extensões e coordena a forma como elas colaboram. Tal arquitetura vai ao encontro dos aspectos e requisitos de projeto não-funcionais levantados na seção anterior e, por isso, é usada como suporte para a proposta do Composer 3.

Nesta proposta, as extensões de funcionalidades do micro-núcleo são feitas por meio de *plug-ins*. *Plug-ins* são programas de computador, distribuídos separadamente, que interagem com uma aplicação hospedeira, com o objetivo de extendê-la ao adicionar funcionalidades e/ou recursos de cunho especifico. O micro-núcleo é o responsável por controlar a troca de mensagens entre os diferentes *plug-ins*, fazer a manutenção de um modelo que representa internamente o documento hipermídia em desenvolvimento e notificar as modificações nesse modelo para os plug-ins interessados.

É importante ressaltar a diferença entre *plug-ins* e as visões existentes nas ferramentas atuais. Visões trazem para o usuário diferentes formas de manipulação de um documento, enquanto os *plug-ins* não necessariamente tem esse retorno visível diretamente para o usuário. Por exemplo, *plug-ins* relacionados a transmissão e validação auxiliam a autoria, mas os usuários não modificam o documento por meio deles. Uma visão sobre o documento certamente é um *plug-in*. Mais ainda, um conjunto de visões podem ser agrupadas em um único *plug-in*.

A Figura 2 detalha o funcionamento do micro-núcleo por passagem de mensagem. O núcleo faz interface com o modelo central e os *plug-ins*, criando uma ponte para a passagem de mensagens. No momento em que o autor da aplicação interage com um determinado *plug-in* (1), tal *plug-in* emite um sinal para o núcleo notificando-o sobre as mudanças efetuadas (2). O núcleo então realiza a validação (3) e, caso a validação não retorne erros, realiza as devidas mudanças no modelo central (4). Logo em seguida, o núcleo emite um sinal para os demais *plug-ins* com a respectiva mudança (5). Dessa forma, os *plug-ins* podem, incrementalmente, realizar as mudanças necessárias. Caso aconteça algum erro, o núcleo notifica o *plug-in* que emitiu o sinal (4) e não repassa a notificação para os demais plug-ins.

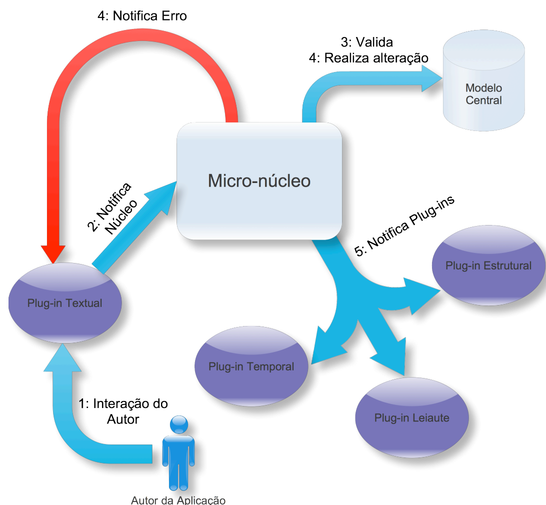

Figura 2.Comunicação entre micro-núcleo e plug-ins

Existem dois tipos de operações sobre o modelo central: operações de *consulta* e operações de *modificação*. As operações de consulta são realizadas pelos *plug-ins* acessando diretamente o modelo central. Por outro lado, as operações de modificação só são realizadas por meio de requisições dos *plug-ins* ao micro-núcleo. Esse mecanismo facilita o controle de acesso concorrente ao modelo central. O micro-núcleo se encarrega de **travar** a parte do documento para modificação, realizar a devida modificação e **destravar**, garantindo a consistência do modelo.

