# 4
# Arquitetura

No Capítulo 2, foram discutidas e analisadas as ferramentas de autoria de aplicações hipermídia. Nenhuma dessas ferramentas foi desenvolvida levando em consideração requisitos não-funcionais. No Capítulo 3 os requisitos não-funcionais foram discutidos e foi levantada a importância de cada um desses requisitos no desenvolvimento de uma ferramenta de autoria para aplicações hipermídia. Este capítulo apresenta uma arquitetura para servir de sustenção no desenvolvimento de novas ferramentas de autoria hipermídia. A arquitetura foi construída levando em consideração os requisitos funcionais amplamente discutidos, assim como os requisitos não-funcionais expostos neste trabalho.

## 4.1.Arquitetura para ferramentas de autoria hipermídia

A explosão de conteúdo nas páginas Web se deu principalmente pelo advento de ferramentas de autoria que possibilitaram transformar os usuários finais em produtores de conteúdo. Atualmente, no cenário brasileiro, as aplicações vêm sendo desenvolvidas por programadores que têm experiência na tecnologia e nas linguagens de autoria hipermídia. De maneira análoga à Web é necessária a criação de conteúdo hipermídia por pessoas não técnicas e detentoras de criatividade e conteúdo. Adicionalmente, em alguns casos como em aplicações voltadas para TV Social (Oehlberg, Ducheneaut, Thornton, Moore, & Nickell, 2006) (Geerts & Grooff, 2009) (Harboe, Massey, Metcalf, Wheatley, & Romano, 2008), usuários finais (telespectadores) podem virar coprodutores. Sendo assim, é preciso o desenvolvimento de uma ferramenta gráfica de autoria abrangente, simples e leve para dar a possibilidade desse grupo de autores criarem conteúdo hipermídia. Entretanto, é recomendável que esse autor seja capaz de ler e interpretar o código-fonte da aplicação desenvolvida. As linguagens declarativas hipermídia foram pensadas com esse intuito, por isso, essa arquitetura tem seu foco no paradigma de programação declarativo.

Em (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 2000) é apresentado um padrão arquitetural baseado em micro-núcleo e extensões. Esse padrão é aplicado a sistemas de software 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 específicas 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 de requisitos não-funcionais levantados anteriormente.

Nesta arquitetura, as **extensões** de funcionalidades do micro-núcleo são feitas por meio de plug-ins. Normalmente, plug-ins são definidos como programas de computador, distribuídos separadamente, que interagem com uma aplicação hospedeira, com o objetivo de estendê-la ao adicionar funcionalidades e/ou recursos de cunho específico. Selecionando os plug-ins que satisfaçam determinados requisitos em sua autoria, os autores podem construir instâncias **adaptadas** da ferramenta.

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.

Nesse ponto é importante ressaltar a diferença entre plug-ins e visões utilizadas nas ferramentas de autoria atuais. Visões permitem diferentes abordagens para manipulação do documento, enquanto plug-ins não necessariamente têm essa resposta visual para o autor. Por exemplo, pode-se ter plug-ins relacionados a validação, armazenamento ou transmissão do documento que facilitam e ajudam a autoria, mas os autores não modificam o documento utilizando-os, em alguns casos o autor não têm ideia da existência de certos plug-ins trabalhando sobre o documento. Uma visão certamente é tida como um plug-in para a ferramenta, mas um plug-in nem sempre será uma visão. Além disso, um único plug-in pode ser composto por um conjunto de visões.

As ferramentas de autoria atuais em sua maioria utilizam visões, e como visto anteriormente, existem dois problemas em potencial em relação a essa abordagem: a comunicação e sincronismo entre as diferentes visões; e as operações de manipulação no modelo central. A Tabela 1 mostra que a maioria das ferramentas não faz o sincronismo incremental de suas visões.

Adicionalmente, as operações realizadas para manipulação do modelo central se tornam custosas, pois é preciso garantir a consistência e integridade dos dados armazenados no modelo.

A Figura 11 apresenta uma nova proposta de comunicação entre as diferentes visões, nesse caso plug-ins, e manipulação do modelo central. Esse novo mecanismo visa minimizar os potenciais problemas mencionados anteriormente. Nessa proposta o micro-núcleo funciona como uma ponte entre o modelo central e os plug-ins e sua principal vantagem é se basear no esquema de sincronismo incremental. Dessa maneira, os plug-ins vão sendo sincronizados a medida que o autor vai manipulando a aplicação.