O sincronismo entre visões nas ferramentas atuais (Composer 2 e LimSee3) é geralmente realizado integralmente. Para cada alteração no documento, os modelos internos das visões são sincronizados em sua totalidade com o modelo central. Esse sincronismo degrada a performance da ferramenta à medida em que o projeto cresce, além de ser um ponto crítico de falha. Uma das principais vantagens da proposta baseada em troca de mensagens aqui apresentada, é que os *plug-ins* são incrementalmente sincronizados. As modificações sobre o modelo central são notificadas para os *plug-ins* à medida em que ocorrem, e este, por sua vez, atualiza a interface gráfica dando um retorno visual ao usuário da ferramenta.

Outro ponto a se destacar nesta arquitetura é que cada *plug-in* pode atuar sobre um subconjunto de entidades da linguagem-alvo. Sendo assim, um determinado *plug-in* só é notificado sobre alterações em entidades de seu interesse. É de responsabilidade do micro-núcleo prover tal mecanismo de filtro de mensagens. O funcionamento desse filtro será detalhado na Seção 5. Adicionalmente, o micro-núcleo ainda provê funções para controle de transações com suporte a *rollback*, controle de *plug-ins*, controle de projetos e controle de validação do modelo central. Os módulos, dentro da arquitetura, referentes a essas funções são detalhadas na subseção seguinte.

3.1 Módulos internos

A arquitetura interna do Composer 3, apresentada na Figura 3, é dividida em 3 módulos principais: *CoreModel*, *CoreControl* e *ComposerGUI*. Esses módulos são detalhados nas próximas subseções. No topo da arquitetura tem-se os *plug-ins* dos usuários.

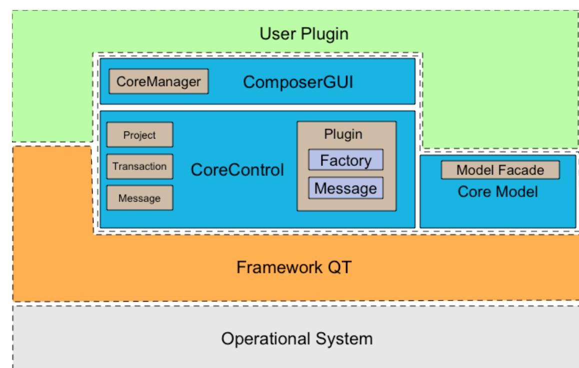

Figura 3. Arquitetura interna do Composer 3

3.1.1 Framework QT:

Utilizado como camada para portabilidade de toda arquitetura, esse framework tem versões para vários sistemas operacionais, inclusive sistemas portáteis (symbian, maemo, windows mobile). Além disso, o QT também provê APIs para construção de interface gráfica, manipulação XML e multimídia (áudio, vídeo, imagens) e acesso a interface de rede.

3.1.2 CoreModel

Construído sobre o *framework* QT, esse modelo central é a representação interna da aplicação em desenvolvimento pelo usuário da ferramenta, e deve ser diferente dos modelos de execução utilizados por máquinas de apresentação dessas aplicações, nas quais o documento deve permanecer consistente em todos os momentos. O modelo central em uma ferramenta de autoria deve ser flexível e aceitar inconsistências em determinados momentos durante a autoria. Não permitir tais inconsistências traz problemas de comprometimento precoce, previstos em [13]. Tais inconsistências se dão na medida em que é permitido ao usuário editar aleatoriamente partes do documento. Este modelo central foi construído com o intuito de prover tal flexibilidade.

Para possibilitar a comunicação entre *CoreControl* e *CoreModel* foi desenvolvida uma fachada que provê a *CoreControl* uma API simples para manipulação sobre *CoreModel*. *ModelFacade* é responsável por receber requisições de modificações sobre o modelo interno e executar os procedimentos necessários para que essas modificações sejam realizadas corretamente.

3.1.3 CoreControl:

O *CoreControl* é o micro-núcleo em si. É nesse módulo que ficam as funções essenciais mencionadas na Seção 3. Ele também é encarregado de fazer a ponte para troca de mensagem entre os *plug-ins* e o modelo central. É composto de quatro módulos principais: *Message, Transaction, Plug-in e Project*.

O módulo *Message* é encarregado pela gerência das trocas de mensagens entre os *plug-ins* e o micro-núcleo. Esse módulo recebe os sinais emitidos pelos *plug-ins*, interpreta-os e delega suas respectivas ações para o *ModelFacade*. Quando a modificação é realizada com sucesso, esse módulo notifica os *plug-ins* interessados nesse elemento.

O módulo *Transaction* é responsável por gerenciar as transações que visam manipular o modelo central. Esse módulo provê uma API em que *plug-ins* terão a possibilidade de abrir sessões e enviar mensagens para o micro-núcleo. Tais mensagens devem ser tratadas de forma atômica. Além disso, o *plug-in* pode solicitar o *rollback* da transação em um determinado momento.

O *Plug-in* é o módulo responsável pelo carregamento dos *plug-ins* externos. Esse módulo executa a função de filtro para cada um dos *plug-ins* e faz a conexão entre os sinais específicos do micro-núcleo, sobre os elementos filtrados, com as funções que irão tratar esses sinais. Esse módulo ainda define as interfaces que devem ser implementadas para a extensão do micro-núcleo. A Seção 4 detalha essa interface e a Seção 5 exemplifica seu uso.

O módulo *Project* gerencia operações sobre projetos (abrir, deletar, salvar) e manipula documentos que serão inseridos ou criados dentro dos projetos.

3.1.4 ComposerGUI:

Os *plug-ins* também podem adicionar elementos gráficos à interface, os quais são incorporados a uma *QmdiArea* – componente do QT que é uma área para exibição de múltiplos componentes gráficos. O desenvolvimento de um *plug-in*, em relação à sua interface gráfica, não difere em nada do desenvolvimento de uma aplicação autônoma sobre o *framework* QT. Além disso, *ComposerGUI* tem um módulo interno denominado *CoreManager*, responsável pela comunicação da interface gráfica com os módulos internos do micro-núcleo.

4. INTERFACE DE PROGRAMAÇÃO DE PLUG-INS

Os *plug-ins* a serem incorporados na arquitetura devem interagir com o micro-núcleo através de uma interface de comunicação única e ortogonal. Tal interface deve ser simples e objetiva, para que o autor de uma nova funcionalidade mantenha o foco no que é de seu interesse e abstraia como o *plug-in* é integrado à ferramenta.

O *framework* QT provê um mecanismo de extensão de suas aplicações por meio de *plug-ins*. Utilizando o *QtPlugin*, o autor de um novo *plug-in* desenvolve uma aplicação autônoma QT, e faz uso de toda as ferramentas de desenvolvimento QT que auxiliam na construção da interface gráfica. Para transformar essa aplicação autônoma em um *plug-in,* basta especificar uma classe que implementa a interface definida pelo micro-núcleo.

4.1 Definição de plug-ins

Para desenvolver um novo *plug-in* para o Composer 3, o autor deve implementar duas interfaces: *IPluginFactory* e *IPluginMessage.*

Ao implementar a interface *IPluginFactory*, o autor de um *plug-in* define como criar e destruir instâncias desse *plug-in.* Ao utilizar essa interface, o *CoreControl* permite que várias instâncias de um mesmo *plug-in* executem ao mesmo tempo. Cada instância de um *plug-in* criada é atrelada a uma instância de um documento.

A Listagem 1 apresenta a interface *IPluginFactory*. A função *createPluginInstance()* deve retornar uma instância do objeto *IPluginMessage* para que o micro-núcleo possa realizar as conexões das trocas de sinais com esse *plug-in.* A função *releasePluginInstance()* é chamada para liberar a instância passada como parâmetro, liberando a memória quando um documento não estiver mais sendo editado por esse *plug-in*.

```
class IPluginFactory {
  public:
   IPluginFactory();
   virtual ~IPluginFactory() = 0;
   virtual IPluginMessage*
          createPluginInstance()  = 0;
   virtual void
           releasePluginInstance
               (IPluginMessage *) = 0;
};
```