Essa nova proposta é ilustrada na Figura 11. No momento em que o autor interage com um plug-in específico, no caso da figura o plug-in textual (**1**), esse plug-in notifica o micro-núcleo sobre a modificação realizada pelo autor (**2**). O micro-núcleo então faz a validação dessa modificação (**3**) e, caso a validação não retorne erros, realiza as devidas manipulações no modelo central a fim de refletir as modificações realizadas pelo autor (**4**). Logo em seguida, o micro-núcleo emite um sinal para os demais plug-ins com a respectiva modificação (**5**). Dessa maneira, os plug-ins realizam incrementalmente (em pequenas modificações) sua sincronização com o modelo central. No caso da validação (**3**) retornar algum erro, o micro-núcleo não realiza a modificação no modelo central e retorna o erro para o plug-in que emitiu a notificação de mudança (**4**).

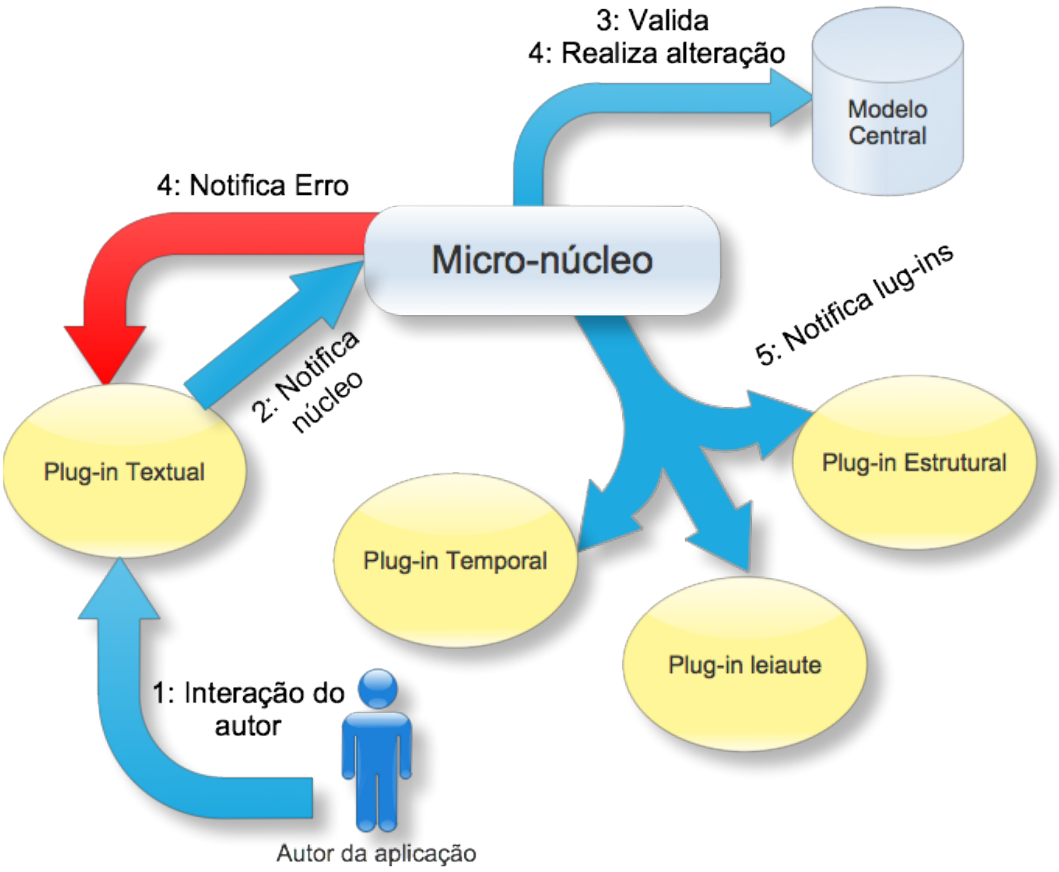


Figura 11 - Mecanismo de comunicação entre micro-núcleo de plug-ins

É importante mencionar que o micro-núcleo deve usar somente um modelo central para a aplicação hipermídia que está sendo criada. Manipular mudanças utilizando um único modelo central é importante para manter a consistência e integridade, reduzindo o custo computacional das operações de edição, melhorando o **desempenho** e **escalabilidade**.

Os plug-ins na maior parte das vezes querem consultar atributos (propriedades) das entidades contidas no modelo central. Com o objetivo de evitar que toda consulta ao modelo central passe necessariamente pelo micro-núcleo, foram instituídas duas operações básicas sobre o modelo central: operações de consulta e operações de modificação. As operações de consulta são realizadas pelos plug-ins diretamente às entidades do modelo central. Por outro lado, as operações de modificação só são realizadas por meio de requisição dos plug-ins ao micro-núcleo. Esse mecanismo facilita o controle de acesso concorrente ao modelo central, visto que somente o micro-núcleo tem permissão de modificar o modelo. O micro-núcleo antes de realizar a alteração é responsável por **travar** a

parte do documento que irá ser modificada, realizar a devida alteração e **destravar**, garantindo a consistência do modelo para chamadas de consultas realizadas por plug-ins.