Listagem 1 – Interface *IPluginFactory*

A interface *IPluginMessage*, detalhada na Listagem 2, define os sinais e funções que tratam da comunicação do *plug-in* com o micro-núcleo. Além disso, essa interface define uma função para definição de filtros de interesse. Esses filtros evitam que *plug-ins* recebam mensagens relativas a elementos da linguagem que não são de seu interesse. Por exemplo, para o *plug-in* que implemeneta a visão de leiaute, apresentado na Seção 5, somente os elementos relacionados à disposição espacial (<*regionBase*> e <*region*> em NCL) são relevantes. O micro-núcleo chama a função de filtro (*listenFilter*) passando um tipo de entidade (no caso um elemento NCL). O autor do *plug-in* deve então retornar verdadeiro caso essa entidade seja de seu interesse ou falso, caso contrário. Todo o mecanismo de filtro de mensagens é gerenciado pelo micro-núcleo.

```
class IPluginMessage : public QObject{
 Q_OBJECT
 private:
  NclDocument *nclDoc;
 public:
  IPluginMessage();
  virtual bool listenFilter(EntityType ) = 0;
 public slots:
  virtual void onEntityAdded(Entity *) = 0;
  virtual void onEntityAddError(string) = 0;
  virtual void onEntityChanged(Entity *) = 0;
  virtual void onEntityChangeError(string) = 0;
  virtual void onEntityAboutToRemove(Entity *) = 0;
  virtual void onEntityRemoved(string) = 0;
  virtual void onEntityRemoveError(string) = 0;
 signals:
  void addEntity(EntityType entity, string
parentEntityId, map<string,string>& atts, bool force);
  void editEntity(EntityType,Entity *,
map<string,string>& atts, bool force);
  void removeEntity( Entity *, bool force);
};
```

Listagem 2 – Interface *IPluginMessage*

O micro-núcleo emite sinais para os plug-ins quando ocorrem modificações nos elementos (criação, alteração ou deleção). Esses sinais são tratados por funções no *plug-in,* as quais serão acionadas somente para os elementos previamente filtrados. Na criação ou alteração de um elemento o *plug-in* recebe um ponteiro com a instância referente ao elemento.

Os *plug-ins* constantemente precisam dos valores de atributos de certos elementos. O modelo interno foi projetado para que operações de consulta, que não modifiquem os elementos, sejam realizadas de forma direta por uma instância desse elemento recebida pelo *plug-in*. Tal mecanismo evita o desperdício de memória, concentrando os dados em uma única área, e diminui o número de mensagens trocadas com o micro-núcleo. Além disso, permite liberdade e flexibilidade para o autor de *plug-in* estruturar seus dados de acordo com sua necessidade.

Diferente da operação de consulta, para alterar um elemento, o *plug-in* deve notificar o micro-núcleo, por meio de mensagens. Nas mensagens enviadas pelos *plug-ins* informando alterações em elementos, a variável *force* informa ao micro-núcleo se tais operações devem ser executadas mesmo que tornem o modelo inconsistente. Conforme já discutido, manter o modelo inconsistente em determinados momentos durante a autoria é necessário para evitar problemas de comprometimento precoce.

5. EXEMPLO DE PLUG-IN

Para validar a arquitetura exposta neste artigo foi desenvolvido, como prova de conceito, um *plug-in* que implementa uma visão de leiaute para a linguagem NCL. NCL permite especificar a disposição dos elementos na tela por meio de dois elementos principais: *<region>* e *<regionBase>*. No *plug-in* desenvolvido, o autor de um documento NCL pode especificar esses elmentos de forma gráfica, mais próxima da apresentação final.

A Figura 4 apresenta a interface gráfica desse *plug-in* de leiaute. À esquerda (1) encontra-se uma área responsável por apresentar a hierarquia das regiões, sendo possível colapsá-las e expandi-las. A região de desenho é apresentada em (2) e representa a tela inteira do dispositivo. Na parte inferior (3) observam-se os atributos de posicionamento referentes aos elementos *<region>*.

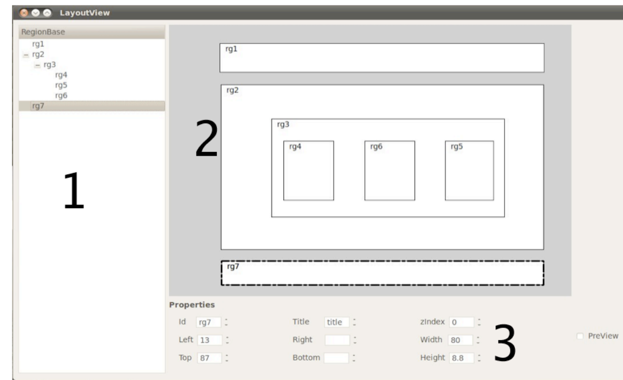

Figura 4 – Interface Gráfica plug-in Leiaute

A Listagem 3 demonstra como o autor do *plug-in* deve informar ao núcleo quais são as entidades de seu interesse. Nesse caso, somente modificações referentes aos elementos *<region>* e *<regionBase>* serão notificadas para esse *plug-in*.

```
bool listenFilter(EntityType ) {
  if (EntityType == REGION ||
      EntityType == REGIONBASE)
     return true;
  return false;
}
```

Listagem 3 – Função de filtragem do plug-in de leiaute

Para a implementação do *plug-in* de leiaute foram desenvolvidas seis classes: *NCLRegion, Attributes, ScribbleArea* e *MainWindow e LayoutController e LayoutFactory*.

A classe *NCLRegion* estende a classe *QRect,* que possui um conjunto de funções para manipulação de retângulos. Foram implementadas funções para lidar com atributos específicos dos elementos *region* de NCL, como *title* e *zIndex*. A classe *Attributes* é responsável por exibir e modificar os atributos da região selecionada no *canvas*.

A classe *ScribbleArea* implementa o *canvas*. O evento *paintEvent()* foi sobrescrito para implementar a lógica de renderização das regiões no *canvas,* levando em conta aspectos específicos de NCL. O gerenciamento dessas regiões criadas é feito por meio de uma lista contendo os ponteiros de instâncias da classe *Region,* repassados pelo micro-núcleo.

A classe *MainWindow* implementa a interface gráfica principal do *plug-in*. É responsável por organizar a disposição na tela dos três componentes de interface apresentados na Figura 4. Finalmente, a classe *LayoutController*, e *LayoutFactory*, são encarregadas de implementar as interfaces providas pelo micro-núcleo. A classe *LayoutFactory* implementa a interface *IPluginFactory*, enquanto a *LayoutController* implementa a interface *IPluginMessage*. É através da classe *LayoutController* que o *plug-in* se comunica com o micro-núcleo.

O autor do documento interage com a interface do plug-in criando uma nova região, esse evento é recebido pela *MainWindow*, que notifica o *LayoutController*, esse por sua vez é encarregado de formatar a mensagem e emitir um sinal ao micro-núcleo sobre a criação de uma nova região. O núcleo recebe esse sinal, cria a região no modelo interno e notifica os plug-ins que filtram o elemento <region>, inclusive o próprio plug-in que emitiu o primeiro sinal.

De forma análoga, o núcleo emite um sinal quando uma região é modificada (por um *plug-in* terceiro), esse sinal aciona o *LayoutController* que interpreta a mensagem e repassa os atributos modificados para a *MainWindow*, que atualiza a interface gráfica dando o feedback visual ao autor do documento.

6. CONCLUSÕES

Diversos são os perfis de usuário que estão direta ou indiretamente ligados com o processo de criação, edição e transmissão de aplicações interativas na TVDi. Cada um desses perfis possui especialidades, necessidades e expectativas diferentes. Além disso, tais usuários também podem estar em ambientes com capacidades diversas (radiodifusor, ambiente de autoria, ambiente doméstico). Prover uma ferramenta de autoria única para todo o processo de TVDi, dessa forma, passa por definir uma arquitetura adaptável, extensível e multiplataforma.