O sincronismo entre as visões nas ferramentas atuais (*LimSee2 e Composer I*) é feito em todo o modelo central. Para cada mudança mínima no documento, o modelo interno da visão é resincronizado com o modelo central em sua totalidade, visto que só é notificado que uma alteração ocorreu no documento e não qual foi essa alteração. Essa antiga abordagem não é escalável, além desse processo de tradução ser um ponto de falha crítico. Uma das principais vantagens desse mecanismo de comunicação proposto é a possibilidade de sincronização incremental entre os diferentes plug-ins. Mudanças no modelo central são notificadas assim que elas ocorrem, através de uma API bem definida que será detalhada no Capítulo 5. Cada plug-in pode atualizar sua interface gráfica de maneira incremental, eliminando os atrasos no retorno visual para o autor, provendo uma maior qualidade de experiência.

Uma particularidade desse mecanismo de comunicação é a possibilidade do plug-in tratar somente um subconjunto das entidades presentes na linguagem alvo, ou seja, o plug-in só é notificado de mudanças referentes a entidades que são do seu interesse. Nesse caso o micro-núcleo provê um mecanismo de filtragem de informações, diminuindo ainda mais o número de mensagens trocadas entre os plug-ins e o micro-núcleo.

Uma última funcionalidade incorporada ao micro-núcleo é o suporte ao controle de transações. Abrindo uma transação com o micro-núcleo (*begin transaction*), o plug-in garante que as alterações enviadas ao micro-núcleo a partir daquele momento serão executadas de maneira atômica, ou seja, notificações de outros plug-ins não irão interferir. As alterações só serão de fato serem executadas sobre o modelo central quando o plug-in concluir a transação (*end transaction*).

Para dar suporte a esse novo mecanismo de comunicação e sincronização é necessária uma arquitetura centrada nos RNFs discutidos amplamente no Capítulo 3. A Figura 12 apresenta a arquitetura que está sendo proposta por esse trabalho. A arquitetura é composta basicamente de 3 camadas: camada de portabilidade, camada de controle (onde se encontra o micro-núcleo) e a camada de plug-ins.

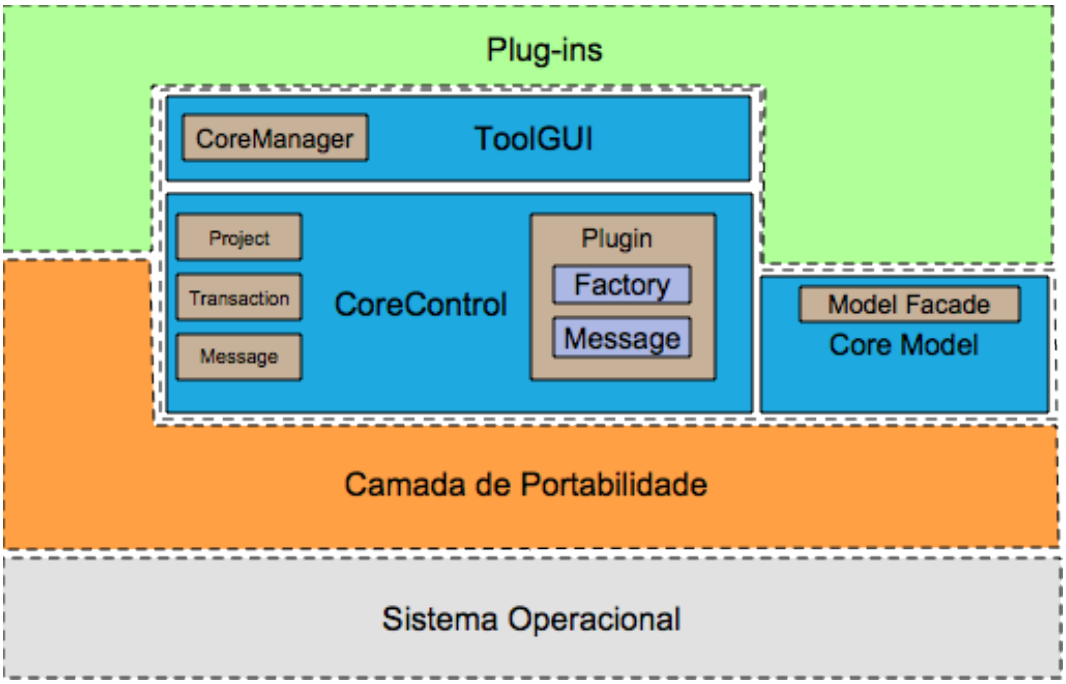


Figura 12 - Arquitetura



**Portabilidade** é um requisito não funcional suportado por essa arquitetura, através da camada de portabilidade, no caso específico do *Composer II*, essa camada de portabilidade é construída sobre o arcabouço QT (Nokia Inc.) , mas ela é flexível o suficiente para aceitar outros arcabouços de interface gráfica como GTK, wxWindow, Fox Toolkit, etc. QT suporta uma grande variedade de sistemas operacionais, incluindo dispositivos portáteis (Symbian, Maemo, WebOs, Windows Mobile e Android). Adicionalmente, QT provê API para construção de interface gráfica, manipulação de documentos XML e reprodutores de mídias (áudio, vídeo e imagens). No Capítulo 5 é feito um detalhamento sobre as vantagens de QT sobre os demais arcabouços gráficos existentes.

### *4.1.1. 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. O modelo utilizado nas máquinas de apresentação precisam estar desde do início consistente e corretos sintaticamente.

O modelo central em uma ferramenta de autoria deve ser flexível e aceitar inconsistências em determinados momentos durante a autoria. O autor de uma

aplicação hipermídia geralmente não segue uma linearidade de construção, a ferramenta deve ser flexível e permitir que autor seja livre para especificar ou modificar qualquer parte da aplicação em qualquer momento. Por exemplo, um autor pode começar definindo os objetos de mídia que irão compor a aplicação sem antes ter definido onde essas mídias deverão ser apresentadas espacialmente. A ferramenta ao não permitir essas inconsistências traz problemas de comprometimento precoce, previstos em (G. & M., 1996). Este modelo central foi construído com o intuito de prover tal flexibilidade.

O micro-núcleo deve funcionar de maneira independente do modelo de dados que é utilizado no *CoreModel*. A comunicação entre o *CoreModel* e *CoreControl* é realizada através de uma fachada que provê uma API simples para o *CoreControl* manipular as entidades presentes no *CoreModel*. *ModelFacade* é essa fachada, e tem como objetivo receber as requisições de modificações do *CoreControl* e executar os procedimentos necessários para que essas modificações sejam realizadas sobre o *CoreModel* corretamente. Esse mecanismo permite que a ferramenta baseada nessa arquitetura seja usada para diferentes modelos de representação.

### 4.1.2. *CoreControl*

O *CoreControl* é o micro-núcleo em si. É nesse módulo que ficam as funções essenciais mencionadas no Capítulo amterior. 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 Document*.

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.

A notificação dos plug-ins, seja em caso de sucesso ou erro, é realizada através de chamadas à API de comunicação que será definida no Capítulo 5. Para garantir uma comunicação rápida entre o micro-núcleo e os plug-ins, o módulo *Message* deve ser implementado com chamadas não bloqueantes, ou seja, deve

somente emitir um sinal para todos os plug-ins e continuar sua execução sem esperar nenhum retorno. Detalhes de como deve ser feita a implementação desse módulo serão discutidos em maior profundidade no Capítulo 5.

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. A partir do momento em que o plug-in iniciar uma transação (*begin transaction*), o *Transaction* irá armazenar suas solicitações em uma fila. Somente quando o plug-in notificar o final da transação (*end transaction*) o *Transaction* irá realizar as alterações solicitadas. Essas alterações são executadas de maneira atômica, ou seja, na ordem em que foram recebidas e sem solicitações de outros plug-ins intercaladas.

Além disso, o *Transaction* gera um número que identifica a transação. Esse número é utilizado pelo plug-in para identificar quais mensagens fazem parte de uma transação. Isso possibilita a abertura de mais de uma transação simultânea com entre o plug-in e o micro-núcleo. Adicionalmente, esse número pode ser utilizado pelos plug-ins para requisitar o *rollback* de uma transação já realizada.

O *Plug-in* é o módulo responsável pelo carregamento dos *plug-ins* externos. Esse é um módulo dinâmico, os plug-ins podem ser carregados e descarregados em tempo de execução da ferramenta. Os detalhes da implementação desse módulo são discutidos no Capítulo 5.

Esse módulo executa a função de filtro para cada um dos plug-ins. Antes de conectar o micro-núcleo com o plug-in carregado, esse módulo faz uma chamada para a função de filtro, tendo como retorno os elementos o plug-in quer ser notificado sobre cujas alterações. Esse módulo ainda define as interfaces que devem ser implementadas para a extensão do micro-núcleo.

*ToolGUI* é a parte gráfica da ferramenta. O importante sobre esse módulo está no *CoreManager*. Assim como existe uma API para comunicação entre os plug-ins e o micro-núcleo, é preciso estabelecer uma API para que a interface gráfica da ferramenta possa se comunicar com o micro-núcleo. Essa API está definida no *CoreManager*. Essa API ainda define como o micro-núcleo repassa informações gráficas entre os plug-ins e a *ToolGUI*. Os detalhes da API e sua implementação serão discutidos no Capítulo 6.