Este trabalho apresenta alguns dos principais requisitos não funcionais que uma ferramenta de autoria para TVDi deve dar suporte e propõe a arquitetura do Composer 3. O Composer 3 é uma IDE baseada em um micro-núcleo, responsável por troca de mensagens, no qual são acoplados diversos *plug-ins* que provêem as funcionalidades específicas aos usuários finais. Como resultado, obtém-se um ambiente altamente extensível e adaptável. Como uma das principais vantagens em relação às ferramentas existentes, essa arquitetura permite um sincronismo incremental entre as visões dos diversos *plug-ins*, além da validação das messagens trocadas e controle de transação.

Alguns dos trabalhos futuros incluem: Um repositório Web para desenvolvimento e publicação de *plug-ins*, sejam eles comerciais ou não; a integração desse repositório com o próprio Composer 3, de modo que seja possível dinamicamente criar distribuições visando um determinado público alvo ou ambiente de execução; e a criação de funcionalidades para a autoria colaborativa de aplicações hipermídia.

7. AGRADECIMENTOS

Os autores gostariam de agradecer a equipe do laboratório Telemídia que contribuíram com este trabalho, em particular Carlos S. Soares Neto e Eduardo Araújo. Os autores também agradecem ao CNPq, CAPES, MCT e CTIC/RNP pelo suporte.

8. REFERÊNCIAS

[1] L. Oehlberg, N. Ducheneaut, J. D. Thornton, R. J. Moore, and E. Nickell, "Social tv: Designing for distributed, sociable television viewing," in *EuroITV*, May 2006.

[2] Coelho, R. M., Rodrigues, R. F., Soares, L. F. G. Integração de Ferramentas Gráficas e Declarativas na Autoria de Arquiteturas Modeladas através de Grafos Compostos. X Simpósio Brasileiro de Sistemas Multimídia e Web – WebMedia 2004, Outubro 2004.

[3] Icareus iTV Suite Author, Icareus Technology. Disponível em: http://www.icareus.com/

[4] Deltour, R. and Roisin, C. 2006. The limsee3 multimedia authoring model. In Proceedings of the 2006 ACM Symposium on Document Engineering (Amsterdam, The Netherlands, October 10 - 13, 2006). DocEng '06.

[5] Araújo, E. C.. NCL Live: ambiente de autoria visual para prototipação de aplicações para TV Digital Interativa. 2009. Trabalho de Conclusão de Curso. Universidade Federal do Maranhão.

[6] Azevedo, R. G. A. ; Lima, B.S; Soares Neto, C. S. ; Texeira, M. A. M. Uma abordagem para autoria textual de documentos hipermídia baseada no uso de visualização programática e navegação hipertextual. XV Simpósio Brasileiro de Sistemas Multimídia e Web (Webmedia), Fortaleza - Brasil. 2009

[7] Guimarães, R. L. Composer: Ambiente de Autoria de Aplicações Declarativas para TV Digital interativa. Dissertação de Mestrado. Junho de 2007

[8] Junehwa Song, Michelle Y. Kim, G. Ramalingam, Raymond Miller, Byoung-Kee Yi, "Interactive Authoring of Multimedia Documents," Visual Languages, IEEE Symposium on, p. 276, 1996 IEEE Symposium on Visual Languages, 1996

[9] Vazirgiannis, M; Kostaas, I; Sellis, T. Specifying and authoring multimedia scenarios. IEEE Multimedia Magazine. Vol6, no. 3. Julho de 1999

[10] Hardman, L., van Rossum, G., and Bulterman, D. C. Structured multimedia authoring. In Proceedings of the First ACM international Conference on Multimedia. Agosto 1993.

[11] Azevedo, R. G. A. ; Soares Neto, C. S. ; Texeira, M. A. M. . NCL Eclipse: Ambiente Integrado para o Desenvolvimento de Aplicações para TV Digital Interativa em Nested Context Language. Salão de ferramentas SBRC 2009.

[12] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-Oriented Software Architecture: A System Of Patterns. West Sussex, England: John Wiley & Sons Ltd., 1996

[13] Green T. R. G., Petre M., Usability Analysis of Visual Programming Environments: A `Cognitive Dimensions' Framework, Journal of Visual Languages & Computing, Junho 1996